Thinking Parallel
— An Introduction to Parallel Computing —

Hinnerk Stüben

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Universität Bremen

13–18 September 2025

Contents

Introduction

Motivation

- Why program in a language like Fortran, C or C++? → *speed*
  (instead of using higher math packages)

- Why compute in parallel? → *speed*

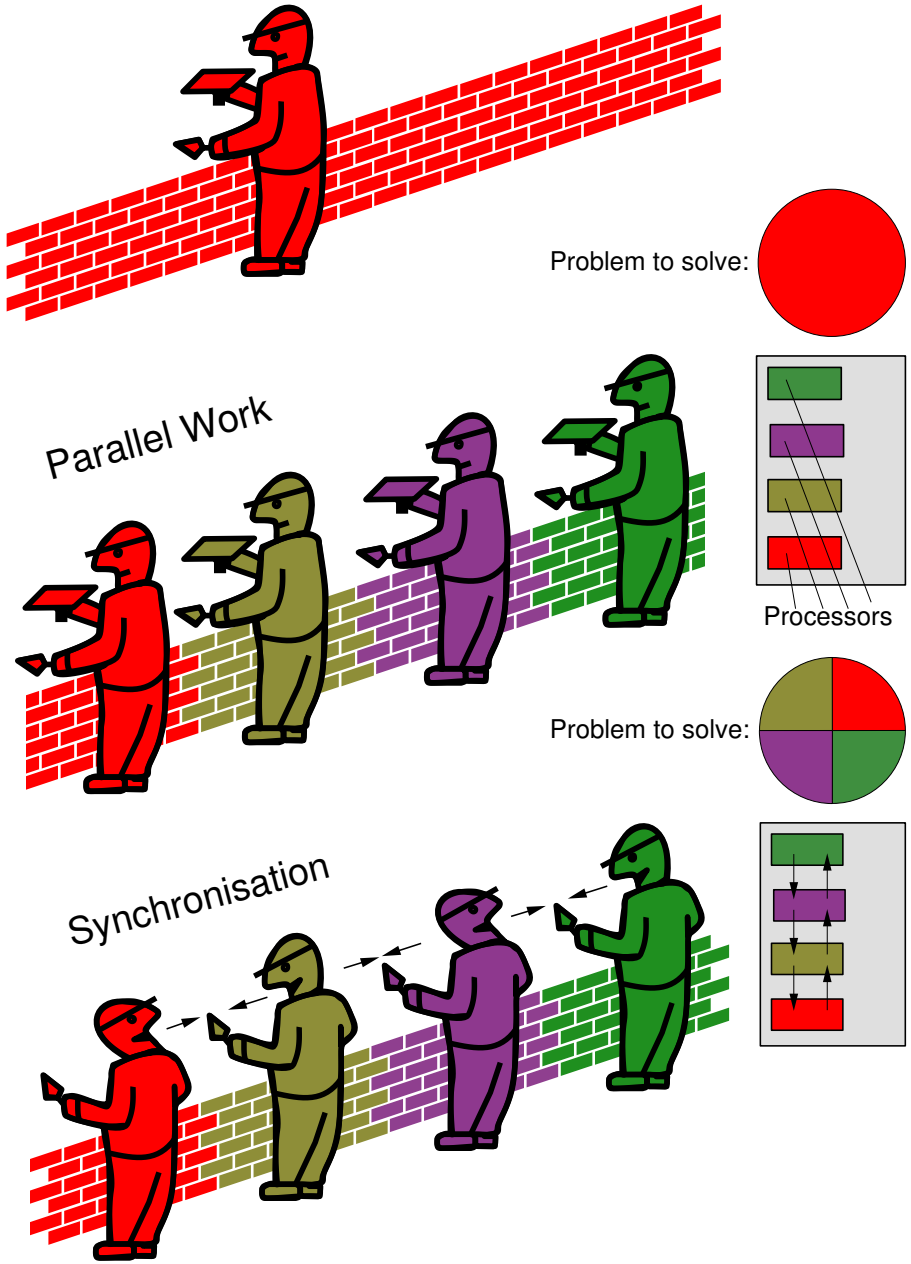
  – One can gain up to orders of magnitudes.

- Since $\approx$ 2005 the need for parallelisation has become more pronounced:

  – Up to then the speed of a single processing unit (CPU) doubled approximately every 18 months.
  – This development came to an end.
  – Now the number of processing units per chip (cores) is growing.
  → *multi-core technology*

→ To take advantage of the continuously growing processing power per chip, applications have to be parallelised.

Basic idea of parallel computing



[picture by W. Baumann]

Parallel computer architectures

Hardware components of parallel computers

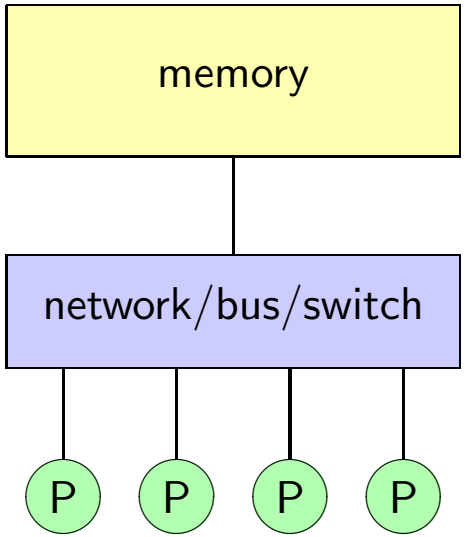
- processors
- communication network
  - topology
  - performance (latency, bandwidth)
- memory architecture
  - shared memory
  - distributed memory
  - hybrid systems
- additional components
  - barrier network
  - hardware for I/O

Glossary

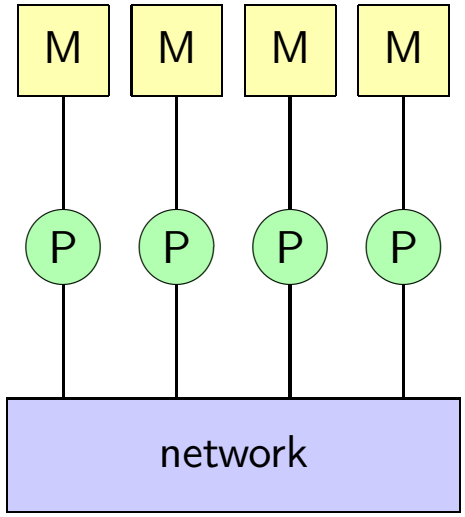
HPC	High Performance Computing
SIMD	Single Instruction, Multiple Data
MIMD	Multiple Instructions, Multiple Data
SPMD	Single Program, Multiple Data
MPMD	Multiple Programs, Multiple Data
MPP	Massively Parallel Processing
SMP	Symmetric Multi-Processor
NUMA	Non-Uniform Memory Access
ccNUMA	Cache Coherent Non-Uniform Memory Access

Overview of parallel computer architectures

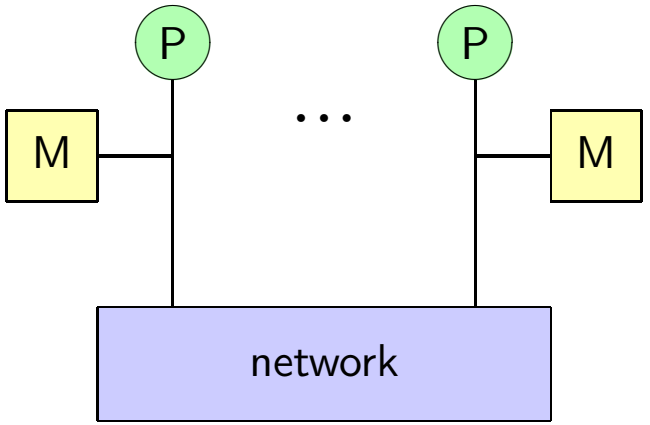
SMP
(shared memory)



MPP
(distributed memory)

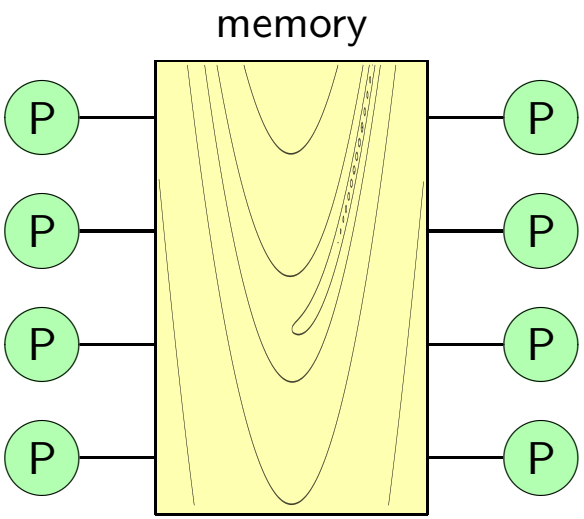


NUMA
(distributed memory, global address space)

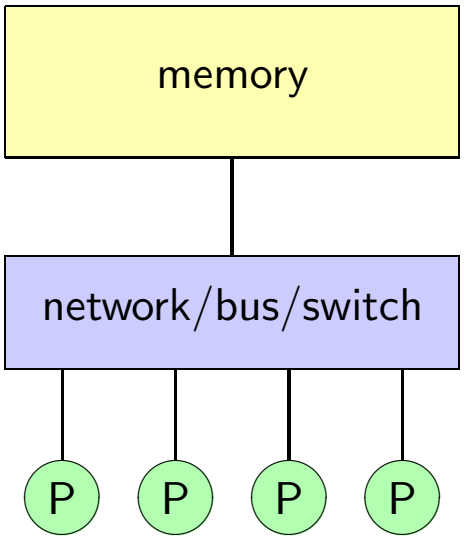


SMP – parallel computer with shared memory

programmer's view

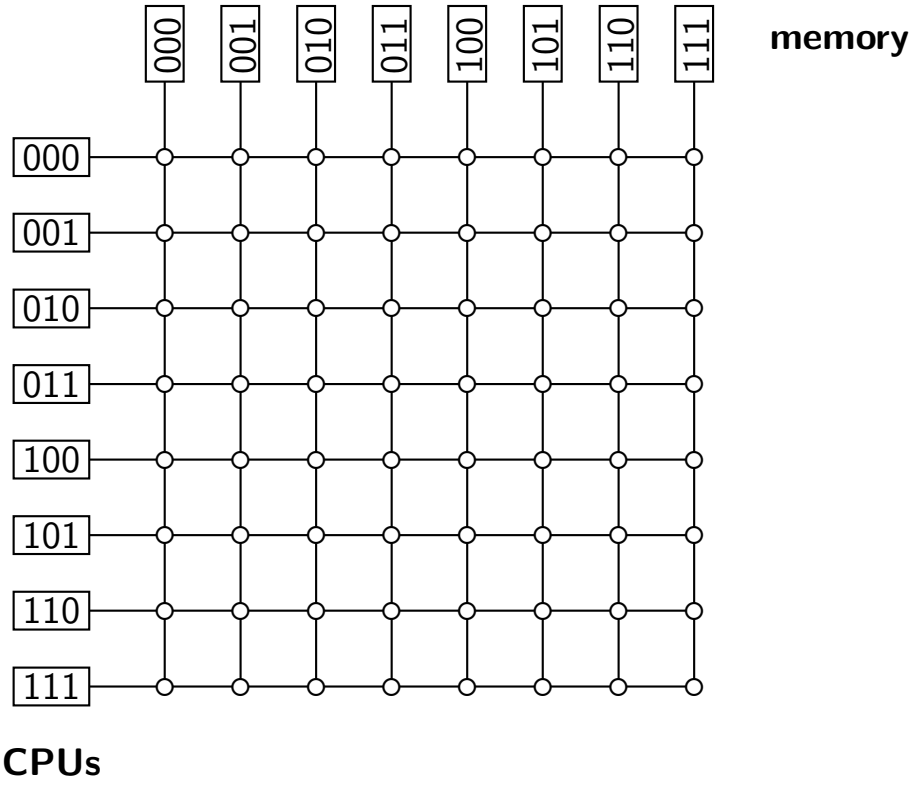


hardware architecture



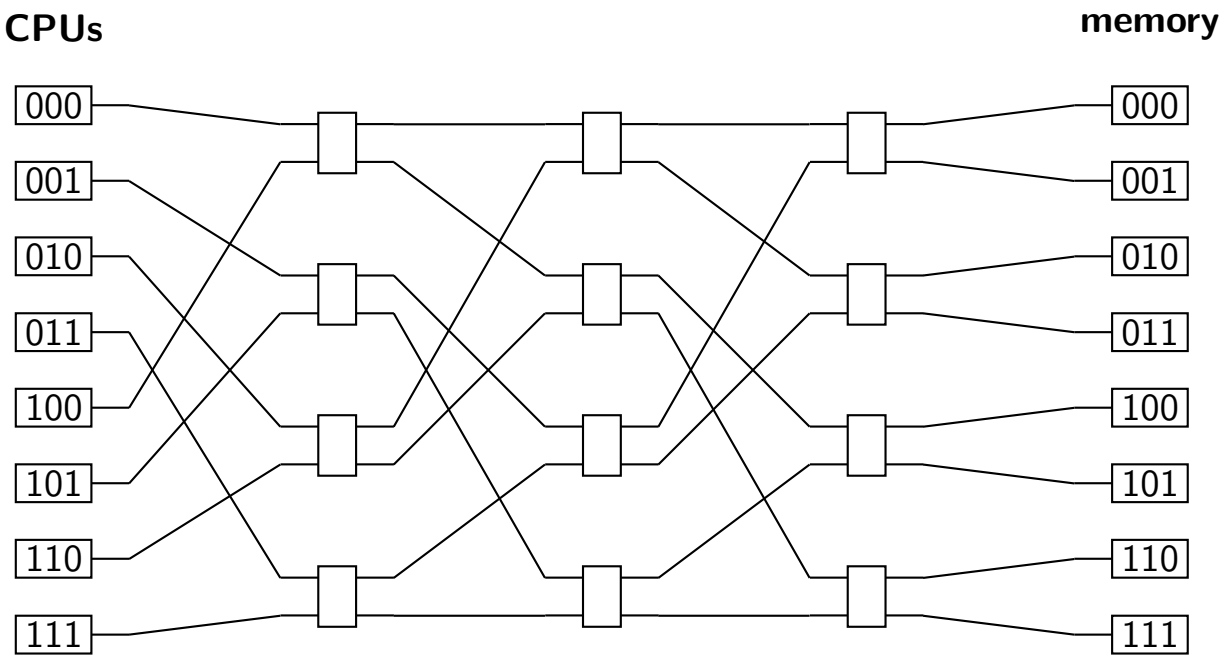
- every process(or) can access the whole memory
- memory access is uniform (in principle) $\rightarrow$ *symmetric multi-processor (SMP)*
- example: multi-core processor ('SMP on a chip')
- standard parallel programming method: OpenMP

SMP – networks (I)



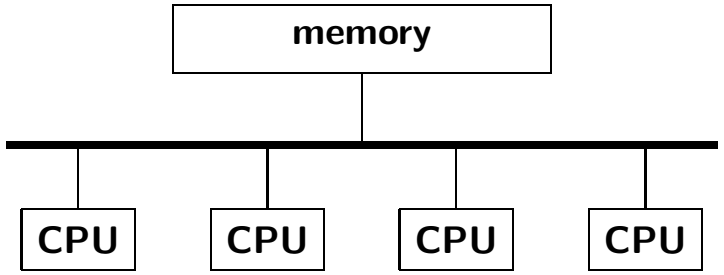
crossbar switch

SMP – networks (II)



multistage switching network

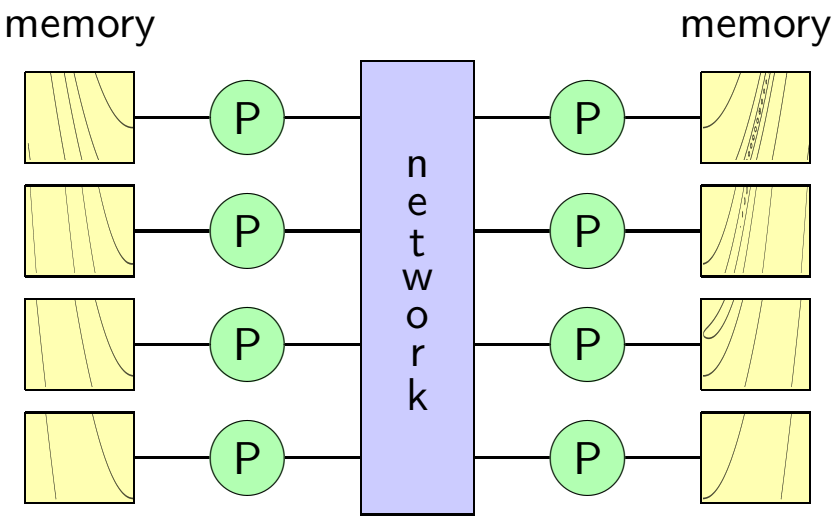
SMP – networks (III)



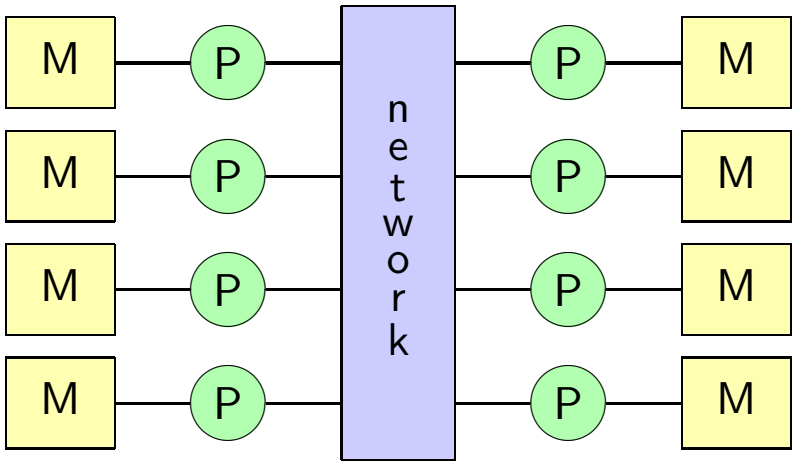
bus

MPP – parallel computer with distributed memory

programmer's view

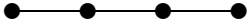


hardware architecture

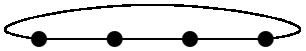


- MPP: *massively parallel computer*
- every process(or) can access its local memory directly
- for remote memory access the remote process(or) has to be involved
- example: PC cluster
- standard parallel programming method: MPI

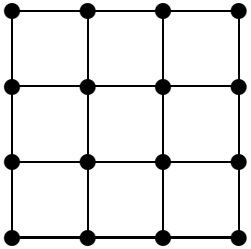
MPP – networks (I)



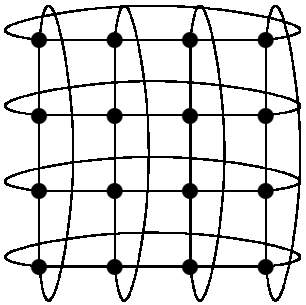
1-dim. mesh



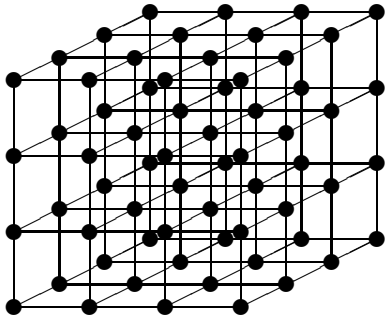
1-dim. torus / ring



2-dim. mesh



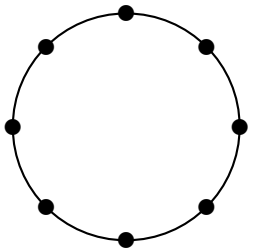
2-dim. torus



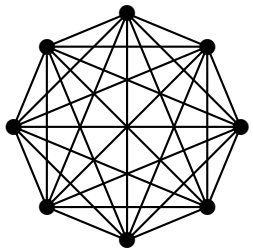
3-dim. mesh

. . .

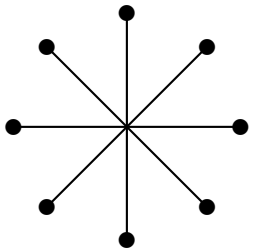
MPP – networks (II)



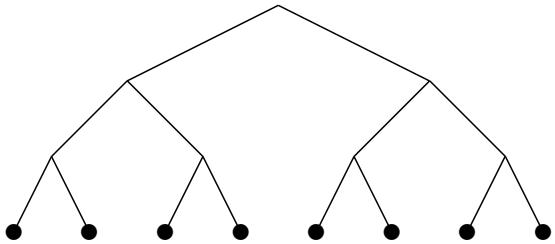
ring



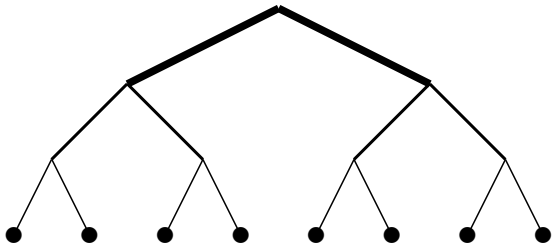
full connectivity



star

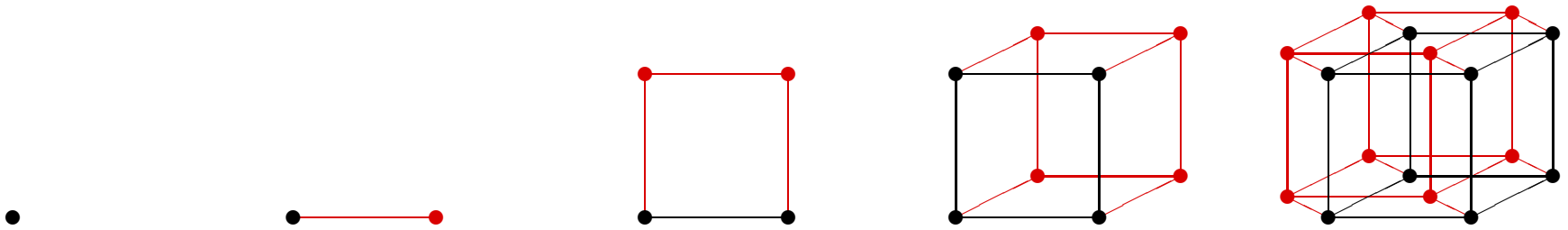


tree



fat tree

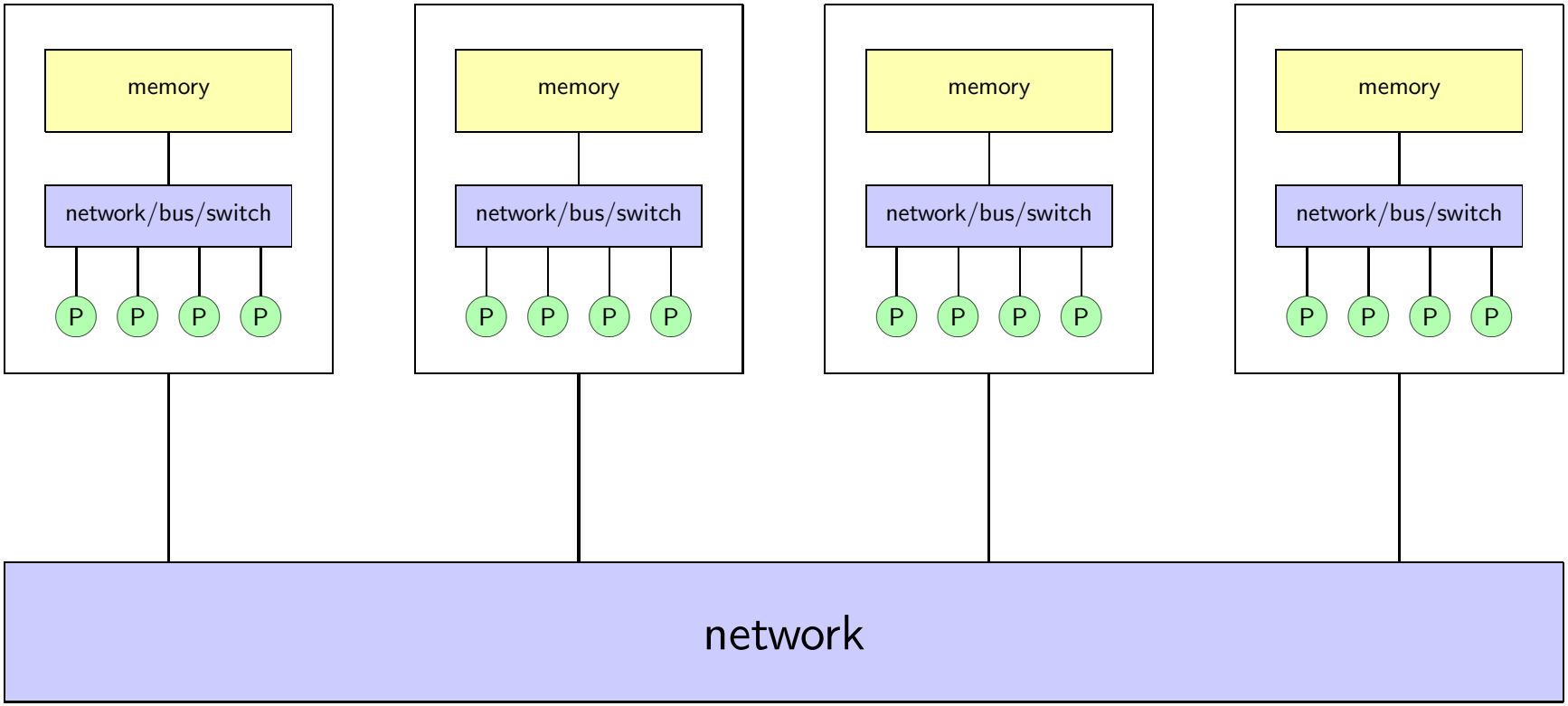
MPP – networks (III)



0-dim. 1-dim. 2-dim. 3-dim. 4-dim.

n-dim. hypercube

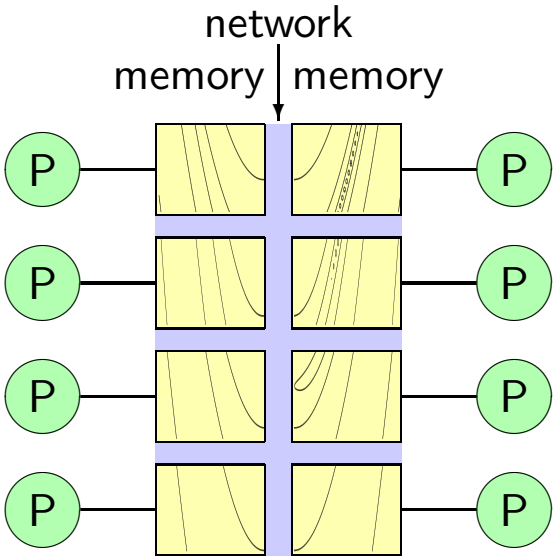
Hybrid architecture



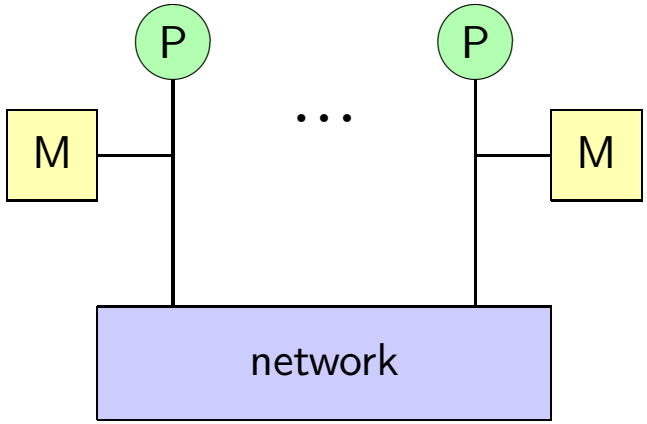
- here: cluster of SMP nodes
- in the era of multi-core processors hybrid systems are standard

NUMA computer – distributed memory, global address space

programmer's view



hardware architecture



- every process(or) can access the whole memory
- local memory access is faster than remote memory access
  $\rightarrow$ *non uniform memory access (NUMA)*
- example: SGI Altix $\rightarrow$ *ccNUMA (cache coherent NUMA)*
- standard parallel programming method: OpenMP or MPI

Remarks on the acronyms CPU, SMP and MPP

- a multi-core processor has no single *central processing unit*
  - today, CPU means processor chip (also: socket; a single hardware item)
- the use of acronyms SMP and MPP is often not quite correct
  - today, compute servers with more than 1 CPU are not symmetric/SMP but rather NUMA
  - we use MPP also for clusters (which are not massively parallel)
  - examples for real MPPs: IBM BlueGene series, Cray XC series

Flynn's taxonomy

instruction stream	data stream: single data	data stream: multiple data
single instr.	**SISD** uniprocessor von Neumann architecture	**SIMD** (following slides)
multiple instr.	**MISD** (research)	**MIMD** mainstream of parallel computing (previous slides)

SIMD / data parallel processing

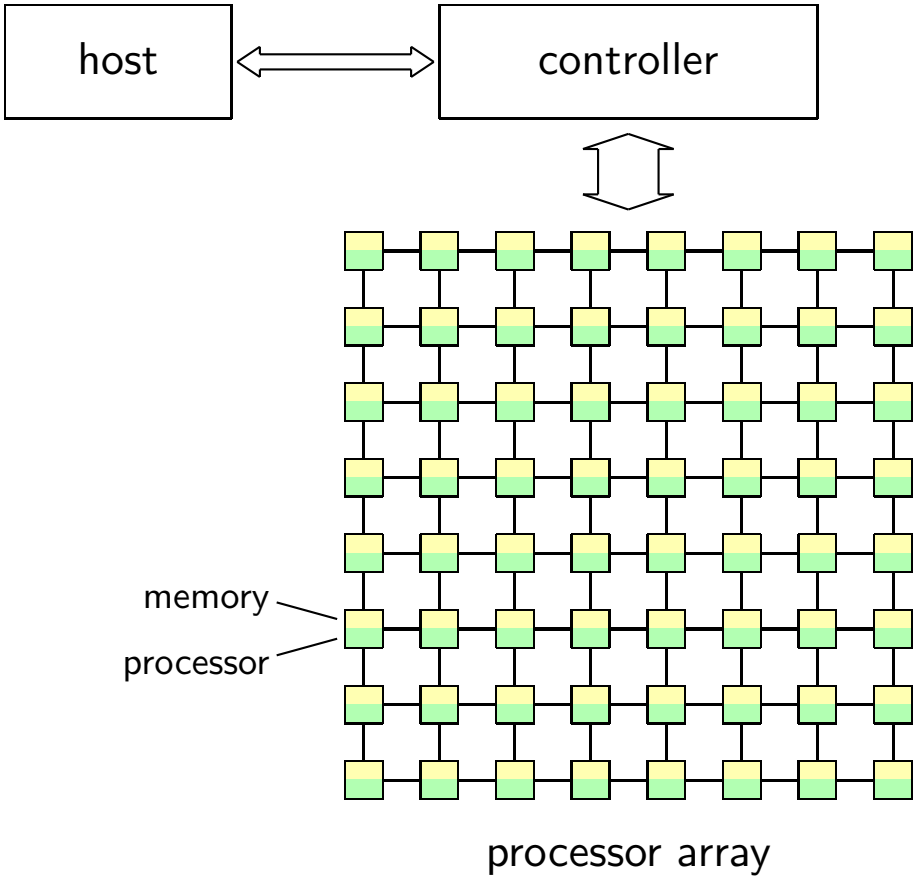
- each operation is executed with multiple data (conceptually) at the same time
- this loop can be executed in data parallel / SIMD mode

```
for i := 1 to N
    a[i] := b[i] + c[i]
```

- this loop *cannot* be executed in data parallel / SIMD mode

```
for i := 1 to N
    if (x[i] < eps)
        y[i] := 1.0 - x[i] * x[i] / 6.0
    else
        y[i] := sin(x[i]) / x[i]
```

SIMD computer



- all processors execute each instruction at the same time
- data can be fetched from or copied to neighbouring processors

SIMD

- past: large SIMD computers with distributed memory – example:
  - Connection Machine CM-2 by Thinking Machines
    up to 65.536 processing elements

- present: SIMD at processor level – examples:
  - Power processors (IBM)
  - x86 processors: SSE, SSE2, AVX, AVX2 . . . (Intel, AMD)
  - Cell processor (IBM/Toshiba/Sony)
  - graphical processing units (ATI, NVIDIA)

Two more acronyms – execution models on MIMD machines

- **SPMD** – Single Program, Multiple Data
  - all processes execute the same program
  - example:

```
$ mpirun -np 3 mpi-a
mpi-a: size = 3, rank = 1
mpi-a: size = 3, rank = 0
mpi-a: size = 3, rank = 2
```

- **MPMD** – Multiple Programs, Multiple Data
  - at least two different programs are being executed
  - example:

```
$ mpirun -np 3 mpi-a : -np 2 mpi-b
mpi-a: size = 5, rank = 1
mpi-a: size = 5, rank = 2
mpi-a: size = 5, rank = 0
mpi-b: size = 5, rank = 3
mpi-b: size = 5, rank = 4
```

Parallel programming models

Programming shared memory computers

- typical approach: *data parallel programming / parallelisation at loop level*

```
directive: parallel do
for i := 1 to 300 do
   a[i] := b[j[i]]
```

- work is decomposed by the compiler (example for 3 CPUs):

```
for i:= 1 to 100 do
   a[i] := b[j[i]]
```

```
for i:=101 to 200 do
   a[i] := b[j[i]]
```

```
for i:=201 to 300 do
   a[i] := b[j[i]]
```

- data decomposition is not necessary (each CPU can access all data)
- auto-parallelisation is possible
- programs can be parallelised loop by loop
- (some) automatic load balancing at loop level

Programming distributed memory computers

- work *and* data have to be decomposed
  (in principle data decomposition is not necessary, if all data fit into a single memory
  $\rightarrow$ problem size is limited; program does not *scale*)

- typical approach: *domain decomposition* and *message passing*

- domain decomposition

  – data is decomposed
  – each process modifies only its data
  – data from remote processes has to be fetched before it can be used

- example: Laplace equation, solve

$$\Delta\varphi = \frac{\partial^2\varphi}{\partial x^2} + \frac{\partial^2\varphi}{\partial y^2} = 0$$

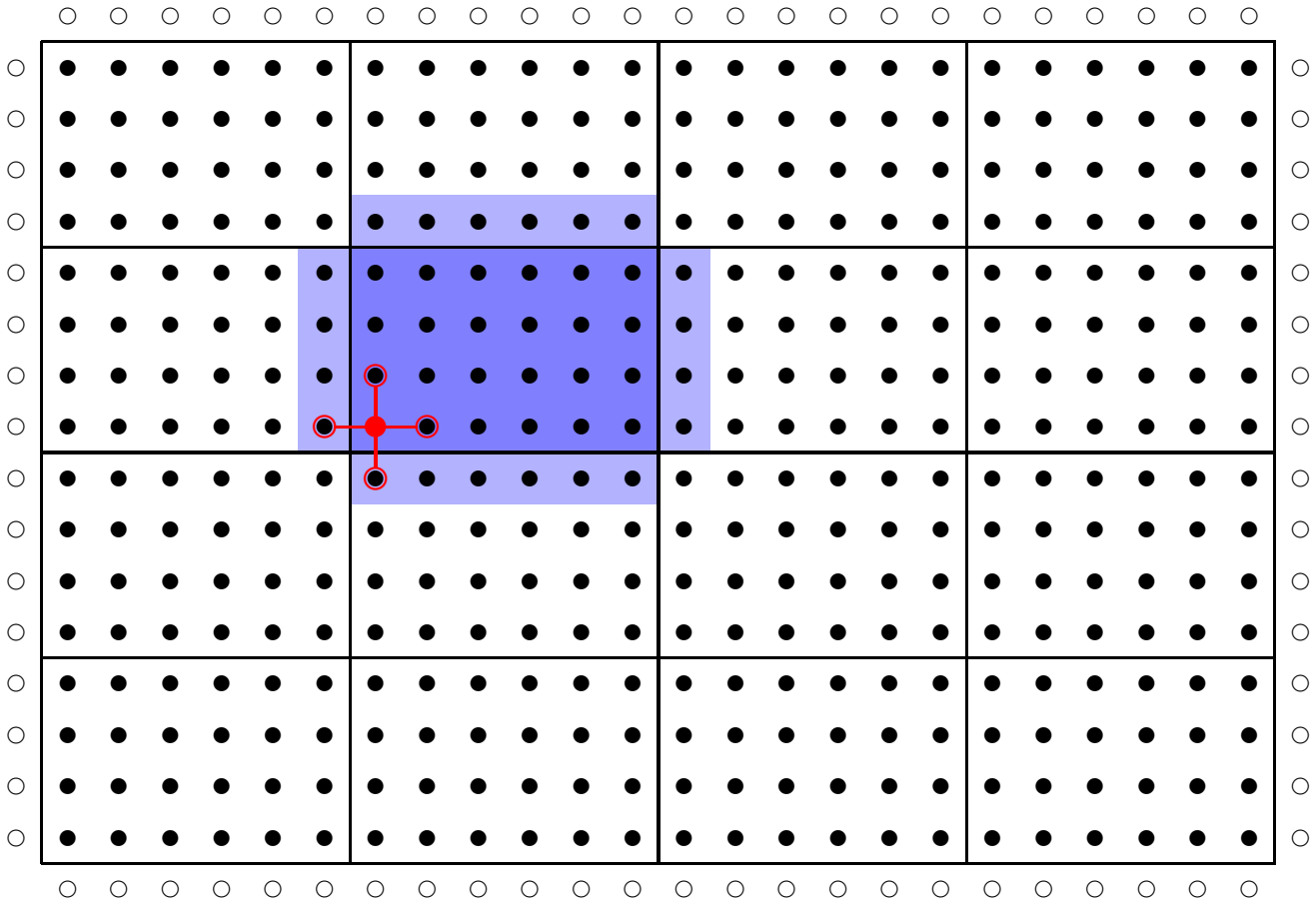
  for given boundary values

Example – Laplace equation

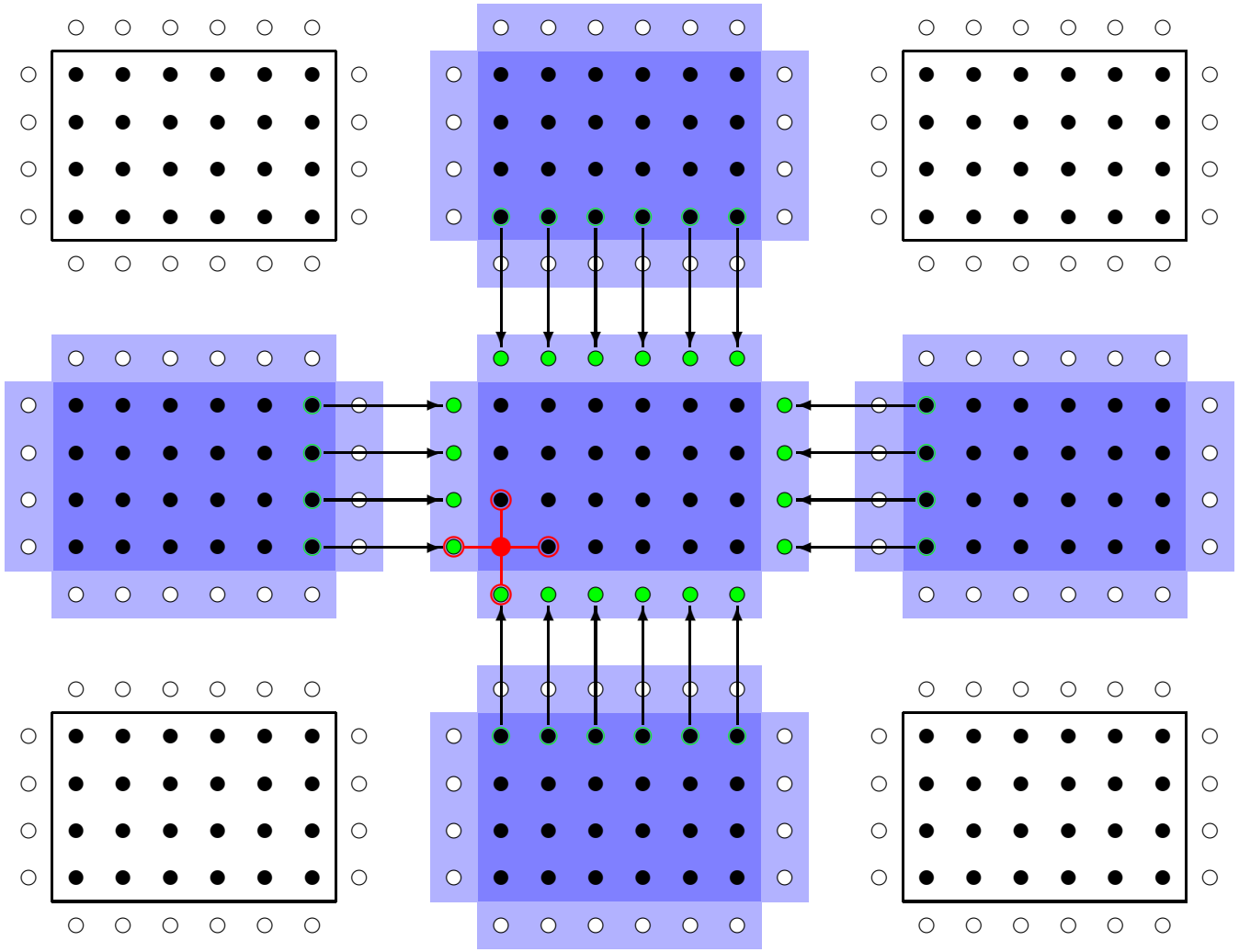
- single processor program

```
repeat

   v_old := v_new

   for x := 1 to N_x do
      for y := 1 to N_y do
         v_new[x, y] = (v_old[x-1, y] + v_old[x+1, y]
                      + v_old[x, y-1] + v_old[x, y+1]) / 4

until (convergence)
```

Example – Laplace equation



Example – Laplace equation



Example – Laplace equation

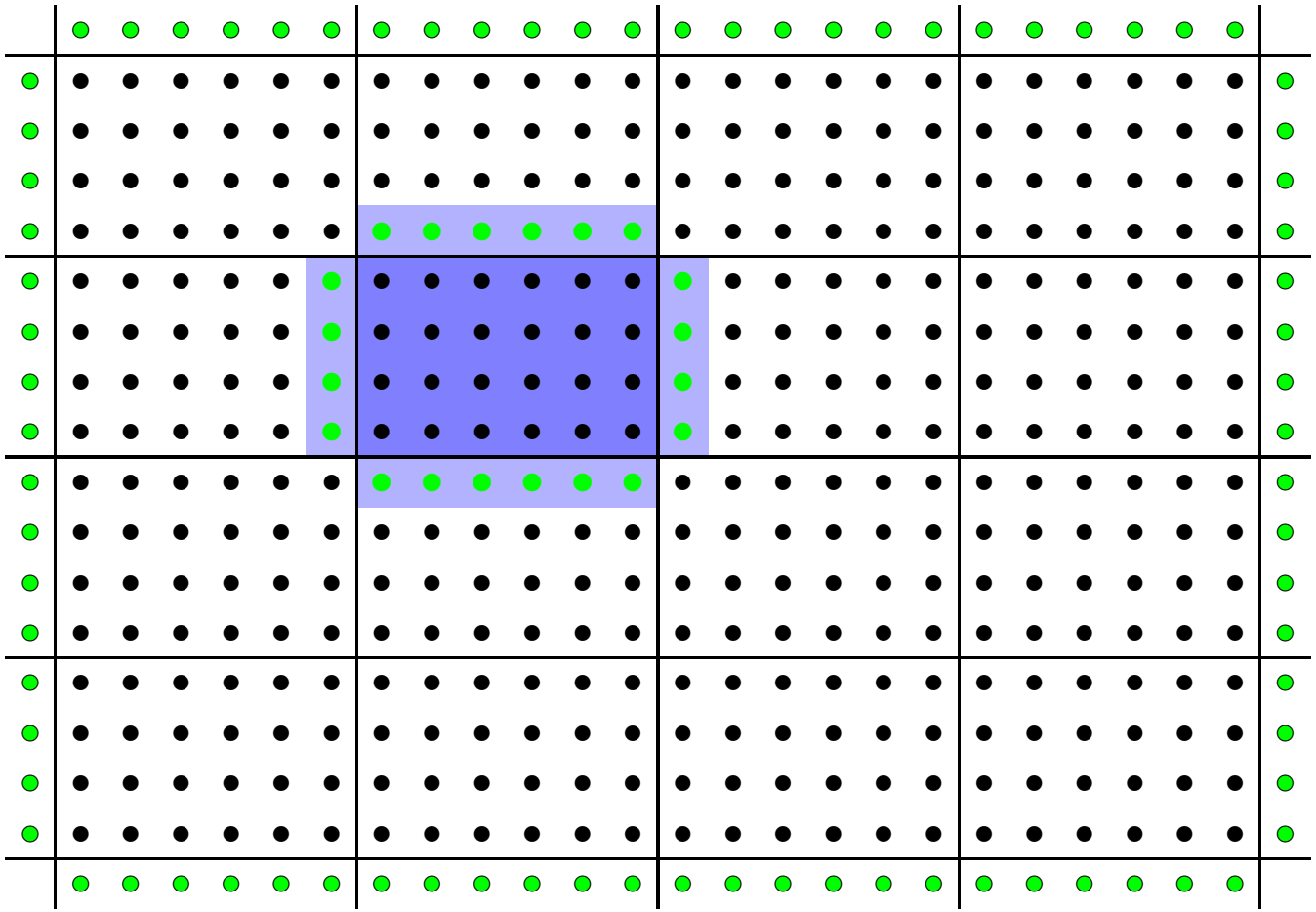
- parallel program

  Changes are indicated in **blue**.

  N_x and N_y are adjusted according to the decomposition.

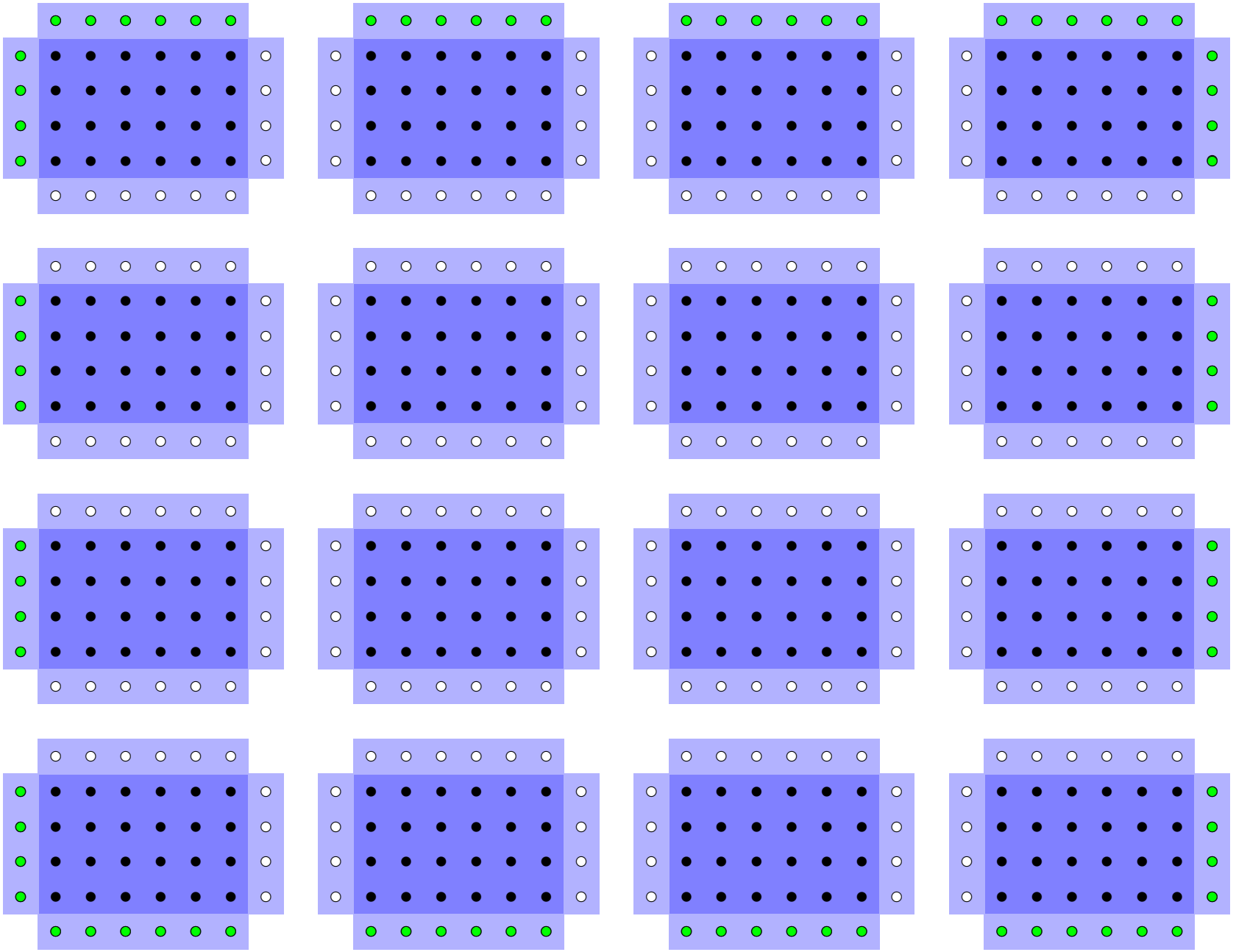
```
repeat

   v_old := v_new

   exchange boundary of v_old

   for x := 1 to N_x do
      for y := 1 to N_y do
         v_new[x,y] = (v_old[x-1,y] + v_old[x+1,y]
                     + v_old[x,y-1] + v_old[x,y+1]) / 4

until (convergence)
```

Inner and outer boundary elements

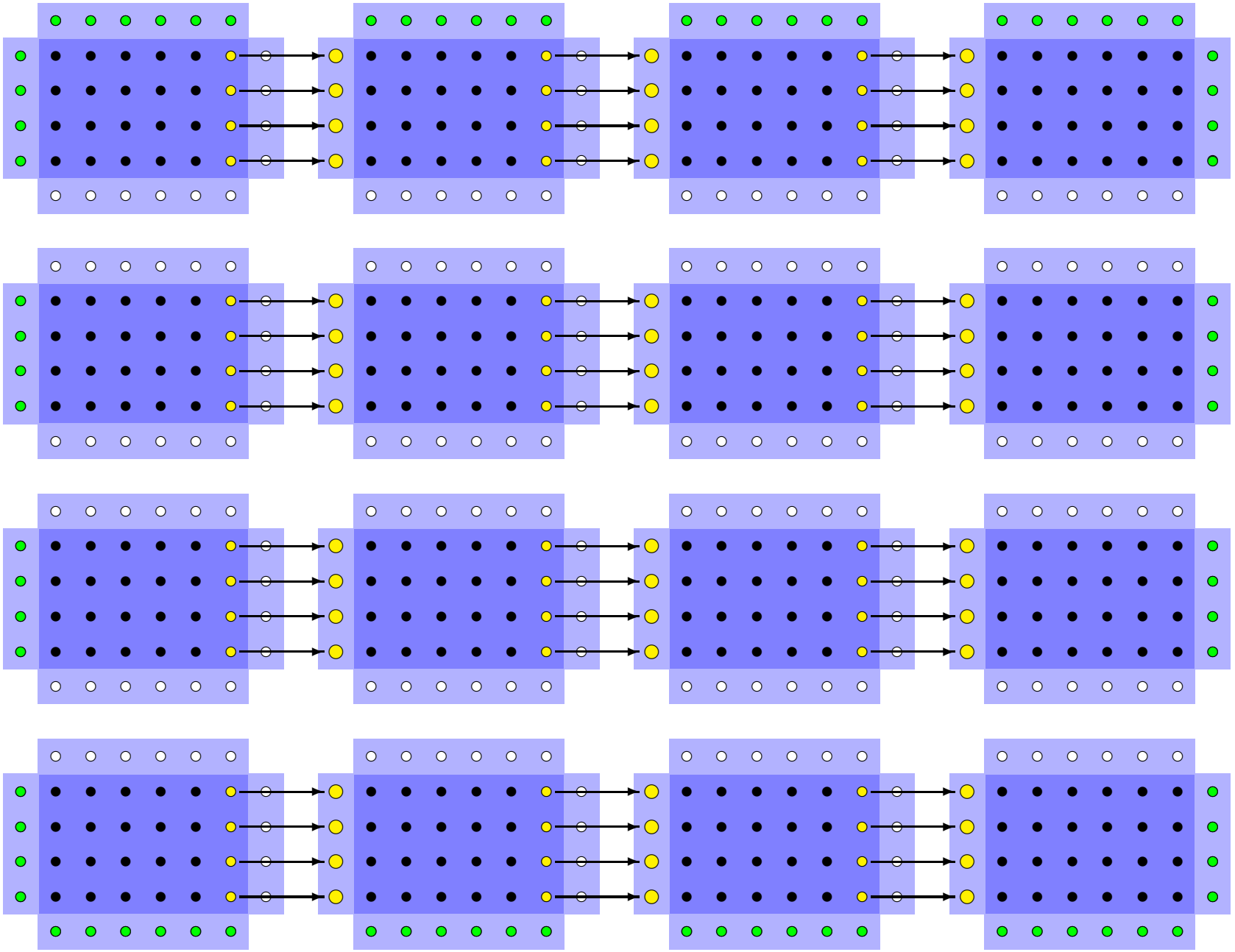


- outer boundary elements are given by the problem
- inner boundary elements (also: halo region, ghost cells) are introduced by the decomposition

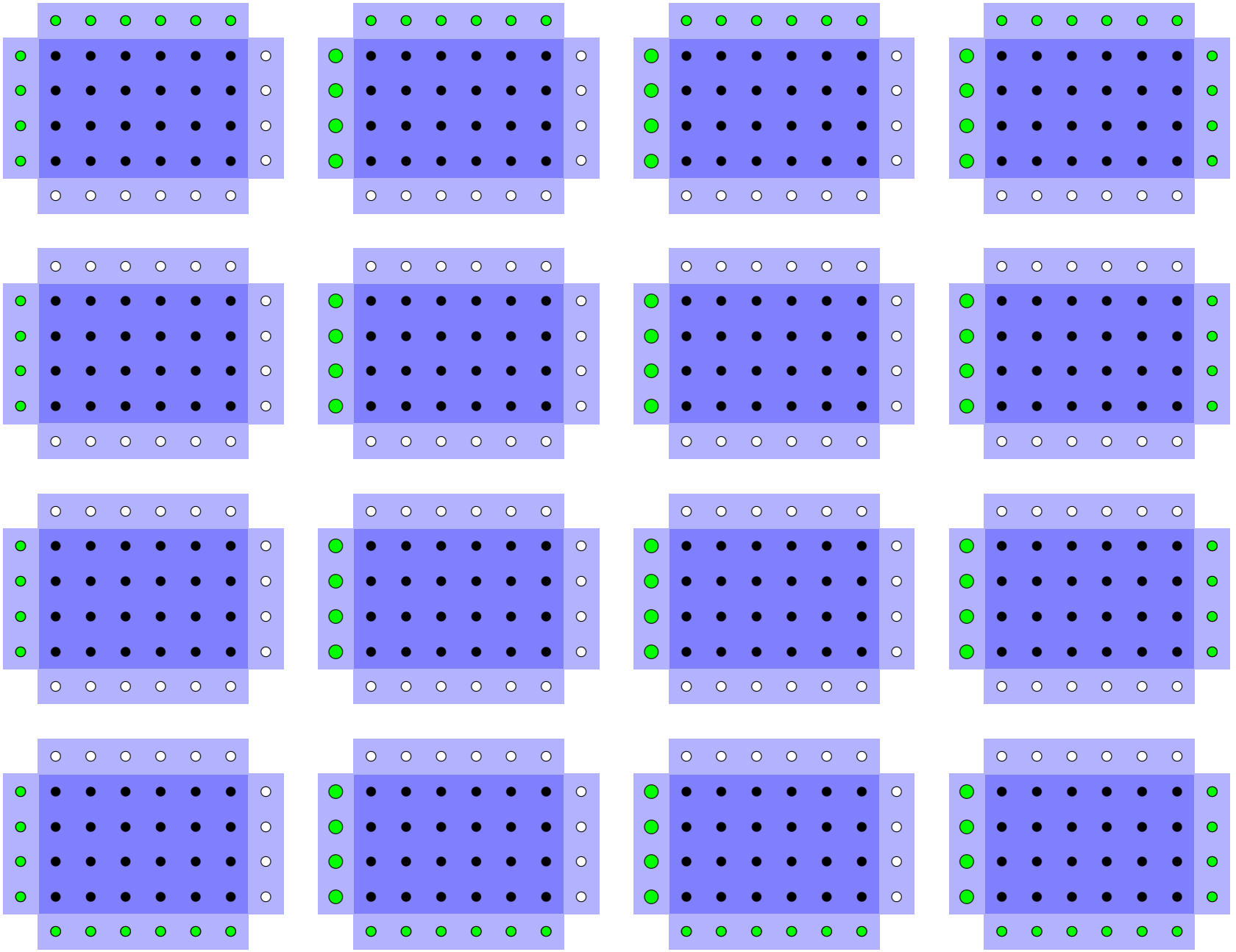
Exchange of boundary elements (with `MPI_Sendrecv`)



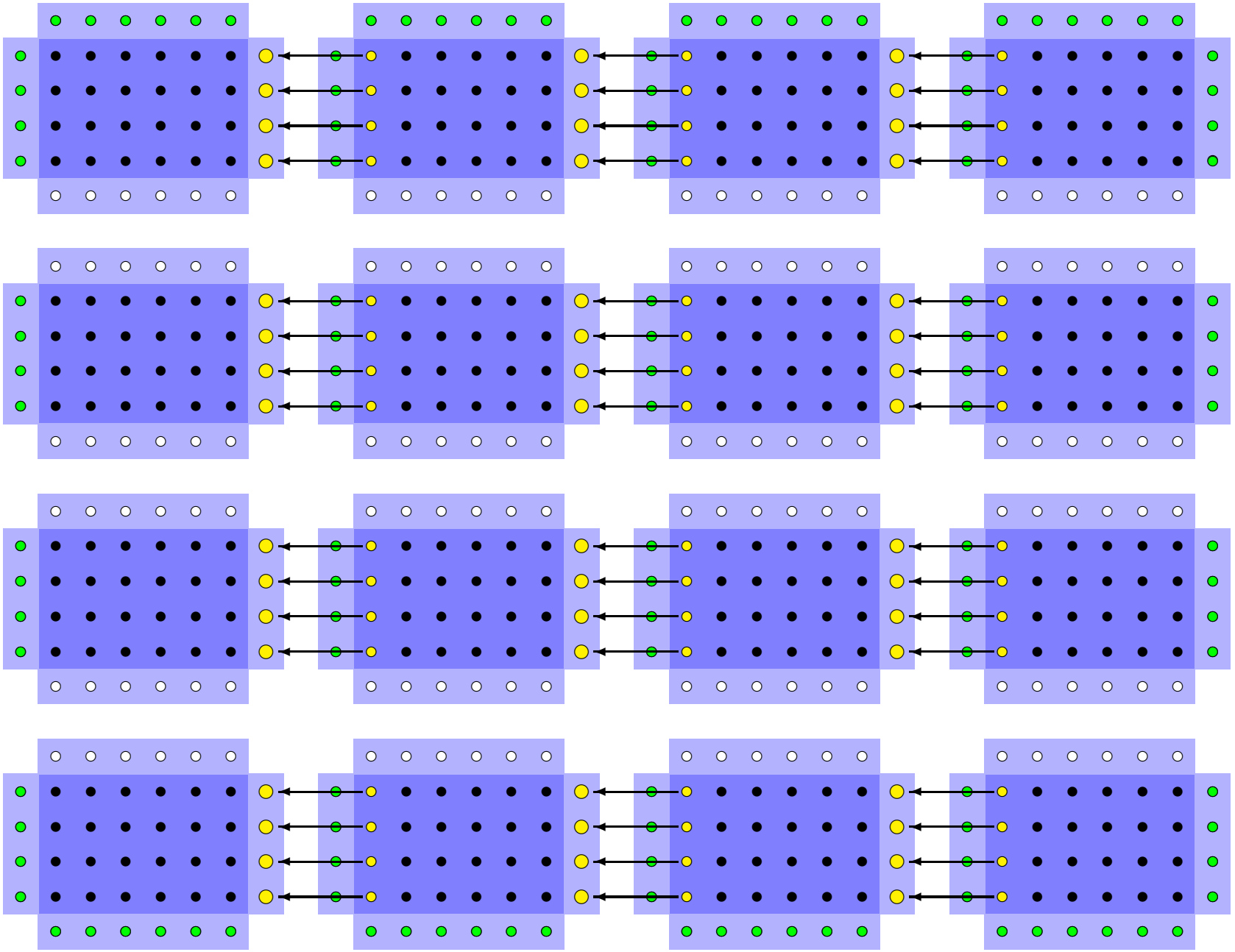
Exchange of boundary elements (with `MPI_Sendrecv`)



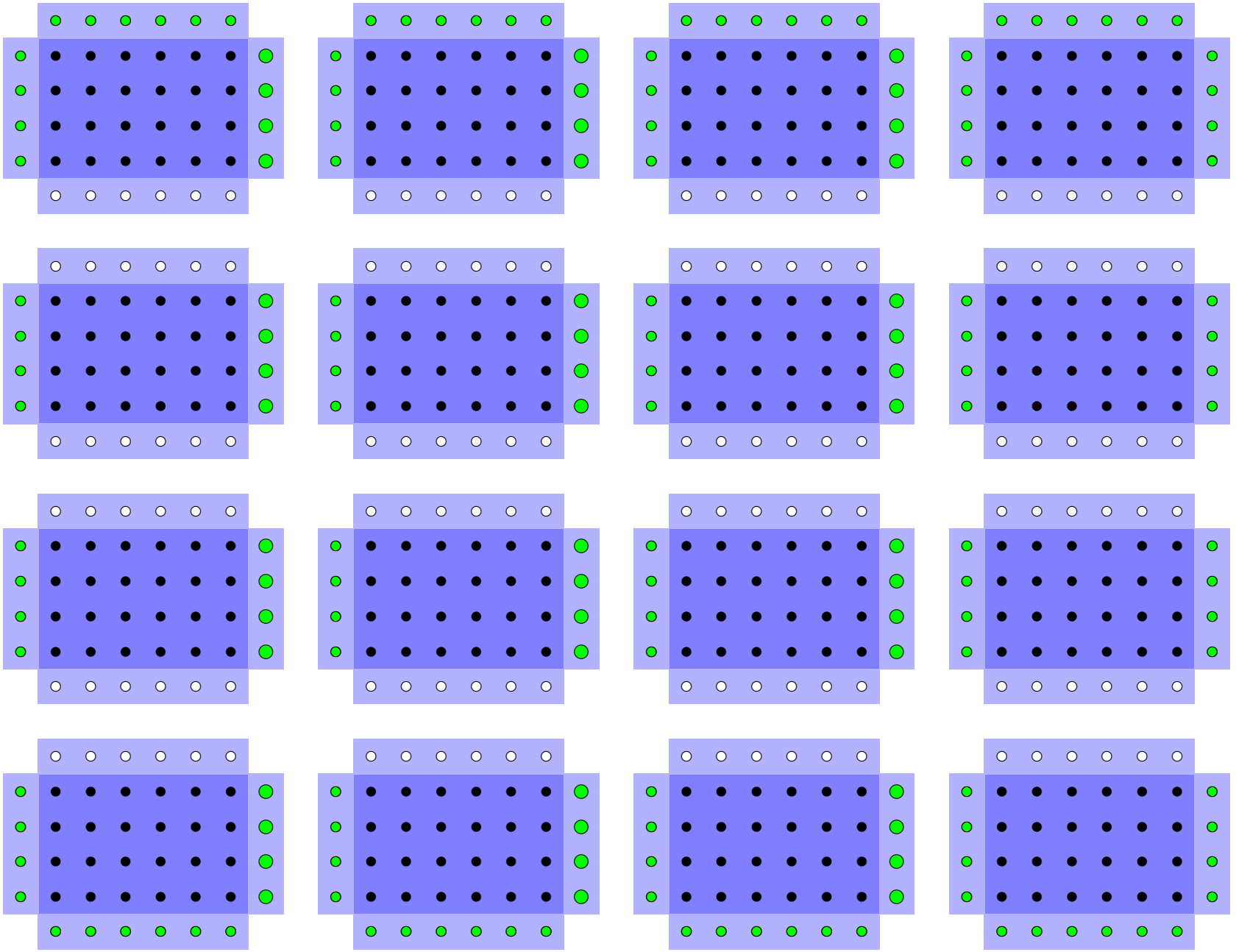
Exchange of boundary elements (with `MPI_Sendrecv`)



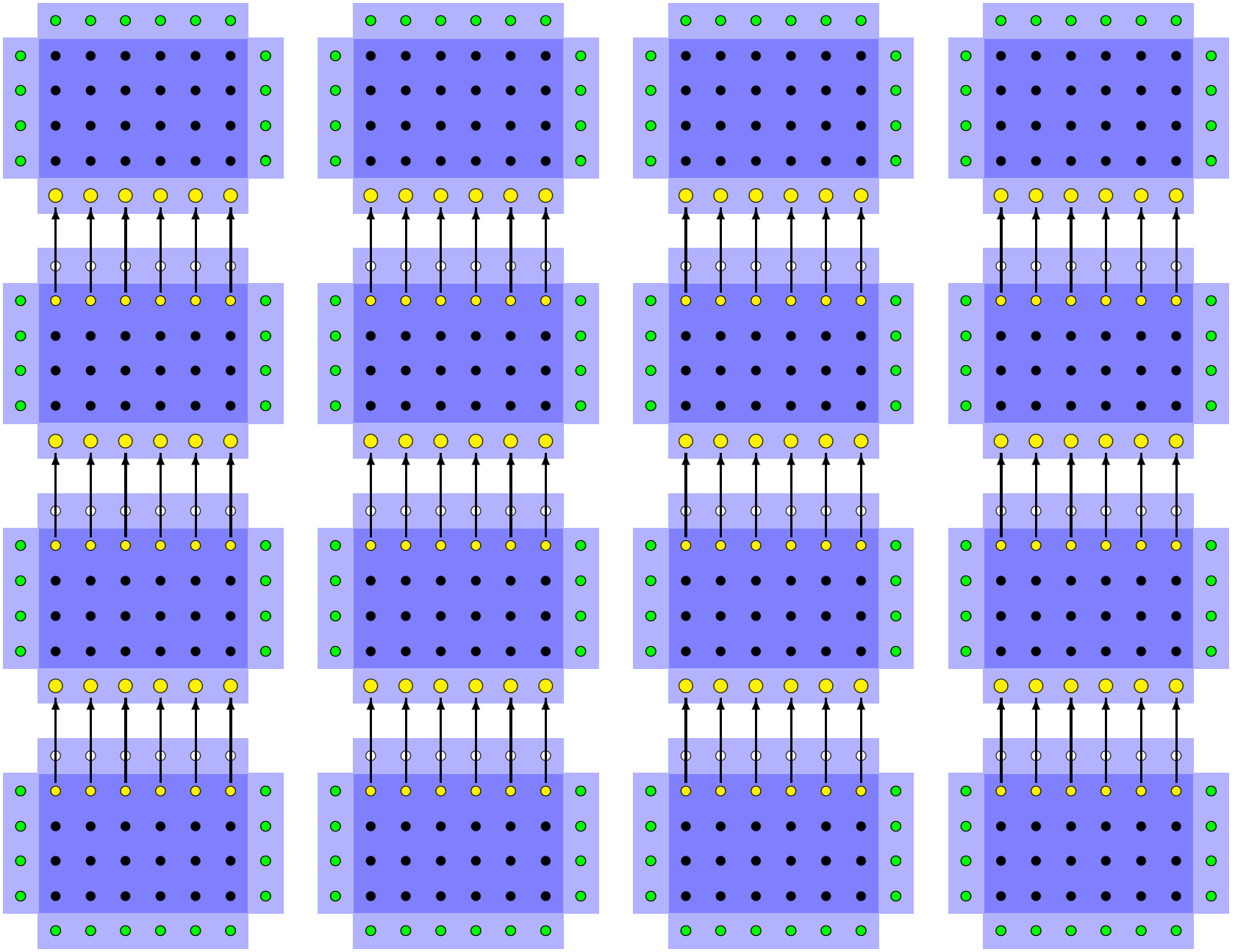
Exchange of boundary elements (with `MPI_Sendrecv`)



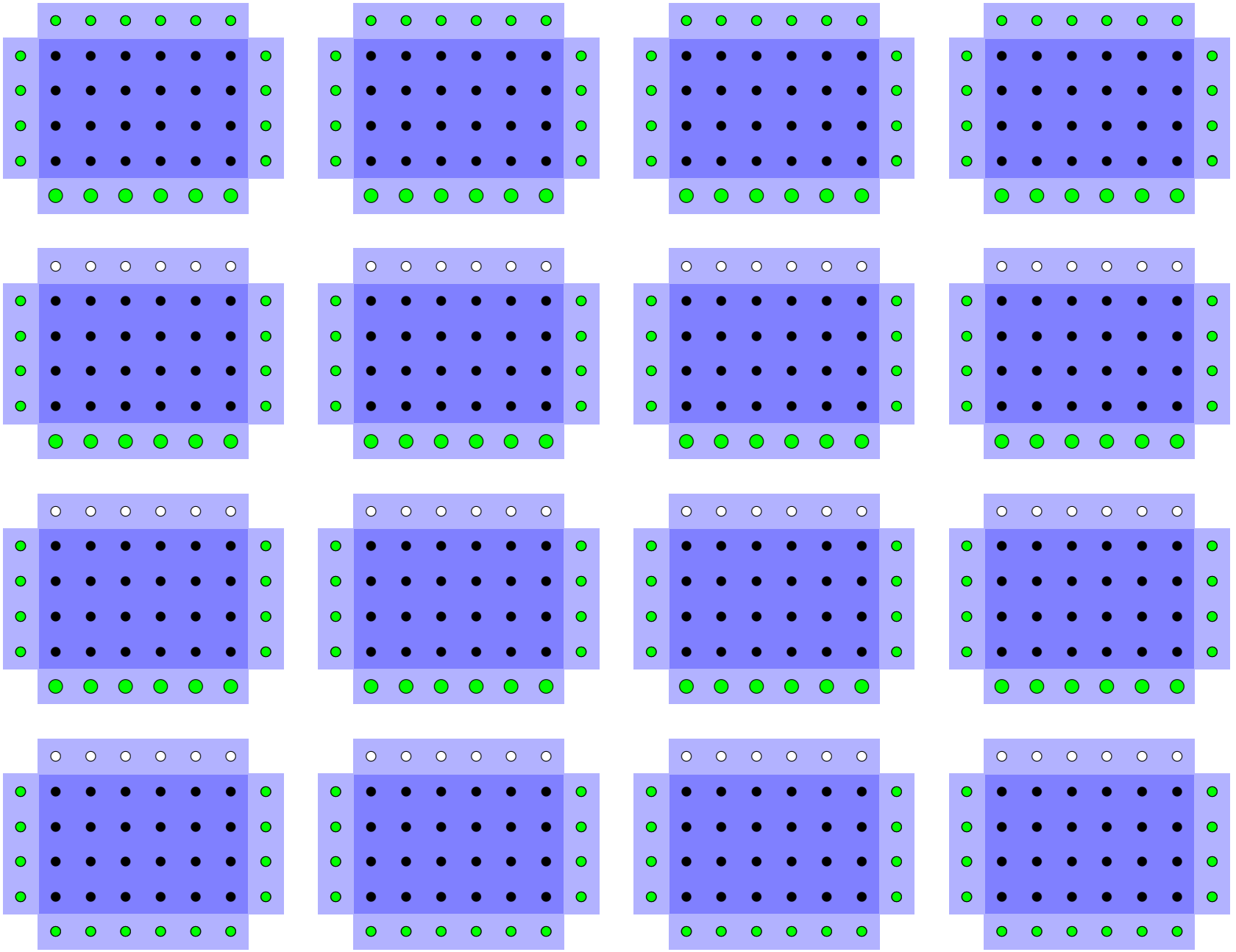
Exchange of boundary elements (with `MPI_Sendrecv`)



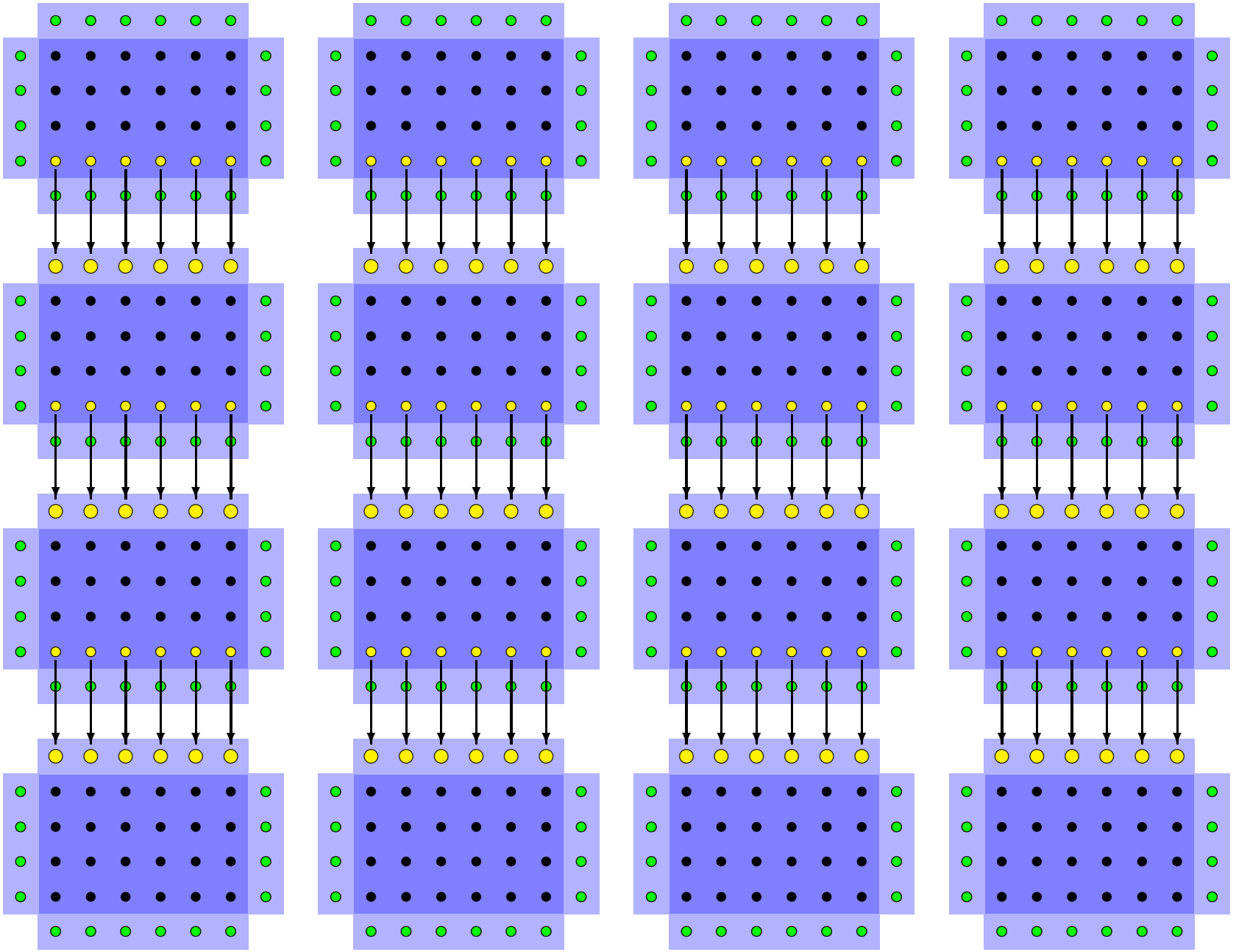
Exchange of boundary elements (with `MPI_Sendrecv`)



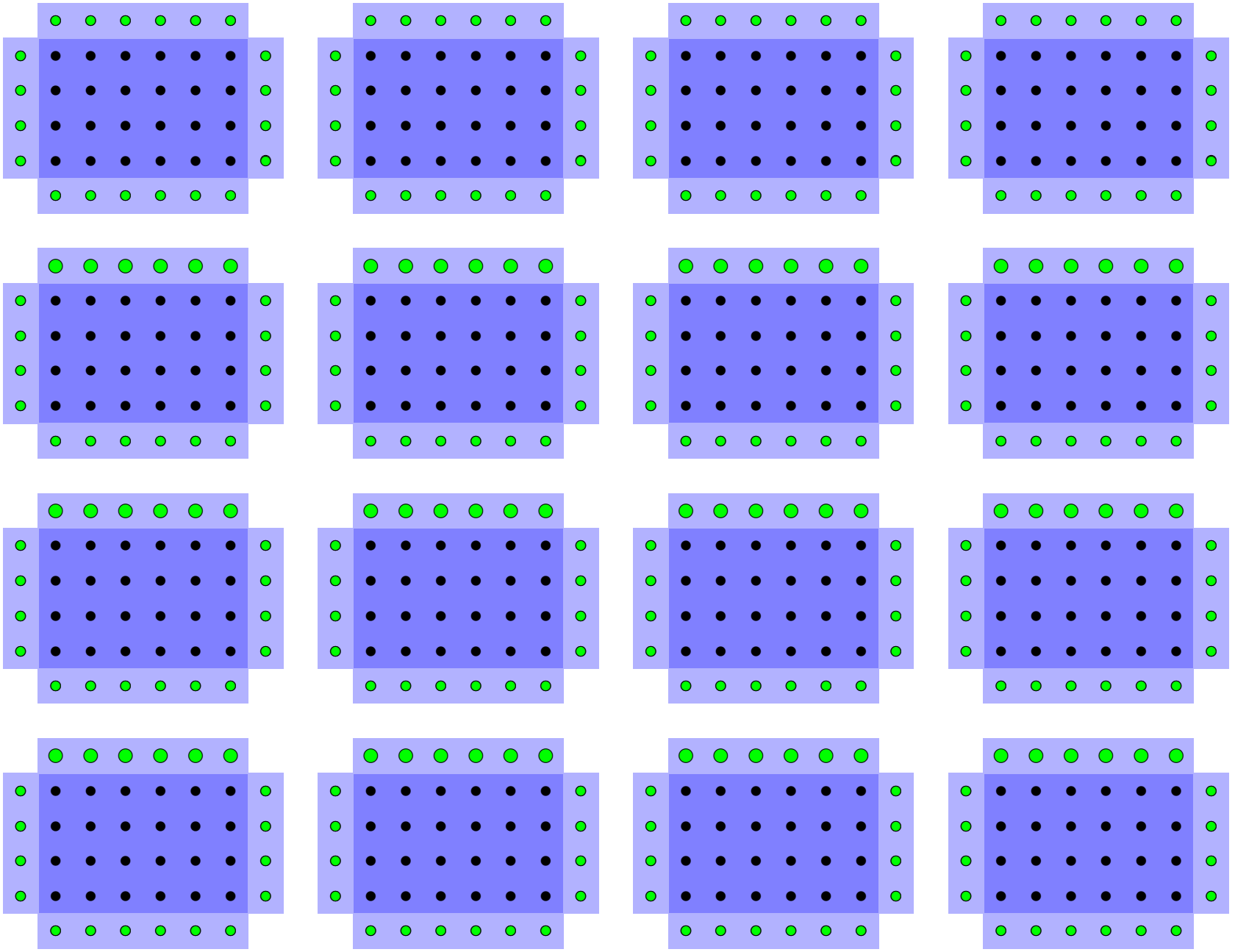
Exchange of boundary elements (with `MPI_Sendrecv`)



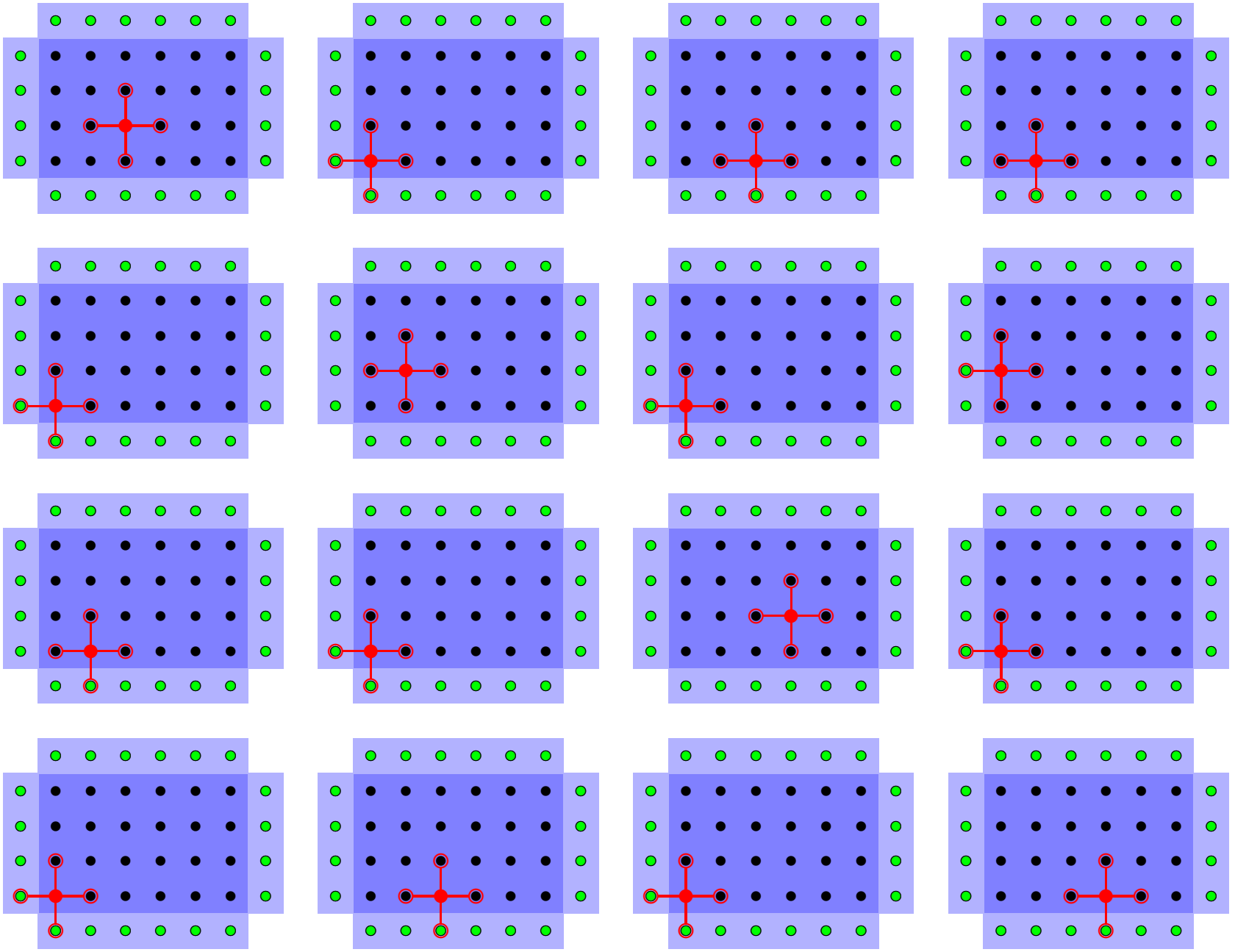
Exchange of boundary elements (with `MPI_Sendrecv`)



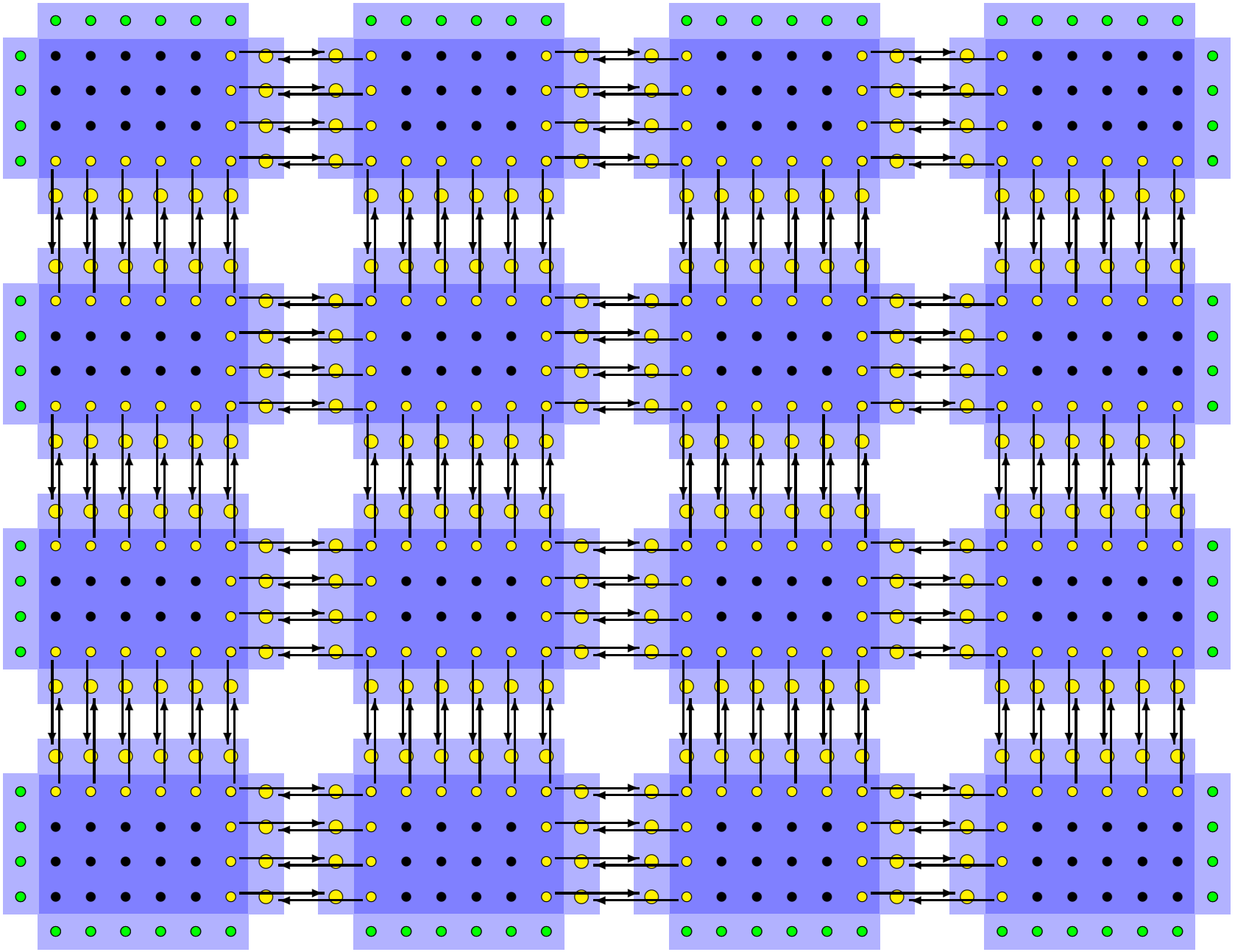
Exchange of boundary elements (with `MPI_Sendrecv`)



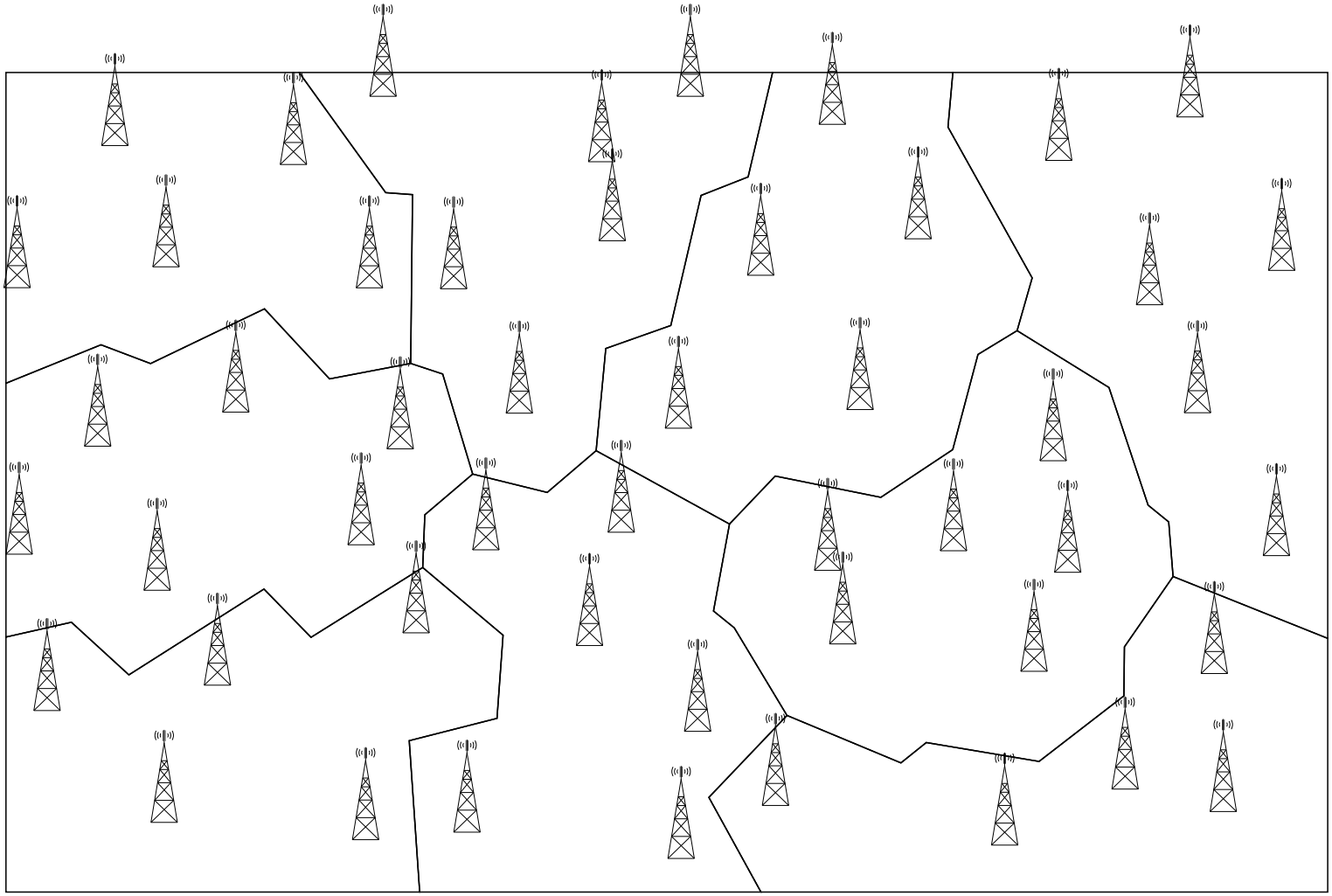
Calculation (MIMD)



Exchange of boundary elements (`MPI_Isend` & `MPI_Irecv`)



Domain decomposition of an unstructured problem



Simulation of a UMTS mobile telephone network

Programming ccNUMA computers

- examples
  - past: ccNUMA machines (SGI Altix and UltraViolet)
  - present: ccNUMA at node level (an HLRN-IV phase-2 node has 4 *NUMA domains)*
- main memory is physically distributed

  $\rightsquigarrow$ performance differences for
    - local and remote memory access
    - accessing memory in different NUMA domains
  - worst case: all data is stored on a single node or a single NUMA domain

    $\rightsquigarrow$ most memory access is remote

    $\rightsquigarrow$ bottleneck
- aim
  - data should be stored where it is processed *(data locality)*

$\rightarrow$ memory pages are allocated according to a *first touch policy*

Performance considerations

Speed-up and efficiency (I)

- compute time

$$T = T(N_{\text{data}}, N_{\text{proc}})$$

- speed-up

$$S = S(N_{\text{data}}, N_{\text{proc}}) = \frac{T(N_{\text{data}}, 1)}{T(N_{\text{data}}, N_{\text{proc}})}$$

- efficiency

$$E = \frac{S}{N_{\text{proc}}}$$

Speed-up and efficiency (II)

- relation of speed-up and efficiency

speed-up	efficiency	remark
$0 < S < N_{\text{proc}}$	$0 < E < 1$	normal situation
$S = N_{\text{proc}}$	$E = 1$	linear/ideal speed-up
$S > N_{\text{proc}}$	$E > 1$	super-linear speed-up

- remark
  - typically, speed-up relates sequential and parallel execution time of a given parallel program
  - fair speed-up: comparison with the best know sequential algorithm

Scaling

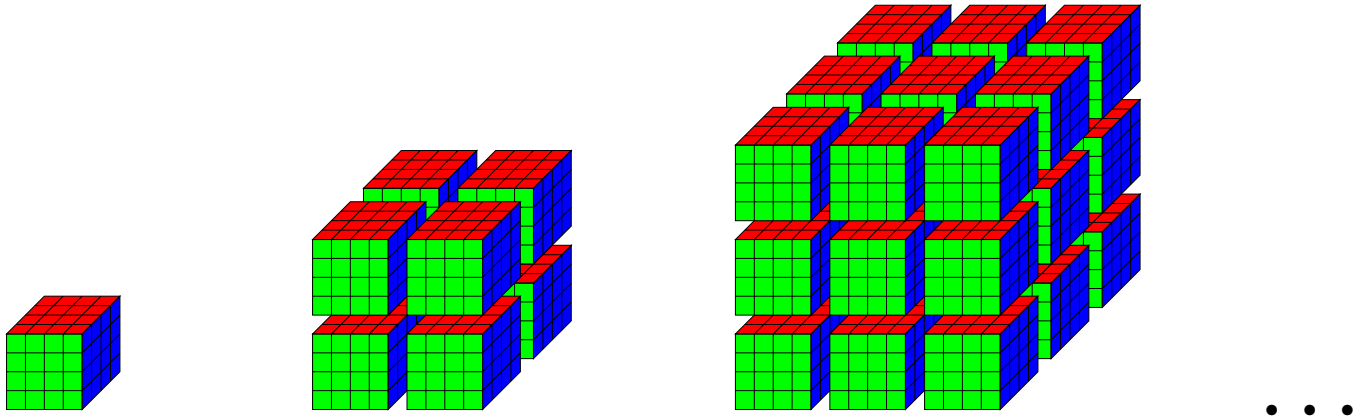
- weak scaling
  - increase N_{proc}, keep domain size per process constant, i.e. $N_{\text{data}} \propto N_{\text{proc}}$
    $\rightarrow$ communication overhead of boundary exchange increases at the beginning
    $\rightarrow$ sustained performance per process is roughly constant at the end
  - use case: performance prediction for very many processes

- strong scaling
  - increase N_{proc}, keep problem size N_{data} constant
    $\rightarrow$ domain size per process decreases
    $\rightarrow$ communication overhead increases
    $\rightarrow$ sustained performance per process decreases
  - use case: determination of optimal number of processes to run on

Scaling examples

- the scaling plots on the following slides come from
  - *conjugate gradient solvers* in simulations of *lattice gauge theories*
  - the problems are 4-dimensional
  - communication patterns
    - boundary exchange
    - global sum

Weak scaling – geometry example



1 process $2^3 = 8$ processes $3^3 = 27$ processes

Weak scaling plots

- plot *performance per process* or *execution time* over the *number of processes*

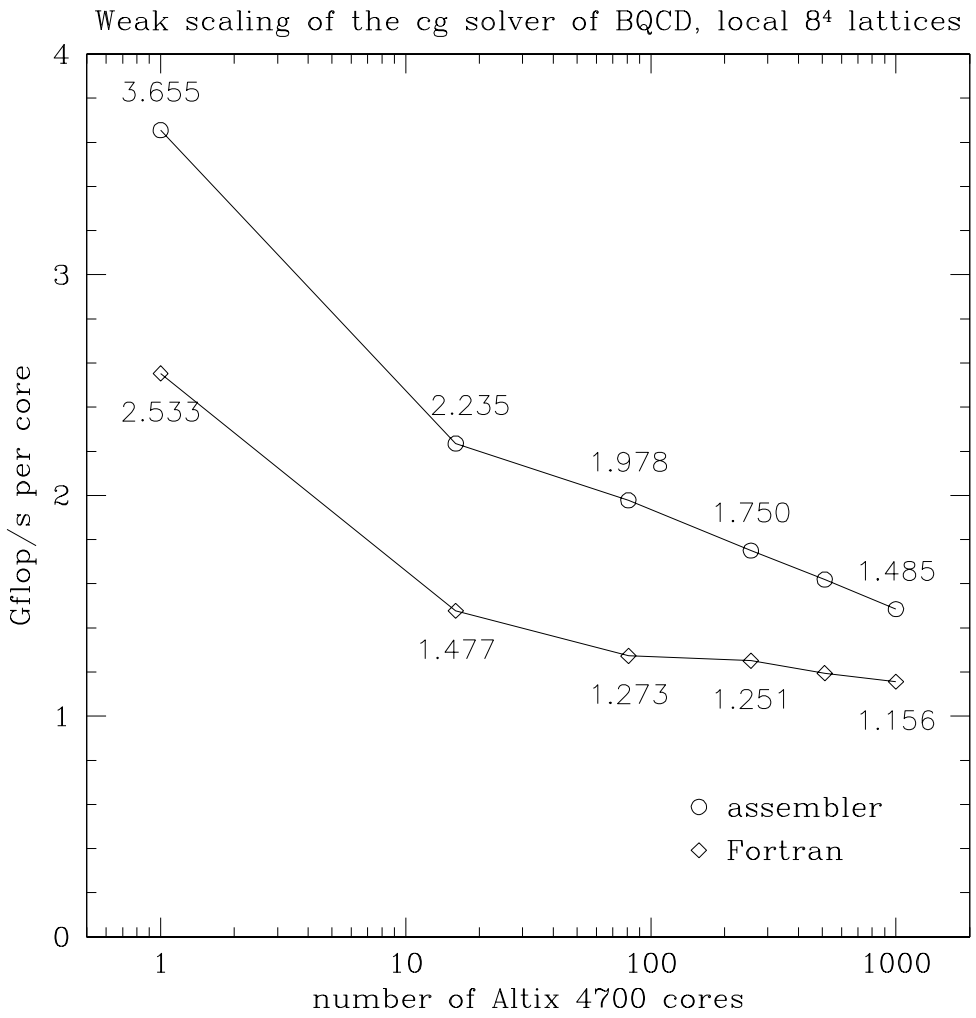
  $$\text{(execution time)} \propto \text{(performance per process)}^{-1}$$

- a *logarithmic x-scale* is better suited if the number of processes varies over orders of magnitudes

- performance per process is expected to decrease an then flatten out

- execution time is expected to increase an then flatten out

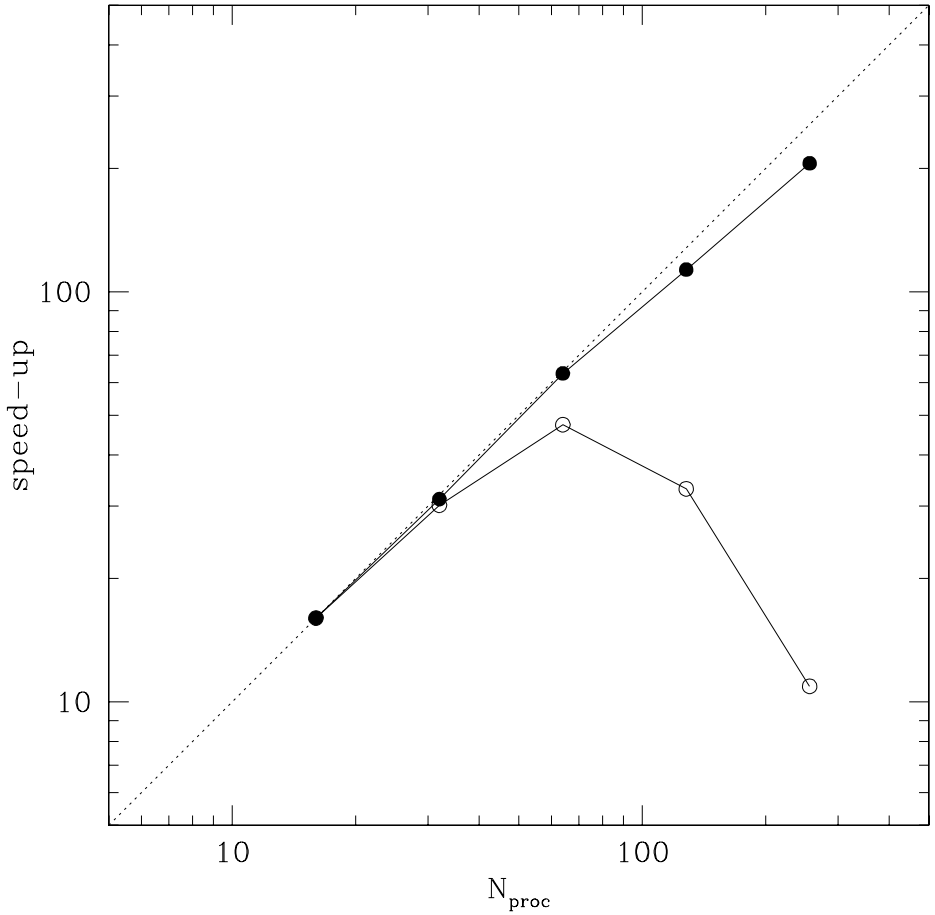
Weak scaling plot example

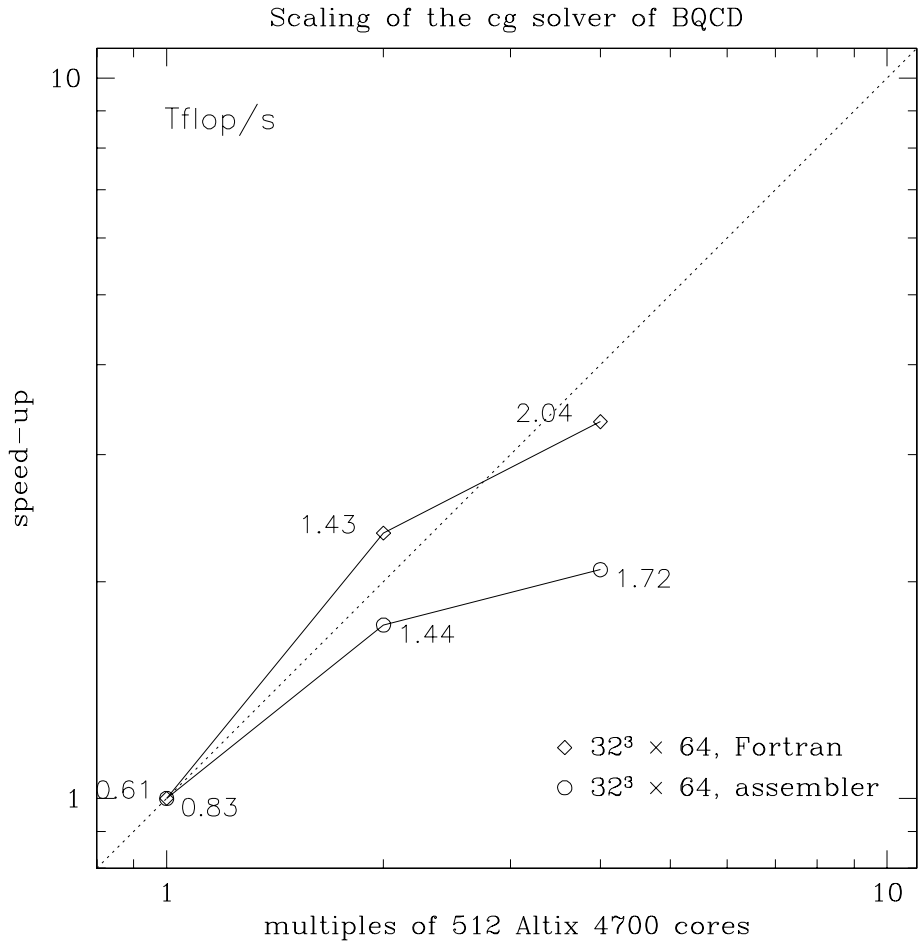


Strong scaling plots (I)

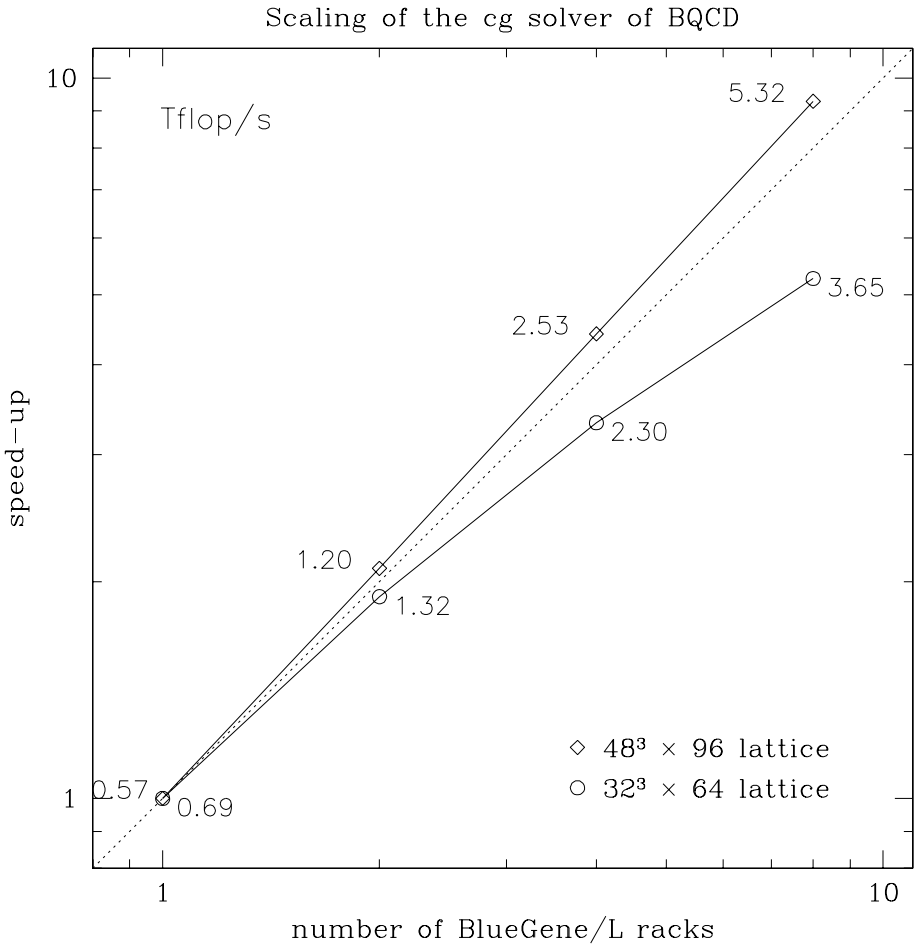
- a classic strong scaling plot shows *speed-up* over the *number of processes*
  - a *reference point* is required, where the speed-up is 1
  - originally a single process was chosen
  - today a single node is a natural reference point
  - other points can be chosen, e.g. a rack
- typically linear scaling is indicated for comparison
- double logarithmic plots are advantageous
  - orders of magnitudes can be better resolved
    (linear scaling is still represented by a straight line)
- plotting speed-up can be misleading when comparing scaling curves
  - scaling can look better, while performance is worse
    (slow programs scale better)

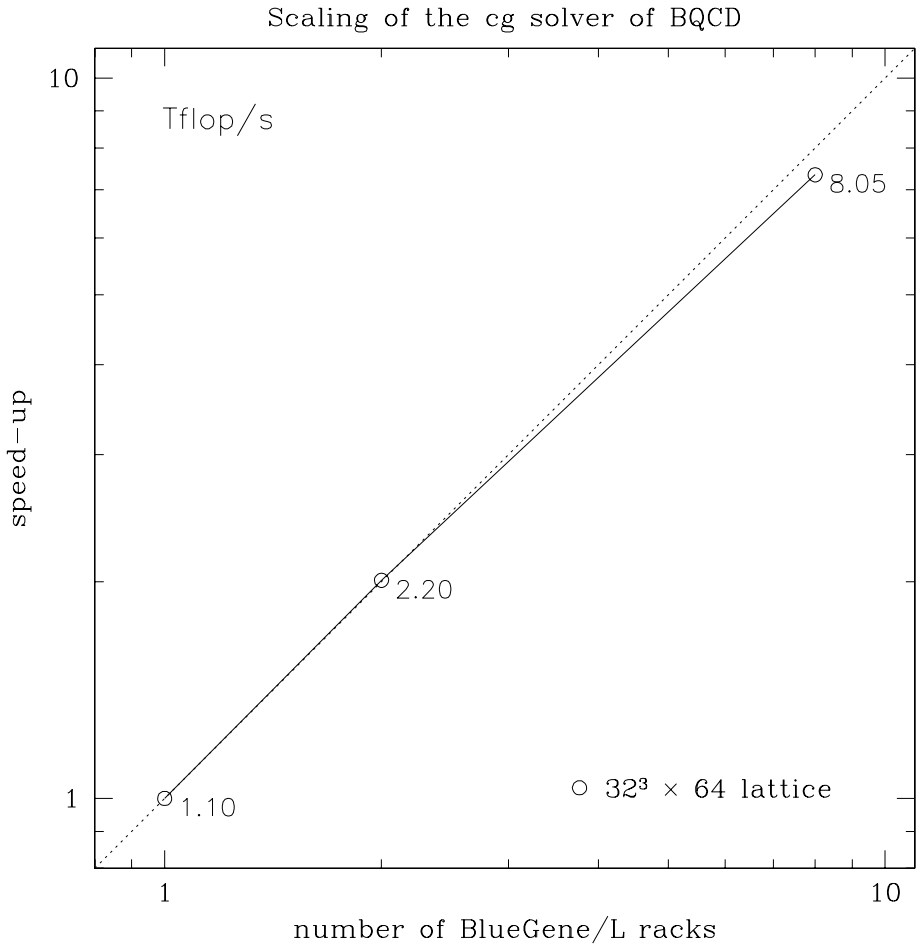
Strong scaling plot examples (I)





Strong scaling plot examples (II)

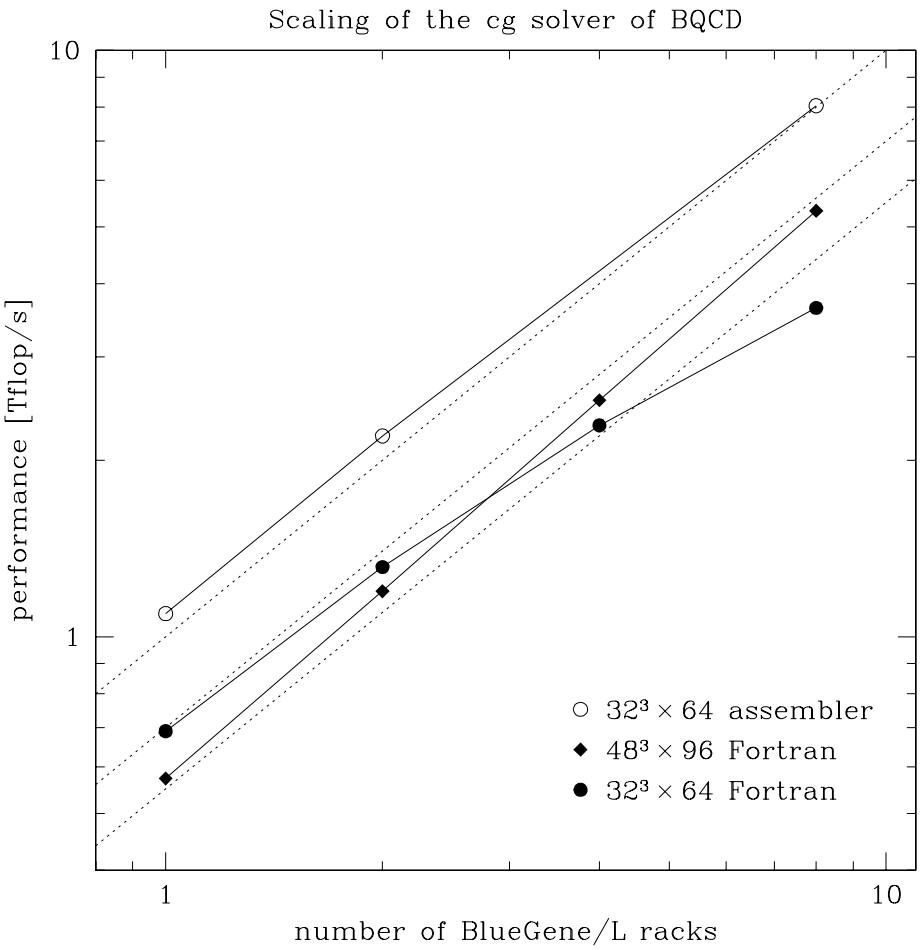


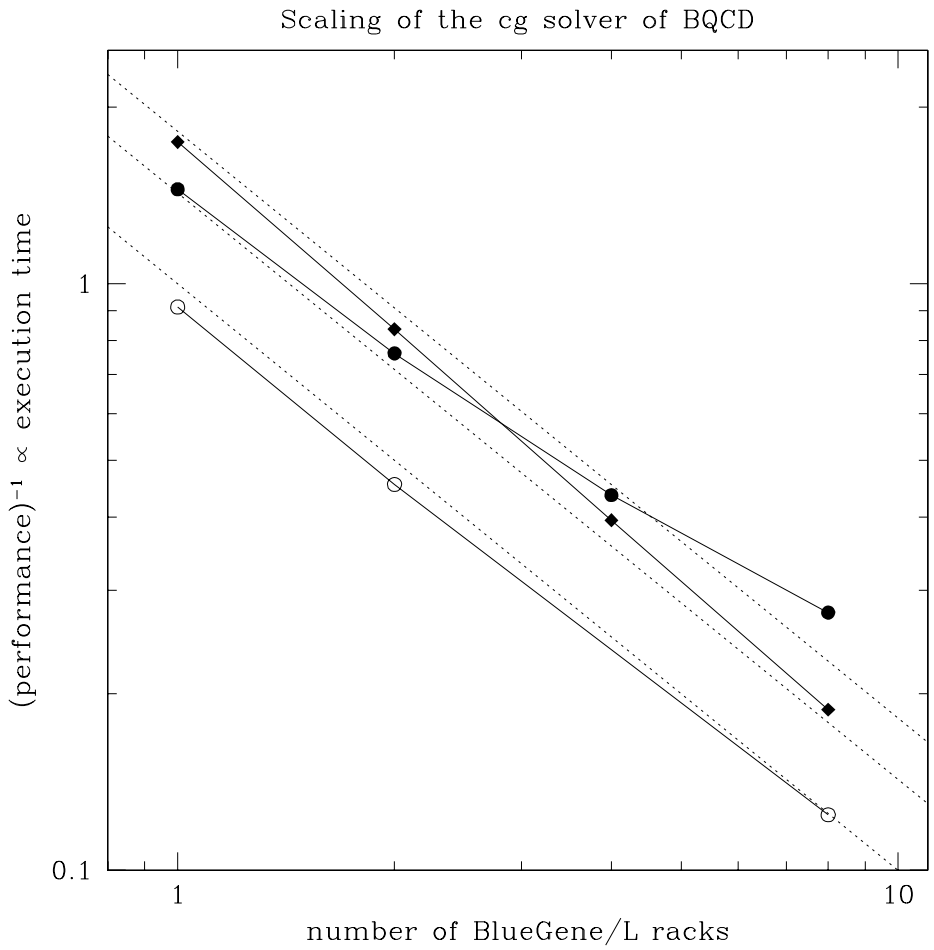


Strong scaling plots (II)

- instead of *speed-up* one can plot *overall performance* or *execution time*
  execution times have to be comparable (e.g. time per mesh point, time per iteration)
- in such plots one can directly compare scaling *and* performance
- linear scaling
  - no reference point $\rightarrow$ no distinct reference line
  - several lines indicating linear scaling can help the eye
- again, double-logarithmic plots are advantageous

Strong scaling plot examples (III)



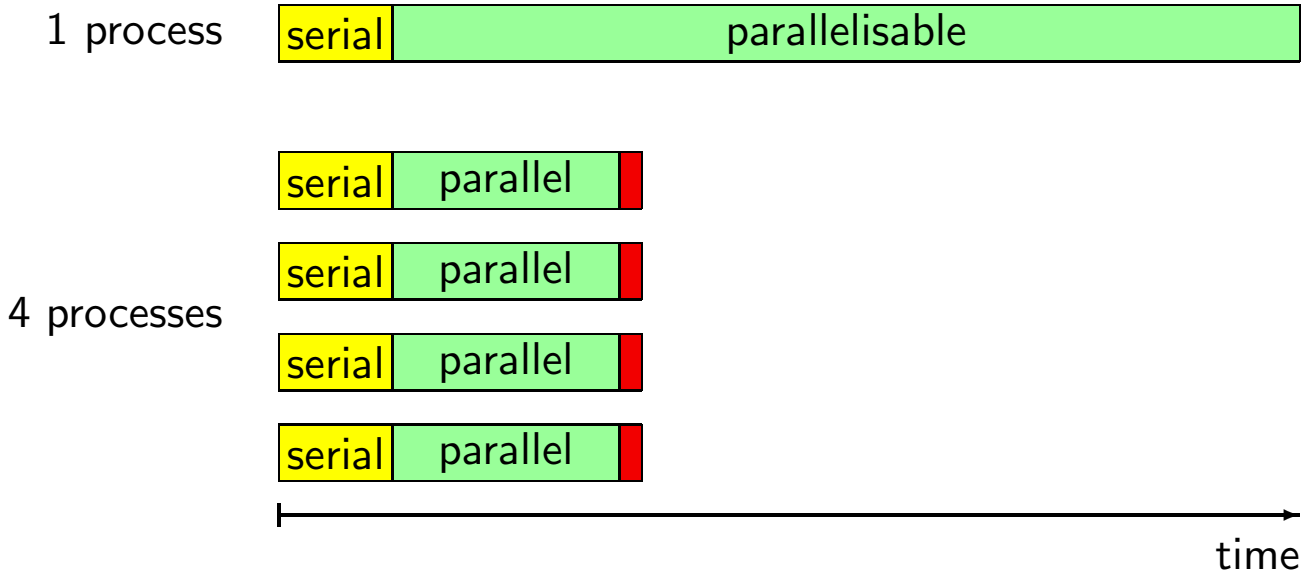


Timing of real parallel programs (I)

- compute time

$$T(N_{\text{data}}, 1) = t_{\text{serial}}(N_{\text{data}}) + t_{\text{parallel}}(N_{\text{data}})$$

$$T(N_{\text{data}}, N_{\text{proc}}) = t_{\text{serial}}(N_{\text{data}}) + \frac{t_{\text{parallel}}(N_{\text{data}})}{N_{\text{proc}}} + t_{\text{overhead}}(N_{\text{data}}, N_{\text{proc}})$$



Timing of real parallel programs (II)

- sources of parallelisation overhead
  - data exchange and synchronisation
  - algorithms (e.g. global reduction operations)
  - software (e.g. address calculations)
- other sources of parallel inefficiency
  - the problem itself
    - unbalanced load
  - software
    - serial parts (see Amdahl's law)
  - hardware (shared memory nodes)
    - NUMA
    - false sharing

Amdahl's law (I)

- compute time neglecting parallelisation overhead

$$T \geq t_{\text{serial}} + \frac{t_{\text{parallel}}}{N_{\text{proc}}}$$

- fraction of work that is parallelisable

$$\alpha = \frac{t_{\text{parallel}}}{t_{\text{serial}} + t_{\text{parallel}}}$$

- fraction of work remaining serial

$$1 - \alpha$$

- speed-up if overhead is neglected / maximal speed-up possible

$$S \leq \left(1 - \alpha + \frac{\alpha}{N_{\text{proc}}}\right)^{-1}$$

Amdahl's law (II)

- speed-up limit

$$S_\infty := S(N_{\text{proc}} \to \infty) = \frac{1}{1-\alpha}$$

- example: speed-up S for a given fraction of parallelisable work

	N_{proc}					
α	4	8	32	256	1024	∞
0.9	3.08	4.7	7.8	9.7	9.9	10
0.99	3.88	7.5	24	71	91	100
0.999	3.99	7.9	31	204	506	1000

Amdahl's law (III)

- maximal number of processes for achieving a given efficiency E

$$N_{\text{proc}}(E) = \left\lfloor \frac{\frac{1}{E} - \alpha}{1 - \alpha} \right\rfloor$$

- examples

	E		
α	90 %	75 %	50 %
0.9	2	4	11
0.99	12	34	101
0.999	112	334	1001

Realistic scaling behaviour (I)

Ansätze for $t_{\text{overhead}}(N_{\text{proc}})$

- logarithmic overhead ← global sum, slide 106

$$t_{\text{overhead}}(N_{\text{proc}}) = \beta \log(N_{\text{proc}})$$

- *sqrt* overhead ← boundary exchange, slide 30

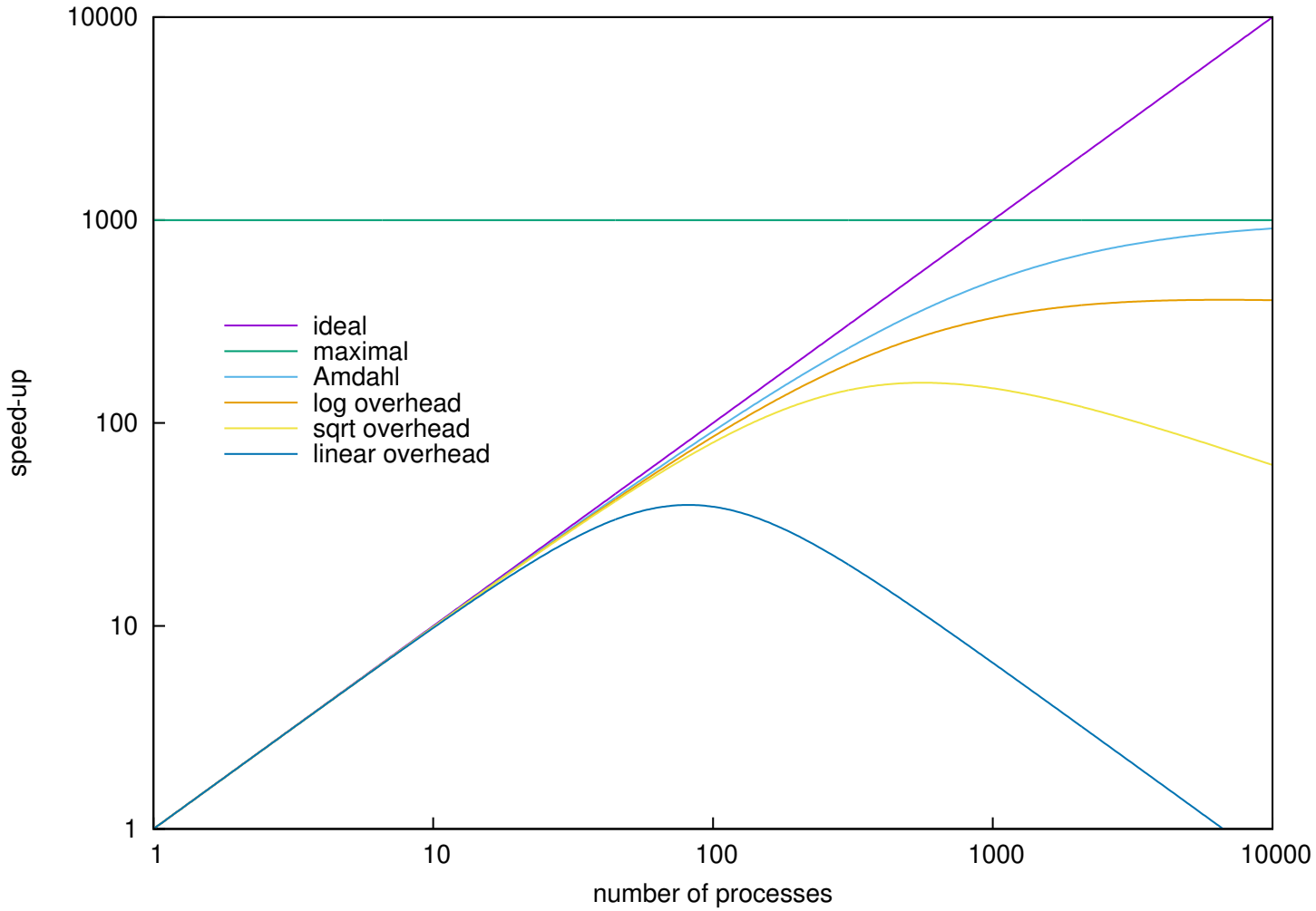
$$t_{\text{overhead}}(N_{\text{proc}}) = \beta \sqrt{N_{\text{proc}} - 1}$$

- linear overhead ← left hand side scaling plot on slide 55

$$t_{\text{overhead}}(N_{\text{proc}}) = \beta (N_{\text{proc}} - 1)$$

Realistic scaling behaviour (II)

- Amdahl vs. realistic scaling behaviour ($\alpha = 0.999$, $\beta = 0.00015$)



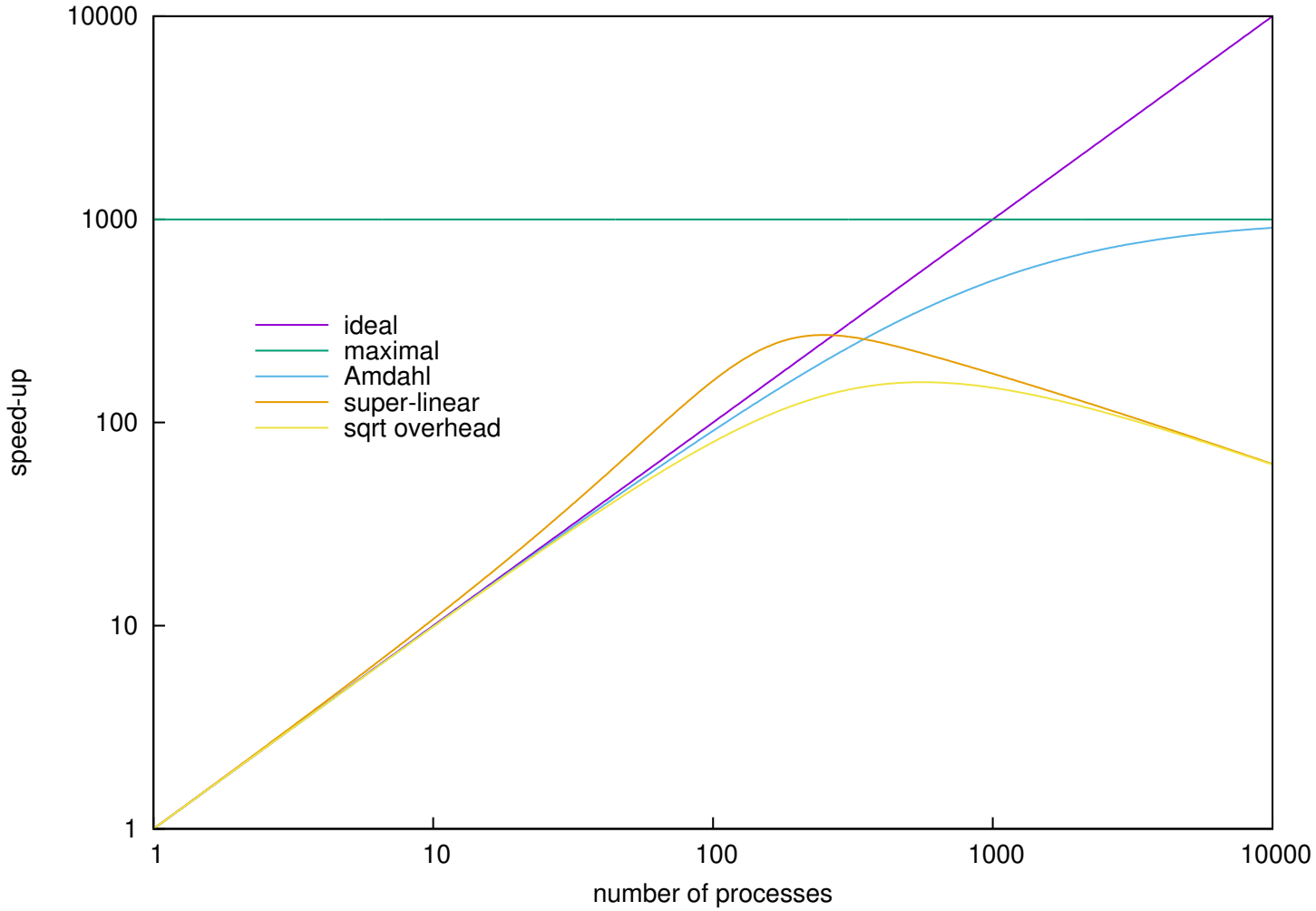
Realistic scaling behaviour (III)

- modeling super-liner speed-up ← right hand side scaling plot on slide 55

$$S = \left(1 - \alpha + \frac{\alpha}{\gamma^{N_{\text{proc}}-1} N_{\text{proc}}} + \beta\sqrt{N_{\text{proc}} - 1}\right)^{-1}$$

Realistic scaling behaviour (IV)

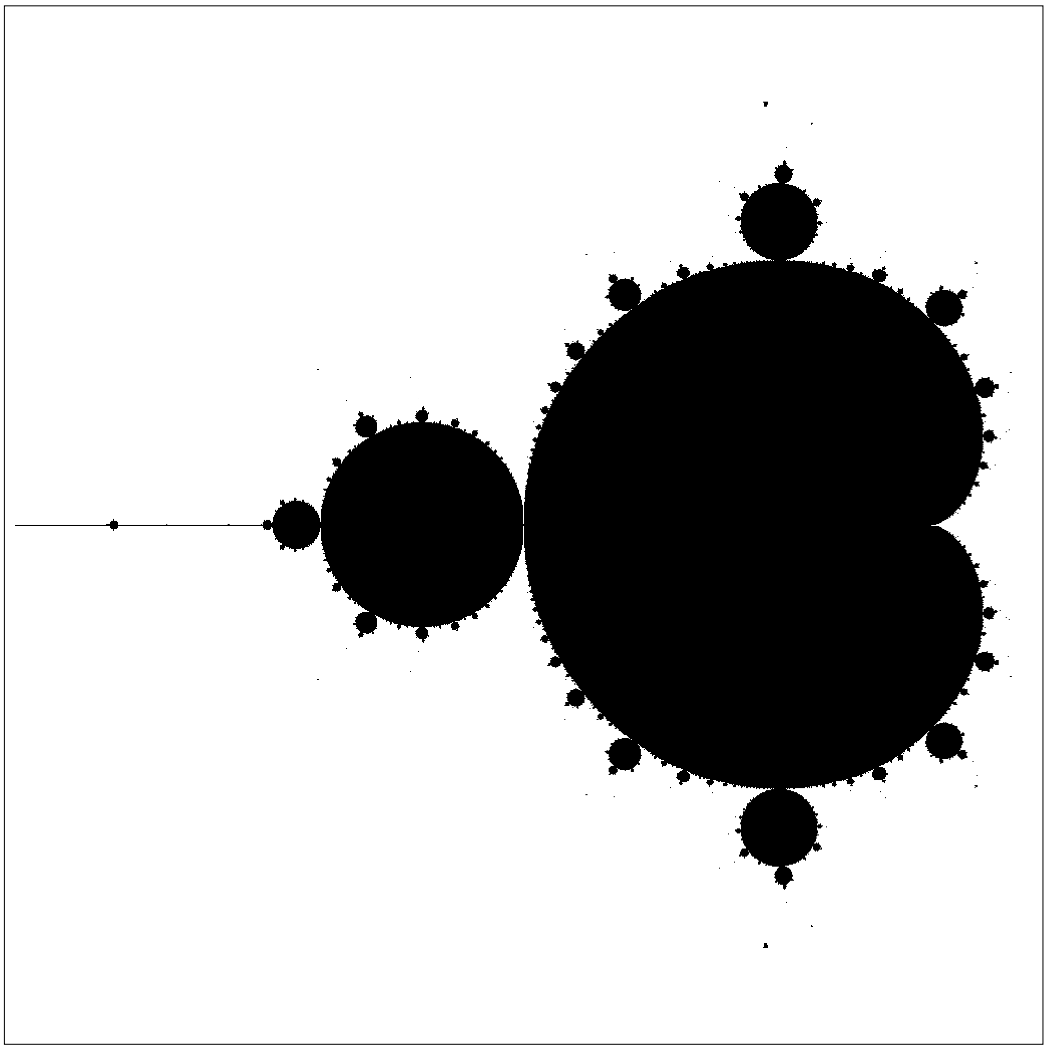
- super-linear scaling behaviour ($\alpha = 0.999$, $\beta = 0.00015$, $\gamma = 1.01$)



Load balancing

- balanced load
  - example: Laplace equation
- unbalanced load
  - example: visualisation of the Mandelbrot set

Mandelbrot set



Mandelbrot set M

- definition
  - for $c \in \mathbb{C}$ consider the iteration

$$z_{k+1} = z_k^2 + c, \quad z_k \in \mathbb{C}, \quad z_0 = 0$$

  - c is an element of M if all $|z_k|$ are bounded

- properties
  1. $|c| \leq 2$
  2. $|z_k| > 2 \Rightarrow c \notin M$

Mandelbrot set M

- algorithm to check $c \overset{?}{\in} M$
  - Fortran implementation

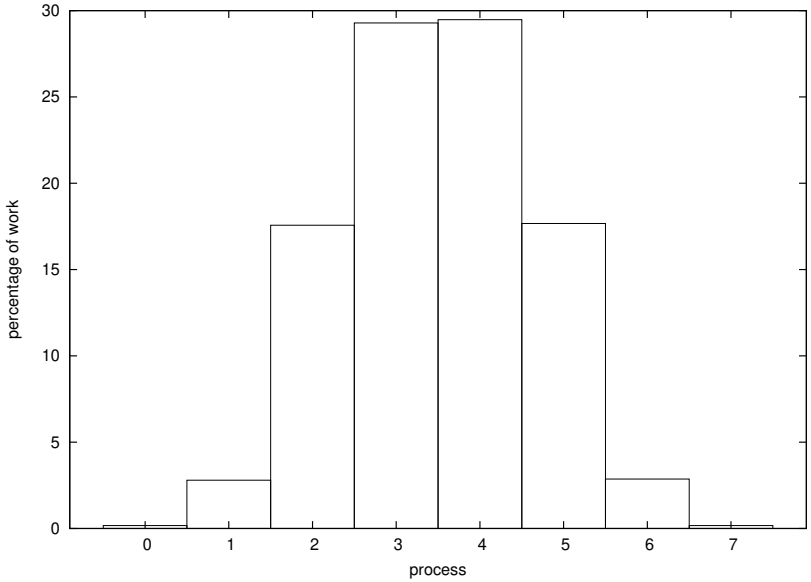
```
is_element = .true.
z = 0
do iter = 1, 1000
   z = z**2 + c
   if (abs(z) > 2) then
      is_element = .false.
      exit
   endif
enddo
```

Mandelbrot set

- calculation
  - quadratic region $[-2.0, 0.5) \times [-1.25, 1.25)$
  - 1024×1024 points
- parallelisation by domain decomposition
  - 8 horizontal strips of equal height
  - 8 vertical strips of equal width
  - $8 \times 8 = 64$ squares of equal size

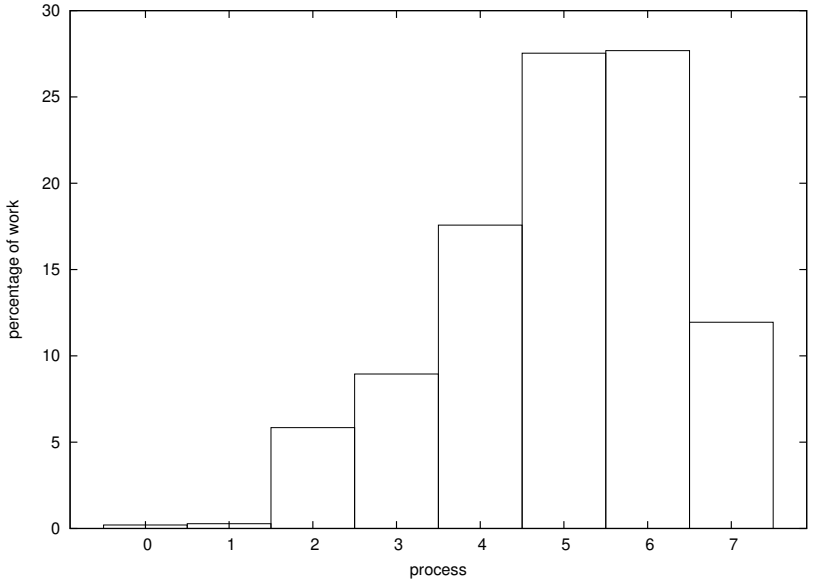
Load distribution (I)

8 horizontal strips



efficiency = 42 %

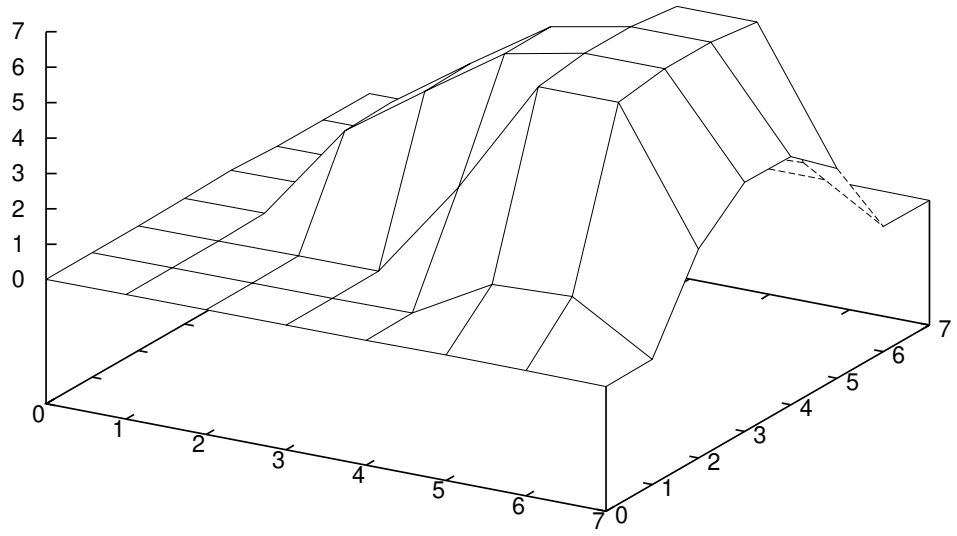
8 vertical strips



efficiency = 45 %

Load distribution (II)

64 squares

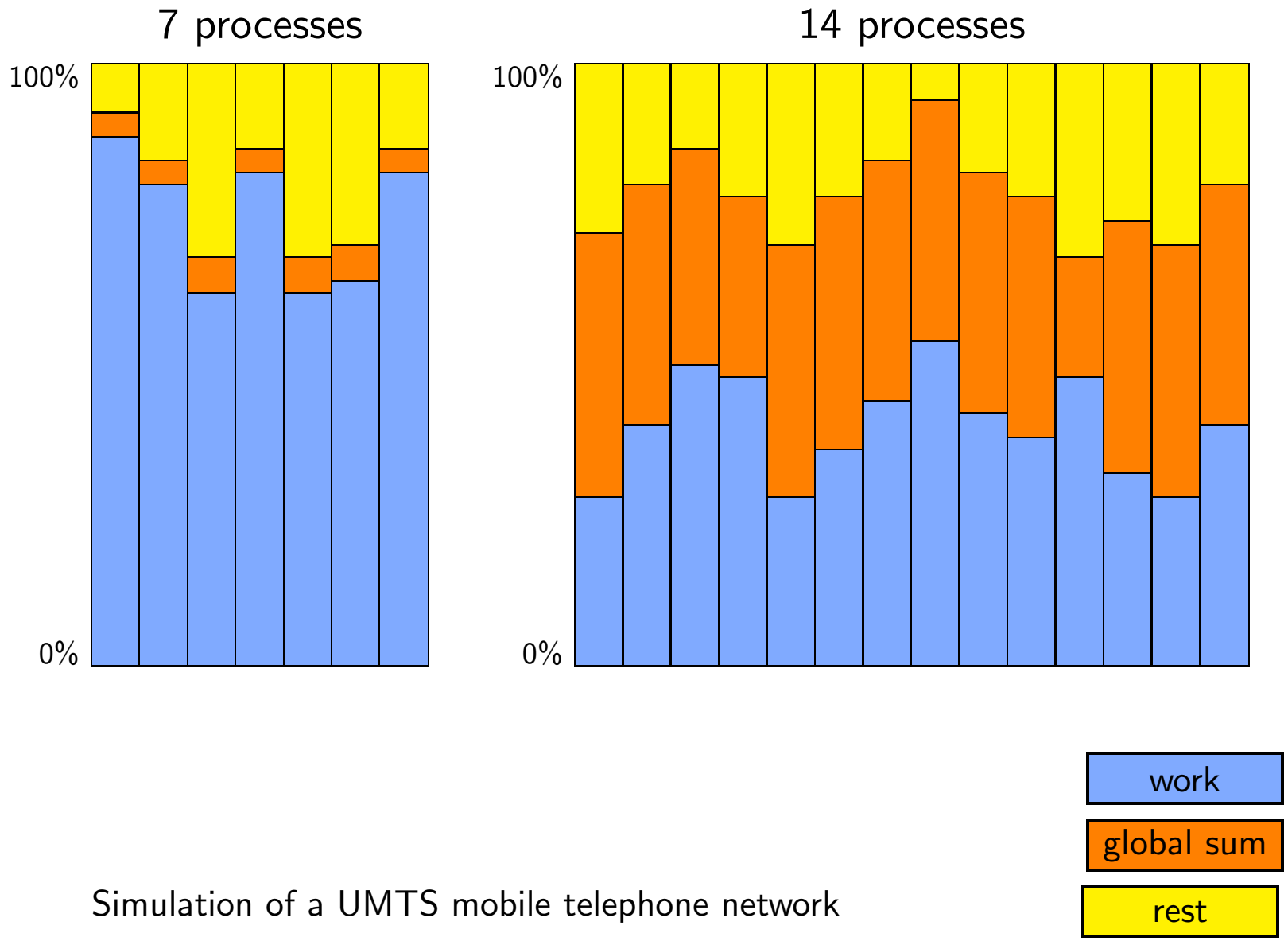


efficiency = 25 %

Discussion

- for each decomposition each process had to check the same number of points
  - *statically* the load is balanced
- however, numbers of iterations differ
  - *dynamically* the load is not balanced

Overall performance analysis example



Simulation of a UMTS mobile telephone network

Optimising parallel performance

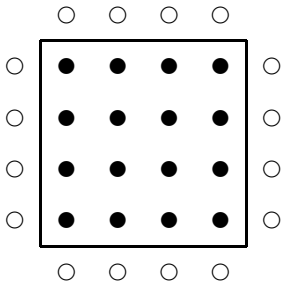
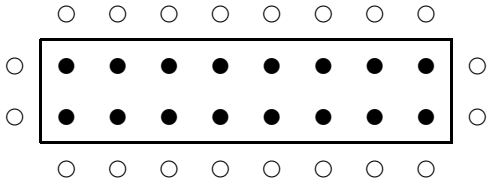
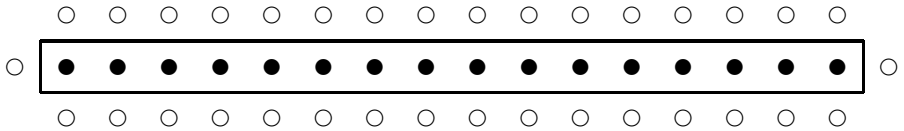
- focus
  - run-time improvements for a given parallel (MPI) program
  - not: optimal parallel implementation

- topics
  - choice of domain decomposition (optimal amount of data traffic)
  - mapping of decomposition to machine (optimal use of the network)

Minimal amount of data traffic

- (minimal amount of data traffic) $\equiv$ (smallest surface to volume ratio)
- the surface of a sphere (circle) is minimal
- for Cartesian grids the surfaces of cubes (squares) are minimal
- practical difference
  - minimum (smallest amount of data)
  - optimum (shorest communication time)

Surface to volume ratio – examples

domain	surface	volume	$\frac{\text{surface}}{\text{volume}}$
	16	16	1
	20	16	1.25
	34	16	2.125

Why is optimal $\neq$ minimal?

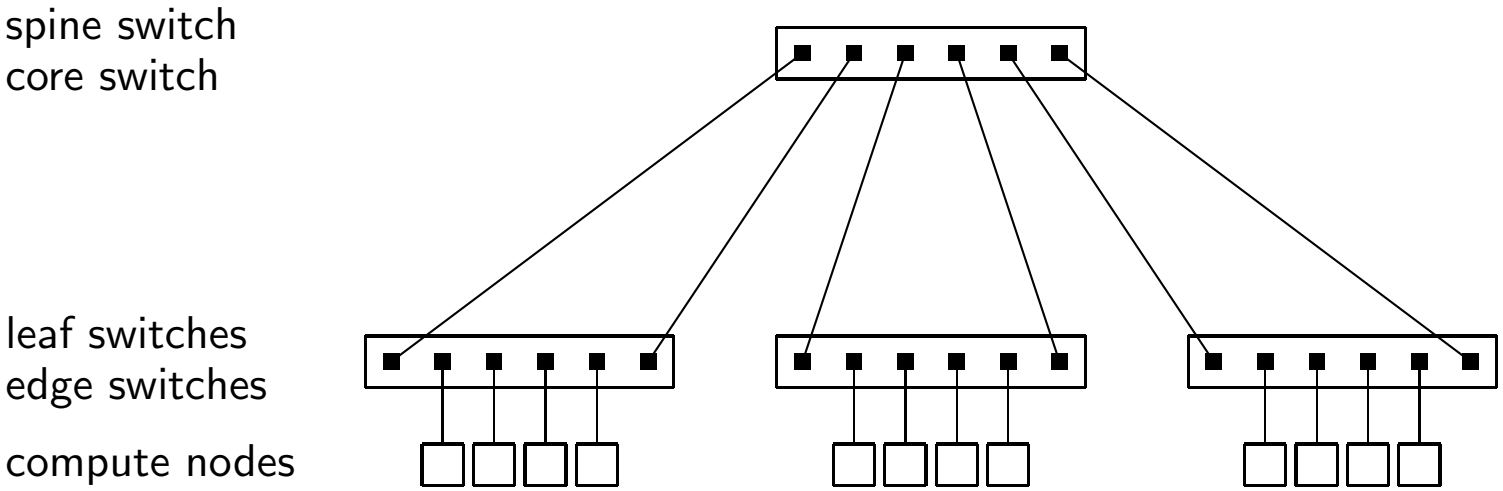
- data communication issues
  - example: boundary exchange in the 2-dim. Laplace case
  - optimal memory access
    - boundaries in $\pm y$ directions are *consecutive* in memory
  - non-optimal memory access
    - boundaries in $\pm x$ directions are *block-strided* (added memory access latencies)
- compute performance issues
  - data cache usage

The mapping problem

- data communication performance is not homogeneous
- communication hierarchy
  - intra-node (speed of main memory)
    - fastest communication
    - NUMA domains
  - inter-node (speed of network)
    - typically latency and/or bandwidth depend on node distance
    - switched networks: are hierarchical
    - torus networks: node distance varies
- mapping problem
  - reduction of communication overhead by optimal process placement
    - match communication demands with connection properties
    - minimise the maximal distance of communication partners

Hierarchical network example

- fat-tree with 2:1 blocking



- within a leaf switch: lowest latency, full network bandwidth
- via spine switch: added latency, half network (bisection) bandwidth

Bisection bandwidth

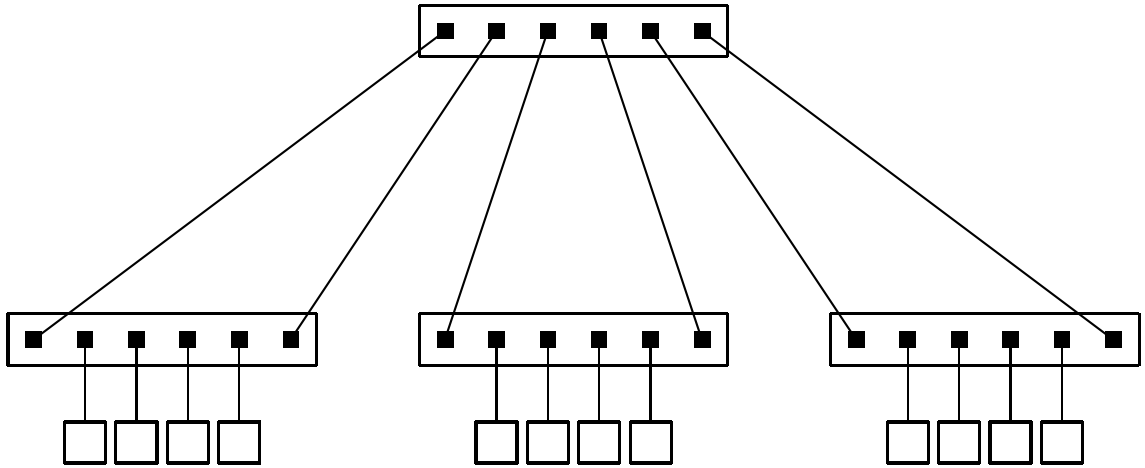
- bisection

  a division into two parts

- bisection bandwidth of a network

  minimum bandwidth between bisections

Two bisections of a 2:1 blocking fat-tree



bisection 1 A A B B | C C D D | E E F F — *full bandwidth (is possible)*

bisection 2 A B C D | E F A B | C D E F — *half bandwidth (bisection bandwidth)*

→ The bisection bandwidth is not the whole story.
With a good *mapping* one might get higher overall bandwidth.

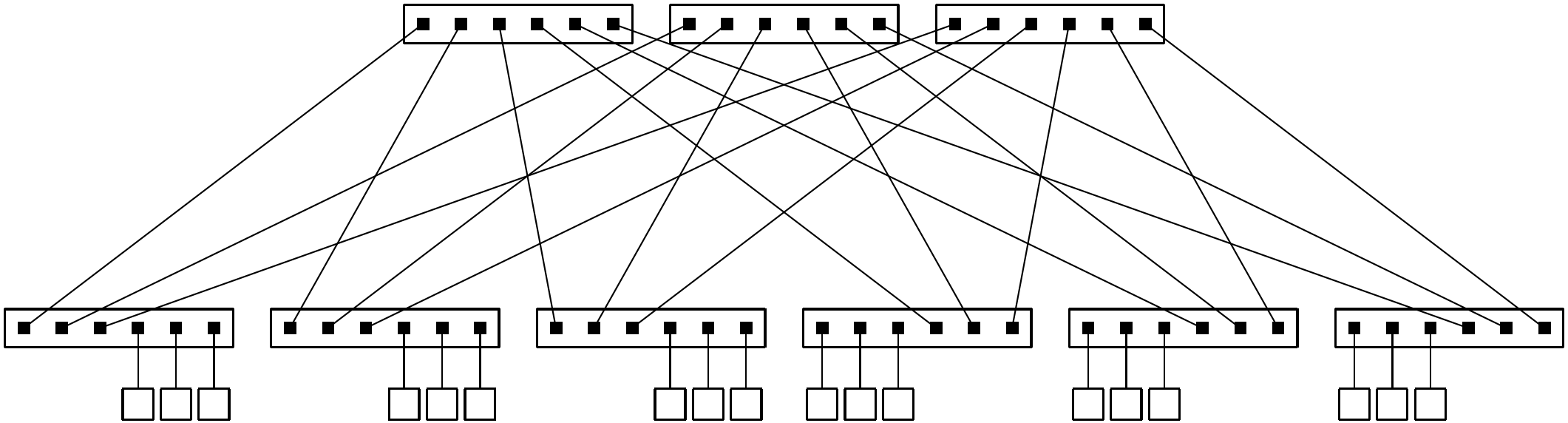
Communication bandwidth hierarchy

- the bandwidth hierarchy is even deeper:

	bandwidth per process*
intra-node (NUMA)	
cores on the same socket	1700 MByte/s
cores on different sockets	1000 MByte/s
inter-node (hierarchical network)	
nodes connected to the same leaf switch	200 MByte/s
nodes connected via spine switches	100 MByte/s

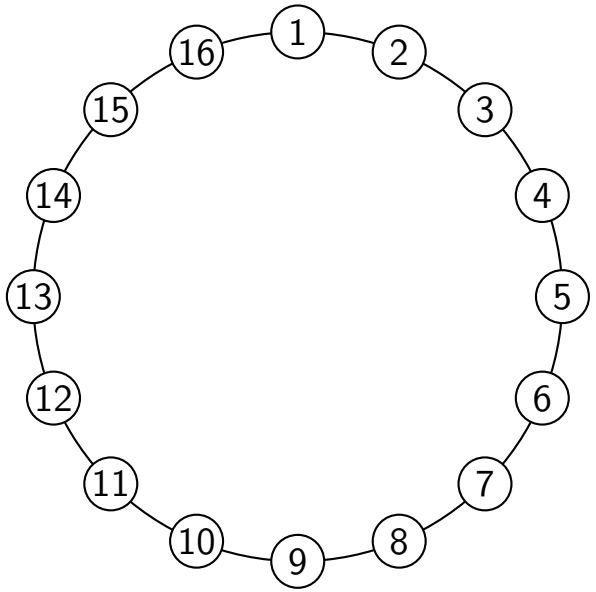
* on the HPC cluster at Universität Hamburg installed in 2015

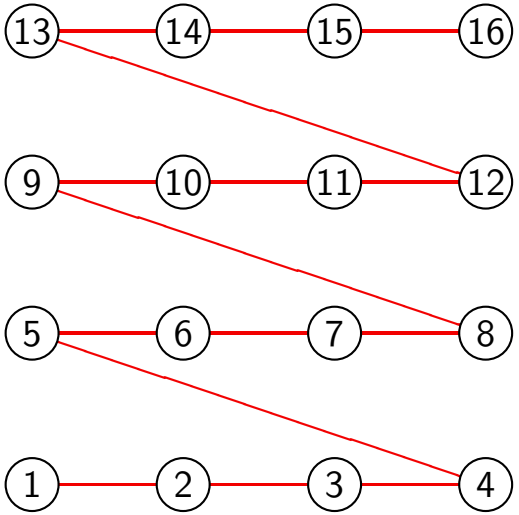
Non-blocking fat-tree



Mapping a ring to a mesh (I)

- canonical mapping

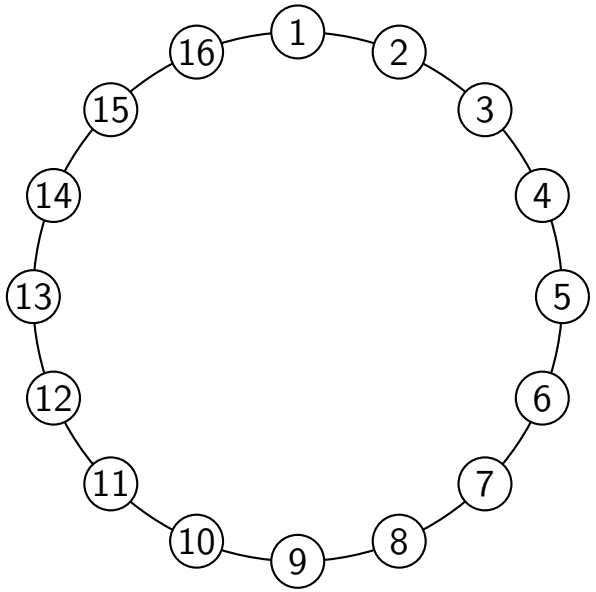


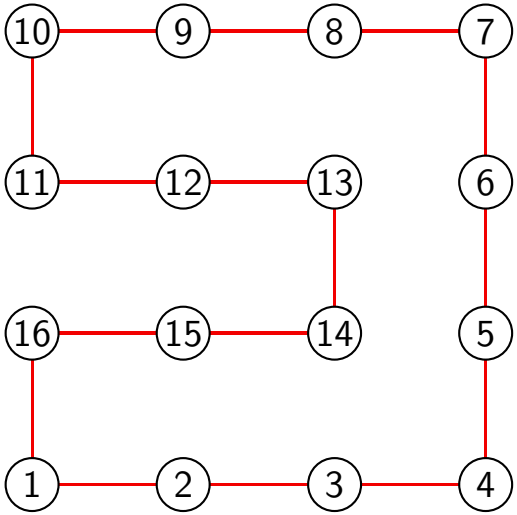


- nearest neighbour communication: maximal number of hops is
  - 6 for this mesh
  - 2 if there was a torus

Mapping a ring to a mesh (II)

- optimised mapping





- nearest neighbour communication: maximal number of hops is 1

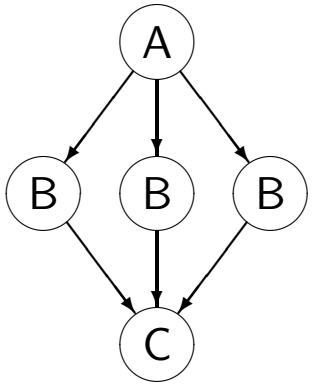
Characterisation of parallelism

Levels of parallelism / granularity

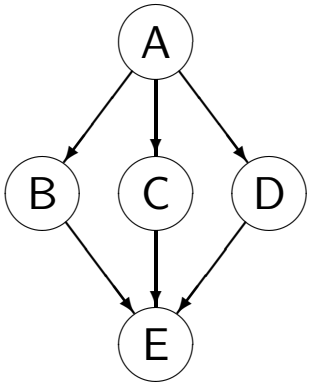
- application level $\rightarrow$ *trivial* parallelism
  - independent runs of one or more programs
- algorithmic level $\rightarrow$ parallel computing
  - decomposing the computational work for multiple processing units
- loop level $\rightarrow$ compiler
  - vectorisation (classical vectorisation and SIMD vectorisation)
  - auto-parallelisation
- machine level $\rightarrow$ hardware
  - multiple functional units (multiplication, addition, . . .)
  - pipelining
  - multiple pipelines
  - SIMD units

Characterisation of parallelism

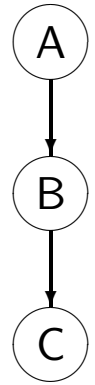
- dependence graphs for operations A,B,C,D,E



data parallelism



functional parallelism



no parallelism

Data parallelism

- identical operations, multiple data
- example

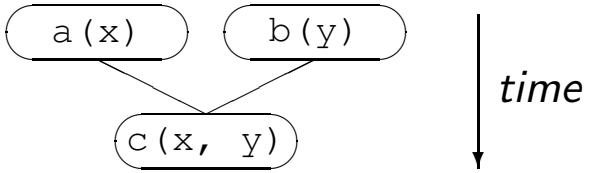
```
for i := 1 to 1000 do
    a[i] := b[i] + c[i]
```

- typically scalability is high

Functional parallelism (I)

- multiple operations
- examples

```
call a(x)
call b(y)
call c(x, y)
```

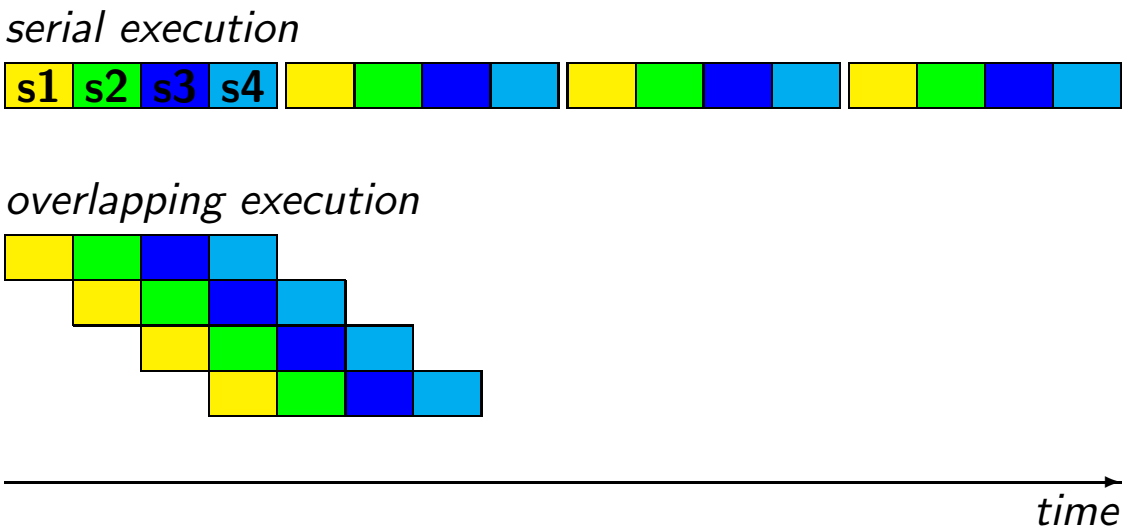



```
x = r * sin(phi)
y = r * cos(phi)
```

- typically scalability is limited

Functional parallelism (II)

- special case: pipeline processing (assembly belt)



Data dependence analysis

Data dependences

- true dependence or flow dependence

```
a := b + c
d := a + e
```

- anti-dependence

```
d := a + e
a := b + c
```

- output dependence

```
a := b + c
a := d + e
```

$\Rightarrow$ dependent statements must not be exchanged

$\Rightarrow$ independent statements can be executed in any order, i.e. in parallel

Data dependences in loops (I)

- true dependence

```
for i := 1 to N do                    a[1] := a[0] + b[1]
   a[i] := a[i - 1] + b[i]            a[2] := a[1] + b[2]
                                      ...
```

- anti-dependence

```
for i := 1 to N do                    a[1] := a[2] + b[1]
   a[i] := a[i + 1] + b[i]            a[2] := a[3] + b[2]
                                      ...
```

- output dependence

```
for i := 1 to N do                    a[5] := b[1]
   a[j[i]] := b[i]                    a[5] := b[2]
                                      ...
```

Data dependences in loops (II)

- we will look at the following loops

```
for i := 1 to N do
    loop body
```

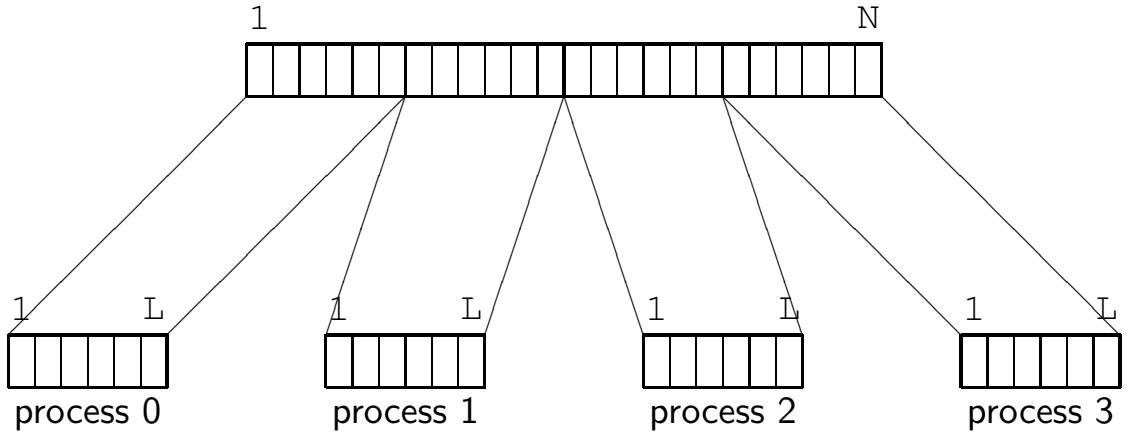
- loop bodies

1.	*full vector*	`a[i] = b[i]`
2.	*gather*	`a[i] = b[j[i]]`
3.	*scatter*	`a[j[i]] = b[i]`
4.	*atomic update*	`a[j[i]] = a[j[i]] + b[i]`
5.	*reduction*	`s = s + a[i]`
6.	*'Jacobi'*	`a[i] = (b[i - 1] + b[i + 1]) / 2.0`
7.	*'Gauß-Seidel'*	`a[i] = (a[i - 1] + a[i + 1]) / 2.0`

Data dependences in loops (III)

- main questions in practical parallelisation
  - Can the work be decomposed into packages that have no data dependences?
  - Can data dependences be resolved by changing the order of execution?
  - Does the change of execution order yield a valid algorithm?

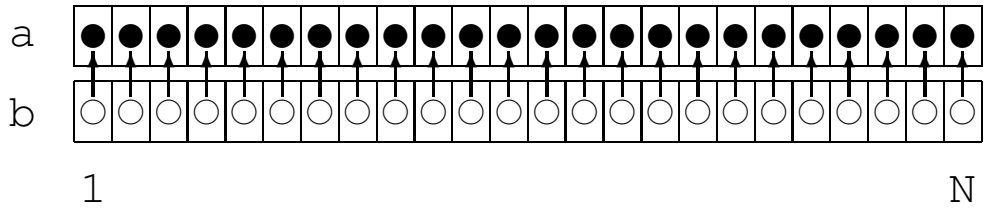
- one possibility of a decomposition



Loop 1

- Local data access (no indirect data access, no index calculations):

```
for i := 1 to N do
   a[i] := b[i]
```

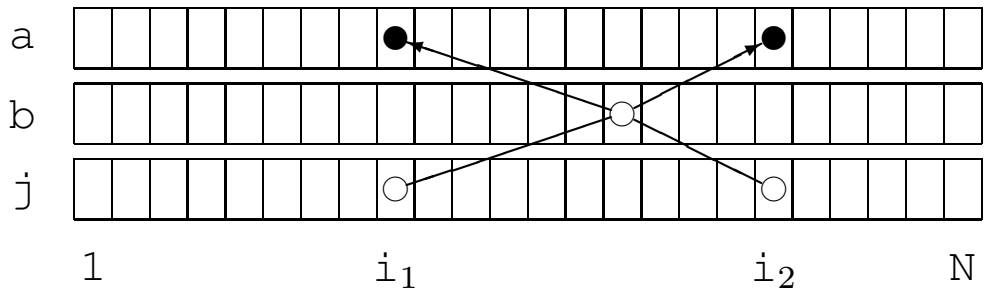


- There are no data dependences ⇒ the loop is parallelisable.

Loop 2

- *Gather* loop (indirect access on the right hand side):

```
for i := 1 to N do
   a[i] := b[j[i]]
```

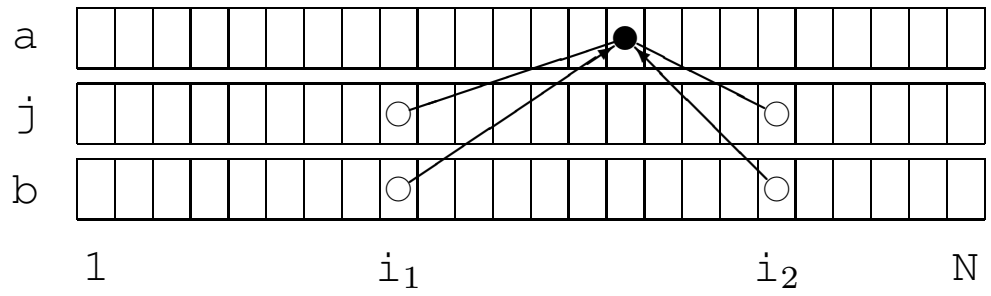


- There are no data dependences $\Rightarrow$ the loop is parallelisable.

Loop 3

- *Scatter* loop (indirect access on the left hand side):

```
for i := 1 to N do
   a[j[i]] := b[i]
```



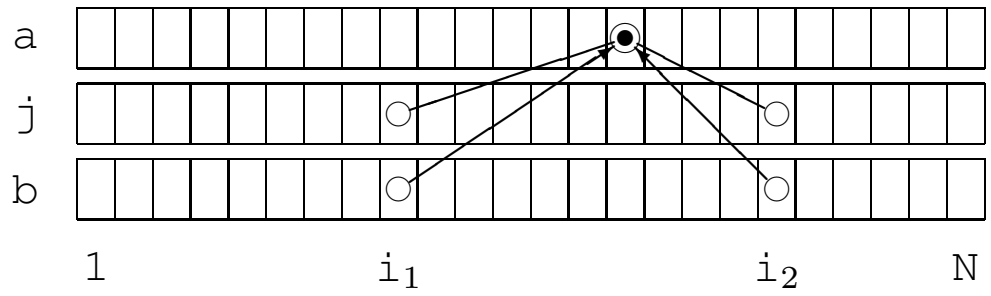
- There is a potential output-dependence depending on `j`
  $\Rightarrow$ for general/unknown `j` the loop has to be processed sequentially.

- There is no dependence if `j` is a permutation of $\{1..N\}$
  $\Rightarrow$ if this is known the loop can be parallelised

Loop 4

- *Atomic update* (indirect access on the left and right hand side):

```
for i := 1 to N do
   a[j[i]] := a[j[i]] + b[i]
```



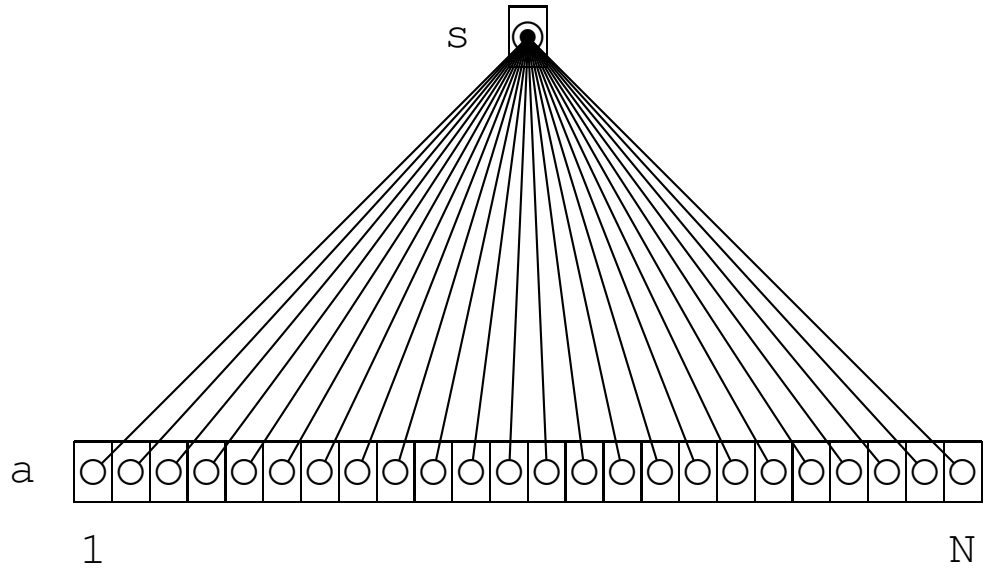
- If `j` is not a permutation of `{1..N}` the loop has flow-, anti- and output-dependences.
- For general/unknown `j` the loop can be processed in such a way that only one process at a time is updating `a[j[i]]`.

  This might change the order of execution and lead to different results because on a computer the addition of floating point depends on the order of operations.

Loop 5

- Global sum:

```
s := 0
for i := 1 to N do
   s := s + a[i]
```

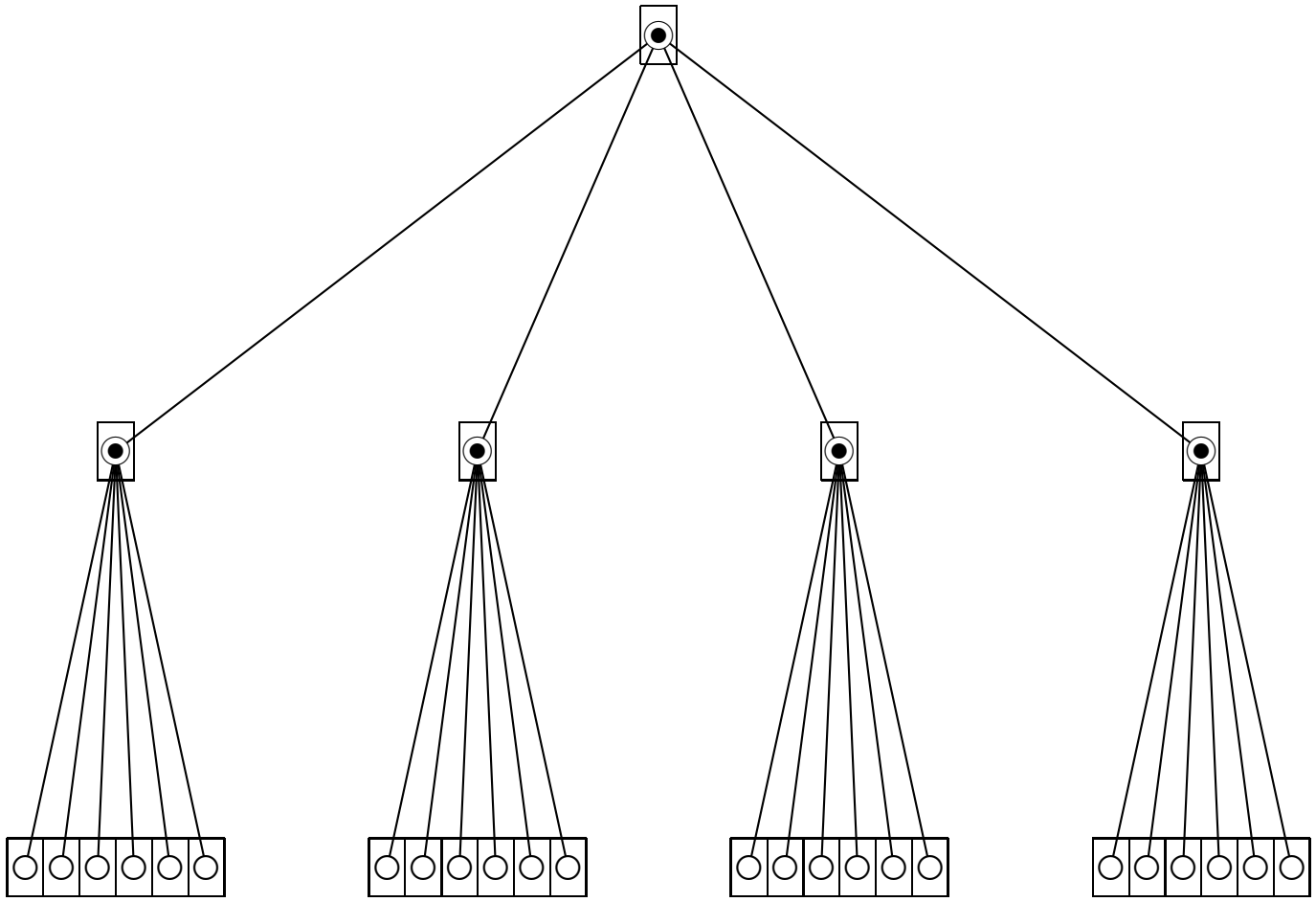


- If one can ignore rounding errors the loop can be processed in any order.
- The loop is parallelisable by introducing several `s`.

Parallelisation of the global sum

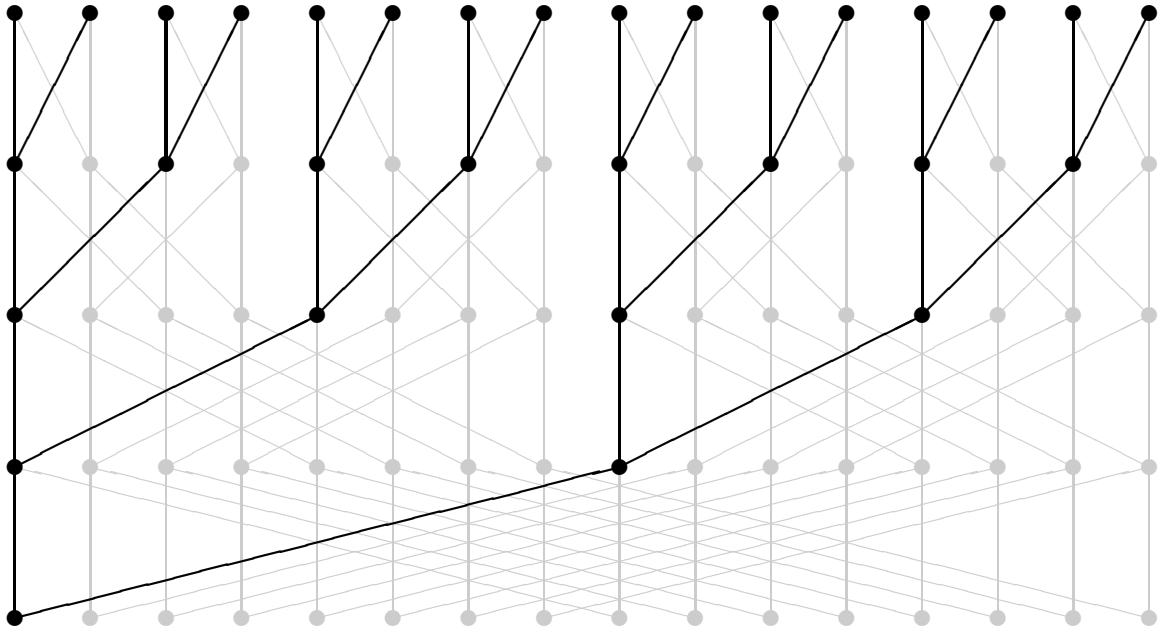
step 2:
global sum

step 1:
local sums



Digression: binary tree

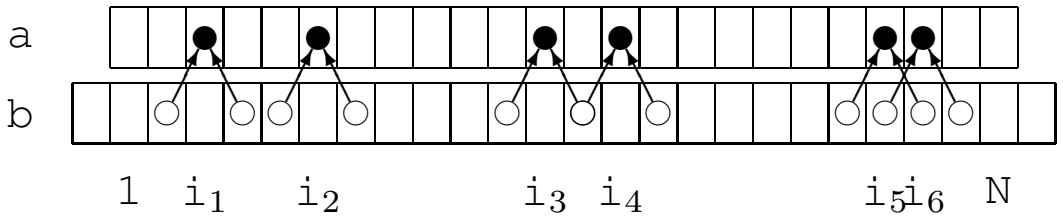
- binary tree algorithm for global operations:
  effort $\propto \log_2$(number of processes)



Loop 6 (nearest neighbours I)

- Different arrays on the left and right hand side (*Jacobi iteration*):

```
for i := 1 to N do
   a[i] := (b[i - 1] + b[i + 1]) / 2
```

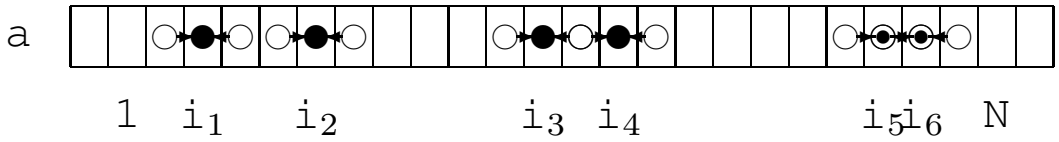


- There are no data dependences $\Rightarrow$ the loop is parallelisable.

Loop 7 (nearest neighbours II)

- The array on the left and right hand side is the same (*Gauß-Seidel iteration*):

```
for i := 1 to N do
   a[i] := (a[i - 1] + a[i + 1]) / 2
```

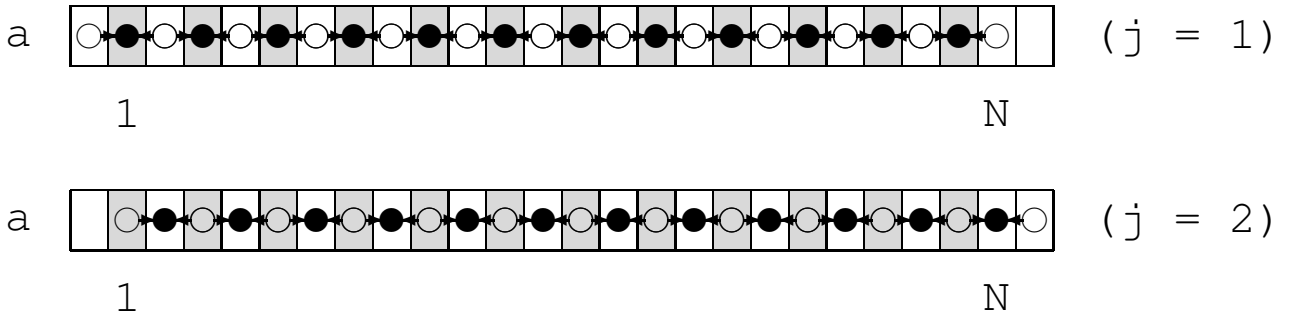


- The loop has flow- and anti-dependences.
- One can remove the dependences by changing the execution order.

Loop 8 (nearest neighbours III)

- Fine grained chessboard decomposition:

```
for j := 1 to 2 do
   for i := j to N step 2 do
      a[i] := (a[i - 1] + a[i + 1]) / 2
```

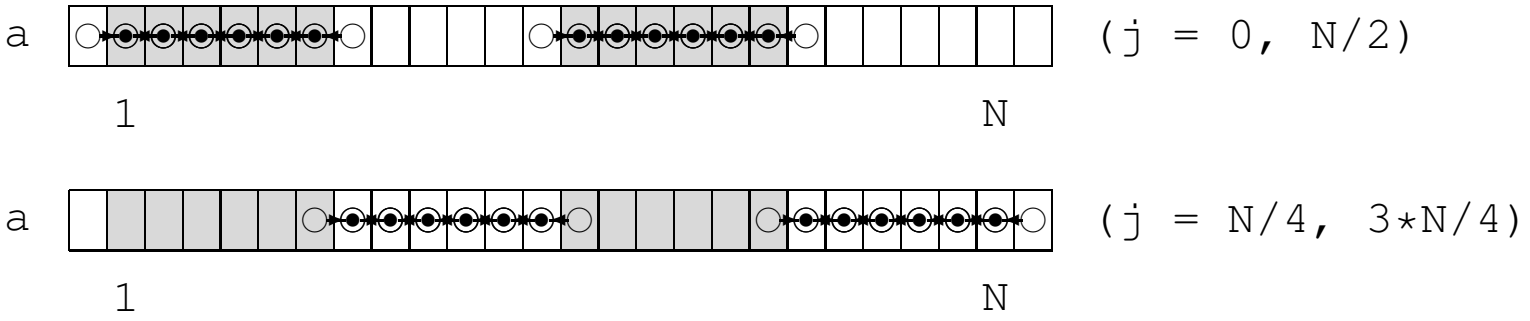


- The inner loop has no data dependences ⇒ the loop is parallelisable.
- The result is independent of the number of processes used.

Loop 9 (nearest neighbours IV)

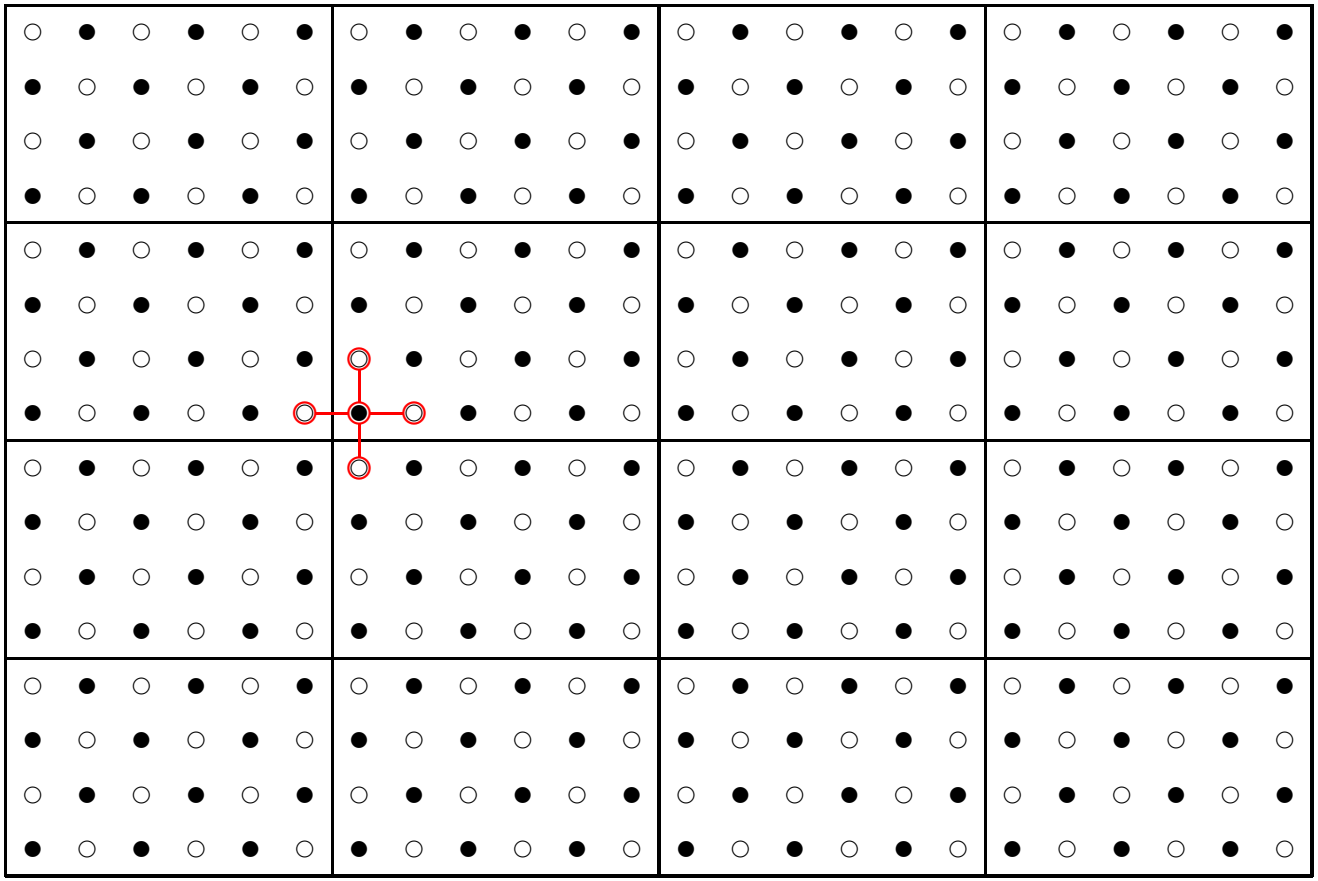
- Coarse grained chessboard decomposition:

```
for j := 0 to N - 1 step N/4 do
   for i := j + 1 to j + N/4 do
      a[i] := (a[i - 1] + a[i + 1]) / 2
```

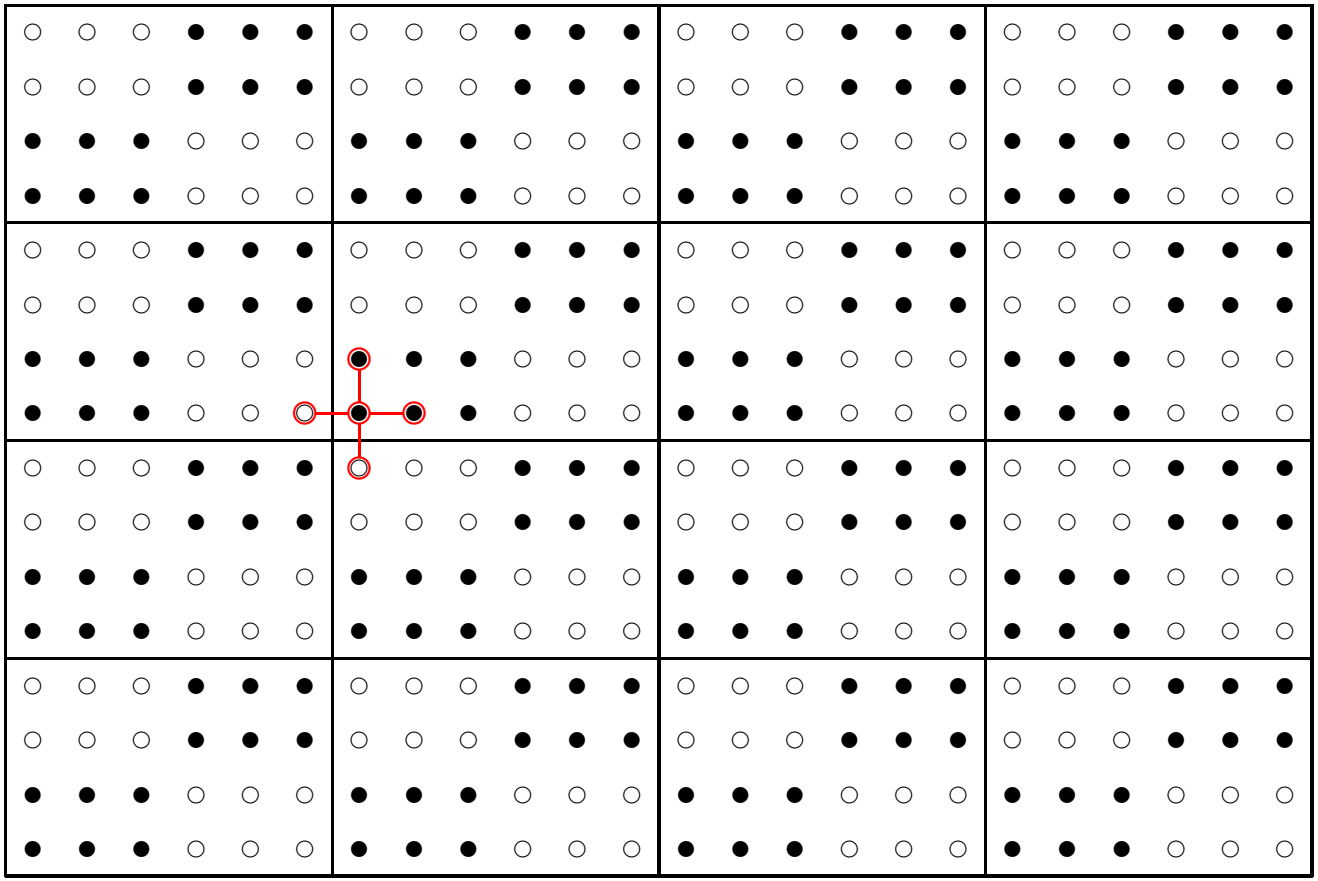


- The outer loop is parallelisable (blocks of the same colour process different data.)
- The results does depend on the number of processes used.

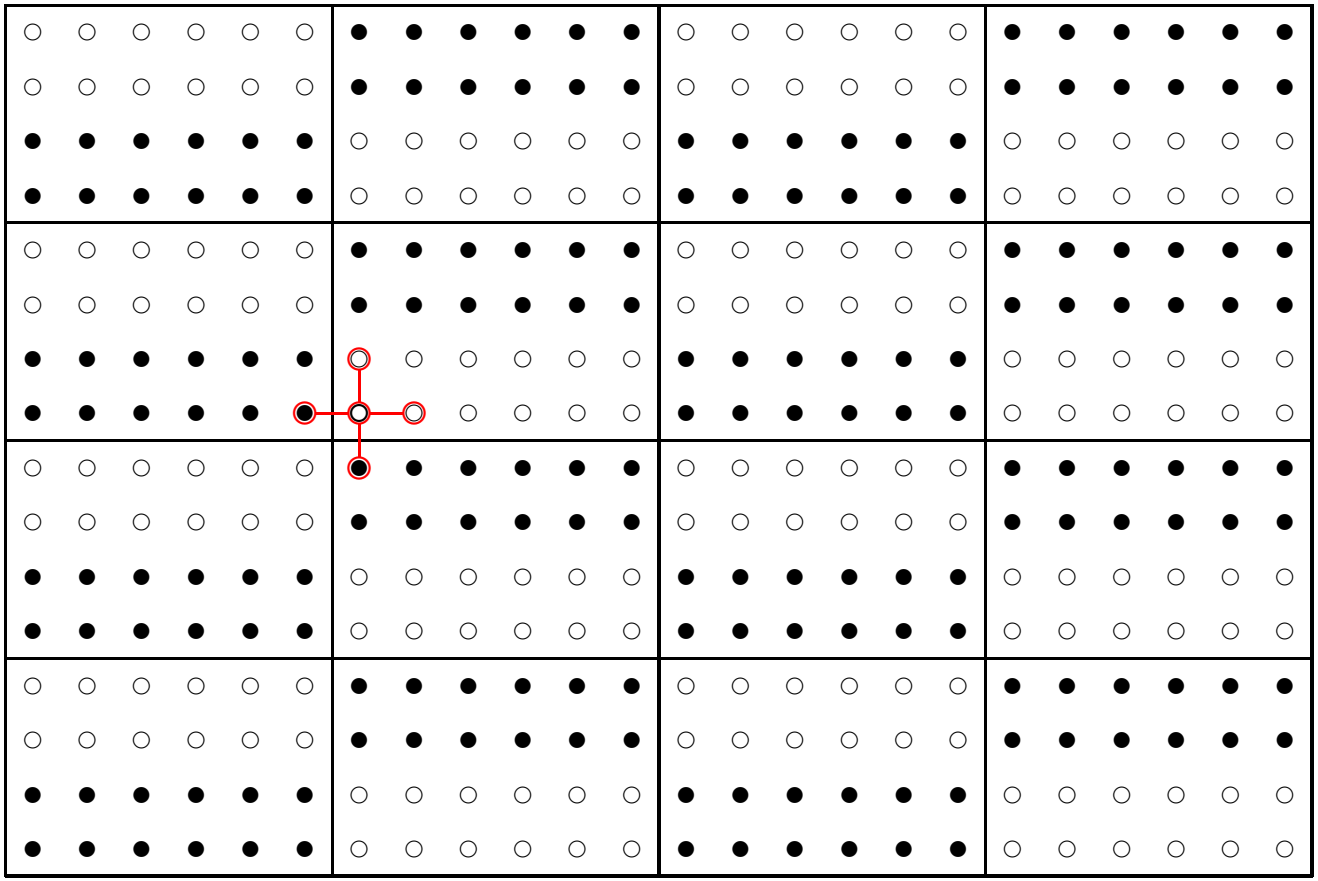
Chessboard decomposition (fine grained)



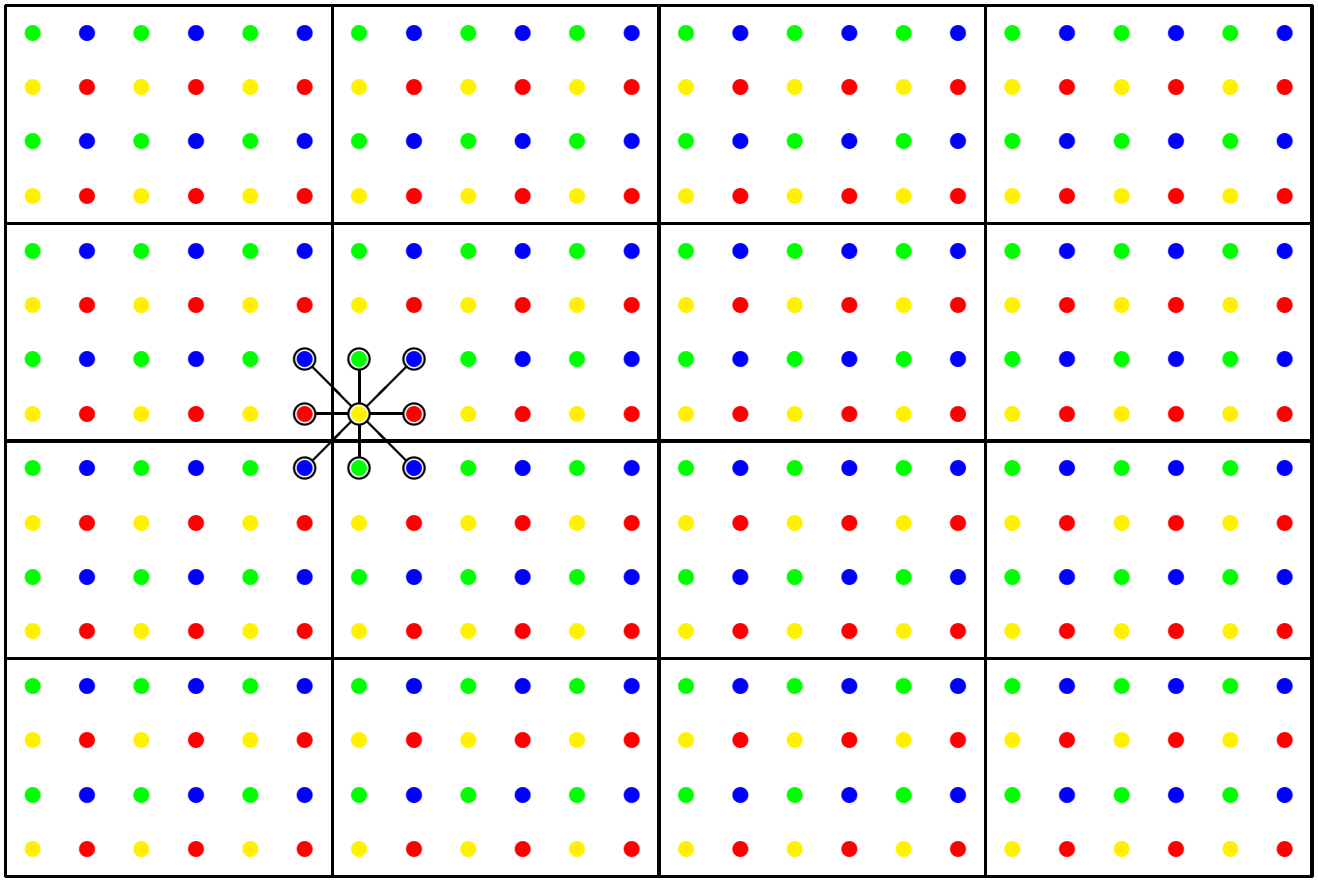
Chessboard decomposition (coarse grained I)



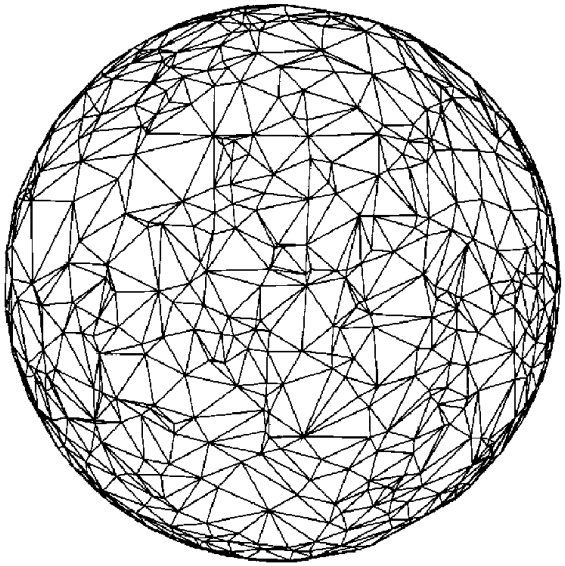
Chessboard decomposition (coarse grained II)



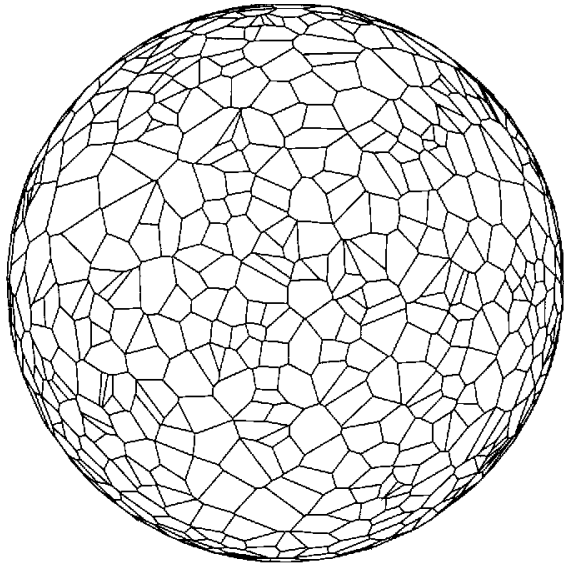
Decomposition into four independent parts



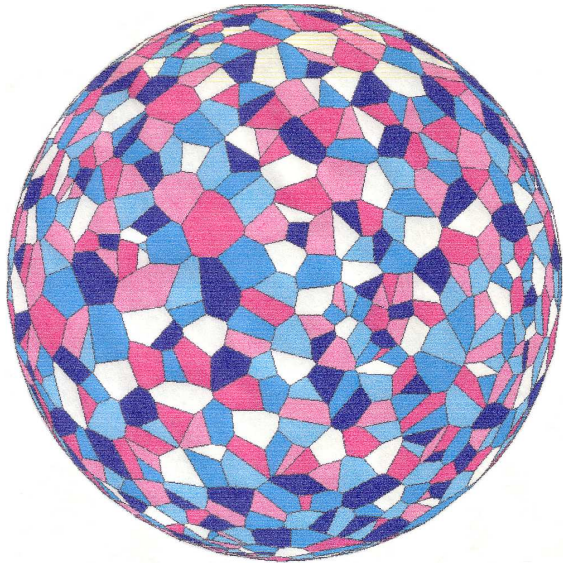
Colouring of an irregular lattice



random lattice



dual lattice



coloured lattice