



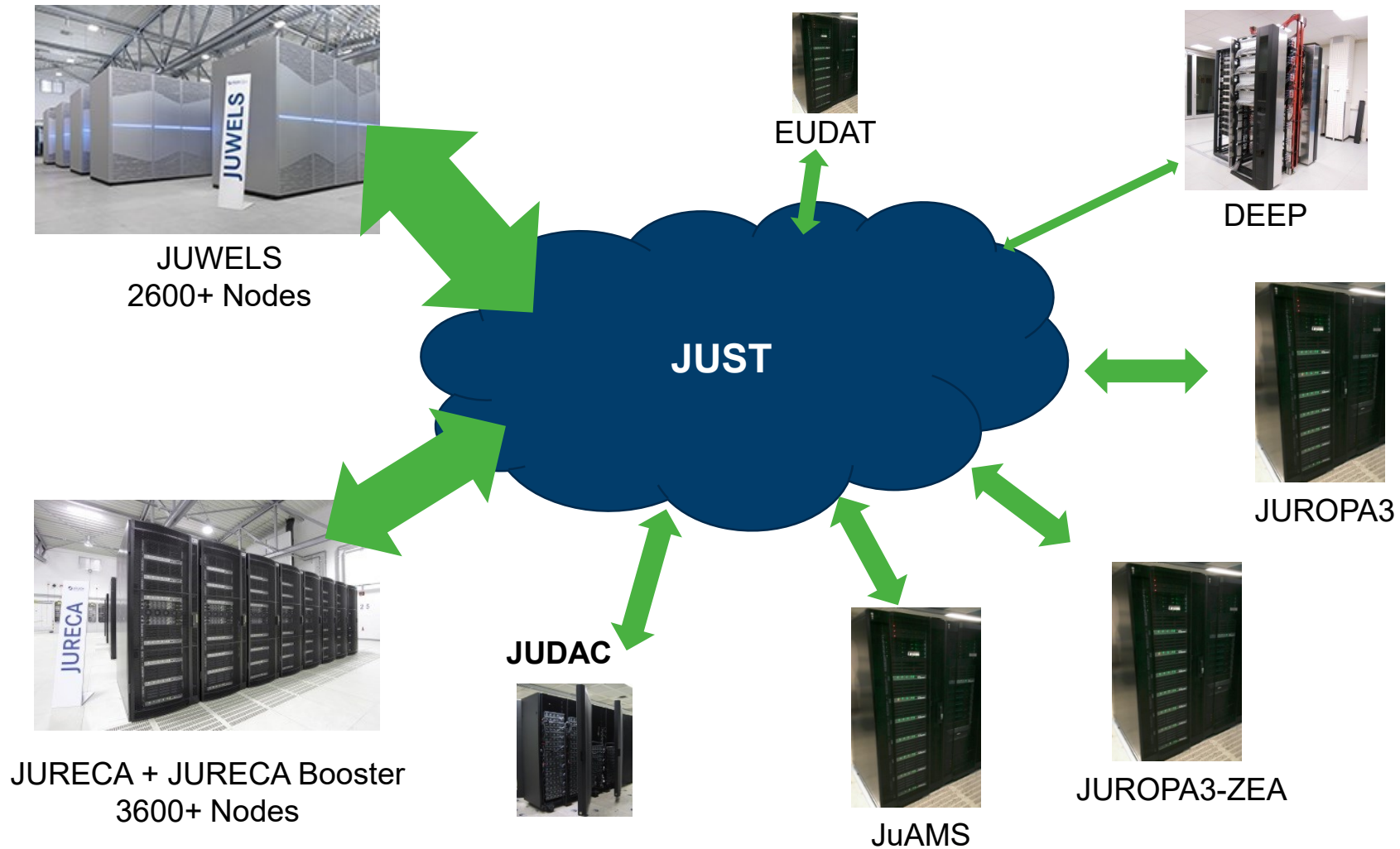
PARALLEL I/O AND PORTABLE DATA FORMATS

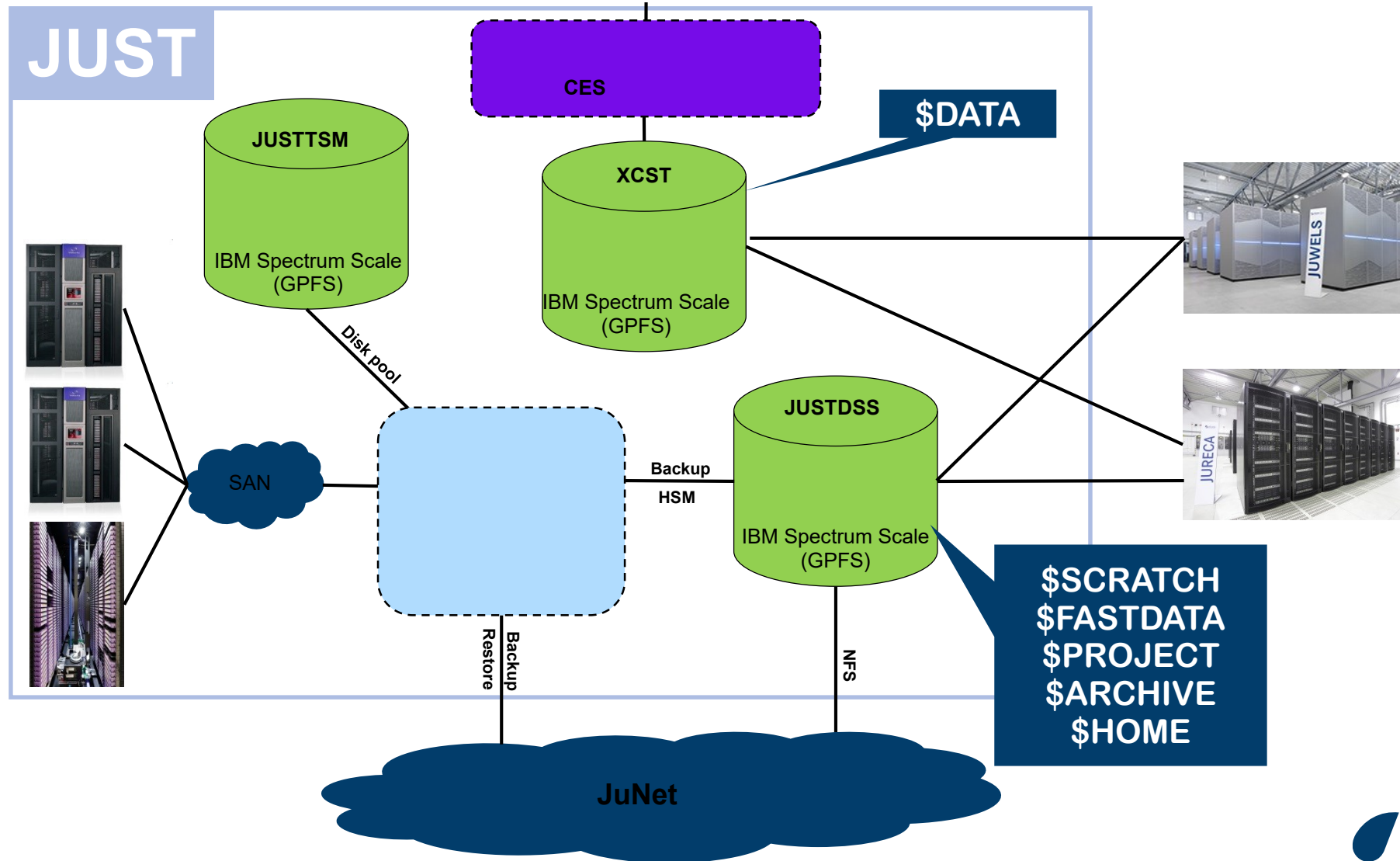
INTRODUCTION AND PARALLEL I/O STRATEGIES

27.01.2020 | SEBASTIAN LÜHRS (S.LUEHRS@FZ-JUELICH.DE)

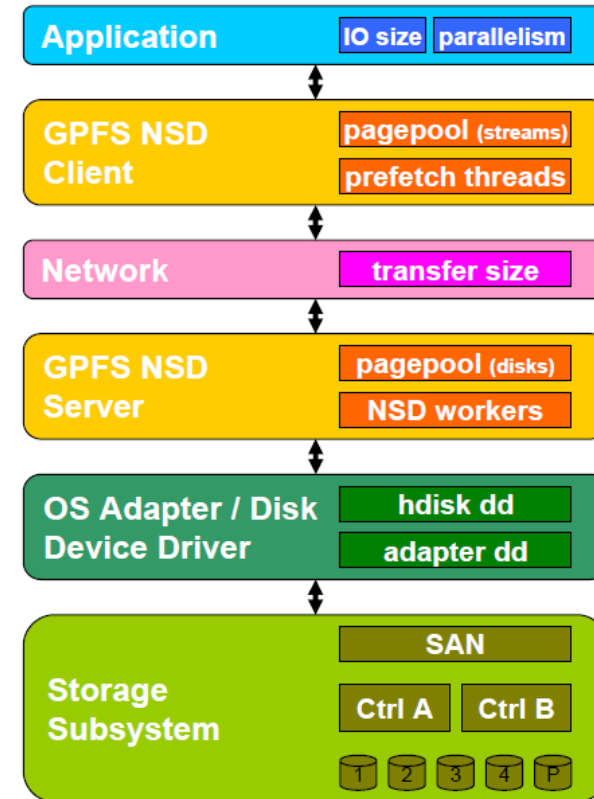
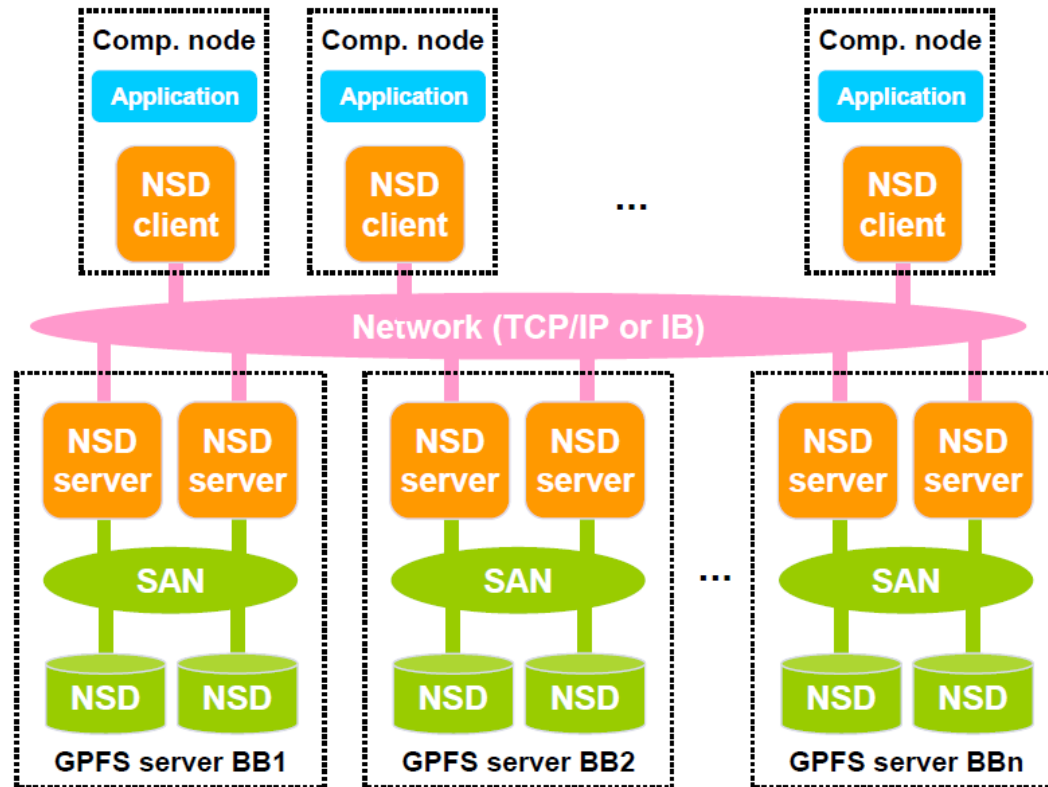
JUST STORAGE SYSTEM

JUelich STorage





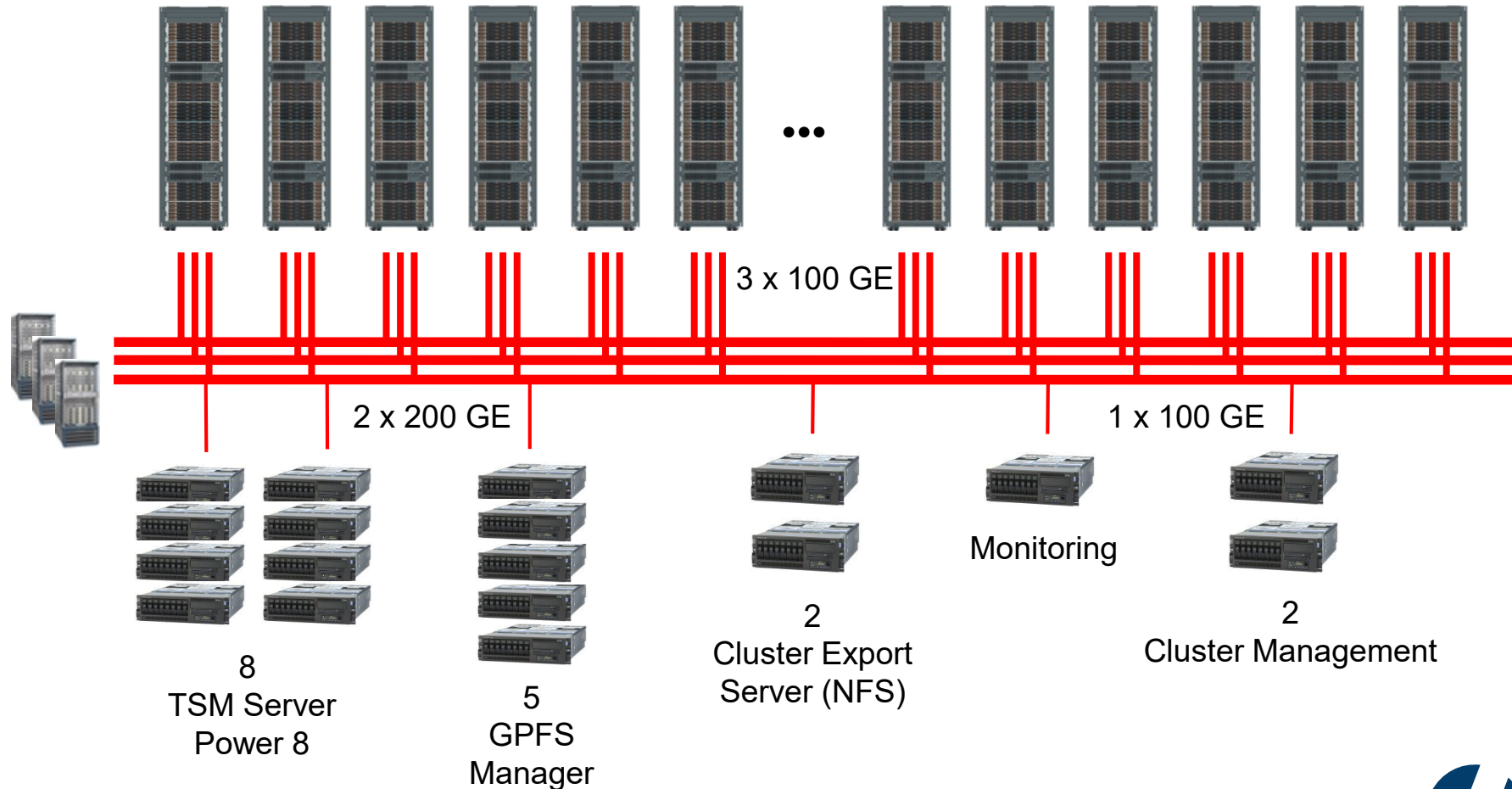
File I/O to GPFS



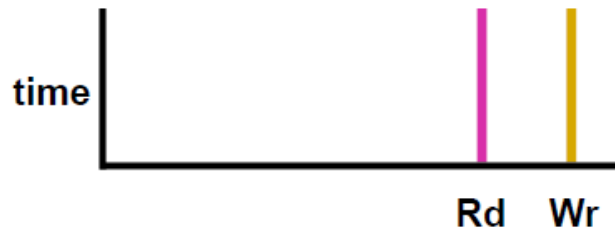
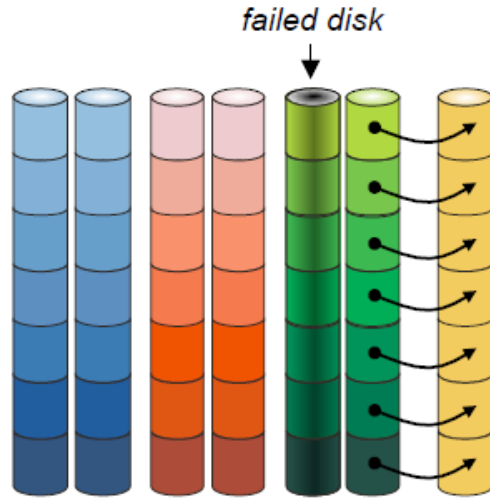
JUST – 5th generation



21 x DSS240 + 1 x DSS260 → 44 x NSD Server, 90 x Enclosure → +7.500 10TB disks

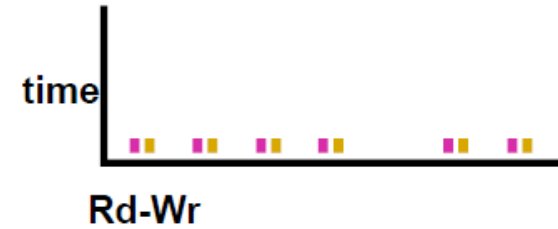
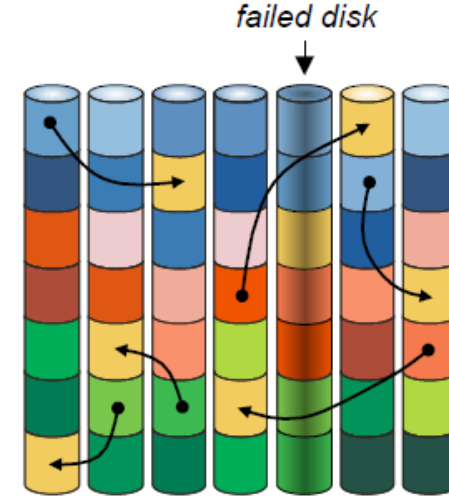


Declustered RAID



Disk failure causes disk rebuild

- Volume degraded for a long time
- performance impact for file system



Disk failure causes strips rebuild

- all disc involved
- Volume degraded for a short time
- minimized performance impact

JUST – Characteristics

Spectrum Scale (GPFS 5.0.1) + GNR (GPFS Native RAID)

- Declustered RAID technology
- End-to-End data integrity

Spectrum Protect (TSM) for Backup & HSM

Hardware:

- x86 based server + RHEL 7
- IBM Power 8 + AIX 7.2
- 100GE network fabric

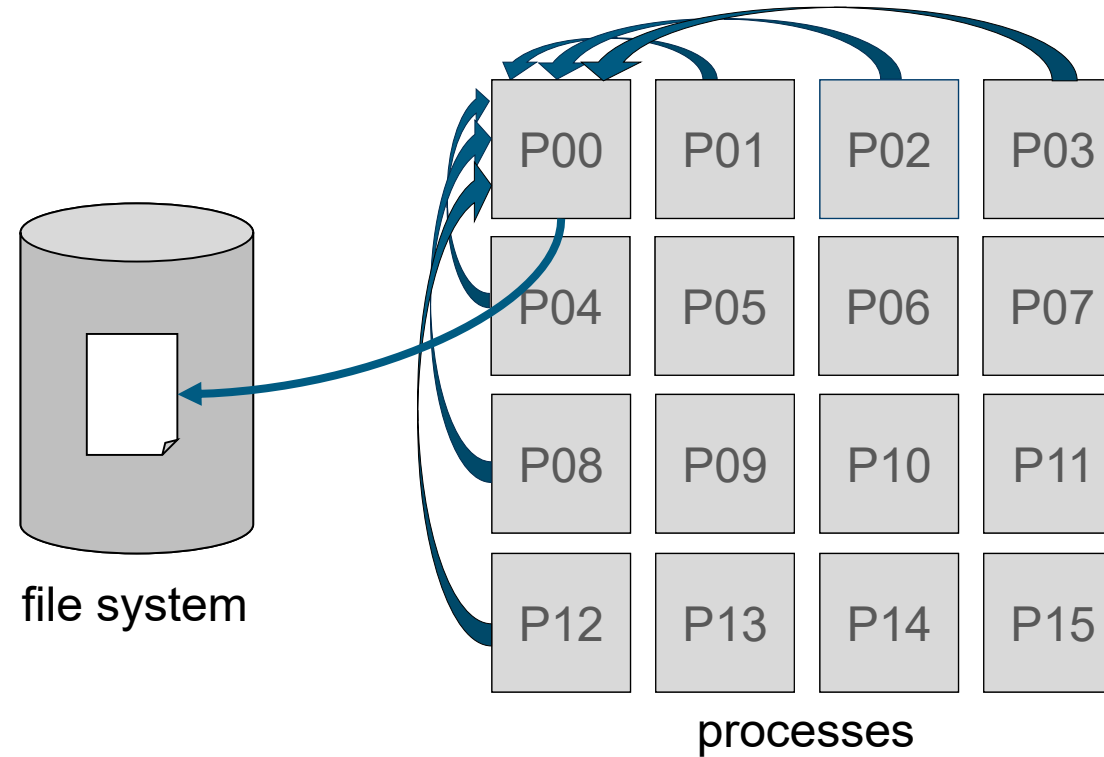
75 PB gross capacity

Bandwidth: **400 GB/s**

PARALLEL I/O STRATEGIES

Parallel I/O Strategies

One process performs I/O



Parallel I/O Strategies

One process performs I/O

- + Simple to implement
- I/O bandwidth is limited to the rate of this single process
- Additional communication might be necessary
- Other processes may idle and waste computing resources during I/O time

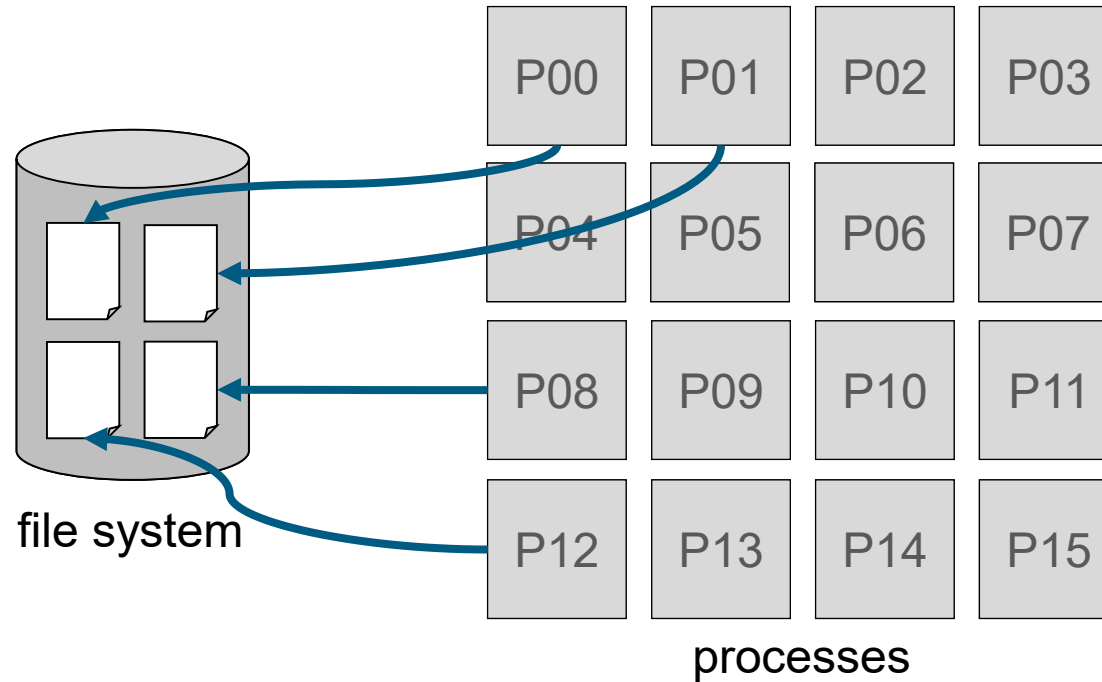
Parallel I/O Pitfalls

Frequent flushing on small blocks

- Modern file systems in HPC have **large file system blocks** (e.g. 4MB)
- A flush on a file handle forces the file system to perform all pending write operations
- If application writes in small data blocks, the same file system block it has to be **read and written multiple times**
- Performance degradation due to the inability to combine several write calls

Parallel I/O Strategies

Task-local files



Parallel I/O Strategies

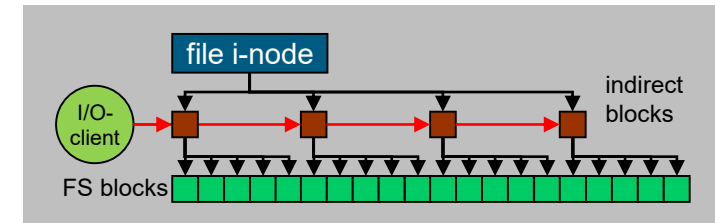
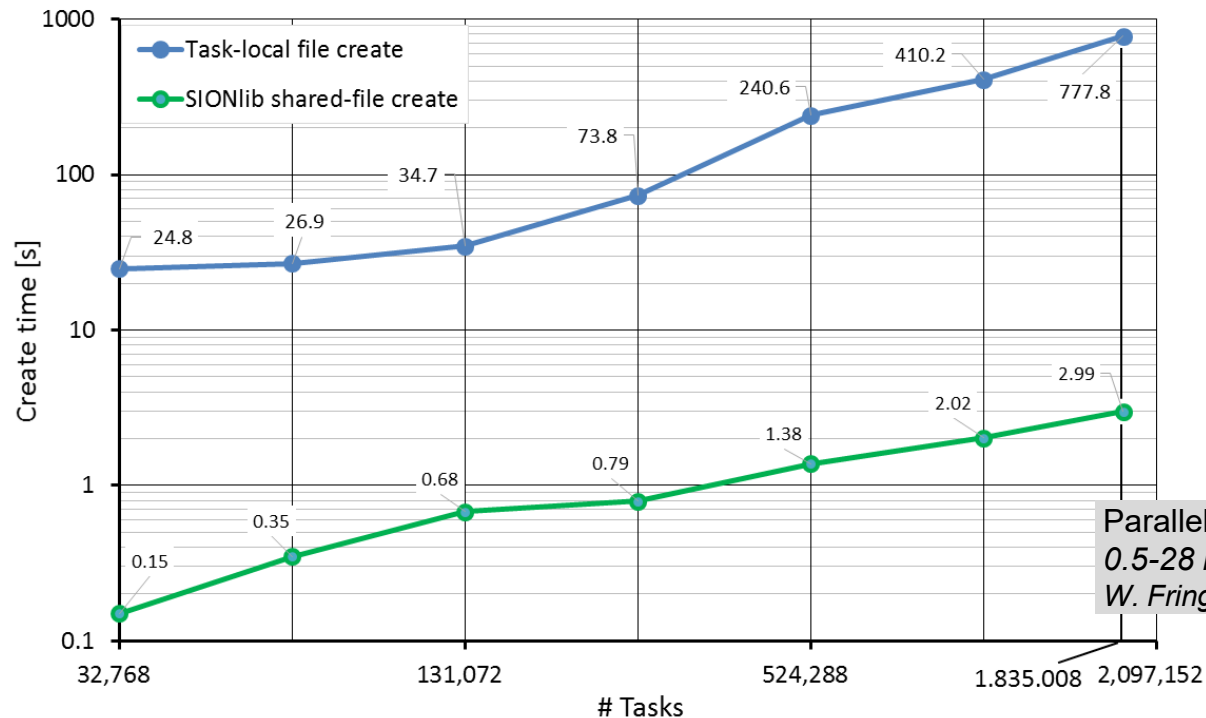
Task-local files

- + Simple to implement
- + No coordination between processes needed
- + No false sharing of file system blocks
- Number of files quickly becomes unmanageable
- Files often need to be merged to create a canonical dataset
- File system might serialize meta data modification

Parallel I/O Pitfalls

Serialization of meta data modification

Example: Creating files in parallel in the same directory



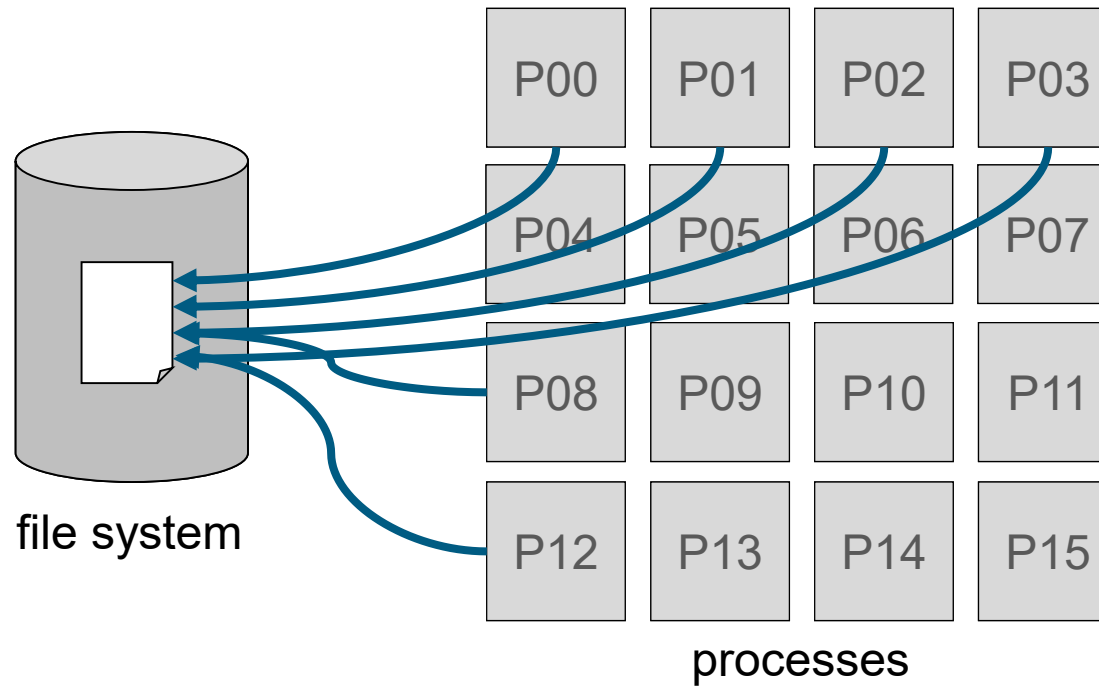
- Meta-data wall on file level
 - File changes by multiple processes can cause serialization
 - File meta-data management
 - Locking

Parallel file creation on JUQUEEN
0.5-28 racks, 64 tasks/node
W. Frings

The creation of 2.097.152 files costs 113.595 core hours on JUQUEEN!

Parallel I/O Strategies

Shared files



Parallel I/O Strategies

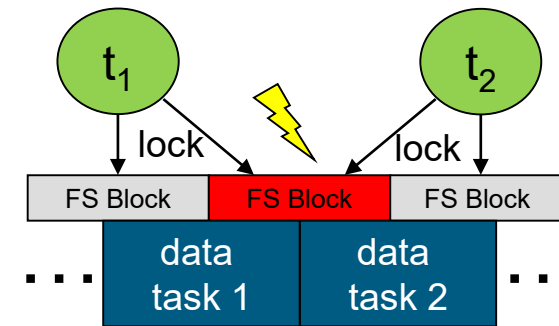
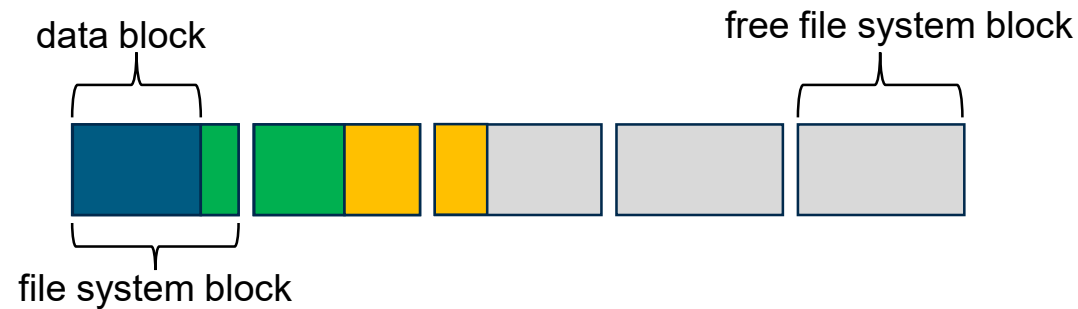
Shared files

- + Number of files is independent of number of processes
- + File can be in canonical representation (no post-processing)
- Uncoordinated client requests might induce time penalties
- File layout may induce false sharing of file system blocks

Parallel I/O Pitfalls

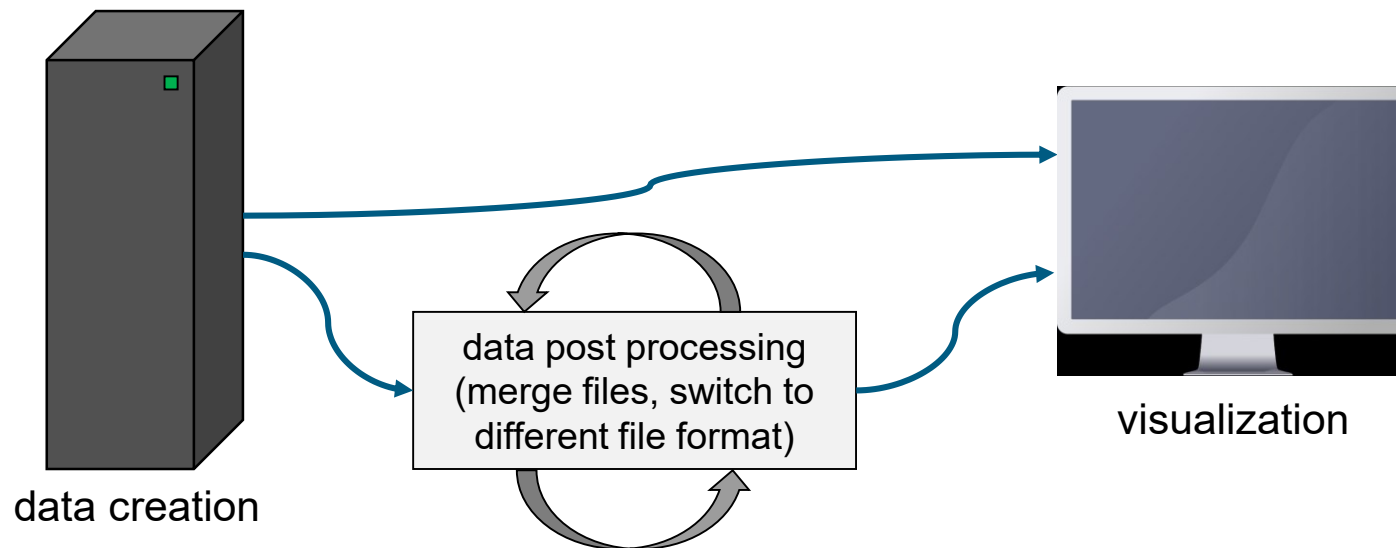
False sharing of file system blocks

- Data blocks of individual processes do not fill up a complete file system block
- Several processes share a file system block
- Exclusive access (e.g. write) must be serialized
- The more processes have to synchronize the more waiting time will propagate



I/O Workflow

- Post processing can be very time-consuming ($>$ data creation)
 - Widely used portable data formats avoid post processing
- Data transportation time can be long:
 - Use shared file system for file access, avoid raw data transport
 - Avoid renaming/moving of big files (can block backup)



Parallel I/O Pitfalls

Portability

- Endianness (byte order) of binary data
- Conversion of files might be necessary and expensive

2,712,847,316

=

10100001 10110010 11000011 11010100

Address	Little Endian	Big Endian
1000	11010100	10100001
1001	11000011	10110010
1002	10110010	11000011
1003	10100001	11010100

Parallel I/O Pitfalls

Portability

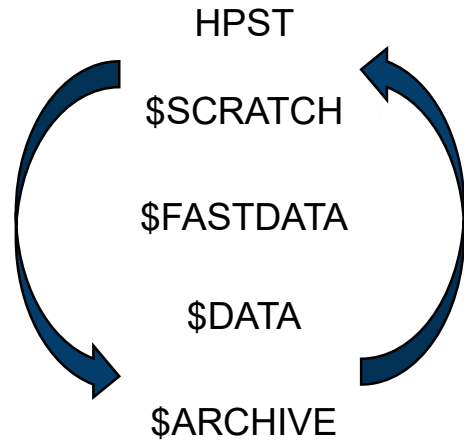
- Memory order depends on programming language
- Transpose of array might be necessary when using different programming languages in the same workflow
- Solution: Choosing a portable data format (HDF5, NetCDF)

	Address	row-major order (e.g. C/C++)	column-major order (e.g. Fortran)
1	1000	1	1
2	1001	2	4
3	1002	3	7
4	1003	4	2
5	1004	5	5
6	...	...	...
7			
8			
9			

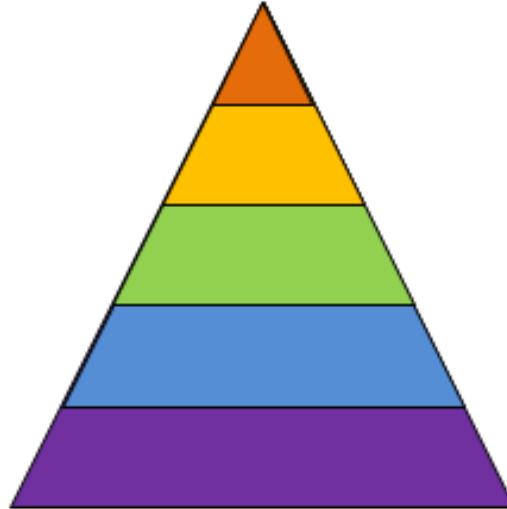
Storage Tiers

Different storage tiers with different optimization targets

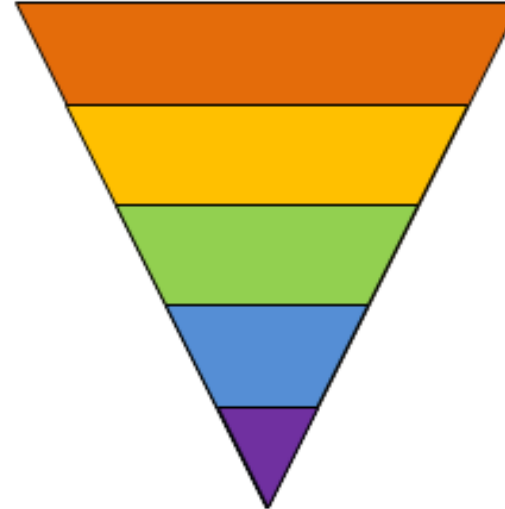
Data staging at JSC



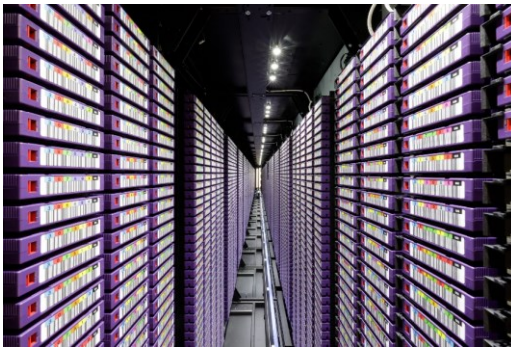
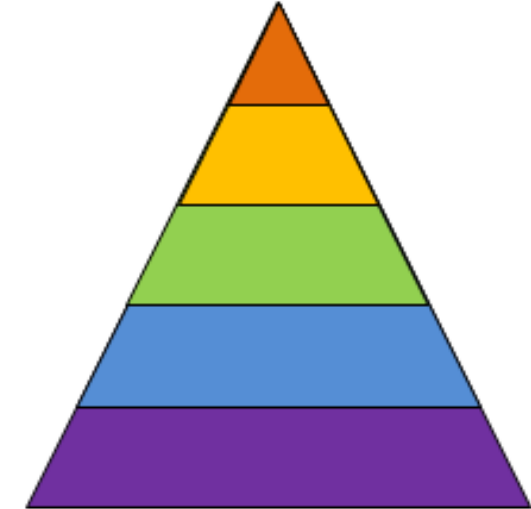
Capacity



Bandwidth



Retention time



Tape Library

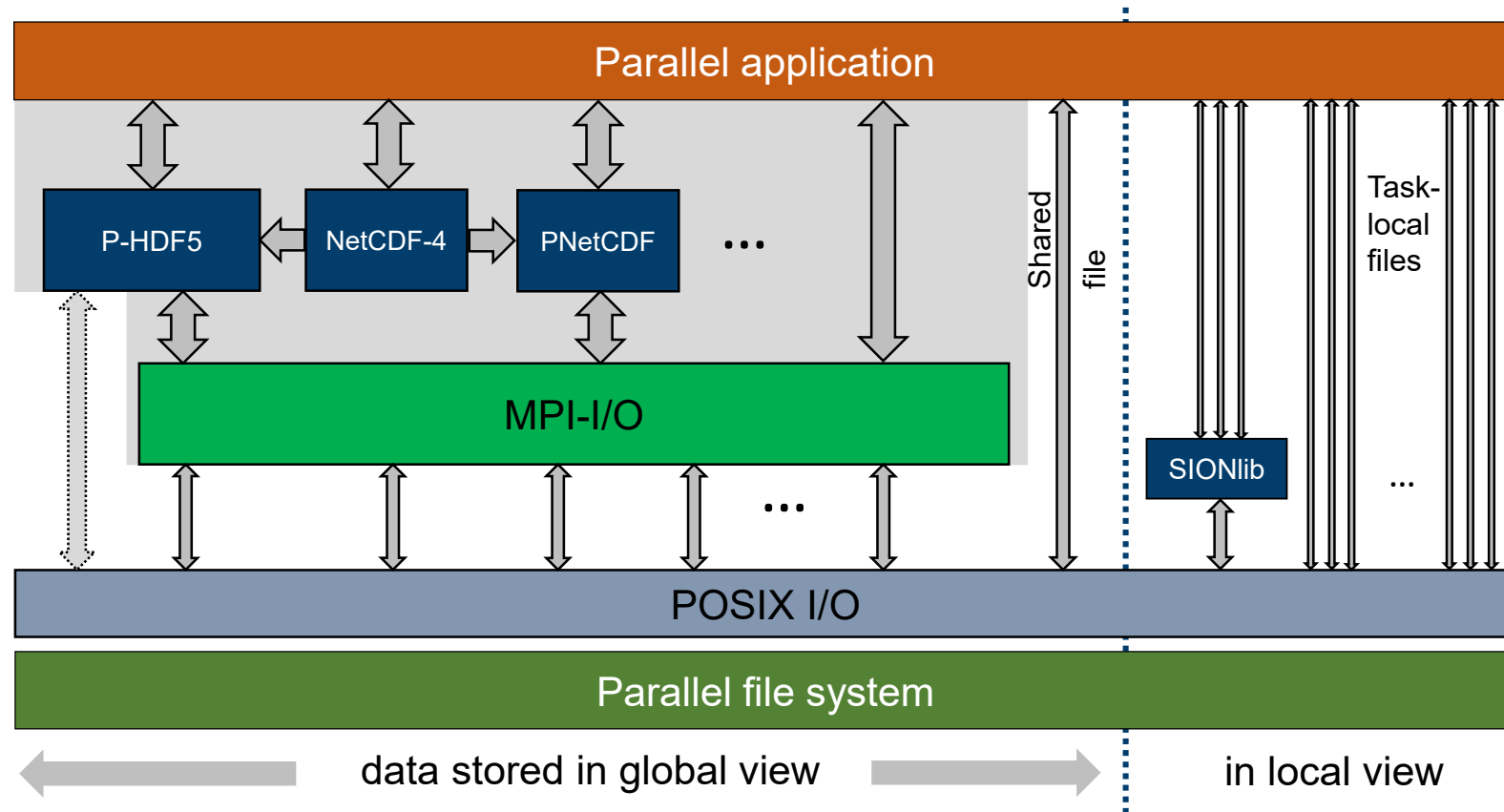


JUST 5

How to choose the I/O strategy?

- Performance considerations
 - Amount of data
 - Frequency of reading/writing
 - Scalability
- Portability
 - Different HPC architectures
 - Data exchange with others
 - Long-term storage
- E.g. use two formats and converters:
 - Internal: Write/read data “as-is”
 - External: Write/read data in non-decomposed format (portable, system-independent, self-describing)

Parallel I/O Software Stack



HANDS ON PREPARATION

HPC access

JUDOOR account and part of the training2000 project?

- Register and join the course project: <https://judoor.fz-juelich.de/projects/join/training2000>

Available SSH-Key?

- Create a new public/private Key-pair: `ssh-keygen` or use puttygen
- Add the private key into your agent: `ssh-add <private_key>`

Public key on the system?

- Upload your public key via <https://judoor.fz-juelich.de> to the system

Login possible?

- Try to login: `ssh <userid>@jureca.fz-juelich.de`

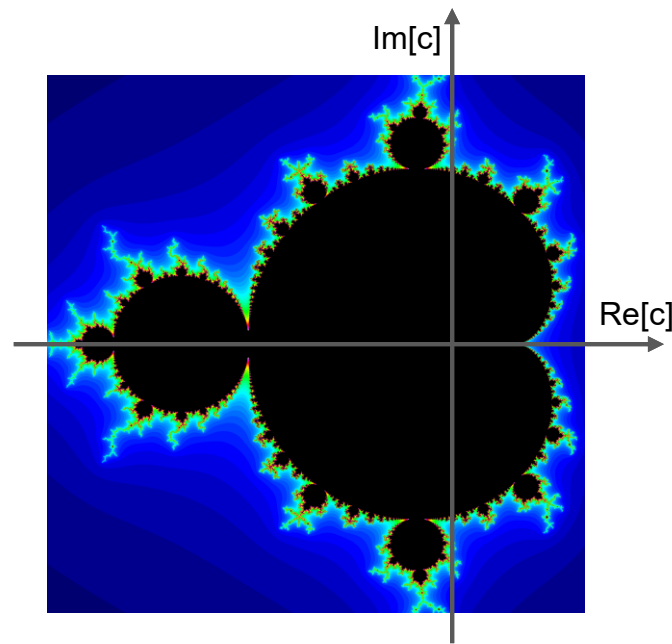
Course exercise: Mandelbrot set

- Set of all complex numbers c in the complex plane for which

$$z_{n+1} = z_n^2 + c$$

$$z_0 = 0$$

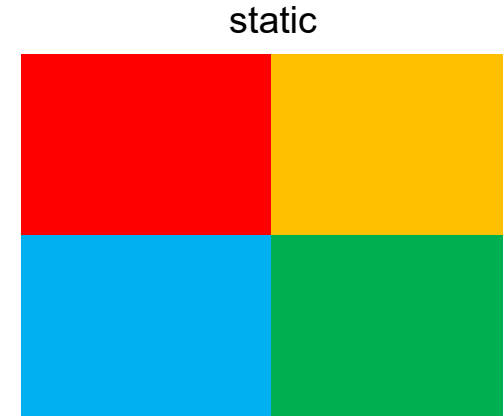
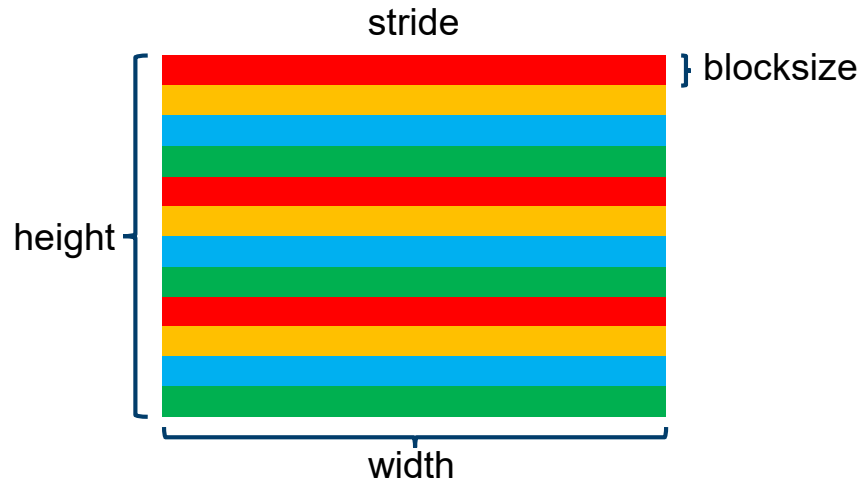
does not approach infinity



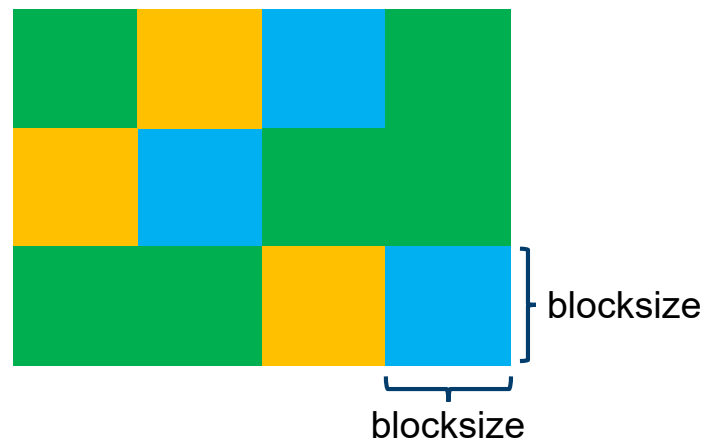
Course exercise: Mandelbrot set

- I/O comparison example
- Four different decomposition types
 - stride
 - static
 - master-worker (workers write)
 - master-worker (master writes)
- Five different output formats
 - SIONlib
 - HDF5
 - MPI-IO
 - parallel-netcdf
 - netcdf4

Decomposition types



master-worker, workers write



master-worker, master writes



mandelmpi

Command line options

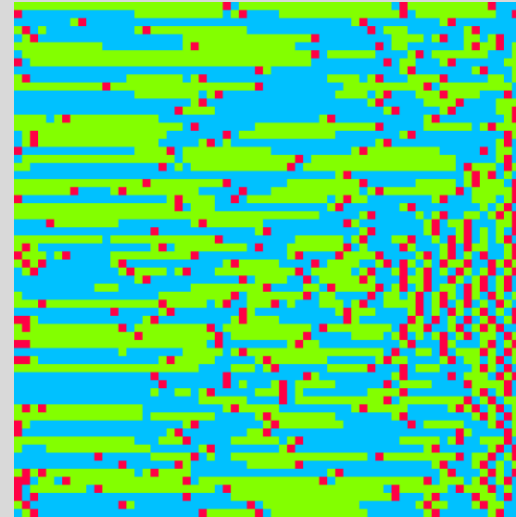
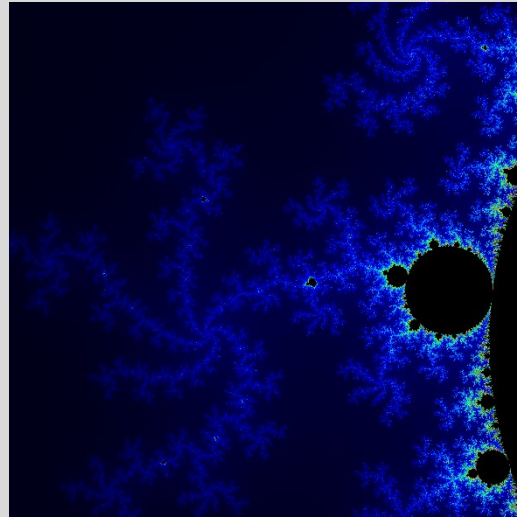
- v use verbose mode
- t decomposition type (0: stride, 1: static, 2: master-worker worker write, 3: master-worker master write), default: 0
- w width, default: 256
- h height, default: 256
- b blocksize (not used for type = 1), default: 64
- p number of procs in x-direction (only used for type = 1)
- q number of procs in y-direction (only used for type = 1)
- x coordinates of initial area: x1 x2 y1 y2,
default: -1.5 0.5 -1.0 1.0
- i max. iterations, default 256
- f output type (0: SIONlib, 1: HDF5, 2: MPI-IO, 3: pnetcdf, 4: netcdf4),
default: 0

mandelseq

Command line options

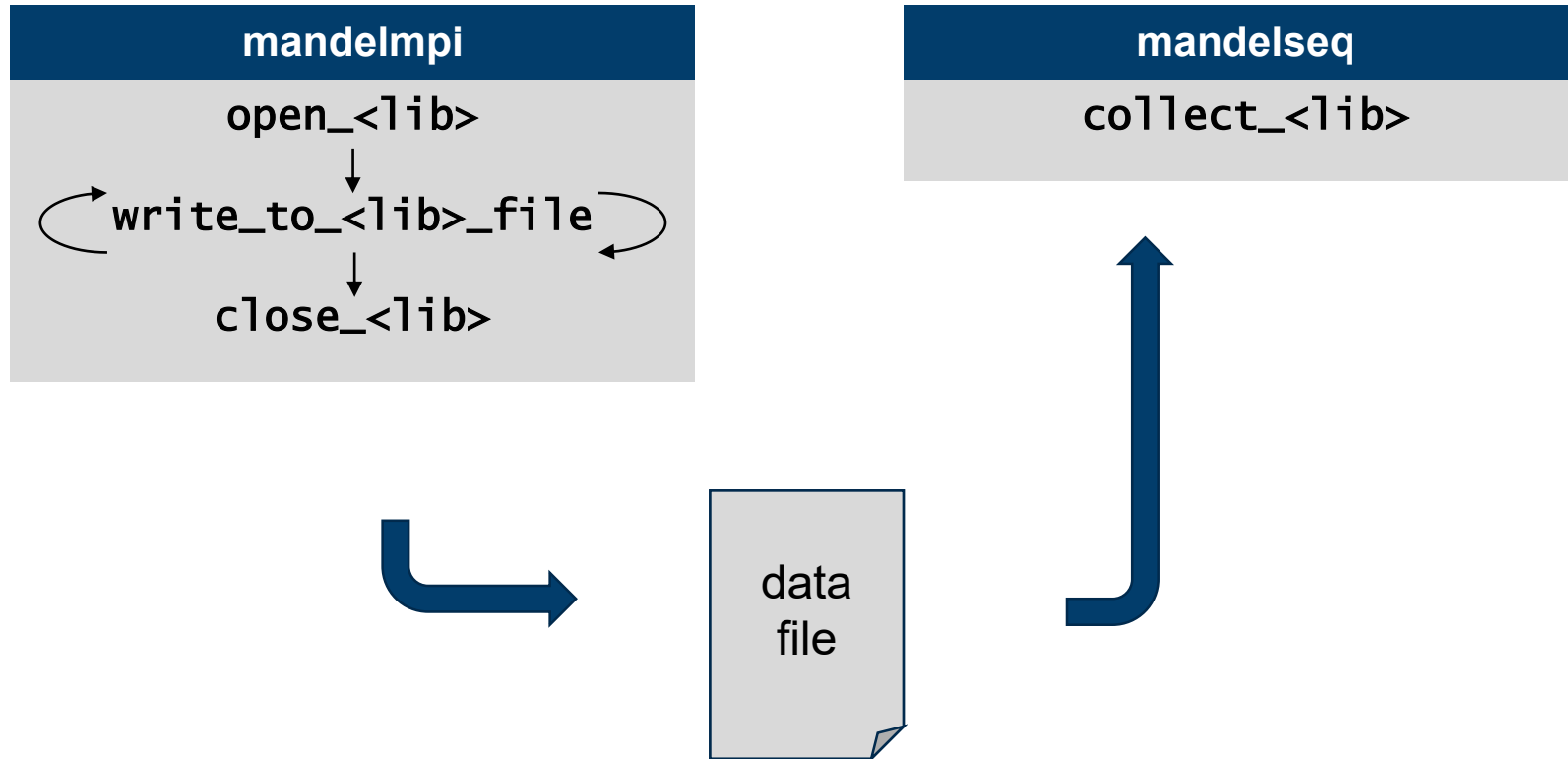
-f output type (0: SIONlib, 1: HDF5, 2: MPI-IO, 3: pnetcdf, 4: netcdf4),
default: 0

Output



process distribution image
only available for SIONlib

Mandelbrot exercise workflow



Mandelbrot exercise workflow

1. Load modules
`. load_modules_jureca.sh`
2. Run compilation
`make`
3. Change runtime parameter in "run.job" file or use `srun`
4. Submit a job if not using `srun` directly
`sbatch run.job`
5. Create result image
`./mandelseq -f <format>`
6. View image (**not** in interactive session)
`display mandelcol.ppm`

Mandelbrot exercise API

C

```
typedef struct _infostruct
{
    int type; int width; int height;
    int numprocs;
    double xmin; double xmax; double ymin; double ymax;
    int maxiter;
} _infostruct;
```

Fortran

```
type :: t_infostruct
    integer :: type, width, height
    integer :: numprocs
    real :: xmin, xmax, ymin, ymax
    integer :: maxiter
end type t_infostruct
```

Mandelbrot exercise API

C

```
void open_<lib>(<type> *fid, _infostruct *infostruct,  
               int *blocksize, int *start, int rank)
```

Fortran

```
open_<lib>(fid, info, blocksize, start, rank)  
<type>, intent(out) :: fid  
type(t_infostruct), intent(in) :: info  
integer, dimension(2), intent(in) :: blocksize  
integer, dimension(2), intent(in) :: start  
integer, intent(in) :: rank
```

fid	lib specific file_id (can occurs twice if multiple ids needed)
info	global information structure
blocksize	chosen (or calculated) blocksizes (C: [y,x], Fortran: [x,y])
start	calculated start point (C: [y,x], Fortran: [x,y], starting at 0)
rank	process MPI rank

Mandelbrot exercise API

C

```
void close_<lib>(<type> *fid, _infostruct *infostruct,  
                int rank)
```

Fortran

```
close_<lib>(fid, info, rank)  
<type>, intent(inout) :: fid  
type(t_infostruct), intent(in) :: info  
integer, intent(in) :: rank
```

fid	lib specific file_id (can occurs twice if multiple ids needed)
info	global information structure
rank	process MPI rank

Mandelbrot exercise API

C

```
void write_to_<lib>_file(  
    <type> *fid, _infostruct *infostruct, int *iterations,  
    int width, int height, int xpos, int ypos)
```

Fortran

```
write_to_<lib>_file(fid, info, iterations, width, height,  
                    xpos, ypos)  
<type>, intent(in) :: fid  
type(t_infostruct), intent(in) :: info  
integer, dimension(:), intent(in) :: iterations  
integer, intent(in) :: width  
integer, intent(in) :: height  
integer, intent(in) :: xpos  
integer, intent(in) :: ypos
```

iterations	data array
width, height	size of current data block (pixel coordinates)
xpos, ypos	position of current data block (pixel coordinates starting at 0)

Mandelbrot exercise API

C

```
void collect_<lib>(  
    int **iterations, int **proc_distribution,  
    _infostruct *infostruct)
```

Fortran

```
collect_<lib>(iterations, proc_distribution, info)  
integer, dimension(:), pointer :: iterations  
integer, dimension(:), pointer :: proc_distribution  
type(t_infostruct), intent(inout) :: info
```

iterations	data array
proc_distribution	process distribution array (only in Sionlib)
info	global information structure