

Parallel I/O and Portable Data Formats I/O strategies

Sebastian Lührs

s.luehrs@fz-juelich.de

Jülich Supercomputing Centre

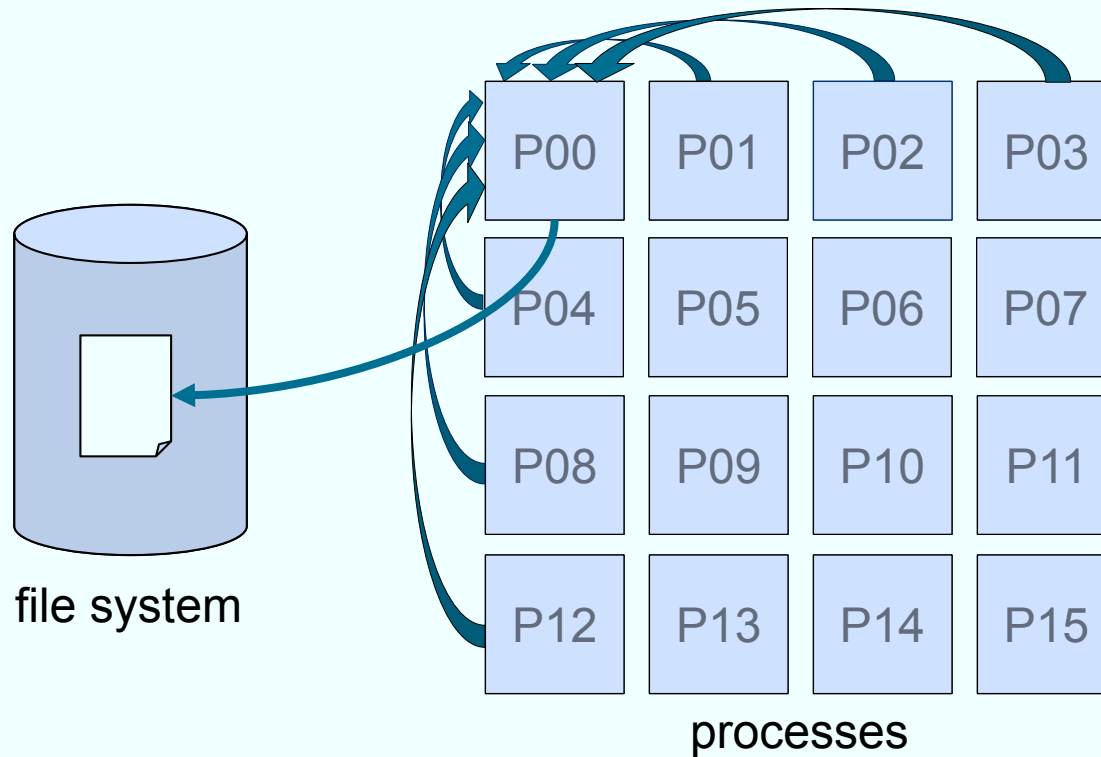
Forschungszentrum Jülich GmbH

Jülich, March 13th, 2017

Outline

- Common I/O strategies
 - One process performs I/O
 - Task-local files
 - Shared files
- I/O workflow
- Pitfalls
- Parallel I/O software stack
- Course exercise description
 - General exercise workflow
 - Mandelbrot set description
 - Exercise API

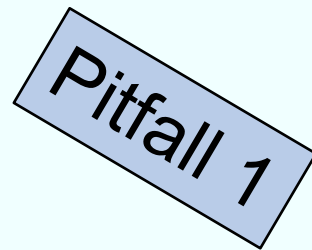
One process performs I/O



One process performs I/O

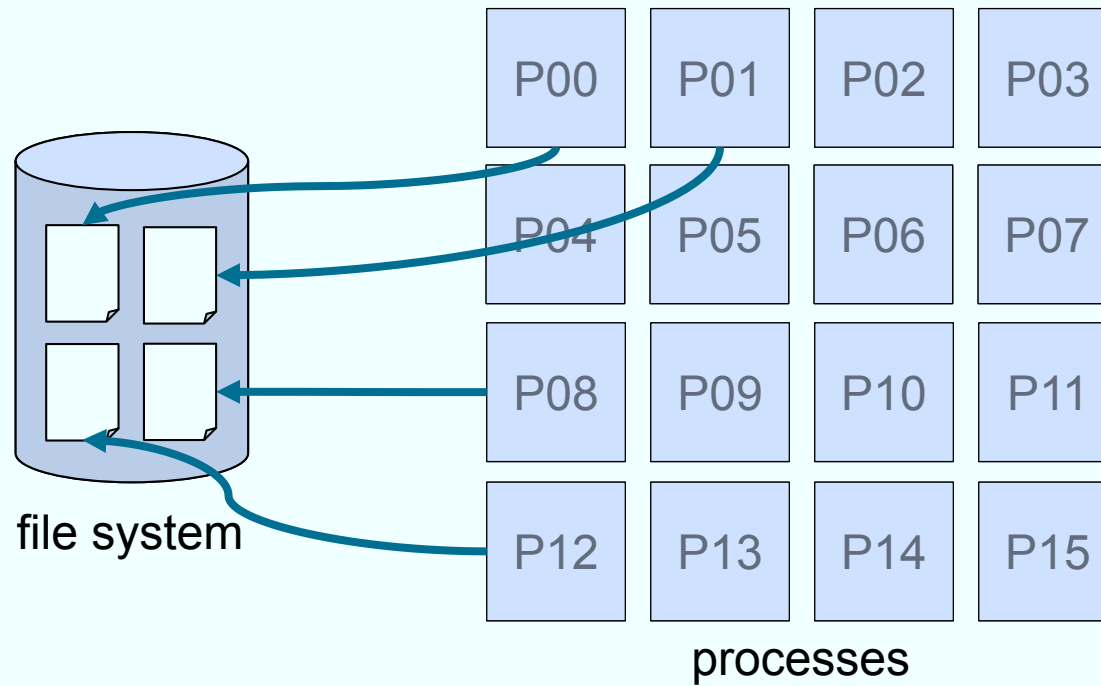
- + Simple to implement
- I/O bandwidth is limited to the rate of this single process
- Additional communication might be necessary
- Other processes may idle and waste computing resources during I/O time

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

Task-local files



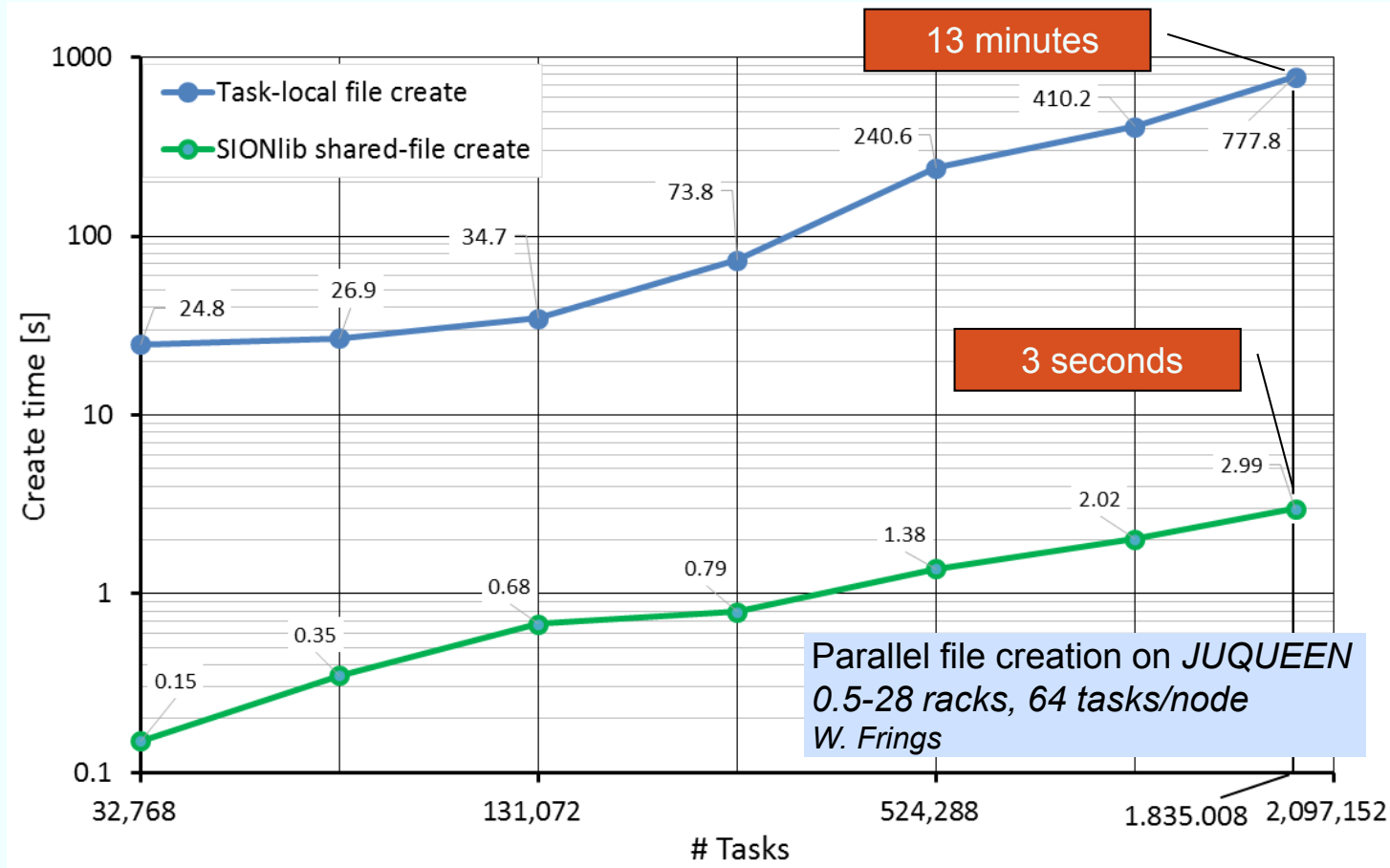
Task-local files

- + 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

Serialization of meta data modification

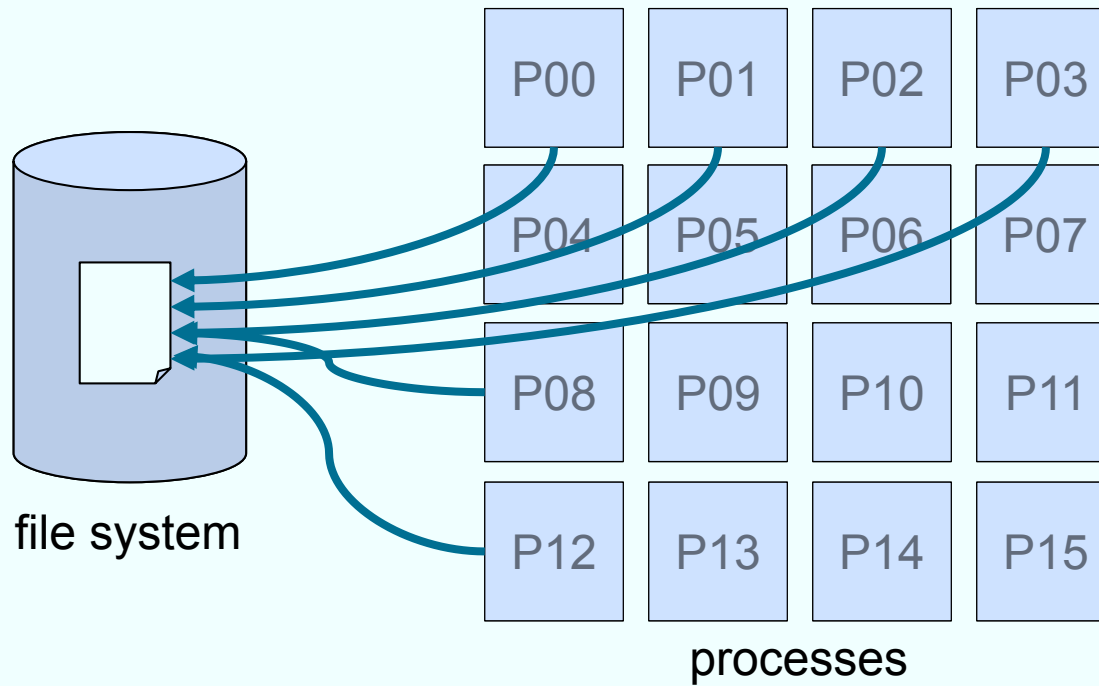
Pitfall 2

Example: Creating files in parallel in the same directory



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

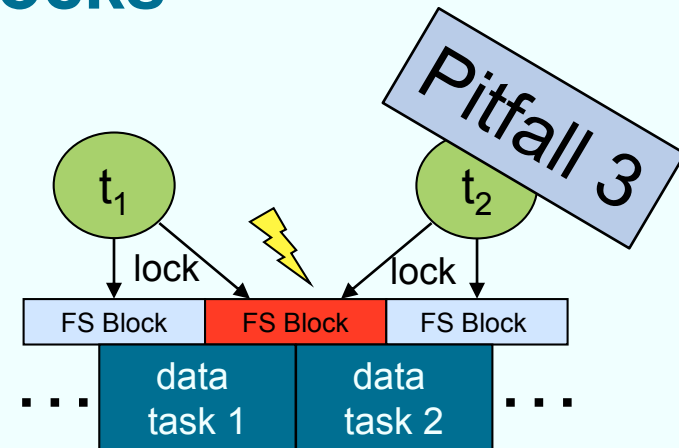
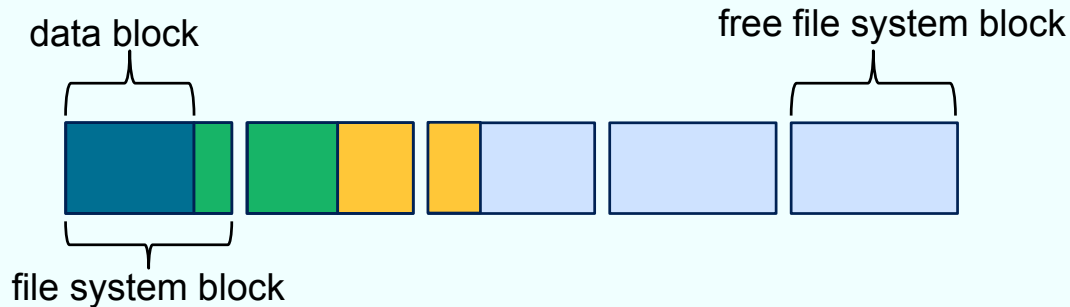
Shared files



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

False sharing of file system blocks



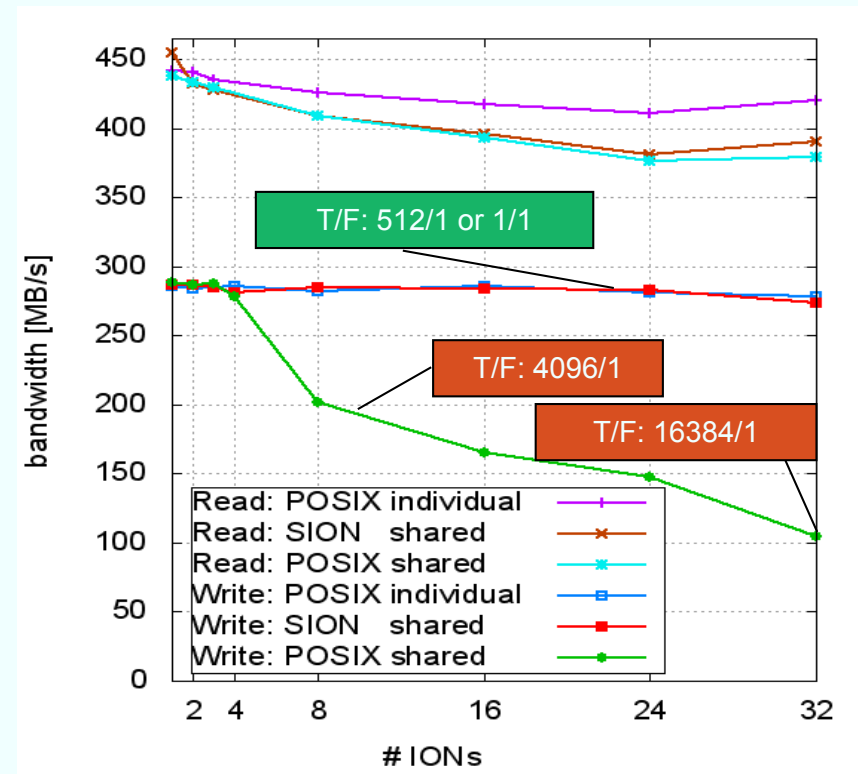
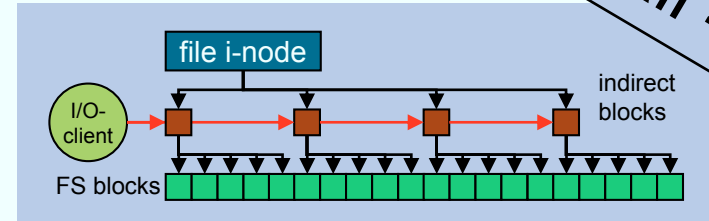
- Data blocks of individual processes **do not fill up a complete file system block**
- Several processes **share a file system block**
- Exclusive access (e.g. write) must be **serialized**
- The more processes have to synchronize the more waiting time will propagate

Number of Tasks per Shared File

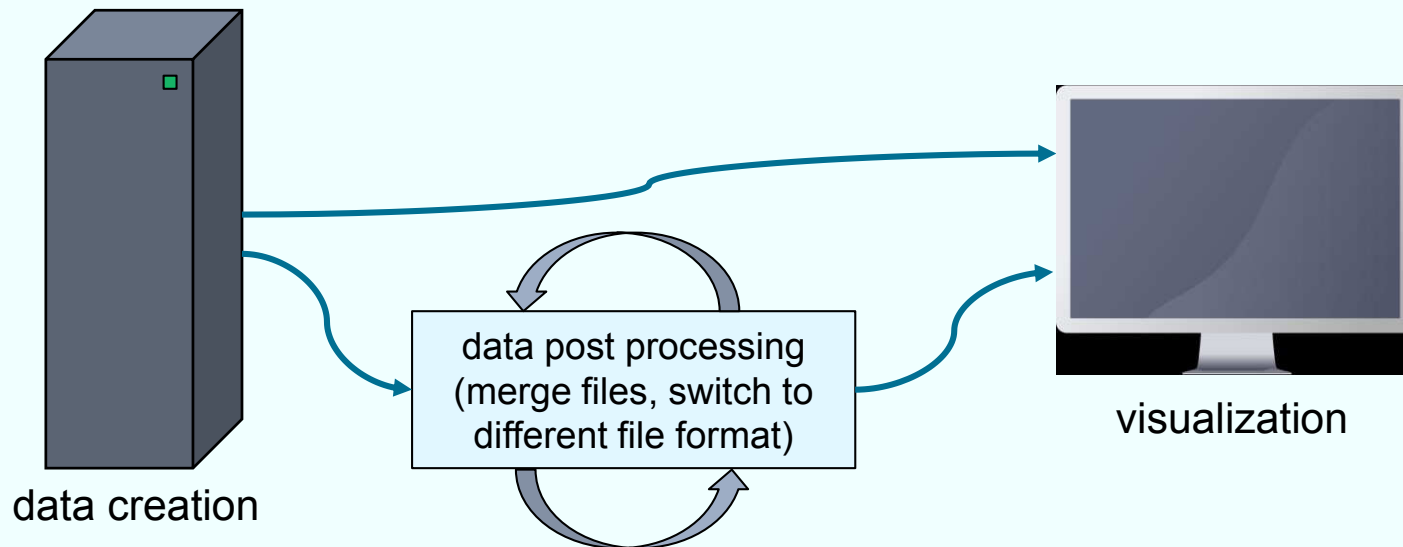
Pitfall 4

- Meta-data wall on file level
 - File meta-data management
 - Locking

- Example Blue Gene/P
 - Jugene (72 racks)
 - I/O forwarding nodes (ION)
 - GPFS client on ION
 - One file per ION



I/O Workflow

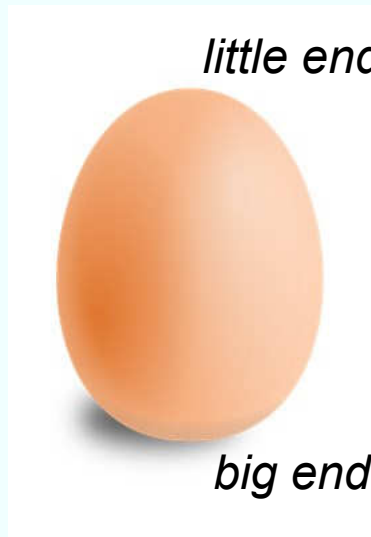


- Post processing can be very time-consuming (> data creation)
 - Widely used portable data formats avoid post processing
- Data transportation time can be long:
 - Use shared file system for file access, avoid raw data transport
 - Avoid renaming/moving of big files (can block backup)

Portability

Pitfall 5

- Endianness (byte order) of binary data



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

- Conversion of files might be necessary and expensive

Portability

Pitfall 6

- Memory order depends on programming language

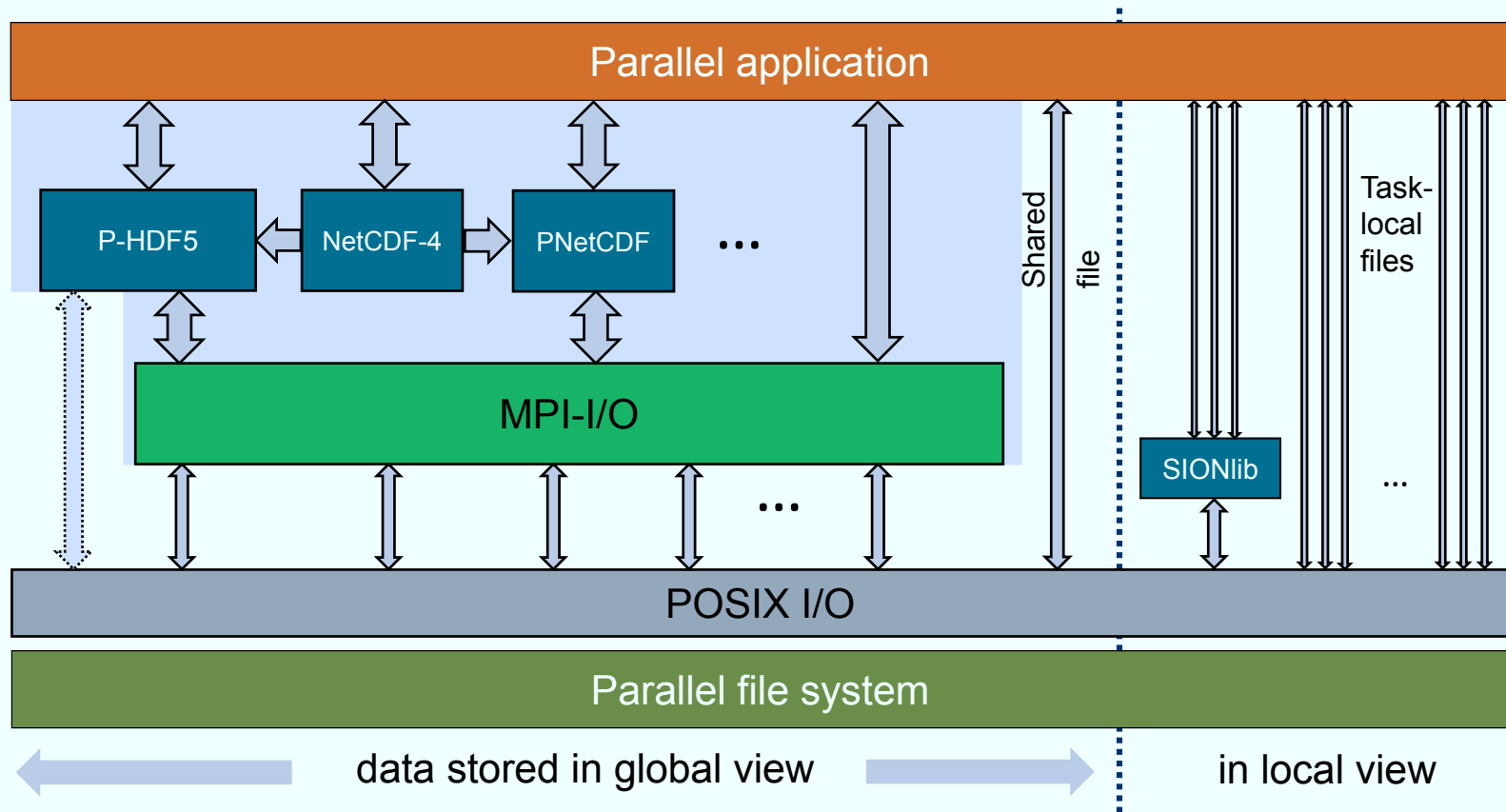
	Address	row-major order (e.g. C/C++)	column-major order (e.g. Fortran)
1	1000	1	1
2	1001	2	4
3	1002	3	7
4	1003	4	2
5	1004	5	5
6
7			
8			
9			

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

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”
→ Restart/checkpoint files
 - **External**: Write/read data in non-decomposed format
(portable, system-independent, self-describing)
→ Workflows, Pre-, Post-processing, Data exchange

Parallel I/O Software Stack



General exercise workflow

- Login to your workstation using `trainXXX` and the given password
- SSH passphrase is the same password
- Start a terminal session and login to the JUROPA3 Cluster:
`ssh -X juropa3.fz-juelich.de`
- Use emacs or vim directly on the JUROPA3 system to avoid copying exercise files from/to the workstation system
- Open another terminal window also connected to the JUROPA3 system
- Start an interactive computing session:
`salloc --reservation=parallel10 --nodes=1
--time=06:00:00`
- Use the interactive session to execute jobs (using `srun`) and your first SSH session to manipulate your files
- You will find exercise and solution files in:
`/work/hpclab/train001`

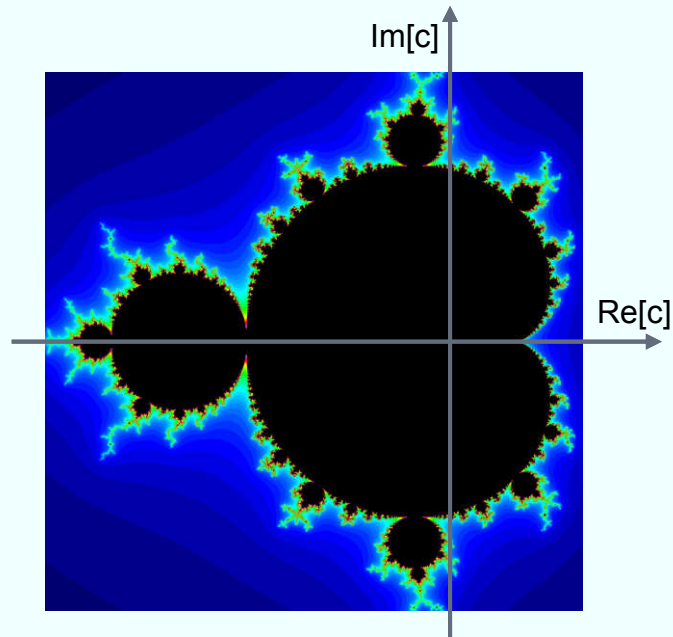
Course exercise: Mandelbrot set

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

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

$$z_0 = 0$$

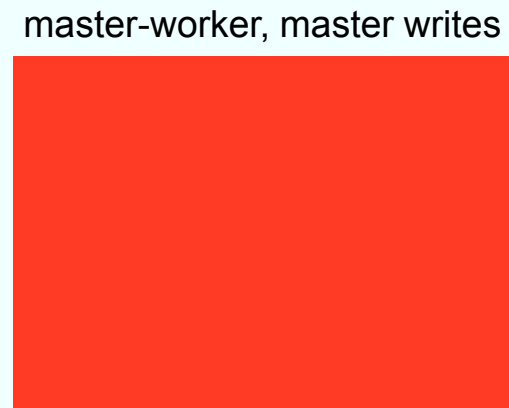
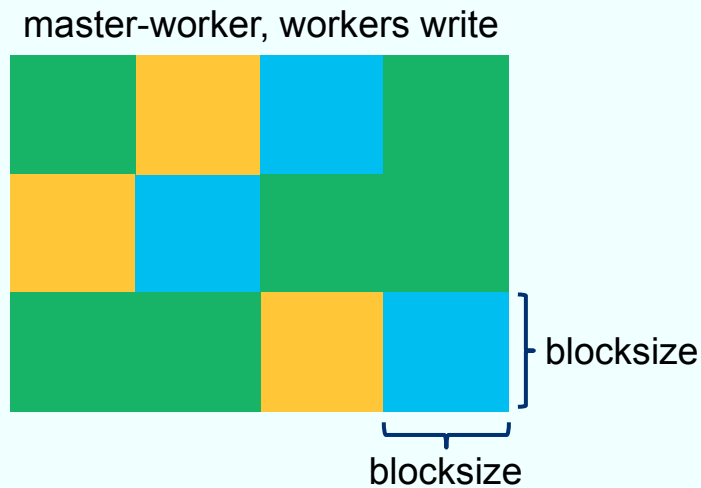
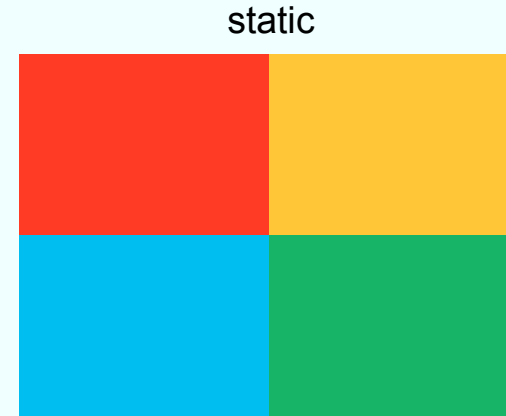
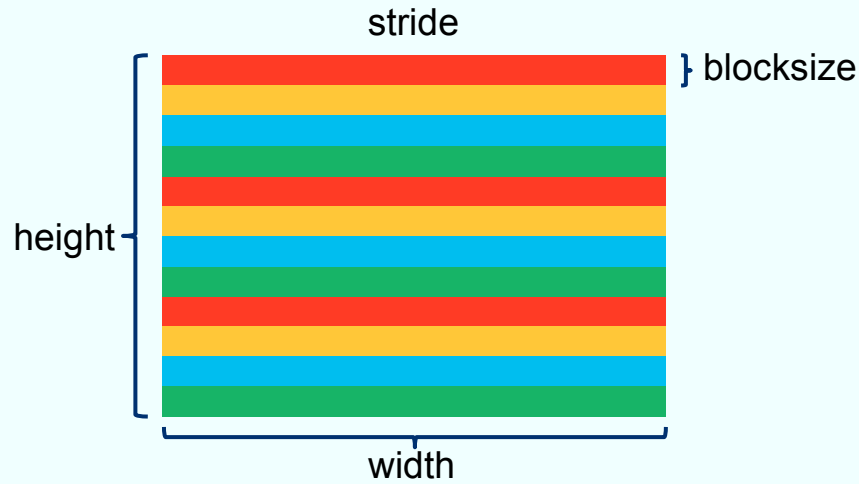
does not approach infinity



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
- Two different programs
 - `mandelmpi`: parallel Mandelbrot calculation
 - `mandelseq`: serial output picture generation

Decomposition types



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

mandelmpi

Example output

```
using SIONlib
start calculation (x= -0.59 .. -0.54,y= -0.58 .. -0.53)
calc_master[00]: 4096x4096
calc_worker[01]: 64x64
calc_worker[02]: 64x64
calc_worker[03]: 64x64

PE 00 of 04: t= 2 4096 x 4096 bs= 64 calc= 50.859, wait= 26.326,
            io= 716.462, mpi= 7893.249, runtime= 8687.163 (ms)

PE 01 of 04: t= 2 4096 x 4096 bs= 64 calc= 5749.752, wait= 25.301,
            io= 805.705, mpi= 2047.276, runtime= 8688.862 (ms)

PE 02 of 04: t= 2 4096 x 4096 bs= 64 calc= 2651.758, wait= 28.214,
            io= 744.241, mpi= 5258.484, runtime= 8693.876 (ms)

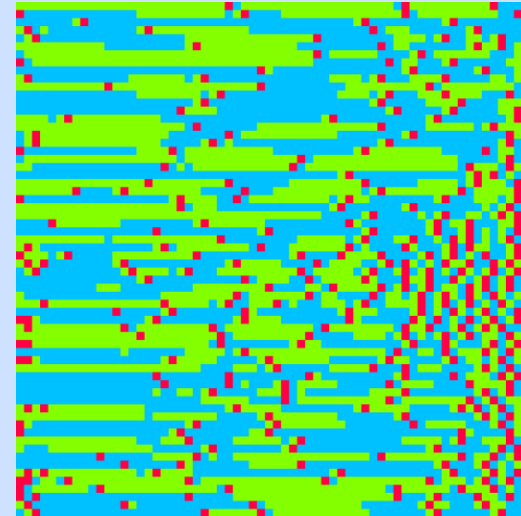
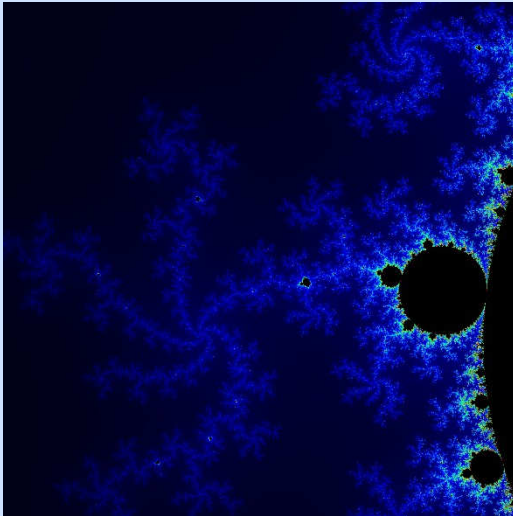
PE 03 of 04: t= 2 4096 x 4096 bs= 64 calc= 4631.728, wait= 42.970,
            io= 793.272, mpi= 3196.786, runtime= 8695.410 (ms)
```

mandelseq

Command line options

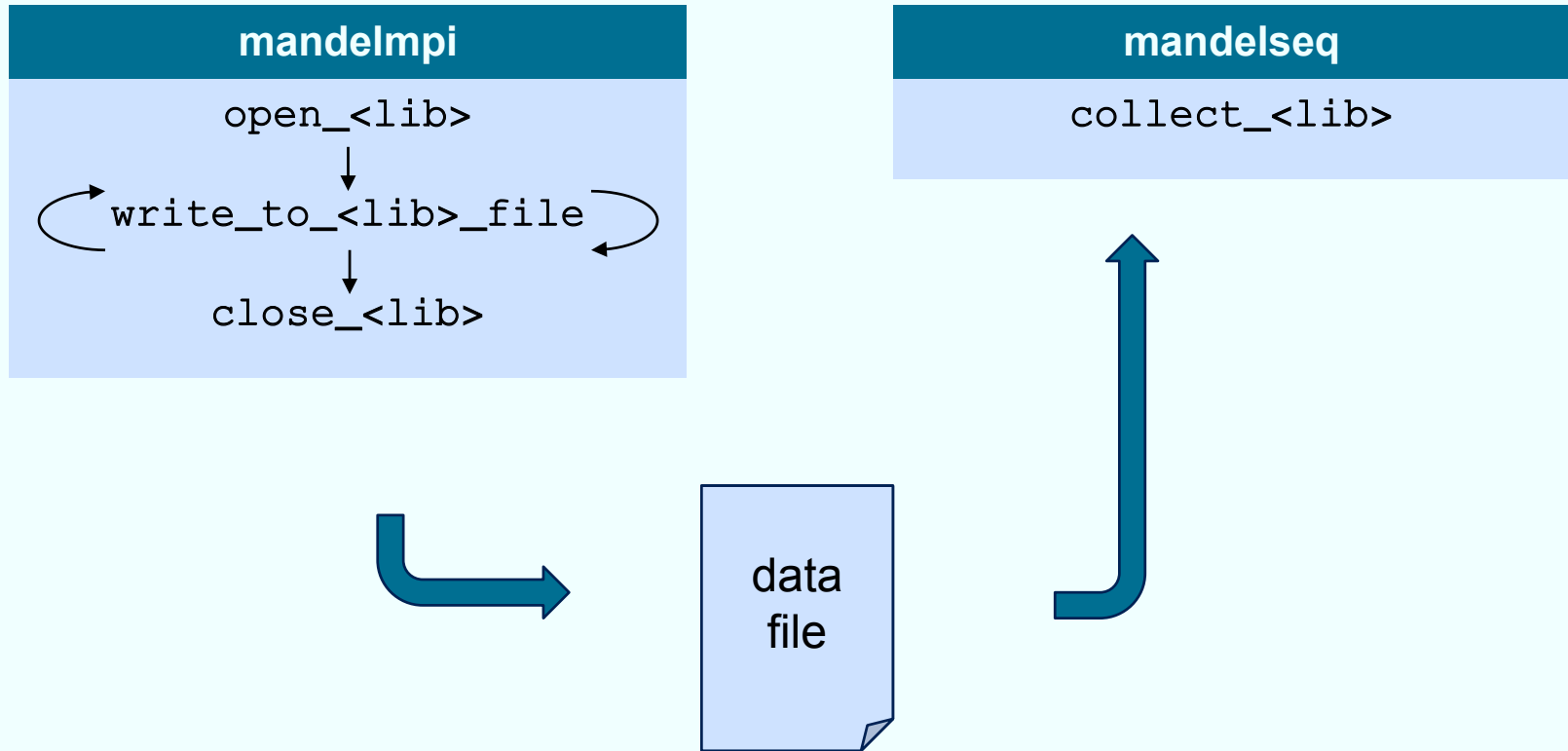
-f output type (0: SIONlib, 1: HDF5, 2: MPI-IO, 3: pnetcdf, 4: netcdf4),
default: 0

Output



process distribution image
only available for SIONlib

Mandelbrot exercise workflow



Mandelbrot exercise workflow

1. Load modules
`. load_modules_juropa3.sh`
2. Run compilation
`make`
3. Change runtime parameter in "run.job" file or use `srun` directly in your interactive session
4. Submit a job if not using an interactive session
`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