HIGH PERFORMANCE PARALLEL IO AND POST-PROCESSING

PARALLEL I/O STRATEGIES AND OPTIMIZATION WITH A FOCUS ON SIONLIB

01.03.2021 I SEBASTIAN LÜHRS (S.LUEHRS@FZ-JUELICH.DE)

HANDS ON PREPARATION

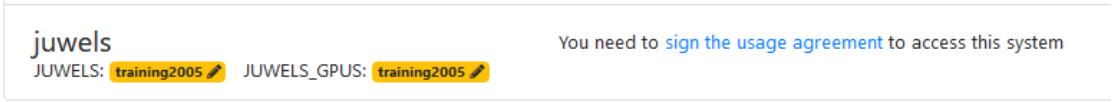
HPC access

JUDOOR account and part of the training2022 project?

- Register and join the course project and wait for acceptance: **https://judoor.fz-juelich.de/projects/join/training2022**

Accepted usage agreement?

- Accept Usage Agreement of JUWELS and JUDAC in your JUDOOR account:



Optional: SSH Key in place?

- Only necessary for non Jupyter based access
- Follow the Guideline document to create your key
- Add your public IP address during upload (need to be renewed on a daily basis)

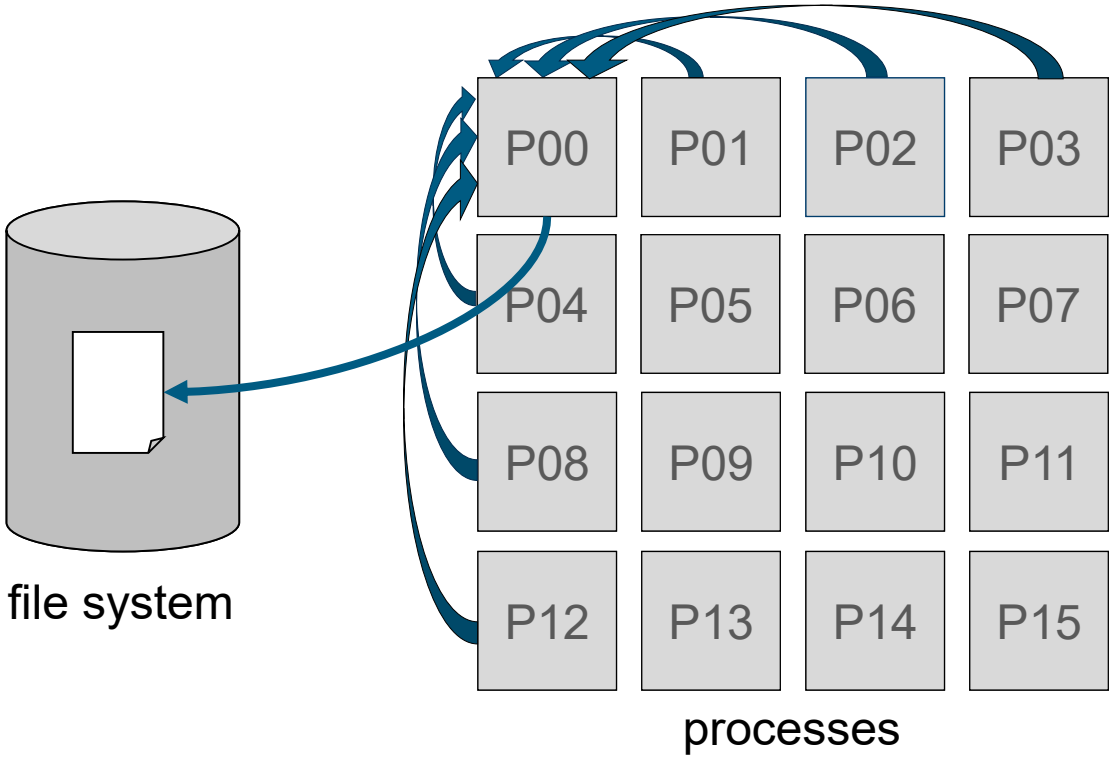
Alternative: Jupyter registration

- Login to https://jupyter-jsc.fz-juelich.de and follow the Jupyter registration steps

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

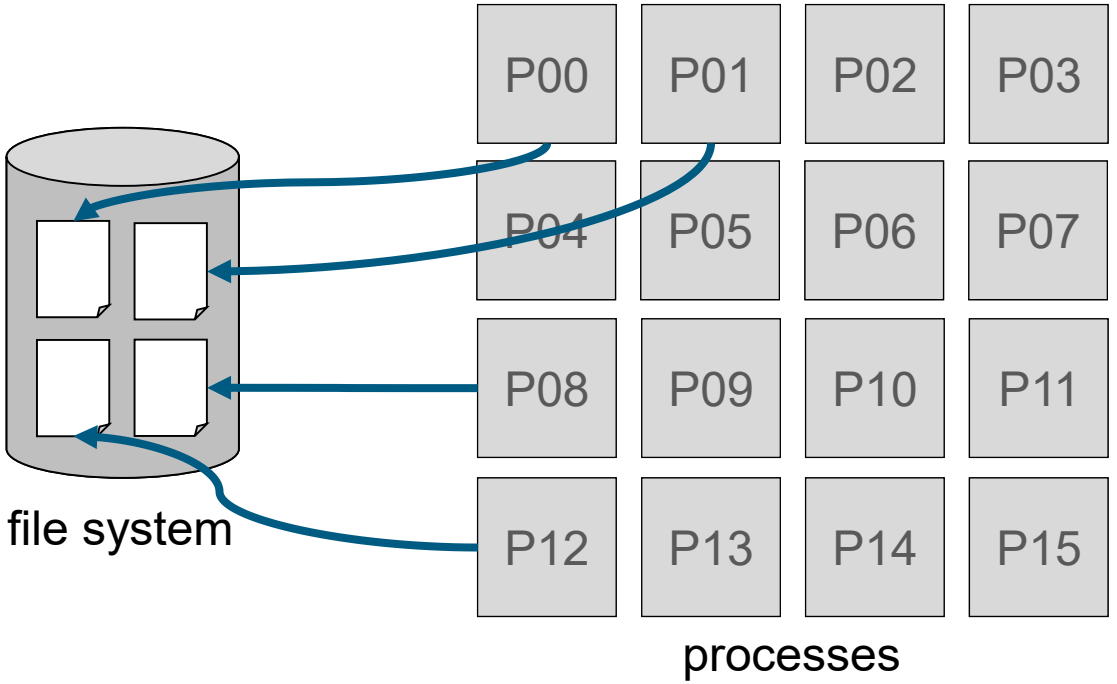
Parallel I/O Pitfalls

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

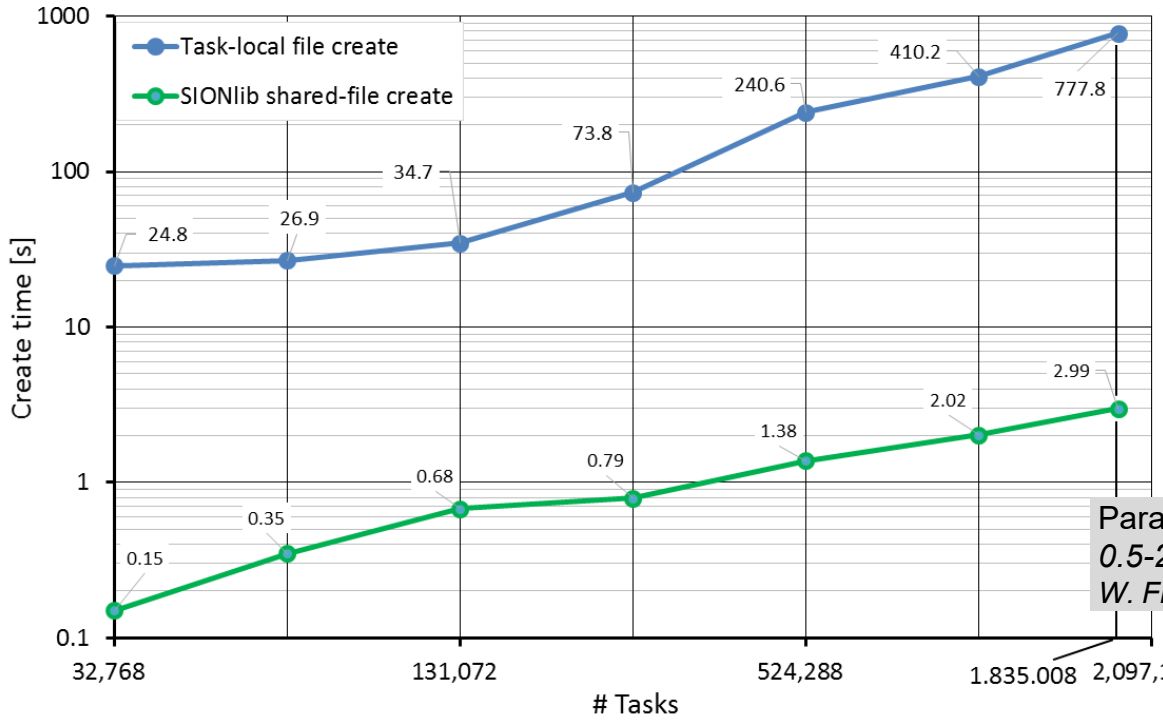
+ Simple to implement
+ No coordination between processes needed
+ No false sharing of file system blocks

- Number of files quickly becomes unmanageable
- Files often need to be merged to create a canonical dataset
- File system might serialize meta data modification

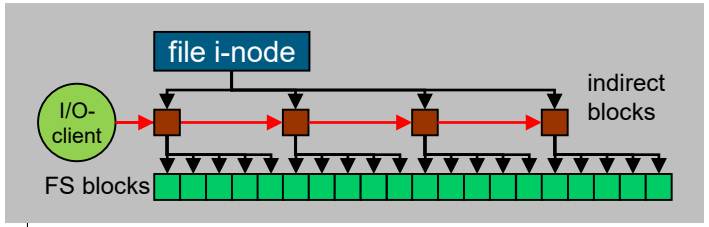
Parallel I/O Pitfalls

Serialization of meta data modification

Example: Creating files in parallel in the same directory



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

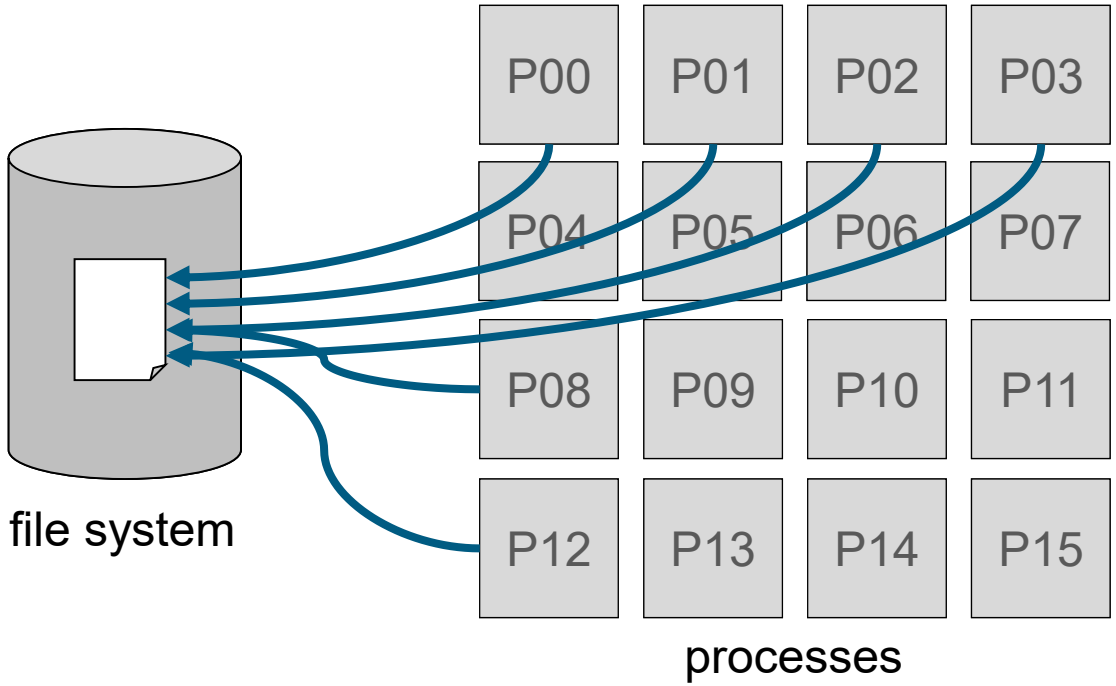


- Meta-data wall on file level
  - File changes by multiple processes can cause serialization
  - File meta-data management
  - Locking

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

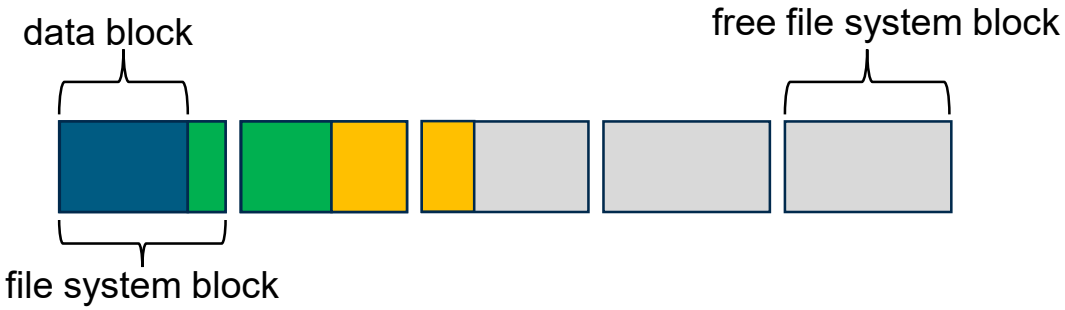
+ Number of files is independent of number of processes
+ File can be in canonical representation (no post-processing)

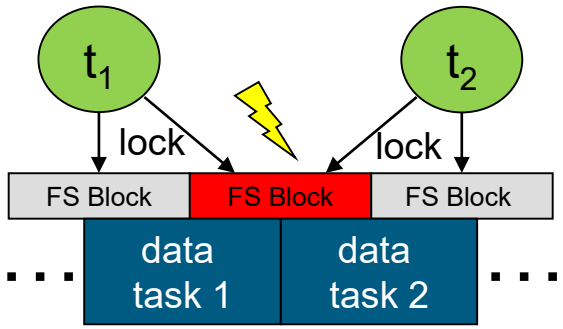
- Uncoordinated client requests might induce time penalties
- File layout may induce false sharing of file system blocks

Parallel I/O Pitfalls

False sharing of file system blocks

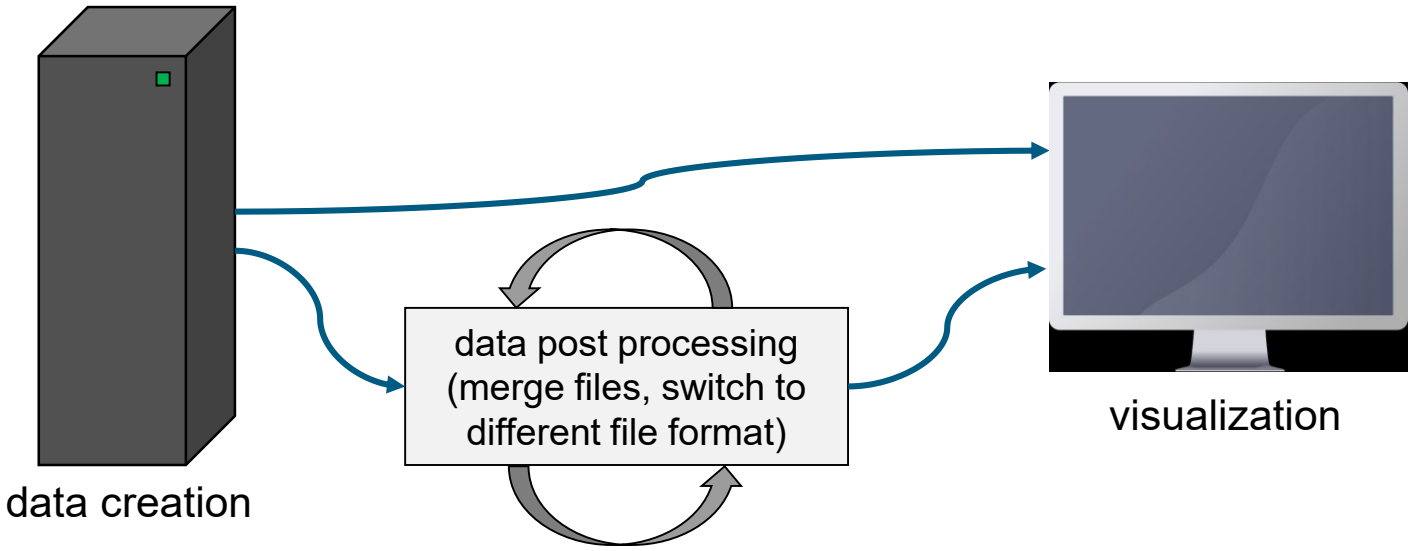
- Data blocks of individual processes do not fill up a complete file system block
- Several processes share a file system block
- Exclusive access (e.g. write) must be serialized
- The more processes have to synchronize the more waiting time will propagate





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

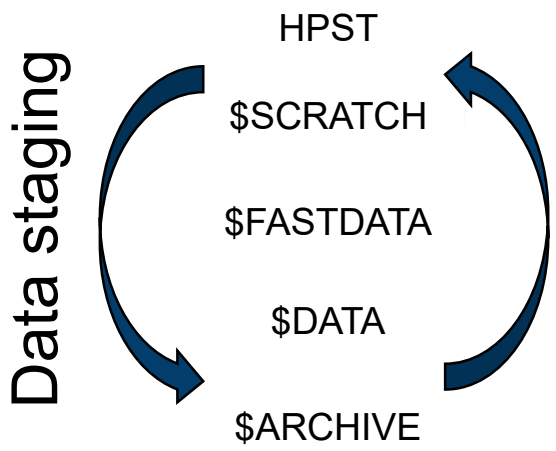
- Memory order depends on programming language
- Transpose of array might be necessary when using different programming languages in the same workflow
- Solution: Choosing a portable data format (HDF5, NetCDF)

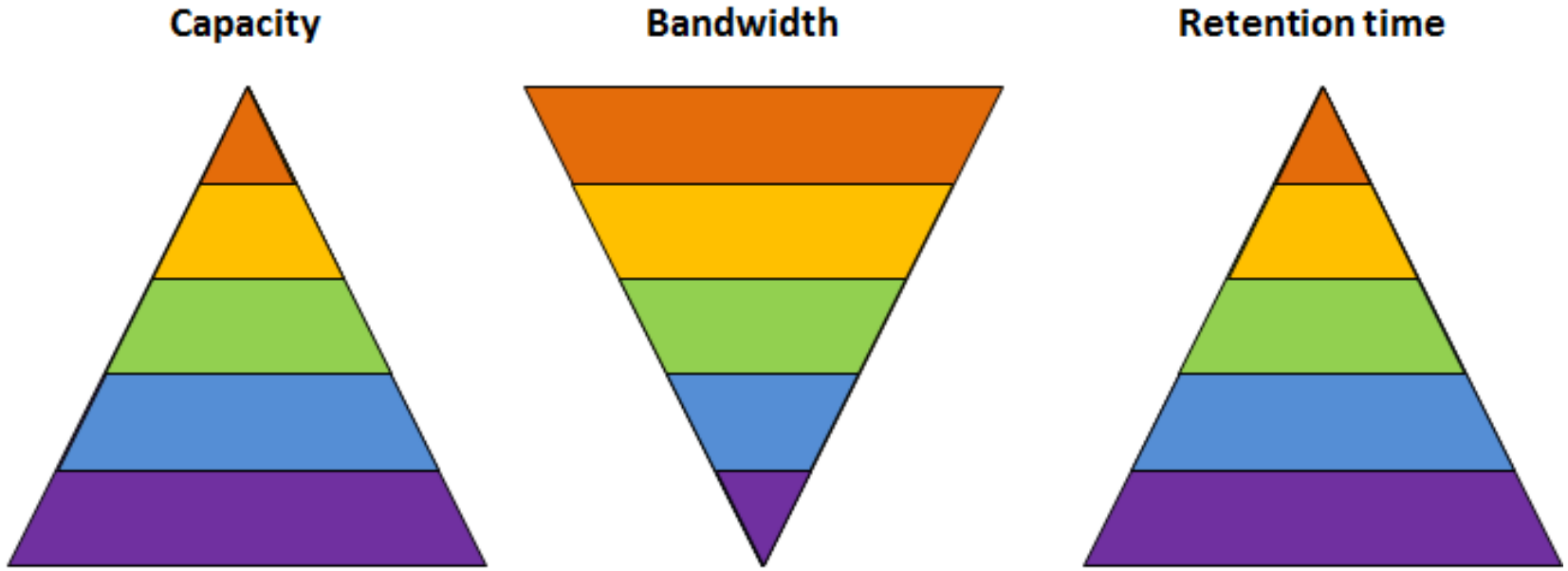
1	2	3
4	5	6
7	8	9

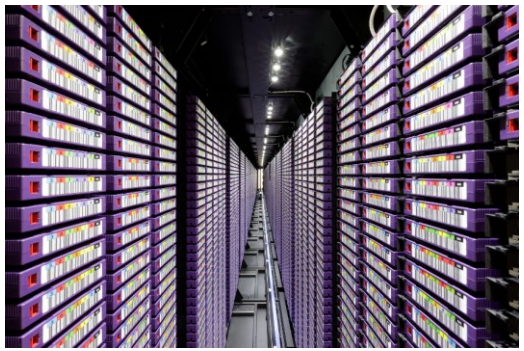
Address	row-major order (e.g. C/C++)	column-major order (e.g. Fortran)
1000	1	1
1001	2	4
1002	3	7
1003	4	2
1004	5	5
…	…	…

Storage Tiers

Different storage tiers with different optimization targets



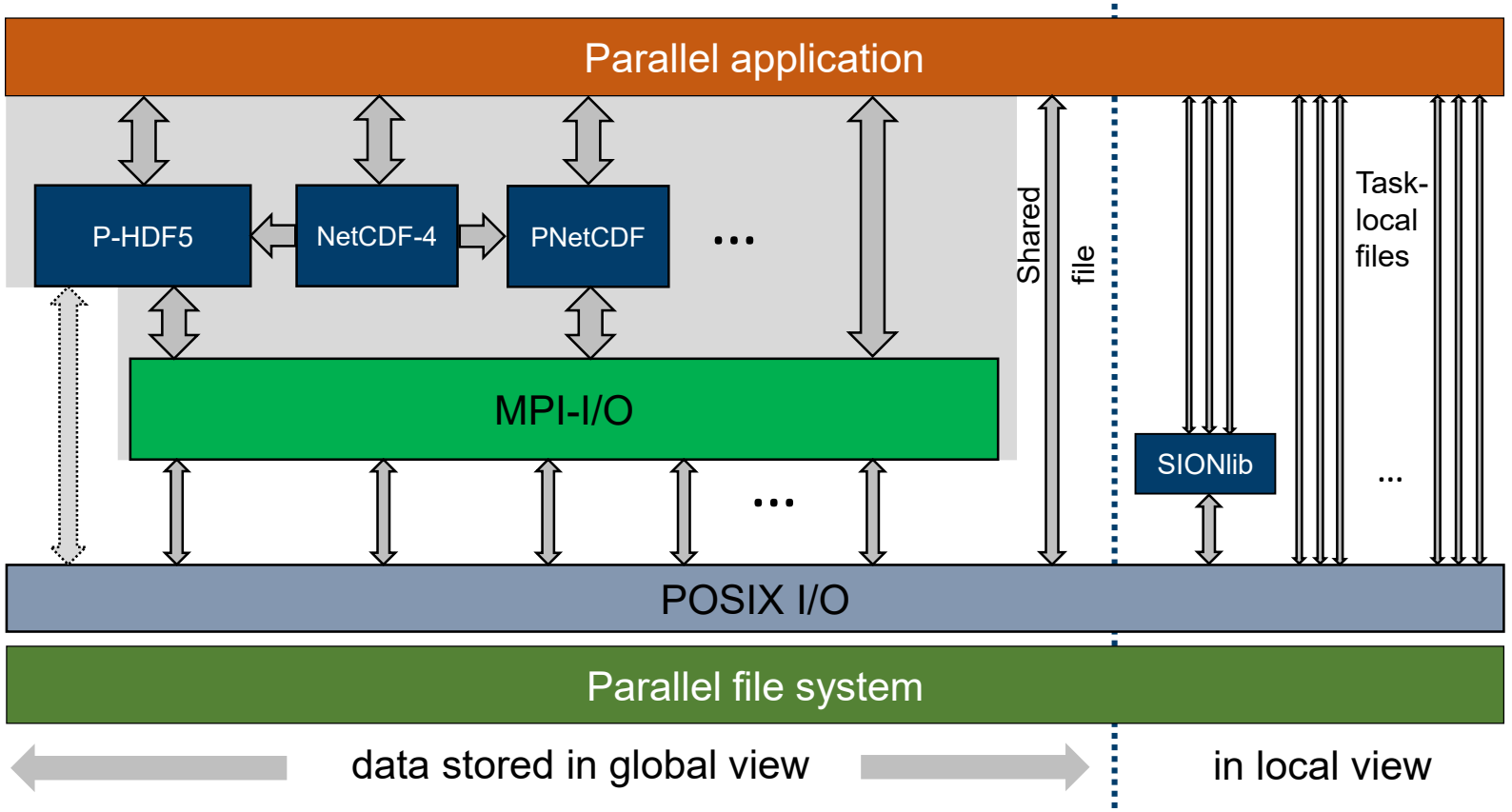




Tape Library

JUST 5

Parallel I/O Software Stack



SCALABLE I/O FOR PARALLEL ACCESS TO TASK-LOCAL FILES WITH SIONLIB

SIONlib Fact Sheet

Data model: n sequences of bytes (untyped data)

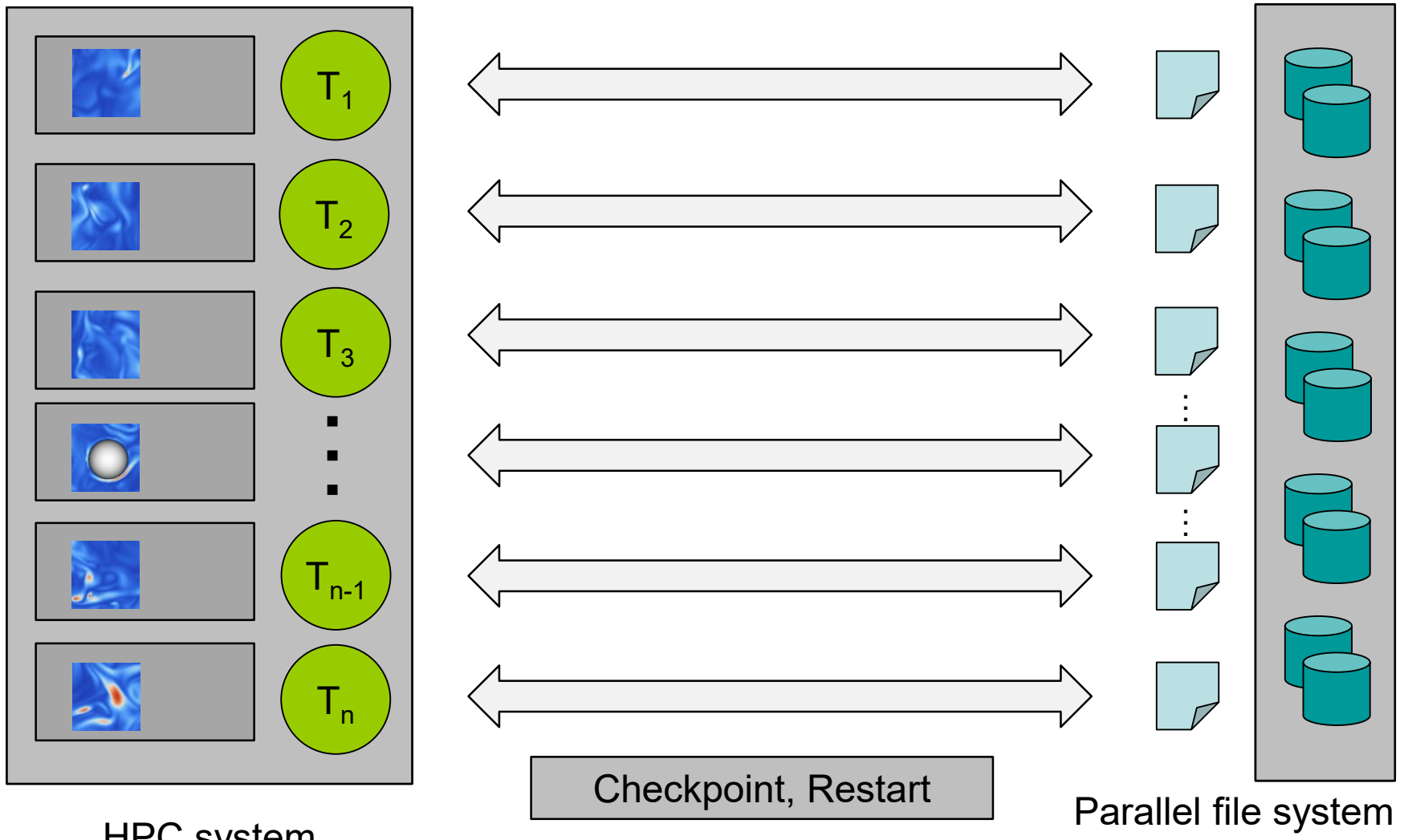
Self describing: no

Full control of file content: no

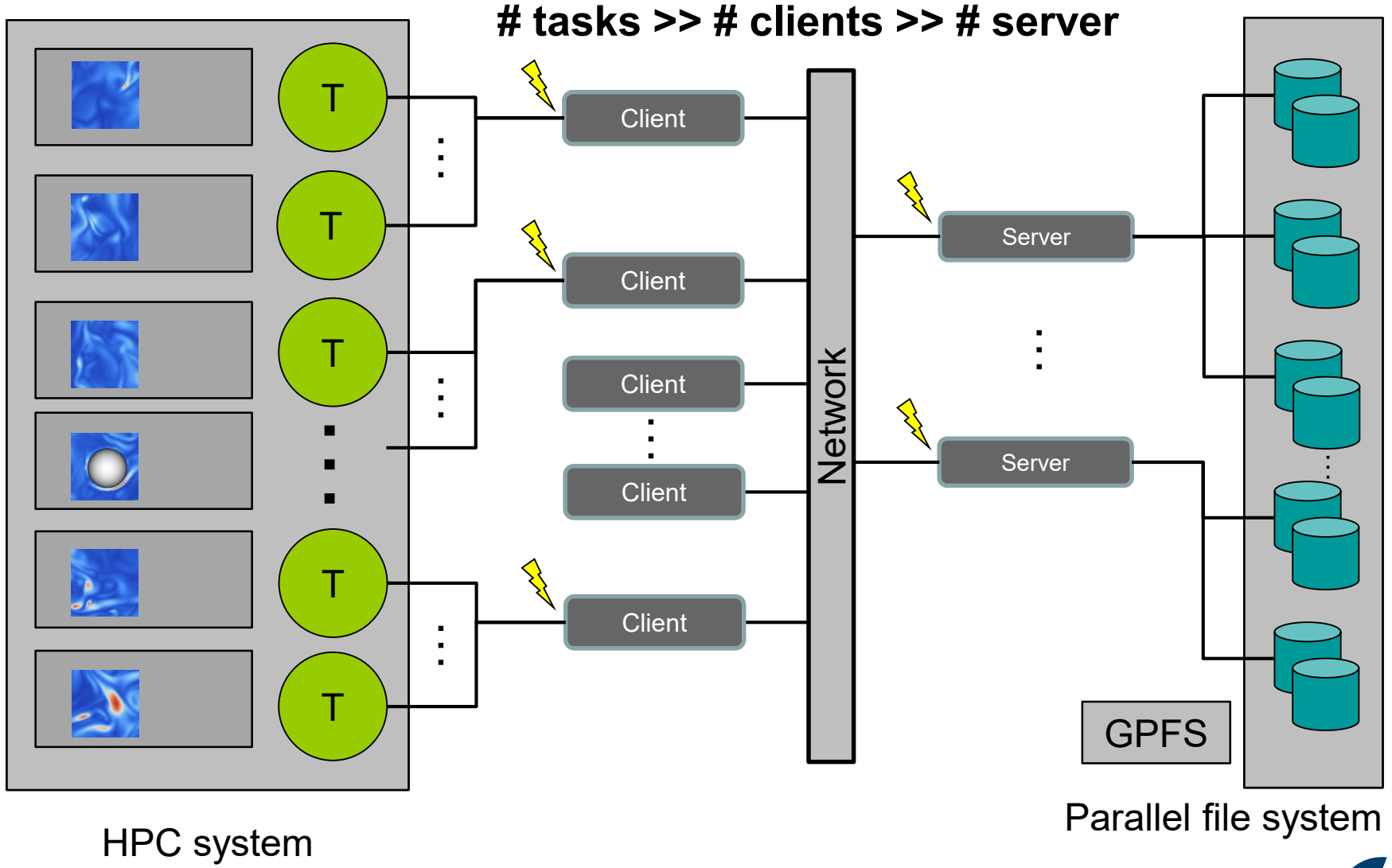
Use cases:

- Data that is by its nature per-task (performance data, logs, …)
- Data for internal use (checkpoints for restarting)

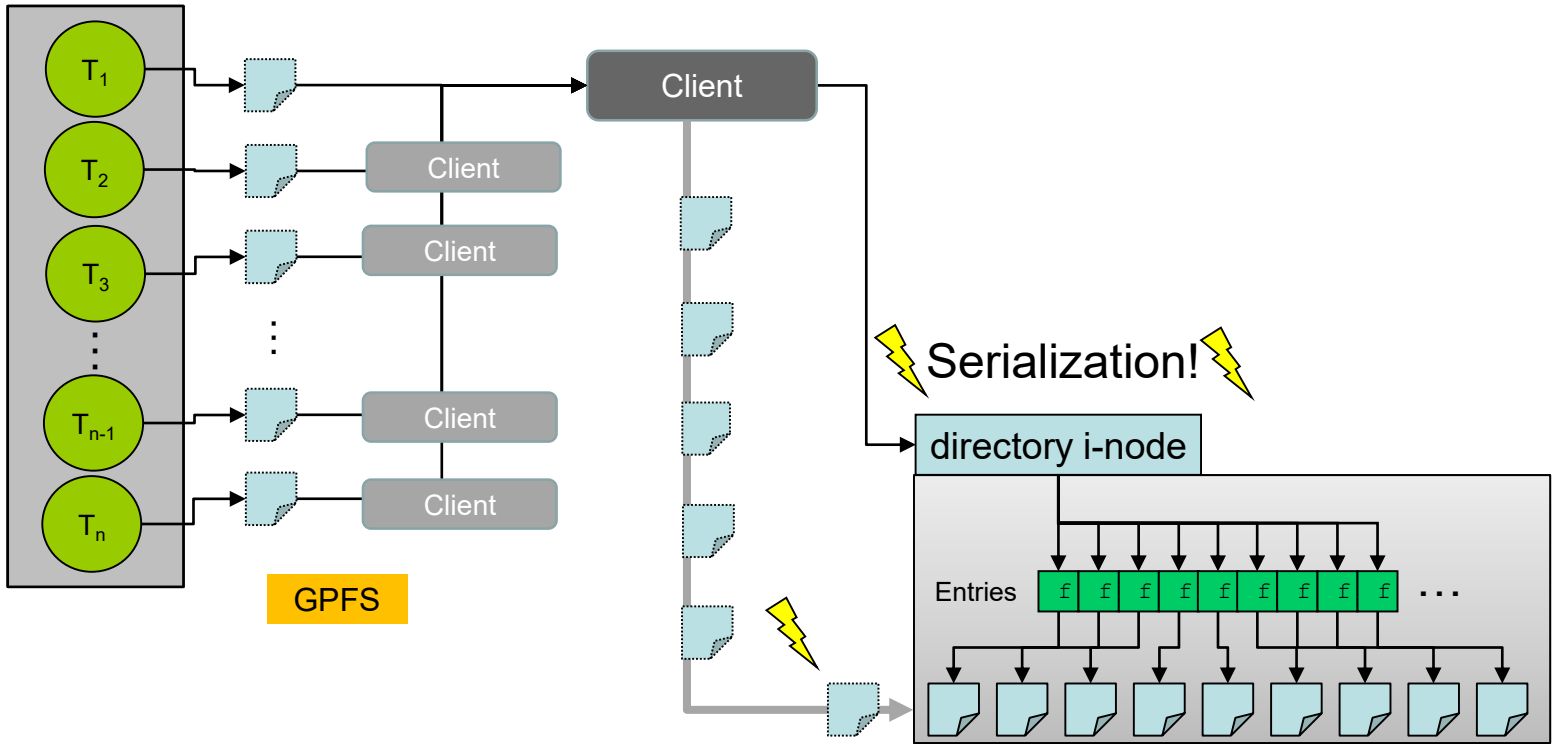
Task-Local I/O



Task-Local I/O

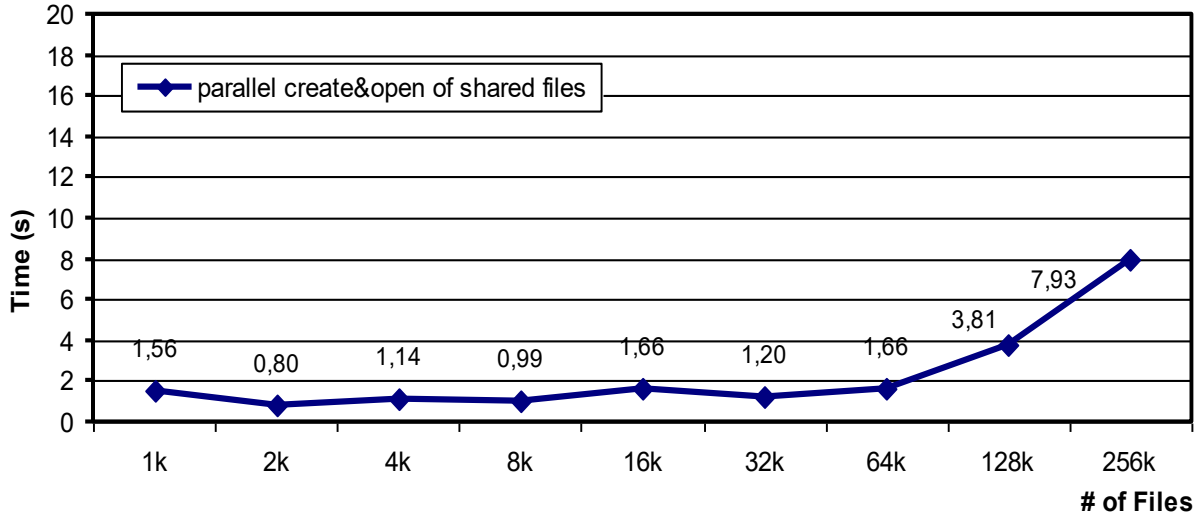


I/O Bottleneck: Parallel File Creation

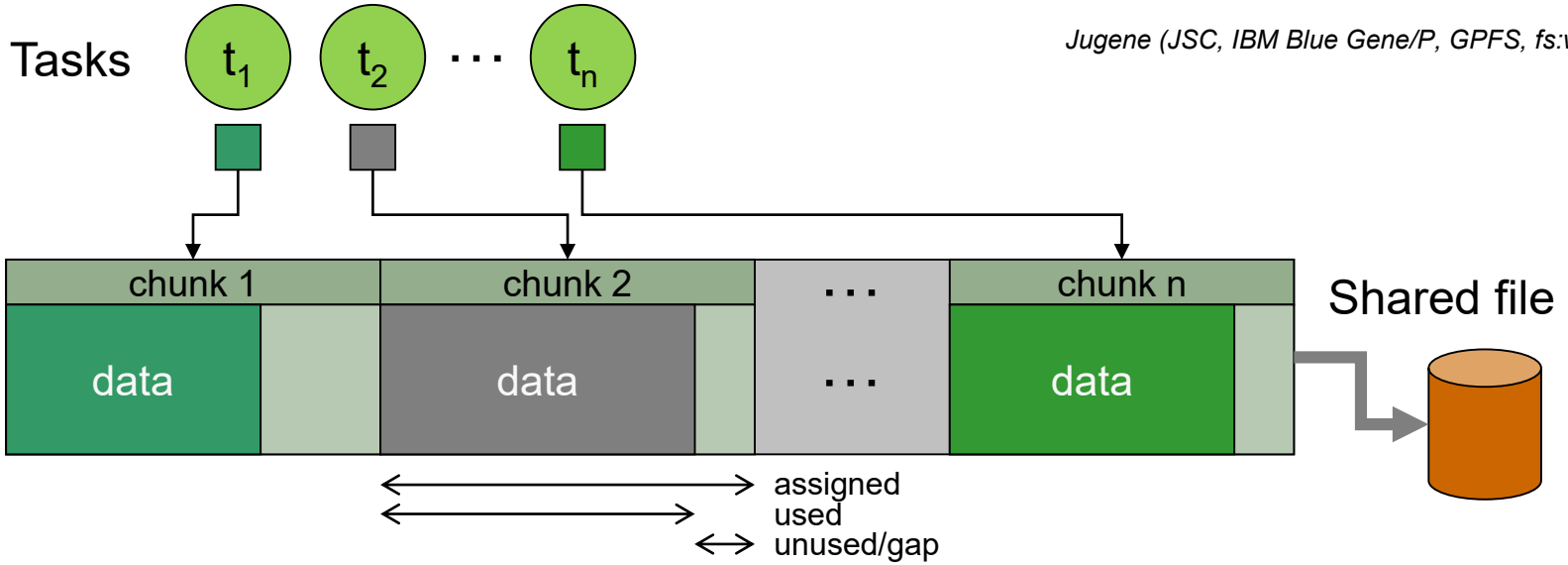


File Format (1): a Single Shared File

- Create and open is fast
- Simplified file handling
- Only one big file
- Logical partitioning required

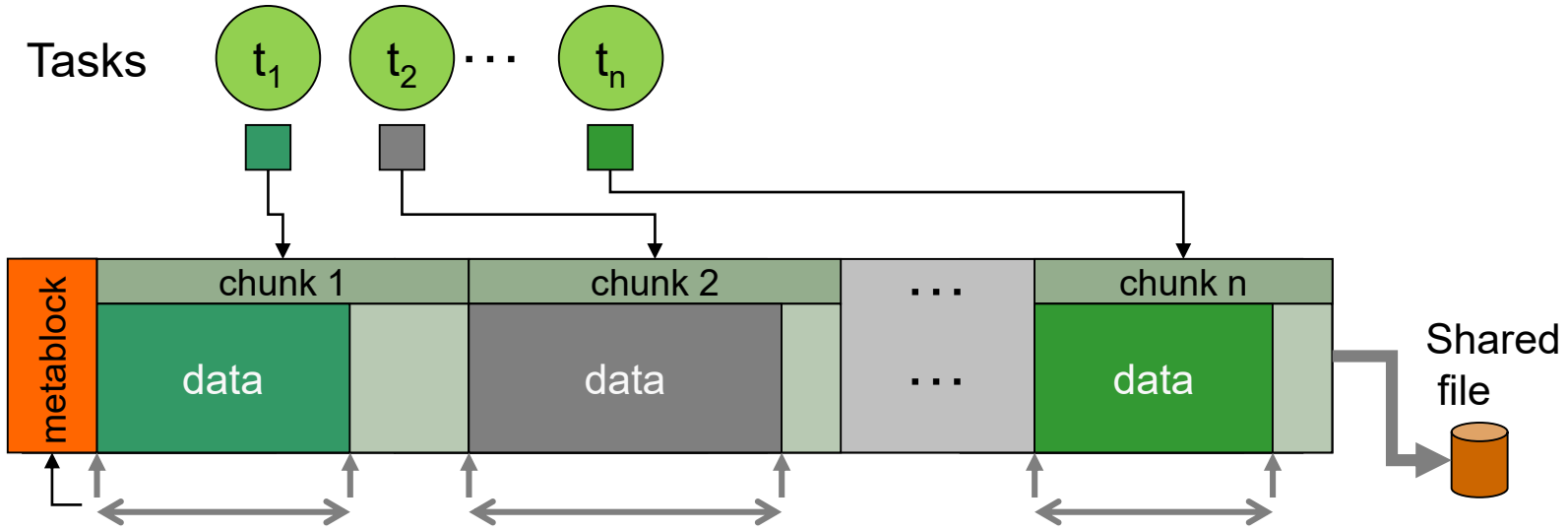


Jugene (JSC, IBM Blue Gene/P, GPFS, fs:work)



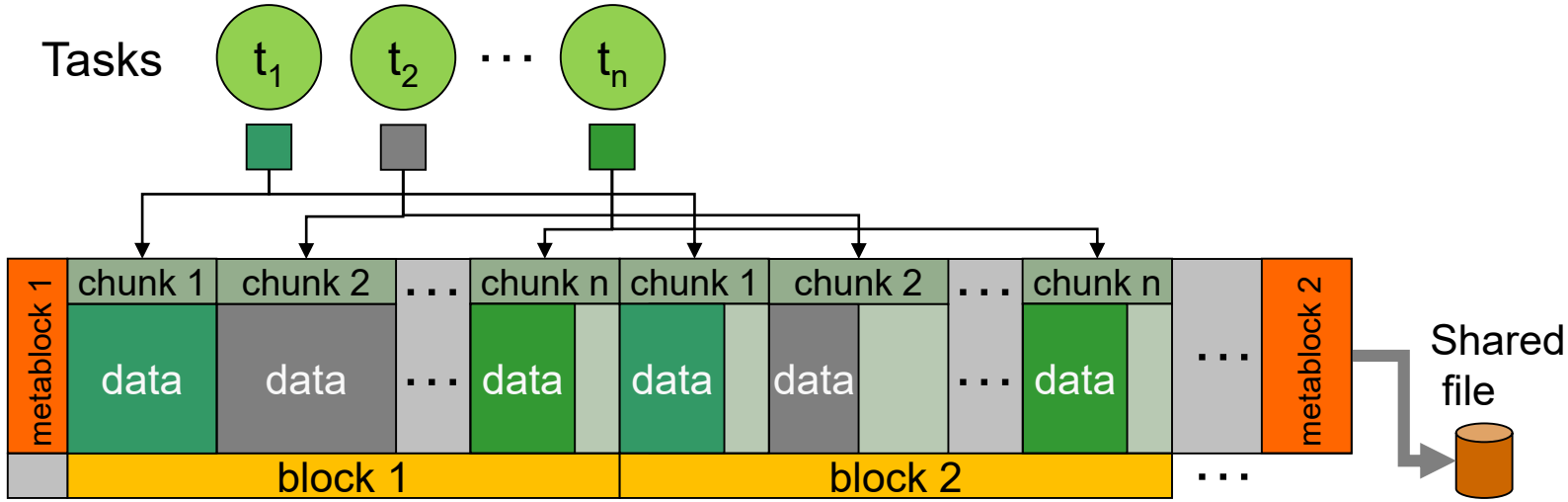
File Format (2): Metadata

- Offset and data size per task
- Tasks have to specify chunk size in advance
- Data must not exceed chunk size



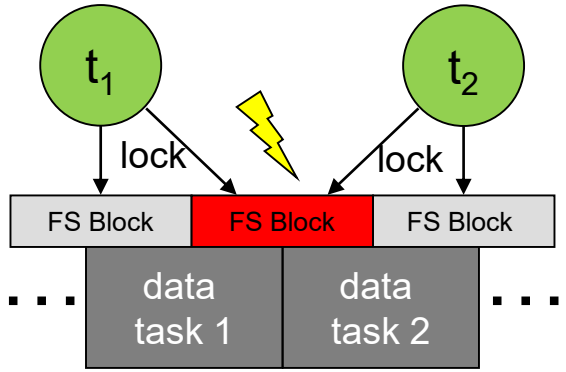
File Format (3): Multiple Blocks of Chunks

- Enhancement: define blocks of chunks
- Metadata now with variable length (#tasks * #blocks)
- Second metadata block at the end
- Data of one block does not exceed chunk size



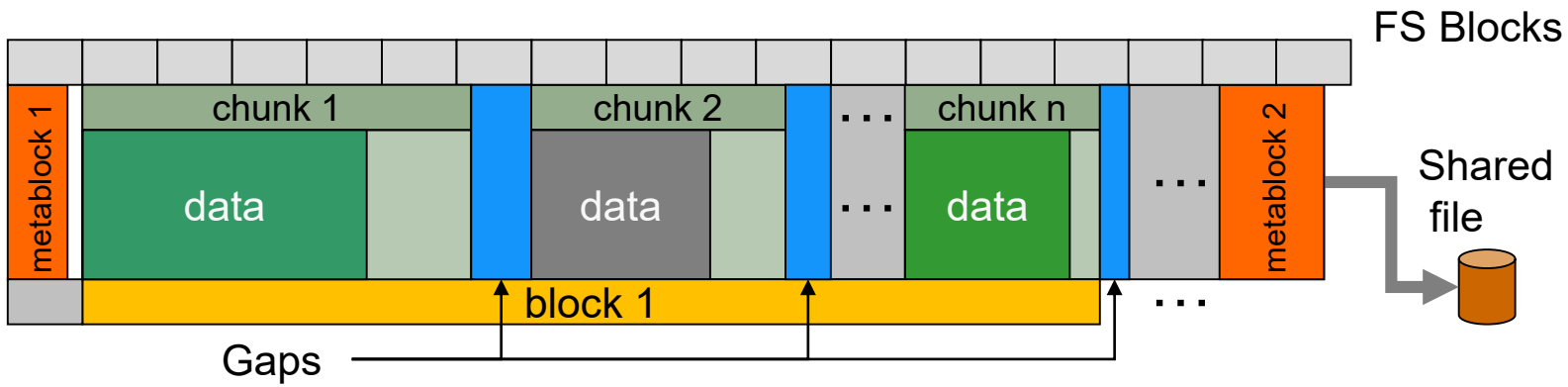
File Format (4): Alignment to Block Boundaries

- Contention when writing to same file-system block in parallel



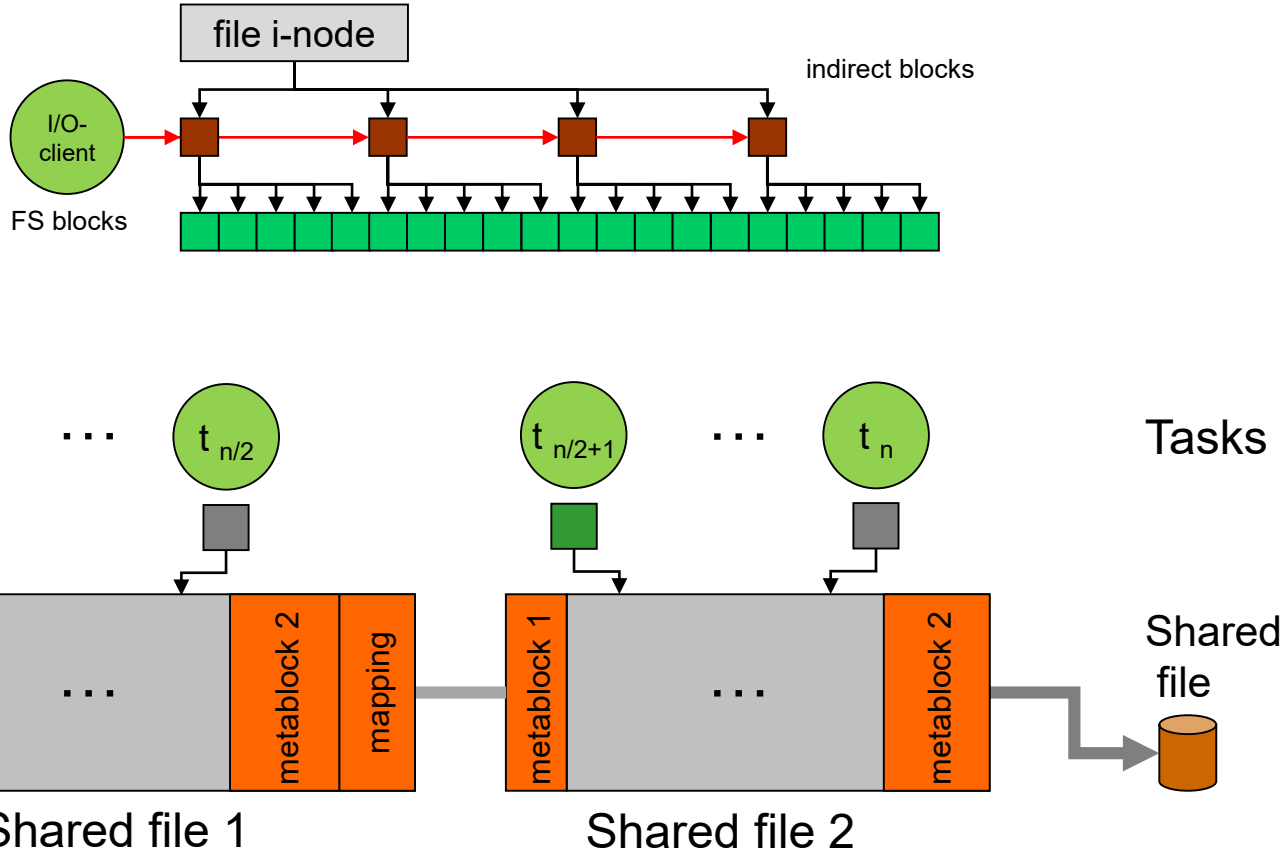
#tasks	data size	blksize	write bandwidth
32768	256 GB	aligned	**3650.2** MB/s
32768	256 GB	not aligned	**1863.8** MB/s

Jugene (JSC, IBM Blue Gene/P, GPFS, fs:work)

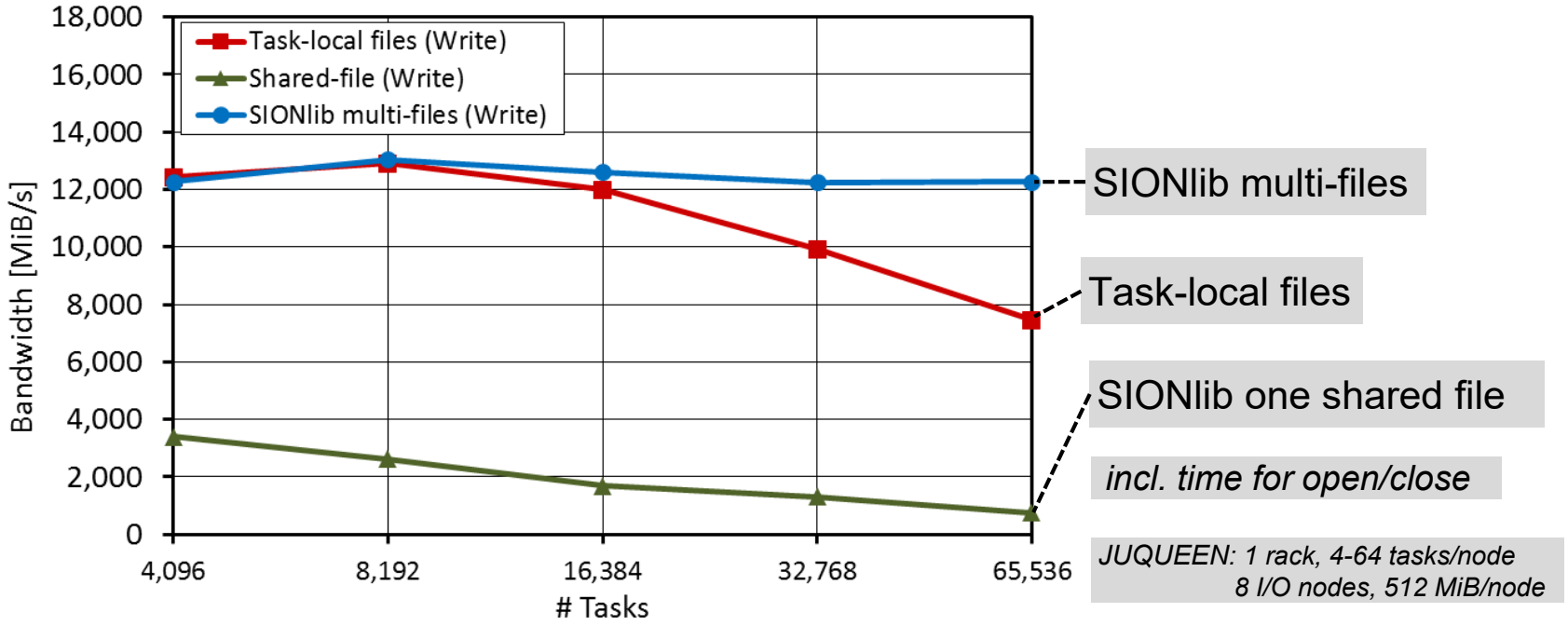


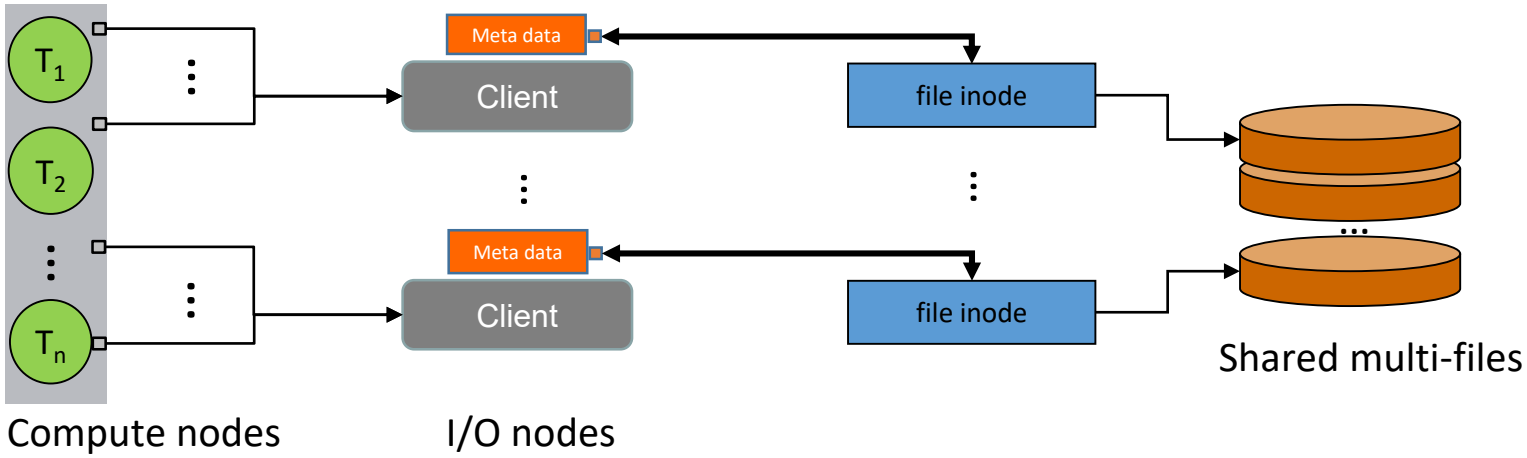
File Format (5): Multiple Physical Files

- Variable number of underlying physical files
- Bandwidth degradation on GPFS by using a single shared file

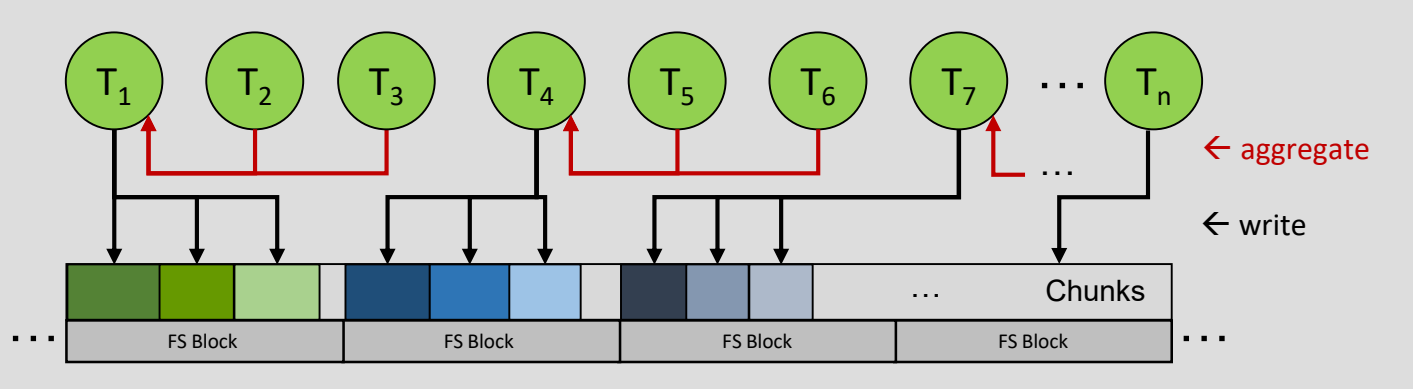


I/O Bottleneck: Large #tasks & Shared fileS





File Format (6): Coalescing I/O



- data/task very small → sparse file container
- Coalescing I/O: Aggregation of data among tasks
- Variable number of senders/collectors
- Collective write/read operations required
- Advantages: No alignment between chunks of the same collector; less gaps, fewer data streams, less congestion in I/O infrastructure
- Reduced buffering on collector task (one file system block)

Version, Download, Installation

- Version: 1.7.6 (November 2019), Version 2.0.0-rc.3 (February 2021)
- Open-source license
- https://www.fz-juelich.de/jsc/sionlib
- *sionlib_jsc@fz-juelich.de*
- Query version of SIONlib:

Shell
```
sionversion
SIONlib Version 1.7.2 (git_rev e7c4192), fileformat
version 5 (sionversion)
```

References:
- Wolfgang Frings, Felix Wolf, Ventsislav Petkov, **Scalable Massively Parallel I/O to Task-Local File**, Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, Portland, Oregon, November 14-20, 2009, SC'09, New York, ACM, ISBN 978-1-60558-744-8.
- Wolfgang Frings, **Efficient Task-Local I/O Operations of Massively Parallel Applications**, Schriften des Forschungszentrums Jülich, IAS Series, 30, 2016, ISBN 978-3-95806-152-1

Header Files & Datatypes

C
```
#include <sion.h>
```

Fortran
```
use sion_f90
use sion_f90_mpi
```

Special datatypes are typically used for all parameters that are used to describe or to compute file positions

C
```
sion_int32
```

Fortran
```
INTEGER(kind=4)
```

C
```
sion_int64
```

Fortran
```
INTEGER(kind=8)
```

Collective Open (MPI)

C

```
int sion_paropen_mpi (char  *fname, const char *file_mode,
                      int *numFiles,
                   MPI_Comm gComm,   MPI_Comm *lComm,
                 sion_int64 *chunksize,
                 sion_int32 *fsblksize,
                      int *globalrank,
                     FILE **fileptr, char **newfname);
```

Fortran

```
FSION_PAROPEN_MPI (FNAME, FILE_MODE, NUMFILES,
                   GCOMM, LCOMM, CHUNKSIZE, FSBLKSIZE,
                   GLOBALRANK, NEWFNAME, SID)
CHARACTER*(*) FNAME, FILE_MODE, NEWFNAME
INTEGER       NUMFILES, FSBLKSIZE, GLOBALRANK, SID
INTEGER       GCOMM, LCOMM
INTEGER*8     CHUNKSIZE
```

- Open a SION file in parallel for reading or writing data
- Collective call, called by each task at the same time
- Accesses one or more physical files of a logical SION file
- Parameters are passed “by reference” to pass back information in read open mode

Parameters of Open Calls I

fname: file name

- Character string describing path and file name
- Will not be extended by SION-specific suffix
- In general multiple physical files are generated
- First file: **filename**
- All other files: **filename** + "." + 6-digit-number (000001 ...)
- All commands and function calls use the base name

file_mode: file mode

- r,rb,br (read block), open existing SION file for reading
- w,wb,bw (write block), create a new SION file, open for write; overwrite if existing
- posix use internally POSIX interface for file access, otherwise ANSI-C

Parameters of Open Calls II

sid: SIONlib file descriptor

- Unique integer value, referring internally to data structure
- Associated to SION file (internal file handle)
- Allows multiple simultaneously opened files
- C: return code, Fortran: last parameter of open call
- Integer file handle for Fortran necessary

chunksize: size of data per task

- Pointer of type: sion_int64* (C), Integer*8 (Fortran)
- Size of data in bytes written by this tasks (maximum size of single sion_fwrite call)
- May be different for each task and must be set if open for writing
- Will be increased internally to the next multiple of the file system block size

Parameters of Open Calls III

fsblksize: file system block size

- Size of file system block in bytes
- Read-mode: file system block size at write time
- Write-mode: automatically detected by SIONlib if set to -1 *(recommended)*

fileptr: ANSI-C file pointer

- Can be replaced by NULL pointer if not needed *(recommended)*
- Will be removed in future versions
- Only needed if wrapper functions for writing and reading are not used
- Not available in Fortran

newfname:

- File name of physical file assigned to this task, NULL can be used

Parameters of sion_paropen_mpi

gComm: MPI Communicator

- Call is collective over all tasks of this communicator
- Each task gets assigned one chunk of SION file
- Read: number of tasks must be equivalent to number tasks written to SION file

lComm: MPI Communicator

- Tasks of the same communicator are writing to the same physical file
- MPI_Comm_null if not specified, otherwise Union of lComm must equal gComm

numfiles: Number of physical files

- If using lComm: set numfiles=-1
- Read-mode: parameters will be set by open call

globalrank:

- Rank of task in global communicator gComm

Serial Open

C

```
int sion_open (char         *fname,
               const char   *file_mode,
               int          *ntasks,   int *nfiles,
               sion_int64 **chunksizes,
               sion_int32   *fsblksize,
               int **globalranks,  FILE **fileptr);
```

Fortran

```
FSION_OPEN (FNAME, FILE_MODE, NTASKS, NUMFILES,
            CHUNKSIZES, FSBLKSIZE, GLOBALRANKS, SID)
CHARACTER*(*) FNAME, FILE_MODE
INTEGER       NUMFILES, NTASKS, FSBLKSIZE, SID
INTEGER       GLOBALRANKS(NTASKS)
INTEGER*8     CHUNKSIZES(NTASKS)
```

- All chunks of all tasks can be selected, via `sion_seek` (does not work in writing mode)
- Multiple physical files can be handled
- Reads all metadata of all tasks into memory

Collective Close

C
```
int sion_parclose_mpi(int sid)
```

Fortran
```
FSION_PARCLOSE_MPI (SID, IERR)
INTEGER SID, IERR
```

- Closes a SION file in parallel on all tasks/threads
- Collective call, called by each task/thread at the same time
- Metadata will be collected from each tasks
- Metadata blocks of SION file will be written in this call

Serial Close

C
```
int sion_close(int sid)
```

Fortran
```
FSION_CLOSE(SID, IERR)
INTEGER SID, IERR
```

- Closes a SION file in serial mode
- Metadata blocks of SION file will be written in this call

Exercise

Exercise 1 – SIONlib hello world

- Login to JUWELS (using SSH or a Jupyter-Terminal) -> See Guidelines PDF
- Run (necessary after every new login)

```
source $PROJECT_training2022/setup
```

- Create a personal copy of the SIONlib template files:

```
cp -r $PROJECT_training2022/SIONlib $HOME
```

- Write a parallel program in C or Fortran which creates and closes an empty SIONlib file
- Use a chunksize to store 1000 Integer
- Use the template file `exercise_1.c` or `exercise_1.f90`

```
mpicc exercise_1.c `sionconfig --libs --mpi`
mpif90 `sionconfig --cflags --mpi --f90` exercise_1.f90 `sionconfig
--libs --mpi --f90`

srun -n 10 --reservation=parallel-io-day1 a.out
```

Check details of the resulting file using:

```
siondump <sionlib_output_file>
```

Write Data

C

```
size_t sion_fwrite (void *data,
                    size_t size,
                    size_t nmemb,
                    int    sid);
```

Fortran

```
FSION_WRITE (DATA, SIZE, NMEMB, SID, RC)
INTEGER SID
INTEGER*8 SIZE,NMEMB, RC
```

- Write `size*nmemb` bytes of data, beginning from current position
  - This size must not exceed the chunksize defined in open call!
- Returns number of elements written

Exercise

Exercise 2 – SIONlib write

- Extend your parallel program in C or Fortran
- Each task should fill a array with 1000 elements using its unique rank
- Each task should write the array into the prepared SIONlib file
- Use the template file `exercise_2.c` or `exercise_2.f90`

Check the resulting file using:

```
siondump <sionlib_output_file>
```

Read Data

C

```
size_t sion_fread (void  *data,
                   size_t size,
                   size_t nmemb,
                   int    sid);
```

Fortran

```
FSION_READ (DATA, SIZE, NMEMB, SID, IERR)
INTEGER SIZE, NMEMB, SID, IERR
```

- Read `size*nmemb` bytes from current position in chunk
- Cannot read more than a chunk
  - `sion_bytes_avail_in_block` must be used to get all available data
- Returns number of elements read

End of File

C
```
int sion_feof(int sid);
```

F
```
FSION_FEOF (SID, IERR)
INTEGER SID, IERR
```

- Internally this function flushes all buffer and checks current positions against chunk boundaries
- Moves file pointer to next chunk if end of current chunk is reached
- The function is a task-local function, which can be called independently from other MPI tasks.
- Returns 1 if pointer is behind last byte of data for this task

Seek: Change File Position

C

```
int sion_seek (int         sid,
               int         rank,
               int         chunknr,
               sion_int64 posinchunk );
```

Fortran

```
FSION_SEEK (SID, RANK, CHUNKNUM, POSINCHUNK, IERR)
INTEGER     SID, RANK, CHUNKNUM, IERR
INTEGER*8   POSINCHUNK
```

- Sets the file pointer to a new position, only available in reading mode
- Seek parameters:
  - `rank`: rank number (0,...), or SION_CURRENT_RANK
  - `chunknum`: chunk number (0,...), or SION_CURRENT_BLK
  - `posinchunk`: position (0,...), or SION_CURRENT_POS

Command line tools

- **siondump:** Show metadata of file
- **sionsplit:** Split SIONlib file into task-local files
- **sioncat:** Extract data from file
- **siondefrag:** Contracting all of the chunks of a task, which are spread in the file over multiple blocks, into a single chunk
- **sionconfig:** Get compile and link options

SIONlib 2.0.0

- Currently 2.0.0-rc.3
- Continuous read and write:
  - `sion_fread` can read without limitations, even if the amount of data requested spans multiple chunks
  - `sion_fwrite` can write without limitations, even if the amount of data specified spans multiple chunks
- POSIX file pointer access is removed
- Simplified open functions:

```
int sion_open(const char *name, sion_open_mode mode, int n, const sion_options *options);
```

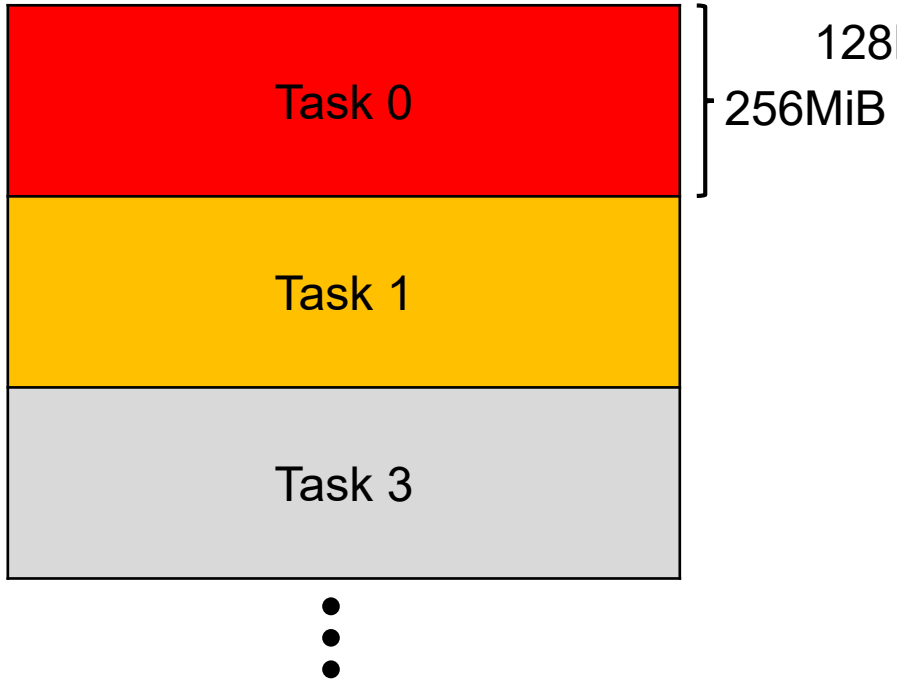
- New seek functions
- Removal of deprecated items
- New build system
- See also porting guide: https://apps.fz-juelich.de/jsc/sionlib/docu/2.0.0-rc.3/porting-2.html

OPTIMIZATION AND PROFILING

I/O patterns

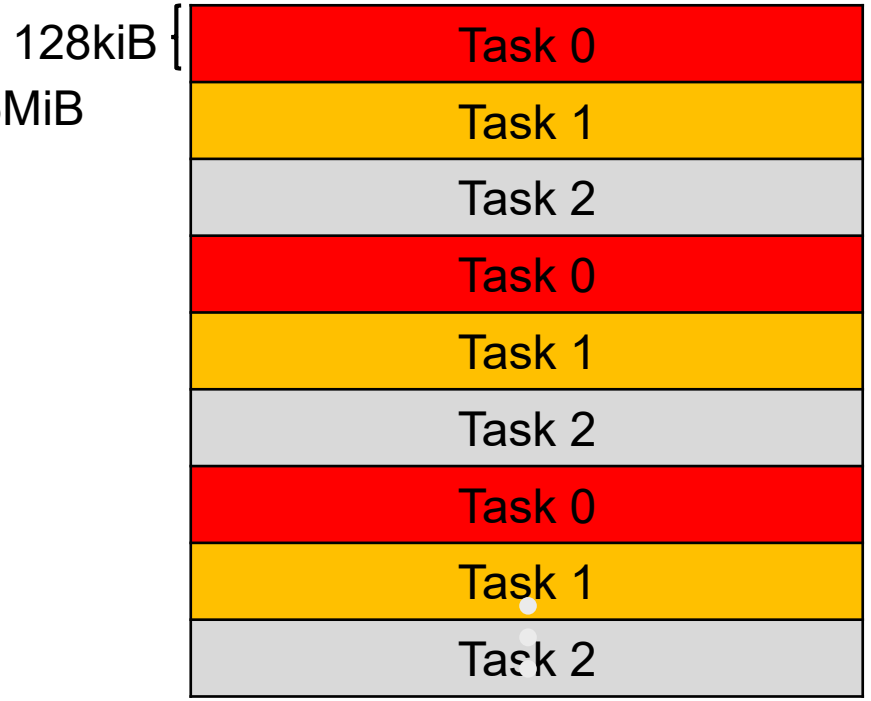
continuous

- Large continuous data blocks for each individual process



striped

- Pattern often found while handling multi dimensional arrays

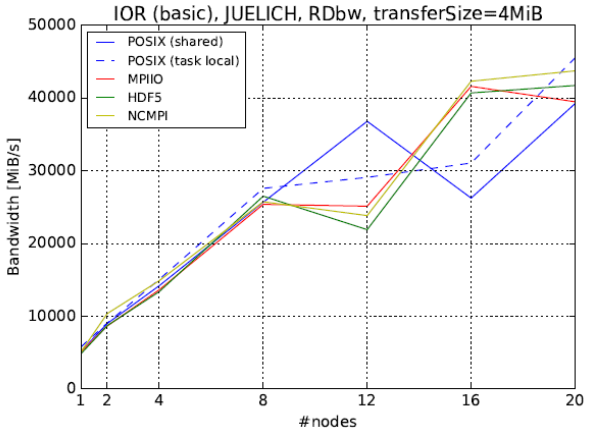


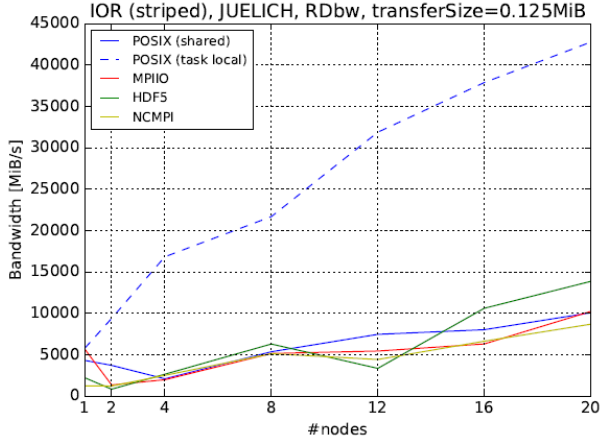
I/O pattern bandwidth

continuous

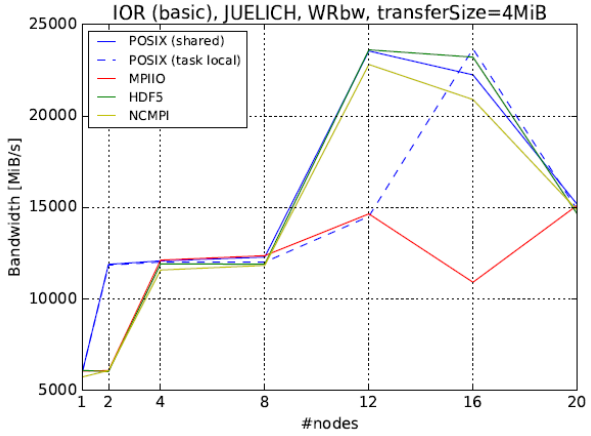
striped

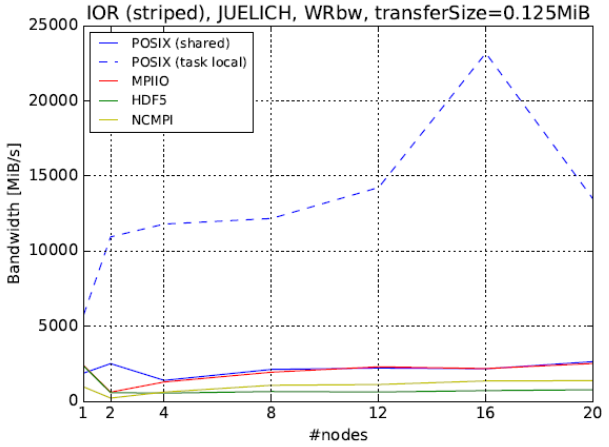
read bandwidth





write bandwidth



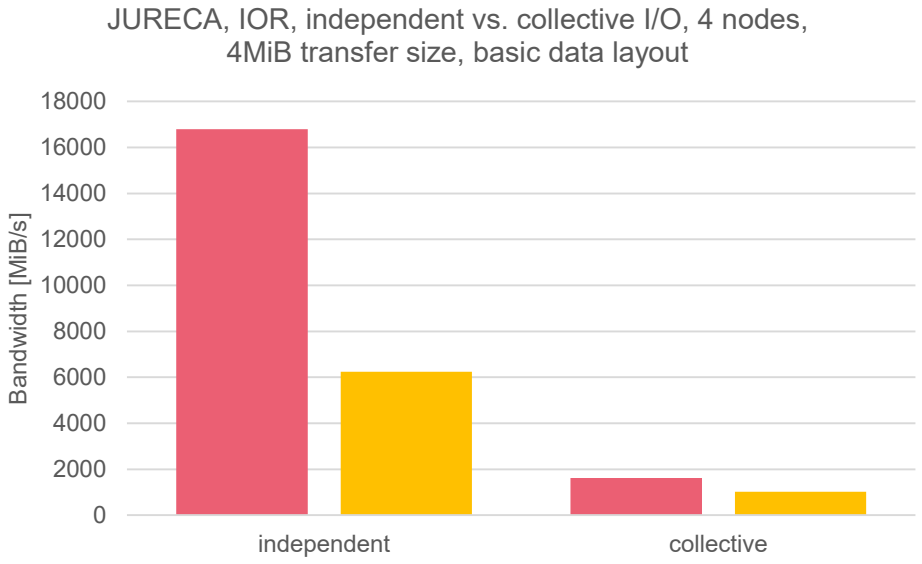


Measurements on JURECA at JSC

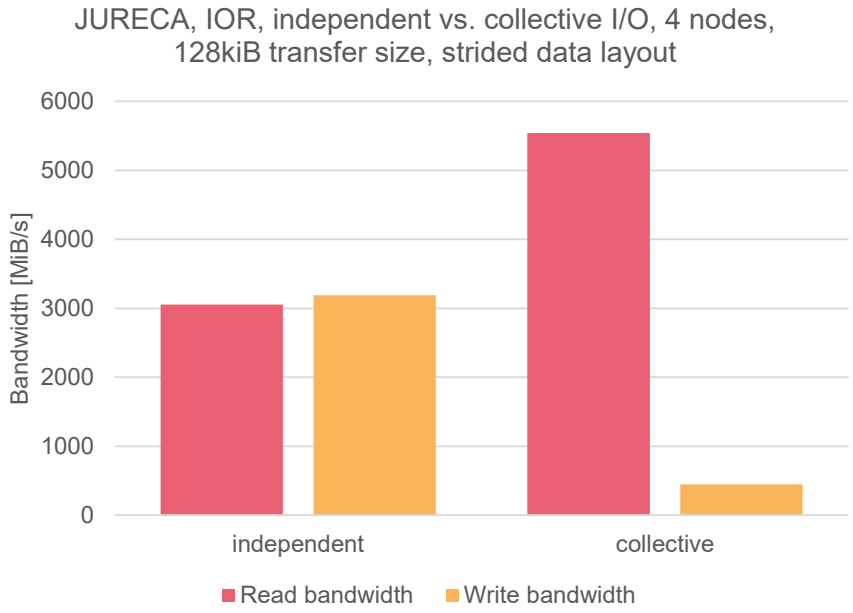
This work was supported by the Energy oriented Centre of Excellence (EoCoE), grant agreement number 676629, funded within the Horizon2020 framework of the European Union.

Collective buffering

- Collective I/O operations not always speed up the general I/O, as more data might be processed than needed



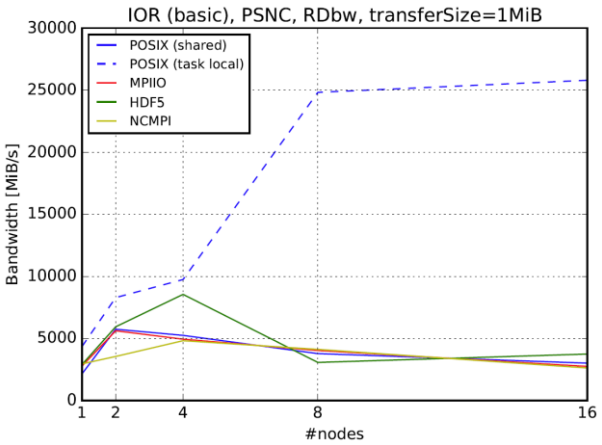
	access size [Byte]	count
MPI-IO	4,194,304	184,320
POSIX	16,777,216	264,574



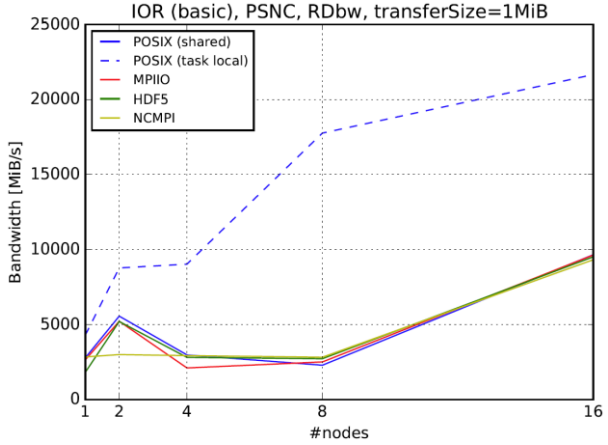
This work was supported by the Energy oriented Centre of Excellence (EoCoE), grant agreement number 676629, funded within the Horizon2020 framework of the European Union.

Filesystem specific options

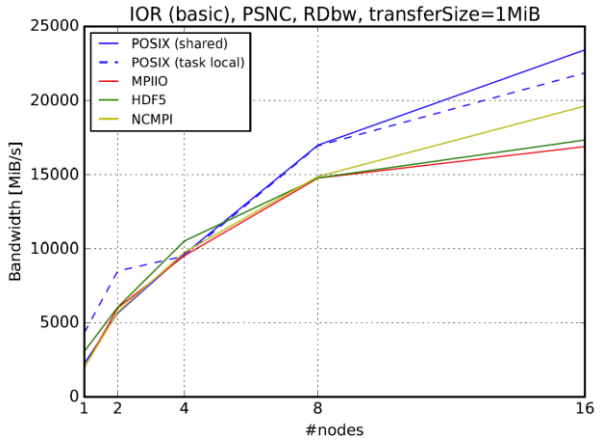
- On Lustre filesystems the user can influence the striping size and the number of involved object storage targets



Default number of OSTs (12) and default strip-size setting (1MiB)



Increased number of OSTs (126)



Increased stripe size to align with the individual amount of data per process (256MiB)

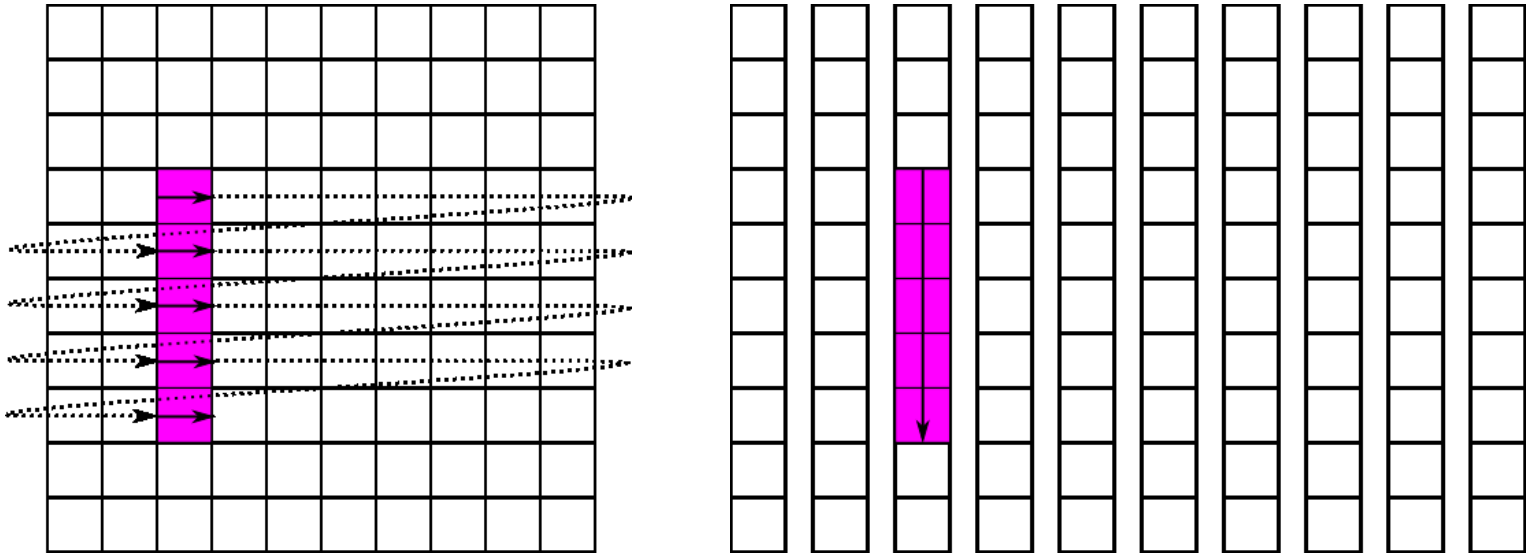
Measurements on Eagle at PSNC

This work was supported by the Energy oriented Centre of Excellence (EoCoE), grant agreement number 676629, funded within the Horizon2020 framework of the European Union.

Performance hints

Chunking

- Contiguous datasets are stored in a single block in the file, chunked datasets are split into multiple chunks which are all stored separately in the file.
- Additional chunk cache is possible



https://www.hdfgroup.org/HDF5/doc/Advanced/Chunking/

More details in the HDF5 part

Performance hints

Compression

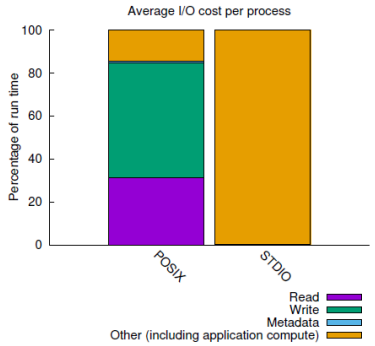
- On the fly compression can help to lower the overall datasize:
- HDF5 and NetCDF4 allows compression within a parallel, collective write commands for chunked datasets
- Gzip (`deflate`) compression available by default (szip can be added on demand)
- Other compression techniques are available by using filters and external plugins: https://support.hdfgroup.org/services/filters.html

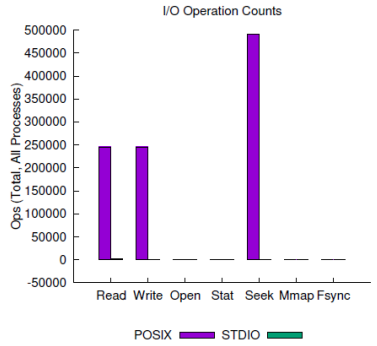
Profiling with Darshan

- I/O profiling tool for parallel applications
  - http://www.mcs.anl.gov/research/projects/darshan/
- Integration by using LD_PRELOAD:
  - `LD_PRELOAD=.../lib/libdarshan.so`

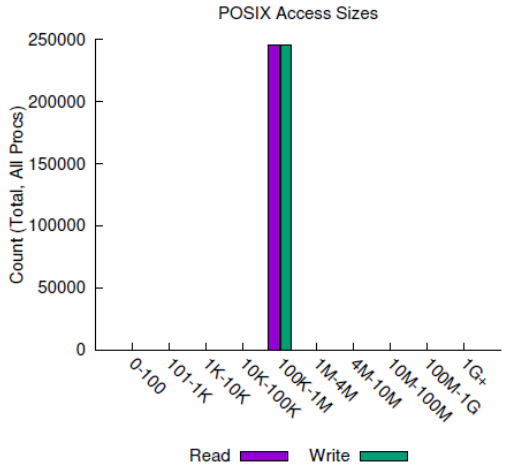
jobid: 4941235	uid: 11901	nprocs: 48	runtime: 10 seconds

I/O performance *estimate* (at the POSIX layer): transferred 37431 MiB at 6692.22 MiB/s
I/O performance *estimate* (at the STDIO layer): transferred 0.0 MiB at 5.27 MiB/s





Profiling with Darshan



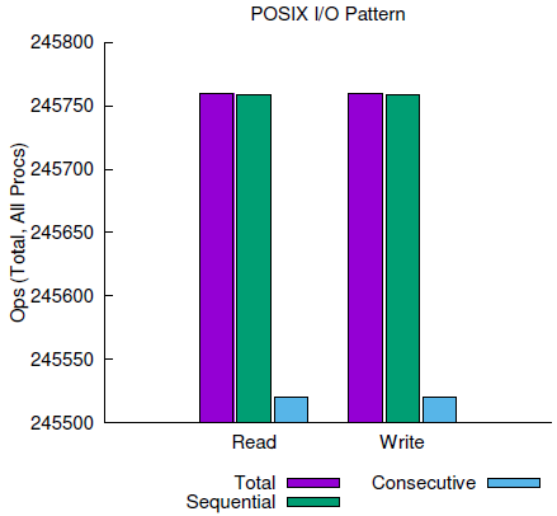
Most Common Access Sizes
(POSIX or MPI-IO)

	access size	count
POSIX	131072	491520

File Count Summary
(estimated by POSIX I/O access offsets)

type	number of files	avg. size	max size
total opened	4	7.6G	30G
read-only files	1	711	711
write-only files	2	1.7K	3.2K
read/write files	1	30G	30G
created files	3	11G	30G

Profiling with Darshan



sequential: An I/O op issued at an offset greater than where the previous I/O op ended.
consecutive: An I/O op issued at the offset immediately following the end of the previous I/O op.

Variance in Shared Files (POSIX and STDIO)

File Suffix	Processes	Fastest			Slowest			σ	
		Rank	Time	Bytes	Rank	Time	Bytes	Time	Bytes
...ehrs/IOR/2_1	48	35	7.507493	1.3G	33	9.180811	1.3G	0.397	0
...or_input.cfg	48	32	0.003404	711	2	0.006366	711	0	0
...<STDOUT>	48	1	0.000000	0	0	0.000392	3.2K	0	455
...<STDERR>	48	1	0.000000	0	0	0.000014	119	0	17