Parallel I/O: Benchmarking and common pitfalls

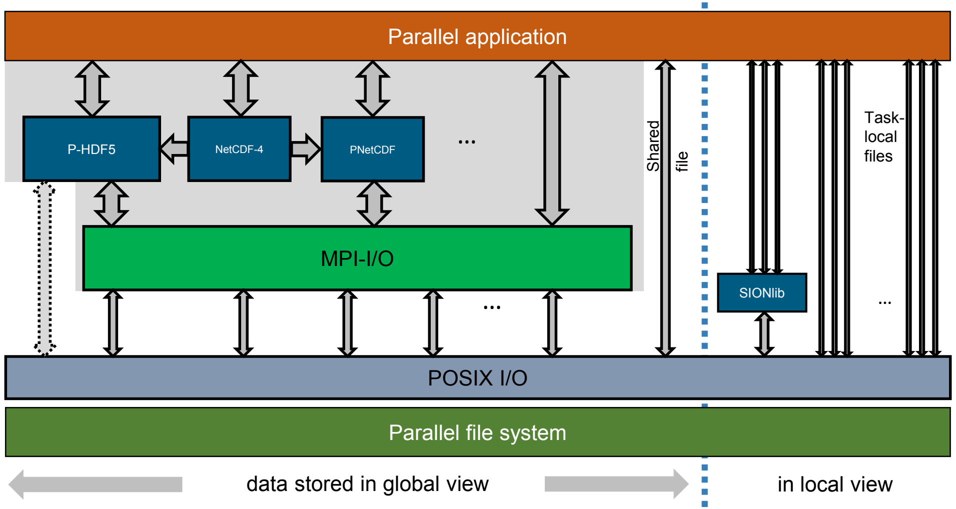
Sebastian Lührs – Jülich Supercomputing Centre
Final conference of the Energy-Oriented Centre of Excellence
Nicosia – Cyprus

Contributors: F. Ambrosino, M. Brzezniak, W. Frings, A. Funel, G. Guarnieri, M. Haefele, F. Iannone, T.Paluszkiewicz, K. Sierocinski

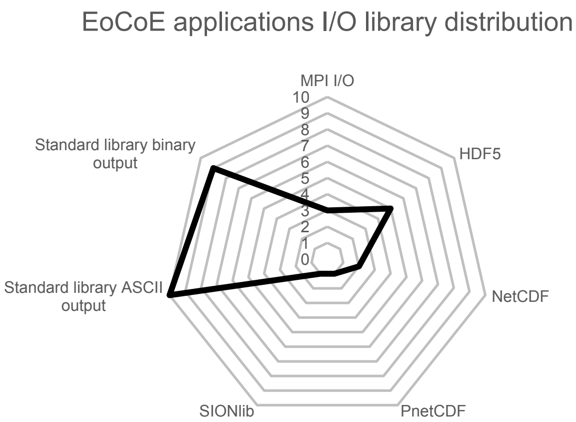
European Commission
Horizon 2020
European Union funding
for Research & Innovation

Parallel I/O usage

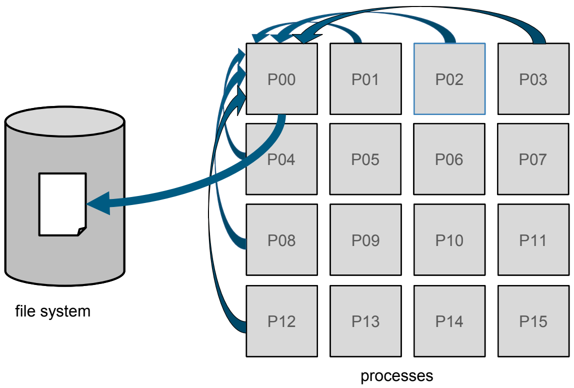
- The I/O behaviour of an HPC application can significantly influence the overall performance.
- With exascale computing also I/O storage and bandwidth demands will increase
- Several different I/O APIs are in use:

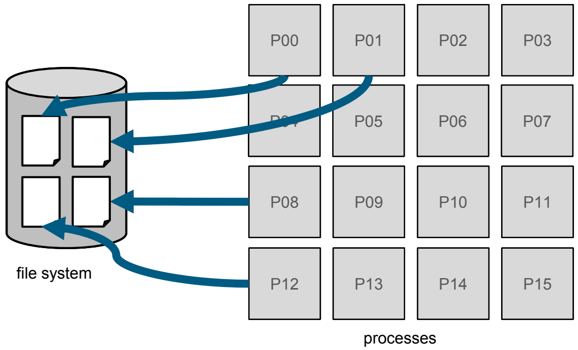


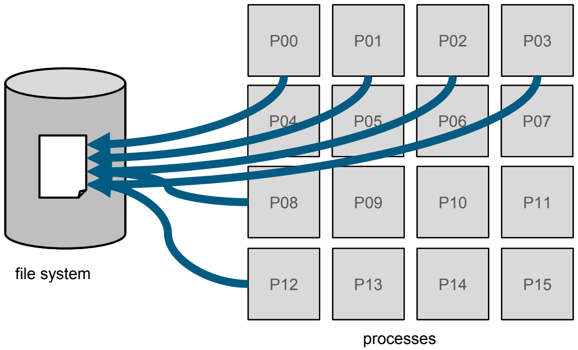
The parallel I/O software stack



Parallel I/O strategies







Serial 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

Task local I/O

- \+ 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

Shared file I/O

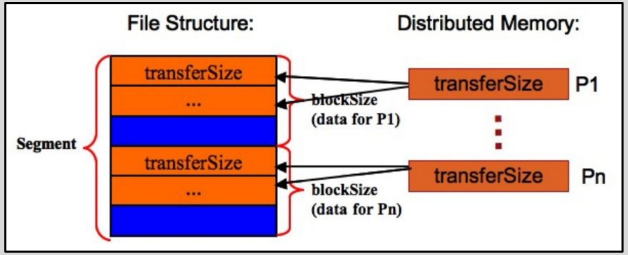
- \+ Number of files is independent of number of processes
- \+ File can be in canonical representation (no post-processing)
- \- Uncoordinated client requests might induce time penalties
- \- File layout may induce false sharing of file system blocks

I/O benchmarking

- Creation of reproducible I/O patterns to investigate API and hardware specific behaviour
- Benchmarks used by the EoCoE I/O benchmarking activity:

IOR

- Well known and established I/O benchmark
- `https://github.com/hpc/ior`
- Supports MPIIO, HDF5, PnetCDF and POSIX
- Allows to validate library overhead, collective vs. independent I/O behaviour and the dependence of different transfer sizes
- IOR file layout:



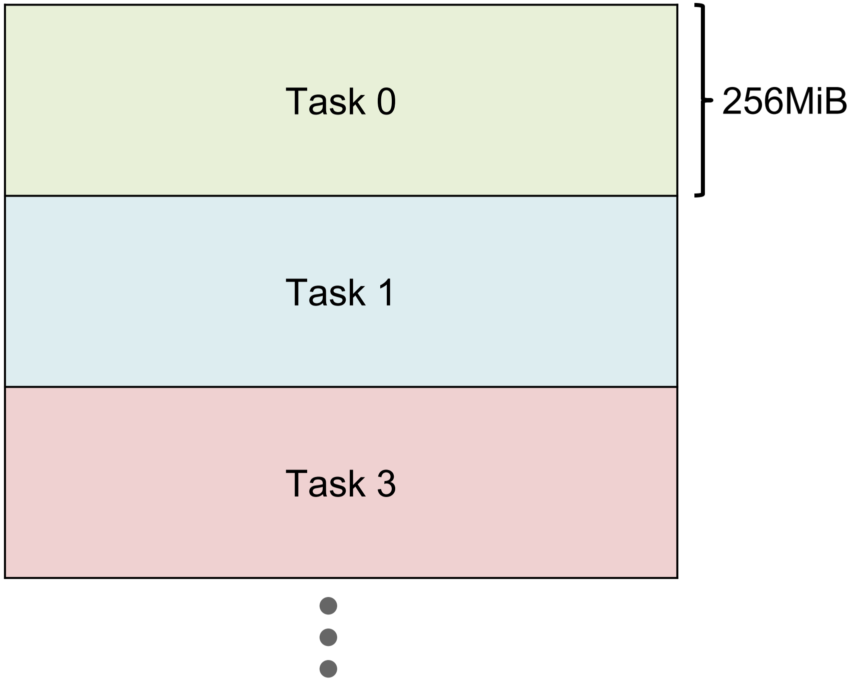
Partest

- Benchmark is part of the SIONlib I/O library
- `www.fz-juelich.de/jsc/sionlib`
- Allows comparison of shared and distributed file I/O
- Supports SIONlib and POSIX
- Simulation of typical checkpointing behaviour

I/O benchmarking: IOR patterns

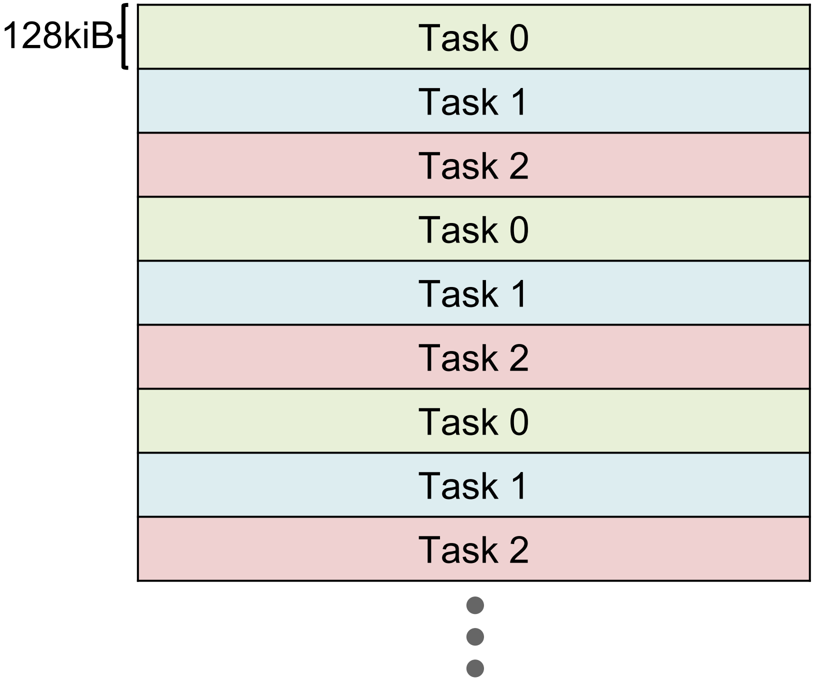
continuous

- Large continuous data blocks for each individual process

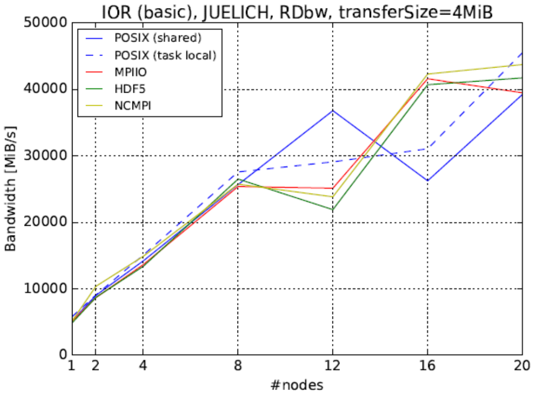
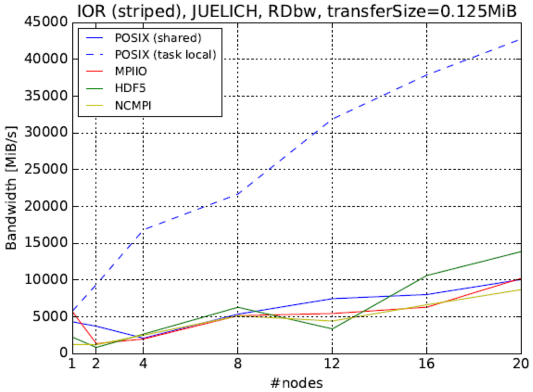
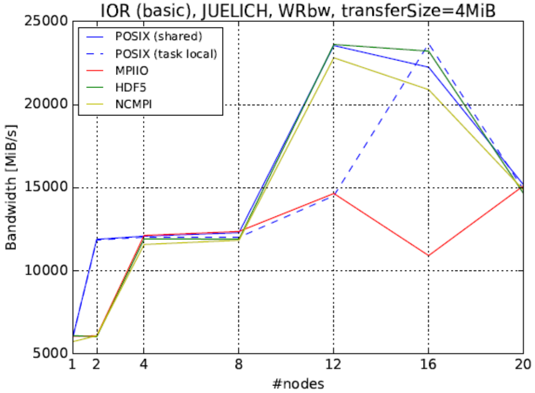
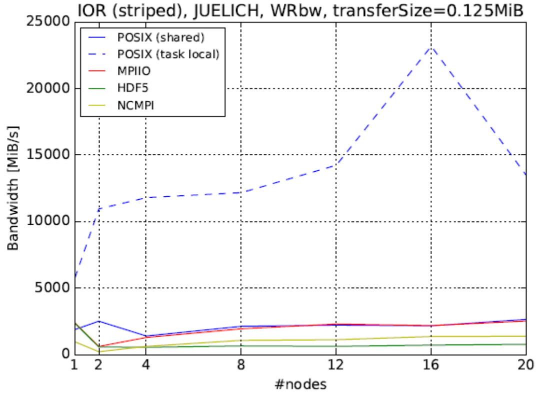


striped

- Pattern often found while handling multi dimensional arrays



I/O benchmarking: Bandwidth

	continuous	striped
read bandwidth		
write bandwidth		

Measurements on JURECA at JSC

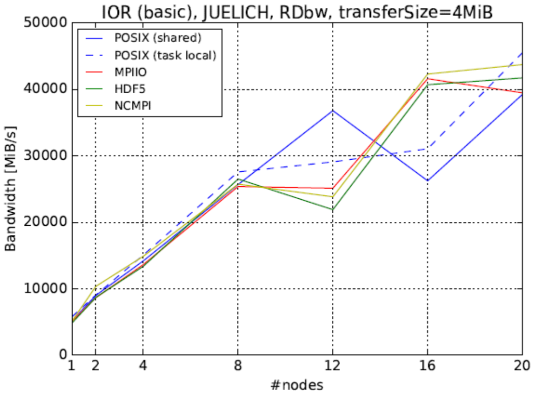
Pitfall 1: Frequent flushing on small blocks

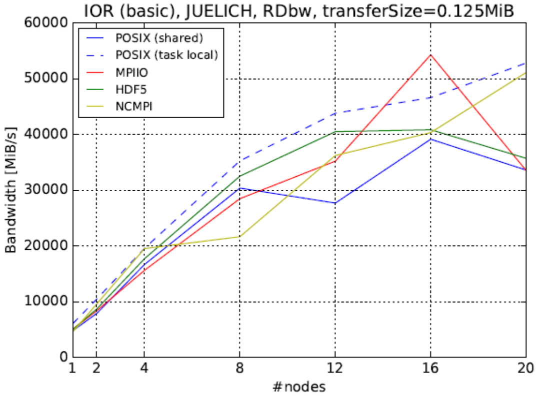
- Modern file systems in HPC have large file system blocks (e.g. 4MB)
- A flush on a file handle forces the file system to perform all pending write operations
- If application writes in small data blocks, the same file system block has to be read and written multiple times
- Performance degradation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

I/O benchmarking: Small transfer size

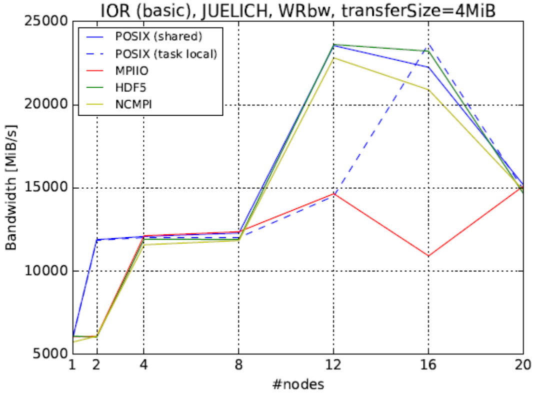
4 MiB | **128 kiB**

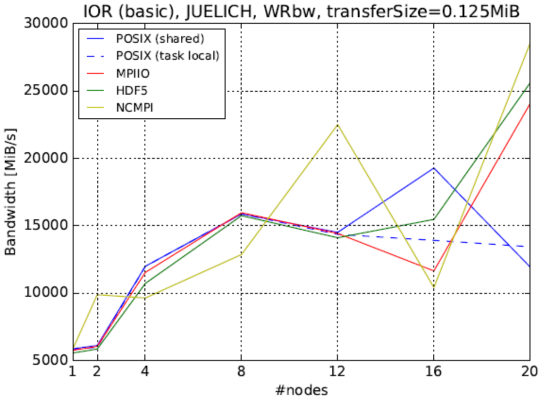
read bandwidth





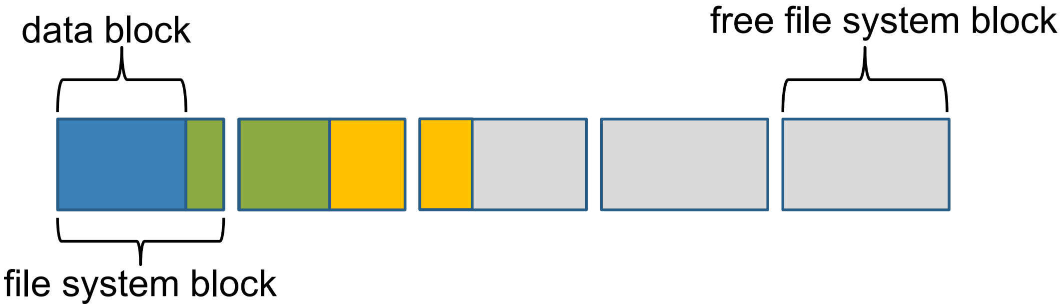
write bandwidth

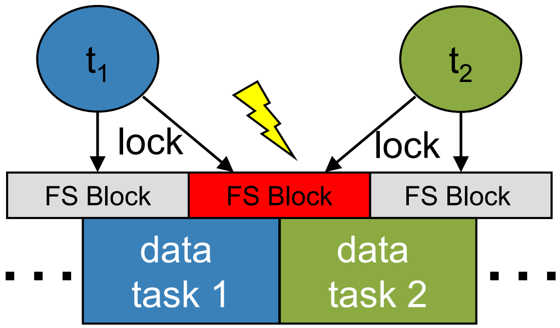




Pitfall 2: False sharing of file system blocks

- 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

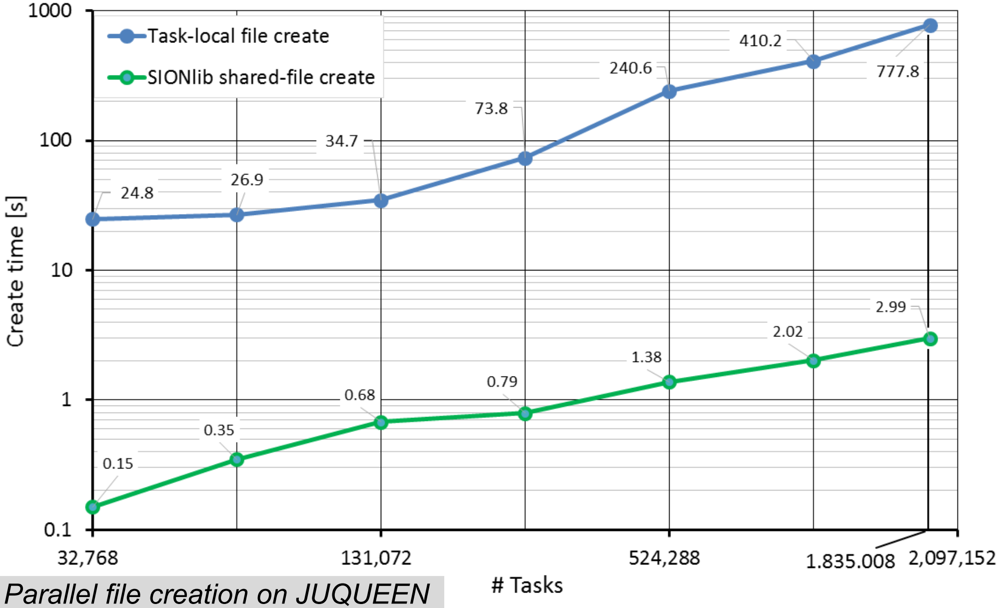




Pitfall 3: Metadata modification

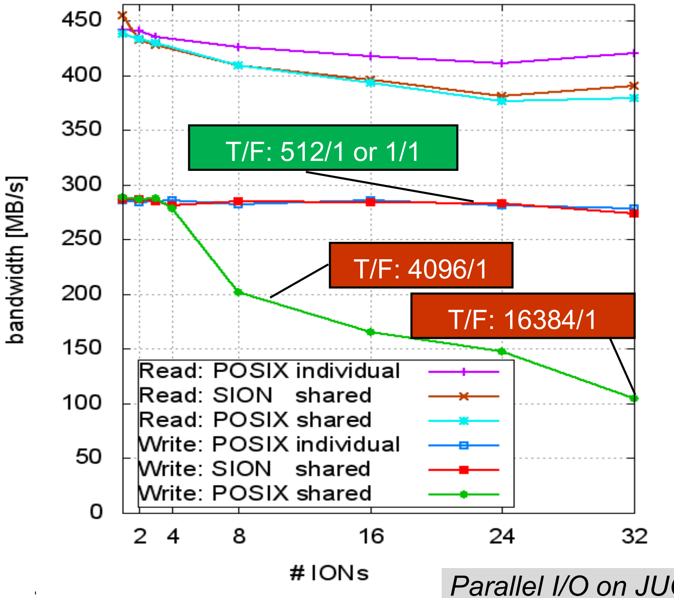
- Metadata operations can serialize I/O operations

Directory metadata



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

File metadata



Parallel I/O on JUGENE
W. Frings

Pitfall 4: Portability

- Data post-processing can be very time consuming
- Portable dataformats (such as HDF5 or NetCDF) allow easy data exchange within application workflows

Endianness

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

Array memory order

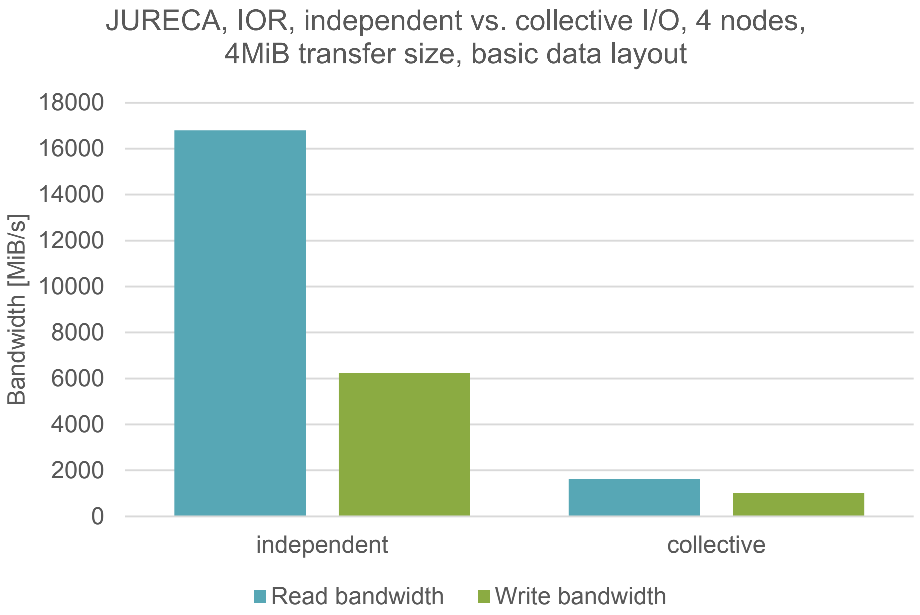
Address	row-major order (e.g. C/C++)	column-major order (e.g. Fortran)
1000	1	1
1001	2	4
1002	3	7
1003	4	2
1004	5	5
…	…	…

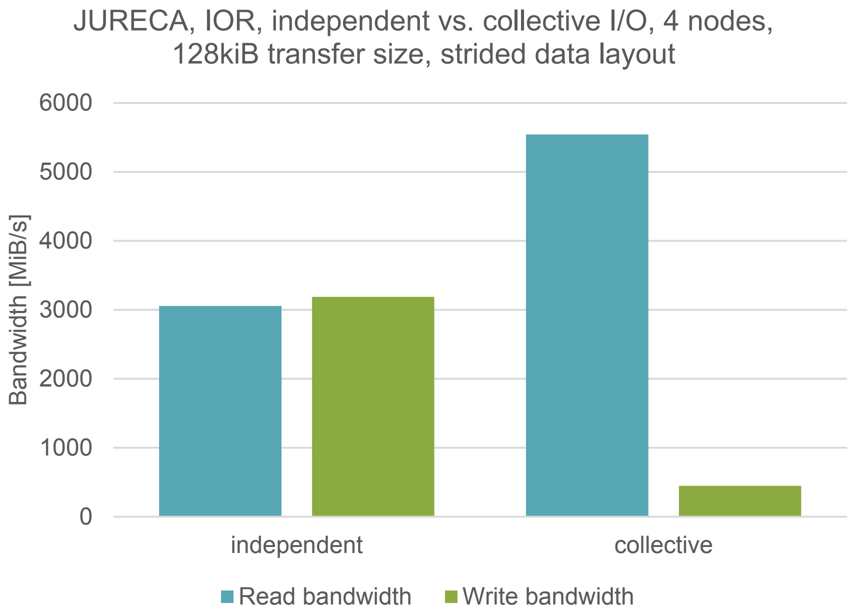
Avoiding pitfalls: General remarks

- Large continuous data chunks show better performance
  - Task local files automatically avoid false sharing of filesystem blocks and file specific metadata problems
  - API specific mechanics allow to rebuild continuous data chunks (e.g. collective buffering or HDF5 chunking)
- Portable data formats allow a global data view and avoid portability problems
- Usage of intermediate cache infrastructure or local flash storage devices

Avoiding pitfalls: Collective buffering

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

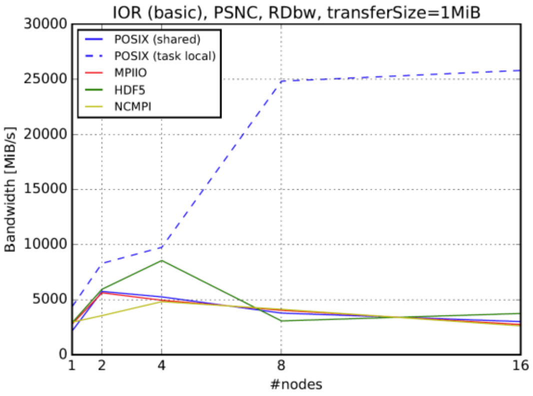




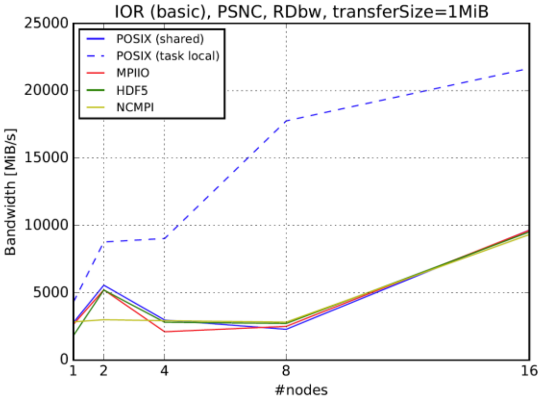
	access size [Byte]	count
MPI-IO	4,194,304	184,320
POSIX	16,777,216	264,574

Avoiding pitfalls: Filesystem specifc options

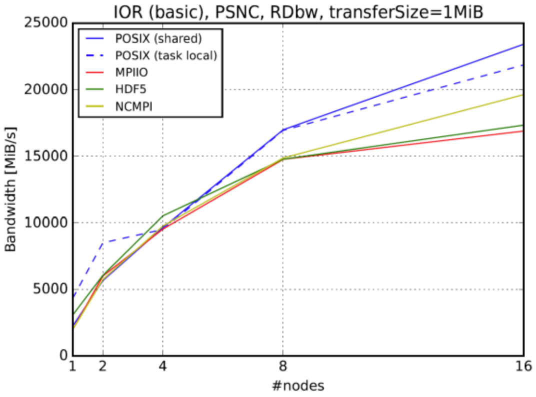
- On Lustre filesystems the user can influence the striping size and the number of involved object storage targets



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

- More details and results on the EoCoE I/O benchmarking activity can be found in deliverable D1.12 of the EoCoE project

Thank you for your attention.

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.