Parallel I/O for High Performance Computing

Matthieu Haefele

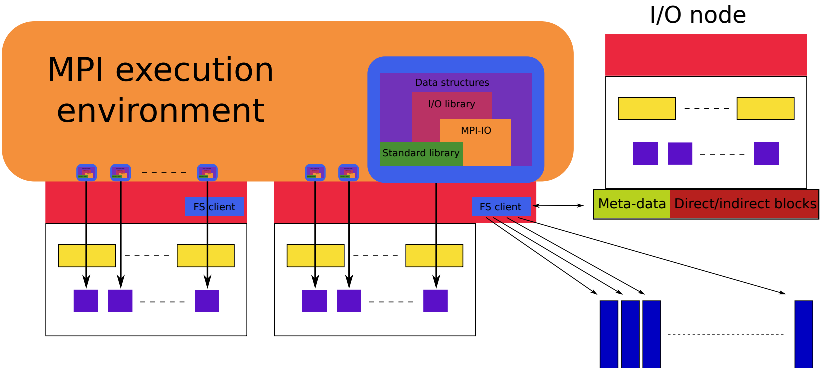
High Level Support Team
Max-Planck-Institut für Plasmaphysik, München, Germany

Autrans, 26-30 Septembre 2011,
École d'été Masse de données : structuration, visualisation

Outline

1. Different IO methods
   - POSIX
   - MPI-IO
   - Parallel HDF5
2. Benchmarks
   - Test case
   - Results
   - Conclusions
3. Focus on MPI-IO and HDF5 API
   - HDF5
   - MPI-IO

The whole hardware/software "stack"



Multi-file method

Each MPI process writes its own file

- Pure “non-portable” binary files
- A single distributed data is spread out in different files
- The way it is spread out depends on the number of MPI processes

⇒ More work at post-processing level

- Files not portable
- Files not self documented
- Very easy to implement
- Very efficient

MPI gather and single-file method

A collective MPI call is first performed to gather the data on one MPI process. Then, this process writes a single file

- Single pure “non-portable” binary file
- The memory of a single node can be a limitation
- Files not portable
- Files not self documented
- Single resulting file

MPI-IO concept

- I/O part of the MPI specification
- Provide a set of read/write methods
- Allow one to describe how a data is distributed among the processes (thanks to MPI derived types)
- MPI implementation takes care of actually writing a single contiguous file on disk from the distributed data
- Result is identical as the gather + POSIX file

MPI-IO performs the gather operation within the MPI implementation

MPI-IO

- No more memory limitation
- Single resulting file
- File not portable
- Files not self documented
- Definition of MPI derived types

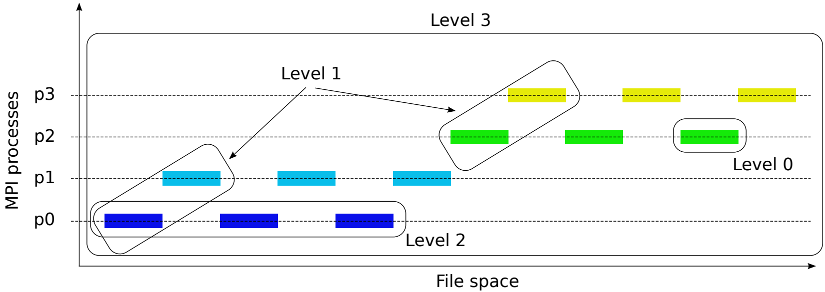
MPI-IO API

Level 0 — Level 1

Positioning	Synchronism	Coordination: Non collective	Coordination: Collective
Explicit offsets	Blocking	MPI_FILE_READ_AT MPI_FILE_WRITE_AT	MPI_FILE_READ_AT_ALL MPI_FILE_WRITE_AT_ALL
Explicit offsets	Non blocking & Split call	MPI_FILE_IREAD_AT MPI_FILE_IWRITE_AT	MPI_FILE_READ_AT_ALL_BEGIN MPI_FILE_READ_AT_ALL_END MPI_FILE_WRITE_AT_ALL_BEGIN MPI_FILE_WRITE_AT_ALL_END
Individual file pointers	Blocking	MPI_FILE_READ MPI_FILE_WRITE	MPI_FILE_READ_ALL MPI_FILE_WRITE_ALL
Individual file pointers	Non blocking & Split call	MPI_FILE_IREAD MPI_FILE_IWRITE	MPI_FILE_READ_ALL_BEGIN MPI_FILE_READ_ALL_END MPI_FILE_WRITE_ALL_BEGIN MPI_FILE_WRITE_ALL_END
Shared file pointers	Blocking	MPI_FILE_READ_SHARED MPI_FILE_WRITE_SHARED	MPI_FILE_READ_ORDERED MPI_FILE_WRITE_ORDERED
Shared file pointers	Non blocking & Split call	MPI_FILE_IREAD_SHARED MPI_FILE_IWRITE_SHARED	MPI_FILE_READ_ORDERED_BEGIN MPI_FILE_READ_ORDERED_END MPI_FILE_WRITE_ORDERED_BEGIN MPI_FILE_WRITE_ORDERED_END

Level 2 — Level 3

MPI-IO level illustration

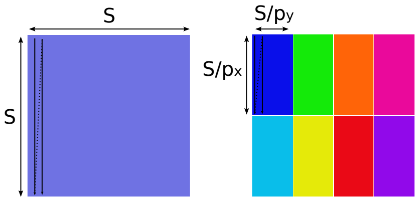


Parallel HDF5

- Built on top of MPI-IO
- Must follow some restrictions to enable underlying collective calls of MPI-IO
- From the programmation point of view, only few parameters has to be given to HDF5 library
- Data distribution is described thanks to hdf5 hyperslices
- Result is a single portable HDF5 file

- Single portable file
- Self documented file
- Maybe some performance issues
- Add library dependancy
- API has to be mastered

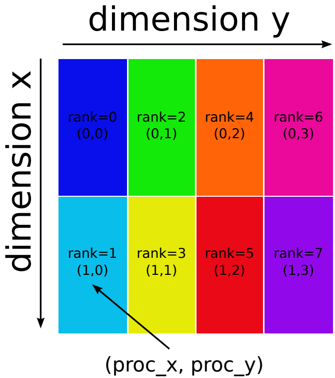
Test case



Let us consider:

- A 2D structured array
- The array is of size $S \times S$
- A block-block distribution is used
- With $P = p_x p_y$ cores

Exercice 4



Let us consider:

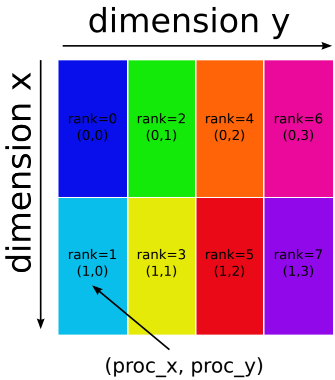
- A 2D structured array
- x contiguous in memory
- x represented vertically
- Fortran language convention

$\Rightarrow$ Dimension x is index=

$\Rightarrow$ Dimension y is index=

```
count(1)  =
count(2)  =
start(1)  =
start(2)  =
stride(1)  =
stride(2)  =
```

Solution 4



Let us consider:

- A 2D structured array
- x contiguous in memory
- x represented vertically
- Fortran language convention

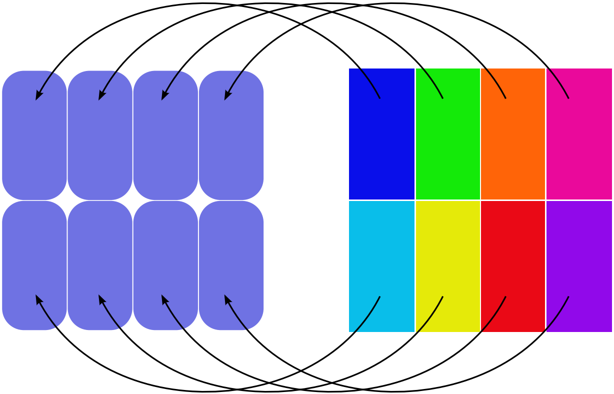
$\Rightarrow$ Dimension x is index=1

$\Rightarrow$ Dimension y is index=2

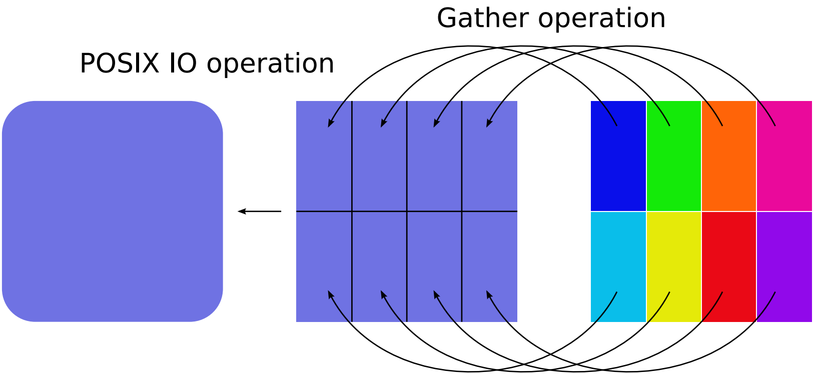
```
count(1) = S/px
count(2) = S/py
start(1) = proc_x * count(1)
start(2) = proc_y * count(2)
stride(1) = 1
stride(2) = 1
```

Multiple POSIX files

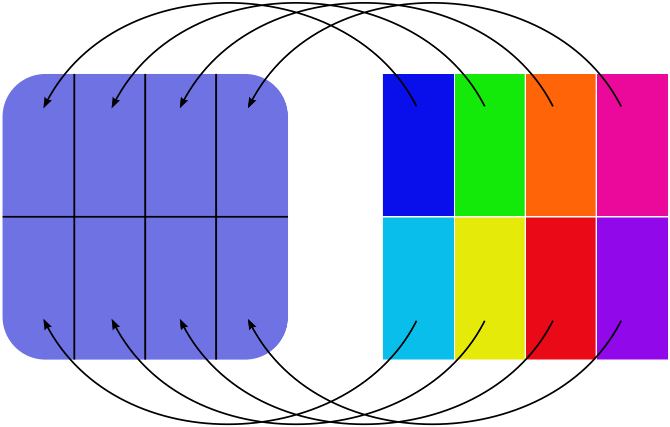
POSIX IO operations



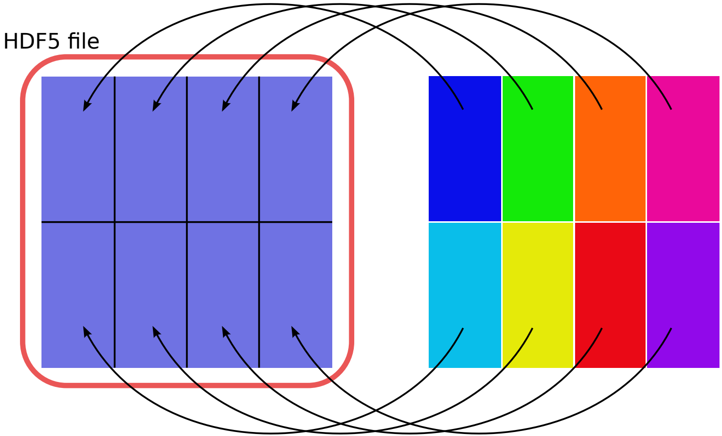
Gather + single POSIX file



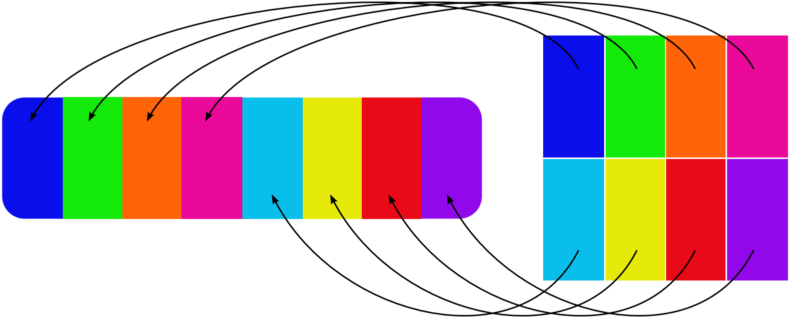
MPI-IO



Parallel HDF5



MPI-IO chunks



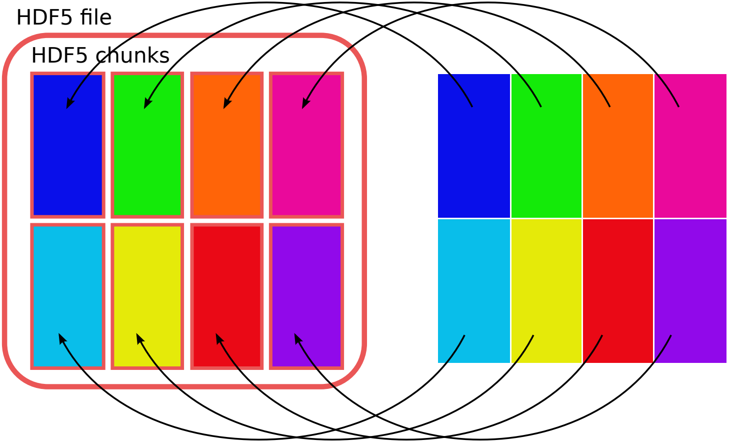
MPI-IO chunks

- One local array contiguous in an MPI process is contiguous in the file

⇒ More work at post-processing level like in the multi-file method

⇒ Concurrent accesses reduction

Parallel HDF5 chunks



Parallel HDF5 chunks

- One local array contiguous in an MPI process is contiguous in the file

$\Rightarrow$ Concurrent accesses reduction

$\Rightarrow$ HDF5 takes care of the chunks himself !!

Benchmark realised on two different machines

High Performance Computer For Fusion (HPC-FF)

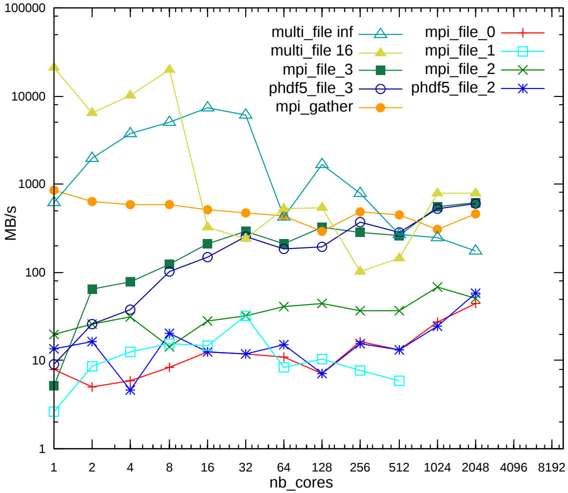
- Located in Jülich Supercomputing Center (JSC)
- Bull machine
- 8640 INTEL Xeon Nehalem-EP cores
- Lustre file system

VIP machine

- Located in Garching Rechenzentrum (RZG)
- IBM machine
- 6560 POWER6 cores
- GPFS file system

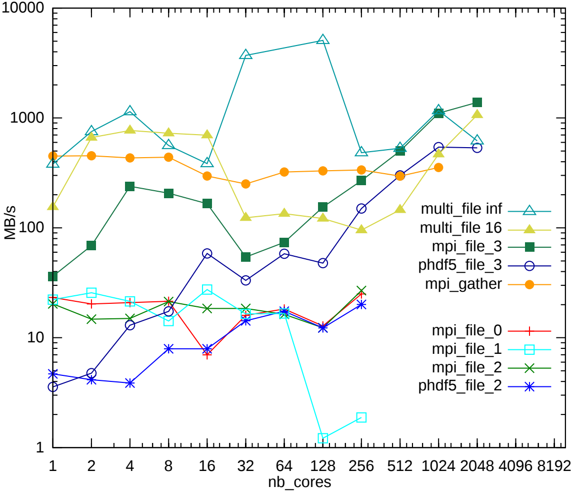
Weak scaling on VIP

4MB to export per MPI task



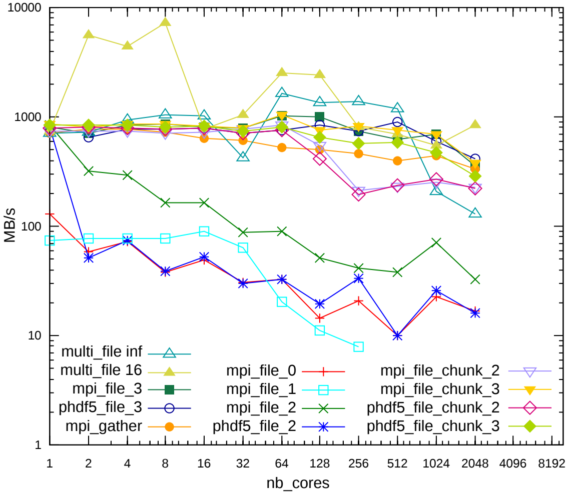
Weak scaling on HPC-FF

4MB to export per MPI task



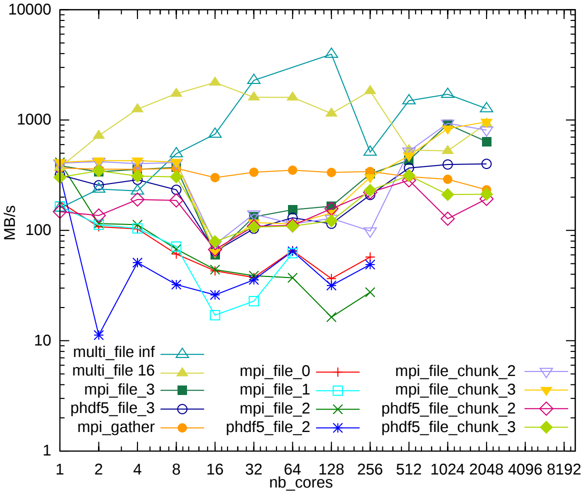
Strong scaling on VIP

A total of 8GB to export



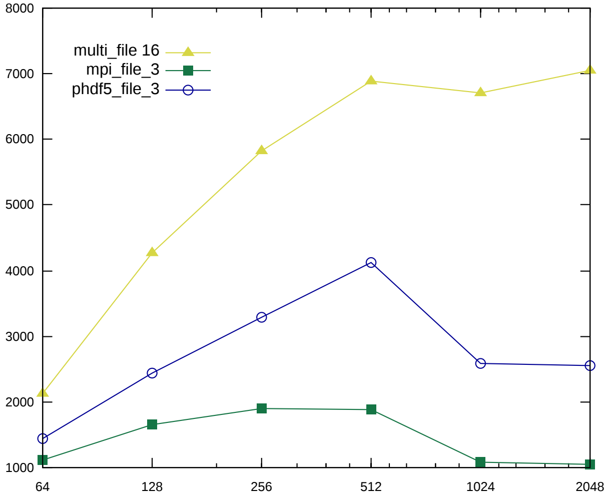
Strong scaling on HPC-FF

A total of 8GB to export



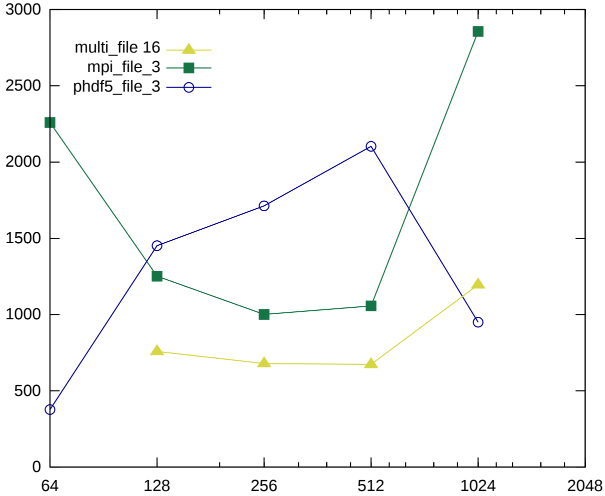
Strong scaling on VIP

A total of 256GB to export



Strong scaling on HPC-FF

A total of 256GB to export



Conclusions

1. The view mechanism should be prefered to MPI-IO explicit offsets
2. For small file size, POSIX interface is still more efficient
3. Gather + single POSIX file is still a good choice
4. To use HDF5 in the context of HPC makes sense
5. Additional implementation work for chunking is not worth
6. Multi-file POSIX method gives very good performance on 1K cores. Will it still be the case on 10K, 100K cores ?

Full report here

http://www.efda-hlst.eu/training/HLST_scripts/comparison-of-different-methods-for-performing-parallel-i-o/at_download/file

http://edoc.mpg.de/display.epl?mode=doc&id=498606

HDF5 implementation

```
INTEGER(HSIZE_T) :: array_size(2), array_subsize(2), array_start(2)
INTEGER(HID_T) :: plist_id1, plist_id2, file_id, filespace, dset_id, memspace
array_size(1) = S
array_size(2) = S
array_subsize(1) = local_nx
array_subsize(2) = local_ny
array_start(1) = proc_x * array_subsize(1)
array_start(2) = proc_y * array_subsize(2)

!Allocate and fill the tab array

CALL h5open_f(ierr)
CALL h5pcreate_f(H5P_FILE_ACCESS_F, plist_id1, ierr)
CALL h5pset_fapl_mpio_f(plist_id1, MPI_COMM_WORLD, MPI_INFO_NULL, ierr)
CALL h5fcreate_f('res.h5', H5F_ACC_TRUNC_F, file_id, ierr, access_prp = plist_id1)

! Set collective call
CALL h5pcreate_f(H5P_DATASET_XFER_F, plist_id2, ierr)
CALL h5pset_dxpl_mpio_f(plist_id2, H5FD_MPIO_COLLECTIVE_F, ierr)

CALL h5screate_simple_f(2, array_size, filespace, ierr)
CALL h5screate_simple_f(2, array_subsize, memspace, ierr)

CALL h5dcreate_f(file_id, 'pi_array', H5T_NATIVE_REAL, filespace, dset_id, ierr)
CALL h5sselect_hyperslab_f (filespace, H5S_SELECT_SET_F, array_start, array_subsize, ierr)
CALL h5dwrite_f(dset_id, H5T_NATIVE_REAL, tab, array_subsize, ierr, memspace, filespace, plist_id2)

! Close HDF5 objects
```

MPI-IO implementation

```
INTEGER :: array_size(2), array_subsize(2), array_start(2)
INTEGER :: myfile, filetype
array_size(1) = S
array_size(2) = S
array_subsize(1) = local_nx
array_subsize(2) = local_ny
array_start(1) = proc_x * array_subsize(1)
array_start(2) = proc_y * array_subsize(2)

!Allocate and fill the tab array

CALL MPI_TYPE_CREATE_SUBARRAY(2, array_size, array_subsize, array_start, &
                    MPI_ORDER_FORTRAN, MPI_REAL, filetype, ierr)
CALL MPI_TYPE_COMMIT(filetype, ierr)

CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'res.bin', MPI_MODE_WRONLY+MPI_MODE_CREATE, MPI_INFO_
                     myfile, ierr)

CALL MPI_FILE_SET_VIEW(myfile, 0, MPI_REAL, filetype, "native", MPI_INFO_NULL, ierr)

CALL MPI_FILE_WRITE_ALL(myfile, tab, local_nx * local_ny, MPI_REAL, status, ierr)

CALL MPI_FILE_CLOSE(myfile, ierr)
```

MPI-IO

Compared to the HDF5 dataspace concept:

- *MPI_TYPE_CREATE_SUBARRAY* plays the role of a dataspace modified by *H5Sselect_hyperslab*
- *MPI_FILE_SET_VIEW* plays the role of the dataspace that describes the portion of the **dataset** that has to be written during an *H5Dwrite*.
- *MPI_FILE_WRITE_ALL* plays the role of the *H5Dwrite* and the dataspace that describes the portion of the **memory** that has to be written

MPI-IO

```
MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes, array_of_subsizes,\
  array_of_starts, order, oldtype, newtype)
```

- IN *ndims* number of array dimensions (positive integer)
- IN *array_of_sizes* number of elements of type oldtype in each dimension of the full array (array of positive integers)
- IN *array_of_subsizes* number of elements of type oldtype in each dimension of the subarray (array of positive integers)
- IN *array_of_starts* starting coordinates of the subarray in each dimension (array of nonnegative integers)
- IN *order* array storage order flag (state)
- IN *oldtype* array element datatype (handle)
- OUT *newtype* new datatype (handle)

MPI-IO

```
MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info)
```

- INOUT *fh* file handle (handle)
- IN *disp* displacement (integer)
- IN *etype* elementary datatype (handle)
- IN *filetype* filetype (handle)
- IN *datarep* data representation (string)
- IN *info* info object (handle)

MPI-IO

```
MPI_FILE_WRITE_ALL(fh, buf, count, datatype, status)
```

- INOUT *fh* file handle (handle)
- IN *buf* initial address of buffer (choice)
- IN *count* number of elements in buffer (integer)
- IN *datatype* datatype of each buffer element (handle)
- OUT *status* status object (Status)

Hands on

1. Understand the MPI version of the pjacobi program
2. Implement parallel IO to export the **v** array
   - with HDF5
   - with MPI-IO
3. Try to visualize the result in VisIt thanks to XDMF