Parallel I/O for High Performance Computing

Matthieu Haefele

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

Lyon, 13 Janvier 2011

Outline

1. Introduction and Prerequisites
   - HPC machine architecture
   - MPI basics
   - Parallel file system
2. Methods for parallel I/O
   - start, stride, count, block
   - POSIX
   - MPI-IO
   - Parallel HDF5
3. Benchmarks
   - Test case
   - Results
   - Conclusions

HPC machine architecture

An HPC machine is composed of processing elements or cores which

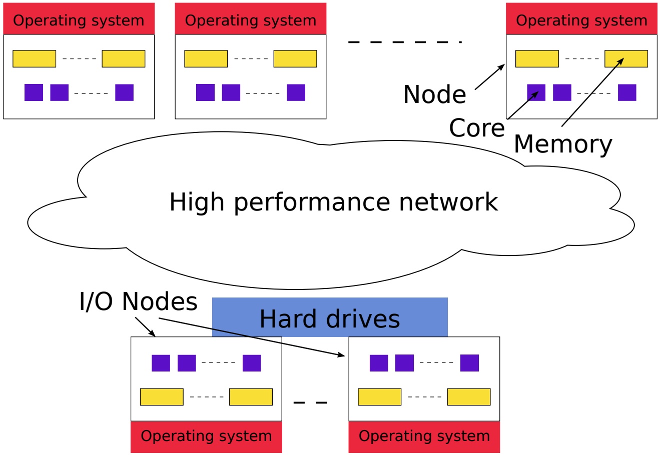
- Can access a central memory
- Can communicate through a high performance network
- Are connected to a high performance storage system

Until now, two major families of HPC machines existed:

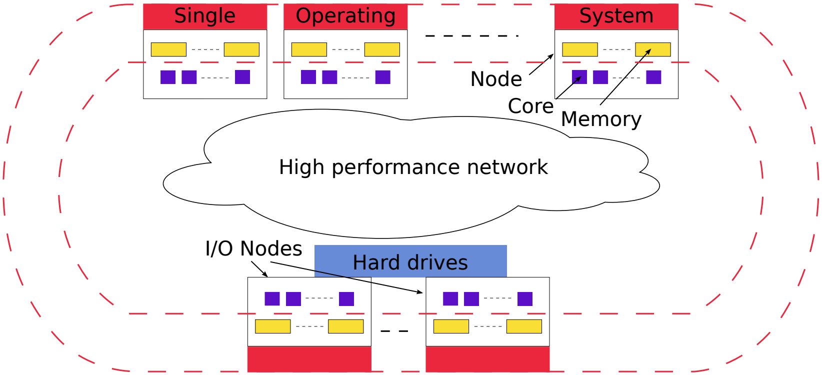
- Shared memory machines
- Distributed memory machines

New architectures like GPGPUs, Cell, FPGAs, . . . are not covered here

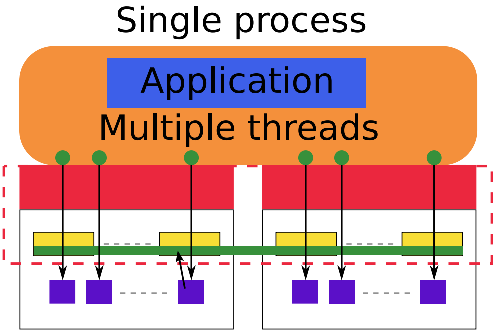
Distributed memory machines



Shared memory machines

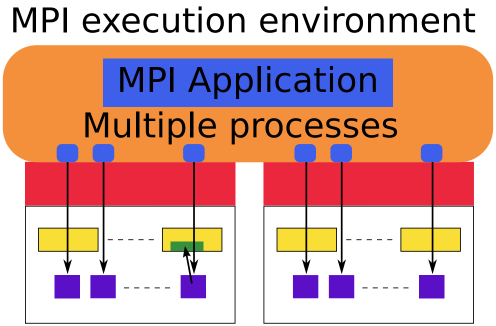


An application within shared memory machines



- One application ⇔ One process of the unique operating system
- Threads of the application are attached to cores
- Each thread can have access to the entire memory of the process

An application within distributed memory machines



- One MPI application $\Leftrightarrow$ Multiple processes of the multiple operating systems
- Processes of the MPI application are attached to cores
- Each process can have access only to its own memory

MPI: Message Passing Interface

MPI is an Application Programming Interface

- Defines a standard for developping parallel applications
- Several implementations exists (openmpi, mpich, IBM, ParTec. . .)

It is composed of

- A parallel execution environment
- A library to link the application with

MPI execution steps

1. The parallel environment is launched with the application and a list of hosts as parameter
2. The application begins its execution as a single process
3. The application calls MPI_INIT function
4. The parallel environment creates the required number of processes on the specified hosts
5. Each process receives its identification number (rank)

From the development point of view, all the parallelization work is parametrized by this rank number

MPI Hello world

```
#include <stdio.h>
#include <mpi.h>

int main (int argc, char **argv)
{
  int rank, size;
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  printf( "Hello world from process %d of %d\n", rank, size );
  MPI_Finalize();
  return 0;
}
```

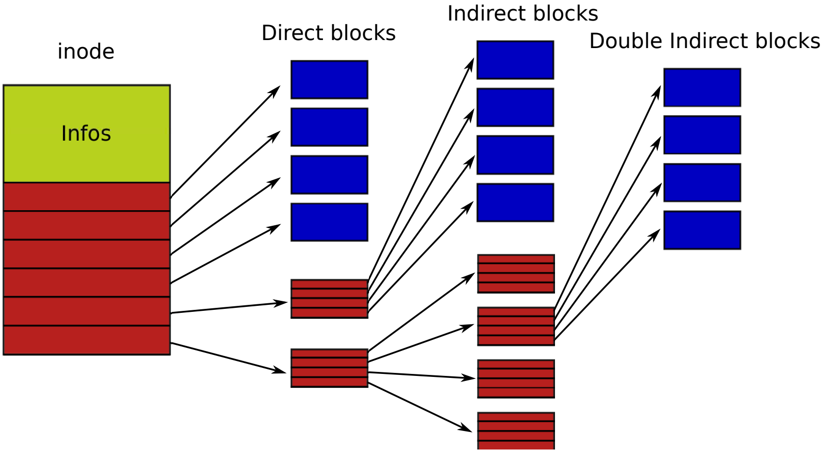
```
mhaef@hlst1:~/afs/dev/mpi_test$ mpicc hello_world.c
mhaef@hlst1:~/afs/dev/mpi_test$ mpirun -np 4 a.out
Hello world from process 2 of 4
Hello world from process 3 of 4
Hello world from process 0 of 4
Hello world from process 1 of 4
mhaef@hlst1:~/afs/dev/mpi_test$
```

MPI communications

Four classes of communications

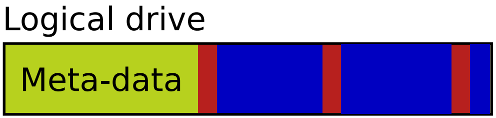
- **Collective**: all processes belonging to a same MPI communicator communicates together according to a defined pattern (scatter, gather, reduce, . . .)
- **Point-to-Point**: one process sends a message to another one (send, receive)
- For both Collective or Point-to-Point, **blocking and non-blocking** functions are available

inode pointer structure (ext3)

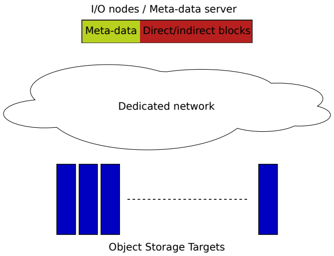


“Serial” file system

Meta-data, block address and file blocks are stored a single logical drive with a “serial” file system

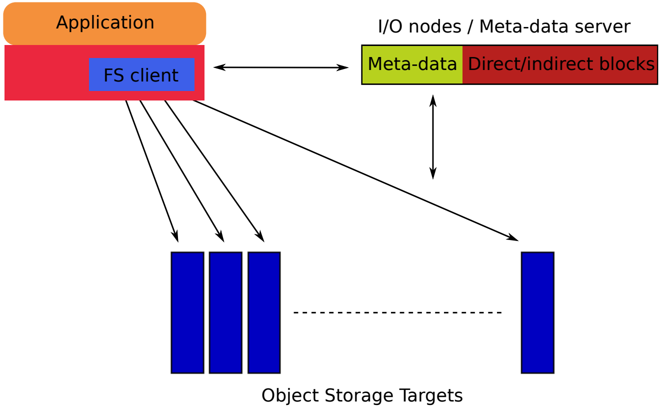


Parallel file system architecture



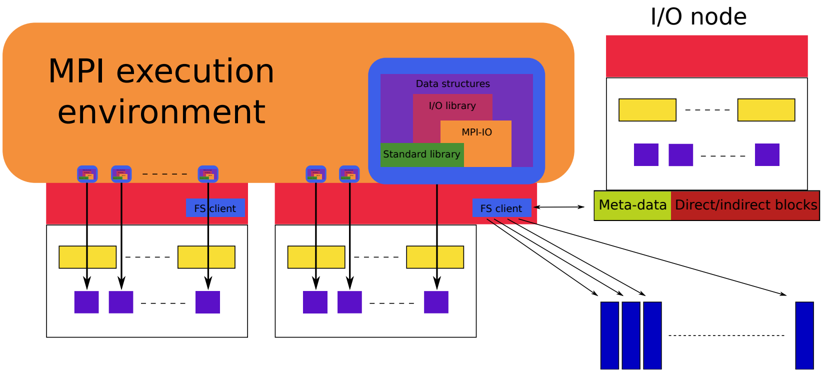
- Meta-data and file blocks are stored on separate devices
- Several devices are used
- Bandwidth is aggregated
- A file is **striped** across different object storage targets.

Parallel file system usage



The file system client gives to the application the view of a “serial” file system

Let us put everything together



The high performance I/O issue

- The software/hardware stack between the application data structures and the object storage targets is large
- Several methods are available
- Every methods are not efficient

Concept

Considering a n-dimensional array, start, stride, count and block are arrays of size n that describe a subset of the original array

- **start**: Starting location for the hyperslab (default 0)
- **stride**: The number of elements to separate each element or block to be selected (default 1)
- **count**: The number of elements or blocks to select along each dimension
- **block**: The size of the block (default 1)

Conventions for the examples

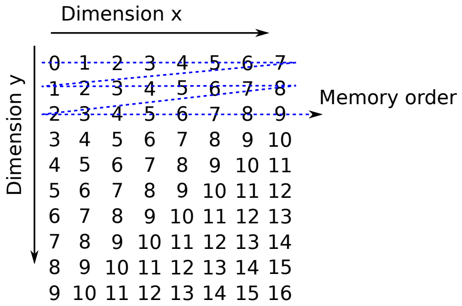
We consider:

- A 2D array $f(x,y)$ with $N_x = 8, N_y = 10$
- Dimension x is the dimension contiguous in memory
- Graphically, the x dimension is represented horizontal
- Language C convention is used for indexing the dimensions

$\Rightarrow$ Dimension y is index=0

$\Rightarrow$ Dimension x is index=1

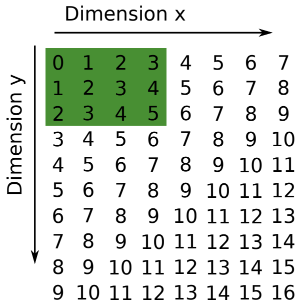
Graphical representation

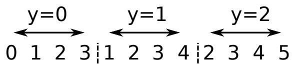


```
int start[2], stride[2], count[2], block[2];
 start[0] = 0;   start[1] = 0;
stride[0] = 1;  stride[1] = 1;
 block[0] = 1;   block[1] = 1;
```

Illustration for count parameter

Selection of the box $((0,0),(3,2))$

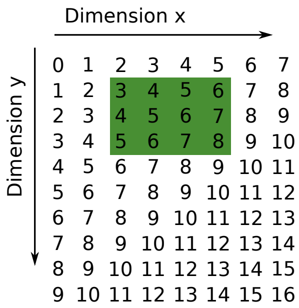


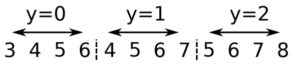


```
count[0] = 3;   count[1] = 4;
```

Illustration for start parameter

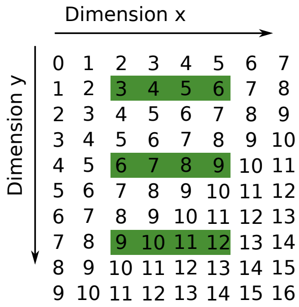
Selection of the box $((2,1),(5,3))$

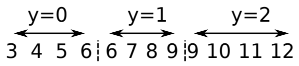




```
start[0] = 1;  start[1] = 2;
count[0] = 3;  count[1] = 4;
```

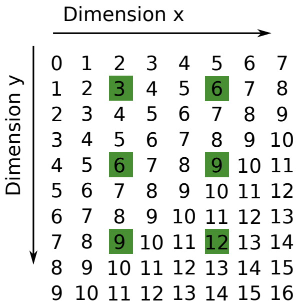
Illustration for stride parameter



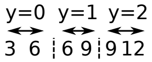


```
start[0] = 1;   start[1] = 2;
count[0] = 3;   count[1] = 4;
stride[0] = 3;  stride[1] = 1;
```

Illustration for stride parameter

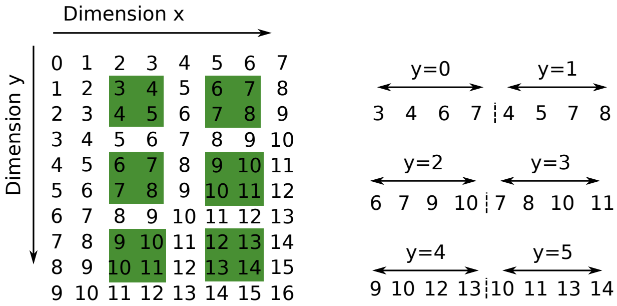


0	1	2	3	4	5	6	7
1	2	**3**	4	5	**6**	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	**6**	7	8	**9**	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	**9**	10	11	**12**	13	14
8	9	10	11	12	13	14	15
9	10	11	12	13	14	15	16



```
start[0] = 1;   start[1] = 2;
count[0] = 3;   count[1] = 2;
stride[0] = 3; stride[1] = 3;
```

Illustration for block parameter



```
start[0]  = 1;   start[1]  = 2;
count[0]  = 3;   count[1]  = 2;
stride[0] = 3;  stride[1] = 3;
block[0]  = 2;   block[1]  = 2;
```

Exercice 1

Please draw the elements selected by the start, stride, count, block set below

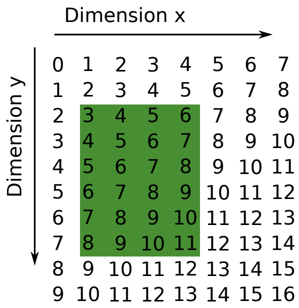
Dimension x →

Dimension y ↓

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14
8	9	10	11	12	13	14	15
9	10	11	12	13	14	15	16

```
 start[0] = 2;   start[1] = 1;
count[0] = 6;   count[1] = 4;
```

Solution 1

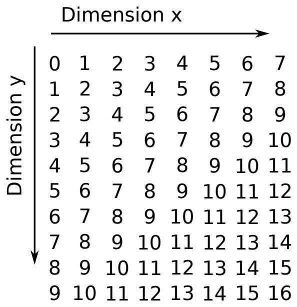


0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14
8	9	10	11	12	13	14	15
9	10	11	12	13	14	15	16

```
start[0] = 2;  start[1] = 1;
count[0] = 6;  count[1] = 4;
```

Exercice 2

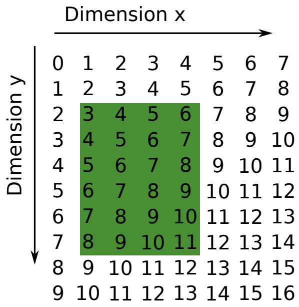
Please draw the elements selected by the start, stride, count, block set below



0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14
8	9	10	11	12	13	14	15
9	10	11	12	13	14	15	16

```
 start[0] = 2;   start[1] = 1;
count[0] = 1;   count[1] = 1;
block[0] = 6;   block[1] = 4;
```

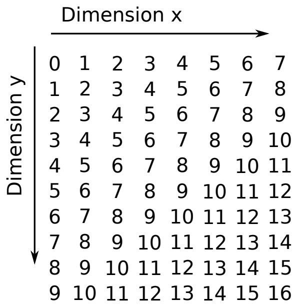
Solution 2



```
start[0] = 2;   start[1] = 1;
count[0] = 1;   count[1] = 1;
block[0] = 6;   block[1] = 4;
```

Exercice 3

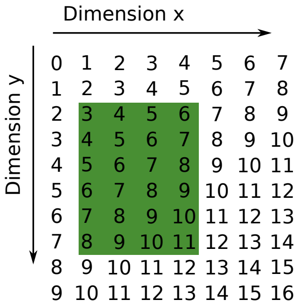
Please draw the elements selected by the start, stride, count, block set below



0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14
8	9	10	11	12	13	14	15
9	10	11	12	13	14	15	16

```
 start[0] = 2;  start[1] = 1;
 count[0] = 3;  count[1] = 2;
stride[0] = 2; stride[1] = 2;
 block[0] = 2;  block[1] = 2;
```

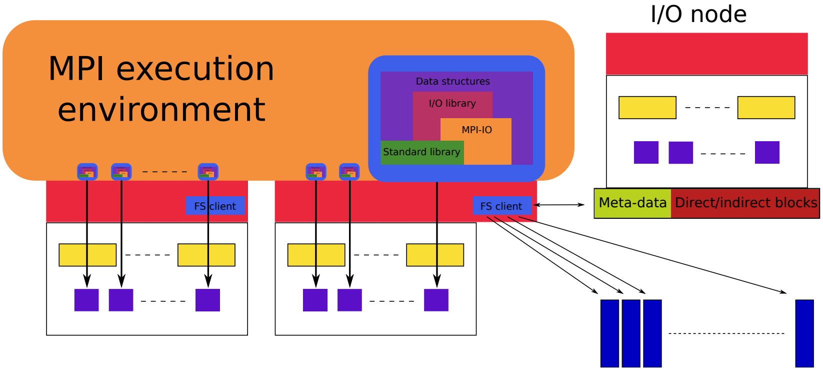
Solution 3



0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14
8	9	10	11	12	13	14	15
9	10	11	12	13	14	15	16

```
 start[0] = 2;   start[1] = 1;
 count[0] = 3;   count[1] = 2;
stride[0] = 2;  stride[1] = 2;
 block[0] = 2;   block[1] = 2;
```

Let us put everything together again



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

$\Rightarrow$ More work at post-processing level

$\Rightarrow$ Very easy to implement

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

$\Rightarrow$ Single resulting file

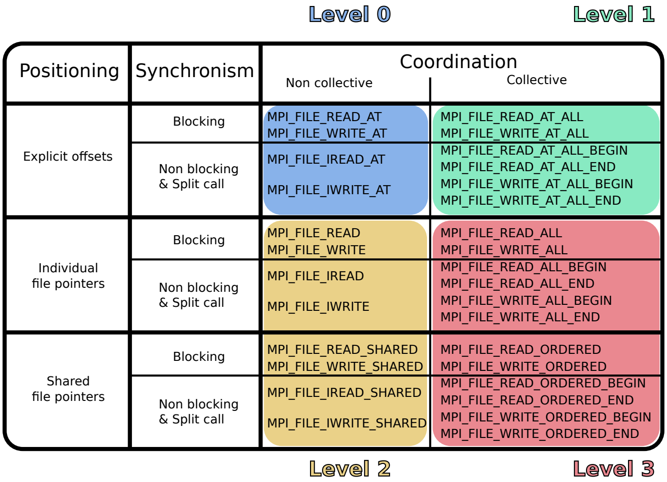
MPI-IO concept

- 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

- No more memory limitation
- Single resulting file
- Definition of MPI derived types

MPI-IO API

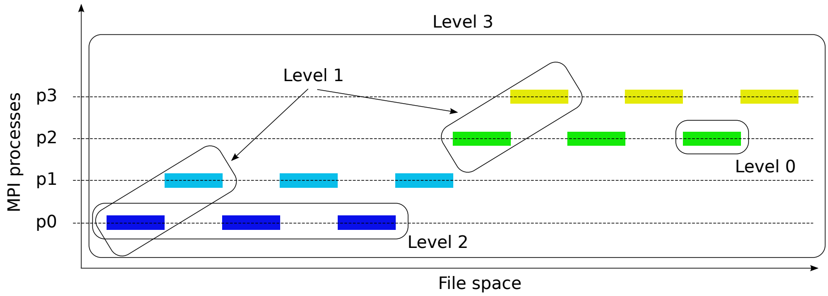


Level 0 Level 1

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
	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
	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
	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

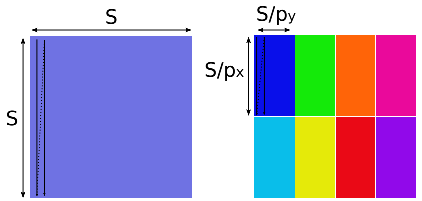


Parallel HDF5

- 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

- Easy to develop
- Single portable file
- Maybe some performance issues
- Add library dependancy

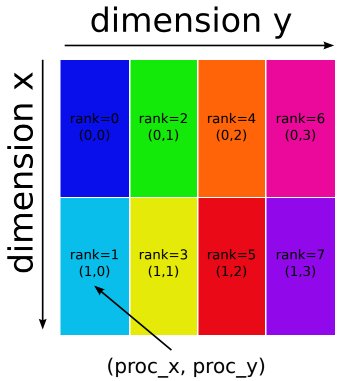
Test case



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:

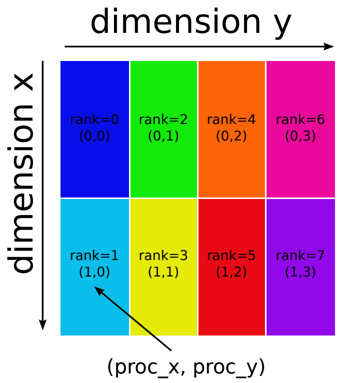
- A 2D structured array
- x contiguous in memory
- x represented vertically
- Fortran language convention

$\Rightarrow$ 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:

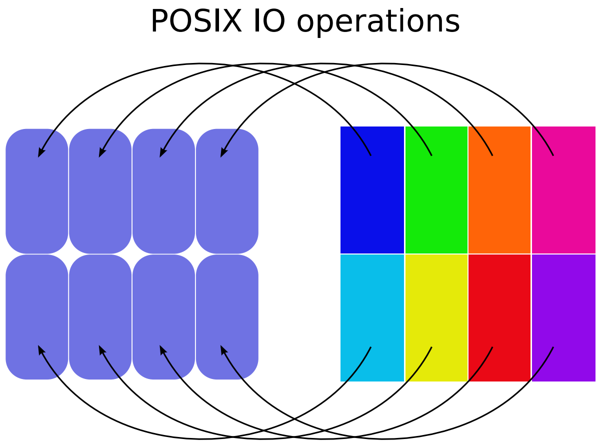
- A 2D structured array
- x contiguous in memory
- x represented vertically
- Fortran language convention

$\Rightarrow$ 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
```

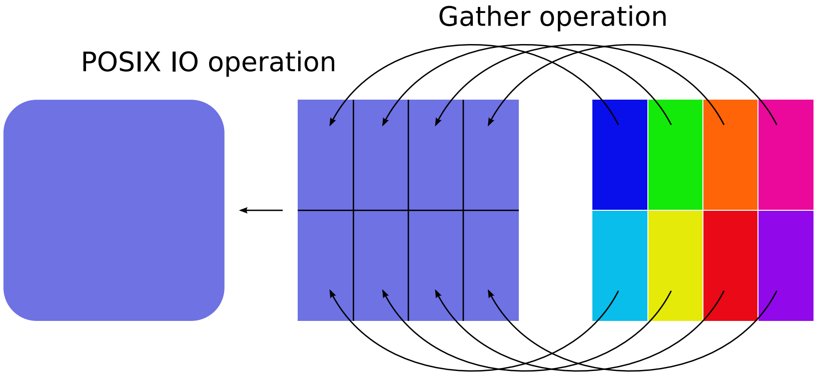
Multiple POSIX files



Multiple POSIX files

```
Integer :: array_size, local_nx, local_ny, rank
Real, allocatable :: tab(:,:)
local_nx = S/px
local_ny = S/py
array_size = local_nx * local_ny * 4

Allocate(tab(1:local_nx, 1:local_ny))
! Fill the tab array...
Open(unit=15, file='res_'//trim(ADJUSTL(rank))//'.bin', &
    status='unknown', form='unformatted', access='direct', &
    iostat=istat, RECL=array_size)
Write(15,rec=1) tab
Close(15)
```

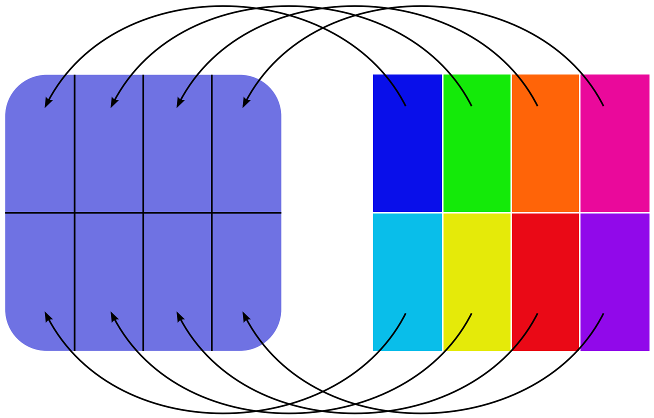
Gather + single POSIX file



Gather + single POSIX file

```
Real, allocatable :: tab(:,:), tab_glob(:,:)
Allocate(tab(1:local_nx, 1:local_ny))
IF (rank == 0) THEN
  Allocate(tab_glob(1:S, 1:S))
END IF

! Fill the tab array...
! Gather the different tab within tab_glob

IF (rank == 0) THEN
  array_size = S * S * 4
  Open(unit=15, file='res.bin', status='unknown', form='unformatted', &
      access='direct', iostat=istat, RECL=array_size)
  Write(15,rec=1) tab_glob
  Close(15)
END IF
```

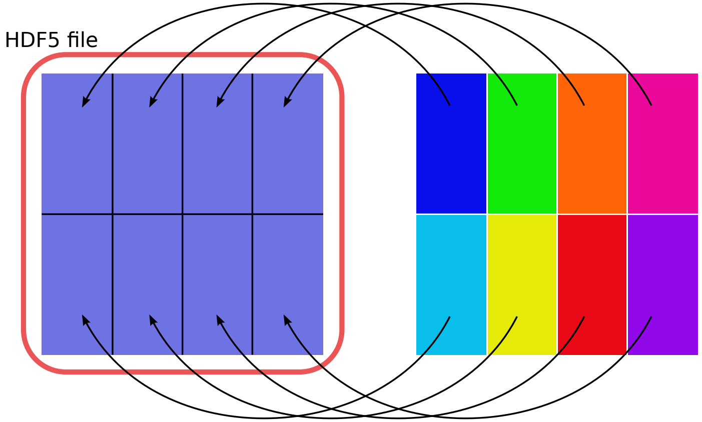
MPI-IO



MPI-IO

```
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_NULL, &
                        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)
```

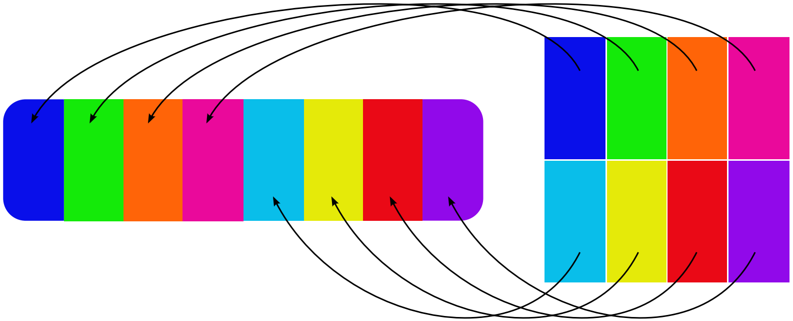
Parallel HDF5



Parallel HDF5

```
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 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 chunks



MPI-IO chunks

```
INTEGER :: array_size(1), array_subsize(1), array_start(1)
INTEGER :: myfile, filetype

array_size(1) = S * S
array_subsize(1) = local_nx*local_ny
array_start(1) = (proc_x  + proc_y*nb_proc_x) * array_subsize(1)

!Allocate and fill the tab array

CALL MPI_TYPE_CREATE_SUBARRAY(1, 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_NULL, &
                      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)
```

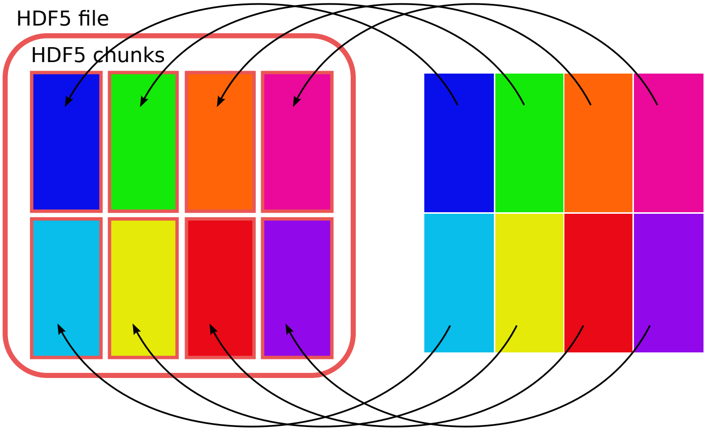
MPI-IO chunks

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

$\Rightarrow$ More work at post-processing level like in the multi-file method

$\Rightarrow$ Concurrent accesses reduction

Parallel HDF5 chunks



Parallel HDF5 chunks

```
INTEGER(HSIZE_T) :: array_size(2), array_subsize(2), array_start(2), chunk_dims(2)
INTEGER(HID_T) :: plist_id1, plist_id2, plist_id3, file_id, filespace, dset_id, memspace
array_size(1) = S
array_size(2) = S
array_subsize(1) = local_nx
array_subsize(2) = local_ny
chunk_dims(1) = local_nx
chunk_dims(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 h5pset_dxpl_mpio_f(plist_id2, H5FD_MPIO_COLLECTIVE_F, ierr)

CALL h5pcreate_f(H5P_DATASET_CREATE_F, plist_id3, ierr)
CALL h5pset_chunk_f(plist_id3, 2, chunk_dims, 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, plist_id3)
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
```

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)

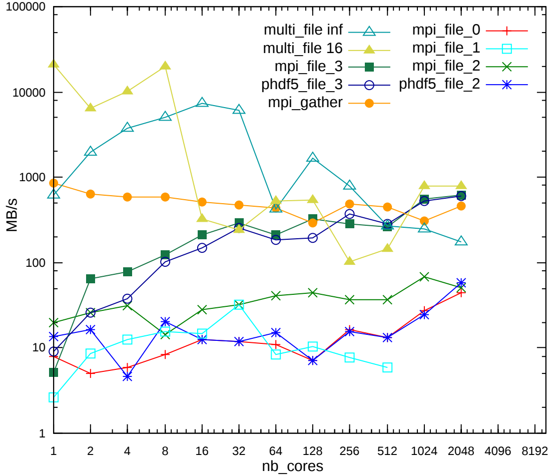
- Located in Jülich Supercomputing Center (JSC)
- Bull machine
- 8640 INTEL Xeon Nehalem-EP cores
- Lustre file system

VIP machine

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

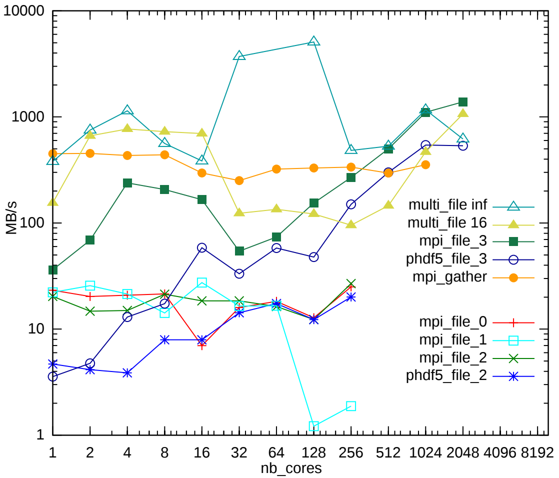
Weak scaling on VIP

4MB to export per MPI task



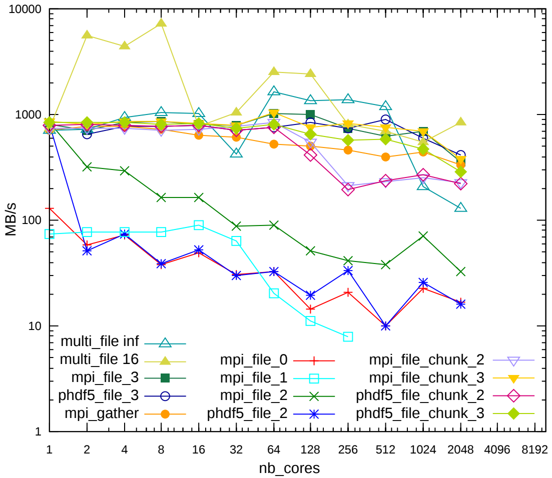
Weak scaling on HPC-FF

4MB to export per MPI task



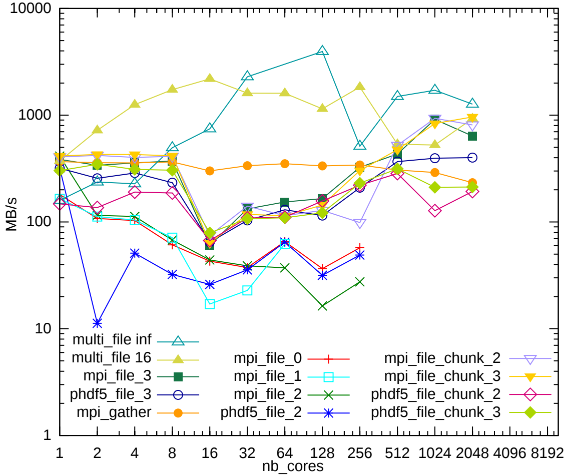
Strong scaling on VIP

A total of 8GB to export



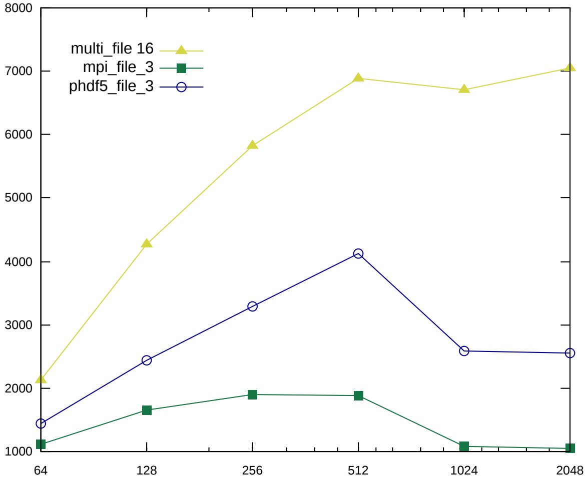
Strong scaling on HPC-FF

A total of 8GB to export



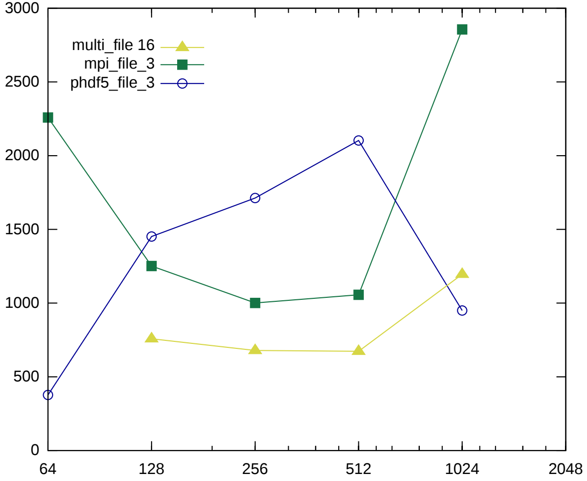
Strong scaling on VIP

A total of 256GB to export



Strong scaling on HPC-FF

A total of 256GB to export



Conclusions

1. Never use MPI-IO explicit offsets, always the view mechanism
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`