CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Parallel I/O

HPC Summer School 2017

Claudio Gheller
cgheller@cscs.ch

Introduction

Reading and Writing data is a problem usually underestimated.

However it can become crucial for the performance of a code.

A code with highly parallelized algorithms can spend most of its time reading/writing data.

Amdahl's law holds also for I/O:

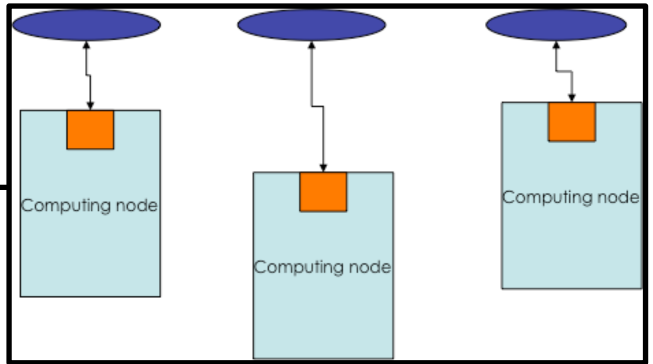
$$\textbf{performance gain} = \frac{1}{(1-F)+\frac{F}{N}}.$$

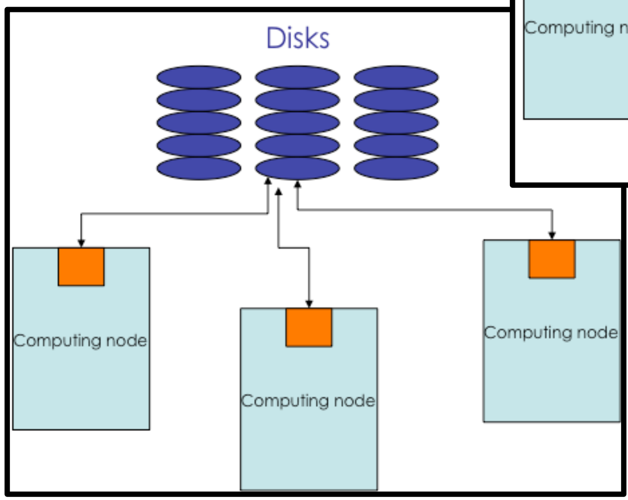
Parallel I/O

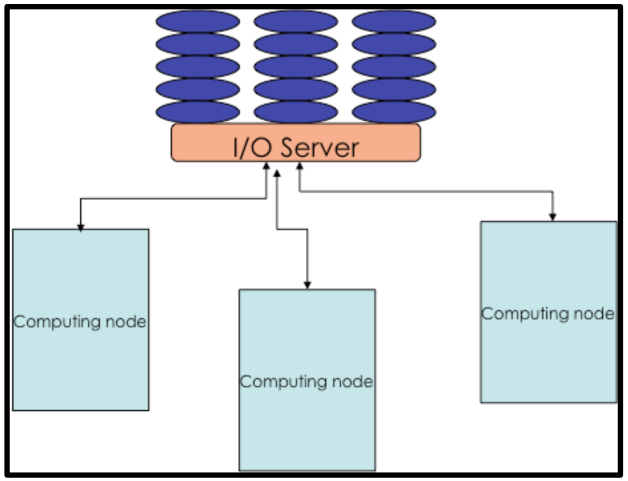
So, we need to parallelize also the I/O

However, challenges and solutions are different from the rest of the code

This is due to the different hardware/software taking care of the I/O







...and many others...

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Parallel I/O: Main Goals

- **Improve the performance, parallel read/write MUST be faster (at least, not slower!) than sequential**
  - Minimize communication
  - Do not introduce latencies/overheads
  - Overlap I/O to computation
  - Read/Write in big chunks
  - Exploit the hardware
- **Preserve usability**
  - Parallel I/O can lead to messy outcomes
  - Parallel I/O can limit the usability of files

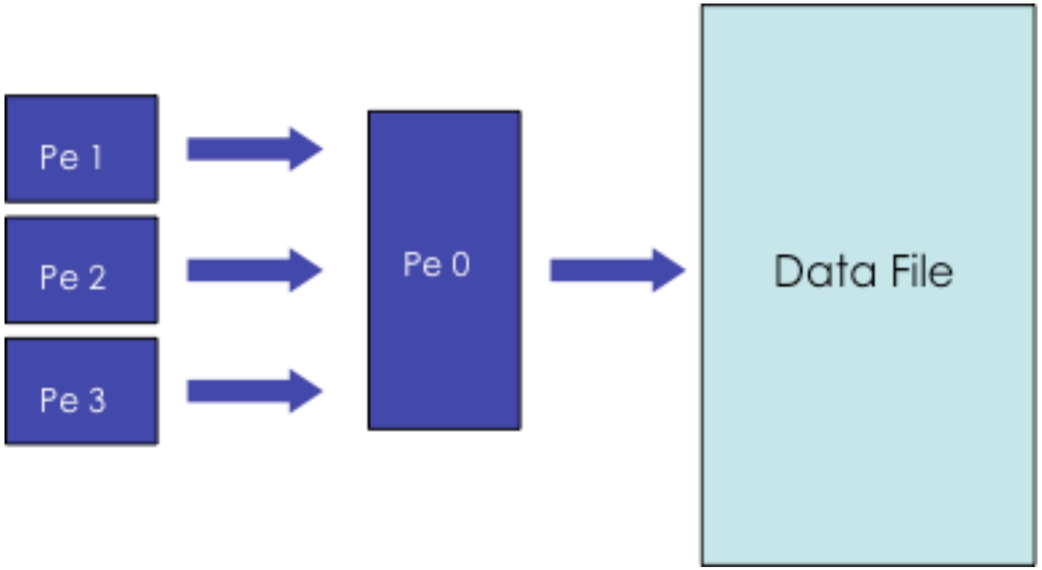
Parallel I/O: possible strategies

- **Master-Slave**
  - Only one processor takes care of the I/O
- **Distributed**
  - All processors perform the I/O independently
- **Coordinated**
  - All processors perform the I/O in an “organized” way
- **MPI I/O**
  - The most efficient solution
- **High level libraries (HDF5, NetCDF, Adios, FITS…)**
  - The most “elegant” (and sometimes efficient) solution

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

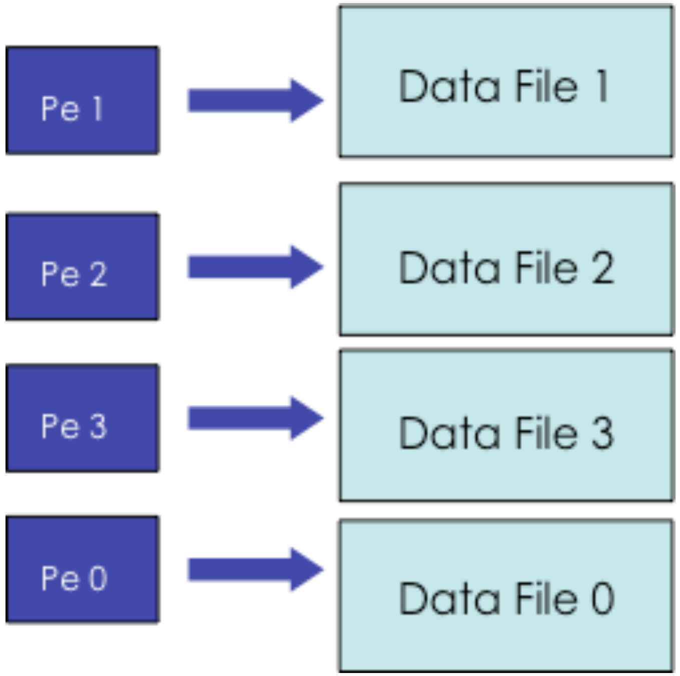
Master-Slave

- **Only one processor performs the I/O**
- **Improves the performance? NO (unless async)**
- **Degrades usability? NO**



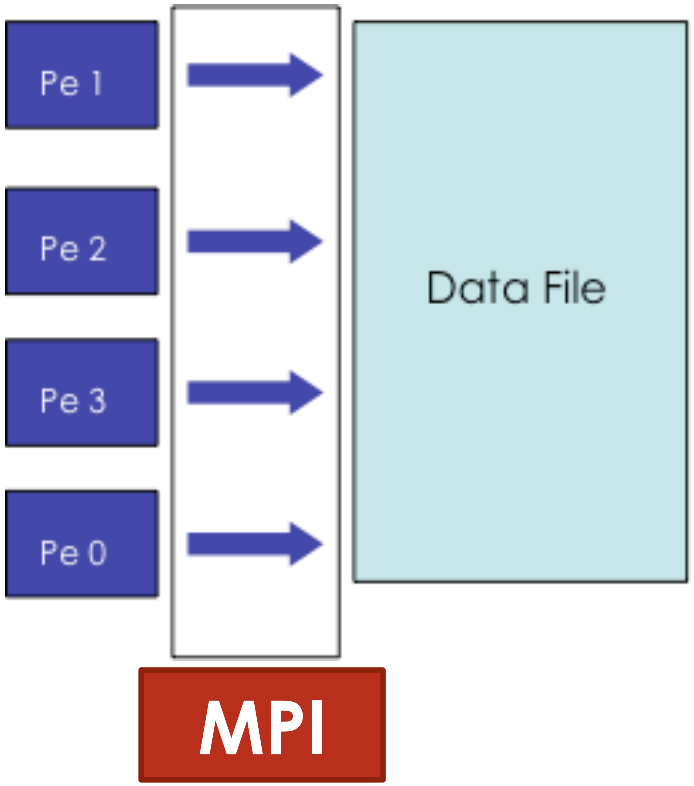
Distributed

- **Each processor reads/writes its own file**
- **Improves the performance?**
  YES (But don't expect linear scaling!)
- **Degrades usability? YES**



Coordinated

- **Each processor reads/writes the same file**
- **Improves the performance?**
  YES (But don't expect linear scaling!)
- **Degrades usability? NO**



MPI I/O

- **MPI extends the same concepts introduced for parallelizing an algorithm to I/O**
- **MPI I/O can be interpreted as message passing to (write) or from (read) a disk**
- **A file written with MPI I/O has NOTHING special. It can be read by a sequential (non MPI) code**

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

MPI FILES

MPI FILE

- **An MPI file is an ordered collection of data items.**
- **MPI supports random or sequential access to any set of these items.**
- **A file is opened collectively by a group of processes (communicator).**

File OPEN

Both opening and closing files are **collective** operations within a communicator. Files are opened using **MPI_FILE_OPEN**:

`MPI_FILE_OPEN(comm, filename, amode, info, fh)`

Each process within the communicator **must specify the same filename and access mode** (`amode`).

`info` = optimization parameter (`info` = MPI_INFO_NULL always possible)
`fh` = file handle, used to reference the file while it is open
The possible `amode` are:

`MPI_MODE_RDONLY` --- read only,
`MPI_MODE_RDWR` --- reading and writing,
`MPI_MODE_WRONLY` --- write only,
`MPI_MODE_CREATE` --- create the file if it does not exist,
`MPI_MODE_EXCL` --- error if creating file that already exists,
`MPI_MODE_DELETE_ON_CLOSE` --- delete file on close,
`MPI_MODE_UNIQUE_OPEN` --- file will not be concurrently opened elsewhere,
`MPI_MODE_SEQUENTIAL` --- file will only be accessed sequentially,
`MPI_MODE_APPEND` --- set initial position of all file pointers to end of file.

File CLOSE

Collective call:

```
MPI_FILE_CLOSE(fh)
```

Reading/Writing Data

```
MPI_File_seek(MPI_File fh, MPI_Offset
        offset, int whence)
```

bytes → offset

whence values
MPI_SEEK_CUR
the file pointer is set to its current position plus offset
MPI_SEEK_END
the file pointer is set to the end of the file position plus offset
MPI_SEEK_SET
the file pointer is set to offset

```
MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype
datatype, MPI_Status *status)
```

```
MPI_File_write(MPI_File fh, void *buf, int count, MPI_Datatype
datatype, MPI_Status *status)
```

Individual file pointers are used:

- Each processor has **its own pointer** to the file
- Pointer on a processor **is not influenced** by any other processor

Reading/Writing "at"

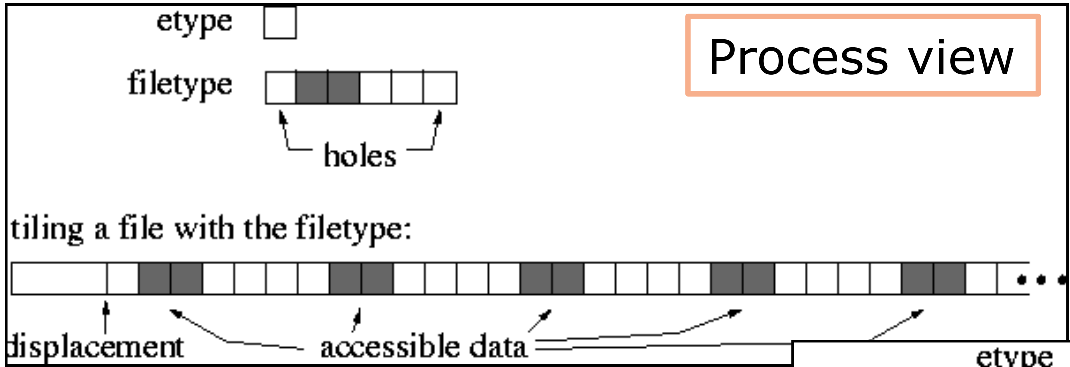
```
MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf,
int count, MPI_Datatype datatype, MPI_Status *status)

MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf,
int count, MPI_Datatype datatype, MPI_Status *status)
```

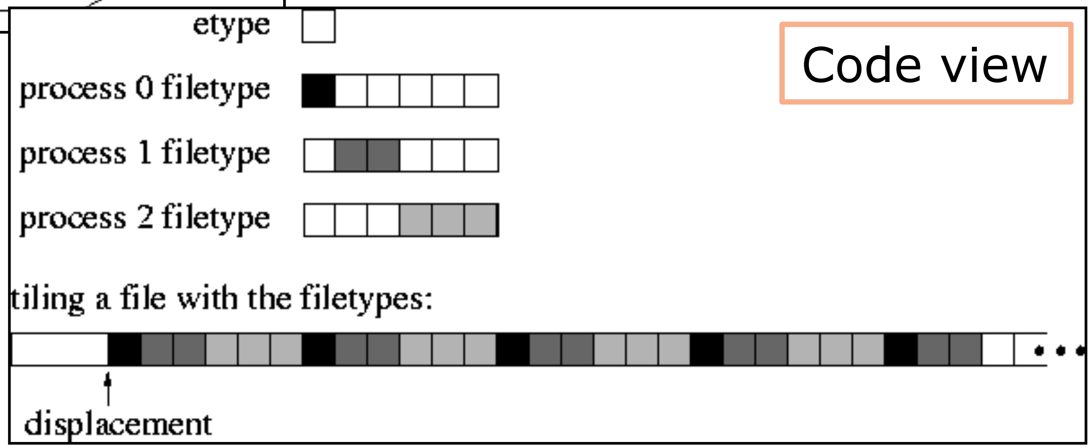
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Views

- **So far nothing new…**
- **Full exploitation of MPI I/O requires the introduction of a new concept: the VIEW**

A file View defines which parts of the file are visible to a MPI process





Creating a VIEW

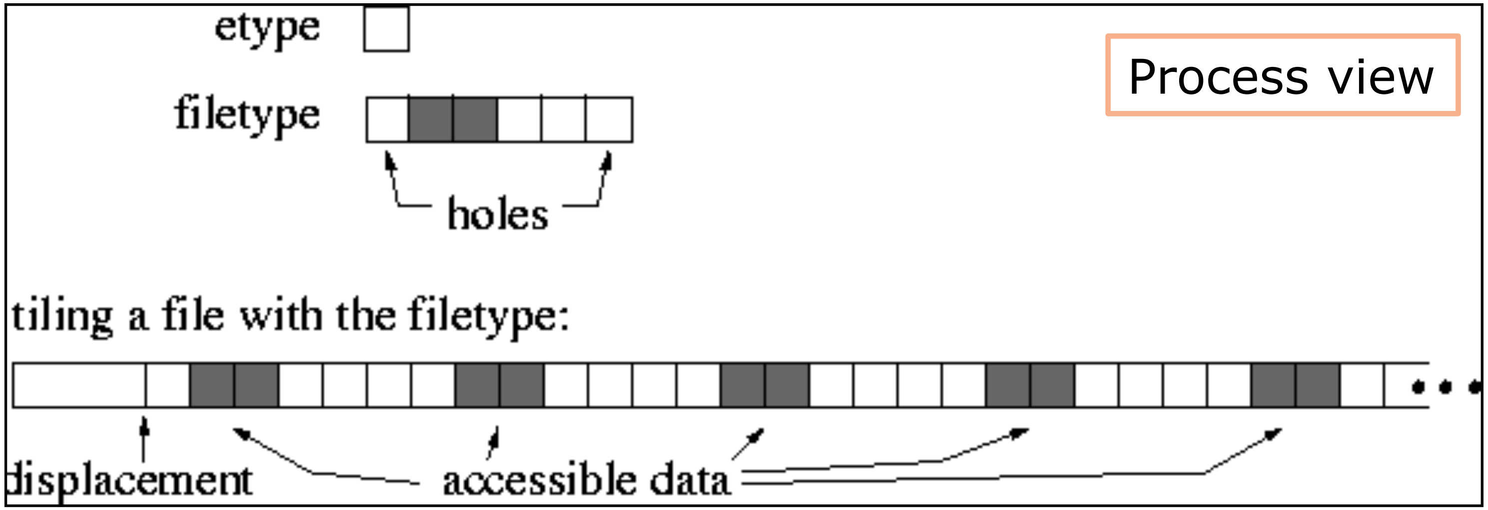
The **MPI_FILE_SET_VIEW** routine is used by each process to describe the layout of the data in the file.

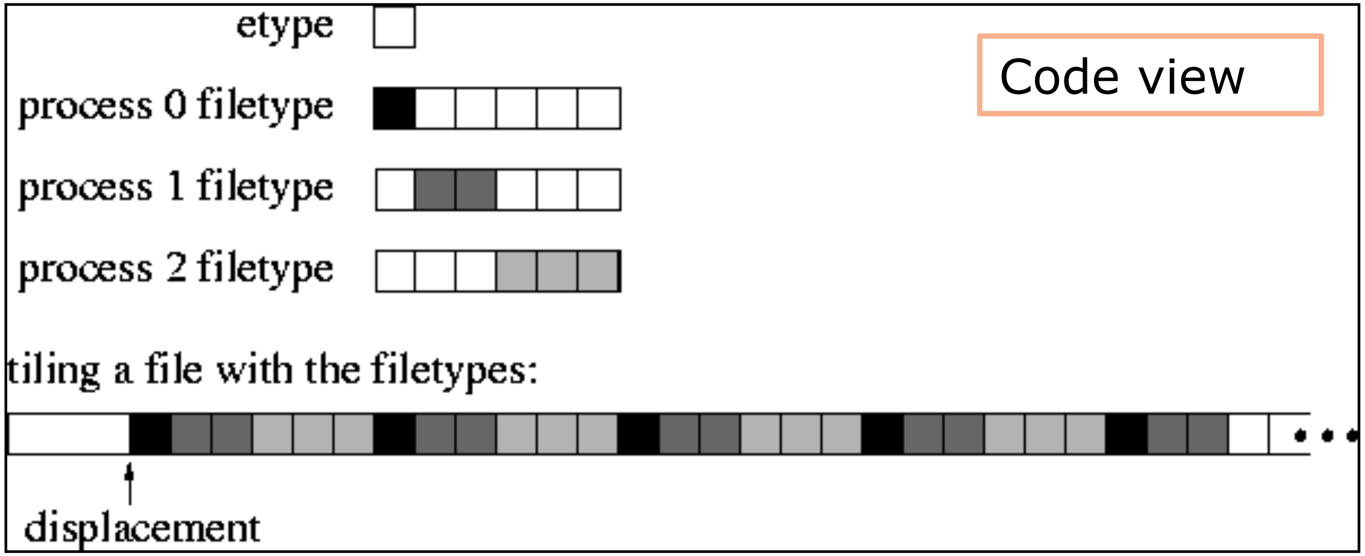
```
MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info)
```

Where:

- `fh` is the file handle.
- `disp` is the displacement from the beginning of the file. In units of bytes.
- `etype` is the elementary datatype. This can be either a pre-defined or a derived datatype but it must have the same value on each process.
- `filetype` is the datatype describing each processes view of the file. These are constructed from units of etypes.
- `datarep` is the data representation (same in all process)

MPI provides functions for creating datatypes for subarrays which can be used in the filetype argument.


etype
filetype
holes
tiling a file with the filetype:
displacement
accessible data
Process view


etype
process 0 filetype
process 1 filetype
process 2 filetype
tiling a file with the filetypes:
displacement
Code view

Data representation (datarep parameter)

- **‘native’: highest performance – data are written as they are in memory**
- **‘internal’: implementation-defined. If necessary data are converted – useful for heterogeneous distributed computing platforms**
- **‘external32’: highest portability: All floating point values are in big-endian 32-bit IEEE format**

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Basic MPI data type (etype, filetype)

elementary datatypes in C

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

elementary datatypes in Fortran

MPI datatype	Fortan datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

MPI Derived Data Type (etype, filetype)

Derived datatypes are new datatypes that are built from the basic MPI datatypes or other derived datatypes.

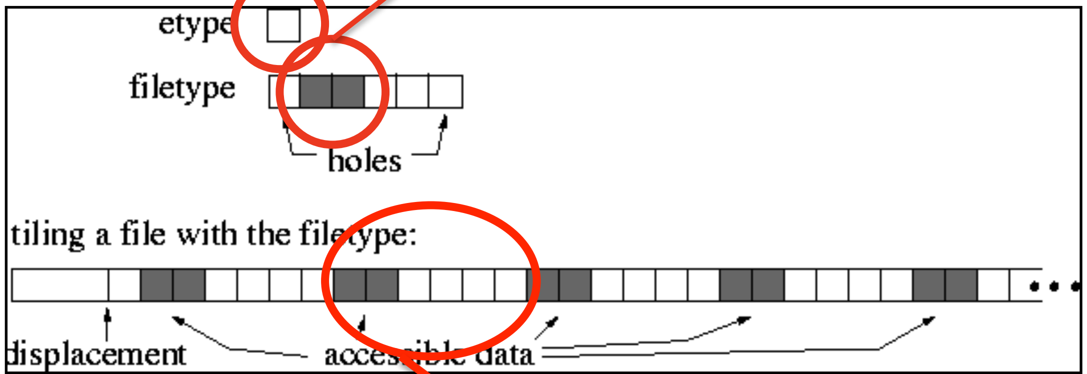
Four steps:

- **Construct the datatype;**
  - MPI_Type_contiguous
  - MPI_Type_vector
  - MPI_Type_indexed
  - MPI_Type_struct
- **Commit the datatype;**
  - MPI_Type_Commit
- **Use the datatype;**
- **Destroy the datatype;**
  - MPI_Type_free

Example

etype= MPI_INT

```
MPI_Type_contiguous(2, MPI_INT, &contig);
```




```
extent = 6 * sizeof(int);
MPI_Type_create_resized(contig, lb, extent, &filetype);
MPI_Type_commit(&filetype);
```

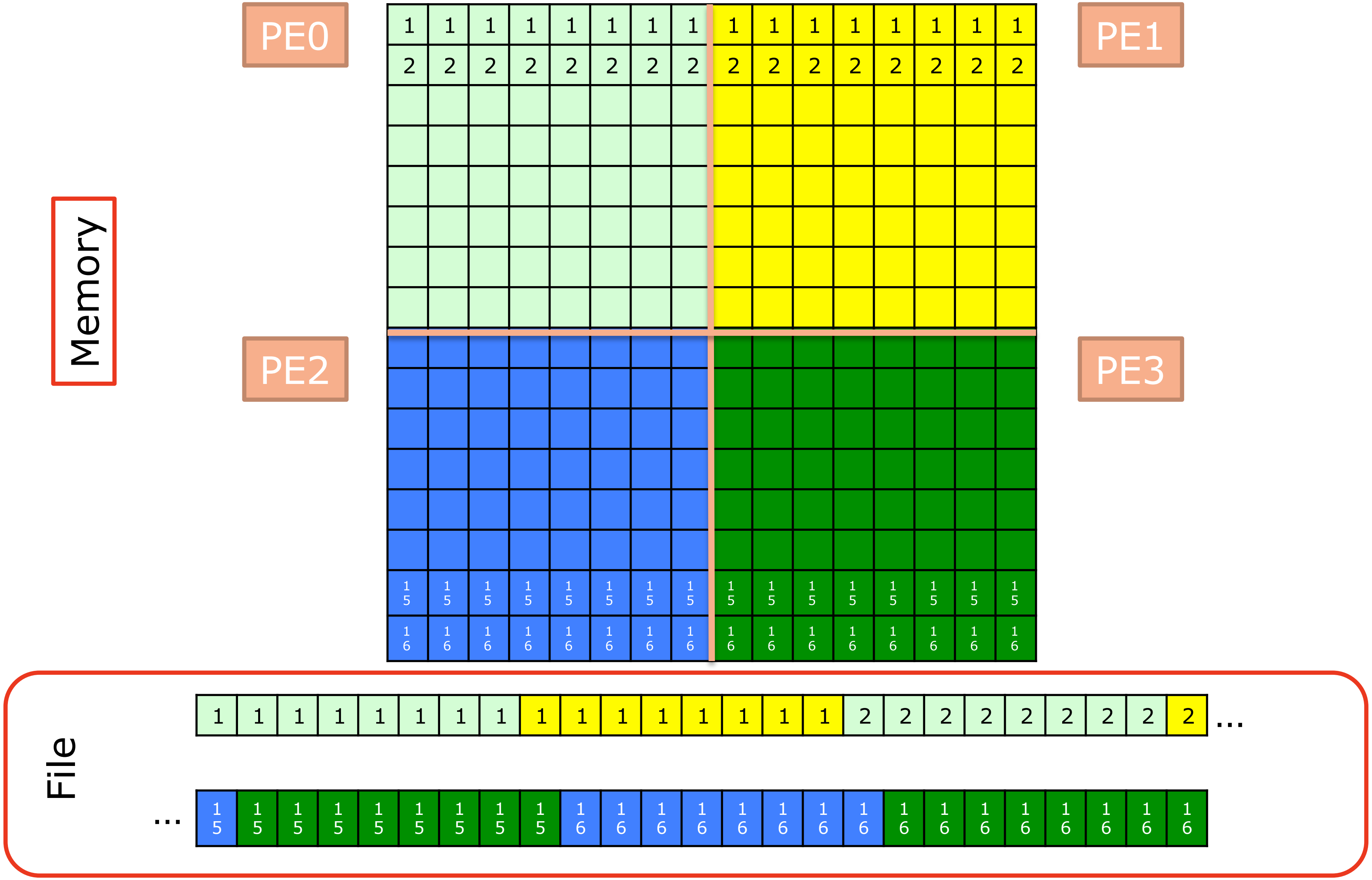
```
MPI_File_set_view(fh, disp, etype, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

Putting all together

```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, contig;
MPI_Offset disp;
MPI_File fh;
int buf[1000];

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);

MPI_Type_contiguous(2, MPI_INT, &contig);
lb = 0;
extent = 6 * sizeof(int);
MPI_Type_create_resized(contig, lb, extent, &filetype);
MPI_Type_commit(&filetype);

disp = 5 * sizeof(int);    /* assume displacement in this file view
                              is of size equal to 5 integers */
etype = MPI_INT;

MPI_File_set_view(fh, disp, etype, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Why should this be useful?



CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

...so what?

Positioning	Synchronisation	Coordination	
		Noncollective	*Collective*
Explicit offsets	*Blocking*	MPI_FILE_READ_AT MPI_FILE_WRITE_AT	MPI_FILE_READ_AT_ALL MPI_FILE_WRITE_AT_ALL
	Non-blocking & ...llective	MPI_FILE_IREAD_AT MPI_FILE_IWRITE_AT	..._BEGIN ..._END ...LL_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 collective	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 pointer	*Blocking*	MPI_FILE_READ_SHARED MPI_FILE_WRITE_SHARED	MPI_FILE_READ_ORDERED MPI_FILE_WRITE_ORDERED
	Non-blocking & split collective	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

Overlap I/O to computation

Coordinate I/O to optimise it

Non-Blocking I/O

This is just like non blocking communication, BUT with the filesystem

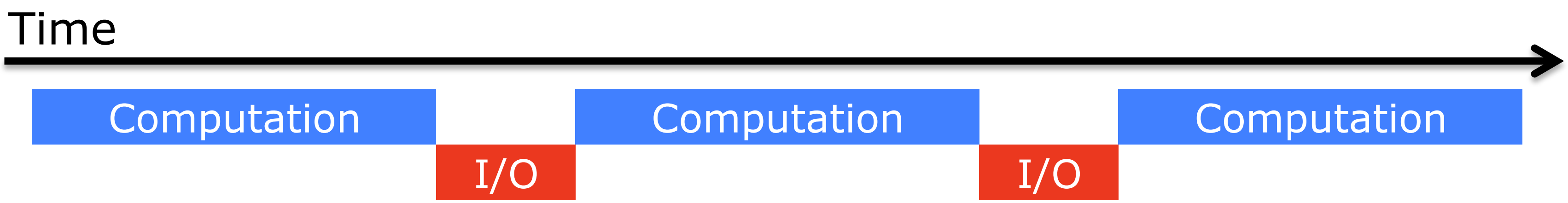
I/O proceed together with computation, so it is hidden

```
MPI_File_iread( MPI_File mpi_fh, void *buf, int count,
MPI_Datatype datatype, MPI_Request *request )
```

Etc. for `write`, `read_at`, `write_at`

`MPI_Wait` **must be used for syncronization.**

Non-Blocking I/O



Total Time = Tc1 + Tio1 + Tc2 + Tio2 + Tc3

Total Time = Tc1 + Tc2 + Tc3 + ε

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Collective I/O

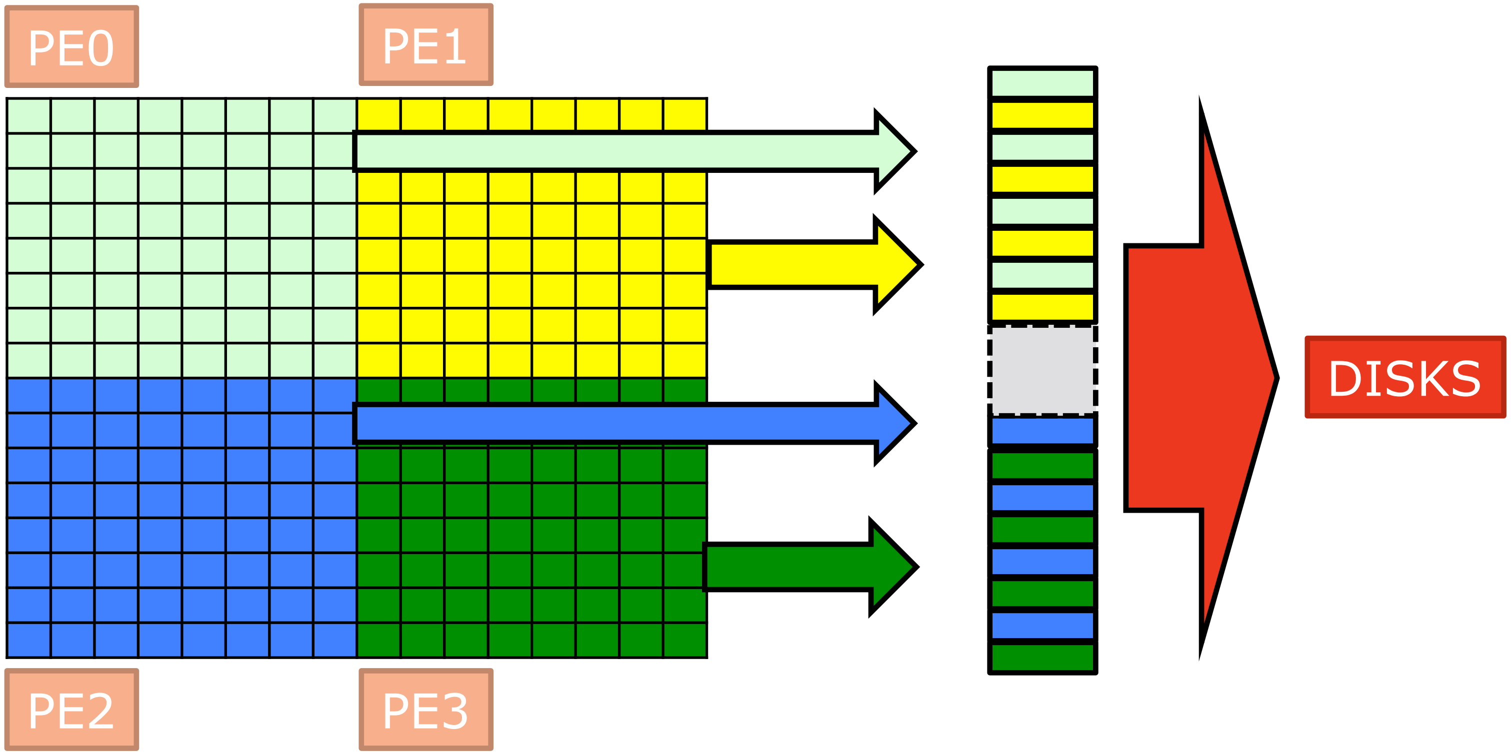
With collective I/O ALL the processors defined in a communicator execute the I/O operation

This permits to optimize the read/write procedure, thanks, in particular, to the VIEW.

```
MPI_File_read_all( MPI_File mpi_fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status )
```

```
MPI_File_write_all(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
```

Collective Write



Split Collective I/O

For collective I/O only a restricted form of nonblocking I/O is supported, called Split Collective:

```
MPI_File_read_all_begin( MPI_File mpi_fh, void *buf, int
count, MPI_Datatype datatype );
     …computation…
MPI_File_read_all_end( MPI_File mpi_fh, void *buf,
MPI_Status *status );
```

The same for write.

Restriction: only one active split operation on a file at a time.

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

NEXT → HDF5