Parallel I/O with MPI

TMA4280—Introduction to Supercomputing

NTNU, IMF

April 13. 2018

Limits of data processing

- Development of computational resource allow running more complex simulations.
- Volume of data produced grows in consequence and can become a problem for performance.
- Reading and writing data to/from a process is named Input/Output (IO).
- Data may be read/written for:
  - meshes used as computational domain
  - saving discrete solution (ex: velocity field for Navier–Stokes)
  - sampling physical quantities for postprocessing

Nowadays data can be produced at a faster rate than it can be processed.

Bottleneck for I/O: Sequential

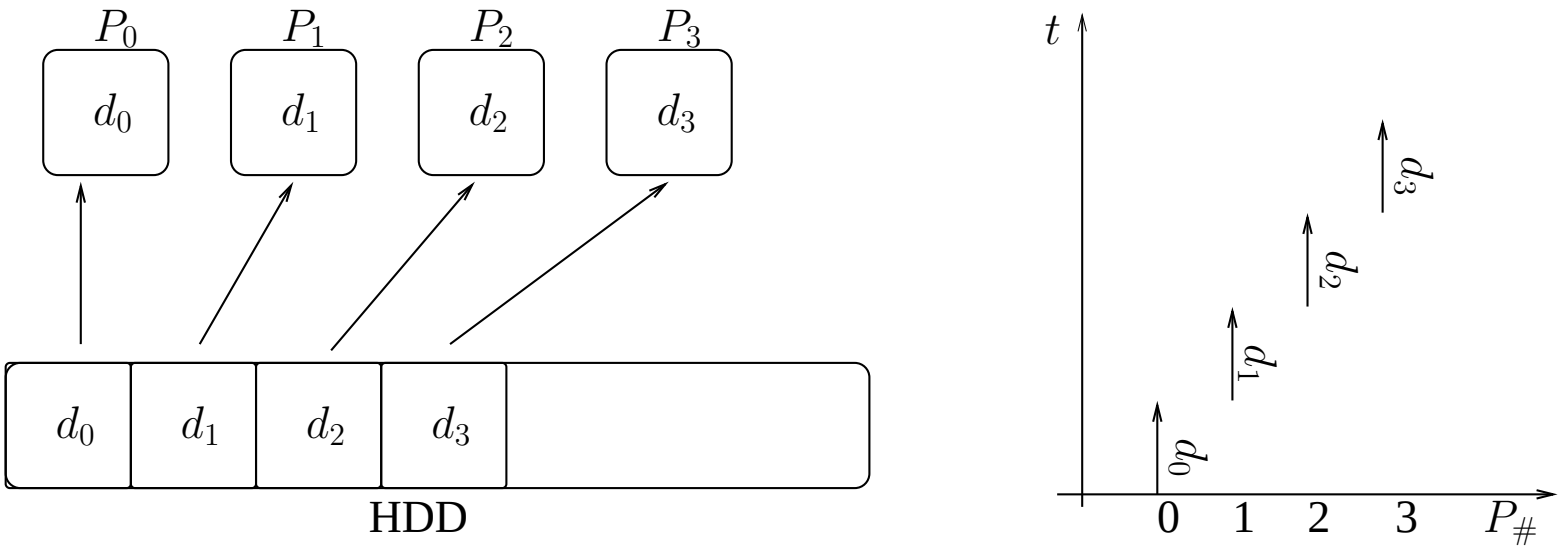

Figure: Illustration of I/O where several processes read from a single file. Since a HDD is serial in nature, only one operation can be performed concurrently.

Bottleneck for I/O: Parallel

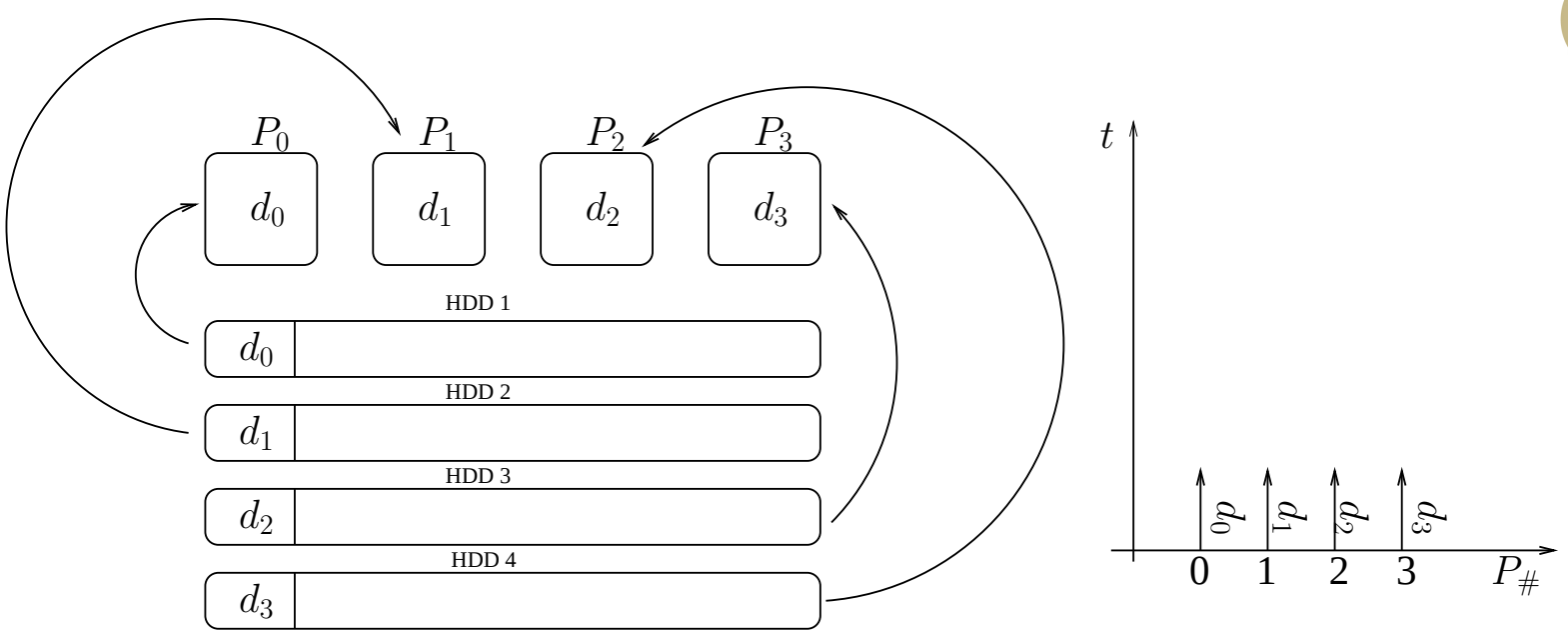

Figure: Illustration of I/O where we harness the aggregated bandwith of several devices. Here each process reads its data from a separate physical device, thus the I/O can be performed in parallel.

Parallel distributed filesystems

- Parallel distributed filesystem are used widely on supercomputers
- The most popular is Lustre (used for instance by LLNL with ZFS)
- They offer high-throughput but also fault-tolerance
- Other popular options are: MooseFS, OrangeFS, GlusterFS.

Example: Landing Gear Simulation

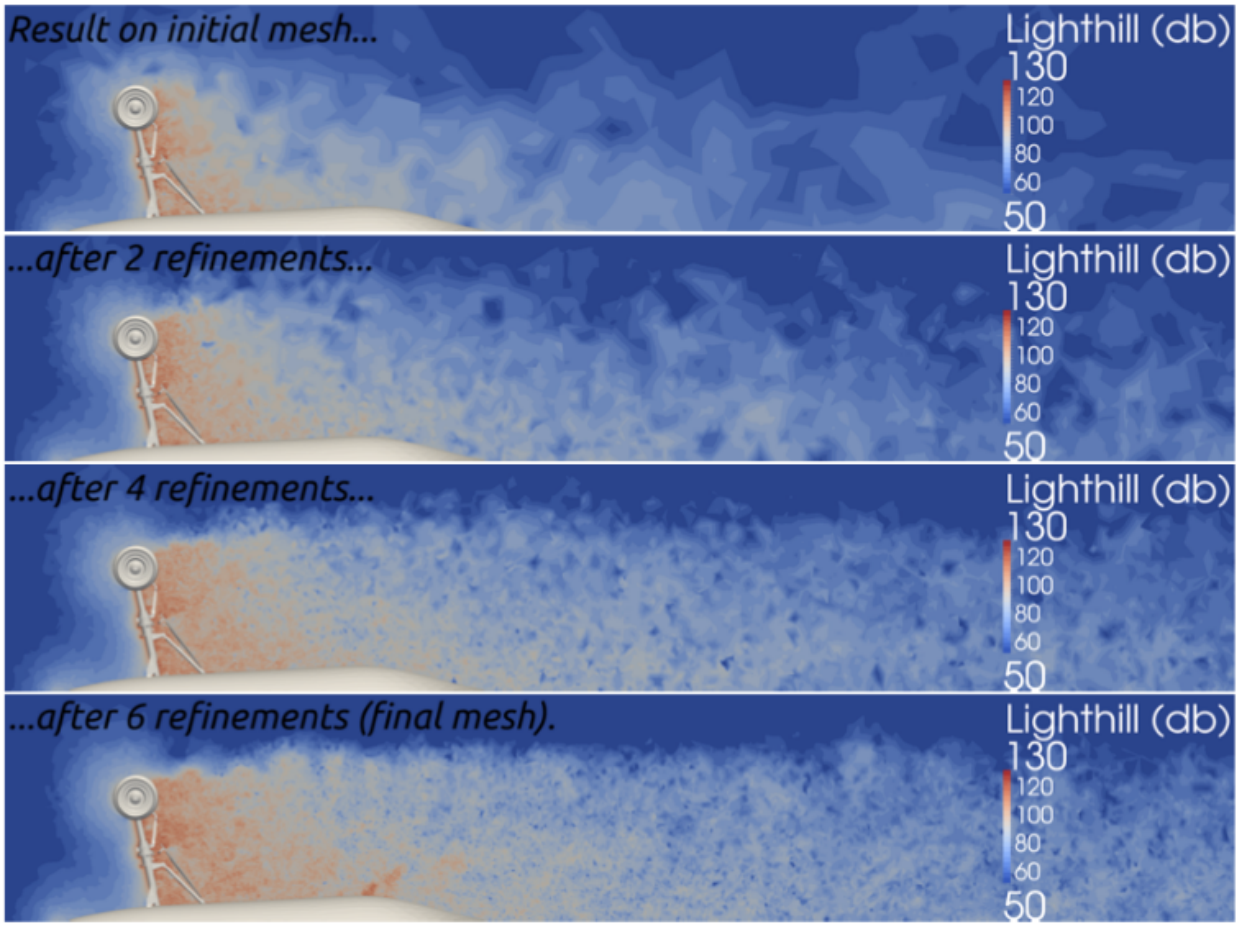

Figure: Initial 3.6M elements, Final 23.8M elements (600K core.h). Vilela De Abreu/N. Jansson/Hoffman, 18th AIAA Aeroacoustics Conference, 2012

Lighthill tensor to evaluate noise generation on the landing gear.

Example: Landing Gear Simulation

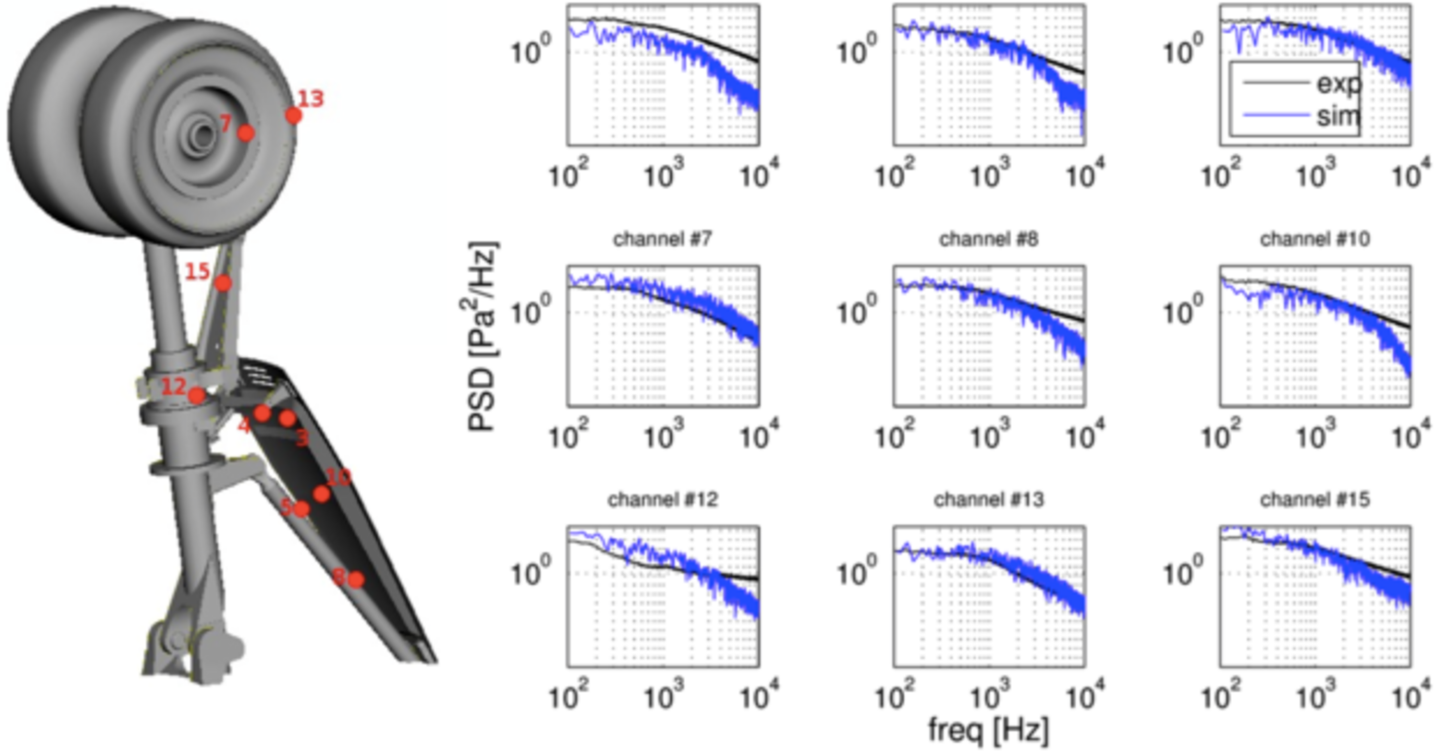

Figure: Sound pressure spectrum. Vilela De Abreu/N. Jansson/Hoffman, 18th AIAA Aeroacoustics Conference, 2012

Sampling pressure fluctuations at 44k hz: 1.4TB written for the sampling.

Example: Full Car Simulation

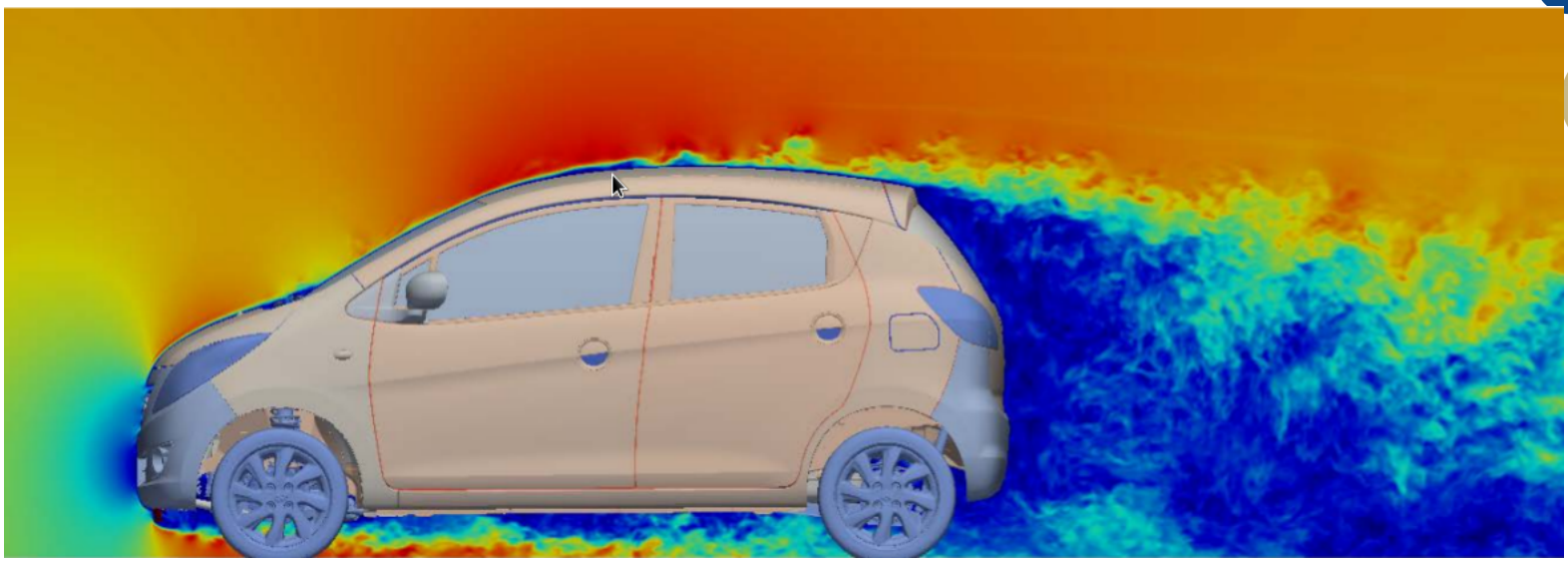

Figure: Full Car Simulation on the K-computer. Image from RIKEN AICS Website.

- Building Cube Method (BCM) with immersed boundary
- 32M cubes, and each cube 32^3 cells
- Surface resolution < 1 mm
- Rotating wheels
- 50TB per file uncompressed, $\approx$ 1TB per file compressed

MPI-IO

- Part of the official MPI standard which covers parallel I/O.
- Natural interface for expressing parallelism with collective I/O routines.
- Opaque handling as the rest of MPI: simple and flexible.
- Implementation details are hidden: lower-level can be optimized without modification of the solver code.

Example of simple code

```
char mytext[BUF_LENGTH];
int rank,numtasks;
MPI_File fp;
MPI_Status status;
MPI_Offset offset;
// Initialize MPI
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
sprintf(mytext,"Rank: %3d Hello World!\n",rank);
offset=rank*strlen(mytext);
// Open file and write text
MPI_File_open(MPI_COMM_WORLD,FILENAME,
              MPI_MODE_CREATE|MPI_MODE_WRONLY,
              MPI_INFO_NULL,&fp);
MPI_File_write_at(fp,offset,&mytext,strlen(mytext),
                  MPI_CHAR,&status);
MPI_File_close(&fp);
// Finalize MPI
MPI_Finalize();
```

MPI File Handle

Before writing a MPI File handle needs to be created:

```
MPI_Comm     comm;
MPI_Info     info;
MPI_File     file;
MPI_File_open(comm, (char *) file, flag, info,
              &file);
```

with:

- `comm`: communicator for processes involved in the I/O
- `info`: info object use to query information on the I/O
- `flag`: mask to define the operation mode
- `file`: file handle, note the address is passed for allocation

After I/O is done, the file handle can be closed:

```
MPI_File_close(&h.file);
```

MPI File Modes

MPI-IO modes are a superset of POSIX modes:

```
/* MPIIO modes: POSIX */
MPI_MODE_RDONLY,                /*    0 ADIO_RDONLY */
MPI_MODE_RDWR,                  /*    1 ADIO_RDWR   */
MPI_MODE_WRONLY,                /*    2 ADIO_WRONLY  */
MPI_MODE_CREATE,                /*    4 ADIO_CREATE */
MPI_MODE_EXCL,                  /*    8 ADIO_EXCL */
/* MPIIO modes: MPI */
MPI_MODE_DELETE_ON_CLOSE,       /*   16 ADIO_DELETE_ON_CLOSE */
MPI_MODE_UNIQUE_OPEN,           /*   32 ADIO_UNIQUE_OPEN */
MPI_MODE_APPEND,                /*   64 ADIO_APPEND */
MPI_MODE_SEQUENTIAL,            /*  128 ADIO_SEQUENTIAL */
```

MPI File Modes

MPI-IO modes are a superset of POSIX modes:

```
/* MPIIO modes: POSIX */
MPI_MODE_RDONLY,                /*    0 ADIO_RDONLY */
MPI_MODE_RDWR,                  /*    1 ADIO_RDWR   */
MPI_MODE_WRONLY,                /*    2 ADIO_WRONLY  */
MPI_MODE_CREATE,                /*    4 ADIO_CREATE */
MPI_MODE_EXCL,                  /*    8 ADIO_EXCL */
/* MPIIO modes: MPI */
MPI_MODE_DELETE_ON_CLOSE,       /*   16 ADIO_DELETE_ON_CLOSE */
MPI_MODE_UNIQUE_OPEN,           /*   32 ADIO_UNIQUE_OPEN */
MPI_MODE_APPEND,                /*   64 ADIO_APPEND */
MPI_MODE_SEQUENTIAL,            /*  128 ADIO_SEQUENTIAL */
```

MPI File Modes

POSIX-like modes have same semantics:

`MPI_MODE_RDONLY`	Read-only
`MPI_MODE_RDWR`	Read/Write
`MPI_MODE_WRONLY`	Write only
`MPI_MODE_CREATE`	Create if file exists
`MPI_MODE_EXCL`	Error if file exists

MPI-IO modes:

`MPI_MODE_DELETE_ON_CLOSE`	Delete file on close
`MPI_MODE_UNIQUE_OPEN`	Only by MPI process (not even outside)
`MPI_MODE_APPEND`	Append to existing file
`MPI_MODE_SEQUENTIAL`	Sequential mode for backup devices

MPI File Write

A simple write operation using the file handle could be to write a chunk of vector of global size *N* distributed uniformly on all the processes:

```
MPI_Offset mysize = N/size;
MPI_File_seek (fh , rank* mysize * sizeof ( double ),
MPI_SEEK_SET );
MPI_File_write (fh , vec , mysize , MPI_DOUBLE ,
MPI_STATUS_IGNORE );
```

Note the seek call to position the file pointer to the right offset.

The equivalent collective operation:

```
MPI_File_write_all (fh , vec , mysize , MPI_DOUBLE ,
                    MPI_STATUS_IGNORE );
```

MPI File Write

To save MPI call to seek, the offset can be managed externally:

```
MPI_Offset mysize = N/size;
MPI_Offset offset = rank * mysize * sizeof(double);
MPI_File_write_at_all(fh, offset, vec , mysize ,
                      MPI_DOUBLE , MPI_STATUS_IGNORE );
```

Note that the offset is in bytes.

Remember that each process usually owns a range of entities (that needs to be written) but also ghosted entities that are copies of entities owned by other processes (that should not be written).

Example of code in DOLFIN-HPC

We are going through classes implemented for DOLFIN-HPC:

- Binary file write with MPI-IO
- ASCII XML file write with MPI-IO

It is a good practice to design an interface for your software package so that parallel I/O is handled transparently.

Writing a vector with MPI-IO

First step is to extract information about the range of owned data:

```
GenericVector x;
uint size = x.local_size();
real *values = new real[size];
uint offset[2] = { 0, 0 };
offset[0] = x.offset();
offset[1] = size;
x.get(values);
```

The library specifies a format for reading/writing to specify for instance the number of processes:

```
BinaryFileHeader hdr;
uint pe_rank = MPI::rank();
hdr.magic = BINARY_MAGIC;
hdr.pe_size = MPI::size();
hdr.type = BINARY_VECTOR_DATA;
```

Writing a vector with MPI-IO

Writing a file header containing metadata:

```
MPI_File fh;
MPI_Offset byte_offset;
MPI_File_open(dolfin::MPI::DOLFIN_COMM, (char *) filename.c_str(),
              MPI_MODE_WRONLY | MPI_MODE_CREATE, MPI_INFO_NULL, &fh);
MPI_File_write_all(fh, &hdr, sizeof(BinaryFileHeader),
                   MPI_BYTE, MPI_STATUS_IGNORE);
byte_offset = sizeof(BinaryFileHeader);
```

Writing range of each process:

```
MPI_File_write_at_all(fh, byte_offset + pe_rank * 2 * sizeof(uint),
                      &offset[0], 2, MPI_UNSIGNED, MPI_STATUS_IGNORE);
byte_offset += hdr.pe_size * 2 * sizeof(uint);
```

Writing contiguous array:

```
MPI_File_write_at_all(fh, byte_offset + offset[0] * sizeof(real), values,
                      offset[1], MPI_DOUBLE, MPI_STATUS_IGNORE);
MPI_File_close(&fh);

delete[] values;
```

Using MPI Data Types

In the lecture notes several examples are given using MPI derived data types, and is useful for example when writing strided data, with a given data packing.

In practice, using MPI derived data types is a good way to write error prone code in an elegant way and will simplify the structure of the implementation.