# HPC Programming

Message Passing Interface (MPI), Part IV

Peter-Bernd Otte, 10.12.2018

# Introduction MPI

Recap

1. Overview / Getting Started
2. Messages & Point-to-point Communication
3. Nonblocking Communication
4. Error Handling
5. Groups & Communicators
6. Collective Communication
7. MPI I/O
8. MPI Derived Datatypes
9. Common pitfalls and good practice (“need for speed”)
10. Debugging and Profiling

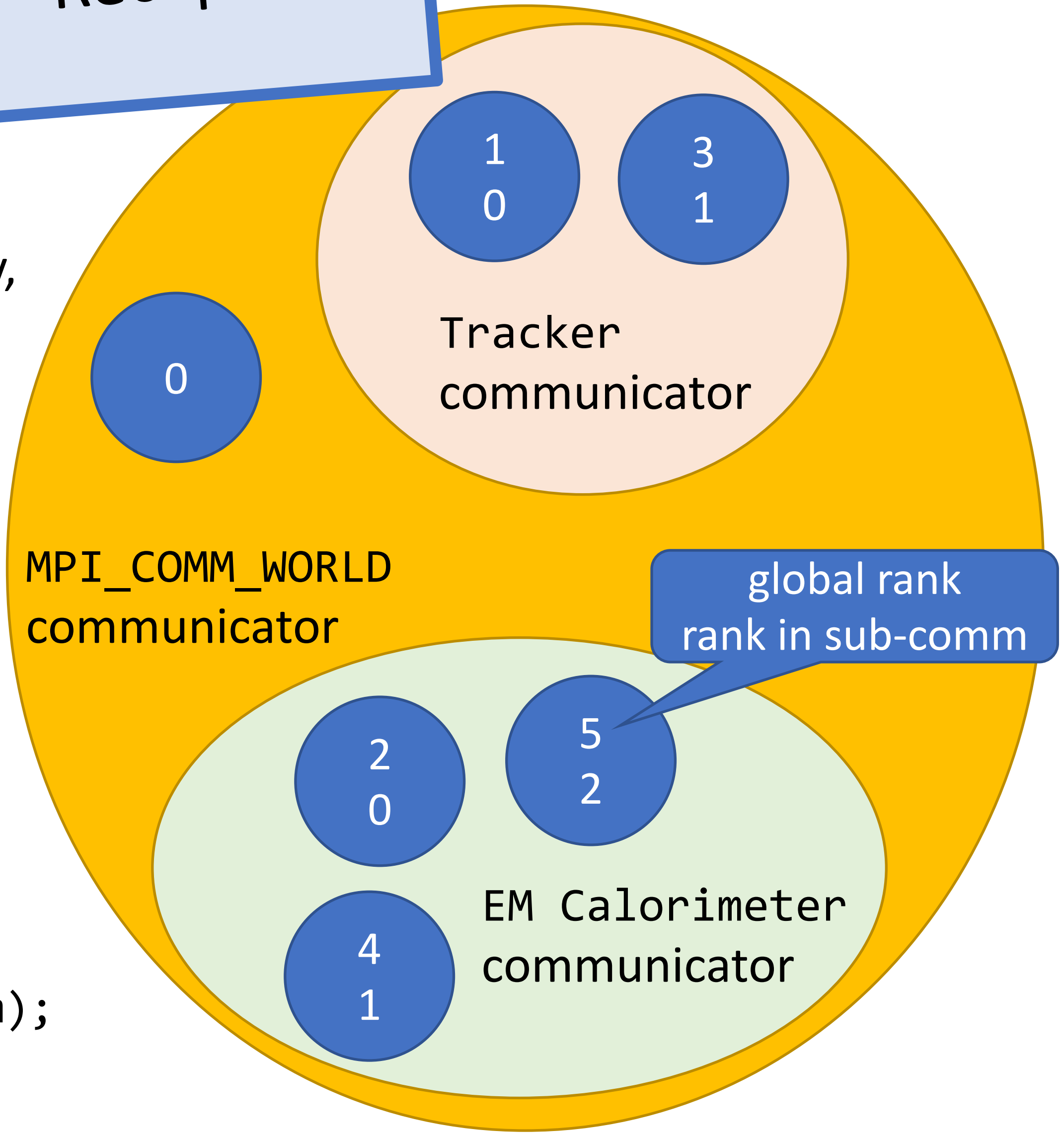
# MPI: MPI_Comm_split

Recap

- Creates new communicators based on colors
- int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
  - ordering in new group:
    - key == 0 → as sorted in old
    - key != 0 → according to key values
  - one member group: color = MPI_UNDEFINED

- Example:

```
MPI_Comm newcomm;
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
mycolor = my_rank/3;
MPI_Comm_split(MPI_COMM_World, mycolor, 0, &newcomm);
MPI_Comm_rank(newcomm, &my_new_rank);
```



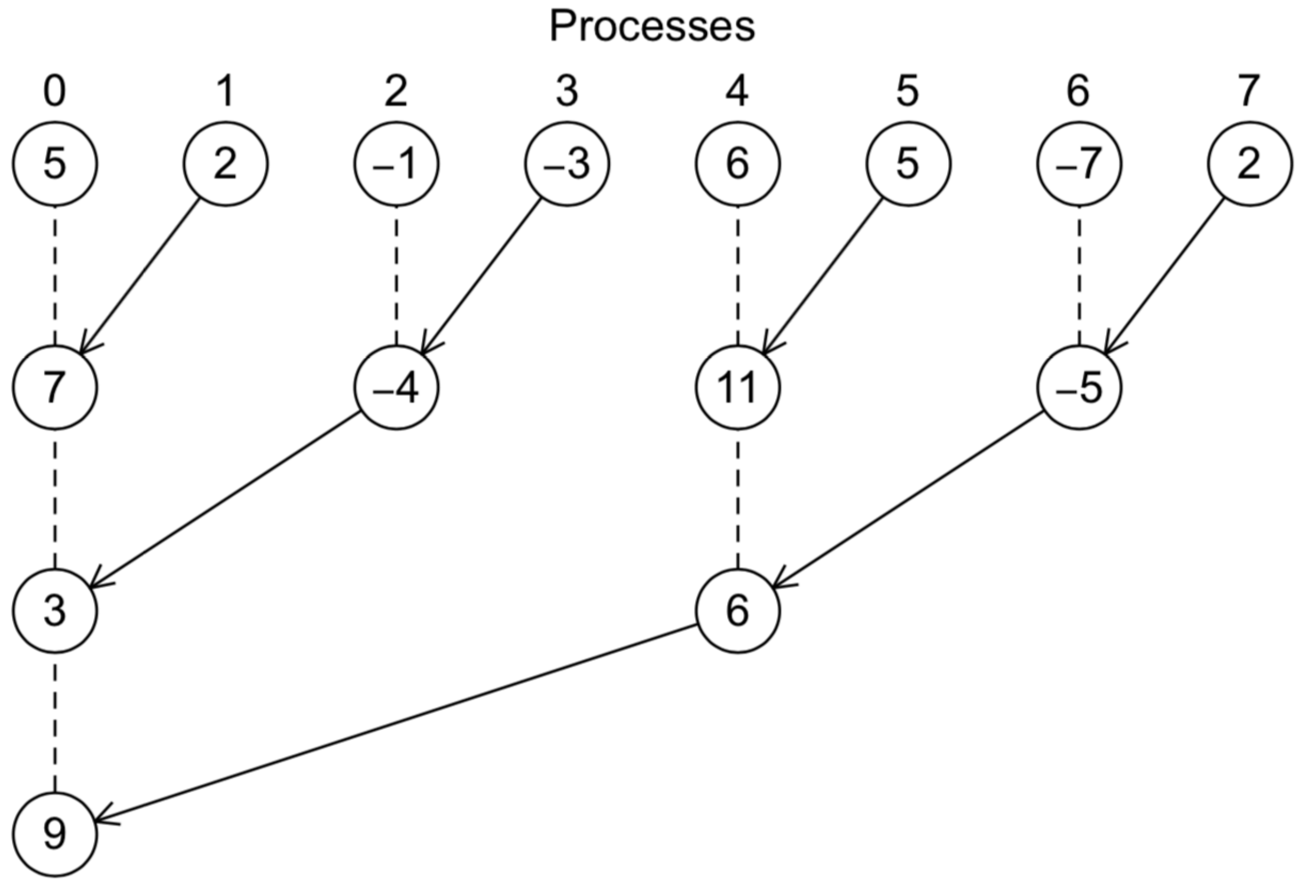
# MPI: MPI_Reduce

Recap

- Reduces values on all processes to a single value
  (eg global sum)

```
int MPI_Reduce(
void *sendbuf /*in*/,
void *recvbuf /*out*/,
int count /*in*/,
MPI_Datatype datatype /*in*/,
MPI_Op operator /*in*/,
int dest_process /*in*/,
MPI_Comm comm /*in*/)
```

- hints:
  - with count>1, MPI can operate on arrays
  - sendbuf and recvbuf need to different
    (no aliasing!)



# MPI: P2P ⇔ Collective Communicatio

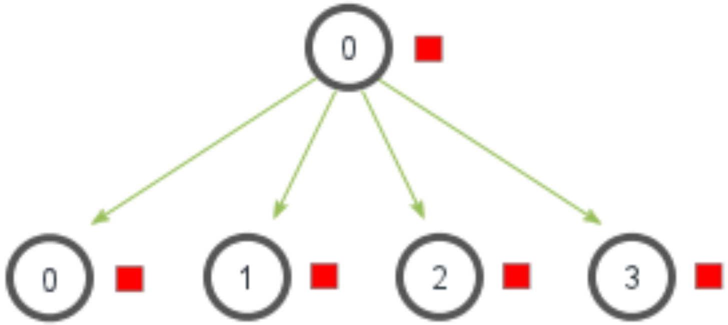
Recap

- ALL processes in communicator must call SAME collective function at the same time.
- Arguments in all ranks must fit:
  - eg. same dest_process, datatype, operator, comm
  - depending on function
- Only rank dest_process may use recvbuf (but all ranks have to provide such argument)
- MPI_Reduce calls matched solely on:
  - the communicator and
  - the order on which they are called.
  - No helping tags or sender id available.

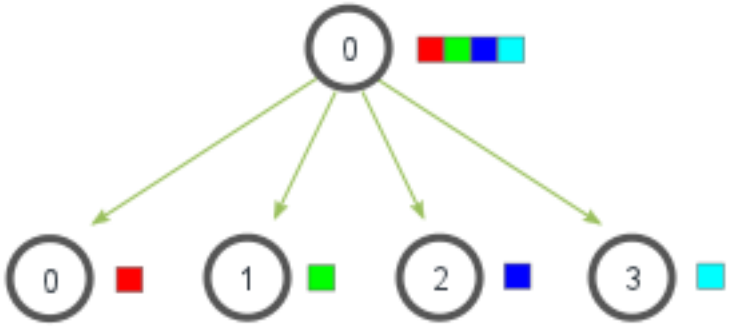
# MPI: Broadcast and Scatter

Recap

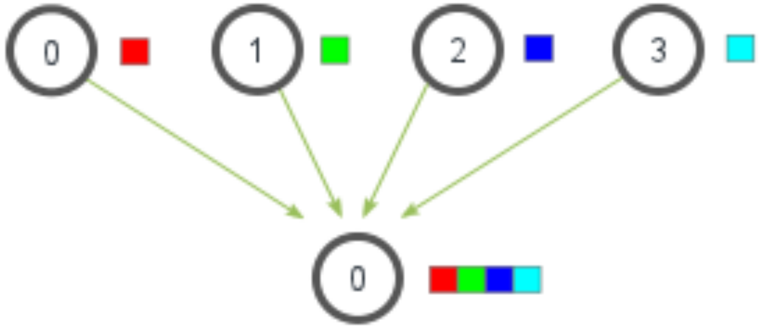
**Broadcasts** the same message to all other processes of the communicator



**Scatter**: Sends data from one process to all other processes in a communicator



**Gathers** (=inverse scatter):

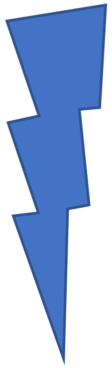


# Introduction MPI

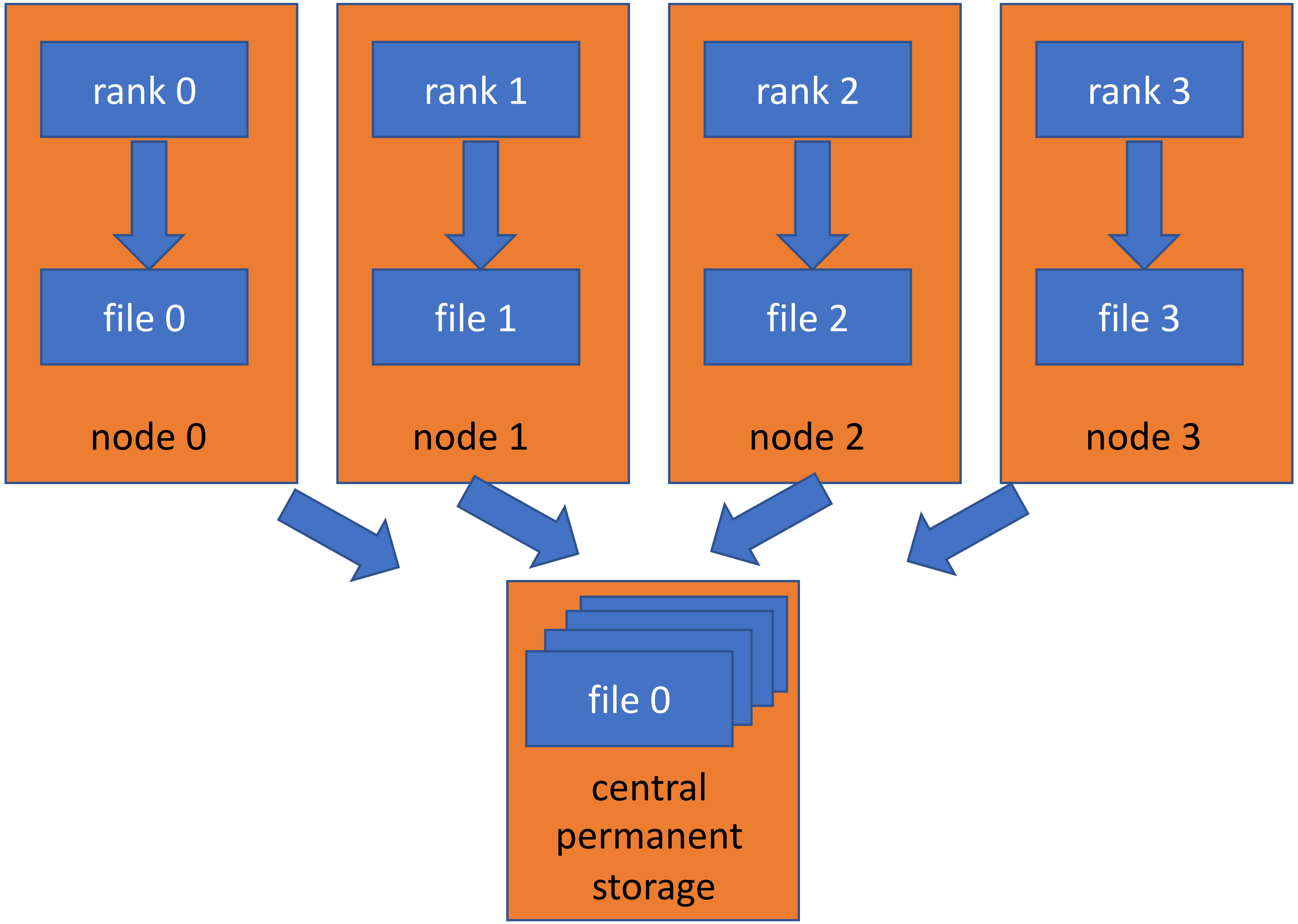
1. Overview / Getting Started
2. Messages & Point-to-point Communication
3. Nonblocking Communication
4. Error Handling
5. Groups & Communicators
6. Collective Communication
7. MPI I/O
8. MPI Derived Datatypes
9. Common pitfalls and good practice (“need for speed”)
10. Debugging and Profiling

# Motivation: MPI I/O 1/4

- Standard (POSIX): each process writes to a single separate file on scratch(!) device
- Typical situation: analysis framework
- parallel → scales!

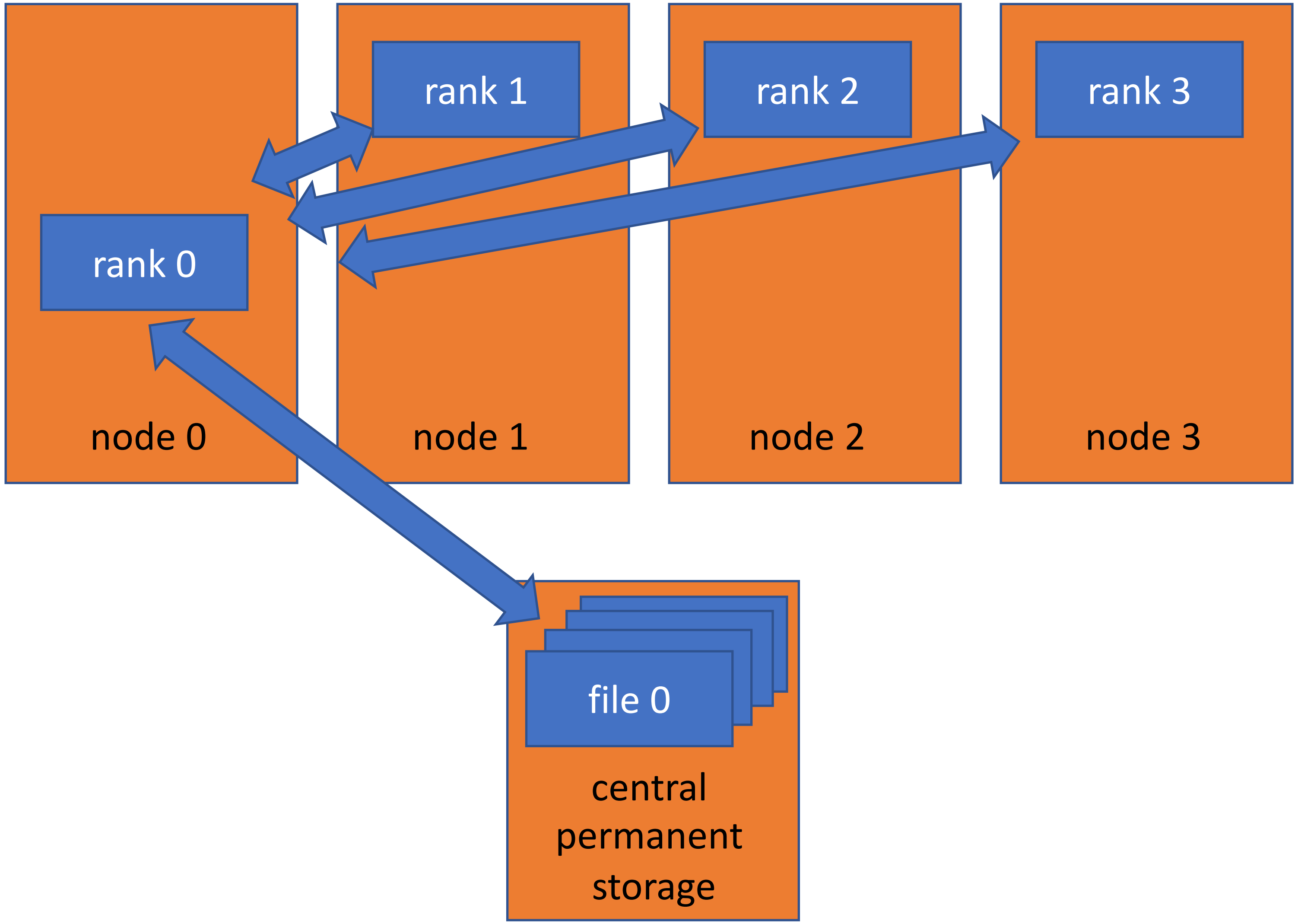


- collection of all these single files → serialisation or worse
- many files → bad for meta data server

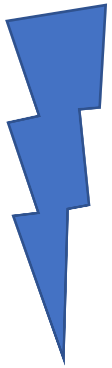


# Motivation: MPI I/O 2/4

- Legacy: only single rank reads/writes
- Typical situation: apps recently parallelised, OpenQCD




Why not ideal:



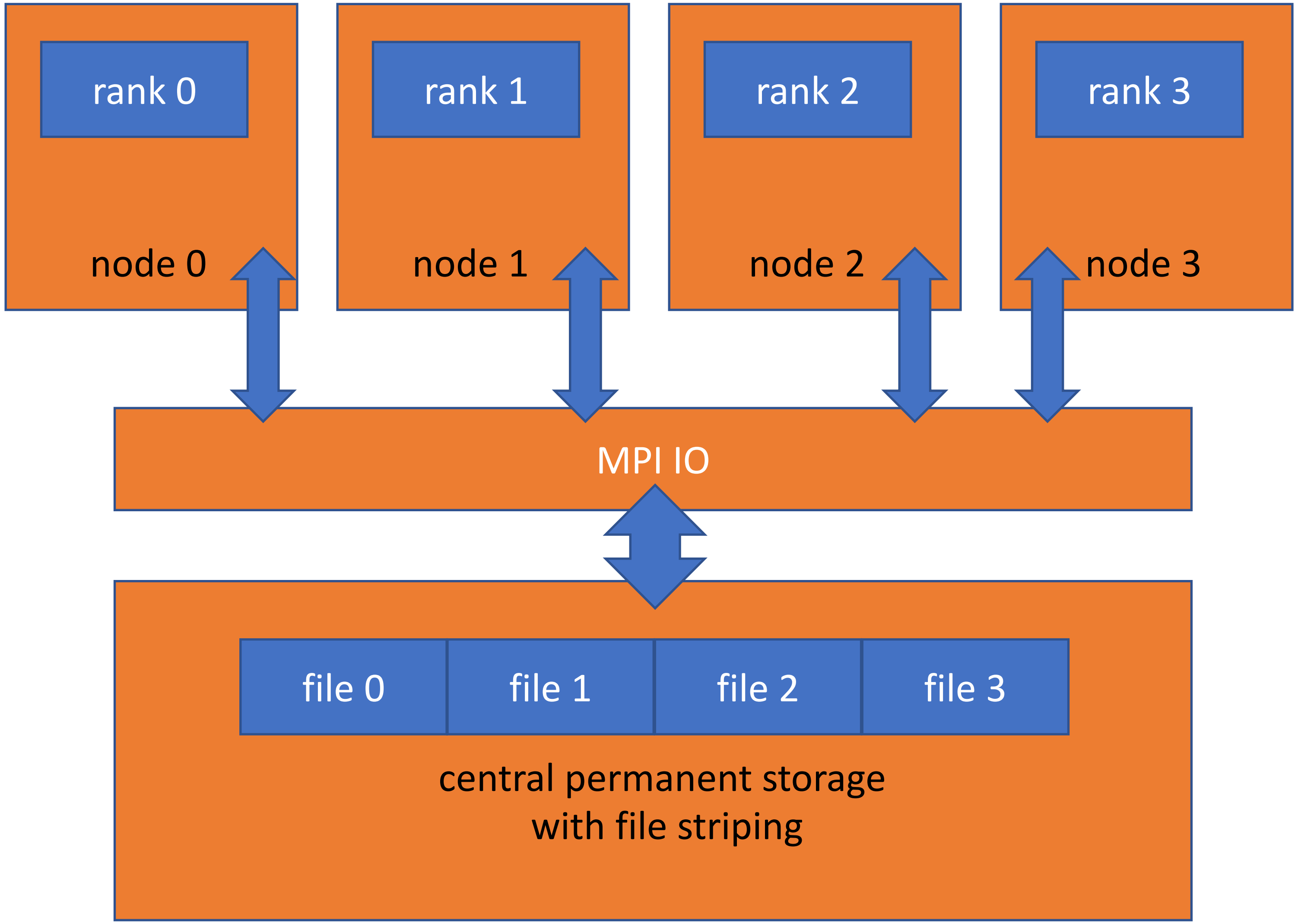
- rank 0: serial access and broadcasts
  → worst-case scenario
- rank 0: reads only a fraction of a file
  → bad for meta data server

# Motivation: MPI I/O 3/4

- Speed up with cooperation and parallelism

MPI IO:

- simultaneous access cooperation
- single file
- provides replacement function for POSIX



# Motivation: MPI & MPI IO 4/4

MPI I/O is based on:

- MPI & parallel FS (→ fast)
- handle read/write accesses like sending/receiving of messages

| parallel I/O requirements | analogy on MPI |
|---|---|
| collective file operations | MPI communicators |
| non-contiguous access | MPI derived datatypes |
| nonblocking operations | MPI functions with immediate return in combination with Wait. |

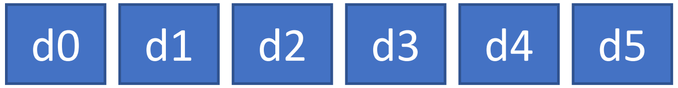
not yet discussed in this lecture

# MPI IO principles

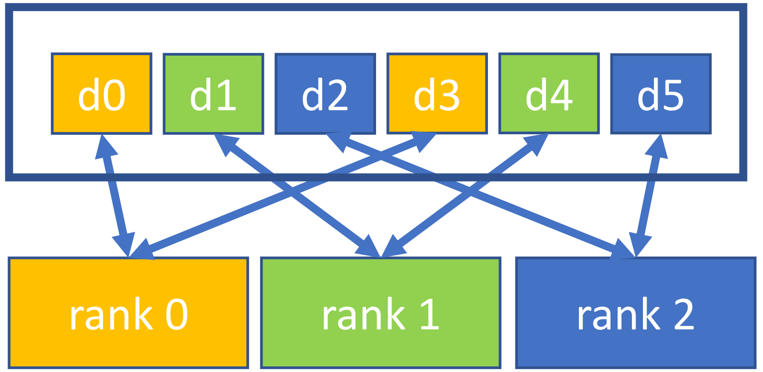
- MPI file contains elements of a single MPI datatype (“etype”)
- rank file access provided by access templates
- read/write routines in MPI IO: nonblocking / blocking and collective / individual reads
- file pointers: individual and shared
- automatic data conversion in heterogenous systems

# MPI: Access possibilities

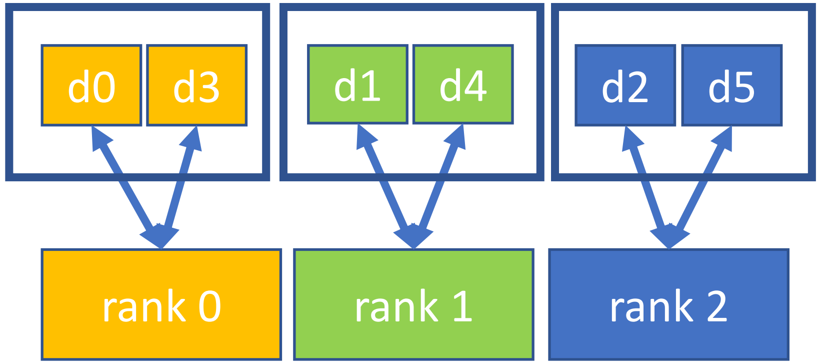
- Array of data in file




- 3 ranks processing this file
  1. full view on file for every rank (like standard POSIX) with `MPI_File_write_at()`




  2. reduced view on file for every rank with `MPI_File_set_view()` and `MPI_File_write()`



# MPI IO: Opening a file

```
int MPI_File_open (
       MPI_Comm comm,
       ROMIO_CONST char *filename,
       int amode,
       MPI_Info info,
       MPI_File *fh /*out*/)
```

- collective within communicator.
  - all processes in comm. call function with same arguments (filename, amode)
  - process-local files with MPI_COMM_SELF as communicator
- returns a file handle
  - representing the file, communicator and the current view (see next slides)
- default:
  - displacement = 0, etype=MPI_BYTE → each process has access to whole file (”slide before: full view”)
- No info = MPI_INFO_NULL, otherwise provide timeouts, buffer sizes or stripe factors here.

# MPI IO: Access Mode

- remember: same amode argument on all processes (collective!)
- combe these arguments bit wise → Operator |    (better not +)
- Be as restrictive as possible to allow for storage optimisation

| Constants | |
|---|---|
| MPI_MODE_APPEND | all file pointers set to end of file |
| MPI_MODE_CREATE | Create the file if it does not exist. |
| MPI_MODE_DELETE_ON_CLOSE | |
| MPI_MODE_EXCL | Error creating a file that already exists. |
| MPI_MODE_RDONLY | Read only. |
| MPI_MODE_RDWR | Reading and writing. |
| MPI_MODE_SEQUENTIAL | only sequential access, eg: tapes |
| MPI_MODE_WRONLY | Write only. |
| MPI_MODE_UNIQUE_OPEN | file not opened concurrently |

caution: any following call of MPI_FILE_SET_VIEW will reset this to 0

# MPI IO: Closing a file

- collective function

```
int MPI_File_close(MPI_File *fh)
```

# MPI IO: File Deletion

1. ```
   int MPI_File_delete(
       ROMIO_CONST char *filename,
       MPI_Info info)
   ```
   - file need not be currently opened

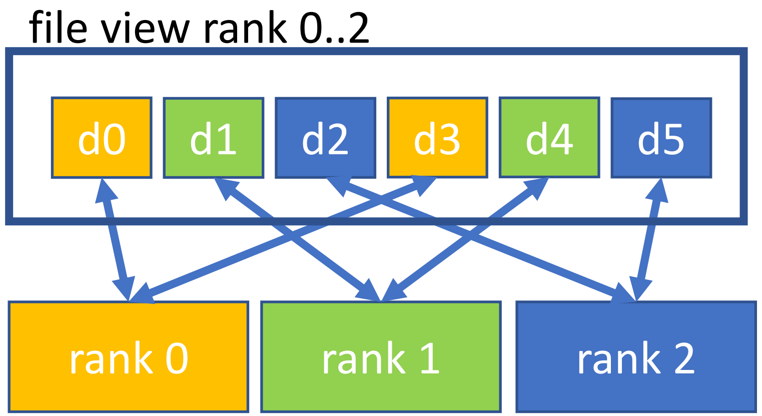
2. Provide argument „amode = MPI_MODE_DELETE_ON_CLOSE“ in MPI_File_Open
   ```

# MPI IO: Writing to file with explicit offset

- (needed for exercise 7)
- `int MPI_File_write_at(`

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

  - buffer includes min count elements of type datatype




- writes count times elements from buffer to to the file
- starting at `offset * sizeof(datatype)` from begin of view

# MPI IO: Reading from a file with explicit offsets

- ```
  int MPI_File_read_at(
        MPI_File fh,
        MPI_Offset offset,
        void *buf,
        int count,
        MPI_Datatype datatype,
        MPI_Status *status)
  ```
- read count elements of datatype
- starting at offset * sizeof(datatype) from begin of view
- EOF is reached, once amount of data read < count
  - use MPI_Get_Count(status, datatype, received_count)
  - note: EOF is no error
  ```

# MPI IO: Individual file pointers 1/2

- `int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`
- `int MPI_File_write(MPI_File fh, ROMIO_CONST void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`
- same functions as those functions with “_at”, except:
  - each process has it’s private current value of file offset (“file pointer”)
  - after access, private offset updates:
    - private offset points to the next datatype of the last accessed.

# MPI IO: Individual file pointers 2/2

- `int MPI_File_seek(MPI_File fh, MPI_Offset offset_new, int whence /*Update mode*/)`
  - Update mode = `MPI_SEEK_SET` → set private file offset to offset_new
  - `MPI_SEEK_CUR` → advance private file offset by offset_new
  - `MPI_SEEK_EOF` → set private file offset to EOF + offset_new

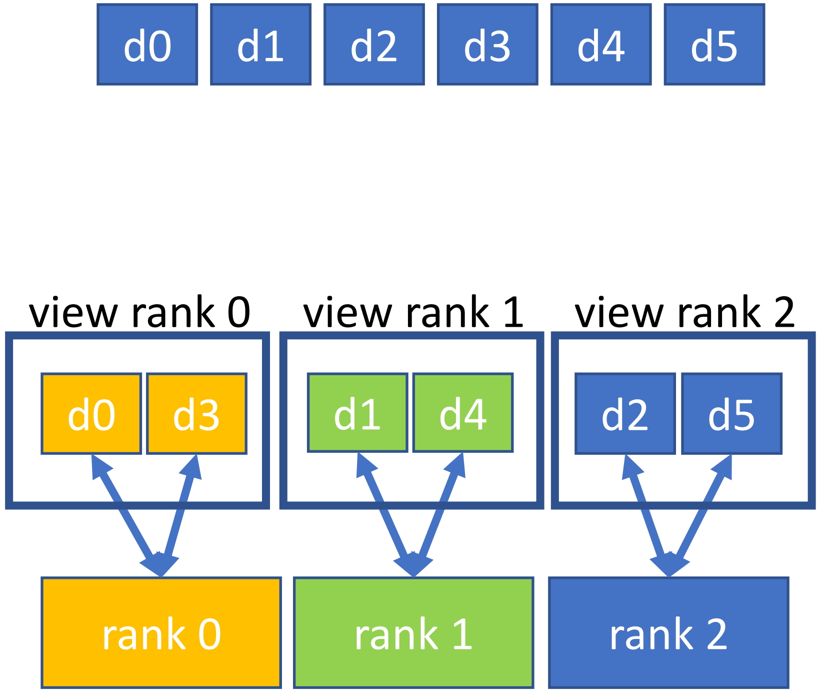
inquire offset:

- `int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)`
- `int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset, MPI_Offset *disp)`
  - `disp` = absolute byte position of offset (nonnegative integer)

- To convert an offset into byte displacement (needed eg for a new view)

# MPI IO: File views 1/2

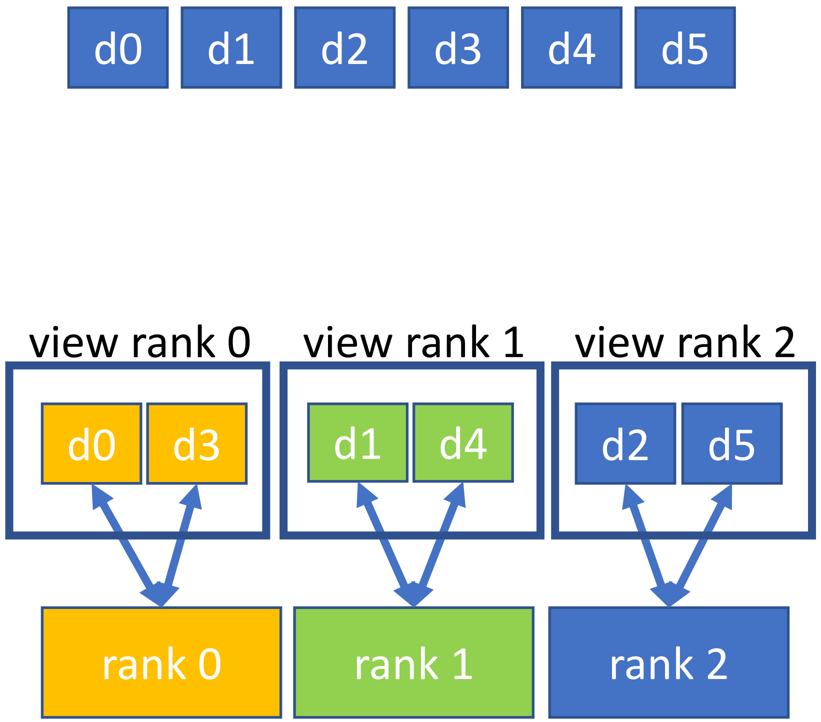
- Each process gets a separate view of the file, collective operation (necessary for exercise 8)
- Defined by (displacement, datatype, filetype)
  - Standard = (0, MPI_BYTE, MPI_BYTE) = linear byte stream
- can be changed during runtime
- ```
  int MPI_File_set_view(
        MPI_File fh,
        MPI_Offset disp,
        MPI_Datatype etype,
        MPI_Datatype filetype,
        ROMIO_CONST char *datarep /*see next
  slide*/,
        MPI_Info info)
  ```
- Get view via `MPI_File_get_view()`


  ```

# MPI IO: File views 2/2

- Worked out example, create MPI_Type filetype first:

```
etype = MPI_CHAR;
ndims = 1;  /*dimensions of following arrays*/
array_of_sizes[0]    = 3;
array_of_subsizes[0] = 1;
array_of_starts[0]   = my_rank;
MPI_Type_create_subarray(ndims,
     array_of_sizes, array_of_subsizes,
     array_of_starts, MPI_ORDER_C, etype,
     &filetype);
MPI_Type_commit(&filetype);
MPI_File_set_view(fh, 0, etype, filetype,…);
```



# MPI IO: Data representation

our choice

- native:
  - data in file = data in memory
  - no type conversions (no loss of precision and I/O performance) on homogenous systems
  - not possible on heterogenous systems
  - no guarantee by MPI to mix C and Fortran
- internal:
  - implementation dependent, for heterogenous systems
- external32
  - follows standardized representation (IEEE)
  - all input/output according to “external32” representation → interoperable between different MPI impl.
  - due to type conversions from/to native: data precision and I/O performance is reduced
  - can be read/written also by non-MPI programs

# Introduction MPI

1. Overview / Getting Started
2. Messages & Point-to-point Communication
3. Nonblocking Communication
4. Error Handling
5. Groups & Communicators
6. Collective Communication
7. MPI I/O
8. MPI Derived Datatypes
9. Common pitfalls and good practice ("need for speed")
10. Debugging and Profiling

# Most common MPI pitfalls:

- Starting multi-core program: do not copy or fork your code, **improve existing**.
- FIRST: optimise single core performance (better algorithm)
- efficiency of MPI application-programming is not portable
  → optimize for every system needed (when aiming for highest speeds)

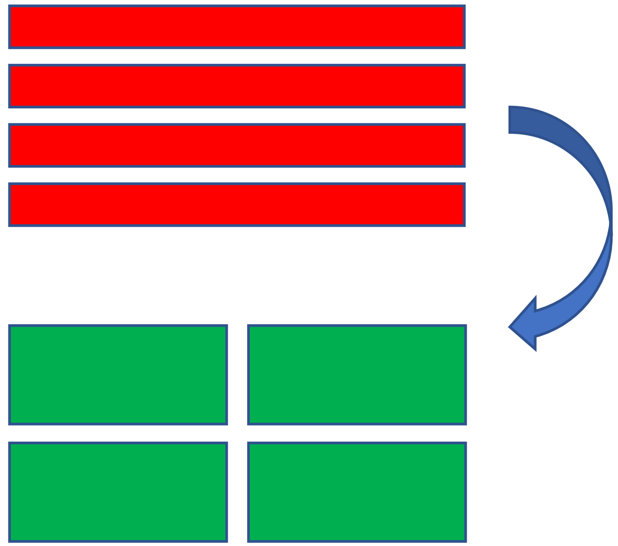
- Most Common pitfalls:
  - Deadlocks and serialization
  - Late sender / late receiver

- further hints:
  - Overlap communication and computation (using “i”-send/-recv)
  - Global communication involving many or all MPI processes include costly synchronizations.
    - combine such reductions to reduce overhead
  - try to share huge buffers instead of copying
  - Check resources, try to avoid local swap → use more machines, less ranks / threads per node

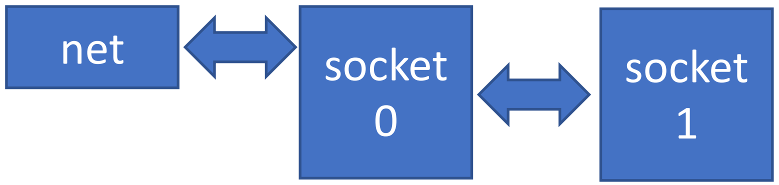
# MPI optimisation: Advanced

- Contention:
  - miss ratio senders / receiver,
  - low bisectional bandwidth between nodes,
  - non ideal network routing

- Non optimal domain decomposition (slicing of your detector/matrix):
  - try different data decomposition
    (divide the problem differently, “slices with smaller surfaces)
  - too much communication overhead,
  - as many ranks as possible on a single node → avoid network



- On multi socket systems: sending rank should be on core in hardware, which is closest to network link
- Check for load imbalances, use tuning tools



# MPI optimisation: Binding

- Binding processes and their threads prevents the OS scheduler from moving them across the available CPU sockets or cores.
- **Memory-bound MPI application** with one MPI process per socket
  - $MPIEXEC $FLAGS_MPI_BATCH --map-by ppr:1:socket --bind-to core a.out
- **Compute-bound MPI application** with as many processes per node as there are cores (try to use HyperThreading)
  - $MPIEXEC $FLAGS_MPI_BATCH --bind-to core --map-by core a.out
- **MPI application with n processes per socket** (n < #cores)
  - MPI applications, that are neither completely compute- nor completely memory-bound, try to run them with less processes per socket than cores available:
  - $MPIEXEC $FLAGS_MPI_BATCH --map-by ppr:2:socket --bind-to core a.out
    ## number of processes per socket ---^
- **Examining the Binding**,
  - OpenMP: `--report-bindings`
  - MPI: `srun --cpu-bind=verbose / mpirun --report-bindings`
  - depends on MPI implementation

# MPI optimisation: Binding

```
mpirun -n 4 --report-bindings -bind-to core ring-sub

[login22:17413] MCW rank 0 bound to socket 0[core 0[hwt 0-1]]: [BB/../../../../../../../../..][../../../../../../../../../..]

[login22:17413] MCW rank 1 bound to socket 1[core 10[hwt 0-1]]: [../../../../../../../../../..][BB/../../../../../../../../..]

[login22:17413] MCW rank 2 bound to socket 0[core 1[hwt 0-1]]: [../BB/../../../../../../../..][../../../../../../../../../..]

[login22:17413] MCW rank 3 bound to socket 1[core 11[hwt 0-1]]: [../../../../../../../../../..][../BB/../../../../../../../..]
```

# MPI: Possible sources of errors

1. Hardware (CPU, RAM, network, storage) free of errors?
2. Check: error free single core program?

- program hangs
  send / receive do not match (sender it, communicator, tag, etc.) → verify parameters
- MPI_Send crashes:
  Buffer address correct? Still correct? eg OpenMP task gets executed with delay (use “omp taskwait”)
- MPI_Recv crashes: MPI library tells, msg is larger than recv buffer
  message from correct sender received? Did tags match? wrong message order? → use unique tag
- received message data is wrong
  Send buffer has been modified (buffered send) before sent / Received buffer has been accessed before arrival of data
- Using OpenMP and MPI in parallel:
  → Tell mpirun about it, use correct MPI multi-thread level (eg MPI_THREAD_SERIALIZED or MPI_THREAD_MULTIPLE)

# MPI I/O Performance

- Best practices of using MPI I/O:
  - make as few file I/O calls in general
  - in order to create big data requests and
  - have as few meta-data accesses (seeks, query or changing of file-size).

- Change MPI_Info key-values, according to your needs, eg:
  - ```
    MPI_Info info;
    MPI_Info_create(&info);
    /* Enable ROMIO's collective buffering */
    MPI_Info_set(info, "romio_cb_read", "enable");
    MPI_Info_set(info, "romio_cb_write", "enable");
    MPI_File_open (MPI_COMM_WORLD, fn, MPI_MODE_CREATE | MPI_MODE_WRONLY, info, &fh);
    ```
    ```

# General File Access Hints

- Bad I/O performance due to:
  - Accessing that same portion of the file → locks
  - Other i/o in parallel
  - random accesses
  - datasize(i/o requests) << filesystem block size
  - files too small / too many files / too many open&closes → metadata servers overloaded
- Avoid data access:
  - Recalculate when it's faster
  - group small operations to larger chunks
  - Reduce data accuracy, possible? → less data!
- Helpful:
  - Use parallel I/O libraries: MPI I/O, HDF5, etc. and use their non-blocking MPI I/O routines
  - large and contiguous requests
  - Use derived datatypes to support MPI I/O in its work
  - Open files in the correct mode (eg only readonly) to allow for optimisations
  - Not too many open files at the same time
  - flushes only when absolutely necessary.
  - Create files independent of the number or processes (easier post processing and restarts with different rank size)

# Optimisation

- Good read for further studies:
    - Hager, Wellein: “Introduction to High Performance Computing for Scientists and Engineers”, CRC Press

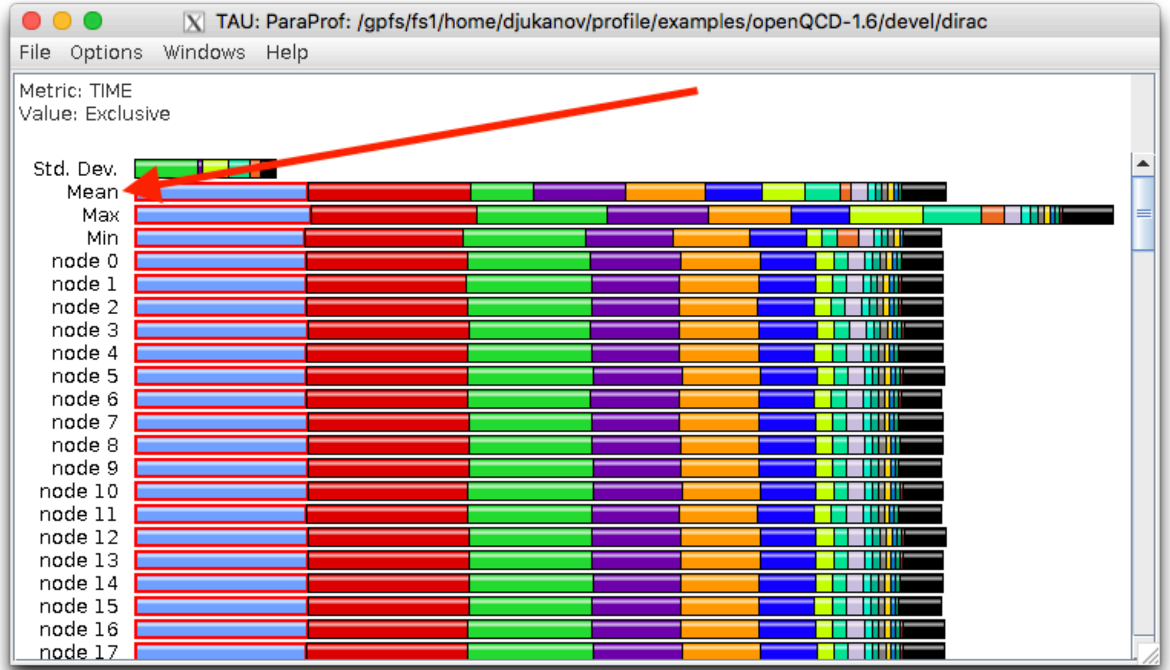
# Introduction MPI

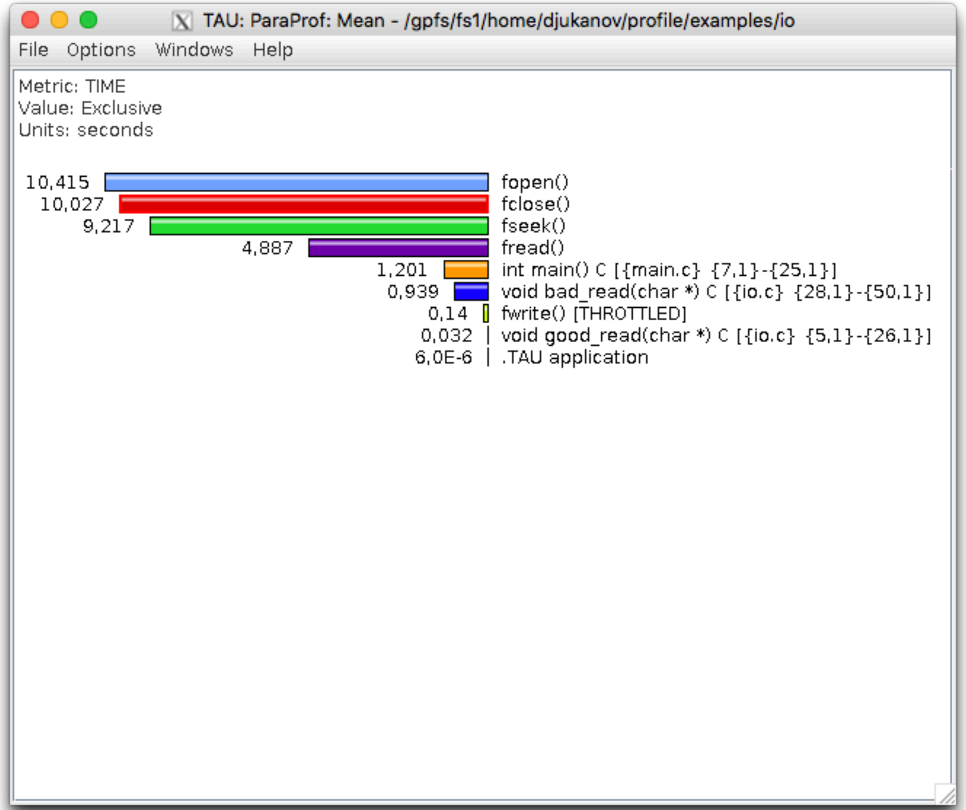
1. Overview / Getting Started
2. Messages & Point-to-point Communication
3. Nonblocking Communication
4. Error Handling
5. Groups & Communicators
6. Collective Communication
7. MPI I/O
8. MPI Derived Datatypes
9. Common pitfalls and good practice (“need for speed”)
10. Debugging and Profiling

topics of future lectures

# MPI: Profiling Glimpse

- See where time is spent
- Identify idle periods



# Exercise 7:

Learning objectives:

- first usage of MPI IO and MPI_File_write_at()

Steps:

1. Clone the skeleton from lecture repository: https://gitlab.rlp.net/pbotte/learnhpc
2. Each rank writes 5 times its rank number into a common file (do not use more than 9 ranks). The output should look like (with 4 ranks): 01230123012301230123

Hints:

- offset = my_rank + Comm_Size * i, i=0..4
- Each process uses the default view
- To write numbers as ASCII characters use buf = '0' + (char)my_rank;
- You can use "cat FILENAME" to check your written output.
- Real world hint: Your home directory is not a parallel FS. For full speed use /lustre/…

# Exercise 8:

Learning objectives:

- Write to a file with MPI_File_set_view

Steps:

1. Clone the skeleton from lecture repository: https://gitlab.rlp.net/pbotte/learnhpc
2. Achieve the same result as in exercise 7 but make use of MPI_Type_create_subarray, MPI_File_set_view and MPI_File_write