PIOSim MPIwrapper

Paul Müller

2009-04-30

The MPI wrapper is a static library that can be used to log MPI function calls which occur during the execution of a program. The log files are intended to be used by the PIOsimHD simulator project.

This document describes the use, internal structure and extension possibilities of the MPI wrapper.

Contents

1 Using the wrapper **4**
- 1.1 Requirements . . . 4
- 1.2 Building the wrapper . . . 4
- 1.3 Building the tests . . . 5
- 1.4 Example: Producing a trace . . . 5

2 Functionality **6**
- 2.1 Goals . . . 6
  - 2.1.1 Tracing method . . . 8
- 2.2 Components . . . 8
  - 2.2.1 HDTraceWritingCLibrary . . . 8
  - 2.2.2 HDTraceMPIWrapper . . . 9
  - 2.2.3 Project Description Merger . . . 9
- 2.3 Design . . . 10
  - 2.3.1 Automatic code generation . . . 10
  - 2.3.2 Log format . . . 10
  - 2.3.3 Files . . . 13
  - 2.3.4 Time . . . 13
  - 2.3.5 Data types . . . 14
  - 2.3.6 Communicators . . . 15
  - 2.3.7 Non-blocking communication . . . 15
  - 2.3.8 Threading . . . 16

3 Environment variables and command line options **16**
3.1 HDTraceMPIWrapper.a . . . 16
3.1.1 HDTRACE_FORCE_FLUSH . . . 16
3.1.2 HDTRACE_NESTED . . . 17
3.1.3 HDTRACE_FILE_INFO . . . 17
3.1.4 HDTRACE_ALL_FUNCTIONS . . . 17
3.2 project-description-merger.py . . . 17
3.2.1 Usage . . . 17

4 Future Work **18**
4.1 Thread safety . . . 18
4.2 File name ambiguity . . . 18
4.3 Packaging . . . 18

A Logged attributes and elements **18**
A.1 Common attributes . . . 18
A.2 List of logged attributes . . . 19
A.3 MPI_Allgather . . . 19
A.4 MPI_Allgatherv . . . 19
A.5 MPI_Allreduce . . . 19
A.6 MPI_Alltoall . . . 19
A.7 MPI_Alltoallv . . . 19
A.8 MPI_Barrier . . . 19
A.9 MPI_Bcast . . . 19
A.10 MPI_Bsend . . . 20
A.11 MPI_Exscan . . . 20
A.12 MPI_File_close . . . 20
A.13 MPI_File_delete . . . 20
A.14 MPI_File_get_size . . . 20
A.15 MPI_File_iread . . . 20
A.16 MPI_File_iread_at . . . 20
A.17 MPI_File_iread_shared . . . 20
A.18 MPI_File_iwrite . . . 21
A.19 MPI_File_iwrite_at . . . 21
A.20 MPI_File_iwrite_shared . . . 21
A.21 MPI_File_open . . . 21
A.22 MPI_File_preallocate . . . 21
A.23 MPI_File_read . . . 21
A.24 MPI_File_read_all . . . 21
A.25 MPI_File_read_all_begin . . . 21
A.26 MPI_File_read_all_end . . . 22
A.27 MPI_File_read_at . . . 22
A.28 MPI_File_read_at_all . . . 22
A.29 MPI_File_read_at_all_begin . . . 22

A.30 MPI_File_read_at_all_end . . . 22
A.31 MPI_File_read_ordered . . . 22
A.32 MPI_File_read_ordered_begin . . . 22
A.33 MPI_File_read_ordered_end . . . 22
A.34 MPI_File_read_shared . . . 23
A.35 MPI_File_seek . . . 23
A.36 MPI_File_seek_shared . . . 23
A.37 MPI_File_set_atomicity . . . 23
A.38 MPI_File_set_info . . . 23
A.39 MPI_File_set_size . . . 23
A.40 MPI_File_set_view . . . 23
A.41 MPI_File_sync . . . 23
A.42 MPI_File_write . . . 24
A.43 MPI_File_write_all . . . 24
A.44 MPI_File_write_all_begin . . . 24
A.45 MPI_File_write_all_end . . . 24
A.46 MPI_File_write_at . . . 24
A.47 MPI_File_write_at_all . . . 24
A.48 MPI_File_write_at_all_begin . . . 24
A.49 MPI_File_write_at_all_end . . . 24
A.50 MPI_File_write_ordered . . . 25
A.51 MPI_File_write_ordered_begin . . . 25
A.52 MPI_File_write_ordered_end . . . 25
A.53 MPI_File_write_shared . . . 25
A.54 MPI_Gather . . . 25
A.55 MPI_Gatherv . . . 25
A.56 MPI_Ibsend . . . 25
A.57 MPI_Iprobe . . . 25
A.58 MPI_Irecv . . . 26
A.59 MPI_Irsend . . . 26
A.60 MPI_Isend . . . 26
A.61 MPI_Issend . . . 26
A.62 MPI_Recv . . . 26
A.63 MPI_Reduce . . . 26
A.64 MPI_Reduce_scatter . . . 26
A.65 MPI_Rsend . . . 26
A.66 MPI_Scan . . . 27
A.67 MPI_Scatter . . . 27
A.68 MPI_Scatterv . . . 27
A.69 MPI_Send . . . 27
A.70 MPI_Sendrecv . . . 27
A.71 MPI_Sendrecv_replace . . . 27
A.72 MPI_Ssend . . . 27
A.73 MPI_Type_commit . . . 27

A.74 MPI_Type_vector 28
A.75 MPI_hdT_Test_nested 28

B Datatype representation in the project description 28
B.1 MPI_COMBINER_DUP 28
B.2 MPI_COMBINER_CONTIGUOUS 28
B.3 MPI_COMBINER_VECTOR 28
B.4 MPI_COMBINER_HVECTOR, MPI_COMBINER_HVECTOR_INTEGER 28
B.5 MPI_COMBINER_RESIZED 28
B.6 MPI_COMBINER_INDEXED 29
B.7 MPI_COMBINER_HINDEXED 29
B.8 MPI_COMBINER_INDEXED_BLOCK 29
B.9 MPI_COMBINER_STRUCT 29
B.10 MPI_COMBINER_STRUCT_INTEGER 29
B.11 MPI_COMBINER_SUBARRAY 29
B.12 MPI_COMBINER_DARRAY 30

1 Using the wrapper

1.1 Requirements

The MPI wrapper project references the HDTraceWritingCLibrary project. It must be built before building the MPI wrapper:

```
cd PIOsimHD/HDTraceWritingCLibrary/
make
```

`libglib` is also needed.

1.2 Building the wrapper

The MPI wrapper library can be built either for the mpich or openmpi implementation of MPI. Due to differences between the implementations you cannot use one version of the wrapper for both.

To build the library for mpich, call make:

```
cd PIOsimHD/PIOsimHD-Maint/mpiwrapperNew/
make
```

To build the library for openmpi, set the environment variable `MPIIMP=openmpi`:

```
cd PIOsimHD/PIOsimHD-Maint/mpiwrapperNew/
MPIIMP=openmpi make
```

1.3 Building the tests

The directory `PIOsimHD/PIOsimHD-Maint/mpiwrapperNew/test/HDTests` contains a number of programs to test the functionality of the wrapper. These tests can be build via

```
cd PIOsimHD/PIOsimHD-Maint/mpiwrapperNew/
make hdtest
```

1.4 Example: Producing a trace

In this example, the PIOsim project tree is located in the home folder, and the necessary libraries have been built as described above. I am going to compile and trace the `mpi-io-test` program [Lab].

```
cd $HOME
mkdir mpi-io-test
cd mpi-io-test/
wget http://mirror.anl.gov/pub/pvfs2/tests/mpi-io-test.c
mpicc -o mpi-io-test mpi-io-test.c \
 $HOME/PIOsimHD/PIOsimHD-Maint/mpiwrapperNew/libs/libHDTraceMPIWrapper.a \
 $HOME/PIOsimHD/HDTraceWritingCLibrary/lib/libhdTrace.a \
 `pkg-config --libs glib-2.0`
mpiexec -n 3 ./mpi-io-test
```

Running the program will produce the following output

```
# Using mpi-io calls.
nr_procs = 3, nr_iter = 1, blk_sz = 16777216, coll = 0
# total_size = 50331648
# Write: min_t = 1.177056, max_t = 1.613190, mean_t = 1.331647, var_t = 0.059641
# Read:  min_t = 0.448252, max_t = 0.544611, mean_t = 0.509273, var_t = 0.002816
Write bandwidth = 31.200076 Mbytes/sec
Read bandwidth = 92.417616 Mbytes/sec
D: [TRACER][node02.1.0] flushLog (hdTrace.c:955): flushing log length: 2172
D: [TRACER][node01.0.0] flushLog (hdTrace.c:955): flushing log length: 2130
D: [TRACER][node01.2.0] flushLog (hdTrace.c:955): flushing log length: 2172
```

This means everything went well and each of the three processes wrote their own log files. The directory now contains three `.trc` and three `.info` files. The latter need to be converted in order to provide a format that is compatible with the simulator software and other tools. The conversion is done via

```
$HOME/PIOsimHD/PIOsimHD-Maint/mpiwrapperNew\
/scripts/project-description-merger.py \
  -o mpi-io-test.proj mpi-io-test_*.trc
```

The resulting four xml trace files can now be processed by other tools. The result of loading the files into HDJumpshot is shown in (1), (2)

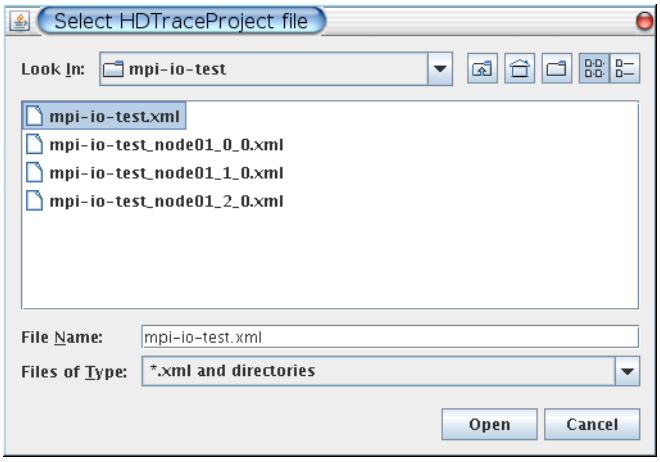

Figure 1: Opening the project description file

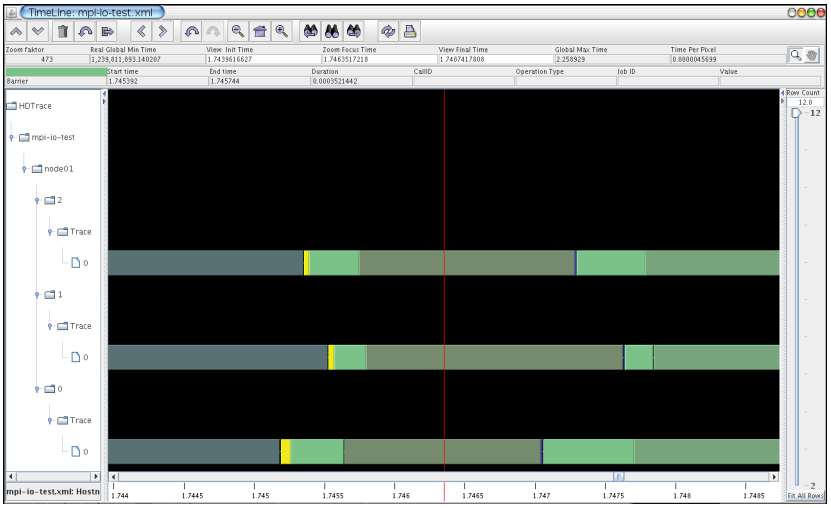

Figure 2: mpi-io-test as displayed by Jumpshot

2 Functionality

2.1 Goals

The main goal of the MPI wrapper is to log the MPI interaction of a parallel program. The log must contain the necessary information to simulate the program at the MPI level and to meaningfully compare the simulation results with the real program execution.

When simulating the execution of traced programs, the simulator assumes the program's logical execution to be independent of the performance of individual MPI functions. It also assumes that the time spent outside of the *important* (i.e. communication and input/output performing) functions does not depend on the performance of those functions. Therefore it suffices to log every call of an *important* function combined with its

start time (time of the call), end time (time of the return) and the relevant function arguments. Relevant arguments are those that indicate which ranks are taking part in a communication, how much data is transferred and which resources are being used.

The use of the following resources must be logged:

1. Files: Every file access via an MPI function must identify, which file is being accessed.
2. Communicators: The communicator indicates, which processes are taking part in a particular collective call and is needed to reconstruct, to which group an MPI call belongs to.
3. Types: Usually, transmission of data is performed on a number of instances of an MPI datatype. Because datatypes can contain holes and there are different strategies to transmit such complex datatypes, it is necessary to log the compositions of the datatypes that are used.
4. Requests: MPI allows the user to use nonblocking calls by returning an `MPI_Request` structure that can be used to poll the state of the call, or, for example, to use a blocking `MPI_Wait`. The latter is relevant for the simulation, because it halts the execution until the corresponding nonblocking call is finished, which must be simulated.
5. Split collective function calls: There are also split collective function calls, that represent a nonblocking, collective access to a file. They are separated into a `*_begin` and an `*_end` routine, where the latter is equivalent to a blocking wait.

There are two main kinds of data that should be logged: Data that references only the local process such as the time and order of MPI function calls, or the request structures that are used to keep track of nonblocking function calls. The other kind has a global meaning, such as files that are seen by all processes and communicators that are created in collective calls and must be equal for all members of the communicator.

Logging of the process-local data can be done in a straightforward fashion where each process writes to its own log file. There are several possibilities to log the global data:

- Create a unique identifier for each global resource on its first use and communicate this identifier to all other processes.
- Create a process internal identifier and store, to which process it refers. Unify the different identifiers in a post-processing step.

Because it is our goal to disturb the program's behaviour as little as possible we choose the second approach. Every process writes two files, an `*.trc` and an `*.info` file. The latter holds the mapping from process-identifiers to the actual global objects (files, communicators). As a minor drawback, the trace must be processed by a script to bring it in a simulator-compatible form.

2.1.1 Tracing method

There are two main requirements that the wrapper library must fulfill:

- It must be easy usable with different MPI implementations, at least with Open MPI and MPICH
- The use of the wrapper must be as easy as possible.

For these reasons, the wrapper is implemented by creating an object file that contains wrapper functions with the same name as the important MPI functions. These functions perform the necessary logging routines, pass the arguments to the underlying MPI library and pass its return value back to the calling program. When the object file is linked to a program, the wrapper functions hide the original MPI functions.

This is reasonably easy and the wrapper is mostly independent of the used MPI implementation. (The `mpi.h` and `mpio.h` headers that are used to compile the MPI wrapper must be the same that are used to compile the traced program.) Also, the `mpicc` compiler can be used as usual.

One disadvantage of the approach is that we need to compile the program that needs to be traced. That means we need to have its source code (which is usually the case). Another problem could occur due to an implementation detail of the wrapper: The wrapper functions need to pass every call to the MPI library. This can easily be done because the library provides every MPI function once using the `MPI_`and once using the `PMPI_` prefix. Thus, passing the function call to the MPI library is done by calling the corresponding PMPI function. This means, the wrapper won't function correctly if the program it is tracing makes use of the `PMPI_` names.

2.2 Components

2.2.1 HDTraceWritingCLibrary

The HDTraceWritingCLibrary is a component that has been split from the MPI wrapper for better reusability. It offers an interface for writing xml logs. It is possible to log xml tags including attributes and elements, and it supports nesting of xml tags up to a certain (defined) depth.

The main logging item is the *state*, which is indicated by `hdT_logStateStart()` and `hdT_logStateEnd()`

```
// trace is the structure that indicates the files to which data
// is being written. It is created by hdT_createTrace(...)
hdT_logStateStart(trace, "StateName");
// execution of the MPI calls,
// logging of elements and attributes that belong to the state
hdT_logStateEnd(trace)
```

It also uses `gettimeofday` to log the time at which the state has been started and

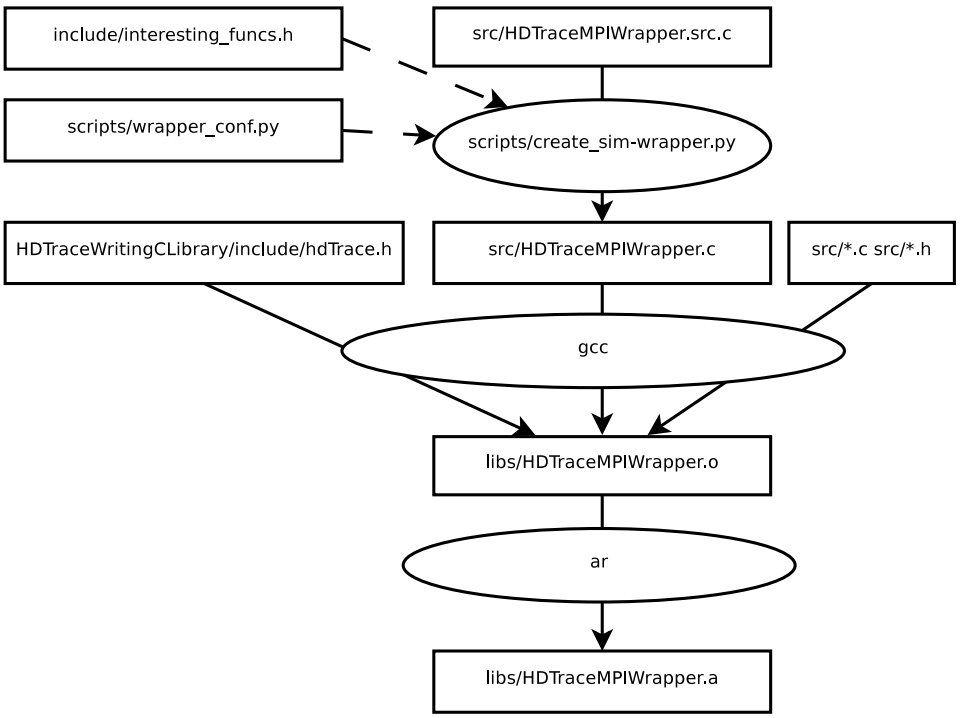

Figure 3: Building the MPI wrapper library from source files.

finished.

There are two files that are created for each trace. One is the xml file that holds data about the states that were logged. This file is compatible with the logging format that is expected by other tools. The second file is a plaintext (`.info`) file that can be used to store additional information.

2.2.2 HDTraceMPIWrapper

The mpi wrapper is the component that actually intercepts the MPI calls and that determines which data is being logged.

2.2.3 Project Description Merger

The script `project-description-merger.py` converts the trace of a program into a form that is compliant with other PIOsim tools. More precise, it uses data from the `.info`-files to create a project description xml file. The collection of xml files – the xml trace files created by all processes and the new description xml – constitute a complete trace. The `.info` files are then no longer needed.

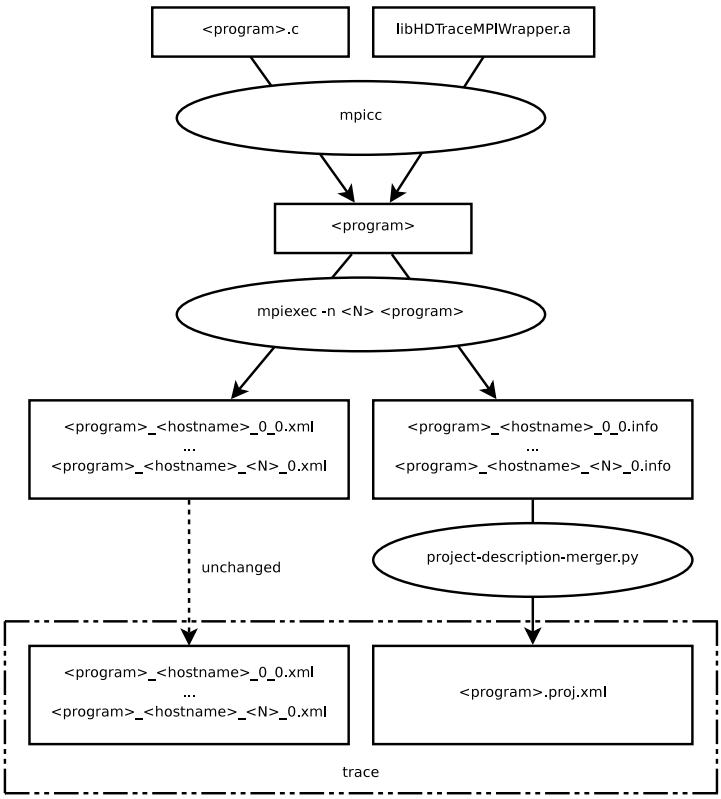

Figure 4: Creating and processing a trace

2.3 Design

This section describes some decisions that were made concerning the design of the MPI wrapper.

2.3.1 Automatic code generation

The wrapper functions that constitute the vital part of the MPI wrapper are very similar in their structure. Because of that, they are automatically generated by a script. This approach, while adding another layer of complexity, greatly reduces possible mistakes that would occur otherwise. The script obtains a list of functions from the file `interesting_funcs.h`. The connections – which function arguments are recorded with which attribute name – are stored in the file `scripts/wrapper_conf.py`.

2.3.2 Log format

It is necessary that the output of the trace library must be compatible with the tracing format that is used by the simulator, HDJumpshot etc. This means a set of xml files that

contain the MPI function calls and their arguments, and one single project description file that contains the information about the names of the other xml files and miscellaneous information. The naming of the process files must fulfill the naming convention

```
<base name>_<level 1>_..._<level K>.trc
```

`<base name>` is the name of the project and `<level 1>` ... `<level K>` constitute the representation of the file in a tree structure.

The MPI wrapper uses the following topology and naming convention:

- `<base name>` is the name of the executable that is being traced. (It is extracted from `*argv[0]` of the arguments to `MPI_Init()`
- `<level 1>` is the hostname of the machine on which the process is running. It is obtained by a call to `gethostname()`
- `<level 2>` is the MPI rank of the process. (`MPI_Comm_rank()`)
- `<level 3>` is the number of the thread that belongs to the current process. The enumeration of threads depends on the order in which `MPI_Init_thread()` or `MPI_Init` is called and is usually arbitrary.

As discussed earlier, the project description file is not generated at runtime because it would require communication between the processes. Because we do not want to disrupt the program's original MPI communication, the description file has to be created in a postprocessing step. The postprocessing is done by the script `project-description-merger.py`.

The project description file is an xml file with the root element `Application`.

It contains the following sections:

- *FileList* This section contains the files that have been accessed by the program and their initial sizes.

  Listing 1: FileList section

  ```
  <FileList>
  <File name="test.out">
   <InitialSize>16777216</InitialSize>
   <Distribution
  class="de.hd.pvs.piosim.model.inputOutput.distribution.SimpleStripe" />
   <ChunkSize>64K</ChunkSize>
  </File>
  </FileList>
  ```

  The `Distribution` and `ChukSize` elements are not derived from the trace file. They are relevant for the MPI simulator and can be set by passing the corresponding command line arguments to the merger script.

- *Topology* The topology section holds the information about the files that constitute the complete trace file. The `Level` entries determine the names of the different levels. Those are fixed for program traces, with the values *Hostname*, *Rank*, *Thread*.

  ```
   <Topology>
  ```

```
  <Level name="Hostname">
   <Level name="Rank">
    <Level name="Thread">
    </Level>
   </Level>
  </Level>
```

The *Label* section determines the names of the single xml trace files. If, for example, the Label section contains the following entries, the folder must also contain the files named `<programname>_node01_0_0.trc` and `<programname>_node01_1_0.trc`.

```
  <Label value="node01">
   <Label value="1">
    <Label value="0" />
   </Label>
   <Label value="0">
    <Label value="0" />
   </Label>
  </Label>
 </Topology>
```

- *CommunicatorList* This section lists all the communicators that are used in any traced call. The communicator's name, the participating ranks (`global`, relative to `MPI_COMM_WORLD`), the ID that this rank has relative to the new communicator (`local`) and the ID that the communicator has in the tracefile (`cid`).

  In the following example there are five ranks that each refer to the world communicator by the ID 0, and a second communicator where rank 1 has the id one and rank 2 has the id 0. Rank 1 refers to the custom communicator by 1 and rank 2 by 2.

```
<CommunicatorList>
  <Communicator name="WORLD">
   <Rank global="1" local="1" cid="0" />
   <Rank global="0" local="0" cid="0" />
   <Rank global="4" local="4" cid="0" />
   <Rank global="3" local="3" cid="0" />
   <Rank global="2" local="2" cid="0" />
  </Communicator>
  <Communicator name="">
   <Rank global="1" local="1" cid="1" />
   <Rank global="2" local="0" cid="2" />
  </Communicator>
```

- *Datatypes* The Datatypes section lists the datatypes that are used by each rank. In the following example, rank 1 uses the predefined, named datatype `MPI_INT` while rank 0 does not use any datatypes. The xml log of rank 1 refers to the datatype by the numerical id 1275069445.

```
 <Datatypes>
  <Rank name="1" thread="0">
  <NAMED id="1275069445" name="MPI_INT"  />
  </Rank>
  <Rank name="0" thread="0">
  </Rank>
 </Datatypes>
```

The name of the tag that represents a datatype is the name of the constant that is returned by `MPI_Type_get_envelope`, without the `MPI_` prefix.

The datatypes are stored per process, so it would have been reasonable to put this section into the individual xml traces. This has not been done because the xml traces should remain unchanged in the postprocessing step.

2.3.3 Files

An MPI function can access a file by using an MPI file handle, a file name or both. An example for the first one would be `MPI_File_write`, `MPI_File_delete` for the second and `MPI_File_open` for the last one. Independent on the method we need to log, which file is being accessed. This is implemented by assigning an id to each file that is used. When the file is opened via `MPI_File_open` for the first time, the id is stored in one hash map where the file name as the key and in a second map where the key is the file handle. When the file is closed, the handle is removed from the second hash map. On any consecutive openinig, the new file handle can be associated with the original identifier via the file name to id map.

The mapping and assigning of identifiers is carried out by the functions

```
static gint getFileId(MPI_File fh);
static gint getFileIdFromName(const char * name);
static gint getFileIdEx(MPI_File fh, const char * name);
```

Whenever a new identifier is assigned, these functions write the file name, the id and the original size to the info file:

```
File name="filetest_03.tmp" Size=0 id=2
```

In the xml trace, the file is refered to by the id (attribute `file`) and the file name on opening:

```
<File_write_at_all_begin file='2' aid='0'
      time='1240424724.531647' end='1240424724.531654' >
  <Data offset='0' size='1' count='1' type='1275068673' />
</File_write_at_all_begin>
```

2.3.4 Time

The hdTrace library sets a time stamp on the beginning and end of every state. The time is obtained by `gettimeofday`. It is represented by the attributes `time` for the start of the state and `end` for its end.

```
<Init  time='1240424724.508182' end='1240424724.508183'  />
```

When the trace is run on different machines, the time must be properly synchronised (via .

2.3.5 Data types

MPI offers the possibility to create and use custom datatypes to optimize access and transmission of complex structures. Datatypes are also used to set the visibility of certain parts of a file.

Because of that it is necessary to identify data types and to log their structure so access to a file can be reconstructed correctly. The exact composition of a datatype is also needed for performance analysis.

An MPI datatype can be created by a number of different functions that take a different number of arguments. Fortunately, it offers a way of reconstructing the call that was used to create a type: First, `MPI_Type_get_envelope` has to be called to get the number of arguments that were passed to the type-creating function, and the *combiner* that identifies the creating function. A second call to `MPI_Type_get_contents` can be used to obtain the arguments themselves. This mechanism is used to log the datatypes to the info file.

Similar to other structures, MPI datatypes are assigned an identifier on their first use by a logging routine. This happens by calling the function `getTypeId(MPI_Datatype type)`. `getTypeId` looks up in a hashmap, if the datatype already has an identifier. If not, it assigns a new id and calls `writeTypeInfo(...)` which writes the creation arguments to the info file. If the datatype is a composet datatype that consists of other datatypes, those are referenced while writing the info file, which causes them to be logged as well. In the end, all the information that is needed to reconstruct the datatype will be recursively logged.

For example, creating a struct consisting of four `MPI_INT`s and five `MPI_DOUBLE`s will produce the following output. `id` is the integer by which the datatype is refered to in the log file. `combiner` is the name of the MPI constant that identifies by which function the type has been created. `integers`, `addresses` and `types` are the parameters that have been passed to that function. `types` uses the trace-intern identifiers to refer to the types.

```
Type id='1275069445' combiner='MPI_COMBINER_NAMED' name='MPI_INT'
Type id='1275070475' combiner='MPI_COMBINER_NAMED' name='MPI_DOUBLE'
Type id='-1946157050' combiner='MPI_COMBINER_STRUCT' name='' \
  integers='2;4;5;' addresses='6;7;' types='1275069445;1275070475;'
```

This is the code segment that would produce the above output.

```
int blens[2] = {4, 5};
MPI_Aint inds[2] = {6, 7};
MPI_Datatype oldtypes[2] = {MPI_INT, MPI_DOUBLE};
MPI_Type_struct(2, blens, inds, oldtypes, &type6);
// use type6 in a transmission that is logged
```

2.3.6 Communicators

There are two main issues with communicators: We need to know who is participating in a transmission, and the translation between global ranks to the ranks that are assigned relative to the communicator.

The logging takes place in a straightforward fashion: Every communicator gets an identifier on the first access and this identifier and the translation map is written to the info file. The name of the communicator is also written, albeit it is rarely useful in cases other than the world and the self communicators.

In the following example the program used three communicators, the world communicator, one communicator with the global ranks 1 and 2 and one with the global ranks 1 and three.

```
Comm map='0->0;1->1;2->2;3->3;4->4;' id=0 name='WORLD'
Comm map='1->1;2->0;' id=1 name=''
Comm map='1->1;3->0;' id=2 name=''
```

2.3.7 Non-blocking communication

There are two types of non-blocking communication provided by MPI.

- `MPI_I` non-blocking function calls that return an `MPI_Request` structure that can be used to query the state of the call.
- Collective I/O operations can be performed using a split collective call. This means that the function, e.g. `MPI_File_read_all` is split into `MPI_File_read_all_begin` and `MPI_File_read_all_end`. Only one split collective call may be active for each file.

Because of semantic similarities between the two mechanisms they share the logging representation. Similar to the logging of files, communicators and datatypes, each `MPI_Request` is assigned an id by which it is refered to when logging related calls.

The first version is logged like this:

```
<Isend size='1' count='1' type='1275068673' toRank='1' toTag='0' \
  cid='0' rid='1'\  time='1240497420.078912' end='1240497420.078981'  />
<Isend size='1' count='1' type='1275068673' toRank='2' toTag='0' \
  cid='0' rid='2'  time='1240497420.078987' end='1240497420.079006'  />
<Wait  time='1240497420.079011' end='1240497420.079028' >
  <For rid='1' />
</Wait>
<Wait  time='1240497420.079034' end='1240497420.079037' >
  <For rid='2' />
</Wait>
```

Here, two non-blocking `Isend`s are issued with the request identifiers (`rid`) 1 and 2. Then the program watis for the first and second sending to complete.

The split collective calls are also assigned an `rid`, which is bound to the file handle. This works because each file may have at most one open split collective call. The `_end` part of the call is then logged exactly like a `Wait`.

This is, for example, how a `MPI_File_write_at_all_begin` - `MPI_File_write_at_all_end` pair is logged:

```
<File_write_at_all_begin fid='0' rid='0'  time='1240497420.073870' end
    ='1240497420.076409' >
  <Data offset='0' size='1' count='1' type='1275068673' />
</File_write_at_all_begin>
<Wait  time='1240497420.076415' end='1240497420.076418' >
  <For rid='0' />
</Wait>
```

2.3.8 Threading

The wrapper has been programmed with threading in mind. The TLS (thread local storage) mechanism of the gcc compiler is used to ensure that global variables are limited to the current thread. Also, `MPI_Init` (or `MPI_Init_thread`) are using a counter to keep track, how often they have been called. This is used to give each thread a separate tracefile. The thread number is the last number in the log file names.

It must be noted that thread support has not been carefully tested. Also, the project description merger handles trace files from different ranks and trace files from different threads in the same way.

3 Environment variables and command line options

3.1 HDTraceMPIWrapper.a

The behaviour of the tracing library can be adjusted by setting certain environment variables before running a program. For example, you can force flushing the write buffer on each write by setting `HDTRACE_FORCE_FLUSH`:

```
HDTRACE_FORCE_FLUSH=1 ./mpi-io-test
```

The following variables are supported:

3.1.1 HDTRACE_FORCE_FLUSH

`HDTRACE_FORCE_FLUSH=0` means, the wrapper flushes the write buffer on its own discretion, usually only when the buffer is full or the file is closed. This is the default.

`HDTRACE_FORCE_FLUSH=1` causes the wrapper to flush the write buffer on every write.

3.1.2 HDTRACE_NESTED

This variable influences, whever nested function calls should be traced. A nested call is a call to an MPI routine from within an MPI routine.

`HDTRACE_NESTED=0` Only the outer function call is being logged.

`HDTRACE_NESTED=1` Nested calls up to a certain depth are logged. The depth is determined by the constant `HD_LOG_MAX_DEPTH }, defined in {\verb hdTrace.h }`

3.1.3 HDTRACE_FILE_INFO

This variable determines, if the MPI_Info structure that is given to `MPI_File_delete`, `MPI_File_set_view`, `MPI_File_set_info } or {\verb MPI_File_open } should be logged.`

`HDTRACE_FILE_INFO=0` The file information is not logged

`HDTRACE_FILE_INFO=1` The file information is logged

3.1.4 HDTRACE_ALL_FUNCTIONS

This variable determines, if ordinary functions should be logged. An ordinary function is one that appears in `include/interesting_funcs.h } but does not have any elements or`attri other than start and end time being logged.

`HDTRACE_ALL_FUNCTIONS=0` ordinary functions are not logged.

`HDTRACE_ALL_FUNCTIONS=1` ordinary functions are logged. This is the default.

3.2 project-description-merger.py

The project description merger creates a single xml file from the collection of info files that have been written by the different MPI processes.

3.2.1 Usage

```
project-description-merger.py -o <outfile.trc> [-d <description>] \
[--distribution-class=<class>] \
[--chunk-size=<size>] \
<log1>.info <log2>.info ... <logN>.info
```

`-o <outfile>` (required): The output file. The naming conventions for the trace files are `<program-name>_<hostname>_<rank>_<thread>.trc`.

It is advised to use `<program-name>.trc` as output filename. This is expected by other PIOsim applications.

`-d <description>` (optional): The description of the project. This can be anything.

4 Future Work

4.1 Thread safety

The wrapper has been programmed for being usable with multithreaded MPI applications. However, this has not yet been excessively tested.

4.2 File name ambiguity

If the same file is accessed using different names it will appear in the log as two different files. Using a `realpath()`-like function is advisable. Unfortunately, `realpath()` does not handle pvfs filenames

4.3 Packaging

Currently, the necessary MPI headers are copied into the project's include directory. This should be handled by the configuration script.

A Logged attributes and elements

This section lists the format of the xml log file. Each xml tag has the name of the logged function without `MPI_` prefix. The type of the values is given in printf-compatible notation.

A.1 Common attributes

size An approximation of the size of the data that is accessed, in bytes. It is calculated by taking the product of the element count and the type size as returned by `MPI_Type_size`.

cid The id of the communicator that is used

type The id of the type that is used.

time The time at which the MPI function is called.

end The time at which the MPI function returned.

fid The id of the file that is referenced.

rid The id of the request that is used.

A.2 List of logged attributes

A.3 MPI_Allgather

```
<Allgather size='%lld' recvSize='%lld' cid='%d' count='%d' tid='%d' recvCount='%d'
    recvtid='%d' time='%f' end='%f' />
```

A.4 MPI_Allgatherv

```
<Allgatherv size='%lld' cid='%d' count='%d' tid='%d' time='%f' end='%f' />
```

A.5 MPI_Allreduce

```
<Allreduce size='%lld' cid='%d' count='%d' tid='%d' time='%f' end='%f' />
```

A.6 MPI_Alltoall

```
<Alltoall size='%lld' cid='%d' count='%d' tid='%d' time='%f' end='%f' />
```

A.7 MPI_Alltoallv

```
<Alltoallv cid='%d' time='%f' end='%f' />
```

A.8 MPI_Barrier

```
<Barrier cid='%d' time='%f' end='%f' />
```

A.9 MPI_Bcast

```
<Bcast size='%lld' rootRank='%d' cid='%d' count='%d' tid='%d' time='%f' end='%f' />
```

A.10 MPI_Bsend

```
<Bsend size='%lld' count='%d' tid='%d' toRank='%d' toTag='%d' cid='%d' time='%f' end='%
    f' />
```

A.11 MPI_Exscan

```
<Exscan size='%lld' cid='%d' count='%d' tid='%d' time='%f' end='%f' />
```

A.12 MPI_File_close

```
<File_close fid='%d' time='%f' end='%f' />
```

A.13 MPI_File_delete

```
<File_delete fid='%d' time='%f' end='%f' />
```

A.14 MPI_File_get_size

```
<File_get_size fid='%d' size='%lld' time='%f' end='%f' />
```

A.15 MPI_File_iread

```
<File_iread fid='%d' rid='%d' offset='%lld' size='%lld' count='%d' tid='%d' time='%f'
    end='%f' />
```

A.16 MPI_File_iread_at

```
<File_iread_at fid='%d' rid='%d' offset='%lld' size='%lld' count='%d' tid='%d' time='%f
    ' end='%f' />
```

A.17 MPI_File_iread_shared

```
<File_iread_shared fid='%d' rid='%d' time='%f' end='%f' />
```

A.18 MPI_File_iwrite

```
<File_iwrite fid='%d' rid='%d' offset='%lld' size='%lld' count='%d' tid='%d' time='%f'
    end='%f' />
```

A.19 MPI_File_iwrite_at

```
<File_iwrite_at fid='%d' rid='%d' offset='%lld' size='%lld' count='%d' tid='%d' time='%
    f' end='%f' />
```

A.20 MPI_File_iwrite_shared

```
<File_iwrite_shared fid='%d' rid='%d' time='%f' end='%f' />
```

A.21 MPI_File_open

```
<File_open cid='%d' name='%s' flags='%d' fid='%d' time='%f' end='%f' />
```

A.22 MPI_File_preallocate

```
<File_preallocate fid='%d' size='%lld' time='%f' end='%f' />
```

A.23 MPI_File_read

```
<File_read fid='%d' offset='%lld' size='%lld' count='%d' tid='%d' time='%f' end='%f' />
```

A.24 MPI_File_read_all

```
<File_read_all fid='%d' offset='%lld' size='%lld' count='%d' tid='%d' time='%f' end='%f
    ' />
```

A.25 MPI_File_read_all_begin

```
<File_read_all_begin fid='%d' rid='%d' time='%f' end='%f' />
```

A.26 MPI_File_read_all_end

```
<File_read_all_end  time='%f' end='%f' />
```

A.27 MPI_File_read_at

```
<File_read_at fid='%d' offset='%lld' size='%lld' count='%d' tid='%d' time='%f' end='%f'
     />
```

A.28 MPI_File_read_at_all

```
<File_read_at_all fid='%d' offset='%lld' size='%lld' count='%d' tid='%d' time='%f' end
    ='%f' />
```

A.29 MPI_File_read_at_all_begin

```
<File_read_at_all_begin fid='%d' rid='%d' offset='%lld' size='%lld' count='%d' tid='%d'
     time='%f' end='%f' />
```

A.30 MPI_File_read_at_all_end

```
<File_read_at_all_end  time='%f' end='%f' />
```

A.31 MPI_File_read_ordered

```
<File_read_ordered fid='%d' size='%lld' count='%d' tid='%d' time='%f' end='%f' />
```

A.32 MPI_File_read_ordered_begin

```
<File_read_ordered_begin fid='%d' rid='%d' time='%f' end='%f' />
```

A.33 MPI_File_read_ordered_end

```
<File_read_ordered_end  time='%f' end='%f' />
```

A.34 MPI_File_read_shared

```
<File_read_shared fid='%d' size='%lld' count='%d' tid='%d' time='%f' end='%f' />
```

A.35 MPI_File_seek

```
<File_seek fid='%d' relative-offset='%lld' whence='%s' offset='%lld' time='%f' end='%f'
     />
```

A.36 MPI_File_seek_shared

```
<File_seek_shared fid='%d' relative-offset='%lld' whence='%s' offset='%lld' time='%f'
    end='%f' />
```

A.37 MPI_File_set_atomicity

```
<File_set_atomicity fid='%d' flag='%d' time='%f' end='%f' />
```

A.38 MPI_File_set_info

```
<File_set_info fid='%d' time='%f' end='%f' />
```

A.39 MPI_File_set_size

```
<File_set_size fid='%d' size='%lld' time='%f' end='%f' />
```

A.40 MPI_File_set_view

```
<File_set_view fid='%d' offset='%lld' etid='%d' filetid='%d' representation='%s' time
    ='%f' end='%f' />
```

A.41 MPI_File_sync

```
<File_sync fid='%d' time='%f' end='%f' />
```

A.42 MPI_File_write

```
<File_write fid='%d' offset='%lld' size='%lld' count='%d' tid='%d' time='%f' end='%f'
    />
```

A.43 MPI_File_write_all

```
<File_write_all fid='%d' offset='%lld' size='%lld' count='%d' tid='%d' time='%f' end='%
    f' />
```

A.44 MPI_File_write_all_begin

```
<File_write_all_begin fid='%d' rid='%d' time='%f' end='%f' />
```

A.45 MPI_File_write_all_end

```
<File_write_all_end  time='%f' end='%f' />
```

A.46 MPI_File_write_at

```
<File_write_at fid='%d' offset='%lld' size='%lld' count='%d' tid='%d' time='%f' end='%f
    ' />
```

A.47 MPI_File_write_at_all

```
<File_write_at_all fid='%d' offset='%lld' size='%lld' count='%d' tid='%d' time='%f' end
    ='%f' />
```

A.48 MPI_File_write_at_all_begin

```
<File_write_at_all_begin fid='%d' rid='%d' time='%f' end='%f' />
```

A.49 MPI_File_write_at_all_end

```
<File_write_at_all_end  time='%f' end='%f' />
```

A.50 MPI_File_write_ordered

```
<File_write_ordered fid='%d' size='%lld' count='%d' tid='%d'  time='%f' end='%f' />
```

A.51 MPI_File_write_ordered_begin

```
<File_write_ordered_begin fid='%d' rid='%d' time='%f' end='%f' />
```

A.52 MPI_File_write_ordered_end

```
<File_write_ordered_end  time='%f' end='%f' />
```

A.53 MPI_File_write_shared

```
<File_write_shared fid='%d' size='%lld' count='%d' tid='%d' time='%f' end='%f' />
```

A.54 MPI_Gather

```
<Gather size='%lld' recvSize='%lld' root='%d' cid='%d' count='%d' tid='%d' recvCount='%
    d' recvtid='%d' time='%f' end='%f' />
```

A.55 MPI_Gatherv

```
<Gatherv size='%lld' root='%d' cid='%d' count='%d' tid='%d' time='%f' end='%f' />
```

A.56 MPI_Ibsend

```
<Ibsend size='%lld' count='%d' tid='%d' toRank='%d' toTag='%d' cid='%d' rid='%d' time
    ='%f' end='%f' />
```

A.57 MPI_Iprobe

```
<Iprobe source='%d' tag='%d' cid='%d' time='%f' end='%f' />
```

A.58 MPI_Irecv

```
<Irecv fromRank='%d' fromTag='%d' cid='%d' rid='%d' time='%f' end='%f' />
```

A.59 MPI_Irsend

```
<Irsend size='%lld' count='%d' tid='%d' toRank='%d' toTag='%d' cid='%d' rid='%d' time
    ='%f' end='%f' />
```

A.60 MPI_Isend

```
<Isend size='%lld' count='%d' tid='%d' toRank='%d' toTag='%d' cid='%d' rid='%d' time='%
    f' end='%f' />
```

A.61 MPI_Issend

```
<Issend size='%lld' count='%d' tid='%d' toRank='%d' toTag='%d' cid='%d' rid='%d' time
    ='%f' end='%f' />
```

A.62 MPI_Recv

```
<Recv fromRank='%d' fromTag='%d' cid='%d' time='%f' end='%f' />
```

A.63 MPI_Reduce

```
<Reduce size='%lld' rootRank='%d' cid='%d' count='%d' tid='%d' time='%f' end='%f' />
```

A.64 MPI_Reduce_scatter

```
<Reduce_scatter cid='%d' time='%f' end='%f' />
```

A.65 MPI_Rsend

```
<Rsend size='%lld' count='%d' tid='%d' toRank='%d' toTag='%d' cid='%d' time='%f' end='%
    f' />
```

A.66 MPI_Scan

```
<Scan size='%lld' cid='%d' count='%d' tid='%d' time='%f' end='%f' />
```

A.67 MPI_Scatter

```
<Scatter size='%lld' recvSize='%lld' root='%d' cid='%d' count='%d' tid='%d' recvCount'%
    d' recvtid='%d' time='%f' end='%f' />
```

A.68 MPI_Scatterv

```
<Scatterv recvSize='%lld' root='%d' cid='%d' recvCount='%d' recvtid='%d' time='%f' end
    ='%f' />
```

A.69 MPI_Send

```
<Send size='%lld' count='%d' tid='%d' toRank='%d' toTag='%d' cid='%d' time='%f' end='%f
    ' />
```

A.70 MPI_Sendrecv

```
<Sendrecv size='%lld' toRank='%d' toTag='%d' fromRank='%d' fromTag='%d' cid='%d' count
    ='%d' tid='%d' time='%f' end='%f' />
```

A.71 MPI_Sendrecv_replace

```
<Sendrecv_replace sendSize='%lld' toRank='%d' toTag='%d' fromRank='%d' fromTag='%d' cid
    ='%d' count='%d' tid='%d' time='%f' end='%f' />
```

A.72 MPI_Ssend

```
<Ssend size='%lld' count='%d' tid='%d' toRank='%d' toTag='%d' cid='%d' time='%f' end='%
    f' />
```

A.73 MPI_Type_commit

```
<Type_commit tid='%d' time='%f' end='%f' />
```

A.74 MPI_Type_vector

```
<Type_vector oldTid='%d' time='%f' end='%f' />
```

A.75 MPI_hdT_Test_nested

```
<hdT_Test_nested depth='%d' time='%f' end='%f' />
```

B Datatype representation in the project description

B.1 MPI_COMBINER_DUP

```
<DUP id="%d" name="%s" oldType="%d" />
```

B.2 MPI_COMBINER_CONTIGUOUS

```
<CONTIGUOUS id="%d" name="%s" count="%d" oldType="%d" />
```

B.3 MPI_COMBINER_VECTOR

```
<VECTOR id="%d" name="%s" count="%d" blocklength="%d" stride="%d" oldType="%d" />
```

B.4 MPI_COMBINER_HVECTOR, MPI_COMBINER_HVECTOR_INTEGER

```
<HVECTOR id="%d" name="%s" count="%d" blocklength="%d" stride="%d" oldType="%d" />
<HVECTOR_INTEGER id="%d" name="%s" count="%d" blocklength="%d" stride="%d" oldType="%d"
    />
```

B.5 MPI_COMBINER_RESIZED

```
<RESIZED id="%d" name="%s" lowerBound="%d" extend="%d" oldType="%d" />
```

B.6 MPI_COMBINER_INDEXED

```
<INDEXED id="%d" name="%s" count="%d" oldType="%d" >
  <BLOCK len="%d" index="%d />
  <BLOCK len="%d" index="%d />
  ...
</INDEXED>
```

B.7 MPI_COMBINER_HINDEXED

```
<HINDEXED id="%d" name="%s" count="%d" oldType="%d" >
  <BLOCK len="%d" index="%d />
  <BLOCK len="%d" index="%d />
  ...
</HINDEXED>
```

B.8 MPI_COMBINER_INDEXED_BLOCK

```
<INDEXED_BLOCK id="%d" name="%s" count="%d" oldType="%d" blockLength="%d">
  <Block displacement="%s" />
  <Block displacement="%s" />
  ...
</HINDEXED_BLOCK>
```

B.9 MPI_COMBINER_STRUCT

```
<STRUCT id="%d" name="%s" count="%d" >
  <Type id="%s" displacement="%s" blocklen="%s"/>
</STRUCT>
```

B.10 MPI_COMBINER_STRUCT_INTEGER

```
<STRUCT_INTEGER id="%d" name="%s" count="%d" >
  <Type id="%s" displacement="%s" blocklen="%s"/>
</STRUCT_INTEGER>
```

B.11 MPI_COMBINER_SUBARRAY

```
<SUBARRAY id="%d" name="%s" count="%d" order="%s">
  <Dimension size="%s" subsize="%s" start="%s" />
  <Dimension size="%s" subsize="%s" start="%s" />
  ...
</SUBARRAY>
```

The attribute `order` is either `MPI_ORDER_C` or `MPI_ORDER_FORTRAN`.

B.12 MPI_COMBINER_DARRAY

```
<DARRAY id="%d" name="%s" size="%d" order="%s" rank="%d" dims="%d">
  <Dimension gsize="%d" distrib="%s" darg="%d" psize="%d" />
  <Dimension gsize="%d" distrib="%s" darg="%d" psize="%d" />
  ...
</DARRAY>
```

The attribute `order` is either `MPI_ORDER_C` or `MPI_ORDER_FORTRAN`.

The attribute `distrib` has one of the values

```
MPI_DISTRIBUTE_BLOCK
MPI_DISTRIBUTE_CYCLIC
MPI_DISTRIBUTE_NONE
MPI_DISTRIBUTE_DFLT_DARG
```

References

[Lab] Argonne National Laboratory. mpi-io-test.c.