Leveraging Machine-readable MPI API Specification for Tool Development

Felix Tomski, Joachim Jenke (jenke@itc.rwth-aachen.de),
Simon Schwitanski

Motivation

- Each MPI release adds dozens of new functions
- Adding all functions to tools is tedious and error-prone
- Automate how support for new MPI functions is added
  - What is the semantic of all the arguments?
  - What is the length of arrays?

```
int MPI_Scatterv_init(const void *sendbuf, const int sendcounts[],
                      const MPI_Aint displs[], MPI_Datatype sendtype,
                      void *recvbuf, int recvcount, MPI_Datatype recvtype,
                      int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)
```

Common approach: Wrapper generator (e.g., wrap.py expanded)

```
int MPI_Scatterv_init(const void *sendbuf, const int sendcounts[],
                      const MPI_Aint displs[], MPI_Datatype sendtype,
                      void *recvbuf, int recvcount, MPI_Datatype recvtype,
                      int root, MPI_Comm comm, MPI_Info info, MPI_Request *request) {
   preType(sendtype);
   preType(recvtype);
   preComm(comm);
   int retval = PMPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf,
                                   recvcount, recvtype, root, comm, info, request);
   postRequest(request);
   return retval;
}
```


NHR4CES NHR for Computational Engineering Science

POP Performance Optimisation and Productivity
A Centre of Excellence in HPC

i12 High Performance Computing

RWTH AACHEN UNIVERSITY

GTI: xml-guided wrapper generation

```
int MPI_Scatterv_init(const void *sendbuf, const int sendcounts[],
                      const MPI_Aint displs[], MPI_Datatype sendtype,
                      void *recvbuf, int recvcount, MPI_Datatype recvtype,
                      int root, MPI_Comm comm, MPI_Info info, MPI_Request *request) {
   datatypeCheck::validForCommunication(sendtype); …
   forwardPreAnalysis(…);
   int retval = PMPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf,
                                   recvcount, recvtype, root, comm, info, request);
   requestTrack::Collective_init(…, request); …
   forwardPostAnalysis(…);
   return retval;
}
```

Limitations of wrapper generators

Pros:

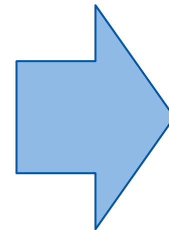
- Automated generation of wrappers based on mpi.h
- Automatically adapt MPI implementation issues (e.g., “interpretation” of const-ness)

Cons:

- Manually manage exceptions
- No information about sizes of arrays
- Which arguments constitute the *message data* (base, type, count)?

`apis.json` generated during building the MPI standard

```
"mpi_scatterv_init": {
    "attributes": { …
        "deprecated": false, … },
    "name": "MPI_Scatterv_init",
    …
    "parameters": [ …
        { …
            "asynchronous": true,
            "kind": "BUFFER",
            "name": "sendbuf", … },
    … ],
    "return_kind": "ERROR_CODE"
},
```



MPI_SCATTERV_INIT(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, info, request)

IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank (significant only at root)
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to sendbuf) from which to take the outgoing data to process i (significant only at root)
IN	sendtype	datatype of send buffer elements (handle, significant only at root)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	recvcount	number of elements in receive buffer (non-negative integer)
IN	recvtype	datatype of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

What can we learn from apis.json?

```
int MPI_Scatterv_init(const void *sendbuf, const int sendcounts[],
                      const MPI_Aint displs[], MPI_Datatype sendtype,
                      void *recvbuf, int recvcount, MPI_Datatype recvtype,
                      int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)
```

Function level:

- Deprecated
- Callback

What can we learn from `apis.json`?

```
int MPI_Scatterv_init(const void *sendbuf, const int sendcounts[],
                      const MPI_Aint displs[], MPI_Datatype sendtype,
                      void *recvbuf, int recvcount, MPI_Datatype recvtype,
                      int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)
```

Parameter level:

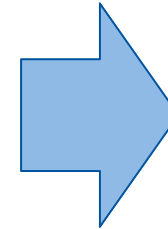
- `param_direction`: in/out
- `root_only`: parameter only valid on root
- `length`: “*”, other argument (e.g., `in-count`), MPI-constant (e.g., `MPI_MAX_PORT_NAME`)
- `asynchronous`: array is on return owned by MPI
- `kind`: `BUFFER`, `RANK`, `*_NNI`

What information do we miss?

Function level:

- Deprecated/deleted in version x.y?
- Semantics table (MPI 4.1, Appendix A.2)
  - collective, p2p, rma, i/o, management
  - all-to-all, one-to-all, all-to-one
  - (non-)blocking, (non-)local, persistent
  - stages: init/starting/completing/free

➢ Scripted insertion of attributes based on the table



```
"mpi_scatterv_init": {
    "attributes": {
        "c_expressible": true,
        ...
        "stages": "i",
        "cpl": "ic",
        "loc": "nl",
        "blk": "b",
        "op": "pop",
        "collective_c": "c",
        "collective_sq": "sq",
        "collective_sync": "w1"
```


NHR4CES NHR for Computational Engineering Science

Performance Optimisation and Productivity
A Centre of Excellence in HPC

i12 High Performance Computing

RWTH AACHEN UNIVERSITY

What information do we miss?

Parameter level:

- Identify *message data* parameters (e.g., `MPI_Pack` vs. `MPI_Reduce` vs. `MPI_Scatter`)
  - `MPI_Pack`: outbuf is marked as BUFFER, but is not a *message data* buffer
  - `MPI_Reduce`: *message data* (sendbuf, type, count) and (recvbuf, type, count)
  - `MPI_Scatter`: *message data* (sendbuf, sendtype, sendcount*group.size()) and (recvbuf, recvtype, recvcount)
- Handle creation vs. free vs. modify (e.g., `MPI_Type_commit`)
- Allowed special handles/values (e.g., `MPI_BOTTOM`, `MPI_IN_PLACE`, `MPI_PROC_NULL`)

What seems inconsistent / what does it mean?

- `execute_once = false` for all functions - even for `MPI_Init` / `MPI_Finalize`
- length = “*” mostly where length should be `group.size()`
- `kind` declaration with _NNI suffix seems to be inconsistent
  - All integer parameters should have a value range attribute

WIP: Verify consistent mapping of analyses to all MPI functions

```
int MPI_Scatterv_init(const void *sendbuf, const int sendcounts[],
                      const MPI_Aint displs[], MPI_Datatype sendtype,
                      void *recvbuf, int recvcount, MPI_Datatype recvtype,
                      int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)
```

- sendcounts is an array of non-negative integers

```
[rule.valid_array_count]
analysis = ['IntegerChecks:errorIfLessThanZeroArray']
condition = 'func.has_param(kind == "POLYXFER_NUM_ELEM_NNI" && length == "*")'
```

- sendtype/recvtype must be predefined or committed types

```
[rule.datatype_root_communication]
analysis = ['DatatypeChecks:errorIfNotValidForCommunication']
condition = 'func.has_param(kind == "DATATYPE" && !root_only) && func.has_other(kind == "BUFFER")'
```

Planned: Map existing analyses to new MPI functions

```
int MPI_Scatterv_init(const void *sendbuf, const int sendcounts[],
                      const MPI_Aint displs[], MPI_Datatype sendtype,
                      void *recvbuf, int recvcount, MPI_Datatype recvtype,
                      int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)
```

- Collective matching: arguments the whole communicator must be consistent:
  - Same root, info,
  - Consistent type matching
- Sendtype/recvtype must be predefined or committed types
- Root must be non-negative and smaller than comm.size()

Conclusions

- `apis.json` already provides some meta data that helps creating tools
- Identified useful extension for apis.json
- Sketched use cases in MUST

- For access to apis.json ask your MPI reps

Thanks! Questions?

NHR4CES NHR for Computational Engineering Science

Performance Optimisation and Productivity
A Centre of Excellence in HPC

i12 High Performance Computing

RWTH AACHEN UNIVERSITY