The XMPI API and trace file format

Nicholas J. Nevin

January 29, 1997

1 Overview

This document describes the interface between the portable and system independent parts of XMPI, and the format of an XMPI trace file.

2 API

The API currently consists of the following functions

2.1 `void xmpi_sys_cleanup(void)`

This function cleans up all system specific state upon exiting XMPI. It is called when XMPI exits.

2.2 `void *xmpi_sys_comm(struct _gps *proc, int cid)`

This function is invoked when in snapshot mode to obtain information on the communicator defined by process proc with context ID cid. If communicator information is found the function returns a pointer to allocated memory (XMPI will free it) containing the communicator trace otherwise it returns `0`.

2.3 `void *xmpi_sys_dtype(struct _gps *proc, int dtype)`

This function is invoked when in snapshot mode to obtain information on the datatype defined by process proc with label dtype. In the case of a basic datatype it returns `0`. In the case of a derived datatype for which a trace is found it returns a pointer to allocated memory (XMPI will free it) containing the datatype trace . In the case of a derived datatype for which no trace is found the function returns `-1`.

2.4 `char *xmpi_sys_errorstr(int err)`

Checks err for a system specific error code and if it finds one returns a pointer to a statically allocated system specific error message. The function returns `0` if err is not a system specific error code.

2.5 `char **xmpi_sys_hosts(char *bootfile)`

Given a bootfile returns a NULL terminated vector of node names.

2.6 `int xmpi_sys_init(void)`

Performs any system specific initialization required by the MPI implementation in order for XMPI to be able to launch MPI applications. It is invoked once only (before `xmpi_sys_run()` is called) upon the first occasion the user launches an application.

2.7 `int xmpi_sys_kill(struct _gps *app_procs, int app_nprocs)`

This function kills the application described by the GPS array `app_procs` of length `app_nprocs`.

2.8 `void xmpi_sys_logo(void *pix, int *width, int *height)`

Returns in pix a pointer to the pixmap of the vendor specific logo, and returns in width and the height the width and height of the pixmap. Failure to create the logo pixmap can be indicated by returning `0` in pix.

2.9 `int xmpi_sys_run(char *aschema, struct _gps **app_procs, int *app_nprocs)`

Starts the MPI application described by the application schema file aschema. It returns the number of processes in the application in app_nprocs and an array of the process GPS structures in app_procs.

This function is invoked by the run button of the application build dialog, the run button of the application schema browser dialog and the rerun button or menu option.

2.10 `int xmpi_sys_snapshot(struct _gps *app, int app_n, struct xmproc *procs)`

Gets state information for the running application described by the GPS array app of length `app_n` and fills it into the corresponding array procs of XMPI process descriptions. This function is invoked by XMPI when an application is launched and when the user selects the snapshot button or menu item.

2.11 `int xmpi_sys_trace(int fd, struct _gps *app_procs, int app_nprocs)`

Dumps the current traces for the running application described by the GPS array app of length `app_n`, into the open file with descriptor `fd`.

2.12 `char *xmpi_sys_version(void)`

Returns a pointer to a statically allocated vendor specific version identification string.

3 Trace file format

Most entries in the trace file correspond to C structures and they will be described in terms of these structures. All of the structures contain only LAM portable datatypes `char`, `int4` and `float8`. All values in the trace file are stored in LAM (big-endian) byte order.

The trace file format consists of a sequence of traces. Traces can be divided into three main categories.

1. world trace containing a magic number and a description of the processes making up the application.
2. object traces describing MPI objects such as datatypes and communicators.
3. run-time traces describing run time actions of the application, e.g. message passing.

All traces other than the world trace start with a source subtrace.

3.1 World trace

The world trace starts the trace file and consists of the following in the order,

1. magic number (`int4`).
2. size of `MPI_COMM_WORLD` (`int4`).
3. process descriptions (see Section 3.1.1) in order of rank in `MPI_COMM_WORLD`.

3.1.1 Process description

```
struct _gps {

    int4 gps_node;
```

Node that process ran on. This can be any number that uniquely identifies the host the process ran on but must correspond to the node identifier placed in source traces (see Section 3.2).

```
    int4 gps_pid;
```

This can be any number (such as a UNIX pid) that uniquely identifies the process within its host, and must correspond to the pid identifier placed in source traces.

```
    int4 gps_idx;
```

Index of process. Ignored by XMPI.

```
    int4 gps_grank;
```

Rank of process in `MPI_COMM_WORLD`.

```
};
```

3.2 Source subtrace

All traces other than the world trace start with a source subtrace which describes the type of the following trace and the process which made it.

```
struct trsrc {
    int4 trs_node; /* node of process that made trace */
    int4 trs_pid; /* pid of process that made trace */
```

The node and pid must correspond with those given in the world trace.

```
    int4 trs_listno; /* list number of the trace */
```

This indicates the type of the following trace. Possible values are given below.

```
    int4 trs_pad; /* for alignment */
};
```

The value for `trs_listno` must be one of

1. `TRCOMM` communicator object trace
2. `TRDTYPE` datatype object trace
3. `TRONOFF` run-time trace of class `TRONOFF`
4. `TRRUNTIME` run-time trace of class `TRRUNTIME`

3.3 Object trace

Currently there are two types of object trace. A communicator trace describes an MPI communicator and a *datatype* trace describes an MPI derived datatype. Remember that these are always preceded by a source subtrace.

3.3.1 Communicator trace

These follow a source trace with type TRCOMM.

```
struct trcid {

    int4 trc_cid; /* cid */
```

The cid (context ID) uniquely identifies a communicator for any given process. This must a nonnegative number. Values 0 through 2 are reserved for the pre-defined communicators MPI_COMM_WORLD, MPI_COMM_SELF and MPI_COMM_PARENT respectively.

```
    int4 trc_nlg; /* size of local group */
```

Size of the local group.

```
    int4 trc_nrg; /* size of remote group */
```

Size of the remote group.

```
    int4 trc_pad; /* for alignment */
};
```

Following this structure come process descriptors for the local group in order of rank in the group and then process descriptors for the remote group in order of rank in the group.

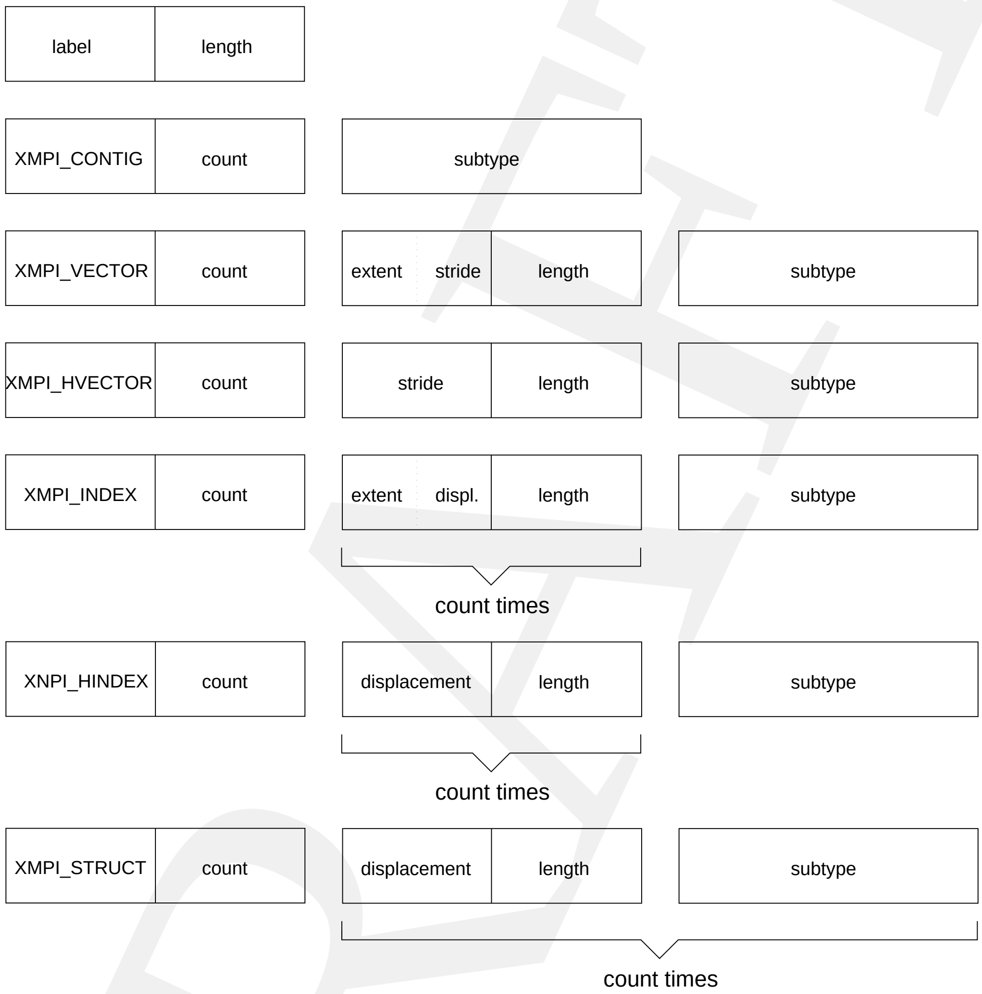
3.3.2 Datatype trace

Datatype traces are generated for committed derived types only. A datatype trace follows a source trace with type `TRDTYPE`. It is a flattened representation of the MPI datatype and consists of a sequence of elements of C type `struct trdtype`.

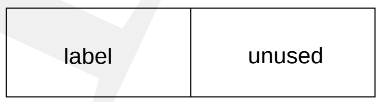
```
struct trdtype {

    int4 trd_param0;
    int4 trd_param1;
};
```

The meaning of the fields in the structure depends on its position in the trace. See the figure below. The first element in the sequence contains the datatype label and the length in bytes (including the first element) of the flattened representation. The following elements describe the derived using the format in the figure below. In the figure each box is an element (`struct trdtype`).



When a basic type occurs in the flattened representation it is represented by an element with the following form.



In this case label must be one of the basic datatype labels from Appendix A.

3.4 Run-time trace

Run-time traces are three-part. The first is a source subtrace. Following this comes a run-time header subtrace (type `struct trrthdr`) followed by a trace whose format depends on the type indicated in the run-time trace.

Run-time traces currently come in two main classes, TRONOFF and TRRUNTIME. The class is indicated in the type field of the source subtrace.

Run-time traces in class TRONOFF come directly after the world trace in the trace file and before the communicator traces. They come in three flavors described in Section 3.4.

Run-time traces in class TRRUNTIME come after all other traces and come in various flavors as described Section 3.4.

3.5 Run-time header

```
struct trrthdr {
```

```
    float8 trr_time; /* trace time (sec) */
```

Trace time in seconds. It indicates the time the event the trace is describing occurred. See the description of the various flavors of run-time trace for details.

```
    int4 trr_type; /* trace type */ The flavor of the following trace.

    int4 trr_pad; /* for alignment */;
```

3.6 Class TRONOFF

These come in three flavors, init traces `TRTINIT`, tracing on traces `TRTRON` and tracing off traces `TRTROFF`.

Init traces have the form below. The time in the prior run-time header is the time at which the process finished `MPI_Init()`. The XMPI time-line for the process starts here.

```
struct trinit {
```

```
    float8 tri_skew; /* clock skew (sec) */
```

The skew of the processes clock relative to that of process 0.

```
    char tri_name[32]; /* application name */
```

The name of the the process binary.

```
};
```

The tracing on/off traces describe the start and end of tracing epochs. The epochs are controlled by users via switches to mpirun and MPI extension functions which control the turning off/on of trace generation. For each on/off trace there should be a corresponding run-time trace of class `TRRUNTIME` with the same values. XMPI checks the correspondence between the on/off traces and the run-time ones to determine if run-time traces were dropped.

The time in the prior run-time header is the time at which the tracing mode was changed and the structure of the trace is,

```
struct tronoff {
```

```
    int4 tro_trnum; /* trace epoch number */
```

These must uniquely identify a trace epoch (period for which tracing is on).

```
    int4 tro_pad; /* for alignment */
};
```

3.6.1 Class TRRUNTIME

These come in the following flavors.

1. TRTSUBCHG substrate change. This is used to indicate that we have entered a top-level function that starts, tests or waits for non-blocking requests to complete. e.g. MPI_Startall(). These traces are of type struct trxchg.

2. TRTOUTPUT, TRTINPUT and TRTNOIO correspond to collective operations and point-to-point operations. Output traces are drawn with an outgoing arrow to the matching input trace and vice-versa. No IO traces represent collective operations and cases when no message transfer is done e.g. MPI_Probe(). All these traces are of type struct trmsg.

3. TRTRON tracing on and TRTROFF tracing off. These correspond to traces of the same format in the TRONOFF class. See Section 3.4.

```
struct trmsg {
```

```
int4 trm_topfunc; /* top-level function */
```

The top level MPI function which generated the trace or in the case of a non-blocking request started, tested for or waited for, the function which created the request. In the latter case the top level function which started it, tested it or waited on it, will be given in the field trm_wrapfunc.

```
int4 trm_wrapfunc; /* wrapper function */
```

See discussion of the previous field.

```
int4 trm_syst; /* trace time in system (usec) */
```

Time spent in system overhead (microseconds).

```
int4 trm_blkt; /* trace time blocked (usec) */
```

Time spent blocked (microseconds).

```
int4 trm_peer; /* peer rank */
```

The global rank of the peer process if any. This is the rank specified by the user and hence can be the wild-card MPI_ANY_SOURCE which is represented by the constant XMPI_ANY_SOURCE. In the wild-card case the rank of the actual peer synchronized with is given in the field trm_mrank.

```
int4 trm_tag; /* tag */
```

The tag used in the communication. This is the tag specified by the user and hence can be the wild-card MPI_ANY_TAG which is represented by the constant XMPI_ANY. In the wild-card case the tag actually synchronized with will be given in the field trm_mtag.

```
int4 trm_cid; /* context ID */
```

The context ID of the communicator used in the synchronization.

```
int4 trm_dtype; /* datatype label */
```

The label of the datatype given in the user call.

```
int4 trm_count; /* message count */
```

The count of datatype elements given in the user call.

```
int4 trm_mrank; /* src rank matched in case of wcard */

int4 trm_mtag; /* tag matched in case of wcard */

int4 trm_seqnum; /* sequence number */
```

The sequence number of a point to point communication. The sequence numbers in the traces of a matched send/receive pair must be the same.

```
};
```

Some of the above fields may not be applicable in all cases, e.g. the datatype (`trm_dtype`) field in the the case of `MPI_Barrier()`. In such cases the value of the field may be left undefined, i.e. there is no need to insert any value to indicate the field is not applicable.

```
struct trxchg {

    int4 trx_topfunc; /* top-level function */
```

The top-level function being entered or exited.

```
int4 trx_sub; /* new substrate */
```

The new substrate. One of system overhead `XMPI_SSYSTEM`, blocked `XMPI_SBLOCK` or running `XMPI_SRUN`

```
};
```

3.7 Order of traces in the trace file

Traces appear in the trace file in the following order.

1. world trace
2. run-time traces of class `TRONOFF` (order is immaterial)
3. communicator traces (order is immaterial)
4. datatype traces (order is immaterial)
5. run-time traces of class `TRRUNTIME`. All traces generated by a process must be in order of increasing time. No order is required between traces generated by different processes.

A Basic datatype labels

Trace label constant	MPI basic datatype
XMPI_BYTE	MPI_BYTE
XMPI_CHAR	MPI_CHAR
XMPI_UCHAR	MPI_UNSIGNED_CHAR XMPI_SHORT
MPI_SHORT	XMPI_USHORT MPI_UNISGNED_SHORT
XMPI_INT	MPI_INT XMPI_UINT
MPI_UNSIGNED_INT	XMPI_LOGICAL
MPI_LOGICAL	XMPI_LONG
MPI_LONG	XMPI_ULONG MPI_UNSIGNED_LONG
XMPI_FLOAT	MPI_FLOAT XMPI_DOUBLE
MPI_DOUBLE	XMPI_COMPLEX
MPI_COMPLEX	XMPI_UB MPI_UB
XMPI_LB	MPI_LB XMPI_PACKED
MPI_PACKED	XMPI_2INT MPI_2INT
XMPI_FLOAT_INT	MPI_FLOAT_INT
XMPI_DOUBLE_INT	MPI_DOUBLE_INT
XMPI_LONG_INT	MPI_LONG_INT
XMPI_SHORT_INT	MPI_SHORT_INT
XMPI_LONG_DOUBLE	MPI_LONG_DOUBLE
XMPI_LONG_DOUBLE_INT	MPI_LONG_DOUBLE_INT
XMPI_INTEGER	MPI_INTEGER
XMPI_CHARACTER	MPI_CHARACTER
XMPI_REAL	MPI_REAL
XMPI_DOUBLE_PRECISION	MPI_DOUBLE_PRECISION
XMPI_DOUBLE_COMPLEX	MPI_DOUBLE_COMPLEX
XMPI_2INTEGER	MPI_2INTEGER
XMPI_2REAL	MPI_2REAL
XMPI_2DOUBLE_PRECISION	MPI_2DOUBLE_PRECISION
XMPI_LONG_LONG_INT	MPI_LONG_LONG_INT
XMPI_INTEGER1	MPI_INTEGER1
XMPI_INTEGER2	MPI_INTEGER2
XMPI_INTEGER4	MPI_INTEGER4
XMPI_INTEGER8	MPI_INTEGER8
XMPI_REAL4	MPI_REAL4
XMPI_REAL8	MPI_REAL8
XMPI_REAL16	MPI_REAL16

Note: the trace label constants can be found in file `xmpi_sys.h`.