Open Trace Format Specification

Andreas Knüpfer, Holger Brunst, Ronny Brendel

Center for Information Services and High Performance Computing (ZIH)
Technische Universität Dresden, Germany

andreas.knuepfer@tu-dresden.de
holger.brunst@tu-dresden.de
ronny.brendel@tu-dresden.de

November 17, 2011

Contents

1 Introduction

The development of scalable tracing tools for high-performance computing (HPC) platforms with thousands of processors requires both a low-overhead trace measurement system to generate the trace data and efficient trace analysis tools to process the data. Of vital importance for tracing tools development is an open specification of the trace information that provides a target for trace generation and enables trace analysis and visualization tools to operate efficiently at large scale. The integration of facilities for trace generation with trace analysis and visualization tools is enabled by a well-defined trace format with open, public domain libraries for writing and reading the trace in that format. In addition, features of the trace format can directly support analysis tool capabilities to speedup up trace data access and processing. These two benefits combined addresses concerns for a format that can target future cross-platform tracing solutions for high-end ASC production systems.

The current document gives a detailed specification of the Open Trace Format (OTF), created specifically to support the development of scalable performance tracing tools for High Performance Computing. The specifications covers the trace storage and processing model, record types, and an API specification for reading and writing OTF event traces. The format addresses large applications written in an arbitrary combination of Fortran77, Fortran (90/95/etc.), C, and C++. The trace representation supports efficient scalable access and information processing by structural mechanisms for fast queries and features to increase trace processing flexibility.

The OTF specification forms the basis for a trace measurement and analysis toolset. In particular, the OTF will be introduced by its use in the Vampir Suite for trace analysis and visualization. Figure 1 shows the integration of OTF in this set of tools. EPILOG, VTF3 and OTF trace files can be input to Vampir/VampirServer, but OTF is required to enable its full efficiency and all performance feature .

The remainder of the document is organized in two main sections. Section 2 discusses the design and architecture of OTF. Section 3 presents the application programming interface for OTF, consisting of ten components for reading, writing, managing, and buffering OTF traces. The document concludes with a few OTF application examples in section 4.

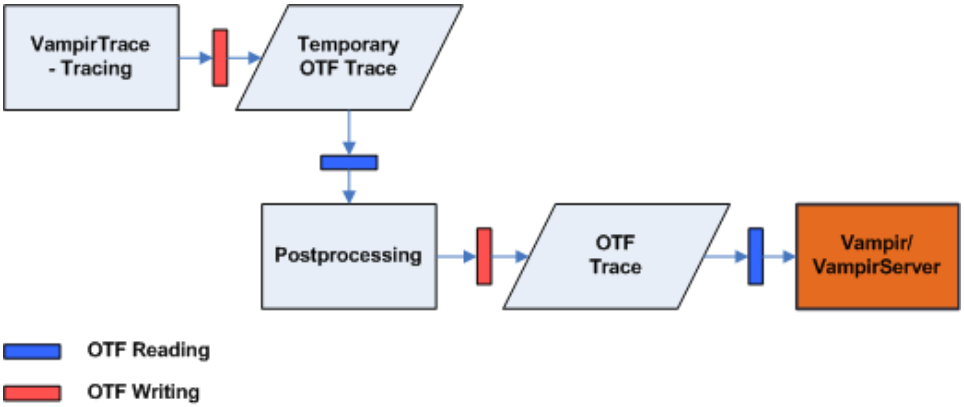

Figure 1: Integration of OTF in VampirTrace and Vampir/VNG.

2 OTF Design

The design of OTF is directed at three objectives: Openness, flexibility, and performance. The open format defines the record types and file structure so that OTF trace files can be both generated and read correctly. The flexibility objective in the OTF design comes from choices made with regards to trace data representation and storage, as well as parameters that OTF tools can control to organize and work with OTF traces. Performance is determine by how efficient and fast OTF trace querying and manipulation can be done. This section discusses the components of the OTF design in the context of these objectives.

2.1 ASCII Format

OTF uses a special ASCII data representation to encode its data items. ASCII encoding allows reduced storage sizes for small values since leading zeros can be omitted. All numbers and tokens are encoded in hexadecimal without the need of a special prefix which allows for a more efficient back and forth transformation compared to decimal numbers. Altogether, this enables a very powerful format with respect to storage size, human readability, and search capabilities on timed event records. Furthermore, it avoids platform dependent byte ordering issues.

2.2 Streams and Files

In order to support fast and selective access to large amounts of performance trace data, OTF is based on a *stream-model*, i. e. single separate units representing segments of the overall data. OTF streams may contain multiple independent processes whereas a process belongs to a single stream exclusively. The latter is

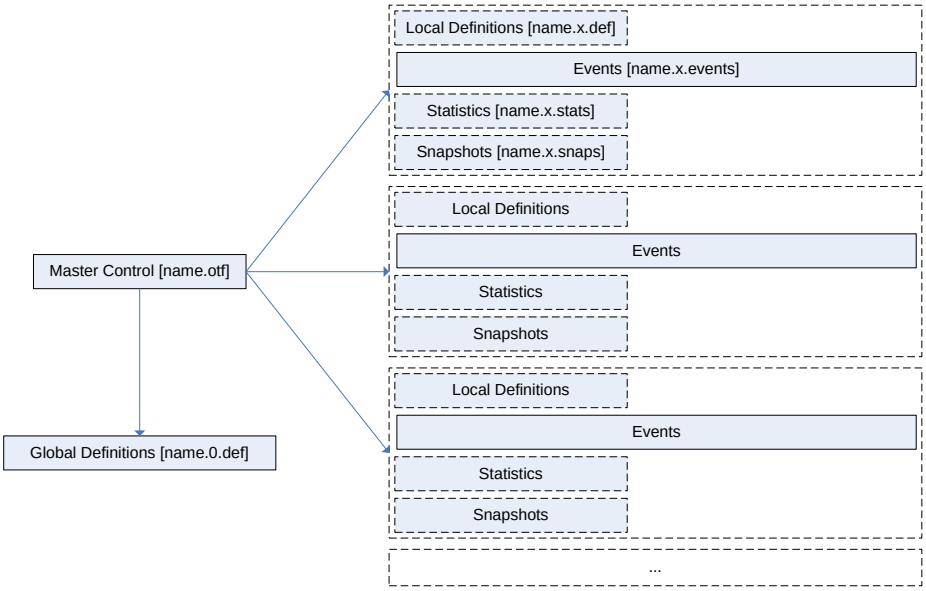

Figure 2: Files assembling an OTF trace.

needed for consistency reasons and cannot be relaxed without reducing the format's expressiveness.

Each stream is represented by multiple files which store definition records (see Section 2.6), performance events (see Section 2.7), status information (see Section 2.8), and event summaries (see Section 2.9) separately. A single global *master file* holds the necessary information for the process to stream mappings.

The names of trace file parts follow a strict naming convention. Each file name starts with an arbitrary common prefix which can be defined by the user. It is followed by a token identifier used for internal purposes (process mapping) and a suffix according to the file type. OTF files are *not* intended to be accessed directly but through the OTF library's API. This is a strict requirement to guarantee future compatibility.

The master file is always named '`<name>.otf`'. The global definition file is named '`<name>.0.def`'. Events and local definitions are placed in files '`<name>.x.events`' and '`<name>.x.defs`' where the latter files are optional. Snapshots and statistics are placed in files named '`<name>.x.snaps`' and '`<name>.x.stats`' which are optional, too.

When copying, moving or deleting traces it is important to take all according files into account. Deleting or modifying single files of a trace will render the whole trace invalid!

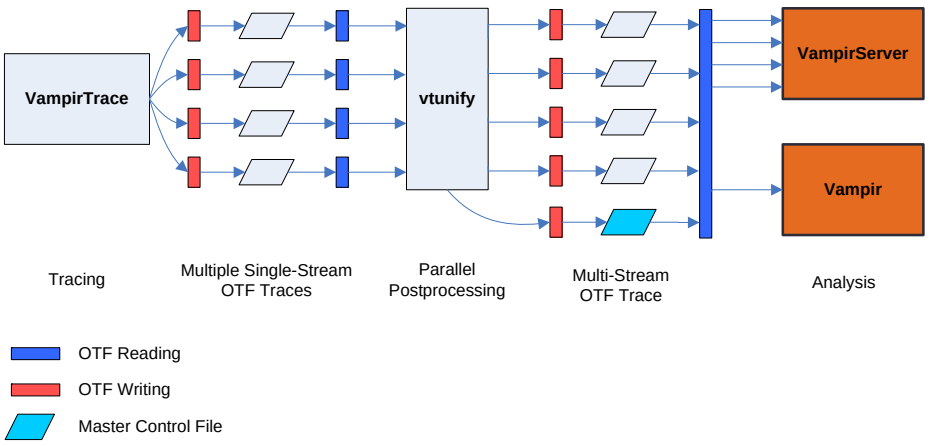

Figure 3: OTF Streams and File Architecture

The OTF library allows to transparently read and write trace data independently of the underlying partitioning of streams. Yet, if requested partitioning parameters can be queried and altered. This additional information can be used for various purposes. A good example is the tuning of the parallel loading process which obviously depends on the inherent process to stream mapping.

Merging with respect to the temporal order of the traced events is done on the fly by the library. Apart from that, the OTF library is able to access trace data consisting of n files by using a given number of m file descriptors with $1 \leq m < n$. This is an important feature to maintain scalability on very complex traces.

Figure 3 provides a high-level view of the OTF architecture showing streams and files used by the trace generation and analysis components.

2.3 State Machine

Within every OTF file, records are arranged as single lines of text whereas the detailed structure of every record type is defined separately.

However, some very frequently used properties are not included in the record lines but are handled by a state machine. This includes time stamp information, process/thread information and maybe others. For those there are special record types internally that set the respective property to a value. This value is then going to be valid for all following records until reassigned. With this approach, for example, time stamps need to be stored only once when multiple records refer to them. The read and write handlers for all record types are not affected by this and will remain as known by existing trace format libraries like VTF3.

2.4 Sorted Streams

Every OTF file needs to be sorted in temporal order. Unsorted files are regarded invalid and no sorting operation will be made available. There is no need to write unsorted traces in the first place. Furthermore, OTF is designed to handle huge traces and explicitly sorting streams won't scale well for very large streams.

2.5 Binary Search for Time Stamps

All OTF files can be searched very efficiently for time stamps in order to support fast selective access, since records are always sorted by time stamps such that binary search is applicable. The search mechanism is based on the fact that record boundaries can be quickly identified in the ASCII format.

2.6 Definition Record Types

As usual with trace formats, there is always a number of definition records. Such records carry some global properties like timer resolution, process count, etc. Furthermore, they define tokens to be referenced by event records, which allows for a more efficient encoding.

Definitions can be contained in single streams or globally as desired. There will always be a separate file for definition records. This makes it possible to define tokens late without disturbing the sorting order of events. All definitions are accompanied by a stream identifier which specifies the scope of the definition - stream specific or global.

With OTF, all identifiers are tokens, not indices. That means a set of N identifiers is not restricted to $\{0, 1, ..., N-1\}$. The value of zero (0) is always a reserved token used for special purposes. Apart from that, actual numerical values of identifiers are not important. Identifiers are only compared with respect to $=$ resp. $\neq$, where $<, >$ and hashing operations might be used for internal optimization of table look up and so on. Tokens are always of type **uint32_t**, i. e. unsigned integers of 32 bit size.

2.7 Event Record Types

Event records are the actual payload for traces. There is one event file per stream, which is sorted in temporal order. Events are for example entering/leaving of functions, sending/receiving of inter process messages or measurement of hardware counters.

2.8 Snapshot Record Types

Usually, traces are read linearly from the beginning. As OTF introduces the possibility to access arbitrary time stamps fast, some auxiliary information becomes necessary.

In order to start reading from an arbitrary time stamp, the current state of all participating processes needs to be known. If this information is not available from having read all preceding records as well, it needs to be stored explicitly. This is what *snapshot records* are designed for.

Snapshots provide the call stack (i.e. all active function calls), a list of pending messages, ongoing I/O activities, current OpenMP regions, etc... at a point in time (**not** including events at that very time stamp itself). Based on this information one can start reading event records at that very time stamp.

Snapshots are not generated by the OTF library itself but must explicitly be added. However, because they live in a separate file, it is possible to add/manipulate/replace/delete snapshots of a stream without affecting event data.

It is suggested to create snapshot information for time stamps on regular time distances. Different granularities for different phases of a trace might be convenient as well. Snapshots can be added right after trace file generation as an automated batch job or later on based on specific preferences.

For detailed insight on how writing and reading of snapshot records works have a look at the doxygen documentation.

2.9 Statistical Summary Record Types

A second class of auxiliary information is provided by *summary records* (sometimes called statistics records). They provide an overview over a whole interval of time, which might serve as a hint whether to read all events of that interval of time or not.

The data provided for this purpose is not explicit values for particular intervals of time but in a differential fashion like follows:

In order to provide summary information about a monotonous increasing property $p(t)$ for a time interval $[a, b]$, store the values $p(a)$ and $p(b)$. The result $p([a, b])$ can be computed as

$$p([a, b]) := p(b) - p(a).$$

With n points in time $t_0, ...t_{n-1}$, there are $n * (n - 1)$ possible interval results $p([t_i, t_j]), i \neq j$ of varying granularity that can be queried directly.

In comparison, with n explicit intervals potentially expensive accumulation of multiple (small) time intervals would be necessary to query for more than the n basic intervals.

Like snapshots, summaries can be added/modified/replaced/deleted without affecting events. Also, they need to be created explicitly and are not generated by the OTF library.

For detailed information on writing and reading summary records check out the doxygen documentation.

3 Application Programming Interface

The application programming interface (API) consists of ten components as shown in Figure 4. There are high level trace read and write interfaces called `Reader` and `Writer` that address whole traces. Both refer to the stream management interface `Master`. For dealing with single streams, read and write access is handled by `RStream` and `WStream`. Finally, the low level access to single `Files` is left to the `RBuffer` resp. `WBuffer`. The `FileManager` interacts with `Files` to make sure not to open more than a specified amount of file handles at a time.

All public components are described below with short examples. See Section 4 for typical usage for them.

For more detailed information on the API see the doxygen documentation. This chapter will only give a brief overview.

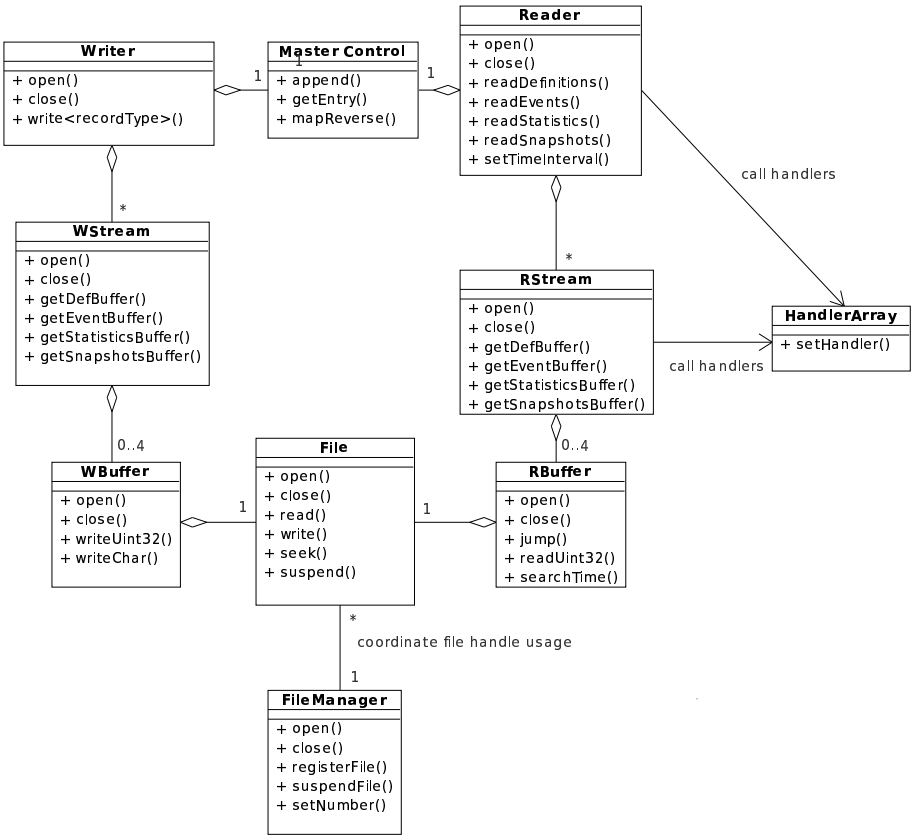

Figure 4: OTF API Classes Overview. This class diagram shows only few example methods.

3.1 Trace Write Interface

OTF has a low level and a high level trace writing interface. Latter is for simultaneous writing of multiple streams (files). The low level interface targets single streams only.

3.1.1 Global Write Interface

```
#include <otf.h>
#include <assert.h>

int main( int argc, char** argv ) {
```

Declare a file manager and a writer.

```
    OTF_FileManager* manager;
    OTF_Writer* writer;
```

Initialize the file manager. Open at most 100 OS files.

```
    manager= OTF_FileManager_open( 100 );
    assert( manager );
```

Initialize the writer. Open file "test", writing one stream.

```
    writer = OTF_Writer_open( "test", 1, manager );
    assert( writer );
```

Write some definition records. Have a look at the Doxygen documentation for more information on the functions used here.

```
    OTF_Writer_writeDefTimerResolution( writer, 0, 1000 );
    OTF_Writer_writeDefProcess( writer, 0, 1, "proc one",
            0 );
    OTF_Writer_writeDefFunctionGroup( writer, 0, 1000,
            "all functions" );
    OTF_Writer_writeDefFunction( writer, 0, 1, "main",
            1000, 0 );
```

Write an Enter and a Leave Record. time = 10000, 20000 process = 1 function = 1 Sourcecode location doesn't matter, so it's zero.

```
    OTF_Writer_writeEnter( writer, 10000, 1, 1, 0 );
    OTF_Writer_writeLeave( writer, 20000, 1, 1, 0 );
```

Clean up before exiting the program.

```
    OTF_Writer_close( writer );
    OTF_FileManager_close( manager );

    return 0;
}
```

Compile and link this using $ gcc -o test test.c ‘otfconfig --includes --libs‘.

3.1.2 Stream Write Interface

```
#include <otf.h>
#include <assert.h>

int main( int argc, char** argv ) {
```

Declare a file manager and a writer.

```
    OTF_FileManager* manager;
    OTF_WStream* wstream;
```

Initialize the file manager. Open at most 100 OS files.

```
    manager= OTF_FileManager_open( 100 );
    assert( manager );
```

Initialize the wstream object. Open file "test", writing the first stream.

```
    wstream = OTF_WStream_open( "test", 0, manager );
    assert( wstream );
```

Write some definition records.

```
    OTF_WStream_writeDefTimerResolution( wstream, 0,
          1000 );
    OTF_WStream_writeDefProcess( wstream, 0, 1,
          "proc one", 0 );
```

Clean up before exiting the program.

```
    OTF_WStream_close( wstream );
    OTF_FileManager_close( manager );

    return 0;
}
```

Compile and link this using $ gcc -o test test.c ‘otfconfig --includes --libs‘.

3.2 Trace Read Interface

Similar to the writing interface, OTF comes with a dual layer reading interface. The global interface provides transparent access to multiple streams while the local interface allows access to single streams only.

3.2.1 Global Read Interface

```
#include <otf.h>
#include <stdio.h>
#include <assert.h>
```

Define handlers/callbacks for the records you want to read.

```
int handleEnter (void *userData, uint64_t time, uint32_t
       function, uint32_t process, uint32_t source) {

    printf("we just entered function %u\n", function);

    return OTF_RETURN_OK;
}

int handleLeave (void *userData, uint64_t time, uint32_t
       function, uint32_t process, uint32_t source) {

    printf("byebye\n");

    return OTF_RETURN_OK;
}
```

```
int main( int argc, char** argv ) {
```

Declare a file manager, a reader, and a handler array.

```
    OTF_FileManager* manager;
    OTF_Reader* reader;
    OTF_HandlerArray* handlers;
```

Initialize the file manager. Do not open more than 100 files.

```
    manager= OTF_FileManager_open( 100 );
    assert( manager );
```

Initialize the handler array.

```
    handlers = OTF_HandlerArray_open();
    assert( handlers );
```

Initialize the reader.

```
    reader = OTF_Reader_open( "mytrace", manager );
    assert( reader );
```

Register your callback functions to the handler array.

```
    OTF_HandlerArray_setHandler( handlers,
           (OTF_FunctionPointer*) handleEnter,
           OTF_ENTER_RECORD );
    OTF_HandlerArray_setHandler( handlers,
           (OTF_FunctionPointer*) handleLeave,
           OTF_LEAVE_RECORD );
```

Do the actual reading.

```
    OTF_Reader_readEvents( reader, handlers );
```

Clean everything up before exiting the program.

```
    OTF_Reader_close( reader );
    OTF_HandlerArray_close( handlers );
    OTF_FileManager_close( manager );

    return 0;
}
```

Compile and link this using $ gcc -o test test.c ‘otfconfig --includes --libs‘.

3.2.2 Stream Read Interface

```
#include <otf.h>
#include <stdio.h>
#include <assert.h>
```

Define the Handler(s). We want to process the def process event and print out all appearing processes.

```
int handleDefProcess (void *userData, uint32_t stream,
       uint32_t process, const char *name, uint32_t
       parent) {

    printf( "process %u is named \"%s\"\n", process, name );

    return OTF_RETURN_OK;
}
```

```
int main( int argc, char** argv ) {
```

Declare a file manager, a reader, and a handler array

```
    OTF_RStream* rstream;
    OTF_FileManager* manager;
    OTF_HandlerArray* handlers;
```

Initialize the file manager. Do not open more than 100 files.

```
    manager= OTF_FileManager_open( 100 );
    assert( manager );
```

Initialize the handler array.

```
    handlers = OTF_HandlerArray_open();
    assert( handlers );
```

Initialize the rstream object. Open only Stream 0 of 'mytrace'

```
    rstream = OTF_RStream_open( "mytrace", 0, manager );
    assert( rstream );
```

Register your callback functions to the handler array.

```
    OTF_HandlerArray_setHandler( handlers,
           (OTF_FunctionPointer*) handleDefProcess,
           OTF_DEFPROCESS_RECORD );
```

Read all definition records from the global stream.

```
    OTF_RStream_readDefinitions( rstream, handlers );
```

Clean everything up before exiting the program.

```
    OTF_RStream_close( rstream );
    OTF_HandlerArray_close( handlers );
    OTF_FileManager_close( manager );

    return 0;
}
```

Compile and link this using $ gcc -o test test.c ‘otfconfig --includes --libs‘.

3.3 Stream Management Interface

This interface is dedicated to managing master control files (*.otf) A master control file contains a mapping from process streams to processes and vice versa.

The need for reading and writing master control files yourself arises, when you are using the low level stream reading/writing interface. The global reader/writer manages mastercontrols itself.

```
#include <otf.h>
#include <stdio.h>
#include <assert.h>

int main( int argc, char** argv ) {
```

Declare a couple of variables.

```
    uint32_t streamcount;
    uint32_t a;
    uint32_t i;
    OTF_MapEntry* entry;
    OTF_FileManager* manager;
    OTF_MasterControl* mc;
```

Initialize the file manager and the mastercontrol.

```
    manager= OTF_FileManager_open( 100 );
    assert( manager );

    mc = OTF_MasterControl_new( manager );
```

Add four processes (100,101,102,103) to two streams (1,2)

```
    OTF_MasterControl_append( mc, 1, 100 );
    OTF_MasterControl_append( mc, 1, 101 );
    OTF_MasterControl_append( mc, 2, 102 );
    OTF_MasterControl_append( mc, 2, 103 );
```

Write everything to the file "mytrace.otf" and close the master control.

```
    OTF_MasterControl_write( mc, "mytrace" );
    OTF_MasterControl_close( mc );
```

Now initialize the master control structure and read the newly written master control file.

```
    mc = OTF_MasterControl_new( manager );
    OTF_MasterControl_read( mc, "mytrace" );
```

Visit all stream-process pairs and print them out

```
    streamcount = OTF_MasterControl_getCount( mc );

    for( i = 0; i < streamcount; ++i ) {
        entry = OTF_MasterControl_getEntryByIndex( mc,
              i );
        for( a = 0; a < entry->n; ++a ) {
            printf( " stream %u contains process %u\n",
```

```
                    entry->argument, entry->values[a] );
            }
        }
```

Clean everything up before exiting the program

```
    OTF_MasterControl_close( mc );


    return 0;
}
```

Compile and link this using $ gcc -o test test.c ‘otfconfig --includes --libs‘.

The program will show this output:

```
*  stream 1 contains process 100
*  stream 1 contains process 101
*  stream 2 contains process 102
*  stream 2 contains process 103
```

4 Application Examples

Some typical application examples shall emphasize the usage of the different interfaces.

4.1 Trace Library

Within a trace library, the OTF library will be used in order to flush in-memory event buffers to trace file once in a while. Usually this is performed by every process/thread independently. Communication inside the trace library should be avoided if possible.

Under those conditions, the local write interface is most convenient. Every process/thread will open an exclusive stream **OTF_WStream** with an unique identifier. Thus, all N processes can write their events to N different stream independently.

Definition records can be added at any time: in the beginning, during tracing or in the end. If there are local (process/thread specific) definitions, they go the the respective stream. Global definitions can be written to the special stream with identifier 0, which is reserved for such purposes.

Finally, one process (the master) has to write the master control file, which states the mapping of processes/threads to streams. In this case, it will be a simple $1:1$ mapping. So, there is a valid trace without merging or reprocessing. The event data need not to be touched again.

However, reprocessing might be necessary in order to synchronize timers, translate different local token sets to a single consistent global token set or other post-processing steps. Furthermore, snapshot and summary information are to be added to this trace later. This might be combined with explicit merging like explained in the next scenario.

4.2 Trace Merging

Merging is not strictly necessary if traces are generated like outlined above. However, one might want to create a different granularity with the mapping of processes/thread to streams. For example, one wants to have fixed number of $M < N$ streams such that efficient parallel input is possible on a parallel analysis environment with M nodes. Or a fixed number of K processes/threads might be desirable.

The actual merging operation is supported by the OTF library itself already. Everything that is left to do is reading a trace with the given granularity and writing it again with another. Therefore, reading should be performed via the global read interface. Otherwise, merging is not handled by OTF.

The copy handlers provided by the read interface can be used directly. It requires just an **OTF_Writer** object of the global write interface. It has to be initialized with the number of output streams. Alternatively, the mapping can also be defined explicitly.

If, besides merging, additional translation or filtering is to be performed, customized handlers that receive the records, manipulate them and pass them on to a **OTF_Writer** object, should be used.

It is possible to do merging in parallel if the input streams are distributed in a disjoint way.

4.3 Parallel Reading and Searching

Reading a trace for analysis provides a number of new possibilities with OTF. Of course, classical linear reading from beginning to end is supported.

First, every analysis application will read definitions - global as well as all local ones. This is supposed to be done by every process of an parallel/distributed analysis application. All further reading can take advantage of efficient parallel input. Every analysis process can select only a subset of the trace processes to read. For best efficiency, this mapping should be aligned to the existing processes to streams mapping such that every reader process accesses a minimum number of different streams.

Second, it might refer to summary records (provided the trace contains some) in order to find interesting spots for selective access.

Third, the application might decide to read a selected time interval $\mathbb{I} = t_0, t_1$ only. Then it has to search for the latest snapshot time stamp $t^* < \mathbb{I}$. This search is performed by OTF via binary search which is very fast and efficient.

Processing that snapshot enables the analysis application to start reading from time t^* in the event time stamp. This time stamp inside the event stream is again detected by OTF via binary search. Now, event by event is delivered by the callback handlers as usual. Multiple select and search operations can be performed subsequently or in parallel.

5 Additional Python Interface

In order to use the OTF library within Python, compile the OTF sources as follows:

```
./configure --enable-python-bindings --enable-shared
```

Set an environment variable `$PYTHONPATH` that contains the path to the OTF module:

```
PYTHONPATH={install_dir}/lib/python{ver}/site-packages
export PYTHONPATH
```

Now, the Python support is enabled and you can simply use the OTF library by including the OTF module in your Python script:

```
from otf import *
```

To get an understanding of how to handle OTF in Python, you can look at some examples placed here:

```
./tests/otf_python/
```

6 Conclusion

This project was partially funded by the University of California (UC), Lawrence Livermore National Laboratory (LLNL), subcontract #B548849. OTF has been created in cooperation with LLNL, UC and the University of Oregon . We would like to thank Allen D. Malony and Sameer S. Shende for their contribution.