

# Towards Parallel Performance Analysis Tools for the OpenSHMEM Standard

Sebastian Oeste<sup>1</sup>, Andreas Knüpfer<sup>1</sup>, and Thomas Ilsche<sup>1</sup>

Technische Universität Dresden, Center for Information Services and HPC (ZIH)

**Abstract.** This paper discusses theoretic and practical aspects when extending performance analysis tools to support the OpenSHMEM standard for parallel programming. The theoretical part covers the mapping of OpenSHMEM's communication primitives to a generic event record scheme that is compatible with a range of PGAS libraries. The visualization of the recorded events is included as well. The practical parts demonstrate an experimental extension for Cray-SHMEM in Vampir-Trace and Vampir and first results with a parallel example application. Since Cray-SHMEM is similar to OpenSHMEM in many respects, this serves as a realistic preview. Finally, an outlook on a native support for OpenSHMEM is given together with some recommendations for future revisions of the OpenSHMEM standard from the perspective of performance tools.

**Keywords:** OpenSHMEM, Performance Analysis, Tracing, Tools Infrastructure

## 1 Introduction

In the field of High Performance Computing (HPC), MPI was and is the dominating parallelization standard. It provides a huge range of point-to-point and collective communication operations and is the de-facto standard in for highly parallel programming beyond multi-threading in High Performance Computing (HPC). Another model for parallel programming is gaining more and more attention lately, the Partitioned Global Address Space (PGAS) paradigm. It promotes the idea of shared memory parallelization even for large scale distributed memory parallel machines. For this purpose it employs Remote Direct Memory Access (RDMA) operations, also called one-sided communication. A number of efforts in the HPC community currently follow this paradigm. Among them may be promising candidates for a true rival for MPI. One of the main prospects of PGAS is the reduced complexity compared to hybrid MPI plus OpenMP parallelism. An indication that this may be true is the addition of one-sided operations to the MPI standard.

On the one hand, the set of PGAS approaches contains language extensions like Unified Parallel C (UPC) or Co-Array Fortran (CAF) which need their own compilers. On the other hand, it contains PGAS libraries that have the advantage that well-known languages like C/C++ and Fortran gain PGAS functionalities

and no extra compilers are needed. SHMEM is one flavor of such PGAS libraries. In the history of SHMEM there are several implementations like Cray-SHMEM, SGI-SHMEM or Quadrics SHMEM with quite different APIs. With OpenSHMEM there is a new effort to create an open standard to make all vendor versions of SHMEM API-compatible.

This paper presents a concept for a generic infrastructure to record event traces from OpenSHMEM parallel application programs. Its goal is an integration in the Score-P monitoring system which produces event traces in the Open Trace Format 2 (OTF2) that are to be visualized and analyzed with the Vampir tool. In section 2 we describe the common techniques for the performance monitoring of parallel applications. Section 3 presents a concept for a generic tracing infrastructure for PGAS approaches in general and OpenSHMEM in particular. Section 4 gives an overview of an existing prototypical implementation for trace recording of SHMEM events using Cray-SHMEM and the VampirTrace package. The paper ends with an outlook on a native OpenSHMEM event tracing solution for the parallel performance analysis.

## 2 Parallel Performance Analysis Tools

Tools for parallel performance analysis are an important part of the HPC software ecosystem. Since the parallel execution performance and the scalability are most significant properties in HPC, developers need to pay attention to performance analysis and tuning. And they need support by appropriate tools.

Just like debugging is very difficult without dedicated debugging tools, the detailed investigation of the run-time behavior apart from the wall clock duration is almost impossible. And both kinds of task get notably more complicated in the parallel case.

### 2.1 Instrumentation

To monitor HPC applications it is necessary to be able to measure the run-time behavior inside target programs exactly. To reach this we need concrete run-time information from the application during execution. A way to get run-time information of a program during execution is instrumentation. Instrumentation refers to a modification of the program code to measure certain events of interest. The measurement code is inserted before and after an event appears and triggers routines in the measurement environment to gather the information. Instrumentation can take place at different times of the program transformation process. There are several techniques to instrument a program.

- **Static instrumentation** The code will be inserted before execution.
- **Dynamic instrumentation** The code will be inserted during execution.
- **Manual instrumentation** The code is inserted manually this affords the most flexibility. The user can place measurement calls exactly on those parts of the program provides the most interest.

- **Automatic instrumentation** The code will be inserted automatically for example through certain linker options, or compiler functionalities. Which invokes corresponding functions with information like file name, line number, function name. Automatic instrumentation typically inserts enter and leave events.
- **Binary instrumentation** The instrumentation appears on the binary, executable file. Toolkits like the DyninstAPI are able to insert measurement code after the actual program is compiled [12].

The different ways to instrument an application makes instrumentation in general a flexible solution. We are currently developing an instrumentation tool for C and C++ sources which creates a wrapper library by parsing a header file and re-defining all symbols with performance annotations. This tool uses mechanisms for dynamic and static linking libraries and even supports profiling interfaces based on weak symbols.

## 2.2 Sampling

Another technique to record information is sampling. Sampling works without any modification of the programs source code or binary executable. During the execution of the program periodic interrupts occur to collect information about the state of the program. Popular information for sampling are values of performance counters or the state of the call stack. The granularity depends on the sampling frequency. An advantage of sampling compared with instrumentation is the ability to make forecasts about the overhead behavior. The overhead of program analysis using sampling grows linear with the sampling frequency.

## 2.3 Profiling

One way to present the acquired information from instrumentation or sampling is to create a so called profile. A profile in general is an summary of performance metrics e.g. the time difference between enter and leave events or the number of calls of a function. A profile can be created from information acquired by sampling or by instrumentation. Common kinds of profiles are the *flat profile* which is the simplest form of a profile. Because the aggregated information just depends on the regions or instrumented functions. Significantly for a flat profile is that it provides no caller context. Another form of a profile is a *call-path profile* which is a representation of the execution path. A call-path profile gives information about possible routes through a program - each route through the program is a own record.

## 2.4 Event Tracing

Parallel event tracing tools try to capture the run-time behavior of a target application by recording events of interest. Usually, event tracing relies on instrumentation to be notified about such events, but there are also sampling-based approaches. For every event a so called *event record* is stored. All records

are stored separately in every processing element (PE) in temporal order. Every record contains at least the type (what), the time (when) and the location (where) of an event and might carry further type-specific data. Usually, the stream of event records is stored in a local memory buffer and is stored to the file system only at the end of the measurement. See [13] for more background.

The events of interest that are used for parallel event tracing of High Performance Computing (HPC) applications can be grouped into three groups: Sequential events, parallel events, and scalar values.

**Sequential execution events** This includes all activities in all sequential phases of the parallel PEs. They have no direct effects on other PEs and are the same for all parallel programs no matter if they use MPI, Pthreads, or any other parallelization model. The monitoring system merely needs a way to find out the location (where) of the events. The granularity of such events may vary. Typical examples are the call to (enter event) or return from (leave event) for subroutine calls or the begin and end of loop bodies.

**Events for parallel communication and synchronization** Communication and synchronization event records cover the remaining parts of parallel programs. They represent all activities that directly affect two or more PEs. They are represented as local event records on some or all of the involved PEs. In particular, they allow to identify the communication peer(s) explicitly or implicitly.

This group of events is closely modeled according to the perspective of the programmer, i.e. the events represent the basic building blocks for communication and synchronization that the parallelization model provides. For parallelization libraries such as OpenSHMEM, this is close to their APIs.

There are a number of examples from established HPC parallelization models. First, there are point-to-point communication calls and collective communication calls from MPI [11], including both, the blocking and the non-blocking modes. Second, there are parallel regions and synchronization points from OpenMP which are not implemented as API calls but as code pragmas. Also there are data transfers calls between hosts and devices as well as synchronization points for GPGPU computing models such as CUDA and OpenCL. And there are PGAS-style communication and synchronization types which are applicable to OpenSHMEM and several other PGAS-libraries or PGAS language extensions, see also [9].

Usually, all communication and synchronization event records are surrounded by events for the API call to reference which exact API call was used and to capture the duration of the call (the time between the enter and the leave events).

**Hardware counters, system metrics, and user-defined metrics** In addition to the previous two groups, there are event records for *counters* or *metrics*. Usually, they provide summaries of events of interest which are too fine-grained to be recorded individually. The prime examples are CPU hardware performance

counters counting floating point instructions, memory accesses, cache misses, TLB misses, and many more. Sampling such counters at the enter and leave points of subroutine calls provides interesting data about the floating point instructions, memory accesses, etc. that happened inside the call. Besides hardware performance counters, also system counters like memory consumption or temperature can be captured in this way, even though they are not strictly counting discrete events. Also external metrics like the power consumption or the throughput rates of storage subsystems can be captured like this. Last but not least, so called *user defined metrics* can be provided by the application code itself to indicate relevant properties of its inner workings. Examples could be the number of outstanding requests at a point in time, partition sizes of problem decompositions, or the residual value of an iterative solver.

## 2.5 Existing tools and related work

There are a couple of performance analysis tools which support the techniques described above with a strong focus on High Performance Computing (HPC). The Tuning an Analysis Utilities (TAU) specialize in profiling with some tracing functionalities [18]. Scalasca focuses on automatic detection of well-known performance problems in parallel programs based on an event trace replay mechanism [5]. Vampir provides interactive visualization and exploration of parallel event traces [13]. All of the previously mentioned tools work together with the Score-P monitoring infrastructure for code instrumentation, profile collection and event trace recording [10]. There is also Vampir's previous default monitoring system called VampirTrace with a similar feature set. VampirTrace comes as a regular component of the Open MPI package and is therefore available on a large number of HPC machines worldwide [13]. HPCToolkit is one established example that relies on periodic sampling including call stack unwinding to capture the dynamic run-time behavior of parallel target applications [1].

With respect to performance tools for the rather young OpenSHMEM standard, there are very few pieces of related work. A prototypical extension to VampirTrace has been created by S. Jana and J. Schuchart at ORNL which capture traces of OpenSHMEM API calls but not data transfers. Using their extension, they facilitated an analysis of the energy consumption of certain OpenSHMEM library calls using the counter plug-in infrastructure of VampirTrace [7]. As far as we know this was not published yet. The extension of VampirTrace for the recording of API calls as well as data transfers for Cray-SHMEM, which was the basis for Section 4, was published in [17].

## 3 Concept of a tracing infrastructure for OpenSHMEM

The effort to provide recording of sequential events and counters or metrics for OpenSHMEM programs in existing monitoring systems is minimal. Yet, the PGAS-style communication scheme requires theoretical and practical changes in the recording and representation of events. Furthermore, the monitoring infrastructure needs to implement its internal communication with OpenSHMEM.

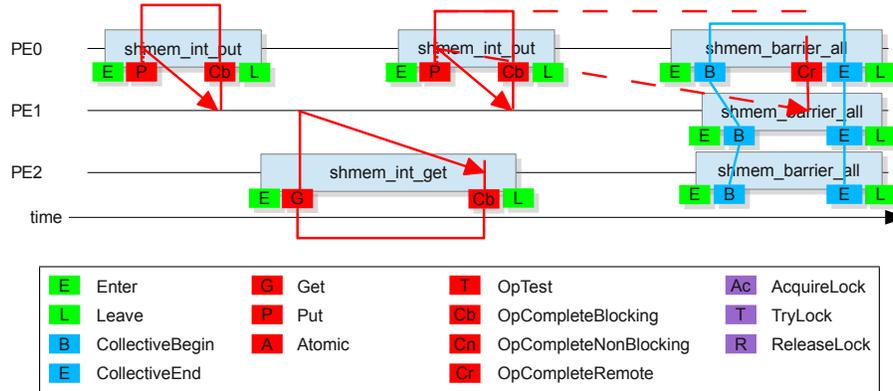


Fig. 1. Timeline visualization of put and get operations and their points of completion.

### 3.1 Modeling and recording PGAS activities

While MPI is clearly the dominating message passing model in the HPC landscape, there are several promising players in the PGAS realm. There are language extensions like Co-Array Fortran [16] and UPC [3] on the one hand and libraries like OpenSHMEM [4], GASPI [2], ARMCI [14], GlobalArrays [15] but also MPI 3.0 with the on-sided operations [11] on the other hand. All of them share the concept of the Partitioned Global Address Space with 'put' and 'get' operations (sometimes called 'write' and 'read', though) as the key communication operations. Therefore, there should be a single model and a single set of event records to represent all PGAS parallelization libraries. This allows analysis tools to be generic and usable for all PGAS libraries<sup>1</sup>.

Still, there are semantic differences between all the flavors of 'put' and 'get' operations and there are various kinds of synchronization mechanisms. Furthermore, there are some more concepts present in some of the PGAS libraries but not in all of them. In [9] a combined model is presented covering all of them. In the remainder of this paper, only the OpenSHMEM operations are discussed.

**Put and get operations** OpenSHMEM defines a variety of put and get operations. There are calls for individual numbers, for arrays or blocks, and strided ones for regular subarrays. All of them are blocking, i.e. the API calls only returns after the local completion of the operation. For get operations, local completion also ensures remote completion, that means the operation is not affected by following activities on the remote (passive) side. For put operations, only the local completion is given, i.e. following changes at the source address won't influence

<sup>1</sup> The model is also applicable to PGAS language extensions even though their 'put' and 'get' calls are hidden from the programmer. The compilers will generate them from the language constructs like loops. However, the remainder of this paper focuses on the PGAS libraries in general and OpenSHMEM in particular.

the locally completed operation. Remote completion, which means that the sent data is visible at the remote (passive) side, is not ensured and may happen later.

Put and get operations are recorded on the active PE only. Four event records are used for this, see also Fig.1. As first an last, an enter record (E) and a leave record (L) are written which denote the name of the called API function. The time between them is the duration of the API call. The actual data transfer is recorded as a RMA put event or an RMA get event, see Fig.2. They include the target PE, the transfer size in bytes, and a matching number. They also include a reference to a memory window, which is only relevant for other PGAS flavors which use multiple “memory windows” or “communicators”. In OpenSHMEM there is always a single “symmetric heap” and the specification of the target PE always uses global IDs.

The completion of put or get operations is marked with completion event records on the same (active) PE. In all cases, the local completion has to be marked with an ‘RmaOpCompleteBlocking’ record with the same matching number. Since all OpenSHMEM ‘put’ or ‘get’ calls are blocking until the local completion, this completion event is put at the end of the associated API call, just before the leave event, see Fig.1 and Fig.2. For ‘put’ operations where there is a separate remote completion which may or may not be detectable. From the OpenSHMEM API level it is visible for example when there is a following call to ‘shmem\_barrier\_all’, see Fig.1(top right). In this case, an optional ‘RmaOpCompleteRemote’ record can be written. It is connected to the originating put event with the same matching number. However, an OpenSHMEM program may proceed without ever notifying the active PE (that issued the put operation) about the remote completion. In such cases, the remote completion record is left out, see Fig.1(top left).

**Visualization of put and get operations** The graphical visualization of put and get operations relies on the event records over time on the active PE. Besides depicting the API call, the data transfer will be shown with an arrow from the source PE to the destination PE. The start time of the arrow is the time of the

<b>OTF2_Rma(Put Get)</b>		
OTF2_LocationRef	location	local PE
OTF2_TimeStamp	time	time stamp
OTF2_RmaWinRef	window	memory window
uint32_t	target	target PE in context of window
uint64_t	size	number of bytes transferred
uint64_t	matching	matching number

<b>OTF2_RmaOp(Test (Complete(Blocking NonBlocking Remote))</b>		
OTF2_LocationRef	location	process or thread of execution
OTF2_TimeStamp	time	time stamp
OTF2_RmaWinRef	window	memory window
uint64_t	matching	matching number

**Fig. 2.** OTF2 record definitions for put, get, and completion events.

respective put or get event and that is equal to the start time of the API call. The end time for the arrows should be the time of completion. This is trivial for get operations where the local completion is the same as the remote completion. Thus, it is visualized as in Fig.1(bottom) giving a realistic impression of the duration and the speed of the data transfer.

It is not well defined for put operations, because local completion is not equal to remote completion and because remote completion might be invisible for the monitoring system. Even if the remote completion is visible, it is most probably visible only at a point in time later than the actual arrival of the data transfer. At least it is generally recommended by all PGAS programming models to issue individual remote memory accesses as early as possible and to use barriers, fences, or other memory synchronization operations as late as possible. Since neither of the local or remote completion points give a good indication of the actual duration of a put data transfer, the local completion time is used because it is always available. It follows that the shown arrows for put operations need to be read differently. They are an indication when the put operation started and between which PEs it happens. Yet it does not indicate the transfer time.

**Atomic RMA operations** OpenSHMEM supports a number of atomic RMA operations that read and write remote variables in an atomicity manner, i.e. with the guarantee that no other local or remote memory access can interfere in between. A separate event record type is defined for atomic operations, see Fig.3. It is to be used like the put and get record types. In addition to those, it stores the type of atomic operations and it has separate fields to store the data volume sent and received. They should contain the number of bytes that two separate put and get operations would carry if they would try to mimic the effect of the atomic operation (without the atomicity).

<b>OTF2_RmaAtomicType</b>		
OTF2_RMA_ATOMIC_TYPE_SWAP		swap
OTF2_RMA_ATOMIC_TYPE_COMPARE_AND_SWAP		compare and swap
OTF2_RMA_ATOMIC_TYPE_FETCH_AND_ADD		fetch and add
OTF2_RMA_ATOMIC_TYPE_FETCH_AND_INCREMENT		fetch and increment
OTF2_RMA_ATOMIC_TYPE_ADD		remote add
OTF2_RMA_ATOMIC_TYPE_INCREMENT		remote increment
...		

<b>OTF2_RmaAtomic</b>		
OTF2_LocationRef	location	process or thread of execution
OTF2_TimeStamp	time	time stamp
OTF2_RmaWinRef	window	window
uint32.t	target	rank of target in context of window
OTF2_RmaAtomicType	type	type of atomic operation
uint64.t	size_sent	number of bytes transferred to target
uint64.t	size_received	number of bytes transferred from target
uint64.t	matching	matching number

**Fig. 3.** OTF2 record definitions for atomic events including the types of atomic operations that are relevant for OpenSHMEM.

Like put and get records, atomic RMA records should be followed by a local complete record and an optional global complete record. For the visualization a single arrow like that of a put operation should be used. If the 'size\_received' field is larger than 0 then a second arrow head pointing backward should be added.

**Collective operations** Collective operations in OpenSHMEM are operations performed simultaneously by a subset of PEs. They are represented by a pair of event records 'OTF2\_RmaCollectiveBegin' and 'OTF2\_RmaCollectiveEnd' as shown in Fig.4. The former merely denotes the begin of the operation, the latter contains all information about it. This pair of records is to be written by every participating PE.

The field 'sync\_level' is always set to 'OTF2\_RMA\_SYNC\_LEVEL\_ALL' for OpenSHMEM which means that memory and the execution is synchronized by OpenSHMEM collectives. Since OpenSHMEM collectives have no 'root' PE, the 'root' field is always set to a special value 'NONE'. The fields 'size\_sent' and 'size\_received' contain the number of bytes sent and received by the current PE if the collective operation would be mimicked by the minimal number of put and get operations. In the OpenSHMEM API these subsets of PE's in collective operations are known as *active set*. The participating PE's are managed in a group which refers to a memory window.

In the visualization all matching collective operations should be connected like shown in Fig.1(right) for the 'shmem\_barrier\_all' operation.

<b>OTF2_RmaCollectiveBegin</b>		
OTF2.LocationRef	location	process or thread of execution
OTF2.TimeStamp	time	time stamp

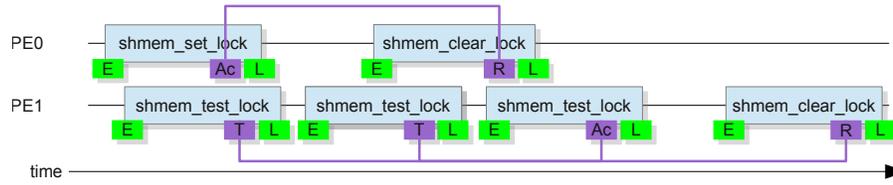
  

<b>OTF2_RmaCollectiveEnd</b>		
OTF2.LocationRef	location	process or thread of execution
OTF2.TimeStamp	time	time stamp
OTF2_RmaSyncLevel	sync_level	synchronization level
OTF2_RmaWinRef	window	memory window
uint32_t	root	root process/rank if there is one
uint64_t	size_sent	number of bytes sent
uint64_t	size_received	number of bytes received

**Fig. 4.** OTF2 record definitions for marking collective RMA operations.

**Locks** For the mutex lock concept, a separate set of events are introduced. Besides the API calls, they keep track of the lock type (exclusive lock or write lock vs. shared lock or read lock) and the lock instance. In the visualization, all operations working on the same lock on the same PE instance are connected until the lock is cleared. When several PEs compete for the same lock, then no connections are drawn between the PEs but when they are displayed side by side

X



**Fig. 5.** Timeline visualization of two PEs performing competing lock operations.

OTF2_LockType		
OTF2_LOCK_TYPE_EXCLUSIVE		only one lock allowed at the same time, e.g., write-lock, mutex, MPI exclusive lock
OTF2_LOCK_TYPE_SHARED		multiple shared locks allowed at the same time, e.g., read-lock, MPI shared lock

OTF2_(Request Try)Lock		
OTF2_LocationRef	location	process or thread of execution
OTF2_TimeStamp	time	time stamp
OTF2_RmaWinRef	window	memory window
uint32.t	target	rank of target in context of window
OTF2_RmaLockType	lock_type	Type of lock (shared vs. exclusive)
uint64.t	lock_id	lock id in context of window

OTF2_ReleaseLock		
OTF2_LocationRef	location	process or thread of execution
OTF2_TimeStamp	time	time stamp
OTF2_RmaWinRef	window	
uint32.t	target	rank of target in context of window
uint64.t	lock_id	lock id in context of window

**Fig. 6.** OTF2 record definitions for lock operations.

with aligned time axes, it becomes apparent, that only one can hold the lock at any time. See Fig.5 for an example.

The record definitions are shown in Fig.6, see also [9]. The reference to a memory window and a reference to a target PE are not relevant for OpenSHMEM but only important for other PGAS flavors.

### 3.2 Communication on lower level layers

All OpenSHMEM implementations will translate OpenSHMEM API functions to calls of lower transport layers. There are multiple such layers which may range from the second-level DMAPP library in Cray systems to the low-level Infiniband network layer, compare also the OSI model [6].

The presented event trace recording will only capture the OpenSHMEM layer but not the lower layers beneath. The primary reason for this is, that the performance analysis shall reflect the source code of the OpenSHMEM application. If a performance problem is detected in the way OpenSHMEM is used, then the (typical) optimization step will be to change the OpenSHMEM calls in the source code instead of changing details in the given OpenSHMEM library.

Should there be interest in the performance analysis of lower transport layers, then this should not be specific for OpenSHMEM but address this particular layer. For example, a monitoring extension for the low-level Infiniband layer might be used with OpenSHMEM, MPI, or GASPI. At the same time, it means that the connection between high-level API calls and low-level operations is lost.

### 3.3 Internal communication inside the monitoring system

Apart from the user-visible aspects, there are internal tasks in the monitoring system for bookkeeping and synchronization which depend on parallel communication. They usually rely on the same parallelization method as used by the target program to avoid conflicts between different parallelization libraries. Those tasks include the collective initialization and finalization of the monitoring instances, the synchronization of the local timers used by every parallel instance, the unification of identifiers in the event records, and more.

While this constitutes a considerable part of the effort when porting an existing monitoring system to OpenSHMEM, this paper focuses on the user’s perspective of performance analysis tools for parallel applications.

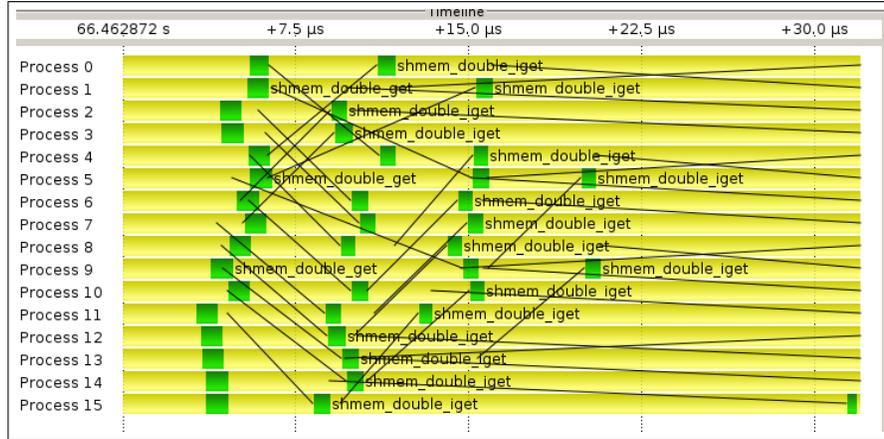
## 4 A demonstration for Cray-SHMEM with VampirTrace

We already developed an experimental solution to monitor the communication of SHMEM applications based on Cray-SHMEM. Because Cray-SHMEM can co-exist with MPI and because it guarantees that all SHMEM PE numbers are equal to the MPI ranks (within `MPI_COMM_WORLD`), the steps from Section 3.3 could be re-used from the existing MPI monitoring infrastructure. This was accomplished with the VampirTrace library [13]. `MPI_Init` is called just before `start_pes` and `MPI_Finalize` after `shmem_finalize`. The communication semantics of the SHMEM operations were mapped to related MPI operations, for example a `shmem_int_put` was mapped to `MPI_Put`.

The instrumentation of SHMEM API functions was realized by using the weak symbols of the Cray-SHMEM library. The Cray SHMEM library provides weak symbols for all library functions that can be overwritten by so called wrapper functions. Inside the wrapper functions the real call to the Cray SHMEM library is executed, yet before and after the data to be recorded is collected and stored to memory buffers. Eventually, the memory buffers are flushed to traces files. At the very end, a post processing is performed before the parallel event traces are ready to be analyzed with the Vampir visualization tool (see also further down).

### Demonstration example

Our demonstration example is a 2D simulation of the heat equation which runs with 16 PEs on a  $1500 \times 1500$  matrix [17]. We ran the program several times on a the petascale Cray XE6 system Hermit at HLRS Stuttgart. The heat equation



**Fig. 7.** Representation of a SHMEM communication section in Vampir. The run-time is shown horizontally from left to right while all PEs are arranged vertically.

program use a regular block-wise data distribution. Each node computes its own area for every iteration whereupon a halo exchange with the four neighbors is performed using one-sided operations.

### An example of SHMEM Performance Analysis

For performance analysis the parts of communication are of particular interest. Fig.4 shows a zoomed view of one of the communication sections in Vampir. The lines between the functions of the processes represent the direction of the communication. Vampir even provides further information such as the message size or the transfer rate in a more detailed view. The figure shows that the communication between the processes is done with the `shmem_double_get` and the `shmem_double_iget` functions. Besides individual communication operations, Vampir also provides a *communication matrix*, an overview of all communication operations between sender and receiver PEs.

Figure 8 shows the communication matrix for the average transfer time. The communication matrix indicates that the upper and lower borders (to/from the PE  $\pm 4$ ) are transferred faster than the left and right borders (to/from the PE  $\pm 1$ ). Yet, in all directions the same amount of data is transferred, as the same communication matrix view for the sum of message sizes would reveal.

The reason for the differing speeds are two different communication operations. For the horizontal halos `shmem_double_get` is used which can transfer the entire data block with a single low-level RMA operation. For the vertical halos the `shmem_double_iget` function has to be used, because the vertical halos of the two-dimensional array are not located in contiguous memory ranges. Thus, multiple small RMA operations have to be issued.

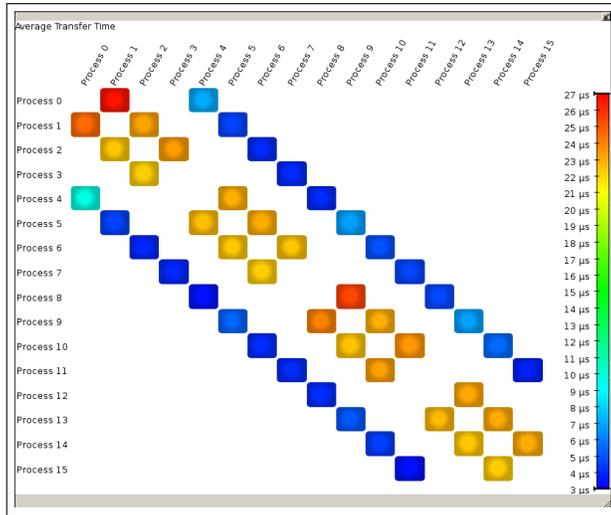


Fig. 8. View of Vampir communication matrix showing average transfer time.

### Overhead and perturbation

The monitoring approach explained above including the instrumentation and run-time data collection will induce a certain overhead, of course. As long as this overhead is small enough and evenly distributed over the entire test run, the resulting perturbation of the recorded trace will be negligible, i.e. the recorded behavior is sufficiently close to the “real” behavior without the presence of the monitoring system. Then, it is sensible to reason about the parallel performance behavior of the “real” program run based on the event trace analysis. If used with some caution and with the help of some advanced features of VampirTrace such as selective instrumentation and filtering the overhead can be controlled.

In our example we compared the wall clock times of the un-instrumented and the instrumented cases for eight different configurations with 4 to 81 PEs. For averages of 10 runs in each configuration, the instrumented execution time was within 6% of the original execution time.

## 5 Outlook to native OpenSHMEM support and Summary

The prototype solution described in the previous section shows an approach to the performance analysis of PGAS libraries. Implementing this for OpenSHMEM does pose some practical challenges. While the prototype assumes the direct relation to MPI, this is not generally the case for OpenSHMEM. Therefore the infrastructure of the monitoring library has to rely on OpenSHMEM for internal communication. OpenSHMEM does provide all the necessary communication primitives, including collectives, to implement measurement infrastructure such as the timer synchronization and the unification of distributed identifiers.

The instrumentation mechanism via weak symbols provides a reliable, portable and convenient way for the library wrapping step. While this functionality is optionally offered in the OpenSHMEM reference implementation, it is not currently defined by the standard. Performance analysis tools would greatly benefit if this was defined there. The MPI profiling interface (PMPI) has seeded a wide variety of tools - from lightweight profiling to flexible and complex measurement infrastructures. Alternatives using library pre-loading or linker wrapping are feasible but less straightforward, harder to make portable and generally divert the tool developers from core features.

Another open issue for implementing a parallel measurement infrastructure for OpenSHMEM is the lack of a finalization function. This would provide a reliable way to run measurement related code (e.g. combining results from multiple ranks) after the logical end of the application while it is still valid for the measurement infrastructure to use OpenSHMEM for communication. Also it would be guaranteed that the application does not call any other OpenSHMEM functions afterwards – so the recording is already complete at finalization time.

The possibilities for OpenSHMEM performance analysis described in this paper make no claim to be complete. It would be very interesting to hear from the OpenSHMEM user community what other information would be helpful to improve their applications. This extends to the developers of OpenSHMEM implementations that want to optimize their implementations. For instance, performance metrics that are internal to the OpenSHMEM implementation could be exposed to the monitoring infrastructure and recorded along with the application events. Examples might be the sizes of internal buffers or the current length of message queues.

## Summary

In the first part of this paper we presented a concept for the event trace recording for OpenSHMEM applications, in particular the representation of one-sided communication primitives as event records and their suggested visualization. The second part shows a preliminary solution for the event trace recording of Cray-SHMEM applications. It includes results from an example case together with the Vampir visualization of the produced traces and a brief study of the introduced run-time overhead. Finally, an outlook for a native event tracing tool for OpenSHMEM is given.

## Acknowledgments

This work is supported in a part by the German Research Foundation (DFG) in the Collaborative Research Center 912 “Highly Adaptive Energy-Efficient Computing“. The authors would like to thank the HLRS for providing the compute time on *Hermite* used for the Cray-SHMEM demonstration.

## References

1. L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.
2. T. Alrutz, J. Backhaus, T. Brandes, V. End, T. Gerhold, A. Geiger, D. Grünewald, V. Heuveline, J. Jägersküpfer, A. Knüpfer, O. Krzikalla, E. Kügeler, C. Lojewski, G. Lonsdale, R. Müller-Pfefferkorn, W. Nagel, L. Oden, F.-J. Pfreundt, M. Rahn, M. Sattler, M. Schmidtobreck, A. Schiller, C. Simmendinger, T. Soddemann, G. Sutmann, H. Weber, and J.-P. Weiss. GASPI – a partitioned global address space programming interface. In R. Keller, D. Kramer, and J.-P. Weiss, editors, *Facing the Multicore-Challenge III*, volume 7686 of *Lecture Notes in Computer Science*, pages 135–136. Springer Berlin Heidelberg, 2013.
3. W. W. Carlson, J. M. Draper, and D. E. Culler. S-246, 187 introduction to UPC and language specification.
4. B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing openshmem – shmem for the pgas community, 2010.
5. M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, Apr. 2010.
6. Information technology – Open Systems Interconnection – Basic Reference Model, 1994.
7. S. Jana and J. Schuchart. Tracing and visualizing power consumption of OpenSH-MEM applications. personal communications, Sept. 2013.
8. A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. Nagel. Introducing the open trace format (OTF). In V. Alexandrov, G. Albada, P. Sloot, and J. Dongarra, editors, *Computational Science – ICCS 2006*, volume 3992 of *Lecture Notes in Computer Science*, pages 526–533. Springer Berlin Heidelberg, 2006.
9. A. Knüpfer, R. Dietrich, J. Doleschal, M. Geimer, M.-A. Hermanns, C. Rössel, R. Tschüter, B. Wesarg, and F. Wolf. Generic support for remote memory access operations in Score-P and OTF2. In A. Cheptsov, S. Brinkmann, J. Gracia, M. M. Resch, and W. E. Nagel, editors, *Tools for High Performance Computing 2012*, pages 57–74. Springer Berlin Heidelberg, 2013.
10. A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, et al. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *Tools for High Performance Computing 2011*, pages 79–91. Springer Berlin Heidelberg, 2012.
11. Message Passing Interface Forum. MPI: A message-passing interface standard, version 2.2. Specification, September 2009.
12. B. P. Miller and A. R. Bernat. Anywhere, any time binary instrumentation. In *ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE)*, Szeged, Hungary, September 2011.
13. M. S. Müller, A. Knüpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. E. Nagel. Developing scalable applications with vampir, vampirserver and vampir-trace. In *Parallel Computing: Architectures, Algorithms and Applications*, volume 15, pages 637–644. IOS Press, 2008.
14. J. Nieplocha and B. Carpenter. Armci: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Lecture Notes in Computer Science*, pages 533–546. Springer-Verlag, 1999.

15. J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A non-uniform-memory-access programming model for high-performance computers. *THE JOURNAL OF SUPERCOMPUTING*, 10:10–197, 1996.
16. R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *ACM FORTRAN FORUM*, 17(2):1–31, 1998.
17. S. Oeste. Aufzeichnung einseitiger Kommunikation zur Leistungsanalyse paralleler SHMEM-Anwendungen, 2012. Bachelor thesis in German.
18. S. S. Shende and A. D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.
19. TU Dresden Center for Information Services and High Performance Computing (ZIH). *VampirTrace 5.14.4 User Manual*, 2013.