The GASPI API: A Failure Tolerant PGAS API for Asynchronous Dataflow on Heterogeneous Architectures

Christian Simmendinger, Mirko Rahn, and Daniel Gruenewald

Abstract The Global Address Space Programming Interface (GASPI) is a Partitioned Global Address Space (PGAS) API specification. The GASPI API specification is focused on three key objectives: scalability, flexibility and fault tolerance. It offers a small, yet powerful API composed of synchronization primitives, synchronous and asynchronous collectives, fine-grained control over one-sided read and write communication primitives, global atomics, passive receives, communication groups and communication queues. GASPI has been designed for one-sided RDMA-driven communication in a PGAS environment. As such, GASPI aims to initiate a paradigm shift from bulk-synchronous two-sided communication patterns towards an asynchronous communication and execution model. In order to achieve its much improved scaling behaviour GASPI leverages request based asynchronous dataflow with remote completion. In GASPI request based remote completion indicates that the operation has completed at the target window. The target hence can (on a per request basis) establish whether a one sided operation is complete at the target. A correspondingly implemented fine-grain asynchronous dataflow model can achieve a largely improved scaling behaviour relative to MPI.

1 Introduction

As the supercomputing community prepares for the era of exascale computing, there is a great deal of uncertainty about viable programming models for this new era. HPC programmers will have to write application codes for systems which are hundreds of times larger than the top supercomputers of today. It is unclear whether the two-decades-old MPI programming model [1], by itself, will make that transition gracefully. Despite recent efforts in MPI 3.0, which have significantly improved

C. Simmendinger (✉)
T-Systems Solutions for Research, Stuttgart, Germany
e-mail: christian.simmendinger@t-systems-sfr.com

M. Rahn • D. Gruenewald
Fraunhofer ITWM, Kaiserslautern, Germany
e-mail: mirko.rahn@itwm.fraunhofer.de; daniel.gruenewald@itwm.fraunhofer.de

M.M. Resch et al. (eds.), *Sustained Simulation Performance 2014*,
DOI 10.1007/978-3-319-10626-7_2

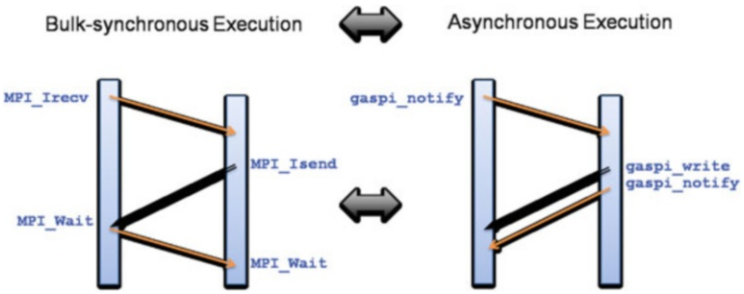

Fig. 1 MPI two-sided communication vs GASPI notify. An MPI_Irecv is complete at the receiver side whenever the corresponding wait returns. IN GASPI the one sided communication is complete at the receiver side, whenever the test for the corresponding notification successfully returns. Remark: The initial gaspi_notify here requests the subsequent write and notify, but this initial request is not mandatory

support for one-sided communication and an asynchronous execution model, MPI offers little support for asynchronous dataflow models.

We here present an alternative to the programming model of MPI. GASPI is a Partitioned Global Address Space (PGAS) API. In contrast to MPI, GASPI leverages one-sided RDMA driven communication with remote completion in a Partitioned Global Address Space. In GASPI data may be written asynchronously and one-sided, whenever it is produced, along with a corresponding notification. On the receiving side GASPI guarantees that data is locally available whenever this notification becomes locally visible. This mechanism enables fine-grain (request based) asynchronous dataflow implementations on top of the GASPI API (see also Fig. 1).

The notification mechanism in GASPI enables its users to reformulate existing bulk-synchronous MPI applications towards an asynchronous dataflow model with a very fine-grained overlap of communication and computation. GASPI leverages zero-copy asynchronous writes and reads with RDMA queues and aims at a minimum of communication overhead.

GASPI allows for a lot of flexibility in using its Partitioned Global Address Spaces (which in GASPI are called segments), but does not enforce a specific memory model. We remark that implementing a symmetric global memory management on top of the GASPI segments is straightforward. We have, however, opted against including this latter functionality in the core API, since enforcing a symmetric global memory management would make the GASPI API a bad match for e.g irregular problems.

GASPI allows its users to span multiple segments with configurable sizes and configurable participating ranks. GASPI also supports a variety of devices for its segments, like e.g GPGPU memory, main memory of Xeon Phi cards, main memory of host nodes or non-volatile memory. All these segments can directly read/write from/to each other—within the node and across all nodes.

With a growing number of nodes, failure tolerance becomes a major issue as machines expand in size. On systems with large numbers of processes, all non-local communication should be prepared for a potential failure of one of the communication partners. GASPI features timeouts for non-local functions, allows for shrinking or growing node sets and enables applications to recover from node-failures.

In contrast to other efforts in the PGAS community, GASPI is neither a new language (like e.g. Chapel from Cray [2], UPC [3] or Titanium [4]), nor an extension to a language (like e.g. Co-Array Fortran [5]). Instead—very much in the spirit of MPI—it complements existing languages like C/C++ or Fortran with a PGAS API which enables the application to leverage the concept of the Partitioned Global Address Space.

While other approaches to a PGAS API exist (e.g. OpenShmem or Global Arrays), they lack remote completion, support for multiple segments and failure tolerance.

2 GASPI Overview

2.1 History

GASPI inherits much of its design from the Global address space Programming Interface (GPI) [6, 7], which was developed in 2005 at the Competence Center for High Performance Computing (CC-HPC) at Fraunhofer ITWM. GPI is implemented as a low-latency communication library and is designed for scalable, real-time parallel applications running on cluster systems. It provides a PGAS API and includes communication primitives, environment run-time checks and synchronization primitives such as fast barriers or global atomic counters.

GPI communication is asynchronous, one-sided and, most importantly, does not interfere with the computation on the CPU. Minimal communication overhead can be realized by overlapping communication and computation. GPI also provides a simple, run-time system to handle large data sets, as well as dynamic and irregular applications that are I/O- and compute-intensive. As of today, there are production-quality implementations for ×86 and IBM Cell/B.E architectures.

GPI has been used to implement and optimize CC-HPC industry applications like the Generalized Radon Transform (GRT) method in seismic imaging or the seismic work flow and visualization suite PSPRO. Today, GPI is installed on Tier 0 supercomputer sites in Europe, including the HLRS in Stuttgart and the Juelich Supercomputing Centre.

The GPI library has yielded some promising results in a number of situations. In particular, GPI outperforms MPI in significant low-level benchmarks. For process to process communication, GPI asynchronous one-sided communication, as opposed to both MPI one-sided communication and MPI bulk-synchronous two sided-

communication, delivers full hardware bandwidth. As a function of message size, GPI reaches its peak performance much earlier than MPI.

GPI has also shown excellent scalability in a broad spectrum of typical real world HPC applications like the Computational Fluid Dynamics (TAU code from the DLR) [8], or BQCD, a four dimensional nearest neighbor stencil algorithm. GPI has also been used in the implementation of fastest Unbalanced Tree Search (UTS) benchmark on the market [9].

In 2010 the request for a standardization of the GPI interface emerged, which ultimately lead to the inception of the GASPI project in 2011. The work was funded by the German Ministry of Education and Science and included project partners Fraunhofer ITWM and SCAI, T-Systems SfR, TU Dresden, DLR, KIT, FZJ, DWD and Scapos.

2.2 Goals

The GASPI project intends to drive the dissemination and visibility of the API by means of highly visible lighthouse projects in specific application domains, including CFD, turbo-machinery, weather and climate, oil and gas, molecular dynamics, as well as in the area of sparse and dense matrices. On the basis of these applications, requirements for the emerging standard have been analyzed. The following requirements have been derived:

- fine-grained control over one-sided asynchronous read/write operations
- collective operations (also on a subset of ranks)
- passive communication
- atomic counters

For the GASPI API we have set the following design goals:

- Extreme scalability, targeting both performance and minimal resource requirements
- Timeout mechanisms and failure tolerance
- Multi-segment support
- Dynamic allocation of segments
- Group support for collectives
- Large flexibility in runtime parameter configuration
- A slim, yet powerful core API providing a minimal still complete set of functionality
- A strong standard library extension, which takes care of convenience procedures.
- A maximum of freedom for the implementation

Ultimately the GASPI project aims at establishing a de-facto standard for an API for scalable, fault-tolerant and flexible communication in a Partitioned Global Address Space.

3 The GASPI Concepts

3.1 GASPI Execution Model

GASPI follows an SPMD (Single Program, Multiple Data) style in its approach to parallelism. Hence, a single program is started and initialized on all desired target computational units. How the GASPI application is started and initialized is not defined by the standard and is implementation specific.

GASPI provides the concept of ranks. Similarly to MPI each GASPI process receives a unique rank with which it can be identified during runtime. Ranks are a central aspect which allows applications to identify processes and to assign different tasks or data to the processing elements.

GASPI also provides the concept of segments. Segments are PGAS memory regions which can be globally available, to be written to or read from.

The GASPI API has been designed to coexist with MPI and hence in principle provides the possibility to complement MPI with a Partitioned Global Address Space. We note however, that while such an approach provides an opportunity for increased scalability, fault-tolerant execution will not be possible due to the corresponding limitations of MPI. GASPI aims at providing interoperability with MPI in order to allow for incremental porting of such applications.

The start-up of mixed MPI and GASPI code is achieved by invoking the GASPI initialization procedure in an existing MPI program. This way, MPI takes care of distributing and starting the binary and GASPI just takes care of setting up its internal infrastructure.

GASPI provides high flexibility in the configuration of the runtime parameters for the processes and allows for a dynamic process set during runtime. In case of a node failure, a GASPI process can be started on a new host, freshly allocated or selected from a set of pre-allocated spare hosts. By providing a modified list of machines (in which the failed node would be substituted by the new host) the new GASPI process would register with the other processes and receive the rank of the failed GASPI process.

Similarly, in case of starting additional GASPI processes, these additional GASPI processes have to register with the existing GASPI processes.

3.2 GASPI Groups

Groups are sub-sets of processes identified by a sub-set of the total set of ranks. The group members have common collective operations. A collective operation is then restricted to the ranks forming the group. Each GASPI process can participate in more than one group.

A group has to be defined and declared in each of the participating GASPI processes. Defining a group is a two step procedure. An empty group has to be created first. Then the participating GASPI processes, represented by their rank,

have to be attached. The group definition is a local operation. In order to activate the group, the group has to be committed by each of the participating GASPI processes. This is a collective operation for the group. Only after successful group commitment the group can be used for collective operations.

The maximum number of groups allowed per GASPI process is restricted by the implementation. A desired value can be passed to the GASPI initialization procedure.

In case of failure, where one of the GASPI processes included within a given group fails, the group has to be reestablished. If there is a new process replacing the failed one, the group has to be defined and declared on the new GASPI process(es). Reestablishment of the group is then achieved by re-commitment of the group by the GASPI processes which were still "alive" (functioning) and by the commitment of the group by the new GASPI processes.

3.3 GASPI Segments

GASPI does not enforce a specific memory model, like, for example, the symmetric distributed memory management of OpenSHMEM [10]. Rather GASPI offers PGAS in the form of configurable RDMA pinned memory segments. Since an application can request several PGAS segments symmetric, asymmetric or stack based memory management models can readily coexist.

Modern hardware typically involves a hierarchy of memory with respect to the bandwidth and latencies of read and write accesses. Within that hierarchy are non-uniform memory access (*NUMA*) partitions, solid state devices (*SSD*s), graphical processing unit (*GPU*) memory or many integrated cores (*MIC*) memory.

The GASPI memory segments can thus be uses as an abstraction, which represents any kind of memory level, mapping the variety of hardware layers to the software layer. A segment is a contiguous block of virtual memory. In the spirit of the PGAS approach, these GASPI segments may be globally accessible from every thread of every GASPI process and represent the partitions of the global address space (Fig. 2).

By means of the GASPI memory segments it is also possible for multiple memory models or indeed multiple applications to share a single Partitioned Global Address Space. The segments can be accessed as global, common memory, whether local—by means of regular memory operations—or remote, by means of the communication routines of GASPI. Allocations inside of the pre-allocated segment memory are managed by the application.

In a Partitioned Global Address Space, every thread can asynchronously read and write the entire global memory of an application. On modern machines with RDMA engines, an asynchronous PGAS programming model appears as a natural extension and abstraction of available hardware functionality.

For systems with DMA engines (such as tile architectures), this also holds true for a node-local level.

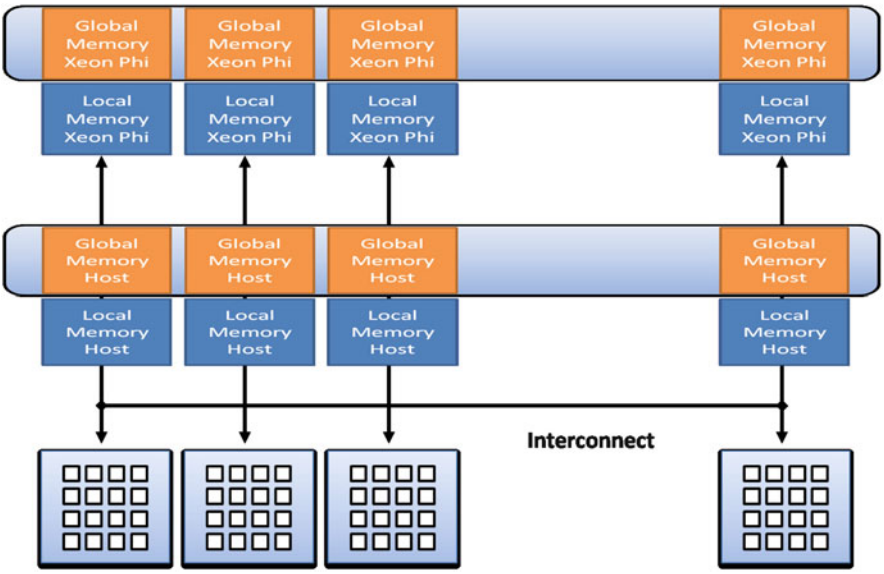

Fig. 2 Memory segments in a Partitioned Global Address Space. Here one segments is spawned across all Xeon Phi cards and a different segment is spawned across the ×86 host memory. The segments can directly read/write from/to each other—within the node and across all nodes

4 GASPI One-Sided Communication

One sided asynchronous communication is the basic communication mechanism provided by GASPI. Here one GASPI process specifies all communication parameters, both for the local and the remote side.

GASPI provides the concept of message queues. All operations placed on a certain queue q by one or several threads are finished after a single wait call for q has returned. These queues facilitate higher scalability and can be used as channels for different types of requests where similar types of requests are queued and then get synchronized together but independently from the other ones (separation of concerns, e.g. one queue for operations on data and another queue for operations on meta data). The several message queues guarantee fair communication, i.e. no queue should see its communication requests delayed indefinitely. Furthermore, the queue preserves the order of the messages on the local and the remote side if the remote rank of two messages is the same. The number of message queues and their size can be configured at initialization time, otherwise default values will be used. The default values are implementation dependent. Maximum values are also defined.

Memory addresses within the global partitioned address space are specified by the triple consisting of the rank, segment identifier and the offset. This prevents a

global all to all distribution of memory addresses, since the memory addresses of memory segments normally are different on different GASPI processes,

4.1 Basic Calls

The one-sided communication calls are derived from two fundamental operations types: read and write. The read operation initiates the data transfer from a remote part of a PGAS segment to the local part of a PGAS segment. The write operation initiates the data transfer from a local part of a PGAS segment to the remote part of a PGAS segment.

Listing 1 shows an alltoall implementation with one-sided writes. This implementation performs an out-of-place matrix transpose of a distributed square matrix. Each process iProc hosts one row (src[iProc]) of the distributed matrix and also one row (dst[iProc]) of the transposed matrix.

```
let nProc be the number of processes;
let iProc be the unique local process id;
let src[iProc] be an array of size nProc;
let dst[iProc] be an array of size nProc;

foreach process p in [0,nProc):
  write src[iProc][p] into dst[p][iProc];
  // ^^^^^^^^^^ ^^^^^^
  // | local | remote if p != iProc

work()
// do some work along the data transfer

wait for the completion of the writes;
// now this process sent the data

barrier;
// now all processes sent the data
```

Listing 1 Alltoall with one-sided writes

Each process loops over its row entries src[iProc][p] and initiates the data transfer to the corresponding row entry src[p][iProc] of process p. The communication requests are posted to one of the communication queues and are asynchronously processed by the network infrastructure.

Parallel to the data transfer, a local task (work) can be executed in order to overlap the communication with the computation. Upon completion of the local task, the local process needs to know whether the source buffers can be reused again. Hence a wait call has been invoked on the respective queue. This is a blocking call. However,

if the computation of the local task has been sufficiently long, the data transfer has finished and the wait call returns immediately.

4.2 Weak Synchronization

One-sided communication procedures have the characteristics that the entire communication is managed by the local process only. The remote process is not involved. This has the advantage that there is no inherent synchronization between the local and the remote process in every communication request. Nevertheless, at some point, the remote process needs the information as to whether the data which has been sent to that process has arrived and is valid.

To this end, GASPI provides so-called weak synchronization primitives which update a notification on the remote side. In order to ensure an ordered delivery of data payload and notifications, the notifications have to be posted to the same queue to which the data payload has been posted.

The notification semantic is complemented with routines which wait for an update of a single or even an entire set of notifications. In order to manage these notifications in a thread-safe manner GASPI provides a thread safe atomic function to reset local notification with a given ID. The atomic function returns the value of the notification before reset.

The notification procedures are one-sided and only involve the local process.

```
let buf[0] and buf[1] be data buffers;
let bnd[0] and bnd[1] be boundary buffers;
let hal[0] and hal[1] be halo buffers;
set b to 0;

while (!done)
{
   // send boundary data
   write bnd[b] to remote hal[b]
   // send boundary domain
   // validity notification
   notify remote hal exchange

  // update inner domain
  buf[1-b] = compute(buf[b],bnd[b])

  // wait for halo data validity
  notify_wait_some
```

```
  {
    // atomically reset notification
    notify_reset
    // update boundary domain
    bnd[1-b] = compute(bnd[b],hal[b])
  }
  // swap buffers
  set b to 1-b;
}
```

Listing 2 Computation involving weak synchronization and double buffering

Listing 2 shows a weak synchronization example with double buffering. Here, a non-local operator is propagated in an iteration loop, which uses domain decomposition. This pattern usually appears in a similar way in most parallel stencil algorithms. Due to the non-locality of the computational kernel, the individual GASPI processes are tightly coupled throughout the computation.

The local data is partitioned into three sub-domains. Inner data, boundary data and halo data. Each of these sub-domains is represented by two buffers residing in the local partition of the global address space. They are swapped after each iteration. In order to compute the inner data, information from the boundary data is required. In order to compute the boundary data, the remote boundary data—which is locally stored in the halo data—is required. There hence are data dependencies among the local sub-domains and on an inter-process level. In GASPI, the remote data dependencies are mapped a notifications.

Inside of the iteration loop, first the boundary data is sent to the remote side and a subsequent notification is emitted in order to inform the remote process about the validity of its halo data in the next time step. Subsequently the inner data is updated. The local GASPI process then waits in a notify_wait_some ensuring the halo data validity from the remote side. As soon as the data arrives, the local notification flag is reset and the update of the boundary data is performed. At the end, the data buffers are swapped for the next iteration.

By using the double buffering mechanism together with the weak synchronization, we can avoid all potential data race conditions. The communication of the boundary data completely overlaps with the computation of the inner domain. The weak synchronization reduces the coupling of the processes to a minimum (relaxed synchronization model). The local process is not interrupted by the communication from the remote process.

4.3 Extended Calls

Beside the basic and the weak synchronization calls, GASPI also provides extended one-sided communication calls

- `gaspi_write_notfiy`
- `gaspi_write_list`

- `gaspi_read_list`
- `gaspi_write_list_notify`

These extended calls are semantically equivalent to subsequent calls of basic and/or weak synchronization calls. However, GASPI provides an independent definition in order to allow for implementations which can leverage hardware specific optimizations.

For the write operations, GASPI provides three additional extended calls. One is the communication of a single message combined with a subsequent notification event on the remote side. The remote side can wait on the corresponding notification flag. If the notification event is detected, the data payload has arrived and is valid. The second call initiates a communication call of arbitrarily distributed data in n contiguous blocks. It is semantically equivalent to n subsequent calls of ordinary writes to the same destination rank using the same communication queue.

For this second call a list of offsets for local and the remote side as well as corresponding block sizes has to be provided. The list of offsets does not need to be equal on the local and the remote side. The third call is a combination of the second call with a subsequent notification event on the remote side.

5 GASPI Passive Communication

GASPI also provides two-sided semantics—where a send request requires a matching receiver—in the form of passive communication. Passive communication aims at communication patterns where the sender is unknown (i.e. it can be any process from the receiver perspective) but there is a potentially need for synchronization between processes. Typical example uses cases are:

- Distributed update
- Pass arguments and results
- Atomic operations

The implementation should try to enforce fairness in communication that is, no sender should see its communication request delayed indefinitely.

The passive keyword means that the communication calls avoid busy-waiting, computation. Instead passive receives are triggered by an incoming message in a predefined communication queue.

The send request is asynchronous, whereas the matching receive is *time-based blocking*. Due to the asynchronous nature of the send request, a complete send involves two procedure calls. First, one call which initiates the communication. This call posts a communication request to the underlying network infrastructure. If the application wants to update the data-payload which has been send a second call is required, which waits for the completion of the communication request.

In passing we note that GASPI never uses communication buffers—all calls directly read and write application data.

```
** consumer:
let buffer be one data buffer;
while (!done)
{
  passive_receive into buffer;
  process (buffer);
}

** producer:
let buffer[0], buffer[1] be data buffers;
set b to 0;
while (!done)
{
  produce data in buffer[b];

  wait for the completion of earlier
    passive_send;
  passive_send data from buffer[b]
    to consumer;

  set b to 1-b;
}
```

Listing 3 Single consumer and multiple producers using passive communication. The producer transfers a data packet while producing the next data packet, thus overlapping computation and communication

Listing 3 shows a single consumer multiple producer example using passive communication with overlap of communication and computation on the producer side. The consumer sleeps in a passive receive and waits until some data is received into its receive buffer. As soon as the data has been received, the consumer wakes up and processes the data. This is repeated until some termination condition is met. The producers use double buffering. One buffer is used for the computation. The second buffer is used for the communication. The producer is computing some data along the data transfer. Before the data is sent to the consumer after the computation has been done, a wait operation on the previous non-blocking send is invoked in order to ensure that the communication buffer can be used for storing the result of the next computation. If this is the case, the data transfer is initiated and the communication and computation buffers are swapped. On the producer side, this pattern is also repeated until some termination condition is met.

Such single consumer and multiple producer patterns can be used for example in order to implement a global convergence check which is not time critical and which hence can be offloaded to an extra thread without polluting the actual computation threads.

6 GASPI Global Atomics

GASPI provides atomic counters, i.e. globally accessible integral types that can be manipulated through atomic procedures. These atomic procedures are guaranteed to execute from start to end without fear of preemption causing corruption. GASPI provides two basic operations on atomic counters: `fetch_and_add` and `compare_and_swap`. The counters can be used as global shared variables used to synchronize processes or events. Atomic counters are predestined for the implementation of dynamic load balancing schemes. An example is shown in Listing 4. Here, clients atomically fetch a working package ID and increment the value of the package ID counter by one. In accordance with the package ID a corresponding part of the total work load is performed by the local process. This procedure is repeated until all working packages have been processed.

The standard guarantees fairness, i.e. no process should have its atomic operation delayed indefinitely.

The number of atomic counters available can be defined by the user through the configuration structure at start-up and cannot be changed during run-time. The maximum number of available atomic counters is implementation dependent.

```
do
{
  packet := fetch_and_add (1);
  // increment the value by one,
  // return the old value

  if (packet < packet_max):
    process (packet);
}
while (packet < packet_max);
```

Listing 4 Dynamic work distribution: Clients atomically fetch a packet id and increment the value

7 GASPI Collective Communication

Collective operations are operations which involve a whole set of GASPI processes. That means that collective operations are collective with respect to a group of ranks. They are also exclusive per group, i.e. only one collective operation of a specific type can run at a time. For example, two allreduce operations for one group can not run at the same time; however, an allreduce and a barrier can run at the same time.

Collective operations have their own queue and hence typically will be synchronized independently from the operations on other queues (separation of concerns).

Collective operations can be either synchronous or asynchronous. Synchronous implies that progress is achieved only as long as the application is inside of the call. The call itself, however, may be interrupted by a timeout. The operation is then continued in the next call of the procedure. This implies that a collective operation may involve several procedure calls until completion.

Please note that collective operations can internally also be handled asynchronously, i.e. with progress being achieved outside of the call.

GASPI does not regulate whether individual collective operations should internally being handled synchronously or asynchronously, however: GASPI aims at an efficient, low-overhead programming model. If asynchronous operation is supported, it should leverage external network-resources, rather than consuming CPU cycles.

Beside barriers and reductions with predefined operations, reductions with user defined operations are also supported via callback functions.

Not every collective operation will be implementable in an asynchronous fashion—for example if a user-defined callback function is used within a global reduction. Progress in this case can only be achieved inside of the call. Especially for large systems this implies that a collective potentially has to be called a substantial number of times in order to complete—especially if used in combination with `GASPI_TEST`. In this combination the called collective immediately returns (after completing an atomic portion of local work) and never waits for messages from remote processes. A corresponding code fragment in this case would assume the form:

```
while ( GASPI_allreduce_user (
          buffer_send
        , buffer_receive
        , char num
        , size_element
        , reduce_operation
        , reduce_state
        , group
        , GASPI_TEST ) != GASPI_SUCCESS )
{
    work();
}
```

Listing 5 Collective communication

Listing 5 shows an user defined allreduce using `GASPI_TEST` in order to overlap the reduction operation with work.

8 GASPI Failure Tolerance

8.1 GASPI Timeouts

On systems with large number of processes, all non-local communication should be prepared for a potential failure of one of the communication partners. In GASPI this is accomplished by providing a timeout value as an argument to all non-local communication calls and the possibility to check for the state of each of the communication partners.

The timeout for a given procedure is specified in milliseconds. `GASPI_BLOCK` is a special predefined timeout value which blocks the invoked procedure until the procedure is completed. This special value should not be used in a failure tolerant program, because in a situation in which the procedure cannot complete due to a failure on a remote process, the procedure will not return at all.

`GASPI_TEST` is special predefined timeout value which represents a timeout equal to zero. Timeout equal to zero means that the invoked procedure processes an atomic portion of the work and returns after that work has finished. It does not mean that the invoked procedure is doing nothing. It does not mean that the invoked procedure returns immediately.

8.2 GASPI Error Vector

GASPI provides a predefined vector type to describe the state of the communication partners of each local GASPI process. Each entry of this error vector corresponds to one remote process. The state of a remote process is denoted to be either *HEALTHY* or *CORRUPT*. After a failed completion of a non-local procedure call, the error vector can be updated by means of a dedicated update procedure. Subsequently the state of the communication partners can be analyzed. In case of a failure of the communication partners, the process can enter a recovery phase.

Conclusion
We have presented the Global Address Space Programming Interface (GASPI) as an alternative to the programming model of MPI. GASPI is a Partitioned Global Address Space (PGAS) API, targeting both extreme scalability and failure-tolerance. GASPI follows a single program multiple data (SPMD) approach and offers a small, yet powerful API composed of synchronization primitives, synchronous and asynchronous collectives, fine-grained control over one-sided read and write communication primitives,

(continued)

global atomics, passive receives, communication groups and communication queues.

The GASPI standard is currently being implemented in two flavors: a highly portable open source implementation and a commercial implementation aimed at ultimate performance. This latter implementation will be based on GPI. The TU Dresden, ZIH will provide profiling support for GASPI by means of extending the VAMPIR tool suite.

There are a number of related projects which pursue similar goals as GASPI (such as OpenSHMEM). The GASPI hence sees itself as one of the precursors towards establishing a de-facto standard for an API for scalable, fault-tolerant and flexible communication in a Partitioned Global Address Space.

Acknowledgements The authors would like to thank the German Ministry of Education and Science for funding the GASPI project (funding code 01IH11007A) within the program "ICT 2020 - research for innovation". Further more, the authors are grateful to all of project partners for having fruitful and constructive discussions.

References

1. MPI Forum: MPI: A Message Passing Interface Standard, http://www.mpi-forum.org/docs/docs.html (2014)
2. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the chapel language. Int. J. High Perform. Comput. Appl. **21**(3), 291–312 (2007)
3. Consortium, U.: UPC Specifications v1.2. Lawrence Berkeley National Lab Tech Report LBNL-59208 (2005)
4. Hilfinger, P., Bonachea,D., Datta, K., Gay, D., Graham, S., Liblit, B., Pike, G., Su, J., Yelick, K.: Titanium Language Reference Manual. U.C. Berkeley Tech Report UCB/EECS-2005-15 (2005)
5. Numrich, R.W., Reid, J.: Co-array Fortran for parallel programming. ACM Fortran Forum **17**, 1–31 (1998)
6. Machado, R., Lojewski, C.: The Fraunhofer virtual machine: a communication library and runtime system based on the RDMA model. Comput. Sci. Res. Dev. **23**, 125–132 (2009)
7. Fraunhofer ITWM: GPI - Global Address space programming Interface. http://www.gpi-site.com (2013)
8. Simmendinger, C., Jägersküpper, J., Machado, R., Lojewski, C.: A PGAS-based implementation for the unstructured CFD solver TAU. In: 5th Conference on Partitioned Global Address Space Programming Models, Tremont House, Galveston Island (2011)
9. Machado, R., Lojewski, C., Abreu, S., Pfreundt, F.-J.: Unbalanced tree search on a manycore system using the GPI programming model. Comput. Sci. Res. Dev. **26**(3–4), 229–236 (2011)
10. Poole, S.W., Hernandez, O., Kuehn, J.A., Shipman, G.M., Curtis, A., Feind, K.: OpenSHMEM - toward a unified RMA model. In: Encyclopedia of Parallel Computing, pp. 1379–1391 (2011)

http://www.springer.com/978-3-319-10625-0

Sustained Simulation Performance 2014
Proceedings of the joint Workshop on Sustained Simulation Performance, University of Stuttgart (HLRS) and Tohoku University, 2014
Resch, M.M.; Bez, W.; Focht, E.; Kobayashi, H.; Patel, N. (Eds.)
2015, XIII, 238 p. 154 illus., 132 illus. in color.,
Hardcover
ISBN: 978-3-319-10625-0