Interoperability of GASPI and MPI in Large Scale Scientific Applications

Dana Akhmetova[1], Luis Cebamanos[2], Roman Iakymchuk[1], Tiberiu Rotaru[3], Mirko Rahn[3], Stefano Markidis[1], Erwin Laure[1], Valeria Bartsch[3], Christian Simmendinger[4]

[1] KTH Royal Institute of Technology, Sweden
{danaak,riakymch,markidis,erwinl}@kth.se
[2] EPCC, The University of Edinburgh, UK l.cebamanos@epcc.ed.ac.uk
[3] Fraunhofer ITWM, Germany
{tiberiu.rotaru,mirko.rahn,valeria.bartsch}@itwm.fraunhofer.de
[4] T-Systems Solutions for Research, Germany
christian.simmendinger@t-systems-sfr.com

Abstract. One of the main hurdles of a broad distribution of PGAS approaches is the prevalence of MPI, which as a de-facto standard appears in the code basis of many applications. To take advantage of the PGAS APIs like GASPI without a major change in the code basis, interoperability between MPI and PGAS approaches needs to be ensured. In this article, we address this challenge by providing our study and preliminary performance results regarding interoperating GASPI and MPI on the performance crucial parts of the Ludwig and iPIC3D applications. In addition, we draw a strategy for better coupling of both APIs.

Keywords: Interoperability, GASPI, MPI, Ludwig, iPIC3D, halo exchange.

1 Introduction

The Message Passing Interface (MPI) has been considered the de-facto standard for writing parallel programs for clusters of computers for more than two decades. Although the API has become very powerful and rich, having passed through several major revisions, new alternative models that are taking into account modern hardware architectures have evolved in parallel. Such a model is the *Global Address Space Programming Interface (GASPI)* [9], with *GPI-2*[5] representing an open source implementation of the GASPI standard.

The GASPI standard promotes the use of *one-sided communication*, where one side, the initiator, has all the relevant information for performing the data movement. The benefit of this is decoupling the data movement from the synchronization between processes. It enables the processes to put or get data from remote memory, without engaging the corresponding remote process, or having a synchronization point for every communication request. However, some form of synchronization is still needed in order to allow the remote process to be notified upon the completion of an operation. In addition, GASPI provides what is known as weak synchronization primitives which update a notification on the

[5] www.github.com/cc-hpc-itwm/GPI-2

remote side. The notification semantics is complemented with routines that wait for the update of a single or a set of notifications. GASPI allows for a thread-safe handling of notifications, providing an atomic function for resetting a local notification. The notification procedures are one-sided and only involve the local process.

Thus, there is a potential of enhancing applications' performance by shifting to one-sided communication like in GASPI. There are two possibilities for such shift: 1. Rewriting large legacy MPI codes to use a different inter-node programming model is, in many cases, highly labor intensive and, therefore, not appealing to developers; 2. Replacing MPI with another API – such as GASPI – only in performance critical parts of those codes is an attractive solution from a practical perspective, but this requires both APIs to interoperate effectively and efficiently on sharing communication and on data management. In this article, we address the latter and aim to study *interoperability* of GASPI and MPI in order to allow for incremental porting of applications. GPI-2 supports [5] this interoperability with MPI in a so-called mixed-mode, where the MPI and GASPI interfaces can be mixed in a simple way. As a case study, we consider two large-scale scientific applications: iPIC3D [7] (see Section 3) – an implicit Particle-In-Cell code for space weather simulations; Ludwig [3] (see Section 4) – a large scale Lattice-Boltzmann code for complex fluids. We collect the preliminary performance results for both applications (see Section 5). Furthermore, we derive a strategy for enhancing the MPI and GASPI coupling using so-called shared notifications (see Section 2) and provide evidences that this strategy is beneficial (see Section 5) on simple operations such as Allreduce.

2 A Strategy to Better Interoperate GASPI and MPI

Scientific applications may use MPI features – such as MPI derived data types – not known in the GASPI specification. Due to the fact that GASPI does not support the derived data types the interoperability between MPI and GASPI within a program using MPI derived data types lacks ease of use, because the data of each local process on a node have to be packed, sent, and, then, unpacked.

To mitigate the adverse effect of MPI derived data types on the MPI plus GASPI interoperability so-called shared notifications have been recently implemented in the GPI implementation of the GASPI standard. This feature allows a smoother interoperability between a flat MPI code with shared windows and GASPI. With shared notifications a GASPI memory segment is shared between all processes local to a node. GASPI segments can be created with user allocated memory, e.g. using MPI shared windows in an MPI plus GASPI mixed program. Instead of implicit (via derived data types) or explicit packing/unpacking of communication data, application can share information about node local data layout, structure, and computational state. As all node-local processes can access this shared data, the node local explicit ghost cell exchanges in applications can be replaced with the corresponding state notifications, where the required data can be directly read from the neighboring processes based on previously exchanged information of data layout and type. We believe that the correspondingly required programming interface can be generic and – for node local exchanges – common for both MPI and GASPI. The interface will require

an allocation of a shared memory segment across node-local processes. It will require a universally acceptable format for sharing of process local data layouts and corresponding data offsets. It will require the ability to automatically detect whether or not a neighboring process is node-local; the latter information can be used to signal node-local readiness for the ghost-cell exchange or to perform explicit packing and/or unpacking into/from linear communication buffers for remote nodes. The interface will also require the ability to trigger node-local notifications in shared memory. This will include required memory fences between neighboring node local processes. Last not least – by using shared notifications – the interface becomes able to aggregate data for remote nodes and to perform one single write to the other node (for all local processes on that node) and notify all remote local processes in one step. As all remote processes can detect and access this common buffer, each remote process/rank can retrieve the required partial data for its ghost cell exchange. The ongoing, but converging, development of this generic interface will facilitate the interoperability of MPI and GASPI significantly.

3 iPIC3D: implicit Particle-in-Cell Code

iPIC3D is a Particle-in-Cell (PIC) code for the simulation of space plasmas in space weather applications during the interaction between the solar wind and the Earth's magnetic field. The magnetosphere is a large system with many complex physical processes, requiring realistic domain sizes and billions of computational particles. The numerical discretization of Maxwell's equations and particle equations of motion is based on the implicit moment method that allows simulations with large time steps and grid spacing still retaining the numerical stability. Plasma particles from the solar wind are mimicked by computational particles. At each computational cycle, the velocity and the location of each particle are updated, the current and charge densities are interpolated to the mesh grid, and Maxwell's equations are solved. Figure 1 depicts these computational steps in iPIC3D.

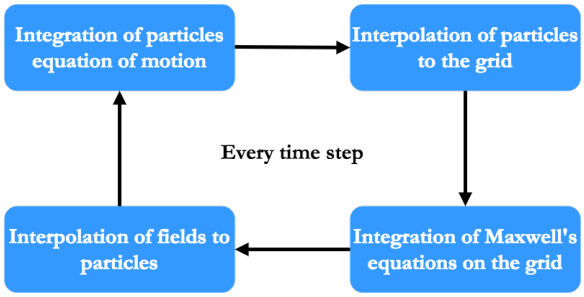


Fig. 1: Structure of the iPIC3D code.

iPIC3D is parallelized using domain decomposition and message-passing communications: an iPIC3D simulation is being run on a number of processors and on a network of cells, so each processor handles a number of cells. However, at certain intervals, each processor must find out the values of the cells adjacent to those in its own domain. The procedure of finding these values out is called

halo exchange. To achieve the full 3D halo exchange, the standard approach of shifting the relevant data in each co-ordinate direction in turn is adopted. This involves extensive communication between processes and requires appropriate synchronization – a receive in the first co-ordinate direction must be complete before a send in the second direction involving relevant data can take place, and so on. Note that only "outgoing" elements of the distribution need to be sent at each edge. In the particle mover part hundreds of particles per cell are constantly moved, resulting in billions of particles in large-scale simulations. All these particles are completely independent from each other, which ensures very high scalability. MPI communication at this stage is only required to transfer some of the particles from one cell or a subdomain to its neighbor.

The iPIC3D MPI communication is dominated by non-blocking point-to-point communication, occurring from communication of particles and ghost cells among neighboring processes (halo exchange), and by global reductions resulting from solving two linear systems every simulation time step. In order to reduce the communication burden in iPIC3D, we aim at replacing the MPI communication with the GASPI asynchronous one-sided communication on the communication critical parts of the code such as halo exchange in the field solver and with the GASPI reduction communication in the iPIC3D linear solver.

Implementation Highlights The main halo exchange routine uses non-blocking MPI and MPI derived datatypes. MPI derived datatypes allow us to specify non-contiguous data in a convenient manner and yet treat it as if it was contiguous. GASPI requires the creation and later use of the so-called GASPI segments. In the case of iPIC3D, there is one GASPI segment per plane and direction. As there are three planes and two directions per plane, iPIC3D will require six different GASPI segments. The size of the segments is defined as twice the size of buffer to be sent as we will use the same segment to send and receive data from the neighbor subdomains. As iPIC3D uses MPI datatypes, complex data layouts, it is necessary to unpack the MPI datatypes and copy the data contiguously into a GASPI segment. Once the data has been sent and notified, we need to put the data back from the GASPI segment to the original buffer to be able to continue with the execution of iPIC3D.

To implement the halo exchange with GASPI, firstly the field values belonging to the boundary are being copied to the local GASPI segment. Secondly, segments of neighbors are being read to get their ghost cells and copied to the local segment. The local copy does not require a barrier: each process writes to its neighbor process' segment directly and sends a notification to that process in order to notify that data writing has accomplished. The remote process does not know that another process writes something into its memory and will not wait for when data writing ends, until it receives a notification from its neighbor. The remote process checks for locally posted notifications to get the information about changes related to a segment. Once a notification arrives, the process starts to work with data related to that particular notification.

In addition, the MPI reduction operations were replaced with the GASPI communication in the linear solvers (CG and GMRes) to calculate the inner products and the norm of vectors located on different processes.

4 The Ludwig Application

Ludwig [3] is a versatile code for the simulation of Lattice-Boltzmann models in 3D on cubic lattices. Some of the problems that could be simulated with Ludwig include detergency, mesophase formation in amphiphiles, colloidal suspensions, and liquid crystal flows. Broadly, the code is intended for complex fluid problems at low Reynolds numbers, so there is no consideration of turbulence, high Mach number flows, high density ratio flows, and so on. Ludwig uses an efficient domain decomposition algorithm, which employs the Lattice-Boltzmann method to iterate the solution on each subdomain. The domain decomposition is carried out by splitting a three dimensional lattice into smaller lattices on subdomains and exchanging information with adjacent subdomains [4]. For each iteration, Ludwig uses MPI for communications with adjacent subdomains using *halo exchange* [2].

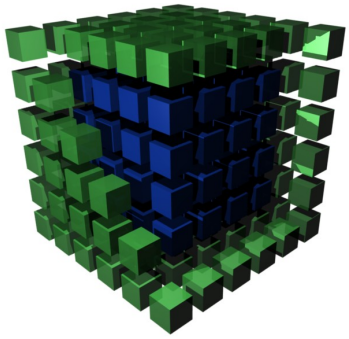


Fig. 2: Lattice subdomain where the internal section represents the real lattice and the external region the halo sites.

In the original implementation of the Ludwig halo exchange, the number of messages sent and received by each MPI process is reduced as much as possible. Each subdomain needs to exchange data with its 26 neighbors in three directions to continue with the solution of the problem. This means that synchronization between the different planes is required. To coordinate the solution, communication between adjacent subdomains is required after each iteration. This is done by creating halos around the dimensions of the subdomain, i.e. extending the dimension of the subdomain by one lattice point in each direction as depicted in Figure 2. After each time step, MPI processes will have to communicate a 2D plane of m velocities to their adjacent MPI processes. Since each plane shares some sites with the other planes, the exchange of information in each direction should be synchronized before continuing with the execution.

GASPI promotes the use of one-sided communication, where the initiator has all the relevant information for performing the data movement. This idea decouples the data movement from the synchronization between processes and it is especially relevant in applications that rely on continuous halo communications between neighbors. We aim at reducing the synchronization between subdomains by porting Ludwig's main halo exchange routines form MPI to GASPI.

Implementation Highlights The halo exchange routine responsible for exchanging data between neighbor subdomains uses non-blocking MPI and MPI derived datatypes. MPI derived datatypes allow us to specify non-contiguous data in a convenient manner and yet treat it as if it was contiguous.

GASPI requires the creation and later on use of what is known as GASPI segments. A GASPI segment is window of memory allocated to be used with the

GASPI model. In our case we have created one GASPI segment per plane and direction. Therefore, since we have three planes and two directions per plane, we will require six different GASPI segments. This number of GASPI segments is sufficient for each subdomain to communicate its faces with its immediate neighbors in the 3D space. The size of the segments is defined as twice the size of buffer to be sent since we will use the same segment to send and receive data from neighbor subdomains.

Listing 1.1: GASPI pointers to GASPI segments in the YZ plane.

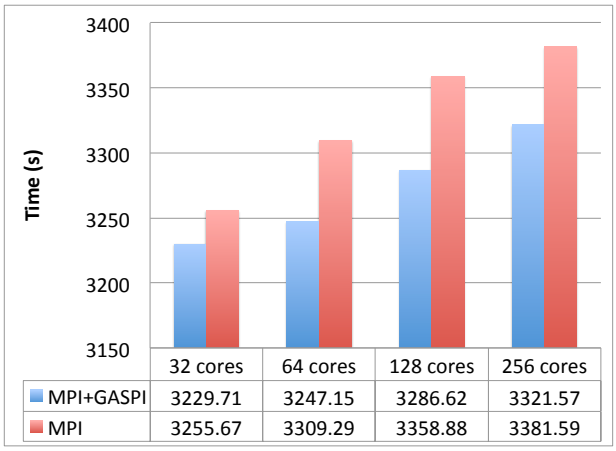
```
int YZ_size = lb->ndist*NVEL*ny*nz;

/* Segment size is exactly twice the size of the buffer.*/
const gaspi_size_t seg_size = 2 * YZ_size * sizeof(double);

 /* segment ids */
const gaspi_segment_id_t seg_id_YZ_L = 0;
const gaspi_segment_id_t seg_id_YZ_R = 1;
gaspi_pointer_t gptr_YZ_L, gptr_YZ_R;

/* pointer to the right */
GASPIERROR(gaspi_segment_ptr(seg_id_YZ_L, &gptr_YZ_L));
double* ptr_YZ_L = (double*)gptr_YZ_L;

/* pointer to the left */
GASPIERROR(gaspi_segment_ptr(seg_id_YZ_R, &gptr_YZ_R));
double* ptr_YZ_R = (double*)gptr_YZ_R;
```

For purposes of clarity, Listing 1.1 shows the GASPI pointer creation only in the YZ plane. For instance, in the YZ plane, each created segment is assigned with an independent id number. Hence, the data is already contiguous in memory and, therefore, a simple copy directly from the buffer that contains the data to a GASPI segment is straightforward. However, since Ludwig uses MPI datatypes, more complicated layouts of the data exist for other planes and it is necessary to unpack the MPI datatypes and copy the data contiguously into a GASPI segment. Once the data has been sent and notified we need to recover the data back from the GASPI segment to the original buffer to be able to continue with the normal execution of Ludwig.

5 Performance Results

iPIC3D We performed our tests on the Beskow supercomputer (Cray XC40) equipped with two 16-core @ 2.3 GHz Intel Haswell-EP processors. To compare the original version of the iPIC3D code with the new, GASPI-based, version, we used a standard simulation cases called Geospace Environment Modeling (GEM) Reconnection Challenge that is adapted to the Earth's magnetotail reconnection [1,6]. In addition, we used two different simulation cases, namely field- and particle-dominated, with a fixed number of iterations (20) in the field solver.

Figure 3 shows the results of the weak scaling tests for one of the iPIC3D simulations. Three-dimensional decomposition of MPI processes on X-, Y- and Z-axes was used, resulting in different topologies of MPI processes. For this particle



	32 cores	64 cores	128 cores	256 cores
MPI+GASPI	3229.71	3247.15	3286.62	3321.57
MPI	3255.67	3309.29	3358.88	3381.59

Fig. 3: Weak scaling results for the GEM 3D simulation of the particle-mover dominated regime of iPIC3D on Beskow.

dominated Magnetosphere 3D simulation on 64 cores (4x4x4 MPI processes x 4 OpenMP threads), 27x106 particles and 30x30x30 cells were used, and the simulation size increased proportionally to the number of processes.

For this simulation test case, the new version, based on GASPI, is slightly faster (by 1-2%) on different number of cores. The challenge of a successful porting of iPIC3D to GASPI depends on the optimal utilization of one-sided communication mechanism to achieve performance gain and scalability on pre-Exascale supercomputers. GASPI provides the one-sided communication that facilitates asynchronous procedures between processes. However, this requires the local processes to manage the communication in an optimized way to maximum the overlapping of communication and computation. The trade-off between asynchronicity and data synchronization requires further investigation.

Ludwig A set of performance tests were carried out on ARCHER, a Cray XC30 system equipped with two 12-core @ 2.7 GHz Intel Ivy Bridge processors. All simulations were executed five times on fully populated nodes, i.e. using 24 MPI/GASPI processes per node.

The time to transfer a message depends on the network latency and bandwidth. The latency is independent of the size of the message being sent, but dependent of the MPI implementation and network use. Figure 4 shows the measured bandwidth against the message size using Cray MPI. The bandwidth is low at very small message sizes because the time spent to send each message is dominated by the latency. As soon as the message size is increased over 0.2 MBytes, the bandwidth quickly rises to the maximum allowed by the fabric interconnect. We have also measured the amount of data required to be sent and received from each process at the end of each iteration in 192^3 lattice size, as represented in Figure 5.

Figure 6 shows the strong scaling results of running Ludwig on up to 3,072 processes on ARCHER. The total time that Ludwig spends on the main stepping loop is represented in Figure 6a, showing small difference in performance between the pure MPI version and the MPI+GASPI version of Ludwig; the performance

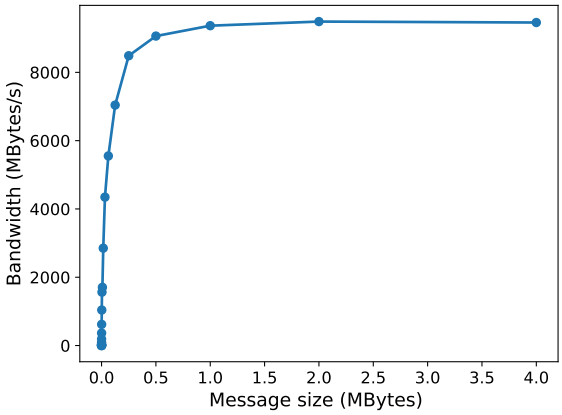


Fig. 4: Bandwidth and message size on ARCHER, using the OSU benchmarks [8].

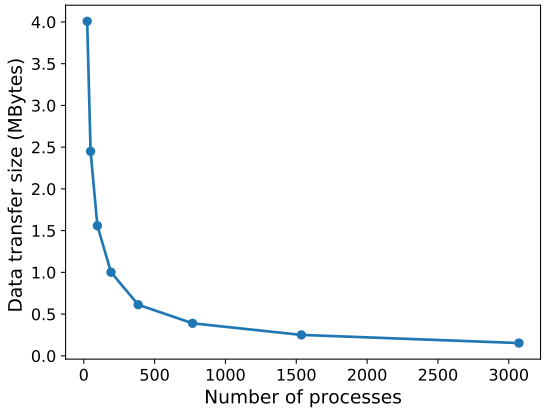
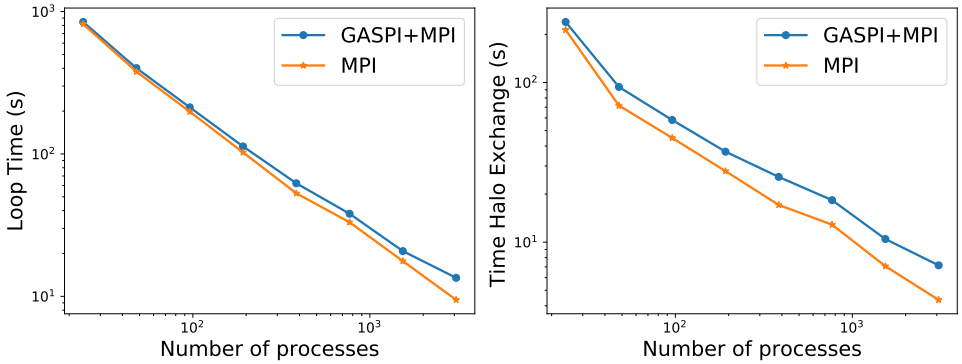


Fig. 5: Data transfer size by each process at the end of 192^3 lattice size simulation.



(a) Total loop time.

(b) Total halo exchange time.

Fig. 6: Strong scaling results of Ludwig for a 192^3 lattice size on ARCHER.

overhead is negligible with less than 1000 processes. When narrowing our focus to the halo exchange (see Figure 6b), which is one of the key components in

the main stepping loop, we can see that this performance penalty is low for a small processes count, but it grows as the number of processes is increased. This is probably due to the fact that the bandwith is at its best in that region as Figure 4 indicates Thus, there is a direct connection between the overhead in the halo exchange and the total loop. Nevertheless, given the performance benefits of one-sided communication in GASPI[6], we attribute this performance penalty to tedious process of unpacking and packing back and forth between the MPI datatypes and the GASPI segments.

Shared Window Communication in GASPI In order to validate this new programming paradigm of shared notifications in GASPI, we have implemented an equivalent to the MPI Allreduce for large messages. The implementation makes substantial use of pipelined rings. The algorithm consists of two stages. In the first stage, each of the N nodes performs a reduction of $1/N$ of the dataset (via the pipelined ring). In the second stage, the partial result from each node is broadcasted to the other nodes (again in the pipelined ring) such that after the broadcast all nodes have access to the complete reduced dataset.

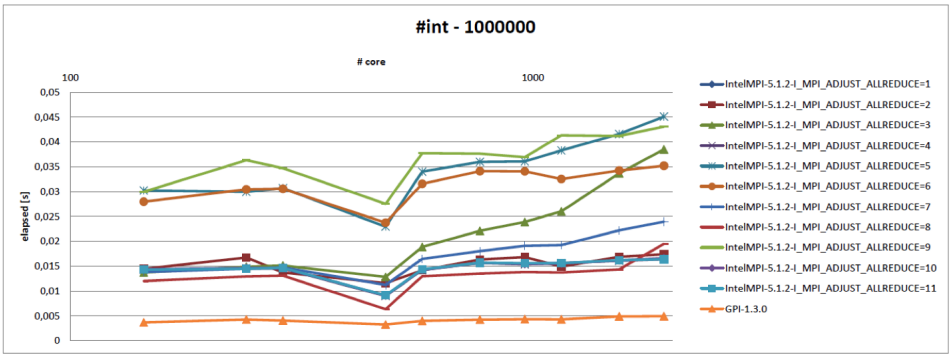


Fig. 7: Performance results of the pipelined ring implementation of Allreduce.

In order to split the reduction and communication loads across all processes, each of the N parts is again subdivided into at least M parts (where M is the number of processes per node) such that there are at least $N \times M$ messages in the ring at any point in time. The GASPI shared notification model allows any process to detect any of these $N \times M$ incoming asynchronous and one-sided notified messages, to reduce and forward them along the pipelined ring. Figure 7 shows a comparison of Allreduce implemented on top of GASPI shared windows against various Allreduce MPI low-level implementations in Intel MPI 5.1.2. Those are 1. Recursive doubling; 2. Rabenseifner's; 3. Reduce + Bcast; 4. Topology aware Reduce + Bcast; 5. Binomial gather + scatter; 6. Topology aware binominal gather + scatter; 7. Shumilin's ring; 8. Ring; 9. Knomial; 10. Topology aware SHM based flat; 11. Topology aware SHM based Knomial. Some of these implementations feature an optimal bandwidth term (Ring based or Rabenseifner's), however they are not able to leverage pipelining as efficiently

[6] `http://www.gpi-site.com/gpi2/benchmarks/`

as the high-level GASPI Implementation. The main problem here is that the underlying MPI point-to-point low-level frameworks (such as e.g. UCX) are not able to make efficient use of notified communication either.

6 Conclusions

The original versions of both iPIC3D and Ludwig – like many other MPI applications – use MPI datatypes. That soon became a problem while interoperating with GASPI since GASPI works on segments of data. This means that we had to unpack the data from the MPI datatypes, copy them to a GASPI segment, send them, and, then, unpack the data. We believe this packing-unpacking was the major burden for the applications' performance.

In order to improve the interoperability with a flat MPI programming model, GASPI has introduced a novel allocation policy for segments where data and GASPI notifications can be shared across multiple processes on a single node. To that end, any incoming one-sided GASPI notification will be visible node-locally across all node-local ranks. The shared notifications should be used with GASPI segments that are employing shared memory, such as MPI windows, provided by the applications under the interoperability mode.

We are currently developing a generic interface which can make use of these shared memory segments for the specific purpose of ghost cell exchanges. The developed interface will not only facilitate the interoperability of MPI plus GASPI significantly, but it will also substantially enrich the programming paradigm of MPI shared windows.

Acknowledgement This work was funded by EU H2020 Research and Innovation programme through the INTERTWinE project (no. 671602). The simulations were performed on resources provided by SNIC at PDC-HPC, KTH.

References

1. J. Birn and M. Hesse. Geospace environment modeling (GEM) magnetic reconnection challenge: Resistive tearing, anisotropic pressure and hall effects. *JGR: Space Physics*, 106(A3):3737–3750, 2001.
2. Er. Davidson. Message-passing for Lattice Boltzmann. Master's thesis, EPCC, The University of Edinburgh, Scotland, UK, 2008.
3. JC. Desplat, I. Pagonabarraga, and P. Bladon. Ludwig: A parallel Lattice-Boltzmann code for complex fluids. *Comp. Physics Comms.*, 134(3):273–290, 2001.
4. A. Gray, A. Hart, O. Henrich, and K. Stratford. Scaling soft matter physics to thousands of graphic processing units in parallel. *IJHPCA*, 29(3):274–283, 2015.
5. R. Machado, T. Rotaru, M. Rahn, and V. Bartsch. Guide to porting MPI applications to GPI-2. Technical report, Fraunhofer ITWM, 2015.
6. S. Markidis, P. Henri, G. Lapenta, K. Rönnmark, M. Hamrin, Z. Meliania, and E. Laure. The Fluid-Kinetic Particle-in-Cell method for plasma simulations. *Journal of Computational Physics*, 271:415–429, 2014.
7. S. Markidis, G. Lapenta, and et al. Multi-scale simulations of plasma with iPIC3D. *Mathematics and Computers in Simulation*, 80(7):1509–1519, 2010.
8. MVAPICH. MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. `http://mvapich.cse.ohio-state.edu/benchmarks/`.
9. C. Simmendinger, M. Rahn, and D. Grünewald. The GASPI API: A failure tolerant PGAS API for asynchronous dataflow on heterogeneous architectures. In *Sustained Simulation Performance 2014*, pages 17–32. Springer, 2015.