Interoperability Strategies for GASPI and MPI in Large Scale Scientific Applications

Journal Title
XX(X):1–12

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

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

Abstract

One of the main hurdles of PGAS approaches is the dominance 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 consider an interoperable GASPI/MPI implementation for the communication/performance crucial parts of the Ludwig and iPIC3D applications. To address the discovered performance limitations, we develop a novel strategy for significantly improved performance and interoperability between both APIs by leveraging GASPI shared windows and shared notifications. First results with a corresponding implementation in the MiniGhost proxy application demonstrate the viability of this approach.

Keywords

Interoperability, GASPI, MPI, iPIC3D, Ludwig, MiniGhost, 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)* Simmendinger et al. (2015), with *GPI-2** 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 remote side. The notification semantics is complemented with routines that wait for the update of a single or a set of notifications. In passing we note that similar weak synchronization primitives might also appear in the upcoming MPI-4 standard Belli and Hoefler (2015). 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 Machado et al. (2015) 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 Markidis et al. (2010) (see Section 4.1) – an implicit Particle-In-Cell code for space weather simulations; Ludwig Desplat et al. (2001) (see Section 4.2) – a large scale Lattice-Boltzmann code for complex fluids. We implement this classic strategy for coupling MPI and GASPI on the communication-intensive halo exchange parts in both applications. The performance results reveal some performance issues due to the packing/unpacking of data

[1]T-Systems Solutions for Research, Germany,
[2]KTH Royal Institute of Technology, Sweden
[2]EPCC, The University of Edinburgh, UK
[3]Fraunhofer ITWM, Germany

Corresponding author:
Roman Iakymchuk, KTH Royal Institute of Technology, Lindstedtsvägen 5, 100 44 Stockholm, Sweden
Email: riakymch@kth.se

*www.github.com/cc-hpc-itwm/GPI-2

from MPI derived datatypes to GASPI segments, clearly indicating a need to revise our strategy.

Thus, in order to tweak the GASPI implementation and programming model, both in terms-of-use and achievable performance we have implemented a strategy for enhancing the interoperability between MPI and GASPI using so-called shared notifications (see Section 3) in the MiniGhost Proxy application from Sandia Barrett et al. (2011). We demonstrate the corresponding transition from a flat MPI model towards a mixed-mode model which interoperates with MPI and makes efficient use of the GASPI communication primitives.

We provide further evidence that this strategy is beneficial (see Section 5.4) by implementing a collective Allreduce operation on top of this model. We compare against existing state-of-the-art for Allreduce operations.

This article is structured as follows. Section 2 outlines the classic way of coupling GASPI and MPI, while Section 3 introduces a novel strategy for better interoperability of the two APIs. We describes three case studies in Section 4 and present their performance results in Section 5. Finally, Section 6 draws conclusions and outlines future work.

2 State-of-the-art interoperability of GASPI and MPI

Writing parallel programs that are mixing MPI and GPI-2 communication sections is currently possible due to the ability of GPI-2 to capture the environment of an existing MPI running instance. For this purpose, GPI-2 must be installed with a specific option pointing to the current MPI installation used by the user. When running in mixed-mode, GPI-2 is able to detect at runtime the MPI environment and to setup its own environment based on this. Thus, in mixed-mode GPI-2 is able to deliver the same information about the ranks and the number of processes as MPI. In order to be able to do this, MPI must be initialized before GPI-2.

Porting a MPI-based application to GPI-2 can be done incrementally, by identifying independent MPI communication sections and replacing them gradually with GPI-2 communication sections, taking thus advantage of the ability of GPI-2 to work in mixed-mode with MPI Markidis et al. (2016). An important rule to follow here is to preserve the application's logic. Another aspect to take into account is not to overlap MPI with GPI-2 communication sections. It is also possible to encapsulate the GPI-2 code in a library and call this from from a MPI program.

The GASPI standard offers the possibility to allow a user to provide already allocated memory for the GASPI segments. For instance, a buffer used in a MPI collective operation can be reused as memory allocated for a segment. Alternatively, memory allocated within a GASPI segment can be used for the MPI communication. It is important to note that when the memory for the segments is allocated by the user, it is the user's responsibility to free it after the segment deletion.

Compared to MPI, which allows a flat model with several MPI processes in a shared memory region, threads are the recommended way to handle parallelism with GASPI within nodes. The interoperability between flat MPI and hybrid GASPI mode has been recently improved. The new shared notification feature allows a smoother interoperability between the flat MPI code with shared windows and GASPI. In GASPI, notifications provide a mechanism for weak synchronization of write processes to a memory segment. Typically, the memory segment belongs to one GASPI process local to the node. With shared notifications a GASPI memory segment can be shared between several GASPI processes.

3 A Strategy for Better Interoperability between GASPI and MPI via Shared Window Communication

Traditionally the GASPI programming model targets multi-threaded or task-based applications. In order to support a migration of legacy applications (with a flat MPI communication model) towards GASPI, we have extended the concept of shared MPI windows Rivas-Gomez et al. (2017) towards a notified communication model in which the processes sharing a common window become able to see all one-sided and notified communication targeted at this window.

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 with the help of shared notifications. 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. Shared notifications hence can mitigate the effect of MPI derived data types for applications which makes use of MPI and GASPI interoperability in a flat MPI communication model.

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. Therefore, this generic interface will facilitate the interoperability of MPI and GASPI significantly.

Below we discuss in details the main concepts of this generic interface. In particular, we outline shared

notifications in GASPI and GASPI shared windows, as well as present three strategies for boundary exchanges.

3.1 Notified communication

Notified communication is the primary means of communication in GASPI. In GASPI, data may be read/written to local (read) or remote (write) memory asynchronously, accompanied by a corresponding notification. Data then can be processed locally (read), remotely (write) whenever the corresponding notification has been flagged to the local (read) or remote (write) memory. In summary, GASPI notified communications target local and remote completion for one-sided communication.

3.2 GASPI Shared Windows

A GASPI Shared Window extends the concept of System-V shared memory (also used in MPI shared windows) towards notified communication in shared memory. In the System-V model, shared memory is allocated per process but is readable and writable for all local processes participating in the corresponding procedure call. The allocated memory always is restricted to a common shared memory region, but allows for a moderate amount of flexibility in configuration in order to support, for example, NUMA architectures.

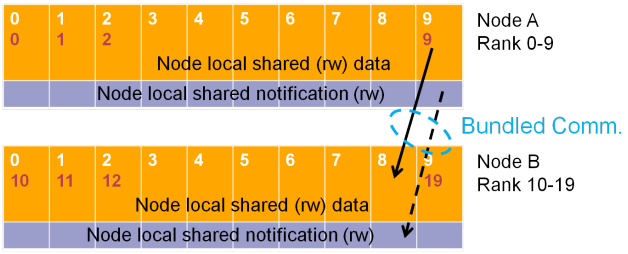


Figure 1. Shared notifications in GASPI.

In GASPI, Shared Windows – the System-V concept of shared application memory – is complemented with a shared GASPI notification space, see Figure 1. Every local process/rank with access to the common shared memory then becomes able to read/write the common notification space. This implies that every local process can see all incoming messages for all other local ranks with access to the shared application window.

3.3 Boundary Exchanges in Shared Memory

Boundary exchanges (or Ghost Cell Exchanges or halo exchanges) are presumably the most prevalent communication pattern in Computer Science, where neighboring elements or particles are exchanged between adjacent processing units. In order to keep communication as simple as possible, most MPI-Only applications will use the same communication pattern for all exchanges, irrespective of whether or not the neighboring elements actually share the same memory. For weak scaling, where communication requires a small fraction of total runtime, this approach is certainly viable. For strong scaling scenarios, however, an explicit use of shared memory can offer substantial benefits.

The following three figures (Figures 2 to 4) demonstrate a potential migration strategy. The first, Figure 2 depicts the current state-of-the-art: A linear communication buffer here is assembled (either explicitly in the application or MPI-internal via MPI data structures), the buffer then is transfered to the receiver with a subsequent unpack. This strategy is outlined in Section 2 for general purpose.

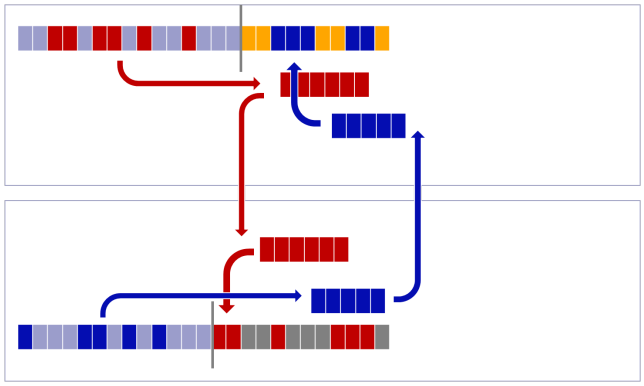


Figure 2. To and from the linear communication buffer - up to three copies are required.

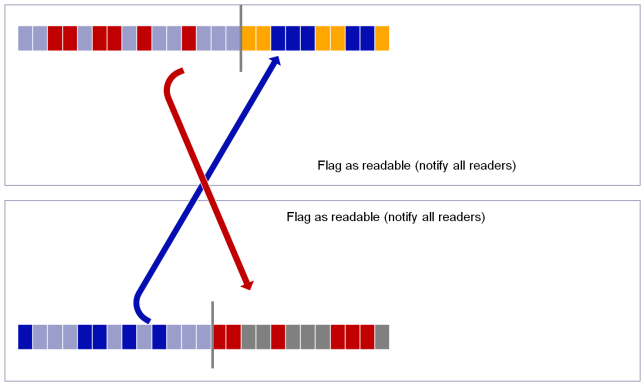


Figure 3. Semaphores in shared memory - flagging inner ghost cells as readable.

The second, Figure 3 demonstrates a possible optimization: Rather than packing/unpacking and sending we here merely flag the inner ghost cells as readable by means of a semaphore. Neighboring processing elements here become able to directly update their process/rank - local outer ghost cells. We note that a substantial part of this optimization step, in principle, might be carried out by MPI internally, where MPI derived datatypes would be directly converted from source MPI datatype to target MPI datatype. We subsequently will refer to this strategy as '1-copy'.

The third, Figure 4 demonstrates a complete removal of outer ghost cells. As all ranks can directly access the solver data of their neighbors in a commonly shared window, the solver can avoid the ghost cell exchange entirely. Instead of converting inner ghost cells from neighboring ranks to local outer ghost cells, the solver here directly accesses inner ghost cells from neighboring ranks. We subsequently will refer to this strategy as '0-copy'.

3.4 Related Work

The idea for a zero-copy framework for ghost-cell exchanges has been discussed in Besnard et al. (2015). Similar to our work the authors strip away the node-local copying

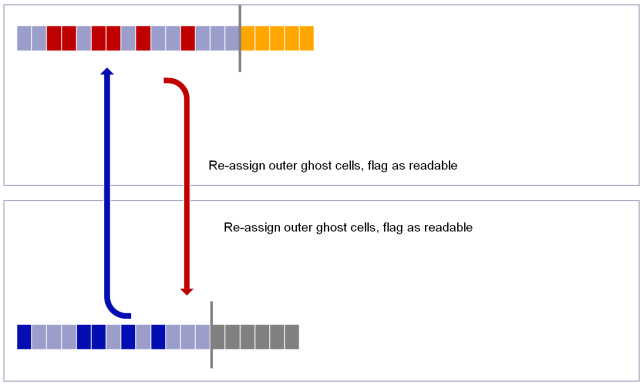


Figure 4. Sempahores in shared memory - reassigning outer ghost cell access

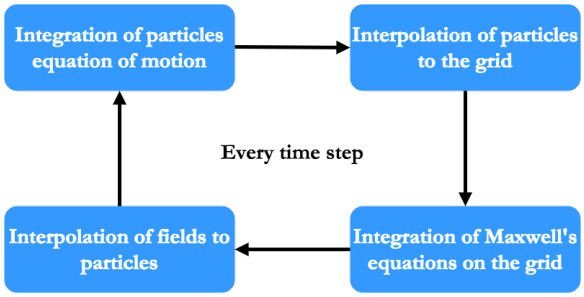


Figure 5. Structure of the iPIC3D code.

of ghost-cell data and instead directly access the inner ghost cells data structures of node-local (and neighboring) ranks. This implementation relies on a 'threads as processes' (MPC) MPI implementation, while we start from a more general approach: we directly use MPI shared memory across multiple processes and run the solver (wherever we subsequently need to be exchange corresponding data) in these shared windows. Our work also conceptually extends the work in Besnard et al. (2015), as we implement a model where communication (not just computation) is visible for all processes in the shared window. The implementation for the pipelined Allreduce, which is presented in Section 5.4, would not be feasible with the above implementation, as all processes require access to node local communication to optimally sustain the pipeline.

4 Case Studies

In this section, we introduce three test cases for our study. That comprises two large scale scientific applications, namely iPIC3D and Luwdig, and the MiniGhost mini-application.

4.1 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 Earths 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 5 depicts these computational steps in iPIC3D.

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, which is also a part of the field solver.

4.1.1 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's 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 Saad (2003) (Conjugate Gradient, CG, and Generalized Minimal Residual, GMRes) to calculate the inner products and the norm of vectors located on different processes.

4.2 The Ludwig Application

Ludwig Desplat et al. (2001) 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 Gray et al. (2015). For each iteration, Ludwig uses MPI for communications with adjacent subdomains using *halo exchange* Davidson (2008).

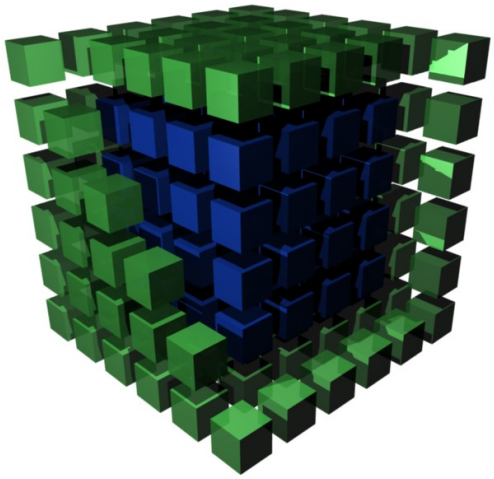


Figure 6. Lattice subdomain where the internal section represents the real lattice and the external region the halo sites.

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 6. 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.

4.2.1 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 a 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: 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 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.

4.3 Migration of MPI-Only Applications: MiniGhost

By combining the concept of shared GASPI windows with dependencies between local ranks, we can migrate flat MPI-Only applications towards an asynchronous communication model. The explicit local communication in shared memory here is replaced with data dependencies on shared memory semaphores and the read-access to local ranks. Communication across shared windows is replaced with a dataflow-oriented notified GASPI communication.

In order to demonstrate this approach we have ported the MiniGhost Barrett et al. (2011) mini-application from the MPI-Only model to the GASPI Shared Windows concept. MiniGhost is a Finite Difference code which implements a difference stencil across a homogeneous three dimensional domain. The application extracts the communication pattern of CTH (Shock Physics, Sandia Labs McGlaun et al. (1990)), a multi-material, large deformation, strong shock wave, solid mechanics code and has been developed as a tool to explore boundary exchange strategies in stencil applications. The MiniGhost proxy supports several options for stencil computations and two main strategies for boundary exchange: A bulk synchronous parallel with message aggregation for multiple boundary exchanges (BSPMA) and a single variable aggregated faces (SVAF) exchange. We here focus on the SVAF model. The MPI-Only SVAF model is implemented in the form of a classical MPI ghost cell exchange, where the posting of `MPI_Irecv` is followed by packing the communication data into a linear communication buffer with a subsequent call to `MPI_Isend`. In passing we note, that in our work we have used a minor optimization relative to the original MiniGhost version: Instead of packing all required data and subsequently sending, we pack and send per communication buffer, which maximizes the potential to overlap the packing procedure with sending data. In order to test for incoming messages, a `MPI_Waitany` function is called, which allows for the moderate overlap of communication and computation: The overlap of packing and unpacking of sent/incoming linear communication buffers into/from the local data structures. Once the entire ghost cell exchange is complete the actual stencil computations of MiniGhost are being started.

4.3.1 MiniGhost Implementations In order to further explore the possible boundary exchange strategies of MiniGhost beyond SVAF, we are comparing seven different versions: The original MPI-Only SVAF version, two versions which make use of MPI shared windows, two different version which make use of GASPI shared windows (where all four versions are '1-copy') and also two '0-copy' versions which avoid the shared memory communication entirely.

The shared MPI implementation (MPI SHARED 1-COPY) directly implements '1-copy' for MiniGhost. Within the node data solver data is allocated as a shared segment. From an implementation point of view we require the replacement of Fortran 'allocate' and associated data structures with a Fortran pointer variable, where the (page-aligned) value is returned from the call to `MPI_Win_allocate_shared`. We also require node local information about the respective pointer offsets for the neighboring ranks in the commonly shared MPI window. We require information about local and non-local ranks and a mapping from global to local ranks. We also require a rank-local semaphore in shared memory in order to be able to satisfy the data dependencies between neighbors. To that end, we are using an atomic counter (a gcc intrinsic with an implied memory fence) such that a node-locally visible increment of the semaphore will guarantee the validity of accessed data. The data we here access are the inner ghost cells of local neighbor ranks, which are required for the boundary exchange. Additionally, we enforce a node-local shared memory barrier at the end of each iteration, such that we can make sure that all inner-ghost cell data has been read by the neighboring ranks.

In order to test the hypothesis of insufficient progress in MPI message delivery, we have replaced the non-blocking MPI send call with a blocking MPI send version (MPI SHARED 1-COPY BLOCKING), such that we pack and directly wait for completed communication in order to minimize potential latency overhead due to the insufficient progress in the MPI stack.

The GASPI SHARED 1-COPY version inherits most of the implementation from the MPI SHARED 1-COPY version. We here have replaced 2-sided MPI communication with the notified communication from GASPI. Instead of `MPI_send` we are using 1-sided `gaspi_write_notify` and instead of `MPI_Waitany` we use `gaspi_notify_waitsome`. We are using page-aligned memory for solver and communication data and we are using the concept of GASPI shared windows, where notifications are visible across all ranks. The reason for page-alignment is performance – write access to memory pages will look the entire page for other ranks. We hence strictly keep all solver and communication data (including notifications) local to the processes and page-aligned. While in this model the processes may read from neighboring ranks, writing should only be occur to process local memory (as returned a pointer value in the call to `MPI_Win_allocate_shared`).

The GASPI SHARED 1-COPY NO BARRIER implementation removes the above shared memory a barrier and

introduces a second semaphore. Shared memory synchronization then no longer is required once per iteration, but is replaced with mutual data dependencies for 'read' and 'have read'.

The MPI and GASPI 0-COPY implementations follow (within shared memory) the '0-copy' concept and directly update their inner ghost cell without a corresponding ghost cell exchange. The data dependencies however obviously still need to be observed.

We note that the potential for overlapping communication and computation in MiniGhost is very small and restricted to the packing/unpacking process. There is substantial room for improvements: With, for example, communication progressing while the actual stencil calculation is being performed, we likely would see even better strong scaling properties than the results we show in section 5.3.

5 Performance Results

In this section, we present the performance results and findings for the two large scale applications (iPIC3D and Ludwig) that implement the linear communication buffer strategy. Then, we analyze in details the MiniGhost mini-application that implements three different strategies for migration in boundary exchanges.

5.1 iPIC3D Results

We perform our tests on the Beskow supercomputer (Cray XC40) equipped with two 16-core @ 2.3 GHz Intel Haswell-EP processors. To illustrate the scalability of the original version and the GASPI-based version of the iPIC3D code, we use two standard iPIC3D simulation cases Birn and Hesse (2001); Markidis et al. (2014):

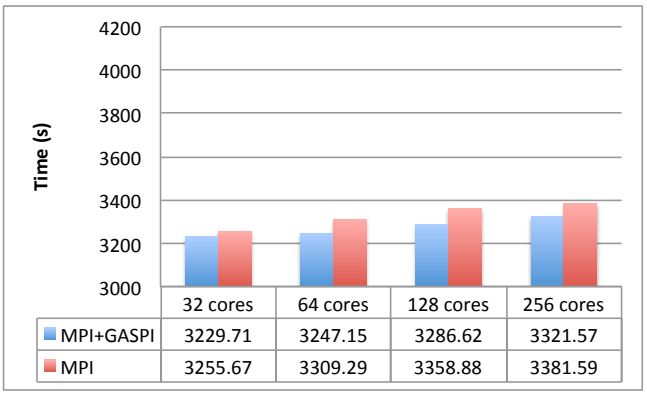
- GEM3D: a 3D magnetotail reconnection simulation with respect to the Geospace Environment Modeling (GEM) Reconnection Challenge;
- Magnetosphere3D: a 3D Earth radiation belts simulation.

In addition to these simulation cases, we use two different input data sizes (regimes) with a fixed number of iterations (20) in the field solver:

- *field-solver dominated regime*: with a relatively small number of particles (27 per cell), so that the most computationally expensive part of the iPIC3D code results in the Maxwell field solver;
- *particle-mover dominated regime*: it is characterized by a large number of particles (1,000 per cell), which, therefore, stresses the particle mover.

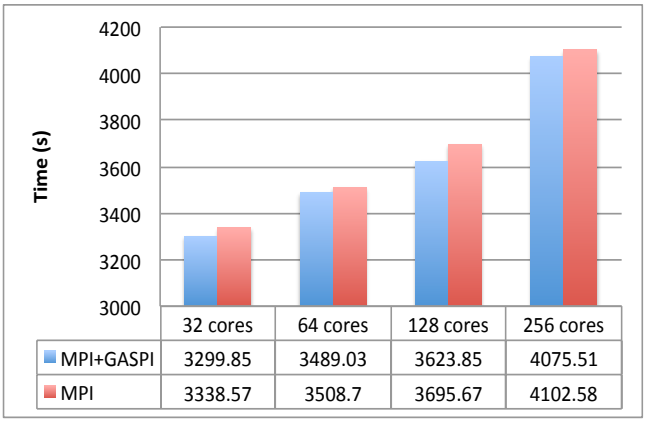
Hence, there are four different test cases (two simulation types with two regimes). Here, we present only two of them as the other two show similar pattern.

Figure 7 and Figure 8 show the results of the weak scaling tests for iPIC3D simulations. The three-dimensional decomposition of MPI processes on X-, Y- and Z-axes was used, resulting in different topologies of MPI processes. For the particle-dominated GEM 3D simulation on 64 cores (4x4x4 MPI processes), 27x10^6 particles and 30x30x30 cells were used, and the simulation size was increased proportionally to the number of processes.



	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

Figure 7. Weak scaling results for the GEM 3D simulation of the particle-mover dominated regime of iPIC3D on Beskow.



	32 cores	64 cores	128 cores	256 cores
MPI+GASPI	3299.85	3489.03	3623.85	4075.51
MPI	3338.57	3508.7	3695.67	4102.58

Figure 8. Weak scaling results for the Magnetosphere 3D simulation of the field-solver dominated regime of iPIC3D on Beskow.

For both plots, we can observe that 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.

5.2 Ludwig Results

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 taken to transfer a message in the Ludwig application is defined as $T = t_0 + B \times m$, where t_0 is the startup time (μs) defined by the latency of the network, B is the bandwidth (MBytes/s), and m is the size of the message (MBytes). Thus, 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 9 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

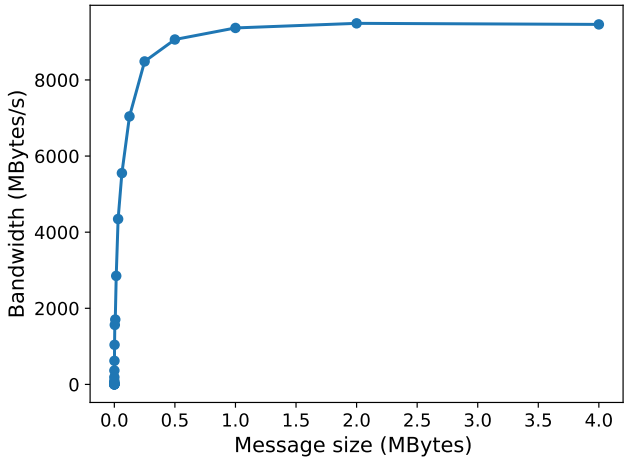


Figure 9. Bandwidth and message size on ARCHER, using the OSU benchmarks MVAPICH (2018).

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 10. As it is possible to see, strong scaling becomes a challenge as the number of nodes increases due to the reduction in size of the messages sent indicating how relevant latency is in this kind of communications.

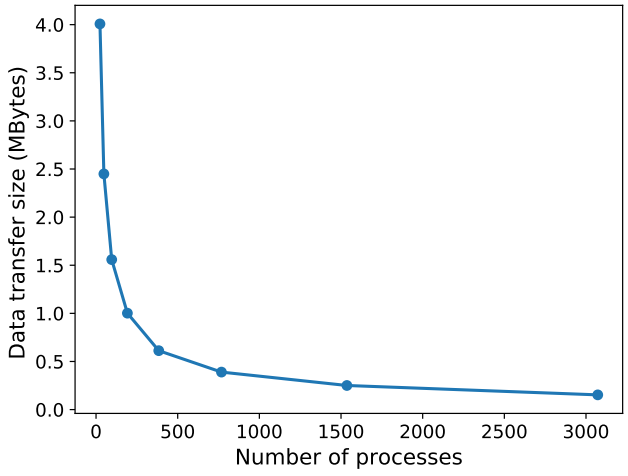
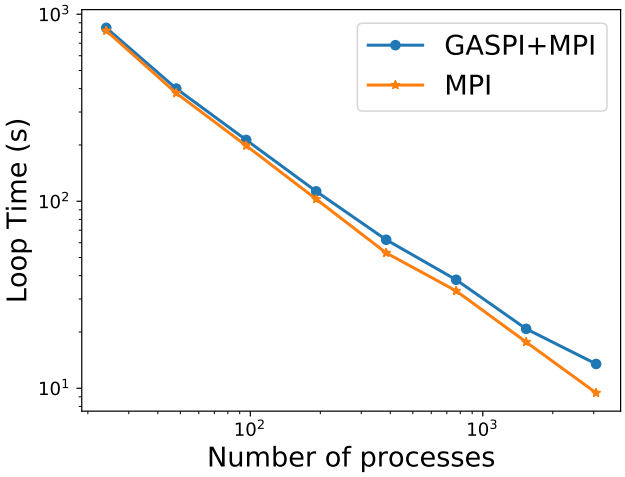
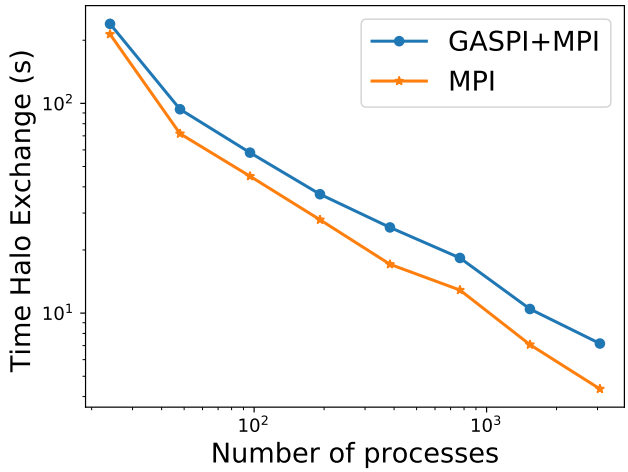


Figure 10. Data transfer size by each process at the end of 192^3 lattice size simulation.

Figure 11 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 11a, showing small difference in performance between the pure MPI version and the MPI+GASPI version of Ludwig; the performance overhead is negligible with less than 1,000 processes. When narrowing



(a) Total loop time.



(b) Total halo exchange time.

Figure 11. Strong scaling results of Ludwig for a 192^3 lattice size on ARCHER.

our focus to the halo exchange (see Figure 11b), which is one of the key components in the main stepping loop, we can see that this performance penalty is low for low processes count, but it grows as the number of processes increases. We believe the reason of this effect is because when the number of processes is low the message size is larger and as it could be seen in Figure 9, the measured bandwidth is maximum with package sizes over 5.5 MBytes. 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[†], we attribute this performance penalty to tedious process of unpacking and packing back and forth between the MPI datatypes and the GASPI segments.

5.3 MiniGhost Results

The following Figure 12 gives an overview over the obtained results for the MiniGhost proxy application. All tests were performed on an Infiniband FDR10 interconnect with Intel Sandy Bridge with 2x8 cores per node. Throughout we have

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

used an 8x8x8 processor grid (512 ranks), while varying the number of processed elements per rank; the number of elements per rank grows as N^3, where N is an even integer. We note that for this processor grid a flat MPI-Only model will require six explicit boundary exchanges for our chosen 3D stencil (exchange of east, west, north, south, top, and bottom neighbors). A shared model (on average) will require only three explicit exchanges, as all east-west communication can be handled within a socket and one half of the north-south communication can be handled between the two sockets.

We show the number of processed elements per rank (x-axis), divided by the corresponding total runtime. Essentially, Figure 12 hence plots the performance per node in an application specific metric. Figure 12 includes both communication and computation time. We observe that the total runtime behavior can be split into two different areas: a part which is dominated by available communication and computation bandwidth (the right side of the plot with the number of elements per rank equal or greater than 195,112) and a strong scaling regime which exhibits super-linear scaling due to cache effects both in communication and computation (the left side of the plot with the number of elements per rank smaller than 195,112).

For the bandwidth dominated part, we see a near constant runtime per element. In this part, we hence would expect a linear scaling behavior of the application. As the '0-copy' implementations also require less node internal computation effort (as we can avoid all intermediate copies) we can save a moderate amount of CPU time relative to the '1-copy' or flat MPI implementation of the order of three percents. Apart from this side effect, communication only plays a minor part in this regime: The computational effort here grows in third order with the number of elements per dimension (N^3) whereas the communication overhead only grows quadratically with a factor of $6 \times N^2$. For large numbers of elements, communication effectively drops out of our performance metric.

In the strong scaling regime, the left side of Figure 12, we see a more varied result. For strong scaling, the flat MPI model is limited by the additional intra-segment communication overhead, as we need to deliver twice the amount of messages per rank for the flat MPI-Only model. The shared memory MPI implementations (MPI SHARED 1-COPY) performs slightly better, however the relative performance gain is rather small: The message size in this strong scaling regime remains moderate and most communication data here resides in one of the caches. Performance in this regime appears as dominated by latency rather than bandwidth.

Given the fact that there is very few code changes relative to MPI SHARED 1-COPY, the GASPI SHARED 1-COPY version performs rather well. For small changes with the number of elements per rank smaller than 10,000 we see a substantial performance benefit in using the GASPI notified communication model. An extrapolation of the bandwidth dominated performance curve into this strong scaling regime exhibits that – in terms of absolute scaling numbers – this version would support a near linear speedup of around 2,744 elements per rank. Beyond this point to the left (with shrinking numbers of elements per rank) however the performance drops rapidly.

In order to get a more detailed insight into the performance figures we show both computation and communication contribution in separate plots. Figure 13 shows the computational performance in the strong scaling regime as the number of processed elements per computation wall-clock. Interestingly, here the MPI-Only implementation outperforms all other implementations. A possible explanation appears to be that memory management per process is superior to all other implementations which used shared windows: An MPI-Only memory management for solver data can be handled in a more flexible (and entirely process-local) manner, whereas memory for solver-data in shared memory is restricted to a page-aligned memory pointer as returned by `MPI_Win_allocate_shared`. We are currently investigating this issue in more details.

Figure 14 plots the 'achievable bandwidth', which in this case is defined as the communication volume $6 \times N^2$ divided by communication time. Strictly speaking the value of $6 \times N^2$ only is applicable to flat MPI-Only. A '0-copy' implementation for example only moves half of the data (in our example of six neighbors in an $8 \times 8 \times 8$ processor cube). For comparison, however, we have fixed the communication volumes to this (equal) theoretical value.

While flat MPI-Only easily wins in scalar performance, the communication performance of flat MPI-Only is only moderate – and actually for the very same reasons. The loose coupling between processes and non-existent use of shared memory now becomes a bottleneck. Very interesting here is the performance of MPI SHARED. As the '0-copy' needs to deliver half of the communication as the flat MPI-Only version, the '0-copy' implementation of MPI SHARED in consequence delivers almost twice the performance. The '1-copy' implementation of MPI SHARED however delivers near identical performance to flat MPI-Only – apparently the single copy overhead equals the node-internal MPI overhead for the considered messages sizes.

While the GASPI SHARED implementations are inferior in scalar computation (as they use the identical shared memory for solver data as MPI SHARED) they are superior in communication. Especially, remarkable is the GASPI SHARED 1-COPY implementation which – up to around 5,000 elements per rank – apparently manages to hide the entire intra-window communication (including pack/unpack) within the inter-node communication. For a value of around 8,664 elements per rank we then reach the peak bandwidth of the network adapter. All implementations here are equal, except for the '0-copy' implementations, which almost by definition are a factor of two faster than the rest. Beyond this point, the MPI implementations need to split available memory and network bandwidth between the implementation and MPI itself. GASPI here with its implementation of 1-sided communication on top of RDMA performs slightly better.

In these experiments, we have used a rather small stencil with six boundary exchanges. We also have used a system with a very moderate number of cores. Still, we were able to save a factor of two in terms of the requirements for network bandwidth and communication for the '0-copy' implementation. In times of ever increasing core counts per

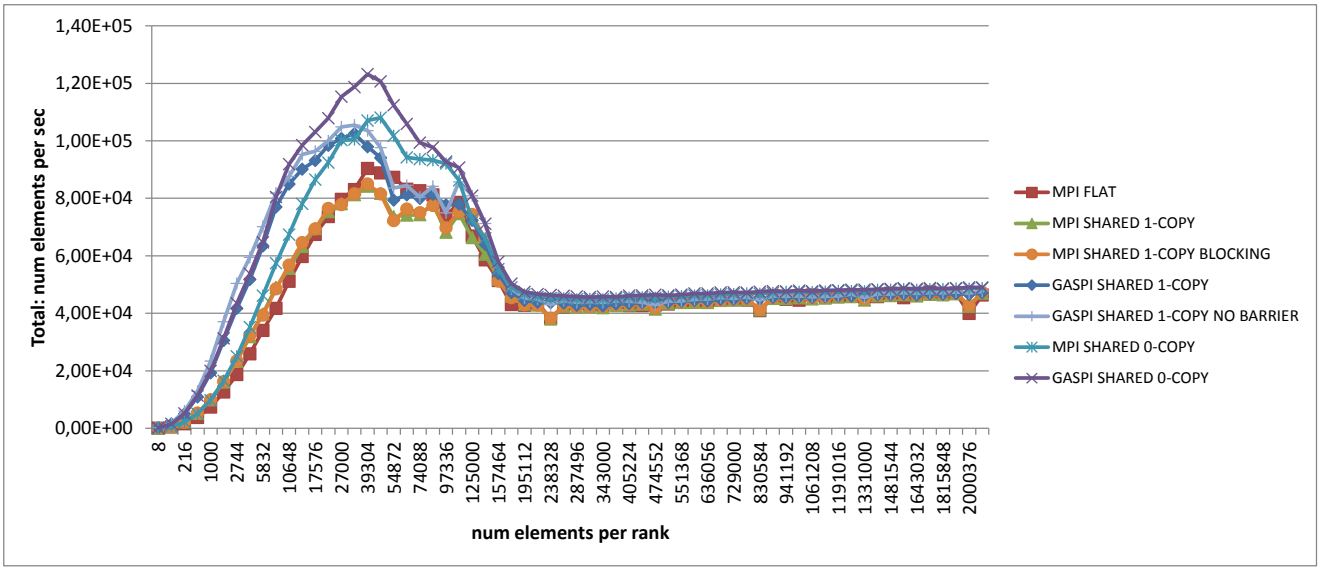


Figure 12. Performance MiniGhost - Full application.

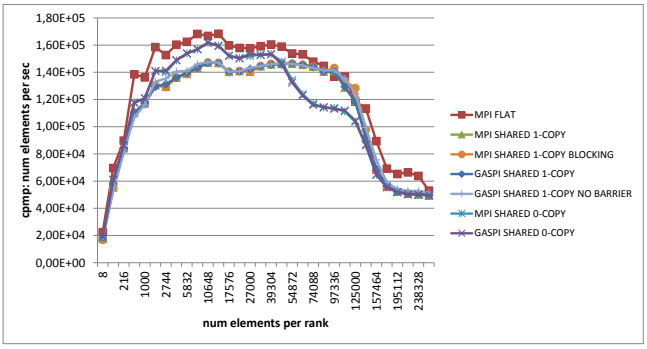


Figure 13. Performance MiniGhost - Computation - Strong scaling domain.

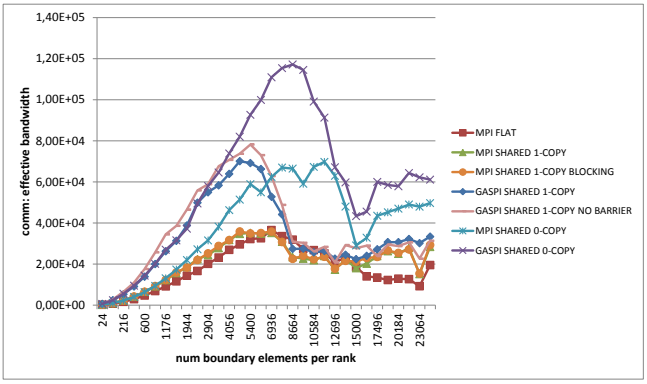


Figure 14. Performance MiniGhost - Communication - Strong scaling domain.

node, it is quite interesting to look at larger stencils and larger core counts.

Let us consider an example: We assume we perform a full 3 D stencil with 26 neighbors and we assume we have a processor mesh with 512 cores ($8 \times 8 \times 8$) with access to a common shared memory. The surface of this processor mesh then would amount to 296 cores. The ratio of outgoing communication (inter-node) to internal (intra-node) communication would then drop from 1-1 to less than 1/4th.

A migration strategy from MPI-Only to a model, which makes explicit use of shared memory (ideally as a '0-copy' implementation), hence would deliver increasing returns for an increased core count in future exascale systems – even more so if we require a large number of communication partners per rank.

5.4 Allreduce Communication in GASPI Shared Windows

In order to validate this new programming paradigm of shared notifications in GASPI, we also 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, see Figure 15). At the end of this stage, each node then contains a complete result of $1/N$ of the data. In the second stage, the partial result from each node is broadcasted to the other nodes (again in the pipelined ring, see Figure 16) such that after the broadcast all nodes have access to the complete reduced dataset.

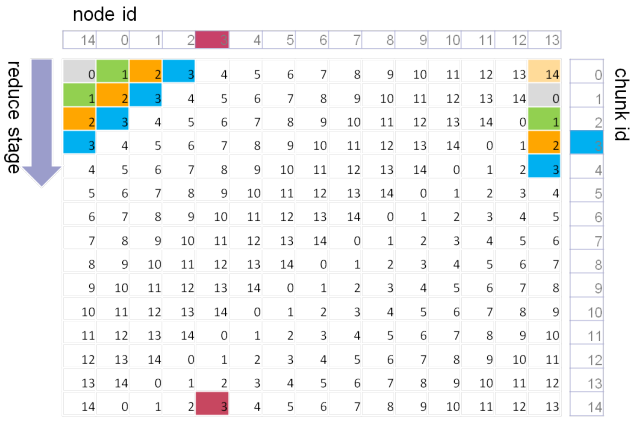


Figure 15. Allreduce - pipelined ring: reduce stage for chunk id 3 for 15 nodes. After 14 steps all nodes have a complete partial results.

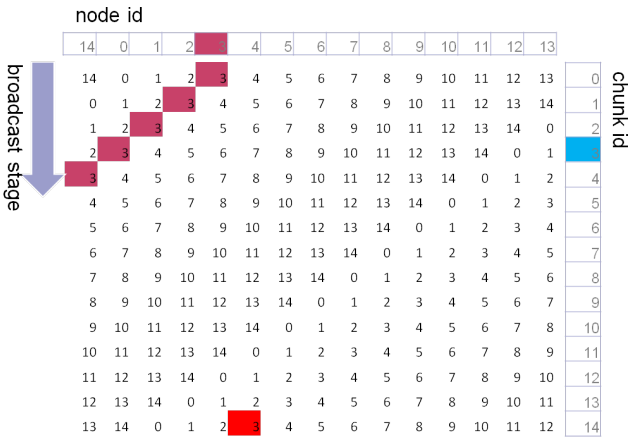


Figure 16. Allreduce - pipelined ring: broadcast stage. Again we use a pipelined ring to broadcast the partial results to the other nodes.

While the above algorithm allows for pipelining the set of N chunks across all nodes, we can still improve this result: In order to split the reduction and communication loads across all processes (and not just all nodes), 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. In doing so, the GASPI implementation manages to use the entire memory and network bandwidth of the system. For the reduction, we have used here a global sum. Thus, we can hide the complete reduction effort in the communication costs. As long as the reduction effort is less time-consuming than the corresponding communication, this will also hold true for more complex reductions.

Figure 17 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 as the high-level GASPI Implementation. The problem here is not restricted to MPI: We note that the underlying frameworks (such as e.g. UCX Shamis et al. (2015)) for MPI point-to-point communication are not able to make efficient use of massively parallel notified communication either.

6 Conclusions

In this work, we studied the interoperability of GASPI and MPI on large-scale applications, namely iPIC3D and Ludwig, implementing rather a classic way of combining both APIs. 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. As this overhead occurs in the critical path of the communication, packing and unpacking has a major impact for the applications' performance.

While in general packing cannot be avoided for inter-node communication, we find that the best solution for intra-node communication is to entirely replace this node-internal communication with a set of mutual data dependencies in shared memory. We directly read from the data structures of the local (intra-node) neighbors. This strategy is applicable for both programming models - MPI and GASPI. With an increasing number of cores per node (and depending on the number of required boundary (ghost-cell) exchanges per rank), this strategy will yield increasing returns for the implementation invest.

In order to complement this intra-node communication strategy with an inter-node dataflow 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. Any incoming one-sided GASPI notification will be hence be visible node-locally across all node-local ranks. Using this policy as well as relying upon the GASPI shared windows and GASPI shared notifications, we developed a generic interface which can make use of these shared memory segments for the specific purpose of ghost cell exchanges. The interface does not only facilitate the interoperability of MPI plus GASPI significantly, but it is also substantially enrich the programming paradigm of MPI shared windows. We conducted a set of successful experiments using the MiniGhost Proxy Application and obtained very convincing performance results.

With the clear gain of the new generic interface on an example of the MiniGhost mini-application, we are in progress of applying the same strategy in the halo exchange part of the iPIC3D application. On the way to adopt this generic interface in iPIC3D, we also aim to derive its proxy and a corresponding implementation which makes use of GASPI shared windows and shared notifications. In doing so, we will be able to adopt the developed improved boundary exchange pattern not only for particle-in-cell simulations but also for fields such as complex fluids.

Acknowledgements

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

Barrett RF, Vaughan CT and Heroux MA (2011) Minighost: a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing. *Sandia National Laboratories, Tech. Rep. SAND* 5294832.

Belli R and Hoefler T (2015) Notified access: Extending remote memory access programming models for producer-consumer synchronization. In: *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, pp. 871–881.

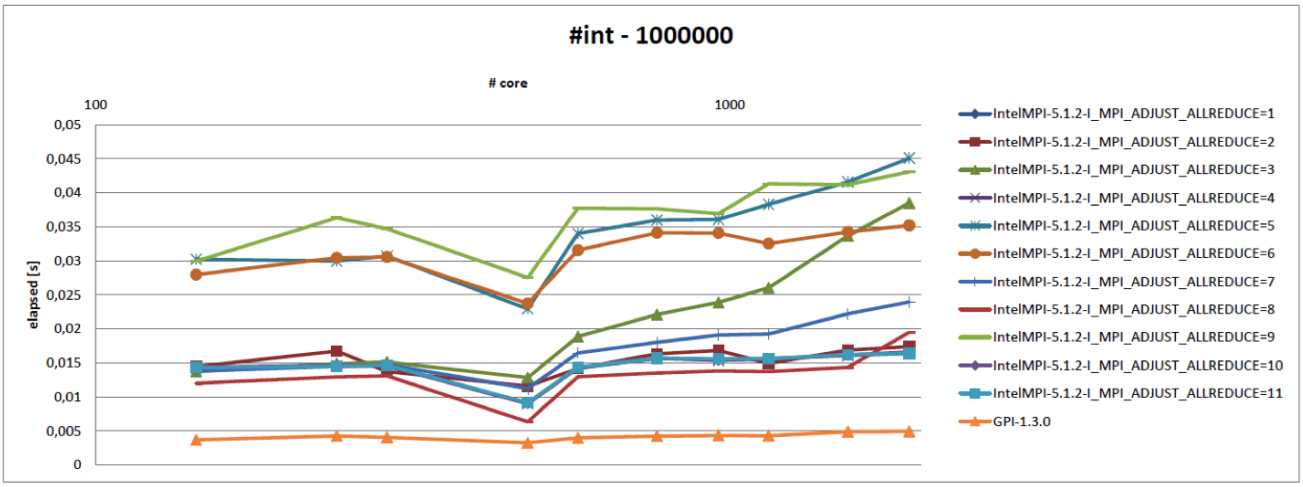


Figure 17. Performance results of the pipelined ring implementation of Allreduce.

Besnard JB, Malony A, Shende S, Pérache M, Carribault P and Jaeger J (2015) An mpi halo-cell implementation for zero-copy abstraction. In: *Proceedings of the 22Nd European MPI Users' Group Meeting*, EuroMPI '15. New York, NY, USA: ACM, pp. 3:1–3:9.

Birn J and Hesse M (2001) Geospace environment modeling (GEM) magnetic reconnection challenge: Resistive tearing, anisotropic pressure and hall effects. *JGR: Space Physics* 106(A3): 37373750.

Davidson E (2008) *Message-passing for Lattice Boltzmann*. Master's Thesis, EPCC, The University of Edinburgh, Scotland, UK.

Desplat J, Pagonabarraga I and Bladon P (2001) Ludwig: A parallel Lattice-Boltzmann code for complex fluids. *Comp. Physics Comms.* 134(3): 273–290.

Gray A, Hart A, Henrich O and Stratford K (2015) Scaling soft matter physics to thousands of graphic processing units in parallel. *IJHPCA* 29(3): 274–283.

Machado R, Rotaru T, Rahn M and Bartsch V (2015) Guide to porting MPI applications to GPI-2. Technical report, Fraunhofer ITWM.

Markidis S, Henri P, Lapenta G, Rönnmark K, Hamrin M, Meliania Z and Laure E (2014) The Fluid-Kinetic Particle-in-Cell method for plasma simulations. *Journal of Computational Physics* 271: 415–429.

Markidis S, Lapenta G and Rizwan-uddin (2010) Multi-scale simulations of plasma with iPIC3D. *Mathematics and Computers in Simulation* 80(7): 1509–1519.

Markidis S, Peng IB, Träff JL, Rougier A, Bartsch V, Machado R, Rahn M, Hart A, Holmes D, Bull M et al. (2016) The epigram project: Preparing parallel programming models for exascale. In: *International Conference on High Performance Computing*. Springer, pp. 56–68.

McGlaun JM, Thompson S and Elrick M (1990) Cth: a three-dimensional shock wave physics code. *International Journal of Impact Engineering* 10(1-4): 351–360.

MVAPICH (2018) MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. http://mvapich.cse.ohio-state.edu/benchmarks/.

Rivas-Gomez S, Gioiosa R, Peng IB, Kestor G, Narasimhamurthy S, Laure E and Markidis S (2017) Mpi windows on storage for hpc applications. In: *Proceedings of the 24th European MPI Users' Group Meeting*. ACM, p. 15.

Saad Y (2003) *Iterative Methods for Sparse Linear Systems*. 2nd edition. Society for Industrial and Applied Mathematics.

Shamis P, Venkata MG, Lopez MG, Baker MB, Hernandez O, Itigin Y, Dubman M, Shainer G, Graham RL, Liss L et al. (2015) Ucx: an open source framework for hpc network apis and beyond. In: *High-Performance Interconnects (HOTI), 2015 IEEE 23rd Annual Symposium on*. IEEE, pp. 40–43.

Simmendinger C, Rahn M and Grünewald D (2015) The GASPI API: A failure tolerant PGAS API for asynchronous dataflow on heterogeneous architectures. In: *Sustained Simulation Performance 2014*. Springer, pp. 17–32.