Interoperability of GASPI and MPI in a large scale Lattice-Boltzmann code

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

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

Keywords: Interoperability, GASPI, MPI, Ludwig, 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 already. 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)* [3], 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 the so-called 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. GASPI allows for a thread-safe handling

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

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 [2] this interoperability with MPI in a so-called mixed-mode, where the MPI and GASPI interfaces can be mixed. As a case study, we consider the Ludwig application [1] (see Sect. 2) – a large scale Lattice-Boltzmann code for complex fluids – for which we collect the preliminary performance results (see Sect. 3).

2 A Case Study: The Ludwig Application

Ludwig [1] is a versatile code for the simulation of Lattice-Boltzmann models in 3D on cubic lattices. 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. For each iteration, Ludwig uses MPI for communications with adjacent subdomains, using a technique commonly referred to as *halo exchange*. 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 3 directions (X, Y, Z) to continue with the solution of the problem. This means that synchronization between the different planes is required.

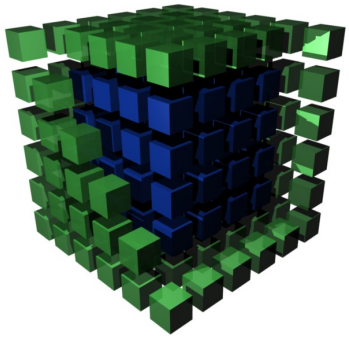


Fig. 1: Lattice subdomain where the internal section represents the real lattice and the external region the halo sites.

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 Fig. 1. 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. We aim at reducing the synchronization

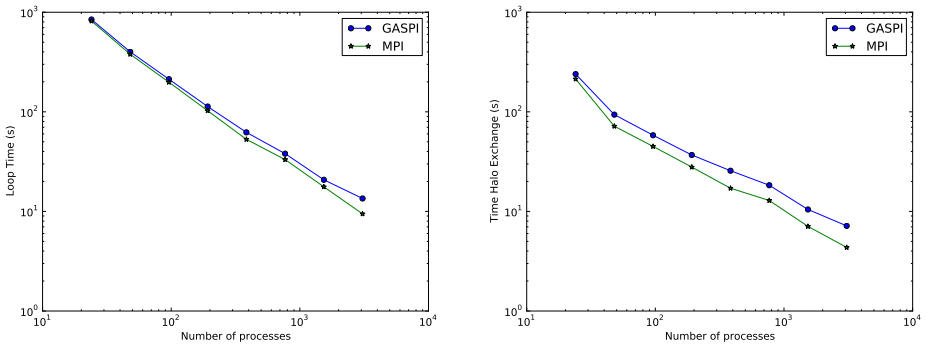
between subdomains by porting Ludwig's main halo exchange routines form MPI to GASPI.

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 the so-called GASPI segments. In our case we have created a GASPI segment per plane and direction. Therefore, since we have 3 planes and 2 directions per plane, we will require 6 different GASPI segments. 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. 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. To differentiate the data sent and received in a GASPI segment, we use an offset variable.

3 Performance Results

We carried out a set of performance tests on ARCHER, which is 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.



(a) Total loop time.

(b) Total halo exchange time.

Fig. 2: Strong scaling results of running Ludwig simulation for a 192^3 lattice size implemented with pure MPI and GASPI-MPI on ARCHER.

Fig. 2 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 has been represented in Fig. 2a, 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 1000 processes. When narrowing our focus to the halo exchange (see Fig. 2b), 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. 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.

4 Discussion

The original version of Ludwig like many other MPI applications, uses MPI datatypes. That soon became a problem for the porting process since GASPI works on segments of data. This means that we had to unpack the data used by MPI datatypes, copy the data required to a GASPI segment, send and then unpack the data. We believe this packing-unpacking was the major burden for Ludwig's performance. In order to improve the interoperability with a flat MPI programming model, GASPI will introduce a novel allocation policy for segments where data and GASPI notifications can be shared across multiple processes on a single node. To that end GASPI will use System V or POSIX shared memory for storing notifications such that 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 using shared memory provided by the applications, such as MPI windows, in interoperability mode. Instead of node locally packing/unpacking datatypes, the implementation then will publish its respective rank-local datatype layout and will subsequently and merely notify the availability of rank-local data for node-local reading. Data for remote nodes can be aggregated across multiple node-local ranks. As the GASPI notifications will be globally visible on the remote target node all the corresponding remote processes running on that node will be able to see and extract their communication parts. All these features are planned to be releases in the new version of GASPI by mid-July 2017 and, therefore, will be tested in Ludwig.

Acknowledgement This work was funded by the European Union's Horizon 2020 Research and Innovation programme through the INTERTWinE project under Grant Agreement no. 671602 (`www.intertwine-project.eu`).

References

1. JC. Desplat, I. Pagonabarraga, and P. Bladon. Ludwig: A parallel lattice-boltzmann code for complex fluids. *Computer Physics Communications*, 134(3):273–290, 2001.
2. R. Machado, T. Rotaru, M. Rahn, and V. Bartsch. Guide to porting MPI applications to GPI-2. Technical report, Fraunhofer ITWM, 2015.
3. 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 International Publishing, 2015.

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