PGAS implementation of SpMVM and LBM using GPI

Faisal Shahzad, Markus Wittmann, Moritz Kreutzer, Thomas Zeiser, Georg Hager and Gerhard Wellein

Erlangen Regional Computing Center
University of Erlangen-Nuremberg, 91058 Erlangen, Germany
{faisal.shahzad, markus.wittmann, moritz.kreutzer, thomas.zeiser, georg.hager, gerhard.wellein}@fau.de

Abstract

GPI is a PGAS model based library that targets to provide low-latency and highly efficient communication routines for large scale systems. We compare and analyse the performance of two algorithms, which are implemented with GPI and MPI. These algorithms are a sparse matrix-vector-multiplication (SpMVM) and a fluid flow solver based on a lattice Boltzmann method (LBM). Both algorithms are purely memory-bound on a single node, whereas at the large scale, the communication between the processes becomes more significant. GPI, in principle, is fully capable of performing communication alongside computation. Both the algorithms are modified to leverage this feature. In addition to the naïve approach with blocking calls in MPI, the algorithms are also evaluated using non-blocking calls and explicit asynchronous progress via an external library. We conclude that GPI implementations handle non-blocking asynchronous communication very effectively and thus hiding communication costs.

1 Introduction and background

In order to increase the efficiency and scalability of MPI-like programming environments, many developments have been made over the past several years. Using Partitioned Global Address Space (PGAS) languages is one part of these efforts. The PGAS model provides a shared memory view for distributed memory systems, thus making it possible to access (read/write) memory of the remote processes without their active involvement. The main motivation behind the PGAS programming model is to increase the simplicity of codes and to provide better efficiency and scalability at the same time. Some of the PGAS languages gained some popularity like Coarray Fortran[2], Unified Parallel C[3], and Chapel[4]. Some other implementations provide a PGAS communication model via API calls. This avoids the need to learn a new language and a complete rewrite of applications. The Global address space Programming Interface (GPI)[5] and Global Arrays Toolkit[6] are examples of such PGAS based APIs.

The GPI library is a relatively new addition to the PGAS programming model libraries. Developed by Fraunhofer ITWM[1], it focuses on providing low-latency, high speed communication routines for large scale systems.

In the scope of this paper, we implement two algorithms using GPI. The first is a sparse matrix-vector-multiplication (SpMVM) algorithm. SpMVM is a significant operation which occurs in many scientific algorithms quite often and makes up a large amount of the overall application runtime. The second algorithm is a fluid flow solver based on a lattice Boltzmann method (LBM)[7]. Due to its effective algorithmic implementation and ease of parallelization,
LBM has gained significant importance in scientific and research communities. We compare the performance and scalability of our GPI vs. MPI implementations and extend our previous work [8] where MPI and Coarray Fortran were compared.

**Related Work:** Grünwald [9] used a stencil based application (BQCD) to compare GPI and MPI implementations. A performance advantage of 20-30% is achieved for the GPI code through its ability to perform communication asynchronously to the work the application performs. In [10] Simmendinger et al. implemented the unstructured CFD solver TAU using GPI and receives a far better speedup as compared to MPI and hybrid MPI/OpenMP implementations. A similar comparison for a parallel adaptive search algorithm is presented by Machado et al. in [11], who achieve a better speedup than the MPI implementation for some test cases. The core reason for the improvement in application performance with GPI is the reformulation of algorithms to benefit from the one-sided asynchronous communication capabilities of GPI.

An important point in optimizing communication is to create or leverage communication hiding. Hereby the communication library (e.g. PGAS or MPI) communicates in the background while the application continues with its own work. In this case asynchronous progress of the message transfer is performed and communication and computation overlap. The MPI standard does not require this behavior (also not for the non-blocking point-to-point semantic) and only high-quality implementations provide support for this. A rough evaluation of MPI implementations concerning this feature can be found in [12] and [13]. To provide asynchronous progress, even if it is not supported by the MPI implementation, the APSM library (Asynchronous Progress Support for MPI) [13] can be used, which requires a thread-safe MPI library. Transpareently to the user, non-blocking MPI communication calls are intercepted by the library and an own progress thread is used to drive the message transfer in the background. The pinning (a.k.a. affinity) of the progress thread can be set by an environment variable. We test the GPI performance against the MPI versions of the algorithms with and without APSM-library support.

The main contributions of this paper are:

1. Implementation of SpMVM with GPI.
2. Implementation of LBM with GPI.
3. Optimization of GPI based SpMVM and LBM implementations to achieve asynchronous communication.
4. A thorough performance comparison of GPI with their respective MPI implementations.
5. The performance benchmark of MPI with APSM support and its comparison with GPI performance.

This paper is structured as follows. In Sect. 2 we present an introduction to GPI. The experimental framework is presented in Sect. 3 In Sect. 4 the GPI implementation of SpMVM is elaborated. Further the performance benchmarks are conducted and compared with the MPI implementation. The same is done for the LBM code in Sect. 5 Finally, Sect. 6 gives the summary and concludes the paper.

## 2 Global address space Programming Interface (GPI)

The Global address space Programming Interface (GPI) is a library based on the PGAS style programming model for C/C++ and Fortran. The GPI API consists of a set of basic routines. Its communication layer can take full advantage of the hardware capabilities to utilize remote
direct memory access (RDMA) for spending no CPU cycles on communication. Each GPI process has a global and local memory. The global memory can be accessed by other processes, i.e. the data that has to be shared with other processes must be located in this memory region. The local memory is private to each process and hence is not accessible to other processes. The amount of global memory for each process must be specified at the start of the application when GPI is initialized by the startGPI call (similar to MPI_Init).

The API of GPI provides one-sided and the usual two-sided communication calls. The one-sided calls consist of readDmaGPI and writeDmaGPI. Both of these calls are non-blocking and require a wait call (waitDmaGPI) to ensure the accomplishment of the read/write operation. The two-sided communication calls are sendDmaGPI and recvDmaGPI. All communication calls can only operate on the global memory. The only exception to this is allReduceGPI call which can use local memory as well.

GPI only supports one GPI process per socket (or node), thus for multi-core environments, GPI relies on a threading module called Multi-core Threading Package (MCTP) for intra-socket resource utilization. The MCTP threading module also is developed by Fraunhofer ITWM and provides fine NUMA-aware control over threads. The more widely used OpenMP programming model can also be used as an alternative threading package. A minimum of two nodes are required to run a GPI program. GPI supports InfiniBand and Ethernet (RDMAoE, RoCE) interconnects.

The present version (1.0) of GPI also provides basic fault-tolerance on a process level. Unlike an MPI application, if one or more processes of a GPI application fail, the remaining processes can proceed with their work as usual.

3 Experimental Framework

For performance evaluation, we have used RRZE’s LiMa\(^2\) cluster. This cluster comprises of 500 compute nodes equipped with two Intel Xeon 5650 “Westmere” CPUs (six physical cores, two-way SMT cores) running at the base frequency of 2.66 GHz with 12 MB shared cache per chip. The ccNUMA system has two locality domains, each with 12 GB RAM (24 GB in total). The STREAM (scale) benchmark \(^{[14]}\) achieves a bandwidth of around 40 GB/s (20 GB/s per socket). Simultaneous multithreading (SMT) and “Turbo Mode” are enabled. The system is equipped with Mellanox QDR InfiniBand (IB) and GBit Ethernet interconnects. GPI can only use the IB interconnect as the onboard GBit chips do not support RDMAoE or RoCE.

In the scope of this paper, we present the implementation details and results of the following two algorithms.

\textbf{SpMVM:} The Sparse Matrix-Vector-Multiplication is a significant operation which occurs in many scientific applications. Mostly it covers a significant portion of the overall computation time. Thus its efficiency is critical to the application performance. A SpMVM operation consists of \(\vec{y} = A \vec{x}\), where \(A\) is an \(n \times n\) dimensional matrix, and \(\vec{x}, \vec{y}\) are \(n\) dimensional vectors. The operation can be written as

\[
y_i = \sum_j (A)_{i,j} \cdot x_j.
\]

\(^{[14]}\)LiMa cluster at the Erlangen Regional Computing Center (RRZE): \texttt{http://www.hpc.rrze.fau.de/systeme/lima-cluster.shtml}
Table 1: The matrices used for benchmarks.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Dimension</th>
<th>Avg. NNZ per row</th>
<th>size in MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRZE3</td>
<td>6.2 \cdot 10^6</td>
<td>19</td>
<td>1530</td>
</tr>
<tr>
<td>DLR1</td>
<td>2.8 \cdot 10^6</td>
<td>144</td>
<td>642</td>
</tr>
<tr>
<td>HV15R</td>
<td>2 \cdot 10^6</td>
<td>140</td>
<td>4545</td>
</tr>
<tr>
<td>RM07R</td>
<td>3.8 \cdot 10^6</td>
<td>98</td>
<td>602</td>
</tr>
</tbody>
</table>

In case of a dense matrix, it is obvious to store the matrix in an $n \times n$ array. On the other hand, for sparse matrices, the effective storage strategy of the matrix is critical for performance. Some of the most famous matrix storage schemes are Compressed Row Storage (CRS), Compressed Columns Storage (CCS), Ellpack-Itpack, Jagged-Diagonal, Blocked Compressed Row Storage (BCRS), Compressed Diagonal Storage (CDS), etc \cite{15}. For CPUs, the CRS storage scheme is known for its best all-round performance for non-structured sparse matrices. Therefore, in this paper, we follow this format. This scheme consists of three one dimensional arrays: \textit{values}, \textit{column_index}, and \textit{row_pointer}. The \textit{values} array consists only of non-zero values of the matrix. The \textit{column_index} array stores the column index of the corresponding non-zero matrix value. The \textit{row_pointer} determines the number of non-zeros in a row. The size of \textit{values} and \textit{column_index} arrays is equal to the number of non-zeros in the matrix. The size of \textit{row_pointer} array is equal to $n + 1$.

The parallel MPI and GPI implementation details and results are provided in Sec. 4. We have performed the benchmarks using four different sparse matrices: RRZE3, DLR1, HV15R\cite{16}, and RM07R\cite{16}. Table 1 shows the details of the matrices used whereas sparsity structure of the matrices is shown in Fig. 1.

\textbf{LBM:} The lattice Boltzmann Method\cite{7} is used as a fluid flow solver. In the recent past it has gained significant popularity in science and research communities also due to the fact that it is simple to parallelize. LBM can be seen as a Jacobi-like stencil algorithm, but with two major differences: (1) each cell has not only one, but (as in our case) 19 values and (2) no values read are reused during the same iteration over the lattice. Our implementation uses the D3Q19 model\cite{17} with the BGK collision operator\cite{18}. In each iteration, the data is read from the 4-D source lattice (three spatial dimensions plus one for the 19 cell values) and modified values are written to the 4-D destination grid. The update of one cell is performed by reading one value of each of the cell’s 19 surrounding neighbors. Out of these values new ones are computed, which are used to update the cell’s own values in the destination lattice. Thereby the values are arranged in a structure-of-array data layout (for details see \cite{19}).

In this paper, we have used a prototype application based on LBM which simulates a 3-D
SpMVM function with synchronous MPI communication.

Listing 1: The SpMVM function with synchronous MPI communication.

For both the applications, the performance of the five following cases is examined:

1. Blocking MPI communication (naive MPI case)
2. Non-blocking MPI communication
3. Non-blocking MPI communication with APSM library
4. Synchronous GPI communication
5. Asynchronous GPI communication

Our evaluation of the benchmarks is based on absolute performance of the application and the degree of overlap efficiency the library can offer. In addition, the scaling behavior of the overlap efficiency is also examined. For benchmarking, the Intel C++ compiler (version 12.1.11) with Intel MPI (version 4.0.3.008) was used. The pinning of the threads is explicitly performed from within the code by `sched_setaffinity()`.

4 SpMVM

The parallel implementation of the SpMVM algorithm is based upon distributing the matrix rows amongst the processes. Each process then calculates its result vector \( \vec{y} \) corresponding to its assigned rows. The division of the number of rows per process can be performed in two ways. A naive way is to divide the number of rows evenly between all processes. For sparse matrices, this approach can potentially result in work imbalance amongst processes. For optimal workload distribution, our implementation is based upon having (more or less) equal number of non-zeros for each process. Each process owns the part of the right hand side (RHS) \( \vec{x} \) that corresponds to its rows of the matrix. The calculation of the result vector \( \vec{y} \) can be seen as a combination of two parts i.e. a “local part” and a “non-local part”. The part for which the RHS values are local to the process is regarded as local part. The other part, for which the RHS values need to be fetched from other processes is regarded as non-local part.

In a simple synchronous communication approach, these parts are combined. Thus, communication of RHS is required before performing the sparse-matrix-vector-multiplication. Listings 1 and 2 show the SpMVM function in a synchronous communication setting for MPI and GPI, respectively. In the MPI case, the communication is two-sided and no global synchronization is required. On the contrary, two synchronization calls are required for the GPI case. The first synchronization ensures the completion of communication on all processes. This guarantees that each process has updated RHS values received by remote processes. The second synchronization is required before the next SpMVM. After that all processes have finished the previous SpMVM iteration and have up-to-dated RHS values.

In an asynchronous (non-blocking) communication setting, the local and non-local parts are computed separately. The communication request for reading the non-local RHS is placed before
the local-SpMVM kernel. A wait routine ensures the completion of this communication request. After receiving the non-local RHS values, the non-local part can now be computed. Listings 3 and 4 show the pseudo code of the SpMVM function with asynchronous communication for MPI and GPI, respectively. As in the previous case, no global synchronization is required for the MPI case. GPI on the other hand, requires the first synchronization call before computing non-local part. The second synchronization is needed before the next iteration.

The GPI asynchronous communication has two effects on the execution of the code. First, the overlapping of communication and computation, which results in better performance. The second effect is due to the introduction of a global synchronization call between local and non-local SpMVM parts. This is matrix dependent and can be large if the ratio of local to non-local NNZ amongst all processes is not even. If this imbalance is big, it can result in large overhead.

For benchmarking, we have followed a hybrid implementation i.e. MPI/OpenMP and GPI/OpenMP. The parallelization within the socket is performed using OpenMP threads. Figure 2 shows the performance comparison of five different cases (as described in Sec. 3) for different matrices. This benchmark has been performed on 32 LiMa nodes with 64 processes each having 6 threads. The performance of non-blocking MPI is nearly similar to naïve MPI (blocking MPI) for all matrices. This shows that Intel MPI (version 4.0.3) can not perform
the non-blocking communication calls in an asynchronous way. The usage of APSM library boosts the performance only in the case of RRZE3. For RRZE3 and DLR1 matrices, the GPI asynchronous communication performs better than the naïve MPI case. The APSM support case with RRZE3 matrix is even better than the asynchronous GPI. The synchronization cost in case of GPI introduces large overhead in the case of HV15R and RM07R and hides the advantage gained due to asynchronous communication. Thus, the performance for HV15R and RM07R is lower than the MPI cases.

Figure 3 shows the performance comparison between various cases of MPI vs. GPI in case of strong scaling from 2 to 96 LiMa nodes. As per GPI requirement, the baseline for this strong scaling starts with two nodes. For all matrices, the baseline performance of MPI and GPI is nearly the same except for RRZE3 where asynchronous GPI performs 35% better than the naïve MPI case. Around 32 nodes, the matrices DLR1 and RM07R fit into the L3 cache completely and thus the strong scaling performance does not increase in a linear fashion from this point on. The asynchronous GPI performance stays similar to the MPI performance for smaller number of nodes. For more than 32 nodes, the global synchronization in each iteration for GPI becomes more expensive and starts to reduce the performance.

5 Lattice Boltzmann Method

The domain in our test application consists of a 3-D lid-driven cavity. A hybrid parallel model is followed using a MPI/OpenMP and GPI/OpenMP. For parallelization, the domain is divided into slices in the Z-direction. Thus the ghost elements are exchanged in the Z-direction. The size of this communication depends on the number of cells in X and Y-directions. The runtime of the benchmark depends on the time-steps and the number of domain cells.

In a typical synchronous stream-collide based implementation, first the domain cells are updated in a stream-collide step. Boundary layers are then exchanged to update the ghost cells of each process before moving on to the next iteration. Listing 5 shows the algorithm for an LBM iteration loop with synchronous communication. As we have followed one-sided communication routines (read/write) in GPI, a barrier is essential before performing communication (i.e.
PGAS implementation of SpMVM and LBM using GPI  

Figure 3: The MPI vs. GPI SpMVM performance comparison in case of strong scaling for different matrices.

for (int t=1; t <= timesteps; ++t)
{
    update_cells();
    barrier();
    exchange_ghost_cells();
}

Listing 5: LBM iteration loop with synchronous communication. The barrier is essential before one-sided communication is performed (i.e. GPI case).

updating ghost cells) in order to make sure that the target buffer has already been used and is ready for the next update. As communication can only take place between global memories of processes, all the distribution-functions (including ghost elements) are allocated in the global memory of GPI. This avoids the in-memory copy of the ghost-elements from local to the global memory.

In order to take the benefit of GPI’s capability to perform communication asynchronously in the background of the application, we adapted the algorithm to make this overlap possible. This involved splitting the stream-collide routine into two parts. The first part includes performing stream-collide on the boundary (next-to-ghost) cells. After performing this step, the
for(int t=1; t <= timesteps; ++t) {
    update_boundary_cells();
    exchange_ghost_cells_begin();
    update_inner_cells();
    exchange_ghost_cells_end();
}

Listing 6: Pseudo code of LBM iteration loop with asynchronous communication (computation-communication overlap)

communication of ghost-layers is initiated. In the second stream-collide stage, the computation on the inner cells of the domain is performed while the ghost-layers communication takes place in the background. The wait call after the second stream-collide step makes sure that communication has been completed. Listing 6 shows a simple pseudo code for such a case.

For GPI, as the communication is one-sided (read/write), the synchronization between the processes is essential to make sure that the read values from the neighboring process are valid values (i.e. the remote process for the read operation has already updated the values to be read). One way to make this synchronization possible is by introducing a global barrier. In order to avoid the cost of global synchronization, we have implemented a synchronization scheme only between the communicating neighbor processes. The approach is similar to the one presented in [9]. It has been combined with a relaxed synchronization approach [20] to further enhance the communication performance. In a relaxed synchronization approach, each process copies its owed values in a separate so called transfer buffer, which is located in the global memory. One flag boundary_ready per process and direction is required in the global memory. The local process sets its flags after its updated data values are written into the transfer buffer. A remote process, who wants to fetch the local process boundary values, polls on this flag. When it becomes ready the remote process reads the desired values and unsets the flag again. The local process waits for the flag to be unset before it continues with the next iteration. Listing 7 shows the LBM iteration loop with such a relaxed synchronization. All cell data is stored in the local memory and only the transfer buffers and synchronization flags are allocated in the global memory.

For the LBM case in particular, some overhead gets inherently induced with the adjustment of the algorithm for GPI in order to achieve the communication overlap. The first overhead originates from the in-memory coping of the transfer buffers (i.e. from local to global memory). The second, rather small overhead is introduced by splitting the “stream-collide” routine which updates the cells and thus causing less efficient cache usage.

Figure 4 presents the results of weak scaling up to 96 LiMa nodes for MPI (naive, non-blocking and non-blocking with APSM) and GPI (synchronous and asynchronous). The domain size of 2100 × 2100 × 24 cells is selected for the base case of 2 nodes. As the communication is carried out over an InfiniBand network, a large cross-section of the domain is chosen (2100 × 2100) to have a significant contribution of communication to the overall runtime. This way, the communication hiding can be clearly seen. For weak scaling the domain size is scaled in Z-direction. The baseline case for this performance benchmark is the naive MPI case (i.e. no overlap between computation and communication). For the case of MPI with non-blocking communication, the performance remains largely unchanged. The case with APSM support (MPI, non-blocking, APSM) shows a performance improvement. Naive GPI beats the naive MPI implementation as the number of nodes get larger. The best performance is achieved with
for(int t=1; t <= timesteps; ++t)
{
    update_boundary_cells();
copy_boundary_cells_to_comm_buffer();
boundary_ready[local_rank][EAST] = 1;
boundary_ready[local_rank][WEST] = 1;

    wait_for(boundary_ready[remote_rank_east][WEST] == 1);
    wait_for(boundary_ready[remote_rank_west][EAST] == 1);

    read_remote_boundary_cells();

    boundary_ready[remote_rank_east][WEST]=0;
    boundary_ready[remote_rank_west][EAST]=0;

    update_inner_cells();

    wait_for(boundary_ready[local_rank][EAST] == 0);
    wait_for(boundary_ready[local_rank][WEST] == 0);
}

Listing 7: Pseudo code of LBM iteration loop having asynchronous communication and relaxed synchronization.

Figure 4: Performance of LBM implemented with MPI and GPI on LiMa in the case of weak scaling with 2100 × 2100 × 12 cells per process.

the algorithm adapted for asynchronous communication with GPI. The performance difference to all other variants becomes increases for larger number of nodes. With 96 nodes, the LBM implementation with asynchronous GPI case is ≈ 30% better than the naïve MPI one.

The communication time can only be fully overlapped as long as it is smaller than the computation time. In order to check the efficiency of overlap, we define the overlap fraction $\mu$ as follows:

$$\mu = \frac{T_{\text{sync}} - T_{\text{async}}}{T_{\text{comm}}}$$  \hspace{1cm} (2)

Here, $T_{\text{sync}}$ and $T_{\text{async}}$ represent the total runtime synchronous and asynchronous communication.
PGAS implementation of SpMVM and LBM using GPI

Figure 5: Fraction of communication that can be overlapped with computation for LBM implemented with MPI and GPI variants.

The overlap with non-blocking MPI is low and inconsistent. Utilizing the APSM library improves this behavior, but yet not to the best. The duration for communication completion gets reduced, but simultaneously the computation time increases. Hence no optimal overlap fraction is reached. The best results are achieved with asynchronous GPI with an overlap fraction close to 95%.

6 Summary

In order to improve the scalability, performance and ease of programmability for very large scale computing clusters, the usage of Partitioned Global Address Space (PGAS) is one implementation candidate. The GPI library is a relatively new addition to the PGAS programming model libraries. In this paper, we have implemented a sparse matrix-vector-multiplication (SpMVM) and a lattice Boltzmann method based application with GPI and MPI. For both algorithms a comparative performance study was concluded. The results revealed that in order to fully utilize the potential of GPI, algorithms have to be adapted to allow for a communication-computation overlap. In principle, this is also possible for MPI, but support for asynchronous communication depends on the used MPI implementation. With SpMVM the performance benefit with GPI is matrix dependent and the global synchronization reduces the performance on large number of nodes. In the case of LBM the expensive global synchronization could be replaced by a more relaxed synchronization scheme, which is the reason for the significant performance-gain over the non-blocking MPI implementation.
Acknowledgment

We are grateful to Fraunhofer-ITWM and Scapos for providing an evaluation license of GPI. We specially thank to Dr. Grünewald for his support and helpful discussions.

This work was supported by the German Research Foundation (DFG) through the Priority Programme 1648 “Software for Exascale Computing” (SPPEXA) and the Federal Ministry of Education and Research (BMBF) under project “A Fault Tolerant Environment for Peta-scale MPI-solvers” (FETOL) (grant No. 01IH11011C).

References
