Hybrid-Parallel Sparse Matrix–Vector Multiplication and Iterative Linear Solvers with the communication library GPI

Dimitar Stoyanov
Fraunhofer Institut fuer Techno- und Wirtschaftsmathematik
Fraunhoferplatz 1, Kaiserslautern
D-67663, Germany
stoyanov@itwm.fraunhofer.de

Franz-Josef Pfreundt
Fraunhofer ITWM
Fraunhoferplatz 1, Kaiserslautern
D-67663, Germany
pfreundt@itwm.fraunhofer.de

Abstract: We present a library of Krylov subspace iterative solvers built over the PGAS-type communication layer GPI. The hybrid pattern is here the appropriate choice to reveal the hierarchical parallelism of clusters with multi- and many- core nodes. Our approach includes asynchronous communication and differs in many aspects from the classical one. We first present the GPI-based implementation of the sparse matrix-vector multiplication and then, using as a benchmark the numerical solution of a Poisson boundary value problem in a unit cube, we compare the performance on Intel/Infiniband and CRAY XE6 architectures of our GPI-based Conjugate Gradients and Richardson methods against the ones available in PETSc. The results show good scalability and performance of our approach, at least comparable to these of PETSc.

Key–Words: multi-core clusters, PGAS, RDMA, GPI, hybrid-parallel iterative solvers, sparse matrix-vector multiplication, performance

1 Introduction

It is nowadays considered that the "flat-MPI" approach is (or will soon become) inappropriate to implement on distributed shared memory cluster architectures with multi- and many- core nodes, see e.g. [10], Chapter 4 of [3]. In general, the hybrid parallelization is regarded as a possible alternative: in its classical variant MPI handles the inter-nodal communication, while on the shared memory nodes a thread-parallelization (often via OpenMP) is assumed ([3]).

The Krylov subspace iterative methods presently have no alternative for the solution of large-scale systems of linear equations arising from the implicit discretization of PDEs. There are many performant linear solver libraries of this type, e.g. PETSc ([15]), based on the flat-MPI parallelization. On the other hand, the design and implementation of such Krylov type solvers using the hybrid-parallelization is still a challenging question.

This paper discusses the design and implementation of hybrid-parallel Krylov subspace methods, based on on the GPI library ([7], [12]). This is a communication layer of a PGAS-type, which uses RDMA. GPI is a clear alternative to the flat-MPI and - thus - belongs to the hybrid approaches. On the other side, certain features of the GPI programming distinguish it from the classical hybrid parallelization. We first present these specific features and advantages of GPI in a more general context, but then particularly with regards to the basic routine of the Krylov solvers - the sparse matrix-vector multiplication (SpMVM).

The main contribution of our work is related to the task-based hybrid design and implementation of the SpMVM kernel with the GPI programming layer. The GPI library supports and combines in a natural way the two levels of hybrid parallelization: the inter-nodal communication between the computational nodes and the local thread parallelization on the node. The task based approach allows to perform inter-nodal exchange from within thread parallel regions of the program. Note, that programming hybrid parallel libraries of iterative linear solvers is sail a challenge and such libraries are rarely available for download. A hybrid PETSc ([15]), for example, is under development and it seems to follow the Aler native vector-mode hybridization, which leaves the inter-nodal exchange outside of the thread parallel sections of the program (see more about the vector and task modes in the next section). In our case, the task based approach used in SpMVM is much more flexible with regards to a better communication-computation overlapping which finally improves the overall performance of the solvers. Describing below our design and implementation of SpMVM we also show how inherently the task mode approach is related to GPI and - therefore - naturally implementable. At the same time, using MPI and threads to implement task based parallelization is still a subject of certain constraints, see below. Our performance results confirm the flexibility of our approach: we have achieved better or comparable performance versus a flat-MPI highly optimized PETSc.

1.1 Related work

There are other flat-MPI alternatives, e.g. UPC([1]) or co-array Fortran (CAF, [11]). Most of them are programming languages and would request a (full) redesign and rewriting of an existing MPI-code, while GPI, as a library with an API semantically close to MPI, would rather require some modifications and not a full redesign.

An overview related to the SpMVM is provided in [2], see also the references therein. One may also mention pOSKI ([16]) - a collection of autotuned SpMVM-related kernels. It is designed for multicore machines, i.e. it provides intra-node optimization but does not consider inter-nodal communication patterns at all, while our implementation allows it. Among the numerical linear algebra libraries, one may also point out LAMA ([5]) which enables hybrid parallelization as well.

1.2 Organization of the paper

We first shortly describe the general features of the GPI-based programming and then we sketch the implementation of the SpMVM routine. Further, for our model problem - Poisson equation in a unit cube - we present the performance of our GPI-based numerical solver and compare it with the performance of identical methods in PETSc. Finally some conclusions are drawn.

2 Hybrid Parallelization and GPI

2.1 Short description of GPI

The idea of GPI (Global address space Programming Interface [7], [12], named FVM earlier) is to stay close to the hardware (Infiniband) limit by using only a thin interface and thus introducing an insignificant overhead. GPI employs the RDMA model: it allows one-sided communication to avoid the double-buffering usually used with MPI. Otherwise, the PGAS API of GPI is semantically very similar to the (asynchronous) MPI-communication commands.

The one-sided transfer mechanism is handled via a (large) Global Address Space (shortly GAS). The latter is constructed by the unifications of the partitions belonging to the corresponding GPI-nodes, i.e. each node contributes its partition(s) to GAS. One may consider GAS as a distributed shared memory: each node can read/write in the GAS partitions(s) of the other nodes using the GPI API. On the other side, a local GAS partition on a node is also directly accessible (e.g. with `memcpy`) from the local memory allocated for the application in the usual way (`malloc`, etc.). Thus the access to the local memory of the remote nodes is manageable through additional transfers to GAS.

Note, that all the threads on a node can directly access the remote GAS partitions of other nodes. This is an essential feature of the GPI architecture which facilitates the task-basic programming model and imposes a threaded view on the computations as an alternative to the process-based computations. A schematics representation of the GPI architecture is given on Fig. 1.

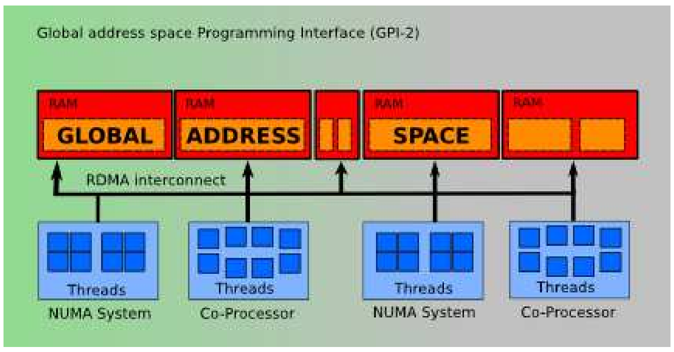

Figure 1: GPI architecture

Despite some similarities between GPI API and MPI, a profound rethinking of the existing parallelization patterns is often requested by the GPI approach, aiming mainly at better overlapping between communication and computation. Particularly with regards to the implicit numerics for PDEs this may be different, because solving the linear system there is the most ”expensive” part of the overall solution process and if a library with Krylov methods could ”hide” the complexity of an optimal GPI communication from the user, then the redesign of an existing application should not request immense efforts.

2.2 Hybrid parallelization and GPI model

The hybrid-design should reveal the inherent parallelism of the (clustered) SMP-nodes, with their built-in hierarchy of multi-cores and multi-sockets. The usual hybrid approach, combining MPI with OpenMP, raises several key issues ([3]):

(i) Mapping problem: how to inter-relate the threads and the MPI-process(es) to the cores on a single node/socket. MPI is not thread-safe, MPI-calls can not be executed within thread-parallel regions. Usually the main thread is associated with a MPI-process (one per node/socket), the other threads are then mapped to it. The mapping between (subset of) threads and MPI-processes is a responsibility of the programmer, the different mapping strategies are practically not supported by MPI;

(ii) NUMA-placement or data locality problem requires minimization of the NUMA-traffic and introduces thread-affinity related problems. The goal is to provide an optimal data locality for a thread or MPI-process on a multi-socket node;

Neither MPI nor OpenMP provide automatically solutions to these two issues. In both cases different combinations are possible, but they should be explicitly programmed. Several options have been proposed in [3] to resolve these issues. The data locality problem is to be handled (by the programmer) by using the ”first-touch” policy, which guarantees that the requested memory chunk is mapped to the locality domain of the core which first writes on it. Therefore, an appropriate memory initialization (performed in parallel by the threads) is needed here. For the mapping problem [3] suggests two alternative programming modes. In Vector mode the MPI-calls are performed outside of the thread-parallel regions of the code, while the Task Mode, on the contrary, allows any MPI-communication within thread-parallel regions. This second mode is more flexible and appropriate for overlapping communication/computation, but is practically not supported by both OpenMP and MPI ([3]) - it depends on the availability and the implementation of the interface mechanism between MPI and OpenMP (there are several interface levels, prescribed by the standards), and MPI is not thread safe.

GPI **automatically** resolves the two issues mentioned above. The data locality problem is handled at initialization time by the ccNUMA-aware version of GPI which maps a logical GPI-node (with the corresponding rank in the GPI space) to a socket (i.e. not to a physical multi-socket node), assigning the first two threads on the node a locality domain belonging to each of the two sockets correspondingly. Each of these two threads becomes a master thread on ”its” GPI-node and afterward all user allocations, thread spawning, etc. are ”pinned” to this locality domain. Further, GPI naturally supports the Task mode programming, i.e. the mapping issue is solved: in the GPI model such mapping problem simply does not exists. The only working agent is the computing thread and each thread can communicate on both inter-nodal and intra-node levels. The main challenge of the GPI programming is the proper management of GAS.

2.3 SpMVM hybrid implementation

SpMVM is a memory–bounded routine ([8], [2]). SpMVM-kernels perform poorly, achieving $\sim 10\%$ from the theoretical peak performance ([2]), being far from reaching the theoretical speedup even on SMP-architectures. The principal problems related to the SpMVM performance are known ([2]), to list some of them:

(i) restricted temporal locality: little data reuse, e.g. the matrix elements used once only;

(ii) irregular access to the input vector;

(iii) large number of matrix rows of a very short row-length to multiply;

(iv) indirect memory access imposed by the sparse matrix storage formats, etc.

To solve large scale linear systems with Krylov subspace methods on **hybrid architectures** one usually uses hierarchical decomposition: the coarse grained parallelism is attained by domain decomposition (e.g. by using graph-partitioning tools like METIS [14]), while the fine-grained parallelism on the node is achieved by thread parallelization.

The standard hybrid approach (MPI and OpenMP) imposes certain limitations on the SpMVM - the implementation of this routine usually follows the Vector mode approach [3]. This pattern clearly separates the data access performed on different levels: the distributed coarse-grained inter-nodal exchange, handled by MPI (usually one process per node), and the shared data access on the node, handled by SMP-aware (thread) implementation [8], [4]. Thus during the iterations one has to continuously switch between the two working agents - the MPI-process and the thread - each of them employing different data access pattern. For memory intensive routines like SpMVM this approach is not optimal, at least because it provides a single channel for remote communication. Double buffering, often used with MPI, introduces additional overhead. Note that even if one-sided (asynchronous) MPI-communication is employed, it seems not to be always properly working ([9]).

3 SpMVM with GPI

The GPI architecture imposes certain constraints on the SpMVM-design:

(i) the SpMVM routine exchanges vector items via GPI, therefore the transfer buffers related to the “local” part of the distributed vectors should permanently reside in the GAS partition of the node, to avoid additional exchange between the local memory and GAS;

(ii) it is neither reasonable nor possible on each GPI node to keep a local replica of the full input vector when SpMVM is performed;

(iii) therefore, one should copy locally only those remote items of the input vector, which are needed "here" (i.e. requested by the matrix elements, distributed on this GPI-node). Further, this inter-nodal transfer should be done once for the whole data-exchange chunk between two nodes, the latter being a set of values for a list of vector indices.

The fulfillment of (iii) at run-time depends on the current sparsity structure of the matrix and thus the communication scheme is irregular and problem-dependent. The domain (or mesh) partitioning determines the interface nodes for each subdomain (SD) and thus provides the information needed in (iii). At the discretization stage we work out the topological information for each discretization node at the interface: for each SD (mapped uniquely to a GPI-node) a set of send- and receive- buffers (linked lists of indices, associated with the discretization nodes at the SD-interface) is created. Then at run-time, when the SpMVM routine is invoked but before starting the actual SpMV-multiplication, GPI uses these buffers to perform the transfer of the remote input vector items.

To briefly describe our approach we assume a row-wise matrix distribution and we designate the locally distributed matrix rows (i.e. on "this" GPI-node) as $\mathbf{A}$, the input vector to be $\mathbf{X}$, and the local part of the output vector - $\mathbf{Y}_{\mathtt{lcl}}$, i.e. the SpMVM-routine on "this" GPI-node finally has to calculate

$$\mathbf{Y}_{\mathtt{lcl}} = \mathbf{A} * \mathbf{X}. \tag{1}$$

A standard way in SpMVM to overlap communication and computation (see e.g. [6], [4]) requires a decomposition of $\mathbf{A}$ into: (i) a local part $\mathsf{A}_{\mathtt{lcl}}$, which multiplies the local part $\mathsf{X}_{\mathtt{lcl}}$ of the input vector X, and (ii) its complementary matrix-chunk $\mathsf{A}_{\mathtt{rmt}}$, containing elements which multiply the remote-part $\mathsf{X}_{\mathtt{rmt}}$ of the input vector. The elements of $\mathsf{X}_{\mathtt{rmt}}$ correspond to the discretization nodes positioned at the DD-interface: these items belong to remote GPI-nodes and should be transferred to the current node before starting the SpMVM. Formally $\mathbf{X} = \mathsf{X}_{\mathtt{lcl}} + \mathsf{X}_{\mathtt{rmt}}$ holds and according to this decomposition the equivalent SpMVM operation can be written as:

$$\mathbf{Y}_{\mathtt{lcl}} = \mathsf{A}_{\mathtt{lcl}} * \mathsf{X}_{\mathtt{lcl}} + \mathsf{A}_{\mathtt{rmt}} * \mathsf{X}_{\mathtt{rmt}} \tag{2}$$

Then (1) and (2) distinguish the synchronous and asynchronous SpMVM. We now briefly sketch three variants of the SpMVM routine.

3.1 Synchronous SpMVM, static load distribution

The idea is to first transfer the remote part $\mathsf{X}_{\mathtt{rmt}}$ of the input vector into the local GAS-partition of the node (actually only the items requested here, on "this" GPI-node) and then to perform the SpMVM as in (1), i.e. the matrix is not split into local- and remote parts. The disadvantage is that the threads which do not take part in the transfer of $\mathsf{X}_{\mathtt{rmt}}$ should just wait idle during this transfer. The work-load distribution to compute (1) is static, because the overall nodal load (matrix rows to multiply) is known and it can be distributed directly (and uniformly) over the available threads.

3.2 Asynchronous SpMVM, static load distribution

The threads on the node are subdivided into two subsets, performing independently the two multiplications (local and remote) in (2). Thus the threads of the "remote" subset should first transfer locally $\mathsf{X}_{\mathtt{rmt}}$ and then multiply the remote part. At the same time the threads belonging to the "local" subset perform the local part of the multiplication. One should at last synchronize all threads on the node before carrying out the addition in (2). Within each subset of threads the work-load for the multiplication is statically distributed over the threads because the overall number of matrix rows to multiply is known in advance. The problem in this SpMVM-variant is that an a priori splitting of the threads into local- and remote-subsets can not take into account the current matrix structure in a flexible way, i.e. it may happen that either the local- or the remote- subset stays idle and waits to start the addition in (2).

3.3 Asynchronous SpMVM, dynamic load distribution

SpMVM is performed according to (2). The multiplication itself - of a single matrix row or of a set of rows - is entirely asynchronous operation. Thus an asynchronous job polling mechanism is applicable to both local- or remote- parts of the matrix (providing the requested input vector items are already locally available for the remote-SpMVM). The idea is that during the transfer of $\mathsf{X}_{\mathtt{rmt}}$ (performed by some subset of threads, as many as the neighboring SDs are), the remaining threads poll jobs to perform the local part of the multiplication. When the transfer of $\mathsf{X}_{\mathtt{rmt}}$ is over, then all threads poll multiplication jobs from both local- and remote- parts of (2). Again a synchronization of all the threads on the node is needed before the addition. Note that pre-setting some reasonable job size in the polling is important, because there are a large number of very short rows to multiply. Therefore, if the job size (i.e. the number of matrix rows to be multiplied) is too small, this would inevitably lead to a bottleneck: the threads will be mostly competing to get a job, instead of performing the multiplication. The important feature here is that all of the threads perform asynchronously and simultaneously the SpMV-multiplication of both local- and remote-parts. The polling mechanism provides dynamic load-balancing, making the overall work load close to the optimal, presumably with no idle threads.

3.4 Advantages of the GPI-based SpMVM

Task Mode ([3]) is - both in general and particularly for SpMVM ([9]) - straightforwardly supported by GPI, i.e.:

(i) each thread can not only locally access data, but may perform data transfer directly on inter-nodal level as well, i.e. the communication with different remote nodes can be handled at the same time by different threads;

(ii) the asynchronous (dynamic) SpMVM-model allows better and subtle fine-grain tuning for the communication - computation overlapping.

The approach (i) works for both synchronous and asynchronous SpMVM and differs from the standard hybrid pattern. The threads are spawned in the beginning of the iterative method routine and are joined at its end, working on both fine-grain (local) and coarse-grain (distributed) exchange levels.

4 Model Problem and Domain Decomposition

We solve a Boundary Value Problem (BVP) for the Poisson equation in a unit cube which allows an (easily constructed) exact solution. The discretization is on a regular rectangular mesh with second order finite differences. Then the $O(h^2)$-convergence of the numerical solution would indicate a correct implementation. If we discretize in the internal mesh-nodes only, the assembled matrix is symmetric and positive definite (SPD), and the linear system can be solved with the Conjugate Gradients (`CG`) method.

Domain Decomposition (DD): The load distribution over the GPI-nodes is handled through a simple "cutting planes" approach with planes perpendicular to the z-axis: the cube is partitioned into slices along the z-axis. A schematic representation of this DD in a two dimensional projection can be seen on Fig. 2. After assigning each slice to a particular GPI-node,

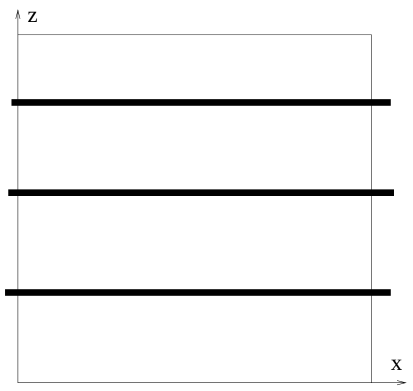

Figure 2: Domain Decomposition: cutting planes perpendicular to the z-axis, two dimensional projection

the discretization nodes that belong to it are then uniformly distributed over the threads. The distribution of the matrix rows over the GPI-nodes and then over the computing threads matches exactly this nodal distribution. Note, that with this DD the attempts to minimize the inter-nodal communication via RCM-type reordering as in the case of FE-meshes ([8], [6]) would have little or no effect. Conversely, in this setup raising the number of SDs (i.e. the number of z-slices) leads to a higher communication/computation ratio on a GPI-node, because the z-slices become thiner and the weight of the communication (between two neighboring z-slices) increases. Focusing mainly at the implementation of the solvers, our DD is not optimal (e.g. we do not use graph partitioning libraries), but it can be identically applied to different solvers and thus allows a fair comparison.

5 Performance Results

Our library (containing presently several methods) follows the pattern of many numerical linear algebra packages: over the vector- and matrix- classes a layer of basic linear algebra routines is built, while the iterative linear solvers themselves stay on top of this hierarchy. The matrix- and vector- entities are distributed in a GPI sense and a further multi-threading is assumed locally on the node - e.g. each thread works over "its" portion of the local part of a vector, thus such a "static" distribution is valid for all linear algebra routines (apart from SpMVM). CRS (Compressed Row Storage) is the matrix format used. It is only internally employed, otherwise the user accesses the matrix elements in a standard way by specifying the corresponding row- and column- indices.

5.1 General Comments about the Performance Comparison

The $O(h^2)$ convergence of our CG implementation has been confirmed on a sequence of nested meshes. Below we present the performance of the solver for large scale problems obtained on the finest meshes. In our first set of tests we do measurements using the Not-NUMA-aware GPI variant where the physical (multi-socket) node is mapped to a GPI-node. These tests have been presented here with the only goal to clearly underline the advantage of the alternative approach, i.e. the ccNUMA-aware GPI. In all other test cases this latter variant of the library has been used: it considers a socket (on a computational node) as a "logical" GPI-node, avoiding the NUMA-effects on the logical node. In fact - due to GPI - the inherent parallelism of an SMP-node is directly exploited: one can start GPI even on a single multisocket node, moreover with a NUMA-placement- and thread-affinity-requirements satisfied by default.

The domain partitioning for the PETSc solver is the same as in the GPI case: each SD (a z-slice of the cube) is assigned to a physical node, and the discretization nodes belonging to the SD are then uniformly distributed over the MPI-processes running on this node. Note, that despite the fact that in the GPI case the "logical" subdomains are twice (or four times, see below) more than in the PETSc-case, the decomposition from the point of view of the physical node is identical.

A convincing and obvious performance comparison between the two solvers was not easy to find with the CG-method used as a basis. Therefore, to attain an evident performance estimate, we implemented over GPI the Jacobi preconditioned Richardson method and compared it with the one available in PETSc. In this case a clean comparison is possible, because the calculations performed by the two routines are identical. This can be shown by monitoring the current residual on each iteration of both solvers. For the resulting linear system of our model problem we have measured the execution time to perform 4000 Jacobi-preconditioned Richardson iterations. The initial approximation of the solution is in both cases zero and after 4000 iterations in both solvers we get identical values for the $L2$-norm of the residual and the C-norm of the error (the difference between the numerical and the exact solution).

5.2 Not-NUMA-aware GPI, Intel cluster, Supermicro X8DAH (5520 Chipset), CG

The physical node has 48 GB RAM and consists of two sockets with 6 cores each. To take into account the system jittering we perform 10 test-runs in each particular case (2, 4, 8 and 16 physical nodes). Table 1. summarizes the results for synchronous static (SS), asynchronous static (AS), and asynchronous dynamic (AD) SpMVM-implementations. The first column of the table gives the type of SpMVM used and the number of threads performing the "remote" transfer in (2). The fastest (averaged) execution times have been printed **bold**.

One can see, that in the case of more (≥ 4) GPI-nodes the advantages of the asynchronous SpMVM with dynamic distribution are clearly visible, the better performance in this case can not be neglected.

5.3 NUMA-aware GPI, Intel cluster, Supermicro X8DAH (Intel 5520 Chipset), CG

In this case each socket on the physical node is considered as a logical GPI-node. Again with **GPI-CG** we perform 10 test-runs in each particular case: 1, 2, 4, 8 and 16 computational nodes, twice more logical GPI-nodes correspondingly. Table 2. presents the results for synchronous static (SS), asynchronous static (AS), and asynchronous dynamic (AD) SpMV-implementations. The first column of the table indicates the type of SpMVM used and the number of threads performing the "remote" transfer in (2) (the total number of threads per logical node is 6). The fastest (averaged) execution times printed **bold** in the table.

Comparing the results in Table 1. with the one from Table 2. the performance gain in the latter case is obvious. Therefore, all of the tests that follow below have been executed exclusively with the ccNUMA-aware GPI.

Aiming at comparing our performance with the one of PETSc-CG, we use PETSc-3.2. optimized for Intel-architecture. But this CG-based test turns out not to be perfect as a way to compare the two solvers. The reason is that the convergence behavior of the PETSc-CG solver (numerical error and number of iterations) is different and depends on the number of computational nodes. Table 3. shows that for a comparable error of the numerical solution the PETSc-number of iterations is always much smaller than in our case, which makes the performance comparison not "clean". The PETSc-CG speedup results have been obtained for several different values of the relative tolerance, attempting to get comparable numerical errors of the two solvers. The appropriate value seems to be $3e-9$: for it we have compared the performance of PETSc-CG against the one of GPI-CG. Table 3. presents the speedup results of our NUMA-aware GPI-based CG-solver (best times from Table 2, RelToll=$1e-8$) versus the PETSc-CG solver

Table 1: NO-NUMA-aware GPI, Synchronous static (SS), Asynchronous static (AS), and Asynchronous dynamic (AD) SpMVM: CG for 3D Poisson eqn. 257^3; ($relTol = 1e-8$, $||err||_C = 1.324256e-7$, $itrs = 866$)

Num Thrds in ThreadSet$_{\mathtt{rmt}}$	2 nds, time[s]			4 nds, time[s]			8 nds, time[s]			16 nds, time[s]		
	Min	Max	Avrg	Min	Max	Avrg	Min	Max	Avrg	Min	Max	Avrg
SS, (0 of 12)	169	172	**170.5**	93	95	94.1	53	55	54.3	33	37	35.9
AS, 1 (of 12)	194	197	195.2	105	107	105.4	58	59	58.9	39	40	39.6
AS, 2 (of 12)	194	196	194.7	101	103	101.9	53	53	53	33	34	33.4
AS, 3 (of 12)	197	200	197.9	103	105	103.5	54	55	54.1	32	33	32.6
AS, 4 (of 12)	198	204	200.5	106	110	107.3	54	57	55.6	32	32	32
AD, 2 (of 12)	174	173	173.5	91	92	**91.3**	48	49	**48.5**	28	30	**29**

Table 2: NUMA-aware GPI, Synchronous static (SS), Asynchronous static (AS), and Asynchronous dynamic (AD) SpMVM: CG for 3D Poisson eqn. 257^3; ($relTol = 1e-8$, $||err||_C = 1.324256e-7$, $itrs = 866$)

#Thr TS$_{\mathtt{rmt}}$	1 nd (2 logcl), t[s]			2 nds (4 logcl), t[s]			4 nds (8 logcl), t[s]			8 nds (16 logcl), t[s]			16 nds (32 logcl), t[s]		
	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
SS,0	179	180	**179.7**	93	94	**93.2**	50	51	**50.3**	30	31	30.6	21	22	21.3
AS,1	205	206	205.8	104	105	104.3	53	54	53.7	29	29	29	20	21	20.4
AS,2	205	206	205.7	119	120	119.7	61	62	61.1	31	32	31.1	15	16	**15.4**
AS,3	285	286	285.9	145	146	145.6	74	74	74	38	38	38	19	20	19.6
AD,2	196	197	196.6	99	100	99.2	51	52	51.8	28	29	**28.5**	15	16	**15.5**

(RelToll=$3e-9$). The graphical representation of the speedup in for NUMA-GPI-CG can be seen on Fig. 3., and the speedup of PETSc-CG - on Fig. 4.

As an outcome of this test it is clear that even on smaller number of nodes our execution time ”per iteration” is shorter than that of PETSc-CG, and on more nodes our advantage increases. This assessment “per iteration” is not quite correct, but it gives certain indication. Anyway, to get a fair comparison one needs more convincing tests.

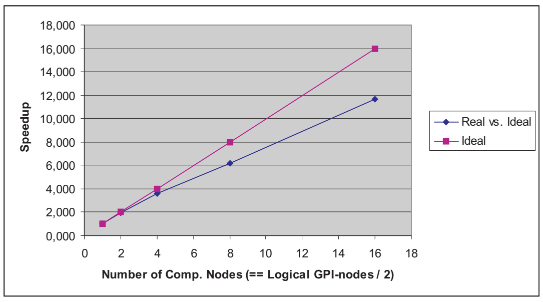

Figure 3: Speedup: GPI-CG, Poisson BVP, 257^3

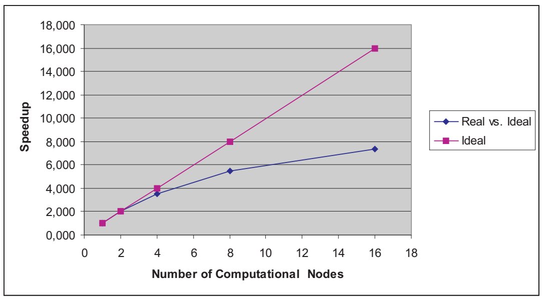

Figure 4: Speedup: PETSc-CG, Poisson BVP, 257^3

5.4 Intel cluster, Jacobi Preconditioned Richardson

With this method the comparison is reliable because the two solvers preform identical computations. On 8 and 16 nodes we use the Asynchronous Dynamic SpMVM with two threads for the remote transfer in (2) as in Table 2. For lower number of nodes (≤ 4) we use Synchronous SpMVM. As before, PETSc executes 12 processes per node. Table 4. presents the execution time (in seconds) of both solvers, while the

Table 3: Speedup CG, 257^3; The data for the numerical error ($||e||_C = ||exact - appr||_C$) to be multiplied by $1e-7$; PETSc-CG (12 procs/nd, RelTol=$3e-9$) vs. NUMA-GPI-CG (best times, 6 threads/GPI-nd, RelTol=$1e-8$)

	computnl. nds: 1			computnl. nds: 2			computnl. nds: 4			computnl. nds: 8			computnl. nds: 16		
	$\|\|e\|\|_C$	itrs	t[s]	$\|\|e\|\|_C$	itrs	t[s]	$\|\|e\|\|_C$	itrs	t[s]	$\|\|e\|\|_C$	itrs	t[s]	$\|\|e\|\|_C$	itrs	t[s]
PETSc	1.152	335	88	1.171	343	44	1.428	361	25	1.321	390	16	1.203	475	12
GPI	1.324	866	179.7	1.324	866	93.2	1.324	866	50.3	1.324	866	28.5	1.324	866	15.4

Table 4: PETSc-Richardson vs. GPI-Richardson, 4000 itrs, 3D Poisson eqn. 257^3, Intel

	1 node (2 logical)	2 nds (4 logical)	4 nds (8 logical)	8 nds (16 logical)	16 nds (32 logical)
PETSc	552	300	167	100	69
ccNUMA GPI	426	220	118	64	34

corresponding graphical comparison for the speedup can be seen on Fig. 5. The results clearly show the advantage of our GPI-based implementation vs. PETSc with regards to the performance.

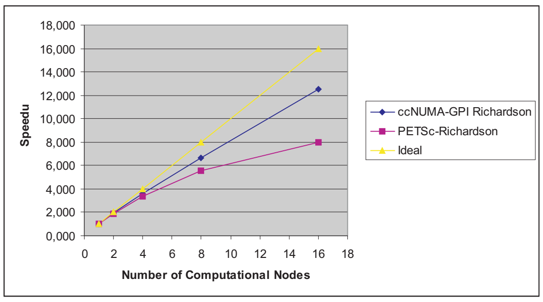

Figure 5: Speedup GPI vs. PETSc: Richardson, 257^3

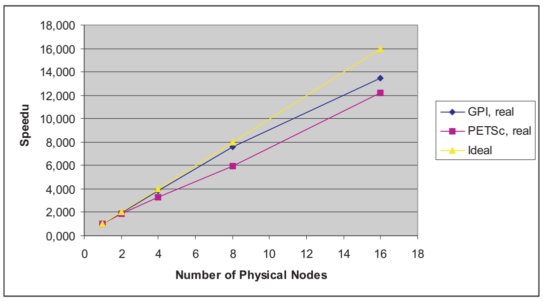

Figure 6: Speedup Richardson, GPI vs. PETSc, 301^3

5.5 Cray XE6, Jacobi Preconditioned Richardson

The physical node has 64 GB memory and four sockets (logical GPI-nodes), 8 cores per socket. PETSc, version 3.2., is hardware optimized for this architecture. On a larger number of cores (> 1000) one should run problems of appropriate size. Therefore, we tested the 301^3-case (299^3 unknowns) on a single node and then up to 16 physical nodes (Table 5. left, and Fig. 6.) and the tests for the 501^3-case (499^3 unknowns) we started on 4 and ended on 32 physical nodes (Table 5., right, and Fig. 7.). The executions times are rather comparable and we consider our performance good, because PETSc uses hardware optimized low-level mathematical libraries, while we do not use machine optimization at all.

6 Conclusion

We have presented a library of Krylov type solvers built on the GPI communication layer. GPI naturally leads to a hybrid parallelization appropriate for clusters of multi- and many-core nodes. The features of the GPI-based programming distinguish it from the classical hybrid techniques (MPI and OpenMP), and we have shown how these principles have been applied to implement the sparse matrix-vector multiplication. Using the solution of the Poisson equation in a unit cube as a benchmark we have compared the performance of our Conjugate Gradient and Richardson methods against those available in PETSc. To ensure a correct comparison we use the Jacobi-

Table 5: Richardson, GPI vs. PETSc, Poisson BVP, 301^3 snd 501^3; CRAY XE6

Problem size	$299^3 = 26730899$					$499^3 = 124251499$			
Total cores	32	64	128	256	512	128	256	512	1024
Physical nodes	1	2	4	8	16	4	8	16	32
GPI-nodes	4	8	16	32	64	16	32	64	128
GPI, exec. time [s]	552	285	145	73	41	692	351	194	111
PETSc, exec. time [s]	476	251	144	80	39	624	356	215	108

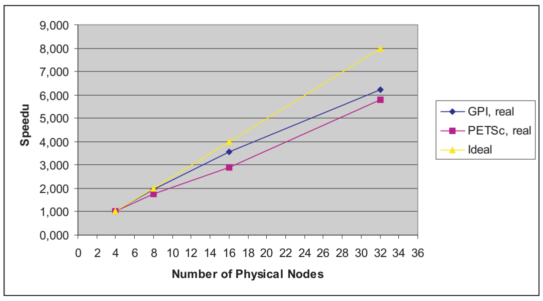

Figure 7: Speedup Richardson, GPI vs. PETSc, 501^3

preconditioned Richardson method, which performs identical computations in both PETSc and our GPI-based solver. The tests have been done on two architectures: Intel/Infiniband (Supermicro X8DAH, Intel 5520 Chipset) and Cray XE6 (up to 1024 cores). On the Intel-cluster our implementation shows clearly a better performance than PETSc, while on Cray both solvers produce comparable execution times. Our code uses no hardware optimization and there is still a potential for improvement of our GPI-based solvers. We plan further development and optimization of our library and to employ it in real applications.

References:

[1] T. El-Ghazawi, W. Carlson, T. Sterling, and K. A. Yelick: UPC: Distributed Shared-Memory Programming, Wiley-Interscience, Hoboken, NJ: Wiley, (2005)

[2] Gormas G. et al.: Performance evaluation of the sparse matrix-vector multiplication on modern architectures. J.Supercomput. DOI 10.1007/s11227-008-0251-8, Springer, (2008)

[3] Hagger G., Wellein G.: Introduction to High Performance Computing for Scientists and Engineers. CRC Press, (2010)

[4] Heuveline V., Lukarski D., Weiss J.-Ph.: Multi-Platform Linear Algebra Toolbox for Finite Element Solvers on Heterogeneous Clusters. 978-1-4244-8396-91101$26.00 IEEE (2010)

[5] Kraus, J., Foerster, M., Brandes, T., Soddemann, T.: Using LAMA for efficient AMG on hybrid clusters, Computer Science - Research and Development, Doi: 10.1007/s00450-012-0223-3, http://www.libama.org/overview.html

[6] Krotkiewski M., Darbowski M.: Parallel symmetric sparse matrix-vector product on scalar multi-core CPUs. Parallel Computing 36, 181–198 (2010)

[7] Machado, R., Lojewski, C.: The Fraunhofer Virtual Machine: A Communication Library and Runtime System based on the RDMA model. Computer Science – Research and Development 23, 125–132 (2009) DOI 10.1007/s00450-009-0088-2

[8] Nakajima, K.: Flat MPI vs. Hybrid: Evaluation of Parallel Programming Models for Preconditioned Iterative Solvers on T2K Open Supercomputer. IEEE Proceedings of the 38th International Conference on Parallel Processing (ICPP-09), 73–80 (2009)

[9] G. Schubert, H. Fehske, G. Hager, G. Wellein: Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems. Parallel Processing Letters **21, Nr. 3**, 339-358 (2011)

[10] Yelick, K.: Beyond UPC. Proceedings of the Third Conference on Partitioned Global Address Space Programing Models, (2009)

[11] http://www.co-array.org/

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

[13] https://wickie.hlrs.de/platforms/index.php/Cray_XE6

[14] http://www-users.cs.umn.edu/∼karypis/metis/

[15] http://www.mcs.anl.gov/petsc/

[16] http://bebop.cs.berkeley.edu/poski/