Task-Based Parallel Sparse Matrix-Vector Multiplication (SpMVM) with GPI-2

Dimitar Stoyanov$^{(\boxtimes)}$, Rui Machado, and Franz-Josef Pfreundt

Fraunhofer ITWM, Kaiserslautern, Germany
{stoyanov,machado,pfreundt}@itwm.fraunhofer.de
http://www.itwm.fraunhofer.de

Abstract. We present a task-based implementation of SpMVM with the PGAS communication library GPI-2. This computational kernel is essential for the overall performance of the Krylov subspace solvers but its proper hybrid parallel design is nowadays still a challenge on hierarchical architectures consisting of multi- and many-core sockets and nodes. The GPI-2 library allows, by default and in a natural way, a task-based parallelization. Thus, our implementation is fully asynchronous and it considerably differs from the standard hybrid approaches combining MPI and threads/OpenMP. Here we briefly describe the GPI-2 library, our implementation of the SpMVM routine, and then we compare the performance of our Jacobi preconditioned Richardson solver against the PETSc-Richardson using Poisson BVP in a unit cube as a benchmark test. The comparison employs two types of domain decomposition and demonstrates the preemptive performance and better scalability of our task-based implementation.

Keywords: GASPI · GPI-2 · PGAS · Task-based hybrid parallelization · Sparse matrix-vector multiplication · Krylov subspace solvers · Performance

1 Introduction and Motivation

The so called pure- or flat-MPI programming (one MPI-process per core) is nowadays no longer the most appropriate approach on systems with multi-core and multi-socket nodes. A hybrid parallelization is considered a natural choice instead: it combines a coarser, inter-node distributed memory parallelization with the more fine-grained, intra-node shared memory parallelization. Particularly, a task-based parallelization, where inter-nodal exchange can be independently performed by each thread from within thread-parallel regions, seems to be the proper alternative to reveal and fully exploit the hierarchical parallelism of such architectures.

The classical and most often used variant of hybrid parallelization is to combine MPI and threads/OpenMP. Particularly with regards to SpMVM this approach is followed for instance in [5,6]. But such a combination imposes

I. Lirkov et al. (Eds.): LSSC 2015, LNCS 9374, pp. 153–160, 2015.
DOI: 10.1007/978-3-319-26520-9_16

certain restrictions and performance issues with respect to thread-safety [2–4]. For instance, the MPI 2.0 standard prescribes four interface levels of threading support, one of them (MPI_THREAD_MULTIPLE) allowing a more task-based parallelization. MPI 3.0 improves some aspects related to threading - e.g., using MPI_Probe when several threads share a rank, etc. But in general, the hybrid parallelization based on MPI is still a challenge, it contains certain open issues (see e.g. [9], where also the new hybrid MPI+MPI approach is discussed), and it is often the case that MPI implementations do not provide a high performance support for task-based multi-threading. Consequently, applications aren't usually developed for such support and hence there is a non-optimal usage of resources - say, of the growing capabilities of high-performance interconnects. Particularly, many numerical libraries still use flat-MPI, e.g. in PETSc [11] threading has only recently appeared in the developers version.

Another point is that currently the trend in computer systems architecture is to see an increasing number of cores per node, with Non-Uniform Memory Access (NUMA) and with heterogenous resources. This not only puts pressure on multi-threaded support but it creates a need for more dynamic and asynchronous execution, to hide the latency of inter-node communication as well as that of intra-node memory and synchronization operations.

The GASPI interface [8] was specified with the previous aspects in mind and GPI-2 [7] was implemented to cope with them. The focus on asynchronous, one-sided communication with multi-threaded support and weak synchronization semantic creates an opportunity for new, more scalable implementations of performance critical building blocks such as the SpMVM, which is crucial for the case of Krylov solvers.

In this work, we present a task-based parallel implementation of SpMVM that takes advantage of our communication library GPI-2. We demonstrate the potential of our approach on the solution of a Poisson Boundary Value Problem (BVP) in a unit cube and we compare the performance against PETSc using two different types of Domain Decomposition (DD). The results show a significant performance advantage and better scalability when using the appropriate DD based on graph partitioning methods (METIS).

The rest of the paper is organized as follows: first we briefly describe the features of GPI-2 and our task-based SpMVM implementation, which differs in many aspects from the classical hybrid approach; then we formulate the model problem and explain the DD used in the comparisons. Further, we present and comment the performance results, and finally some conclusions are drawn.

2 GASPI/GPI-2 and Task-Based Parallelization

GPI-2 is the implementation of the GASPI standard, a relatively recent interface specification which aims at providing a compact API for parallel computations. It consists of one-sided communication routines, notifications-based synchronization, passive communication, global atomics and collective operations. It also defines groups (which are similar to MPI communicators and are used in collective operations) and the concept of segments. Segments are contiguous blocks of memory and can be made accessible (to read and write) to all threads on all ranks of a GASPI program.

GPI-2 is thus a communication library for C/C++ and Fortran based on one-sided communication. It adopts a PGAS-like model where each rank owns one or more memory segments which are globally accessible. Moreover, in GPI-2 all communication routines are thread-safe, allowing a more asynchronous and fine-grained multi-threaded execution as opposed to a bulk-synchronous communication with a single (master) thread, responsible for communication.

From an implementation point of view, GPI-2 aims at introducing a minimal overhead by providing a very thin layer, close to and exploiting hardware capabilities such as RDMA. One focus aspect is to provide truly asynchronous communication, that progresses in parallel as soon as it is triggered. This allows a better overlap of communication and computation, hiding the latency of communication.

Our GPI-2 based SpMVM is implemented in a task-based fashion, where a GPI-2 process (with the corresponding rank) is started per available NUMA socket. Within each rank a pool of POSIX threads is then used. Each thread dynamically polls for tasks to perform: this can be transferring data or computing a locally available part. This ensures that all threads are busy and that communication is overlapped and hidden behind the computation.

Note that such a task-based implementation is applicable to other kinds of large-scale scientific computations. Although it often requires a re-formulation of the algorithm, the attained benefits are considerable (as it will be demonstrated here). Below we provide more details about how is this achieved for the SpMVM kernel.

3 SpMVM with GPI-2

SpMVM is a memory–bounded routine; the SpMVM-kernels perform poorly, achieving $\sim 10\,\%$ from the theoretical peak performance [1], being far from reaching the theoretical speedup even on SMP-architectures. The principal problems related to the SpMVM performance are known (see [1] and the references therein): (i) restricted temporal locality as there is little data reuse, e.g. the matrix elements are 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.

The numerical treatment of (systems of) PDEs on hybrid architectures usually uses hierarchical decomposition: the coarse grained parallelism is attained by domain decomposition (DD), while the fine-grained parallelism on the node is achieved by thread parallelization. Each subdomain (SD) is mapped to a computational node, in our case this is a GPI-2 rank associated with a NUMA socket. The DD defines the distribution of the vector of unknowns (and of the rows of the sparse matrix for row-wise distribution) over the SDs, also the disposition of the discretization nodes at the subdomain interface which gives the topology of the inter-nodal exchange in SpMVM within the Krylov solvers. The resulting communication pattern depends on this topology (i.e., on the sparsity structure of the matrix) and is entirely irregular and problem-dependent. Note, that it is neither reasonable nor possible on each SD to keep a local replica of the full SpMVM input vector. One should copy locally only the remote items of the input vector needed on this SD, i.e. requested by the non-zero matrix elements, distributed on this SD. Our solution of this issue is to gather this topological information at the stages of mesh partitioning and discretization and to create, for each SD, a set of buffers to be written (lists of indices of the mesh nodes at the SD-interface). Then during execution, when the SpMVM routine is invoked, these buffers are used to perform the transfer of the remote input vector items.

Assuming a row-wise matrix and vector distribution, we designate the locally distributed matrix rows as $\mathbf{A}$, the full input vector as $\mathbf{X}$, and the local part of the output vector as $\mathbf{Y}_{\mathtt{lcl}}$. Thus, the SpMVM should calculate the expression $\mathbf{Y}_{\mathtt{lcl}} = \mathbf{A} * \mathbf{X}$ on each SD. A standard way to overlap communication and computation in SpMVM (see e.g. [5]) is to decompose $\mathbf{A}$ into: (i) a local part $\mathtt{A}_{\mathtt{lcl}}$, which multiplies the local part $\mathtt{X}_{\mathtt{lcl}}$ of the input vector $\mathtt{X}$, and (ii) its complementary matrix-chunk $\mathtt{A}_{\mathtt{rmt}}$, containing elements which multiply the "remote" part $\mathtt{X}_{\mathtt{rmt}}$ of the input vector. The elements of $\mathtt{X}_{\mathtt{rmt}}$ correspond to the mesh nodes at the interface of the neighbour SDs and should be locally transferred. Formally $\mathbf{X} = \mathtt{X}_{\mathtt{lcl}} + \mathtt{X}_{\mathtt{rmt}}$ holds and according to this decomposition the SpMVM operation can be written as:

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

The "standard" hybrid implementation of SpMVM usually uses a single "communication thread" per socket or node which runs an MPI-process and performs the inter-nodal exchange; the other threads are eventually "mapped" to it to access MPI, otherwise performing local computations to overlap the communication [5,6]. Our GPI-based SpMVM kernel uses the same idea but is differently organized; a brief sketch of it follows. Taking advantage of GPI-2, it uses task-based parallel, one-sided RDMA transfer of $\mathtt{X}_{\mathtt{rmt}}$ overlapped by computation:

(1) Some number of threads - say, as many as the number of neighbour SDs are - start independently transferring $\mathtt{X}_{\mathtt{rmt}}$, each thread communicating with one neigbour SD;
(2) All other threads start polling jobs to perform the local part $\mathtt{A}_{\mathtt{lcl}} * \mathtt{X}_{\mathtt{lcl}}$ in Eq. (1), where "job" means a subset of matrix rows to be multiplied. Note that the jobs are independent from each other;
(3) When the transfer of $\mathtt{X}_{\mathtt{rmt}}$ is over all threads start polling jobs from both the local and remote parts of the multiplication;
(4) Locally synchronize all threads and then perform the addition in Eq. (1).

Distinguishing features of our approach are: (i) the transfer of $\mathtt{X}_{\mathtt{rmt}}$ is task-based thread parallel; (ii) the multiplication in both local and remote parts of Eq. (1) is asynchronously parallel; (iii) independently on the matrix-sparsity pattern, the job-polling mechanism provides presumably a quasi-optimal dynamic load balancing, with no idle threads (but this feature should be further tested on different matrices); (iv) the threads are spawned in the beginning of the iterative solver routine and are joined at its very end - i.e., we do not have the usual thread fork/join overhead as in the MPI/OpenMP implementations. To shortly summarize: our task-based parallelization allows for effective communication/computation overlapping leading to a better performance.

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.

We apply two variants of Domain Decomposition (DD):

(i) Cutting planes approach (Z-slices): the cube is split via planes parallel to the (x,y)-coordinate plane, i.e. the cube is cut into subdomains (SDs) or slices perpendicular to the z-axis.
(ii) Graph partitioning using the METIS [10] library.

While METIS provides partitions of a quite high quality, the Z-slices approach is far from being optimal, because when the number of SDs increases (strong scaling) the thickness of each slice decreases and the communication/computation ratio gets higher, limiting scalability. On the other side, this DD approach is illustrative and appropriate for benchmarking and comparing different solvers.

5 Performance Results and Comments

The underlying architecture consist of computational nodes connected via FDR Infiniband, each node being composed of two Intel Xeon E5-2680v2 (IvyBridge) sockets, with 10 cores per socket and 64 GB RAM.

We compare our GPI-2 implementation vs. PETSc-3.4.4. linked against the Intel MPI and MKL libraries. The domain partitioning is identical in PETSc and in the GPI-2 cases: two SDs (with successive indices) are assigned to each physical node, both in the case of the Z-slices and the METIS-partitioning. Furthermore, in our case, when a SD is mapped to a GPI-2 rank, the discretization nodes belonging to it, are uniformly distributed over the computing threads. The distribution of the matrix rows over the GPI-2 ranks and then over the computing threads matches exactly this nodal distribution. In the case of PETSc, when two SDs have been assigned to a physical node, again all locally distributed mesh nodes are uniformly split over the MPI-processes running on this computing node.

Table 1. Problem Size 257^3, $||exact - appr||_C{}^{4000\ itrs.} = 5.129525e-1$

Physical nodes		1	2	4	8	16	32	60
GPI-nodes		2	4	8	16	32	64	120
Total cores/MPI-procs.		20	40	80	160	320	640	1200
DD-type: Z-sclices	PETSc, exec. time [s]	359	184	95	52	30	20	15
	GPI-2, exec. time [s]	214	109	55	27	16	12	10
DD-type: METIS	PETSc, exec. time [s]	358	181	91	47	26	16	13
	GPI-2, exec. time [s]	216	111	55	27	15	8	5

We use the CRS-formatted matrix storage. Our library contains several iterative solvers (`CG`, `BiCGstab`, etc.), but we have chosen the Richardson method as a benchmark: it allows for a fair comparison because the calculations performed in the GPI-Richardson and PETSc-Richardson routines are identical - this can be shown by monitoring the residual at each iteration. 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 obtain identical values for the current residual $L2$-norm $||\mathbf{b} - \mathbf{A} * \mathbf{x}||_{L2}$ and for the C-norm of the error $||exact - appr||_C$ (i.e., the C-norm of the difference between the exact and the numerical solutions).

We compare the execution times and the measured real speedup of GPI-2 based Richardson vs. PETSc-Richardson for two different problem sizes. The timings for the size 257^3 are presented in Table 1, while Fig. 1 depicts the obtained speedup (along with the ideal one) for the two DD-techniques we use and taking the execution on a single node as base.

Similarly, Table 2 contains the measurements for the size 351^3, with the obtained speedup presented in Fig. 2. On the finer mesh the convergence of

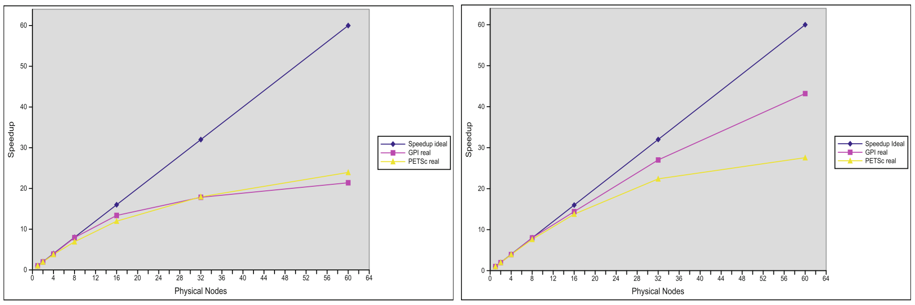

Fig. 1. Speedup GPI-2 vs. PETSc: Jacobi Preconditioned Richardson, 4000 itrs, size 257^3, partitioning using Z-slices (left) and METIS (right)

Table 2. Problem Size 351^3, $||exact - appr||_C{}^{4000\ itrs.} = 6.033188e - 1$

Physical nodes		1	2	4	8	16	32	60
GPI-2 ranks		2	4	8	16	32	64	120
Total cores/MPI-procs.		20	40	80	160	320	640	1200
DD-type: Z-sclices	PETSc, exec. time [s]	922	467	241	128	79	49	36
	GPI-2, exec. time [s]	566	282	148	76	37	24	21
DD-type: METIS	PETSc, exec. time [s]	918	460	233	117	61	33	22
	GPI-2, exec. time [s]	564	289	153	81	36	19	11

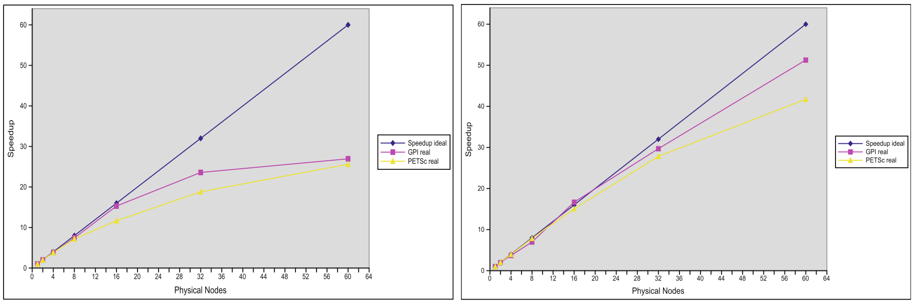

Fig. 2. Speedup GPI-2 vs. PETSc: Jacobi Preconditioned Richardson, 4000 itrs, size 351^3, partitioning using Z-slices (left) and METIS (right)

the Richardson method is slower and - after performing the same number of iterations - the difference with the exact solution is bigger.

In both cases the comparison has been done separately for our two types of DD. One easily sees that - independently of the type of partitioning - GPI-Richardson clearly outperforms PETSc, it is about twice faster, despite the fact that we use no hardware optimization (e.g. vectorization). Furthermore, although the inefficient Z-slices partitioning produces almost the same speedup for the two solvers, our GPI-2 version has shorter execution times.

About the partitioning one may say that compared to the Z-slices the METIS-DD is certainly more appropriate: it produces faster execution times starting from 8 (case 257^3) or 16 (case 351^3) physical nodes on. Using METIS-DD GPI-Richardson is not only faster but also scales better than PETSc-Richardson.

6 Conclusion

From an application point of view, a distinguishing property when working with GPI-2 is that it provides full freedom and flexibility to follow a task based parallelization. In this sense, the GPI-2 model meets the requirements and the challenges of the nowadays hierarchical architectures, proposing an alternative to both pure-MPI programming and the standard hybrid approaches with MPI and threads/OpenMP.

We have briefly sketched our GPI-2 implementation of the SpMVM kernel, which uses asynchronous communication and allows for fine-grained and better communication/computation overlap. We have used this kernel in a small library of Krylov subspace solvers. Using as a benchmark the Jacobi Preconditioned Richardson method to iterate the linear system arising after the discretization of a Poisson BVP in a unit cube, we have shown that our Richardson solver outperforms the Richardson solver of PETSc. We have confirmed this behaviour for two different types of domain decomposition: Z-slices-partitioning and graph partitioning with the METIS library. In the latter case, our version is not only faster than PETSc-Richardson but it also scales better.

As we noted, from a programming model point of view, conceptually similar implementations could bring performance advantages not only in SpMVM but - more generally - in the case of other DD-based parallelization approaches, e.g. additive Schwartz, where a truly asynchronous communication scheme could enable evident performance gains.

References

1. Gormas, G., et al.: Performance evaluation of the sparse matrix-vector multiplication on modern architectures. J. Supercomput. **50**, 36–77 (2009)
2. Gropp, W.D., Thakur, R.: Issues in developing a thread-safe MPI implementation. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 12–21. Springer, Heidelberg (2006)
3. Balaji, P., Buntinas, D., Goodell, D., Gropp, W.D., Thakur, R.: Toward efficient support for multithreaded MPI communication. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 120–129. Springer, Heidelberg (2008)
4. Hagger, G., Wellein, G.: Introduction to High Performance Computing for Scientists and Engineers. CRC Press, Boca Raton (2010)
5. Lange, M., Gorman, G., Weiland, M., Mitchell, L., Southern, J.: Achieving efficient strong scaling with PETSc using hybrid MPI/OpenMP optimisation. In: Kunkel, J.M., Ludwig, T., Meuer, H.W. (eds.) ISC 2013. LNCS, vol. 7905, pp. 97–108. Springer, Heidelberg (2013)
6. Schubert, G., Fehske, H., Hager, G., Wellein, G.G.: Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems. Parallel Process. Lett. **21**(3), 339–358 (2011)
7. http://www.gpi-site.com/gpi2/
8. http://www.gaspi.de/
9. http://openmp.org/wp/sc13-tutorial-hybrid-mpi-and-openmp-parallel-programming/
10. http://www-users.cs.umn.edu/karypis/metis/
11. http://www.mcs.anl.gov/petsc/