The EPiGRAM Project: Preparing Parallel Programming Models for Exascale

Stefano Markidis[1(✉)], Ivy Bo Peng[1], Jesper Larsson Träff[2], Antoine Rougier[2], Valeria Bartsch[3], Rui Machado[3], Mirko Rahn[3], Alistair Hart[4], Daniel Holmes[5], Mark Bull[5], and Erwin Laure[1]

[1] KTH Royal Institute of Technology, Stockholm, Sweden
s.markidis@gmail.com
[2] Vienna University of Technology (TU Wien), Vienna, Austria
[3] Fraunhofer ITWM, Kaiserslautern, Germany
[4] Cray UK, Edinburgh, UK
[5] Edinburgh Parallel Computing Center, Edinburgh, UK
http://www.epigram-project.eu/

Abstract. EPiGRAM is a European Commission funded project to improve existing parallel programming models to run efficiently large scale applications on exascale supercomputers. The EPiGRAM project focuses on the two current dominant petascale programming models, message-passing and PGAS, and on the improvement of two of their associated programming systems, MPI and GASPI. In EPiGRAM, we work on two major aspects of programming systems. First, we improve the performance of communication operations by decreasing the memory consumption, improving collective operations and introducing emerging computing models. Second, we enhance the interoperability of message-passing and PGAS by integrating them in one PGAS-based MPI implementation, called *EMPI4Re*, implementing MPI endpoints and improving GASPI interoperability with MPI. The new EPiGRAM concepts are tested in two large-scale applications, iPIC3D, a Particle-in-Cell code for space physics simulations, and Nek5000, a Computational Fluid Dynamics code.

1 Introduction

Exascale supercomputers will deliver 10^{18} floating-point operations per second (FLOPS) in double precision using the High Performance Linpack (HPL) benchmark. Today, the two fastest supercomputers are the Chinese *Sunway TaihuLight* and *Tianhe-2* supercomputers delivering respectively 93 and 33.8 petaFLOPS. The next American supercomputers from the *Corral* initiative will deliver between 100 and 200 petaFLOPS. Current projections of future supercomputers estimate the delivery of exascale machine in 2024.

While the race to exascale resulted in faster and larger supercomputers that today are only a factor of ten far from exascale, the software stack to support parallel applications on supercomputers has remained almost unchanged with

© Springer International Publishing AG 2016
M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS 9945, pp. 56–68, 2016.
DOI: 10.1007/978-3-319-46079-6_5

respect to the software present on petascale machines. In fact, almost all applications use MPI as main library to support parallel communication. Some of the applications use Partitioned Global Address Space (PGAS) languages, such as Coarray Fortran and UPC, with MPI [19,22]. Several applications employ OpenMP in combination with MPI for programming intra-node communication [13]. Despite the proposal of new disruptive programming models and systems [1,3,5], it is unlikely that such programming systems will reach a level of maturity and reliability to be readily deployed on the full exascale machine. The programming system dominating the exascale era will be MPI, possibly in combination with PGAS and OpenMP. For this reason, it is important both to improve the performance of these existing programming systems and to enhance their interoperability.

The size of next exascale supercomputers and their hardware pose difficult challenges to development of programming models. One of the main challenges is to handle the amount of parallelism that an exascale supercomputer will provide. The current fastest supercomputer *Sunway TaihuLight* provides 10,649,600 cores for parallel computation. Extrapolating this value to future more powerful supercomputers, it is reasonable to expect that an exascale machine will provide an 100 million-way parallelism. While communication cost decreases as part of communication operations is offloaded to the NIC and network technologies improve, still a large amount of memory is needed for storing process and communicator information on 100 million processes and memory footprint becomes a serious bottleneck [2]. In addition, collective operations and synchronization of such a large amount of processes [24] require the development and implementation of more sophisticated collective algorithms. A second main challenge is to guarantee that all the programming systems, such MPI, OpenMP and PGAS approaches efficiently interoperate sharing fairly all the hardware resources. It is therefore important to improve the performance of communication operations on a very large number of process potentially in presence of a combination of different programming systems.

Exascale ProGRAmming Models (EPiGRAM) is a European Commission funded project with the goal of addressing these exascale challenges in programming models. The EPiGRAM consortium consists of KTH Royal Institute of Technology, Vienna University of Technology (TU Wien), Fraunhofer ITWM, Cray UK, University of Edinburgh and University of Illinois (associate partner).

EPiGRAM focuses on the improvement of MPI and GASPI performance on exascale systems. MPI is currently the most used approach for programming parallel applications on supercomputers [11]. Global Address Space Programming Interface (GASPI) is the standard [12] for a PGAS API. GASPI uses one-sided Remote Direct Memory Access (RDMA) driven communication in combination with remote completion in a PGAS environment. Global address space Programming Interface (GPI) is a GASPI implementation, developed by Fraunhofer ITWM. Since GPI-2, Fraunhofer ITWM provides an open-source GPI implementation under GPL v3.

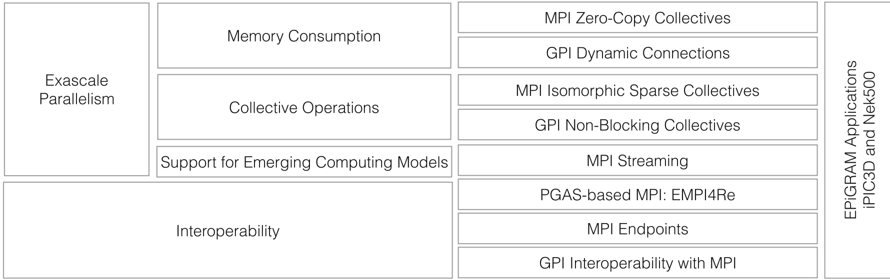

Fig. 1. Overview of the EPiGRAM project.

Figure 1 provides an overview of the different topics that have been investigated during the EPiGRAM project and the remaining part of the paper will describe these topics more in detail.

The paper is organized as follows. The second section presents the EPiGRAM work to address the challenge of exascale parallelism. In particular, we present how MPI and GASPI are further developed to reduce the memory footprint, to improve collective algorithms and implementations and to support emerging computing models. The third section describes the EPiGRAM research on MPI and GASPI interoperability, presenting MPI endpoints, GPI interoperability with MPI and the PGAS-based MPI implementation, called *EMPI4Re*, that integrates message-passing and PGAS in one framework. The fourth section presents the EPiGRAM applications. Finally, the fifth section concludes the paper summarizing the work done in EPiGRAM.

2 The *Exascale Parallelism* Challenge

Some important issues and obstacles that might prevent an effective use of MPI and GASPI programming systems on exascale machines are:

1. Memory-footprint and efficient memory usage. The available memory per core or even per (heterogeneous) shared-memory node will not, as was the case to a large extent in the past, scale linearly with the number of cores or nodes. Thus, implementations and specifications of MPI and GASPI functionalities must use sub-linear space per core or per node.
2. Algorithms and implementations for collective communication. Commonly used implementations often assume a fully connected network, and have relatively dense communication patterns. Better implementations, and in particular, new, space efficient algorithms for sparse collective communication and for collective communication on sparse networks are needed. In addition, current MPI interfaces for sparse collective communication are still limited.
3. Support for emerging computing models on massively parallel supercomputers. Computing models, such streaming models, lack of a convenient interface in MPI to run efficiently on large scale supercomputers. This might prevent the use of emerging computing models on exascale supercomputers.

2.1 Memory-Footprint and Efficient Memory Usage

The first issue EPiGRAM addresses is the memory consumption of MPI and GPI at exascale. This is done by designing and implementing *zero-copy* MPI collectives and GPI dynamic connections.

MPI Zero-Copy Collectives. The MPI datatype mechanism allows application programmers to describe the structure of the data to be communicated in a concise way [11]. In particular, non-consecutive layouts of data can be described as vectors, indexed or structured types, allowing a compact representation of complex layouts. The application programmer describes the layout of the data to be communicated, and the MPI library implementation performs the actual access of the data.

An efficient MPI implementation of data types can save memory copy operations. In fact, an explicit pack operation, implemented by the application programmer, copies data into some intermediate communication buffer, and then the MPI library may entail another copy of this buffer. This extra copy can be sometimes eliminated completely with datatypes, or in part for large data where pipelining may be applied by the library. In particular, the MPI datatype mechanism permits so-called *zero-copy* implementations, in which no explicit data movements are present in the application and all data access and manipulation are carried out implicitly by the MPI library implementation.

In EPiGRAM, we studied the design and implementation of different *zero-copy* collective operations, focusing on obstacles that might prevent the design and implementation of such operations in an efficient way. We have investigated the use of the derived datatype mechanism of MPI in the implementation of the classic all-to-all communication algorithm of Bruck et al. [4]. Through a series of improvements to the canonical implementation of the algorithm we gradually eliminated initial and final processor-local data reorganizations, culminating in a *zero-copy* version that contains no explicit, process-local data movement or copy operations [35,37]. In this case, all necessary data movements are carried out as part of the communication operations. We also showed how the improved algorithm can be used to solve irregular all-to-all communication problems. In particular, in EPiGRAM we used and implemented three new derived datatypes (`bounded_vector`, `circular_vector`, and `bucket`) that are not in MPI. On two supercomputers at the Vienna University of Technology, we experimentally compared the algorithmic improvements to the Bruck et al. algorithm when implemented on top of MPI, showing the zero-copy version to perform significantly better than the initial, straight-forward implementation. One of our variants has also been implemented inside `mvapich`, and we showed it to perform better than the `mvapich` implementation of the Bruck et al. algorithm for the range of processes and problem sizes where it is enabled. Details about this work are provided in [35,37].

However, we showed in EPiGRAM that current collective interfaces cannot support *zero-copy* implementations in all cases [36]. The problem is that the regular collective interfaces use a receive datatype to specify a per-process layout, whereas sometimes a different layout is needed for each process. Such

cases cannot be accounted for; only for applications using all-to-all communication, the required flexibility is provided in the form of the tedious, non-scalable `MPI_Alltoallw` operation. In [36] we show a simple, and in many cases backwards compatible and mostly non-intrusive solution to the problems in the form of slightly changed collective interfaces (all other communication interfaces would have to be reinterpreted in a similar way). The key to the solution is to separate the number of elements to be communicated from the overall structure of the data. The latter is described by a datatype; the former by an element count. The current MPI specification mixes these two concerns, leading to the problems discussed.

GPI Dynamic Connections. We have identified GPI memory consumption as one of the main aspects to be improved for large scale execution of GPI applications. The memory consumption is strongly related to the management of the communication infrastructure in GPI. The GASPI specification, that GPI implements, defines that the communication infrastructure should be either built during initialization or performed explicitly by the application. This can be set through a configuration parameter where the default value is `TRUE`. In this case, the communication infrastructure is built at start-up by default. In fact, details of the initialization of the GPI communication infrastructure are left to the implementation.

Before EPiGRAM, communication infrastructure was built-up statically in GPI. In this case, each computing node (a GPI rank) establishes a connection to all the other computing nodes during initialization. This results in an all-to-all communication topology. Despite this is acceptable on small scale and typical executions, the problem becomes evident when running large-scale GPI applications. For this reason, we have extended GPI to allow three modes of topology building: `GASPI_TOPOLOGY_NONE` where the application explicitly handles the infrastructure setup, `GASPI_TOPOLOGY_STATIC` where, as before, an all-to-all connection is established and `GASPI_TOPOLOGY_DYNAMIC` where connections are dynamically established as the first communication request between two nodes is performed. We were able to verify and measure the effects of such GPI extension during the Extreme Scale Workshop using the full SuperMUC iDataPlex supercomputer, consisting of 3,072 nodes, at the Leibniz Supercomputing Centre. The establishment of dynamic connections provides a much more efficient and scalable resource consumption in terms of memory footprint. Panel *a* of Fig. 2 presents the memory consumption (per rank) after initialization using static and dynamic connections. The non-scalable behavior of the GPI all-to-all connection is evident. We can now alleviate that using GPI dynamic connections.

2.2 Algorithms and Implementations for Collective Communication

The second issue EPiGRAM addresses is the performance of collective communication operations at exascale by investigating improved sparse collectives and studying non-blocking collectives in GPI.

MPI Isomorphic Sparse Collectives. The MPI specification has functionality for sparse collective communication where processes communicate with a subset of other processes in a local neighborhood. Sparse neighborhoods can be explicitly specified using the general graph topology functionalities or Cartesian topology [10]. However, both approaches have problems. In the first case, this mechanism is cumbersome to use, and the necessary collective communication and computation to create the local neighborhoods as well as the creation of a new, possibly reordered communicator can be very expensive. Alternatively, neighborhoods can be given implicitly with a Cartesian communicator. On Cartesian MPI communicators, neighborhood collective communication is possible with these implicit neighborhoods. Although neighborhood collective communication should be oblivious to how neighborhoods are set up, there are some differences between explicit and implicit neighborhoods in the MPI standard. For instance, non-existing neighbors are possible for Cartesian neighborhoods and buffer space needs to be calculated for such non-existing neighbors; this is neither possible nor allowed for graph topologies. On the other hand, graph topologies can associate weights with the graph edges (that may reflect communication costs in different ways and thus can permit better process mappings), but this is not possible for Cartesian topologies.

EPiGRAM provides a middle ground between these two approaches: a mechanism for structured, sparse collective communication with a much smaller overhead than the general graph topology mechanism but with more flexibility and expressivity than the Cartesian neighborhoods. In EPiGRAM, we introduced the concept of *isomorphic* sparse collective [33,34]: isomorphic sparse collective communication is a form of collective communication in which all involved processes communicate in small, identically structured neighborhoods of other processes. Isomorphic sparse collective communication is useful for implementing stencil and other regular, sparse distributed computations, where the assumption that all processes behave symmetrically is justified. The concept of isomorphic neighborhood extends and generalizes what is possible with the limited MPI Cartesian topologies.

In EPiGRAM, a library for isomorphic sparse collective communication has been implemented. The library supports the navigation and query functionality, creation of isomorphic neighborhoods (by attaching the neighborhood information to a Cartesian communicator), functions for using relative neighbor lists to set up MPI graph communicators, and sparse isomorphic collective operations of the `allgather`, `alltoall` and reduction types. The performance improvements that can be achieved by using *isomorphic* sparse collectives are presented in [33,34].

GPI Non-blocking Collectives. Non-blocking collectives have been recently introduced in MPI-3 as a mean to overlap communication and computation during collective operations [10,14]. In EPiGRAM, we have investigated the development of non-blocking collectives in GPI. Currently there are only two collective operations in GASPI: `gaspi_barrier` and `gaspi_allreduce`. Both collective operations in GASPI have a timeout argument that specifies after

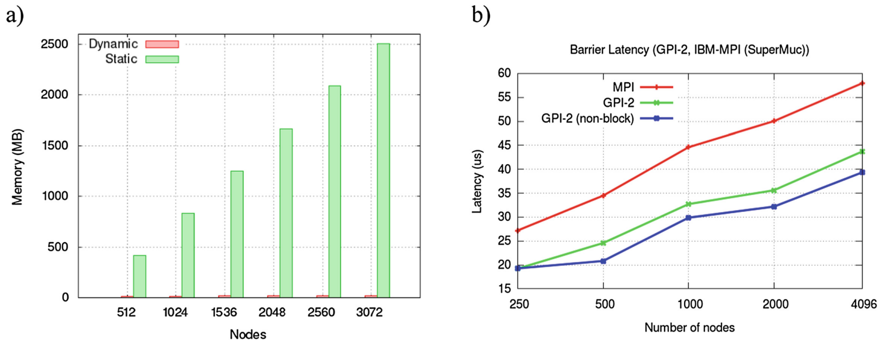

Fig. 2. Panel *a* shows the memory consumption using GPI static all-to-all connection (green bars) and dynamic connections (red bars) when using different number of computing nodes. Panel *b* shows the execution time for MPI blocking (red line), GPI blocking (green line) and non-blocking (blue line) barriers increasing the number of nodes. The tests have been performed on SuperMUC iDataPlex supercomputer at the Leibniz Supercomputing Centre. (Color figure online)

which period of time the collective operation can be interrupted. This timeout argument can be used as a form of non-blocking call. For instance, the time out argument can be set to `GASPI_TEST`. Each GPI process can call a collective function and when the function is interrupted as a consequence of the timeout other work can be performed, effectively implementing a non-blocking collective. In EPiGRAM, we added the support for this kind of non-blocking collectives and we performed performance tests of blocking and non-blocking GPI barriers and a comparison with MPI blocking barrier on the SuperMUC iDataPlex supercomputer up to 4,096 nodes (corresponding to a total of 65,536 cores) at the Leibniz Supercomputing Centre. Panel *b* of Fig. 2 presents the execution time for the different implementations of barriers, showing a reduced execution time for GPI non-blocking barrier (blue line) with respect to the execution of blocking GPI (green line) and MPI (red line) implementations.

2.3 MPI Support for Streaming Computing

EPiGRAM also investigated the support for emerging computing models that will be likely used on massively parallel supercomputers in the future. An example of such computing models is the data streaming computing model that is an effective way to tackle challenges from data-intensive applications. However, streaming computing is not naturally supported in MPI.

In EPiGRAM, we have designed and implemented a library called *MPIStream* [25] that allows HPC applications to globally allocate data producers and consumers on MPI processes, to stream data continuously or irregularly, to receive and process data and to terminate the streaming operations. Use cases of enabling HPC applications to carry out threshold collective operations, to

monitor and control applications and to perform parallel I/O of irregular events are illustrated in [25].

Our MPI streaming library targets the streaming model for distributed systems, where MPI is the dominant programming system. The *MPIStream* library is written in C and built on the top of MPI. A stream is a continuous flow of stream elements, which is the basic unit of transmission between data producer and streamer. MPI data types are used to describe the memory layout of the elements on data producers to achieve *zero-copy* streaming and consequently saving memory consumption on large systems. MPI persistent communication is used to reduce the overhead of repeatedly calling receive routines.

The performance of the *MPIStream* library has been evaluated using a parallel STREAM benchmark [25] on two supercomputers: Beskow Cray CX40 at KTH and Mira BlueGene/Q at the Argonne National Laboratory. The performance results show that the library can achieve acceptable performance (52 %–65 % of the maximum available bandwidth) and demonstrate its potential by reaching as high as 200 GB/s and 80 GB/s processing rate using 2,048 data producers over 2,048 data consumers on the Blue Gene/Q and Cray XC40 supercomputers respectively. Additional performance results of the *MPIStream* library are reported and discussed in [21,25].

3 The *Interoperability* Challenge

Interoperability of programming systems, such as MPI, OpenMP and PGAS, is a key aspect in exascale computing as it is likely that exascale applications will use a combination of programming systems to use efficiently different kind of communications, i.e. inter-node and intra-node communications. In EPiGRAM, we implemented a PGAS-based MPI to fully integrate message-passing and PGAS programming models, we introduced MPI endpoints in this MPI implementation and improved the GPI interoperability with MPI.

EMPI4Re: A PGAS-based MPI Implementation. EPiGRAM integrates and combines message-passing and PGAS programming models in one MPI implementation. The EPiGRAM MPI library for Research (EMPI4Re) is an MPI-1 library created by EPCC at the University of Edinburgh as a vehicle for research into new MPI functionality. The library adopts the conceptual model of PGAS and assumes hardware support for RDMA operations. This conceptual model enables efficient implementation of remotely accessible double buffered first-in first-out (FIFO) queues, used for point-to-point operations, and distributed state control structures, used for collective operations.

The code-base for the EMPI4Re library currently consists of 55,495 lines of C code (OpenMPI version 1.8.6 consists of 933,889 lines for comparison). The current implementation of EMPI4Re is based on DMAPP (a Cray one-sided communication API) [31] and there is an ongoing effort in EPiGRAM to replace DMAPP with GPI. Overall, we found in EPiGRAM that the EMPI4Re library is a useful research vehicle for rapidly prototyping and assessing code changes

to MPI functionality without the complexity of managing a large code-base or production MPI implementation.

MPI Endpoints. MPI is typically targeted at communication between distributed memory spaces. For a pure MPI programming approach, multi-core nodes require an OS process per core in order to take advantage of the available compute capability. This requires multiple instances of the MPI library per shared-memory node including communication buffers, topology informations and connection resources. Hybrid programming, commonly referred to as *MPI+X*, where *X* is programming model that supports threads, only requires a single instance of the MPI library per shared-memory node and so it should scale with increasing per-node core-count better than pure MPI. However, there are restrictions on how MPI can be used in multi-threaded OS processes that make it difficult to efficiently achieve high performance with hybrid programming. In particular, threads cannot be individually identified as the source or target of MPI messages [15].

MPI endpoints have been designed to remove or alleviate threading restrictions in MPI and facilitate high performance communication between multi-threaded OS processes. MPI endpoints allow the programmer to create additional ranks at each MPI process. Each endpoint rank can be then distributed to threads in system-level programming models enabling these threads to act as MPI processes and interoperate with MPI directly [6]. For an initial implementation study, see also [17,30].

The EPiGRAM project is implementing the MPI endpoints in EMPI4Re. The initial approach taken in the EMPI4Re library is to create each communicator handle as normal: generating a new structure for each one, including a full mapping of all ranks to their associated location. This is exactly what would happen if each of the members of the new communicator were individual MPI processes each in their own OS processes. EMPI4Re is already designed to be able to cope with each member of a communicator using a different context identifier for a particular communicator so this approach does not cause a conflict. The next step is to de-duplicate the internal data-structures so that multiple MPI endpoints in the same OS process share a single copy of the mapping information and share a reduced number of matching data-structures and communication buffers. In EMPI4Re, various design choices, such as having a different context identifier at each MPI process for a communicator thereby avoiding the use of a distributed agreement algorithm, simplify the addition of new features.

GPI Interoperability with MPI. Large parallel applications that have been developed over several years often reach several thousands or even millions of lines of code. Moreover, there is a large set of available libraries and tools which run with MPI. For this reason, it is important to enable GPI full cooperation with MPI so that both can be used simultaneously in an efficient way. This interoperability allows an incremental porting of large applications to GPI and an effective usage of existing MPI libraries and infrastructure. This tighter support for MPI interoperability was integrated in GPI during the EPiGRAM project (GPI release v1.1.0 in June 2014) by introducing the so-called *mixed-mode*.

In this mode, GPI sets its environment reusing MPI instead of relying on its own startup mechanism (`gaspi run`). The only constraint is that MPI must be initialized (`MPI_Init`) before GPI (`gaspi_proc_init`). In this mode, as MPI and GPI both follow a Single Program Multiple Data (SPMD) model, there is a direct match between the MPI the GPI ranks. This simplifies the reasoning about the hybrid GPI-MPI application.

In addition, an interface allowing memory management interoperability has been recently established in the GASPI standard [12]. GASPI handles memory spaces in so-called *segments*, which are accessible from every thread of every GASPI process. The GASPI standard has been extended to allow users to provide an already existing memory buffer as the memory space of a GASPI segment. This new function will allow future applications to communicate data from memory that is not allocated by the GASPI runtime system but provided to it, i.e. by MPI. If an MPI program calls GPI libraries, the GPI libraries need to be isolated, so that the communication in a library does not interfere with the communication in the main application, or any other library. This is required to guarantee correct results. The GASPI interface has been extended to offer a clear separation: a library is now able to create its own communication queues and thus have an isolated communication channel. For all other resources, i.e. segments, GPI already provides some mechanism to query their usage and to select an unused resource.

4 EPiGRAM Applications

The effectiveness of concepts that have been developed in EPiGRAM have been tested against two real-world open-source codes, iPIC3D [20,29] and Nek5000 [7]. iPIC3D is a massively parallel Particle-in-Cell code that is written in C++ and using MPI. Nek5000 is a semi-spectral Computational Fluid Dynamics (CFD) code for solving fluid dynamics problems, such as the study of turbulence arising on the surface of airplane wings. Nek5000 is written in large part in Fortran and in a small part in C and it uses MPI for parallel communication.

The EPiGRAM codes have been used for providing feedback to development of the programming systems in EPiGRAM: they have been employed to test new features in MPI and GASPI programming systems, to provide feedback to the developers of EMPI4Re library, and to compare the performance of the EPiGRAM programming system implementations in real-world applications [16].

The new EPiGRAM communication kernel of iPIC3D is now included in the release version of the code [29] and enabled large scale simulations of magnetospheric physics [23,26,27,32]. Together with the improvement of the communication kernels of the applications, also OpenACC/OpenMP porting of the applications to GPU systems [8,9,13,18,28] and new algorithmic strategies [38,39] have been implemented in EPiGRAM.

5 Conclusions

In summary, EPiGRAM is a European Commission project with the goal of improving the performance and the integration of existing parallel programming models. The EPiGRAM project focuses on the two current dominant petascale programming models, message-passing and PGAS, and on the improvement of two of their associated programming systems, MPI and GASPI. In EPiGRAM, we addressed two major exascale challenges: large-scale parallelism and interoperability of programming systems. First, we improve the performance of communication operations on a very large number of processes by decreasing their memory consumption, improving collective operations and introducing emerging computing models. Second, we enhance the interoperability of MPI and GPI by integrating message-passing and PGAS in one implementation, called *EMPI4Re*, implementing MPI endpoints and improving GPI interoperability with MPI. The new EPiGRAM concepts have been validated with experiments in two large-scale applications, iPIC3D, a Particle-in-Cell code for space physics simulations, and Nek5000, a CFD code.

Acknowledgments. This work was funded by the European Commission through the EPiGRAM (grant agreement no. 610598, www.epigram-project.eu) project.

References

1. Balaji, P.: Programming Models for Parallel Computing. MIT Press, Cambridge (2015)
2. Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Kumar, S., Lusk, E., Thakur, R., Träff, J.L.: MPI on a million processors. In: Ropo, M., Westerholm, J., Dongarra, J. (eds.) EuroPVM/MPI 2009. LNCS, vol. 5759, pp. 20–30. Springer, Heidelberg (2009). doi:10.1007/978-3-642-03770-2_9
3. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: expressing locality and independence with logical regions. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 66. IEEE Computer Society Press (2012)
4. Bruck, J., Ho, C.T., Kipnis, S., Upfal, E., Weathersby, D.: Efficient algorithms for all-to-all communications in multiport message-passing systems. IEEE Trans. Parallel Distrib. Syst. **8**(11), 1143–1156 (1997)
5. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the Chapel language. Int. J. High Perform. Comput. Appl. **21**(3), 291–312 (2007)
6. Dinan, J., Balaji, P., Goodell, D., Miller, D., Snir, M., Thakur, R.: Enabling MPI interoperability through flexible communication endpoints. In: Proceedings of the 20th European MPI Users' Group Meeting, pp. 13–18. ACM (2013)
7. Fischer, P.F., Lottes, J.W., Kerkemeier, S.G.: Nek5000 web page. mcs.anl.gov(2008). http://nek5000.mcs.anl.gov
8. Gong, J., Markidis, S., Laure, E., Otten, M., Fischer, P., Min, M.: Nekbone performance on GPUs with OpenACC and CUDA Fortran implementations. J. Supercomput. 1–21 (2016). doi:10.1007/s11227-016-1744-5D

9. Gong, J., Markidis, S., Schliephake, M., Laure, E., Henningson, D., Schlatter, P., Peplinski, A., Hart, A., Doleschal, J., Henty, D., Fischer, P.: Nek5000 with OpenACC. In: Markidis, S., Laure, E. (eds.) EASC 2014. LNCS, vol. 8759, pp. 57–68. Springer, Heidelberg (2015). doi:10.1007/978-3-319-15976-8_4
10. Gropp, W., Hoefler, T., Thakur, R., Lusk, E.: Using Advanced MPI: Modern Features of the Message-Passing Interface. MIT Press, Cambridge (2014)
11. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface, vol. 1. MIT Press, Cambridge (1999)
12. Grünewald, D., Simmendinger, C.: The GASPI API specification and its implementation GPI 2.0. In: 7th International Conference on PGAS Programming Models, vol. 243 (2013)
13. Hart, A.: First experiences porting a parallel application to a hybrid supercomputer with OpenMP4.0 device constructs. In: Terboven, C., Supinski, B.R., Reble, P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2015. LNCS, vol. 9342, pp. 73–85. Springer, Heidelberg (2015). doi:10.1007/978-3-319-24595-9_6
14. Hoefler, T., Lumsdaine, A., Rehm, W.: Implementation and performance analysis of non-blocking collective operations for MPI. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, 2007, SC 2007, pp. 1–10. IEEE (2007)
15. Ibrahim, K.Z., Yelick, K.: On the conditions for efficient interoperability with threads: an experience with PGAS languages using cray communication domains. In: Proceedings of the 28th ACM International Conference on Supercomputing, pp. 23–32. ACM (2014)
16. Ivanov, I., Gong, J., Akhmetova, D., Peng, I.B., Markidis, S., Laure, E., Machado, R., Rahn, M., Bartsch, V., Hart, A., et al.: Evaluation of parallel communication models in Nekbone, a Nek5000 mini-application. In: 2015 IEEE International Conference on Cluster Computing,. pp. 760–767. IEEE (2015)
17. Luo, M., Lu, X., Hamidouche, K., Kandalla, K., Panda, D.K.: Initial study of multi-endpoint runtime for MPI+ OpenMP hybrid programming model on multi-core systems. In: ACM SIGPLAN Notices, vol. 49, pp. 395–396. ACM (2014)
18. Markidis, S., Gong, J., Schliephake, M., Laure, E., Hart, A., Henty, D., Heisey, K., Fischer, P.: OpenACC acceleration of the Nek5000 spectral element code. Int. J. High Perform. Comput. Appl. **29**(3), 311–319 (2015)
19. Markidis, S., Lapenta, G.: Development and performance analysis of a UPC particle-in-cell code. In: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model, p. 10. ACM (2010)
20. Markidis, S., Lapenta, G.: Rizwan-uddin: multi-scale simulations of plasma with iPIC3D. Math. Comput. Simul. **80**(7), 1509–1519 (2010)
21. Markidis, S., Peng, I.B., Iakymchuk, R., Laure, E., Kestor, G., Gioiosa, R.: A performance characterization of streaming computing on supercomputers. Procedia Comput. Sci. **80**, 98–107 (2016)
22. Mozdzynski, G., Hamrud, M., Wedi, N., Doleschal, J., Richardson, H.: A PGAS implementation by co-design of the ECMWF integrated forecasting system (IFS). In: High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion, pp. 652–661. IEEE (2012)
23. Olshevsky, V., Deca, J., Divin, A., Peng, I.B., Markidis, S., Innocenti, M.E., Cazzola, E., Lapenta, G.: Magnetic null points in kinetic simulations of space plasmas. Astrophys. J. **819**(1), 52 (2016)
24. Peng, I.B., Markidis, S., Laure, E.: The cost of synchronizing imbalanced processes in message passing systems. In: 2015 IEEE International Conference on Cluster Computing, pp. 408–417. IEEE (2015)

25. Peng, I.B., Markidis, S., Laure, E., Holmes, D., Bull, M.: A data streaming model in MPI. In: Proceedings of the 3rd Workshop on Exascale MPI, p. 2. ACM (2015)
26. Peng, I.B., Markidis, S., Laure, E., Johlander, A., Vaivads, A., Khotyaintsev, Y., Henri, P., Lapenta, G.: Kinetic structures of quasi-perpendicular shocks in global particle-in-cell simulations. Phys. Plasmas (1994-Present) **22**(9), 092109 (2015)
27. Peng, I.B., Markidis, S., Vaivads, A., Vencels, J., Amaya, J., Divin, A., Laure, E., Lapenta, G.: The formation of a magnetosphere with implicit particle-in-cell simulations. Procedia Comput. Sci. **51**, 1178–1187 (2015)
28. Peng, I.B., Markidis, S., Vaivads, A., Vencels, J., Deca, J., Lapenta, G., Hart, A., Laure, E.: Acceleration of a particle-in-cell code for space plasma simulations with OpenACC. In: EGU General Assembly Conference Abstracts, vol. 17, p. 1276 (2015)
29. Peng, I.B., Vencels, J., Lapenta, G., Divin, A., Vaivads, A., Laure, E., Markidis, S.: Energetic particles in magnetotail reconnection. J. Plasma Phys. **81**(02), 325810202 (2015)
30. Sridharan, S., Dinan, J., Kalamkar, D.D.: Enabling efficient multithreaded MPI communication through a library-based implementation of MPI endpoints. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 487–498. IEEE Press (2014)
31. Ten Bruggencate, M., Roweth, D.: DMAPP - an API for one-sided program models on Baker systems. In: Cray User Group Conference (2010)
32. Tóth, G., Jia, X., Markidis, S., Peng, I.B., Chen, Y., Daldorff, L.K., Tenishev, V.M., Borovikov, D., Haiducek, J.D., Gombosi, T.I., et al.: Extended magnetohydrodynamics with embedded particle-in-cell simulation of Ganymede's magnetosphere. J. Geophys. Res. Space Phys. **121**, 1273–1293 (2016)
33. Träff, J.L., Carpen-Amarie, A., Hunold, S., Rougier, A.: Message-combining algorithms for isomorphic, sparse collective communication. arXiv preprint arXiv:1606.07676 (2016)
34. Träff, J.L., Lübbe, F.D., Rougier, A., Hunold, S.: Isomorphic, sparse MPI-like collective communication operations for parallel stencil computations. In: Proceedings of the 22nd European MPI Users' Group Meeting, p. 10. ACM (2015)
35. Träff, J.L., Rougier, A.: MPI collectives and datatypes for hierarchical all-to-all communication. In: Proceedings of the 21st European MPI Users' Group Meeting, p. 27. ACM (2014)
36. Träff, J.L., Rougier, A.: Zero-copy, hierarchical gather is not possible with MPI datatypes and collectives. In: Proceedings of the 21st European MPI Users' Group Meeting, p. 39. ACM (2014)
37. Träff, J.L., Rougier, A., Hunold, S.: Implementing a classic: zero-copy all-to-all communication with MPI datatypes. In: Proceedings of the 28th ACM International Conference on Supercomputing, pp. 135–144. ACM (2014)
38. Vencels, J., Delzanno, G.L., Johnson, A., Peng, I.B., Laure, E., Markidis, S.: Spectral solver for multi-scale plasma physics simulations with dynamically adaptive number of moments. Procedia Comput. Sci. **51**, 1148–1157 (2015)
39. Vencels, J., Delzanno, G.L., Manzini, G., Markidis, S., Peng, I.B., Roytershteyn, V.: SpectralPlasmaSolver: a spectral code for multiscale simulations of collisionless, magnetized plasmas. J. Phys. Conf. Ser. **719**, 012022 (2016). IOP Publishing