Optimizations to enhance sustainability of MPI applications

Jesus Carretero, Javier Garcia-Blas, David E. Singh, Florin Isaila
University Carlos III of Madrid, Spain

Thomas Fahringer, Radu Prodan
University of Innsbruck, Austria

George Bosilca
University of Tennessee, USA

Alexey Lastovetsky
University College Dublin

Christi Symeonidou
ICS, FORTH, Greece

Horacio Perez-Sanchez, Jose M. Cecilia
Universidad Católica de Murcia

ABSTRACT

Ultrascale computing systems are likely to reach speeds of two or three orders of magnitude greater than today's computing systems. However, to achieve this level of performance, we need to design and implement more sustainable solutions for ultra-scale computing systems, at both the hardware and software levels, while understanding sustainability in a holistic manner in order to address challenges in economy-of-scale, agile elastic scalability, heterogeneity, programmability, fault resilience, energy efficiency, and storage. Some solutions could be integrated into MPI, but others should be devised as higher level concepts, less general, but adapted to applicative domains, possibly as programming patterns or libraries. In this paper, we layout some proposals to extend MPI to cover major relevant domains in a move towards sustainability, including: MPI programming optimizations and programming models, resilience, data management, and their usage for applications.

Categories and Subject Descriptors

D.1.3 [**Software**]: [Concurrent Programming parallel programming]; C.1.4 [**Computer Systems Organization**]: [Parallel distributed architectures]

Keywords

Sustainability, Parallel architectures, MPI, Optimizations

1. INTRODUCTION

The interest of governments, industry, and researchers in very large scale computing systems has significantly increased in recent years, and steady growth of computing infrastructures is expected to continue in data centers and supercomputers due to the ever-increasing data and processing requirements of various domain applications, which are constantly pushing the computational limits of current computing resources. However, it seems that we have reached a point where system growth can no longer be addressed in an incremental way, due to the huge challenges lying ahead. In particular scalability, energy barrier, data management, programmability, and reliability all pose serious threats to tomorrow's cyberinfrastructure.

The idea of an Ultrascale Computing Systems (UCS), envisioned as a large-scale complex system joining parallel and distributed computing systems that cooperate to provide solutions to the users might be one solution to these growing problems at scale. As all the above models rely on distributed memory systems, the Message-Passing Interface (MPI) remains a promising paradigm to develop and deploy parallel applications, and it is already proven at larger scale—with machines running 100K processes. However, can we be sure that MPI will be sustainable in Ultrascale systems? If we understand sustainability as the probability that today's MPI functionality will be useful, available, and improved in the future, the answer is "yes." MPI behaves as a portability layer between the application developer and the hardware resources, hiding most architectural details from application developers. The independence from the computing platform has allowed new versions of MPI to include features that, when carefully combined with other libraries and integrated into dynamic high-level programming paradigms, permit the development of adaptable applications and novel programming paradigms, molding themselves to the scale of the underlying execution platform.

However, we will need to design and implement more sustainable solutions for Ultrascale computing systems, understanding sustainability in a holistic manner to address challenges like economy-of-scale, agile elastic scalability, heterogeneity, programmability, fault resilience, energy efficiency, and scalable storage. Some of those solutions could be integrated and provided by MPI, but others should be devised as higher level concepts, less general, but adapted to applicative domains, possibly as programming patterns or libraries. In this paper, we layout some proposals to extend MPI to cover major relevant domains in a move towards sustainability, including: MPI programming optimizations and programming models, resilience, data management, and

EuroMPI/ASIA '14, September 9-12 2014, Kyoto, Japan
Copyright 2014 ACM 978-1-4503-2875-3/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642769.2642797.

their usage for applications.

The remainder of this paper is organized as follows. Section 2 covers communication optimizations, while Section 3 addresses the area of resilience. Section 4 talks about storage and I/O techniques, Section 5 deals with energy constraints, and Section 6 presents some application and algorithm optimizations. Finally, we conclude in Section 7.

2. ENHANCING MPI RUNTIME AND PROGRAMMING MODELS

As the scale and complexity of systems increases, it is becoming more important to provide MPI users with optimizations and programming models to hide this complexity, while providing a mechanism to expose part of this information for application developers seeking knowledge of low-level functions. One possible way to achieve automatic application optimization is to provide a layered API and allow a compiler tool to convert between MPI and this layered API, as necessary. Another potential approach would involve more efficiently integrating new programming models (e.g., OpenMP or PGAS) for cooperatively sharing not only a common high-level goal—such as a view of the application's time-to-completion—but all resources of the targeted platform. In this section, we focus on some optimizations shown to enhance MPI's scalability and performance. These optimizations provide minimum APIs to transparently enhance portability and sustainability of application software, thus minimizing the adaptation effort.

2.1 Distributed Region-based memory Allocation and Synchronization

Even though the existing distributed global address memory models, such as PGAS, support global pointers, their potential efficiency is hindered by the expensive and unnecessary messages generated by global memory accesses. In order to transfer their data among nodes, they must either marshal and un-marshal their data during the communication, or be represented in a non-intuitive manner.

DRASync [3] is a region-based allocator that implements a global address space abstraction for MPI programs with pointer-based data structures. Regions are a collection of contiguous memory spaces used for storing data. They offer great locality since similar data can be placed together and can be easily transmitted in bulk. DRASync offers an API for creating, deleting, and transferring such regions. It enables MPI processes to operate on a region's data by acquiring the containing region and releasing it at the end of computation for other processes to acquire. Each region is combined with ownership semantics, allowing the process that created it, or one that acquired it, to have exclusive write permissions to its data. DRASync, however, does not restrain other MPI processes, that are not owners, from acquiring read-only copies of the region. Thus, acquire/release operations are akin to reader-writer locks and enable DRASync to provide an intuitive synchronization tool that simplifies the design of MPI applications.

DRASync has been evaluated over the Myrmics [10] allocator using two application-level benchmarks, the Barnes-Hut N-body simulation and the Delaunay triangulation with variant datasets. The encouraging outcome highlighted the fact that DRASync produces comparable performance results while providing a more intuitive synchronization abstraction for programmers.

2.2 Optimization of MPI collectives

Algorithms for MPI collective communication operations typically translate the collective communication pattern as a combination of point-to-point operations in an overlay topology, mostly a tree-like structure. The traditional targets for such an algorithmic deployment are homogeneous platforms with identical processors and communication layers. When applied to heterogeneous platforms, these implementations may be far from optimal, mainly due to the uneven communication capabilities of the different links in the underlying network. In [4], we proposed to use heterogeneous communication performance models and their prediction to find more efficient, almost optimal, communication trees for collective algorithms on heterogeneous networks. The models take into account the heterogeneous capabilities of the underlying network of computers when constructing communication trees. Model predictions are used during the dynamic construction of communication trees either by changing the mapping of the application processes or changing the tree structure altogether. Experiments on Grid5000 using 39 nodes geographically distributed over 5 clusters stretched over 2 sites, demonstrate that the proposed model-based algorithms clearly outperform their non-model-based counterparts on heterogeneous networks (see Fig. 1).

2.3 MPI communications with adaptive compression

Adaptive-CoMPI [5] is an MPI extension which performs the adaptive message compression of MPI-based applications to reduce communication volume, and thus time, and enhances application performance. It is implemented as a library connected through the Abstract Device Interface of MPICH so that it can be used with any MPI-based application in a transparent manner, as the user does not need to modify the source code.

Adaptive-CoMPI includes two different compression strategies. The first one, called *Runtime Adaptive Strategy* analyzes the performance of the communication network and the efficiency of different compression algorithms before the application execution. Based on this information, during the application execution, it decides if it is worth it to compress a message or not, and if so, it chooses the most appropriate compression algorithm. This feature allows the Runtime Adaptive Strategy to offer adaptive compression capabilities without any previous knowledge of the application characteristics. The second approach, called *Guided Strategy*, provides an application-tailored solution based on the prior application analysis using the application profiling. With this approach the MPI application is executed first and all the messages are stored in a log file. Upon completion, the best compression algorithm is determined for each message and it is registered in a *decision rules* file. When the application is executed again with the same input parameters, Adaptive-CoMPI extracts the information from the *decision rules* file and applies the most appropriate compression technique for each message.

Adaptive-CoMPI has been evaluated using real applications (BIPS3D, PSRG, and STEM), as well as using the NAS benchmarks. Figure 2 shows the speedup achieved for each strategy compared to the execution of the application without compression. Note that the Guided Strategy finds

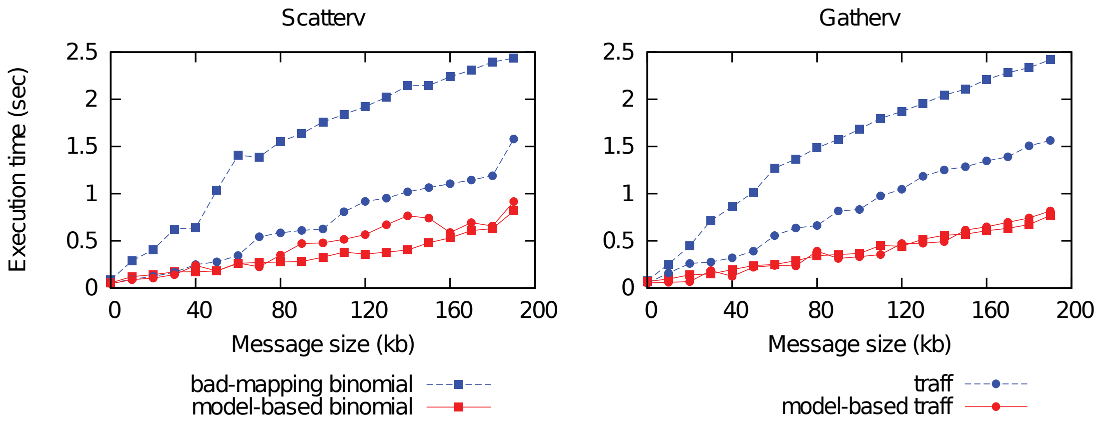

Figure 1: Scatterv and Gatherv operations on geographically distributed clusters from Grid5000.

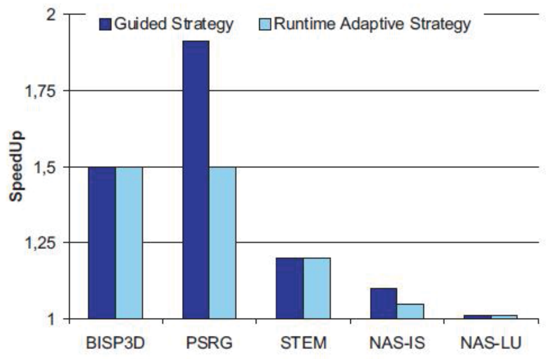

Figure 2: Speedup of Adaptive-CoMPI by using 16 processors.

the best compression technique (including no compressing) for each message, providing the optimal compression rate for each independent message. We can observe that the Runtime Adaptive Strategy obtains a performance similar to the guided one which means that, globally speaking, it is able to efficiently compress the messages with no previous knowledge of the application.

3. RESILIENCE

Should the number of components in supercomputers continue to increase, the mean time between failures is expected to decrease to a handful of hours. As a result, deploying fault tolerant strategies within HPC software would result in a massive improvement in application runtime and efficient resource usage compared to currently deployed techniques used to alleviate the consequence of failures (that is, resubmission of failed jobs, and simplistic periodic checkpointing to disk). However, checkpoint/restart is unable, in its current state, to cope with very adversarial future failure patterns. This presents a clear need for improving checkpointing strategies by 1) permitting optimized checkpoint storage that does not rely on centralized I/O; and 2) permitting independent restart of failed processes without rollback of the same processes [2]. Algorithm based fault tolerant techniques can even forgo checkpointing completely by employing a tailored, scalable protective strategy to maintain sufficient redundancy to restore lost data pieces due to failures. On another front, there are many applications, like domain decomposition, naturally fault tolerant applications, and master-worker, in which the partial loss of the dataset is not a catastrophic event that commands interrupting progress toward the solution. All of these recovery patterns hit one of the historic roadblocks that have hindered the deployment of fault tolerant software: the lack of proper support from the popular MPI communication stack, which thereby limits recovery options to full-job restart upon failure.

Resiliency should refer not only to the ability of the MPI application to be restarted after a failure, but also to the ability to survive failures and to recover to a consistent state from which the execution can be resumed. In recent developments, the MPI Forum has proposed an extension of the MPI standard that permits restoring the capability of MPI to communicate after failures strike [1]. One of the most strenuous challenges is to ensure that no MPI operation stalls from the consequences of failures, as fault tolerance is impossible if the application cannot regain full control of the execution. In the proposed standard, an error is returned when a failure prevents a communication from completing. However, it indicates only the local status of the operation, and does not permit deducing if the associated failure has impacted MPI operations at other ranks. This design choice avoids expensive consensus synchronizations from obtruding into MPI routines, but leaves open the danger of some processes proceeding unaware of the failure. The new is a low level layer, basically the most basic portability layer, that can be used to build higher level concepts, less general, but adapted to specific applicative domains. Thus, one possible solution is to put the resolution of such situations under the control of the application programmer, by providing supplementary interfaces that reconstruct a consistent global view of the application state (typical case for applications with collective communications). Aside from applications, these new interfaces can be used by high-level abstractions, such as transactional fault tolerance, uncoordinated checkpoint-restart, and programming languages, to support advanced fault tolerance models that are thereby portable between MPI implementations.

4. DATA AND INPUT/OUTPUT

Data storage and management is a major concern for Ultrascale systems, as the increased scale of the systems and the data demand from the applications leads to major I/O overheads that are actually hampering the performance of the applications themselves. MPI has proposed asynchronous I/O operations to allow overlapping I/O and computation, but this feature does not reduce the latency of the system, which is inherent in the length of the I/O path. To this end, there is a major trend towards increasing data locality to avoid data movements—a data-centric paradigm.

In this sense, AHPIOS (Ad-Hoc Parallel I/O system for MPI applications)[8] proposes a scalable parallel I/O system completely implemented in the Message Passing Interface (MPI). AHPIOS allows MPI applications to dynamically manage and scale distributed partitions in a convenient way. The configuration of both the MPI-IO and the storage management system is unified and allows for a tight integration of the optimizations of all layers. AHPIOS partitions are elastic as they conveniently scale up and down with the number of resources. The AHPIOS proposes two collective I/O strategies, which leverage a two-tiered cooperative cache in order to exploit the spatial locality of data-intensive parallel applications. The file access latency is hidden from the applications through an asynchronous data staging strategy. The two-tiered cooperative cache scales with both the number of processors and storage resources. The first cooperative cache tier runs along with the application processes and hence scales with the number of application processes. The second cooperative cache tier runs at the I/O servers and, therefore, scales with the number of global storage devices.

Given an MPI application accessing files through the MPI-IO interface and a set of distributed storage resources, AHPIOS constructs a distributed partition on demand, which can be accessed transparently and efficiently. Files stored in one AHPIOS partition are transparently striped over storage resources, each partition being managed by a set of storage servers running together as an independent MPI application. Access to an AHPIOS partition is performed through an MPI-IO interface, allowing it to scale up and down on demand during run-time.

The performance and scalability of AHPIOS for an MPI application that write and read in parallel, disjoint, contiguous regions of a file, stored over an AHPIOS system for different numbers of AHPIOS servers, has been demonstrated on a Blue-Gene supercomputer. Figure 3 shows the aggregate I/O throughput for n MPI processes writing and reading to/from an AHPIOS partition with n AHPIOS servers. The figure shows two scenarios: AHPIOS servers run on the same nodes as the MPI processes; MPI processes and AHPIOS servers run on disjoint sets of nodes. The figure represents the throughput to the AHPIOS servers. We can see that the file access performance scales well with the partition size, independently of the location of AHPIOS servers.

5. ENERGY

Energy became a major concern for the sustainability of future computer architectures. Providing MPI applications of malleable capabilities is a possible technique to enhancing energy efficiency of applications, as shown in this section. FLEX-MPI [9] is an MPI extension which provides performance-aware dynamic reconfiguration capabilities to MPI applications. FLEX-MPI uses the PAPI library to survey the number of FLOP and execution times of each MPI process, and of the MPI interface, to collect the MPI communication data (data type, size of the data transferred, and time spent). Based on this collected data, it decides to spawn or remove processes in order to achieve a user-defined performance objective. In case the number of processes changes, Flex-MPI also includes functionalities for performing the data redistribution, thereby guaranteeing appropriate load balance among the processes.

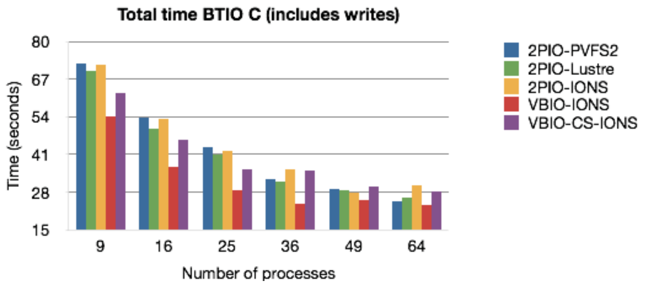

Figure 3: AHPIOS. BTIO class C measurements. ROMIO two-phase I/O over PVFS2, Lustre, AHPIOS and the two AHPIOS-based solutions: server-directed I/O and client-directed I/O.

There are two different objectives: cost and energy efficiency. The cost objective consists of reaching a given level of application performance (in FLOP/s) at the smallest cost. In the case of the energy efficiency objective, we aim to obtain the smallest energy cost under a given performance constraint. For reaching these objectives, we employ a computational prediction model that takes into account both the application and platform characteristics. In the case of energy saving, the goal is to free as many nodes as possible, dynamically at run-time, while maintaining a predefined performance goal.

The results are encouraging, as was demonstrated by executing a Jacobi method to solve a linear equation system with 20,000 variables and a performance improvement objective of 35%, using a heterogeneous platform with four classes of nodes (C8, C6, C7 and C1), each one with different energy cost. For this application, with an energy efficiency goal, Flex-MPI schedules more dynamic processes on the nodes which have more computing power but a better PUE ratio, achieving an average of 14 computational nodes—compared to the almost 30 nodes used in the static standard schedule. In this method, the overall operational energy cost is minimized. This result demonstrates that malleability at runtime would be a good option for achieving energy efficiency in MPI systems.

6. APPLICATIONS AND ALGORITHMS OPTIMIZATIONS

As a library, MPI lacks knowledge about the expected behavior of the whole application, the so called“global-view programming model,” which prevents certain optimizations that would be possible otherwise. In this section, we show some optimizations that are effective at the global level, and are thus proposed for the application level.

6.1 Hierarchical SUMMA

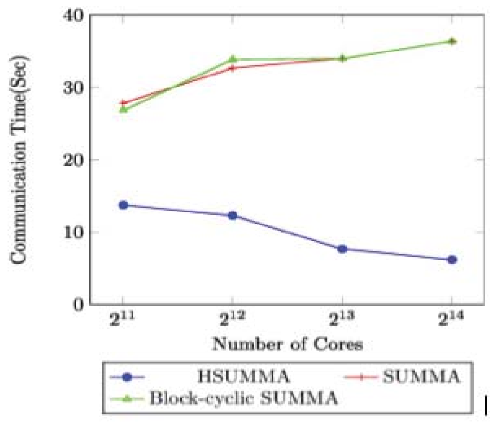

Figure 4: Communication time of SUMMA, block-cyclic SUMMA and HSUMMA on BG/P. p =16K, n = 65,536.

MPI collectives are very important building blocks for many scientific applications. In particular, MPI broadcast is used in many parallel linear algebra kernels such as matrix multiplication, LU factorization, and so on. The state-of-the-art broadcast algorithms used in the most popular MPI implementations were designed in mid 1990s with relatively small parallel systems in mind. Since then, the number of cores in high-end HPC systems has increased by three orders of magnitude and is going to further increase as the systems approach Ultrascale. While some platform-specific algorithms were proposed later on, they do not address the issue of scale, as they try to optimize the traditional general-purpose algorithms for different specific network architectures and topologies. The first attempt to address the issue of scale is made in [7], where the authors challenge the traditional "flat" design of collective communication patterns in the context SUMMA, the state-of-the-art parallel matrix multiplication algorithm. They transform SUMMA by introducing a two-level virtual hierarchy into the two-dimensional arrangement of processors. They theoretically prove that the transformed Hierarchical SUMMA (HSUMMA) can significantly outperform SUMMA on large-scale platforms. Their experiments on 16K cores have demonstrated almost a 6x improvement in communication cost, which translated into more than 2-fold speedup of HSUMMA over SUMMA (see Fig. 4). Moreover, the optimization technique developed is not architecture or topology specific. While the authors aim to minimize the total communication cost of this application rather than the cost of the individual broadcasts, it has become evident that, despite being developed in the context of a particular application, the resulting technique is not application-bound, ensuring sustainability.

6.2 Application-level optimization of MPI applications with Compiler Support

Programming in MPI requires the programmer to write code from the point-of-view of a single processor/thread, an approach known as fragmented programming. One limiting factor for optimizing MPI is the fact that it is a pure library approach and thus only effective during the execution of the application. A lot of effort is put into improving the performance of individual functions offered by MPI implementations in order to speed up the execution of MPI applications. These optimizations cannot be performed at the application level because the structure of the underlying program cannot be analyzed or changed by the MPI library in any way. On the other hand, normal compilers have no knowledge about the semantics of MPI function calls either, and thus have to treat them like black boxes—just like all library calls. A compiler which is aware of the semantics of MPI function semantics could (at least to some extent) analyze the behavior of a program along with its communication pattern, in order to optimize both.

We intend to optimize MPI applications by integrating MPI support in the Insieme compiler project [6]. The Insieme compiler framework enables the analysis of a given parallel application and applies source-to-source transformations to improve the overall performance. The output code of the compiler is intended to run within the Insieme runtime system, which provides basic communication primitives optimized for performance. The combination of a compiler and runtime system enables us to transform the program at compile time and also pass information about the program structure to the runtime system for further optimizations during program execution.

Optimizing message passing programs using specialized compilers has already been done long before MPI even existed. Moving communication calls within the code and replacing blocking with non-blocking communication can improve the communication/computation overlap and thus reduce the program execution time. Our approach should go one step further than previous MPI-aware compilers by analyzing high level patterns to find further optimization potential. An example illustrating such a pattern is depicted in the following code:

```
{
   MPI_Bcast(A, count, MPI_INT, 0, MPI_COMM_WORLD);
   for (int i = 0; i < count; i++) {
     // process A
   }
   MPI_Bcast(B, count, MPI_INT, 0, MPI_COMM_WORLD);
   // process B similarly
}
```

An MPI-aware compiler could change the second call from MPI_Bcast to MPI_Ibcast, and thus send B asynchronously while the application is processing the data transmitted during the first broadcast, A. Additionally, our compiler can detect that, under some constraints, it would be beneficial to combine both broadcast operations into a single operation to reduce the communication overhead for small messages. Similarly, for larger messages it might decide to break down the message transfers into smaller chunks which will then be processed individually, creating a pipelined broadcast at the application level, as shown below. Transformations like these require program analysis in a compiler and simply cannot be done with a pure library approach.

```
{
  for (offset = 0; offset < count; offset += tile_size) {
    MPI_Bcast(&A[offset], tile_size, MPI_INT, 0,
              MPI_COMM_WORLD);
    for (i = offset; i < offset + tile_size; i++) {
      // process tile of A
    }
  }
  //process remainder of A and do the same for B
}
```

6.3 Hybrid MPI-OpenMP Implementations

A hybrid programming solution might be implemented using OpenMP and MPI. Such approaches become more important on modern multi-core parallel systems, decreasing unnecessary communications between processes running on the same node, as well as, decreasing the memory consumption, and improving the load balance. With this implementation, both levels of parallelism, distributed and shared-memory, can be exploited. On one hand, the block-level parallelism is matched by the parallelism between nodes in the cluster (the data is distributed by using MPI). On the other hand, threads cooperate in parallel to perform the calculations within each node in a vectorized fashion. Once the data has been distributed using MPI, the calculation of the energy is performed on each node with OpenMP, using its own memory and executing as many threads as the number of cores per node. Moreover, the communication and computation can be overlapped by asynchronous send/receive instructions. These hybrid solutions are adequate when the kernels are computationally intensive and massively parallel in nature, and thus they are well suited to be accelerated on parallel architectures. A natural evolution can also be made with many-core systems located on each node.

One application example is the discovery of new drugs using Virtual Screening (VS) methods, where the calculation of the non-bonded interactions plays an important role, representing up to 80% of the total execution time. MPI can be used to move molecule related data between nodes, instead of sending all the information to each core. The communication is reduced by a *ratio of number of cores* per node, with respect to the MPI implementation. This hybrid distributed memory system exhibits good scalability with the number of processors, which is explained by the low number of communications required by our simulations in the hybrid MPI-OpenMP implementation. The hybrid optimized version reaches up to a 229x speed-up factor, versus its sequential counterpart [11].

7. CONCLUSIONS

The MPI design and its different implementations have proven to be a critical piece of the roadmap to faster and more scalable parallel applications. Based on it's past successes, MPI will probably remain a major paradigm for programming distributed memory systems. However, in order to maintain a consistent degree of performance and portability, the revolutionary changes we witness at the hardware level must be mirrored at the software level. Thus, the MPI standard must be in a continuous state of re-examination and re-factoring, to better bridge high-level software constructs with the low-level hardware capability. As software researchers, we need to highlight and explore innovative and even potentially disruptive concepts and match them to alternative, faster, and more scalable algorithms.

In this paper, we have called attention to some MPI-level optimizations that are amenable to providing sustainable support to parallel applications. Hybrid programming models allow developers to use MPI as the upper level distribution mechanism, thus reducing the volume of communication and the memory needed. Adaptive compression allows developers to reduce MPI communications and storage overhead, while AHPIOS is aimed at increasing data locality and reducing I/O latency. Most of the proposals made are transparent to applications or can be made transparent through compiler support. Many more optimizations are possible for applications that rely on MPI to evolve better programming models, resilience, data management, and energy efficiency mechanisms to reduce overhead, while creating evolving applications. Some of these mechanisms, like RMA, non-blocking, and neighborhood collectives, are introduced in the new MPI 3.0 standard, but the road to Ultrascale is still unpaved.

Acknowledgments

This work has been partially funded by EU under the COST programme Action IC1305.

8. REFERENCES

[1] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. Dongarra. An evaluation of user-level failure mitigation support in mpi. DOI 10.1007/s00607-013-0331-3:1–14, May 2013.

[2] G. Bosilca, A. Bouteiller, T. Herault, Y. Robert, and J. Dongarra. Assessing the impact of abft and checkpoint composite strategies. In *16th Workshop on Advances in Parallel and Distributed Computational Models, IPDPS 2014*, Phoenix, AZ, May 2014.

[3] A. B. C. Symeonidou, P. Pratikakis and D. Nikolopoulos. Drasync: Distributed region-based memory allocation and synchronization. In *Proceedings of the 20th European MPI UsersâĂŹ Group Meeting. EuroMPI âĂŹ13*, page 49âĂŞ54. ACM, 2013.

[4] K. Dichev, V. Rychkov, and A. Lastovetsky. Two algorithms of irregular scatter/gather operations for heterogeneous platforms. In *Recent Advances in the Message Passing Interface*, pages 289–293. Springer Berlin Heidelberg, 2010.

[5] R. Filgueira, J. Carretero, D. E. Singh, A. Calderon, and A. NuÃśez. Dynamic-compi: Dynamic optimization techniques for mpi parallel applications. *The Journal of Supercomputing*, 59(1):361–391, April 2012.

[6] J. D. S. P. P. G. T. F. H. M. H. Jordan, P. Thoman. A multi-objective auto-tuning framework for parallel codes. In *Proc. of the Intl. Conference for High Performance Computing, Networking, Storage and Analysis (SC 2012).* IEEE Computer Society Press, 2012.

[7] K. Hasanov, J.-N. Quintin, and A. Lastovetsky. Hierarchical approach to optimization of parallel matrix multiplication on large-scale platforms. *The Journal of Supercomputing*, pages 1–24, 2014.

[8] F. Isaila, F. J. Garcia Blas, J. Carretero, W.-K. Liao, and A. Choudhary. A Scalable Message Passing Interface Implementation of an Ad-Hoc Parallel I/O System. *Int. J. High Perform. Comput. Appl.*, 24(2):164–184, May 2010.

[9] G. Martin, M.-C. Marinescu, D. E. Singh, and J. Carretero. FLEX-MPI: an MPI extension for supporting dynamic load balancing on heterogeneous non-dedicated systems. In *International European Conference on Parallel and Distributed Computing, EuroPar*, 2013.

[10] D. N. M. S. T. G. S. Lyberis, P. Pratikakis and B. de Supinski. The myrmics memory allocator: hierarchical,message-passing allocation for global address spaces. In *Proceedings of the International Symposium on Memory Management.* 2012.

[11] Q. Zhang, J. Wang, G. D. Guerrero, J. M. Cecilia, J. M. García, Y. Li, H. Pérez-Sánchez, and T. Hou. Accelerated Conformational Entropy Calculations Using Graphic Processing Units. *Journal of chemical information and modeling*, 53(8):2057–2064, Aug. 2013.