DOI: 10.14529/jsfi150202

Acceleration of MPI Mechanisms for Sustainable HPC Applications

***Jesus Carretero*[1], *Javier Garcia-Blas*[1], *David E. Singh*[1], *Florin Isaila*[1], *Alexey Lastovetsky*[2], *Thomas Fahringer*[3], *Radu Prodan*[3], *Peter Zangerl*[3], *Christi Symeonidou*[4], *George Bosilca*[5], *Afshin Fassihi*[6], *Horacio Pérez-Sánchez*[6]**

© The Authors 2015. This paper is published with open access at SuperFri.org

Ultrascale computing systems are meant to reach a growth of two or three orders of magnitude of today computing systems. However, to achieve the performances required, we will need to design and implement more sustainable solutions for ultra-scale computing systems, understanding sustainability in a holistic manner to address challenges as economy-of-scale, agile elastic scalability, heterogeneity, programmability, fault resilience, energy efficiency, and scalable storage. Some of those solutions could be provided into MPI, but other should be devised as higher level concepts, less generalists, but adapted to applicative domains, possibly as programming patterns or or libraries. In this paper, we show some proposals to extend MPI trying to cover major domains that are relevant towards sustainability: MPI programming optimizations and programming models, resilience, data management, and their usage from applications.

Keywords: MPI, MPI sustainability, programming models, resilience, data management, MPI applications.

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

[1]University Carlos III of Madrid, Spain
[2]University College Dublin
[3]University of Innsbruck, Austria
[4]ICS, FORTH, Greece
[5]University of Tennessee, USA
[6]Universidad Católica San Antonio de Murcia (UCAM), Spain

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 their usage for applications.

The remainder of this paper is organized as follows. Section 1 covers communication optimizations, while Section 2 addresses the area of resilience. Section 3 talks about storage and I/O techniques, Section 4 deals with energy constraints, and Section 5 presents some application and algorithm optimizations. The final section concludes the paper.

1. 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.

1.1. 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 [23] 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

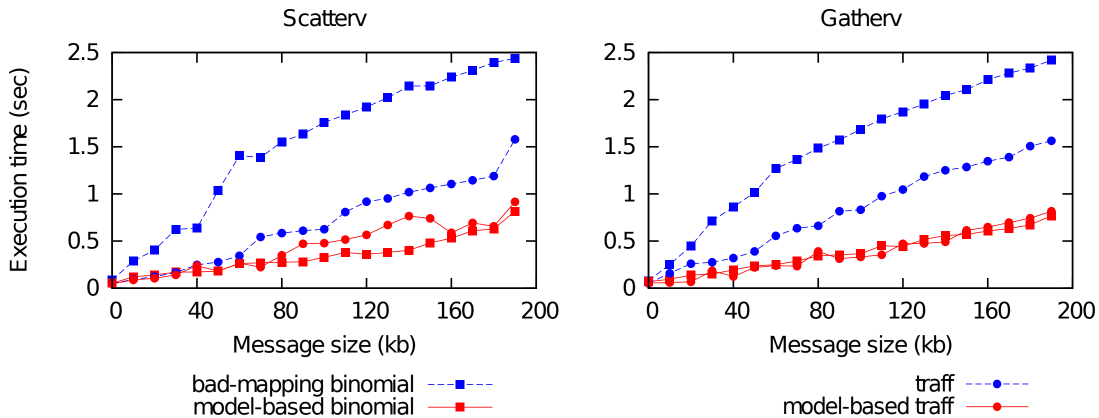


Figure 1. Scatterv and Gatherv operations on geographically distributed clusters from Grid5000

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 [17] 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.

1.2. 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 [10], 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).

1.3. MPI communication with adaptive compression

Adaptive-CoMPI [11] 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 addresses all types of communications, and includes different compression techniques (LZO [26] , RLE [16], HUFFMAN [8], RICE [5] and FPC [19]) that can be used transparently to users (by means of MPI hints).

The architecture of MPICH consists of 3 layers: Application Programmer Interface (API), Abstract Device Interface (ADI) and Channel Interface (CI). The ADI layer is a portable layer, while the Channel layer is not. Therefore, we have modified the ADI layer in order to include the Runtime Compression strategy, independently of the channel and protocol used. Therefore, applying Runtime Compression strategy on point-to-point routines, not only compresses these communications, but also the collective ones. The same is true for blocking and non-blocking communication.

Algorithm 1 Blocking message send

```
len-buffer = contig-size  count
if(len > 2048)
    algorithm-compress ← Read-Hint-User()
    len-compress ← Compression-Message(buff, len, algorithm-compress, buff-compress)
    Send-Message-contiguous(buff-compress, len-compress, src-rank, dest-rank)
end if
```

Algorithm 2 Blocking message reception

```
Check if data is compressed
if(check-request == 1)
    flag-head = Study-Head-of-Buffer(request.buf)
    if(flag-head == yes-compression)
    Decompress the buffer
        buf ← Decompression-Message(request.buf)
    else
        Copy(request.buf → buf)
    end if
end if
```

For all messages sent, a header is included to inform the receiver process if it has to decompress the message or not, and which algorithm must be used.

Adaptive-CoMPI includes two possible compression strategies. The first one, called *Runtime Adaptive Strategy* (RAS) 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. RAS decision process consists of the following steps:

1. Selecting the best compression algorithm at the beginning of the application and everytime the data type changes.
2. Finding the minimum size of the message from which the performance improves.

3. Sending the message (compressed or not).
4. For subsequent messages with the same datatype, the process compares the message size with $length - yes - compression$. If the size of the message is higher, then the message is sent compressed, and uncompressed otherwise.
5. Once the message is sent, and if the decision is to compress, the process checks if the compression-ratio is less than one. In the number of mistakes is higher than a certain threshold within a time interval, the compression is disabled (see fig. 2a).
6. In the other case, if the decision was uncompressed, the process updates the number of messages that have been sent uncompressed. When the number is higher than a reevaluation threshold, then the evaluation is restarted (see fig. 2b).

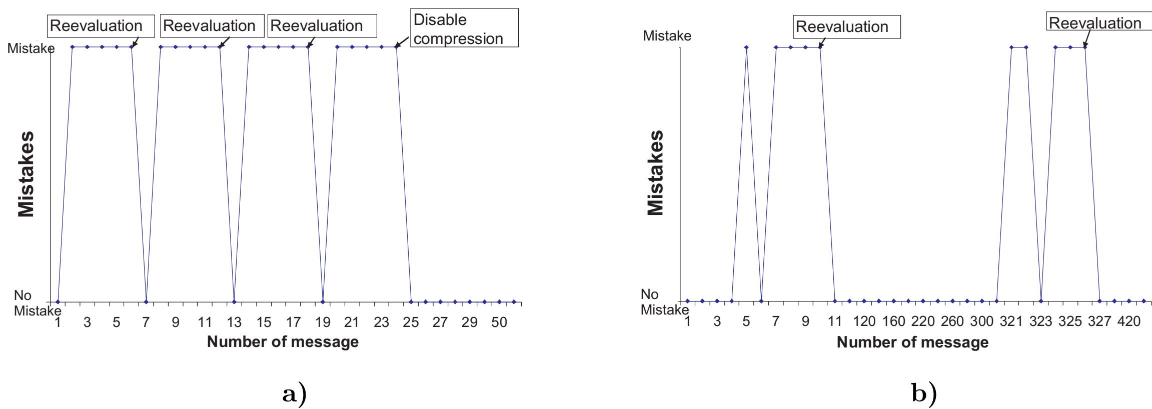


Figure 2. Adaptive-CoMPI with RAS strategy. Learning from errors.

One of the main characteristics of RAS is that it can adapt itself to the applications behavior at runtime. This strategy learns from previous messages which is the compression algorithm to be used and the size from which it obtains a benefit by compressing data.

The second approach, called *Guided Strategy*, provides an application-tailored solution based on the prior application analysis using the application profiling. With this approach along the first execution of the MPI application all the messages are stored in a log file. Upon completion, the best compression algorithm is determined off-line for each message and it is registered in a *decision rules* file. When the application is executed again with the same input parameters and in the same environment, 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. Fig. 3 shows the speedup achieved by BIPS3D when Adaptive-CoMPI techniques are applied. The benchmark has been executed running up to 256 processes in a cluster with dual-core nodes. As may be seen, using Adaptive-CoMPI provides always a performance increase, to a maximum of 1.8 speedup with the same resources.

Fig. 4 shows the speedup achieved for each strategy compared to the execution of the application without compression. Note that the Guided Strategy finds 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.

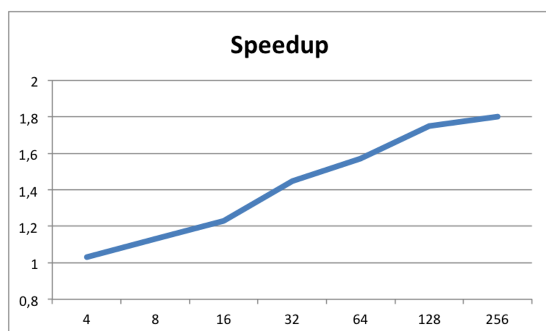


Figure 3. Speedup of Adaptive-CoMPI for the BIPS3D application

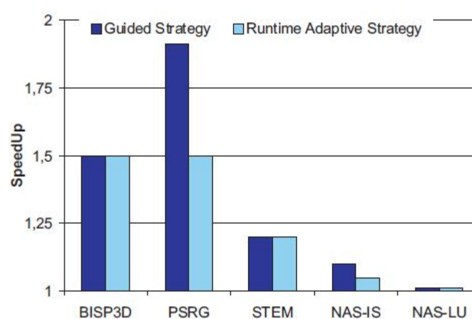


Figure 4. Comparing RAS and Guided-Strategy

2. Resilience

Should the number of components in supercomputers continue to increase, the mean time between failures (MTBF) is expected to decrease to a handful of hours, preventing any capacity application from succesfully delivering its scientific outcome. As a result, deploying fault tolerant strategies within HPC software stack will not only become critically important, but it will have a direct and lasting impact: 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, system level checkpoint/restart is unable, in its current state, to cope with very adversarial future failure patterns. This presents a clear need to improve checkpointing strategies by simultaneously addressing several issues: 1) optimizing the checkpoint procedure by minimizing the storage requirements and by diverging from centralized I/O strategies; and 2) allowing independent restart of failed processes without rollback of all processes [4]. One should understand that adopting such a solution allows the application to complete under a reasonable time interval, but in exchange requires significant investment in reliable hardware (more memory or NVRAM, increased reliable storage and so on). Thus, the relief is only temporary, as the increase in the number of components in the checkpoint restart chain will, by definition, have an impact on the MTBF. Moreover, the total ownership cost of the application will increase, as all these hardware additions will increase the energy requirements for large platforms, a requirement extremely difficult to satisfy at the Exascale level.

Thus, the first potential solution is to simultaneously address two of the major drawbacks of the system-level coordinated checkpoint, by decreasing not only the checkpoint size by also it's frequency. Such solutions have been thriving recently, proposing different interface to address this problem. As an example, in Fault Tolerant Messaging Interface (FMI) [22] employs a survivable communication runtime coupled with a fast, in-memory C/R and dynamic host allocation to enable low-latency recovery. The application developer highlights the critical data for the correct execution of the application, as well as windows of opportunity for a correct checkpoint, allowing the FMI runtime to decide the frequency and the amount of data to be checkpointed. On a somehow similar approach, Fault Tolerance Interface (FTI) [1] proposes to address these challenges by proposing a low-overhead high-frequency multi-level checkpoint approach, in which a highly-reliable topology-aware Reed-Salomon encoding is integrated deep inside the checkpoint scheme. A more data centric approach, named Containment Domain (CD) [7] proposes a programming construct that enable applications to express resilience needs and to interact with

the system to tune and specialize error detection, state preservation and restoration. They behave as weak transactional primitives and can be nested to take advantage of the machine and application hierarchies allowing hierarchical state preservation, restoration and recovery.

Faithful to their coordinated checkpoint/restart roots, these approaches inherit from a former programming period, where synchronous SPMD and BSD application were ruling the parallel application world. They are based on synchronous concepts, forcing a strict coordination not only during the checkpoint, but also during the restart (in addition to requiring a complete restart and a full data recovery). They provide little flexibility to the application to implement specialized fault management approaches, or to take advantage of algorithmic properties in order to code with the faults. Moreover, they do not provide support in the programming paradigm for fault detection without a drastic restart, nor to any kind of support from the message passing substrate.

However, over the last years, algorithm based fault tolerant techniques have proven to be capable to forgo checkpointing completely by employing a tailored, scalable protective strategy to maintain sufficient algorithm-specific redundancy to restore lost data pieces due to failures without a global need for restart. Moreover, a large number of application can cope with a lesser support for fault management from the runtime. 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, are just a few examples of such resilient applications. 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 communication libraries, MPI and PGAS, 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 [2]. 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 assuming if, nor how, 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. This novel proposal is a low level layer, basically the most basic portability layer, that can be exposed at the communication infrastructure level, to allow for flexible and portable higher level concepts, but adapted to specific applicative domains. Thus, these additions propose 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 FMI, FTI, CS, transactional fault tolerance, uncoordinated checkpoint-restart, and programming languages, to implement their own needs and to provide seamless support for advanced fault tolerance models that are thereby portable between MPI implementations.

3. 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 lead 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: the data-centric paradigm.

In this sense, AHPIOS (Ad-Hoc Parallel I/O system for MPI applications) [14] proposes a scalable parallel I/O system completely implemented in MPI. AHPIOS allows MPI applications to dynamically manage and scale distributed partitions in a convenient way. The configuration of both 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. 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. Finally, AHPIOS takes advantage of view-based I/O [3] is a file-system independent I/O optimization based on file views. View-based I/O avoids the necessity of transferring large lists of offset-length pairs at file access time as the present implementation of two-phase I/O.

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.

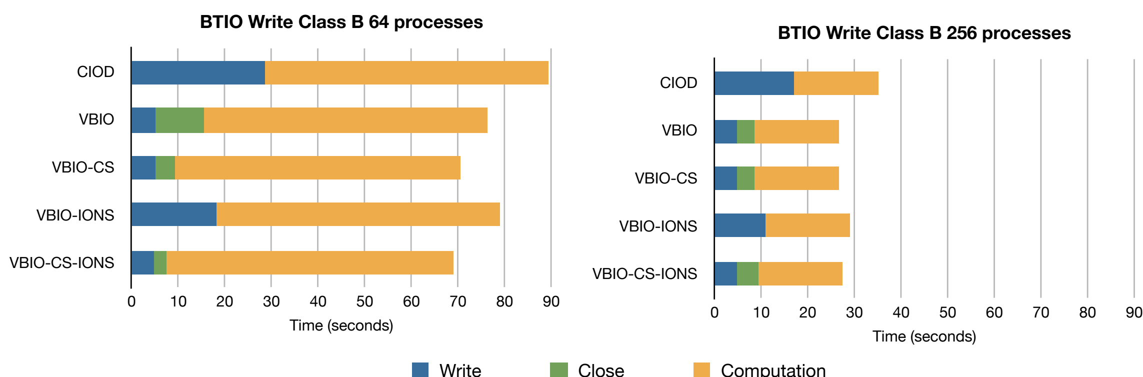


Figure 5. AHPIOS. BTIO class C measurements. ROMIO two-phase I/O over PVFS2, Lustre, AHPIOS and the two AHPIOS-based solutions: server-directed I/O and client-directed I/O

The performance and scalability of AHPIOS for an MPI application that writes and reads in parallel, disjoint, contiguous regions of a file, stored over an AHPIOS system for different numbers of AHPIOS servers, has been demonstrated on both commodity clusters and BlueGene/P

supercomputers. We have evaluated the performance of file writes of the BTIO benchmark for four different setups: two-phase I/O over the IBM solution (CIOD), AHPIOS without cache and view-based I/O as collectives (VBIO), AHPIOS and view-based I/O with client-side caching (VBIO-CS), and AHPIOS with view-based I/O with both client-side and I/O node-side caching (VBIO-CS-IONS). Fig. 5 shows the total time breakdown into compute time, file write time, and close time, for BTIO class B and C. The close time is relevant because all data is flushed to the file system when the file is closed. We notice that in all solutions the compute time is roughly the same. VBIO reduces the file write time without any asynchronous transfers. VBIO-CS reduces both the write time and close time, as data is asynchronously written from compute node to I/O node. For VBIO-CS-IONS, the network and I/O activity are almost entirely overlapped with computation. We conclude that the performance of file writes gradually improves with the increasing degree of asynchrony in the system.

4. Energy

Energy has become a major concern for the sustainability of future computer architectures. Providing MPI applications with malleable and energy-aware capabilities allows executing them more efficiently and with less energy requirements, as shown in this section.

Intel SandyBridge chips include the *Running Average Power Limit* (RAPL) interface that provides an energy estimation based on hardware monitoring. Other manufacturers like AMD, IBM and NVIDIA include similar interfaces in their products. We access to this information (via PAPI [24]) to evaluate the application power consumption. Fig. 6 shows the aggregated processor power of Jacobi and Conjugate Gradient running on a compute node consisting of two 6-core Intel Xeon E5-2620 processors. We can observe that the energy consumption is different for each application given that they have different compute and memory intensity levels. There is a sharp increase of power until 12 processes because the available resources (cores) in the system. From 12 to 24 processes the power (and performance) still slightly increase leveraging the processor hyper-threading capabilities.

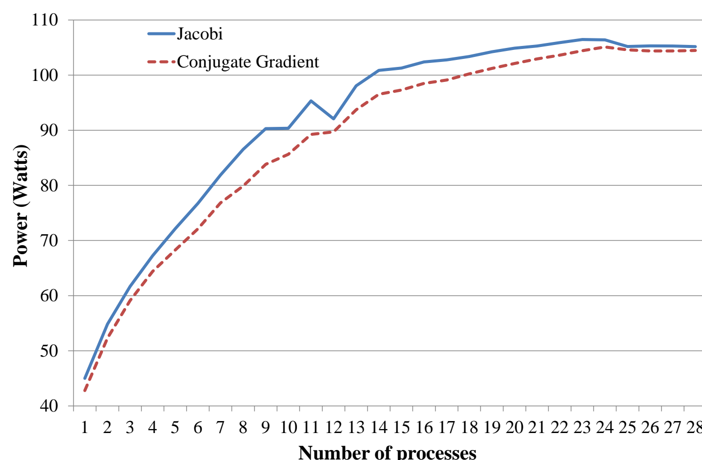
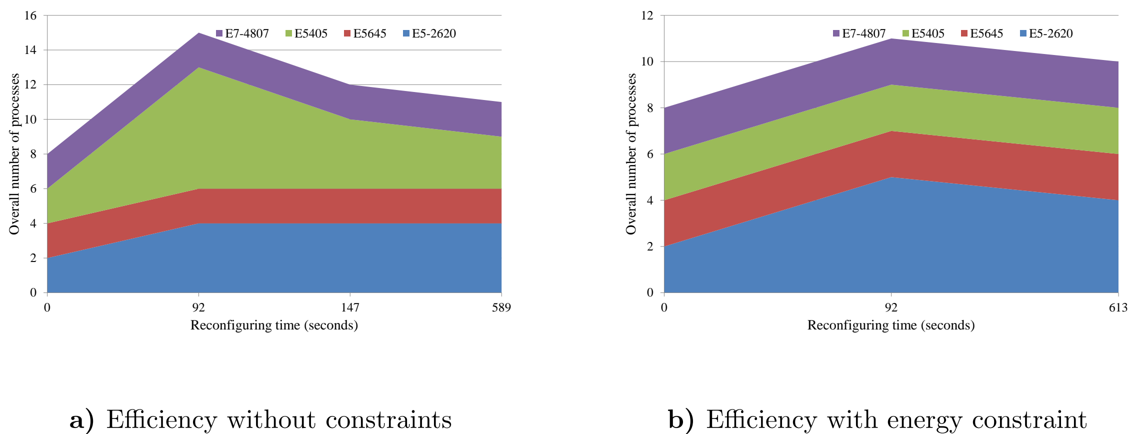


Figure 6. Aggregated CPU energy for Conjugate Gradient and Jacobi executed on a node with two Intel Xeon E5-2620 processors

FLEX-MPI [18] is an MPI extension which provides performance-aware dynamic reconfiguration capabilities to MPI applications. In addition, FLEX-MPI considers two different constraints: cost and energy. The cost constraint consists of reaching a given level of application performance (in FLOPs) at the smallest operational cost (measured in $ per CPU time). In the case of the energy constraint, we aim to reach the performance level with the smallest aggregated energy cost (in Joules) among all the processors involved in the application execution. Note that there are important differences between both constraints. For instance, the operational cost is usually constant for a given processor class, but the energy cost is strongly related to the processor load (as fig. 6 shows).

FlexMPI addresses heterogeneous architectures where each class of nodes has different energy, cost and performance specifications. Finding a solution to the aforementioned problems is usually non trivial given the existing trade-offs between performance and costs (both economic and energetic). For reaching these objectives, we employ a computational prediction model that takes into account both the application and platform characteristics. This model uses hardware counters to characterize the processor power for the considered program under different load scenarios. Later, during the program execution FLEX-MPI uses the PAPI library to survey hardware events (like energy and FLOPS) of each MPI process, and of the MPI interface, to collect the performance of the MPI communications. Based on the collected data, it decides to spawn or remove processes in order to achieve the user-defined performance objectives. In case of a spawn operation, Flex-MPI decides which compute nodes are the most appropriate to run the new created processes. In addition, when the number of processes changes, Flex-MPI also includes functionalities for performing the data redistribution, thereby guaranteeing appropriate load balance among the processes.



a) Efficiency without constraints **b)** Efficiency with energy constraint

Figure 7. Number of processes and type of processors scheduled by Flex-MPI for a performance improvement objective of 30% and Jacobi benchmark with 20K input matrix

The results are encouraging, as was demonstrated by executing Jacobi method with two different matrix sizes (10K and 20K rows and columns) and performance improvement objective of 30%. We consider two scenarios for FlexMPI: the first one tries to reach the performance objective without any constraint. The second scenario combines the performance objective with the energy constraint. In our experiments we used a heterogeneous platform with four classes of Intel Xeon nodes: E52620, E5645, E74807 and E5405. The average GFLOP/Watt ratio per core values are respectively 0.13, 0.16, 0.24, and 0.32. Note that the last node is the most

energy-efficient one. Unfortunately, some compute nodes are not Sandy Bridge and it is not possible to measure the energy by hardware counters. For them, we indirectly obtained the energy under different loads by means of an empirical model based on Intel Xeon E5410 [21] and the extrapolation of the values obtained for E5-2620 node.

In our experiments, all the executions with Flex-MPI reached the performance improvement objectives. The reference version (without Flex-MPI) runs two application processes in each compute node type (summarizing 8 processes). For the 10K and 20K matrices the reference version produced an accumulated energy of 30.3 and 120.3 KJoules, respectively. For the efficiency objective (without constraints), the energy values are respectively 31.1 and 124.8 KJoules. For the efficiency objective with energy constraint the energy values are respectively 28.5 and 109.3 KJoules. As fig. 7 shows with an energy efficiency goal, Flex-MPI schedules more dynamic processes on the nodes which have a better performance/energy ratio. In this method, the overall operational energy cost is minimized keeping the performance objectives. This result demonstrates that the combined use of low-level monitoring and malleability at runtime would be a good option for achieving energy efficiency in MPI systems.

5. 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.

5.1. Hierarchical SUMMA

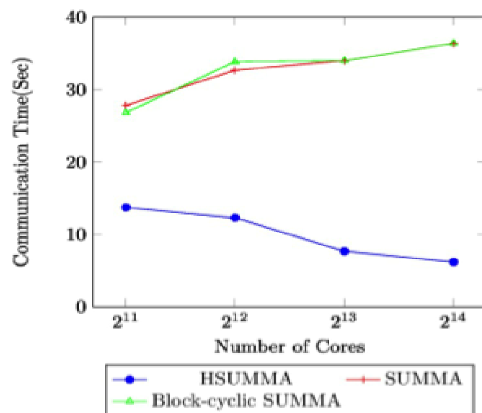


Figure 8. Communication time of SUMMA, block-cyclic SUMMA and HSUMMA on BG/P. p =16K, n = 65,536.

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 [13], where the authors challenge the traditional "flat" design of collective communication patterns in the context of 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. 8). 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.

5.2. 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 has been 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 [15]. 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 code of Algorithm 3.

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

Algorithm 3 Example pattern for MPI_Bcast

```
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
```

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.

Algorithm 4 Example pattern for optimized MPI_Bcast

```
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
```

5.3. 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). We mention below some of the most representative and efficient Hybrid MPI-OpenMP Implementations.

Molecular Dynamics using DL POLY: DL POLY, a large scale Molecular Dynamics (MD) application programmed using MPI was modified to add a layer of shared memory threading [6], and the code was tested on two multi-core clusters. At smaller core numbers on both systems the pure MPI code outperformed the hybrid message passing and shared memory code. The slower performance of the hybrid code at low core numbers was due to the extra overheads from the shared memory implementation, and the lack of any significant benefit from a reduced communication profile. For more cores on both systems, the hybrid code delivered better performance. In general the hybrid code spent less time carrying out communication than the pure MPI code, performing better at point to point communication at all core counts, and collec-

tive communication at higher core counts. This reduced communication was the main driver for performance improvements in the hybrid code. At low core counts the added overheads from OpenMP parallelization reduced the hybrid code performance, but the effects of these overheads decreased as the number of cores increased. The choice of system interconnect had an effect on the performance of the hybrid code when compared to the pure MPI code. Using a fast Infiniband interconnect the pure MPI code outperformed the hybrid up to a larger number of cores than when using a slower 10 GigE interconnect.

Molecular Dynamics using LAMMPS: Pal et al. [20] developed a computational scheme for MD simulations that exploited thread-parallelism as well as message passing techniques and implemented it on a cluster of 6 dual-quad-core blade servers (SMP nodes), connected using Infiniband and were the challenges and issues of such schemes were discussed in detail. They showed that such a coupled scheme could work nearly twice as fast as a pure message-passing based implementation for certain system sizes, owing to the additional overheads in the latter being circumvented by the former scheme. When using unthreaded MPI processes with this algorithm, the speed-up obtained saturated quickly on the cluster, well before the total number of available cores were utilized. A set of hybrid schemes were compared and were found to be competitive. The authors state that certain code-optimizations and computational loads may favor one particular scheme over the other and hence it is unwise to treat a particular scheme as the best processorthread configuration. However, they found that using unthreaded MPI processes was likely to be inefficient as compared to threaded processes. LAMMPS, which does not spawn threads for parallelization, was found to achieve a speed-up that was significantly inferior to that obtained by their hybrid algorithm. However, the algorithm used for the parallelization was not optimal, and its performance can be enhanced further. There is room for further improvements in the serial algorithm as well.

Adaptive Integral Method: Wei et al [25] presented a hybrid MPI/OpenMP parallelization technique for improving the scalability of classical adaptive integral method (AIM) accelerated classical iterative method of moments (MOM) solvers on multi-core clusters. The schemes they used were based on nested decompositions; a nested column-row decomposition was used for the classical MOM computations and a nested 1-D slab decomposition of the 3-D auxiliary regular grid was used for the AIM acceleration. The scalability of the resulting methods matrix fill time, memory requirement, and matrix solve time were examined theoretically and contrasted to that of a pure MPI parallelization. It was shown that when pure MPI parallelization was used on multi-core clusters, the scalability of both classical and AIM accelerated MOM were limited by two factors: (i) the memory needed for storing replicated geometry/basis function data, and (ii) the communications during the iterative matrix solution. The hybrid MPI/OpenMP parallelization was shown to be useful for both limitations because it did not replicate non-parallelized data structures among different cores of a processor making the memory requirement independent of the number of active cores and because it used fewer messages to communicate larger chunks of data among processors and reduced the impact of latency. For classical MOM, for which the matrix solution can be latency or bandwidth limited, hybrid MPI/OpenMP parallelization always alleviated both of the limiting factors effectively. For AIM accelerated MOM, for which the matrix solution can be grid or latency limited, hybrid MPI/OpenMP parallelization always alleviated the memory limitation but could alleviate the communication limitation only when the matrix solution was latency limited. They concluded that as the performance improvements are a function of the number of active cores in a processor, hybrid parallelization methods are

expected to become more important as the general trend of increasing number of cores in multi- and many-core processors continues.

Drug Discovery: Guerrero et al [12] developed a hybrid optimized version for Virtual Screening calculations in Drug Discovery, that reached up to a 229x speed-up factor, versus its sequential counterpart. On their implementation, threads cooperated in parallel to perform the calculations within each node in a vectorized fashion. Once the data had been distributed using MPI, the calculation of the energy was 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 could be overlapped by asynchronous send/receive instructions. Next, MPI was used to move molecule related data between nodes, instead of sending all the information to each core. The communication was reduced by a *ratio of number of cores* per node, with respect to the MPI implementation. This hybrid distributed memory system exhibited good scalability with the number of processors, which is explained by the low number of communications required by the simulations in their hybrid MPI-OpenMP implementation. These hybrid solutions are adequate when the Virtual Screening 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.

Topological Analysis: Cui et al [9] discussed a hybrid MPI/OpenMP approach to implement a parallel processing of topological operations. They showed that implementing an OpenMP application is simpler and quicker than implementing an MPI application. In order to obtain speedup curves for the parallel scheme, they conducted within and overlap operations on a PC cluster. In the first experiment, China County-level Point Data and China County-level Polygon Data were used. In the second experiment, soil type map and land use map in Heilongjiang and Jilin province were used. Experimental performance results demonstrated that a mixed mode code, with the MPI parallelization occurring across the nodes and OpenMP parallelization within the nodes, was more efficient on a cluster as the mode matches the architecture more closely than a pure MPI model.

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.

The work presented in this paper was partially supported by EU under the COST programme Action IC1305, 'Network for Sustainable Ultrascale Computing (NESUS)'.

This paper is distributed under the terms of the Creative Commons Attribution-Non Commercial 3.0 License which permits non-commercial use, reproduction and distribution of the work without further permission provided the original work is properly cited.

References

1. Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. Fti: High performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 32:1–32:32, New York, NY, USA, 2011. ACM. DOI: 10.1145/2063384.2063427.

2. W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J.J. Dongarra. An evaluation of user-level failure mitigation support in mpi. 95:1–14, May 2013. DOI: 10.1007/s00607-013-0331-3.

3. Javier Garcia Blas, Florin Isaila, David E. Singh, and Jesus Carretero. View-based collective I/O for MPI-IO. In *8th IEEE International Symposium on Cluster Computing and the Grid, 2008. CCGRID '08.*, pages 409–416, May 2008.

4. 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. DOI: 10.1109/ipdpsw.2014.79.

5. M. Burtscher and Paruj R. fpc: A high-speed compressor for double-precision floating-point data. *IEEE Transactions on Computers*, 58(1):1831, 2009. DOI: 10.1109/tc.2008.131.

6. Martin J Chorley and David W Walker. Performance analysis of a hybrid MPI/OpenMP application on multi-core clusters. *Journal of Computational Science*, 1(3):168–174, August 2010. DOI: 10.1016/j.jocs.2010.05.001.

7. Jinsuk Chung, Ikhwan Lee, Michael Sullivan, Jee Ho Ryoo, Dong Wan Kim, Doe Hyun Yoon, Larry Kaplan, and Mattan Erez. Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems. In *the Proceedings of SC12*, November 2012. DOI: 10.1109/sc.2012.36.

8. S. Coco, DArrigo V., and Giunta D. A rice-based lossless data compression system for space. In *2000 IEEE Nordic Signal Processing Symposium*, pages 133–142, May 2000.

9. Shulin Cui and Shuqing Zhang. Parallel processing of topological operations by using a hybrid MPI/OpenMP approach. In *Natural Computation (ICNC), 2013 Ninth International Conference on*, pages 1738–1742, 2013. DOI: 10.1109/icnc.2013.6818263.

10. Kiril Dichev, Vladimir Rychkov, and Alexey 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. DOI: 10.1007/978-3-642-15646-5_31.

11. Rosa Filgueira, Jesus Carretero, David E. Singh, Alejandro Calderon, and Alberto Nuez. Dynamic-COMPI: Dynamic optimization techniques for mpi parallel applications. *The Journal of Supercomputing*, 59(1):361–391, April 2012. DOI: 10.1007/s11227-010-0440-0.

12. Ginés D Guerrero, Horacio Pérez-Sánchez, José M Cecilia, and José M García. Parallelization of virtual screening in drug discovery on massively parallel architectures. In *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, pages 588–595. IEEE, 2012. DOI: 10.1109/pdp.2012.26.

13. Khalid Hasanov, Jean-Noel Quintin, and Alexey Lastovetsky. Hierarchical approach to optimization of parallel matrix multiplication on large-scale platforms. *The Journal of Supercomputing*, pages 1–24, 2014. DOI: 10.1007/s11227-014-1133-x.

14. Florin Isaila, Francisco Javier Garcia Blas, Jesús Carretero, Wei-Keng Liao, and Alok 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. DOI: 10.1177/1094342009347890.

15. H. Jordan, P. Thoman, J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch. 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. DOI: 10.1109/sc.2012.7.

16. Donald E. Knuth. Dynamic huffman coding. *Journal of Algorithms*, 6(2):163180, 1985. DOI: 10.1016/0196-6774(85)90036-7.

17. S. Lyberis, P. Pratikakis, DS. Nikolopoulos, M. Schulz, T. Gamblin, and BR. de Supinski. The myrmics memory allocator: hierarchical,message-passing allocation for global address spaces. In *Proceedings of the International Symposium on Memory Management*. 2012. DOI: 10.1145/2258996.2259001.

18. Gonzalo Martin, Maria-Cristina Marinescu, David E. Singh, and Jesus 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. DOI: 10.1007/978-3-642-40047-6_16.

19. M.F.X.J. Oberhumer. *LZO real-time data compression library*. 2012.

20. Anirban Pal, Abhishek Agarwala, Soumyendu Raha, and Baidurya Bhattacharya. Performance metrics in a hybrid MPI–OpenMP based molecular dynamics simulation with short-range interactions. *Journal of Parallel and Distributed Computing*, 74(3):2203–2214, March 2014. DOI: 10.1016/j.jpdc.2013.12.008.

21. M. Pedram and Inkwon Hwang. Power and performance modeling in a virtualized server system. In *2010 39th International Conference on Parallel Processing Workshops (ICPPW),*, pages 520–526, Sept 2010. DOI: 10.1109/ICPPW.2010.76.

22. K. Sato, A. Moody, K. Mohror, T. Gamblin, B.R. de Supinski, N. Maruyama, and S. Matsuoka. Fmi: Fault tolerant messaging interface for fast and transparent recovery. In *Parallel*

and Distributed Processing Symposium, 2014 IEEE 28th International, pages 1225–1234, May 2014. DOI: 10.1109/IPDPS.2014.126.

23. C. Symeonidou, P. Pratikakis, A. Bilas, and DS. Nikolopoulos. Drasync: Distributed region-based memory allocation and synchronization. In *Proceedings of the 20th European MPI Users Group Meeting. EuroMPI 13*, page 4954. ACM, 2013. DOI: 10.1145/2488551.2488558.
24. V.M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore. Measuring energy and power with papi. In *2012 41st International Conference on Parallel Processing Workshops (ICPPW)*, pages 262–268, Sept 2012. DOI: 10.1109/ICPPW.2012.39.
25. Fangzhou Wei and Ali E Yilmaz. Parallel Computing. *Parallel Computing*, 37(6-7):279–301, July 2011.
26. Robert Zigon. Run length encoding. *Dr. Dobb's Journal*, 14(2):126128, 1989.

Received February 11, 2015.