DOI: 10.14529/jsfi150201

Exascale Machines Require New Programming Paradigms and Runtimes

Georges Da Costa[1], *Thomas Fahringer*[2], *Juan-Antonio Rico-Gallego*[3], *Ivan Grasso*[2], *Atanas Hristov*[4], *Helen D. Karatza*[5], *Alexey Lastovetsky*[6], *Fabrizio Marozzo*[8], *Dana Petcu*[7], *Georgios L. Stavrinides*[5], *Domenico Talia*[8], *Paolo Trunfio*[8], *Hrachya Astsatryan*[9]

Extreme scale parallel computing systems will have tens of thousands of optionally accelerator-equipped nodes with hundreds of cores each, as well as deep memory hierarchies and complex interconnect topologies. Such exascale systems will provide hardware parallelism at multiple levels and will be energy constrained. Their extreme scale and the rapidly deteriorating reliability of their hardware components means that exascale systems will exhibit low mean-time-between-failure values. Furthermore, existing programming models already require heroic programming and optimization efforts to achieve high efficiency on current supercomputers. Invariably, these efforts are platform-specific and non-portable. In this article, we explore the shortcomings of existing programming models and runtimes for large-scale computing systems. We propose and discuss important features of programming paradigms and runtimes to deal with exascale computing systems with a special focus on data-intensive applications and resilience. Finally, we discuss code sustainability issues and propose several software metrics that are of paramount importance for code development for ultrascale computing systems.

Keywords: programming models, ultrascale, runtimes, extreme scale.

Introduction

Ultrascale systems are envisioned as large-scale complex systems joining parallel and distributed computing systems that will be two to three orders of magnitude larger than today's systems reaching millions to billions elements. New programming models and runtimes will be necessary to use efficiently these new infrastructures. To achieve results on this front, the European Union funded the *COST Action Nesus IC1305* [72]. Its goal is to establish an open European research network targeting sustainable solutions for ultrascale computing, aiming at cross fertilization among HPC, large-scale distributed systems and big data management. The network contributes to gluing together disparate researchers working across different areas and provides them with a meeting ground to exchange ideas, identify synergies and pursue common activities in research topics, such as sustainable software solutions (applications and system software stack), data management, energy efficiency and resilience.

One key element on the ultrascale front is the necessity of new sustainable programming and execution models in the context of rapid underlying computing architecture changing. There is a need to explore synergies among emerging programming models and runtimes from HPC, distributed systems and big data management communities. To improve the programmability of

[1]University of Toulouse, Toulouse, France
[2]University of Innsbruck, Innsbruck, Austria
[3]University of Extremadura, Badajoz, Spain
[4]UIST, Orhid, Republic of Macedonia
[5]Aristotle University of Thessaloniki, Thessaloniki, Greece
[6]University College Dublin, Dublin, Ireland
[7]West University of Timisoara, Timisoara, Romania
[8]University of Calabria, Rende CS, Italy
[9]National Academy of Sciences of Armenia, Yerevan, Armenia

future systems, the main changing factor will be the substantially higher levels of concurrency, asynchrony, failures and heterogeneous architectures.

At different levels, teams of scientists are tackling this challenge. The goal of the European Cost Action Nesus is to establish an open European research network. One of the key elements this action will target is the proposal and implementation of novel programming paradigms and runtimes in order to make the ultrascale a reality. On the other hand, the main objective of the *European Exascale Software Initiative* [73] is to provide recommendations on strategic European actions with a particular focus on software key issues improvement, cross cutting issues advances and gap analysis. The common goal of these two different actions, is to provide a coherent agenda for research, but also establish a code and regulation at a European level, in order to reach ultrascale computing.

At the international level, the goal of the *International Exascale Software Project* [74] is also to provide an international common vision for cooperation in order to reach ultrascale: "The guiding purpose of the IESP is to empower ultra-high resolution and data-intensive science and engineering research through the year 2020, by developing a plan for: (a) a common, high-quality computational environment for petascale/exascale systems and (b) catalyzing, coordinating and sustaining the effort of the international open source software community to create that environment as quickly as possible".

This article explores programming models and runtimes required to facilitate the task of scaling and extracting performance on continuously evolving platforms, while providing resilience and fault-tolerant mechanisms to tackle the increasing probability of failures throughout the whole software stack. However, currently no programming solution exists that satisfies all these requirements. Therefore, new programming models and languages are required towards this direction. The whole point of view on application will have to change. As we will show, the current wall between runtime and application models leads to most of these problem. Programmers will need new tools but also new way to assess their programs. As we will see, data will be a key concept around which failure-tolerant high number of micro-threads will be generated using high-level information by adaptive runtime.

This article is structured as follows: the next section describes the requirements from the programmability point of view for extra large-scale systems such as ultrascale systems. The second section describes how data shift the paradigm of processor-centric management toward a data-centric one in next generation systems. The third section describes how resilience will be of critical importance, since faults and reconfiguration will be a recurring element in such a large-scale infrastructure. The fourth section presents the elements required to reach sustainable software development, whereas the final section concludes this article.

1. Improved programmability for extra large-scale systems

Supercomputers have become an essential tool in numerous research areas. Enabling future advances in science requires the development of efficient parallel applications, which are able to meet the computational demands. The modern high-performance computing systems (HPCS) are composed of hundreds of thousand computational nodes. Due to the rapidly increasing scale of those systems, programmers cannot have a complete view of the system. The programmability strongly determines the overall performance of a high performance computing system. It is a substrate over which processors, memory and I/O devices are exchanging data and instructions. It should have high scalability, which will support the development of the next generation exascale supercomputers. Programmers also need to have an abstraction that allows them to manage hundreds of millions to billions of concurrent threads. Abstraction allows organizing programs into comprehensible fragments, which is very important for clarity, maintenance and scalability of the system. It also allows increasing of programmability by defining new languages on top of the existing language and, by defining completely new parallel programming languages. This makes abstraction an important part of most parallel paradigms and runtimes. Formerly, computer architectures were designed primarily for improved performance or for energy efficiency. In future exascale architectures, one of the top challenges will be enabling a programmable environment for the next generation architectures. In reality, programmability is a metric, which is really difficult to define and measure. The next generation architectures should minimize the chances of parallel computational errors while relieving the programmer from managing low-level tasks.

In order to explore this situation more precisely, one aim of this research is to investigate the limitations of current programming models along-with evaluations of promises of hybrid programming model to solve these scaling-related difficulties.

1.1. Limitations of the current programming models

Reaching exascale in terms of computing nodes requires the transition from current control of thousands of threads to billions of threads as well as the adaptation of the performance models to cope with an increased level of failures. One simple model that is used to program at any scale is a utopian idea as proved in the last twenty years of 'standardized' parallel computing. Unfortunately the exascale has become the reason for improving programming systems, as existing programming models implementations, more than a reason for a change in the programming models. This approach was classified by Gropp and Snir in [2] as evolutionary. According to their view, the five most important characteristics of the programming models that are affected by the exascale transition are: the thread scheduling, the communications, the synchronization, the data distribution and the control views.

1.1.1. Limitations of message passing programming model

The current vision on exascale system is at the moment to exploit distributed memory parallelism, and therefore the message passing model is likely to be used at least partially. Moreover the most popular system implementation of the model, MPI, has been shown to run with millions of cores for particular problems. MPI is based upon standard sequential programming languages, augmented with low-level message passing constructs, forcing users to deal with all aspects of parallelization, including the distribution of data and work to cores, communication and synchronization. MPI primarily favors static data distribution and is consequently not well suited to deal with dynamic load balancing.

However, it has been shown that the all-to-all communication algorithms used in the message passing models are not scalable (most commonly used implementations often assume a fully connected network and have dense communication patterns) while all-to-some, one-sided or sparse communication patterns are more reliable.

Furthermore, parallel I/O is a limiting factor in the MPI systems, showing that the current MPI-IO model should be reconsidered. In particular the limitations are related to the collective access to the I/O request and the data partitioning.

1.1.2. Limitations of shared-memory programming models

The exascale system is expected to handle hundreds of cores in the one CPU or GPU. Using shared-memory systems is a feasible alternative to message passing in the case of medium size parallel systems in order to reduce the programming overhead as is moving the parallelization burden from the programmer to the compiler.

The most popular shared-memory system, OpenMP, is following a parallelism control model that does not allow the control of data distribution and uses non-scalable synchronization mechanism like locks or atomic sections. Moreover, the global view of data leads easily to non-efficient programming as encouraging synchronization joins of all threads' remote data accesses similar to the local ones.

The emerging Partitioned Global Address Space model (PGAS) is trying to overcome the scalability problems of the global shared-memory model [3]. The PGAS model is likely to have benefit where non-global communication patterns can be implemented with minimal synchronization and overlap of computation and communication. Moreover, the scalability of I/O mechanisms in PGAS depends only on the scalability of the underlying I/O infrastructure and is not limited by the model. However, the scalability is limited to thousands of cores (with the exception of X10 which is implementing an asynchronous PGAS model). The load balancing is still an open issue for the systems that implement the model. Furthermore, it is not possible yet to sub-structure threads into subgroups.

1.1.3. Limitations of heterogeneous programming

Clusters of heterogeneous nodes composed of multi-core CPUs and GPUs are increasingly being used for High Performance Computing due to the benefit in peak performance and energy efficiency. In order to fully harvest the computational capabilities of such architectures, application developers often employ a combination of different parallel programming paradigms (e.g. OpenCL, CUDA, MPI and OpenMP). However, heterogeneous computing also poses the new challenge of how to handle the diversity of execution environment and programming models. The Open Computing Language [60] introduces an open standard for general-purpose parallel programming of heterogeneous systems. An OpenCL program may target any OpenCL-compliant device and today many vendors provide an implementation of the OpenCL standard. An OpenCL program comprises a host program and a set of kernels intended to run on a compute device. It also includes a language for kernel programming, and an API for transferring data between host and device memory and for executing kernels.

Single node hardware design is shifting to a heterogeneous nature. At the same time many of today's largest HPC systems are clusters that combine heterogeneous compute device architectures. Although OpenCL has been designed to work with multiple devices, it only considers local devices available on a single machine. However, the host-device semantics can be potentially applied to remote, distributed devices accessible on different compute nodes. Porting single-node multi-device applications to clusters that combine heterogeneous compute device architectures is not straightforward and in addition it requires the use of a communication layer for data exchange between nodes. Writing programs for such platforms is error prone and tedious. Therefore, new abstractions, programming models and tools are required to deal with these problems.

1.2. Exascale promise of the hybrid programming model

Using a message passing model for the inter-node parallelism and a shared-memory programming model for intra-node parallelism is nowadays seen as a promising path to reach the exascale. The hybrid model is referred as MPI+X, where X represents the programming model that supports threads. The most common X is OpenMP, while there are options for X, like OpenACC.

However, restrictions on the MPI+X model are still in place, for example how MPI can be used in a multi-threaded process. In particular, threads cannot be individually identified as the source or target of MPI messages, or an MPI barrier synchronize the execution of the processes but does not guarantee their synchronization in terms of memory views. The proposal to use MPI Endpoints in all-to-all communications from [4] is a step forward in order to facilitate high performance communication between multi-threaded processes.

Furthermore, combining different programming styles like message passing with shared memory programming lends itself to information hiding between different layers that may be important for optimization. Different runtime systems involved with these programming models lack a global view that can have a severe impact on the overall performance.

However, the biggest problem of MPI+X is the competition for the resources like bandwidth (accessing memory via the inter-node interconnect) [2]. Furthermore, an important obstacle is the memory-footprint and efficient memory usage, as the available memory per core or node is not expected to scale linearly with the number of cores and nodes, and the MPI+X functionality must cope with the expected decrease of space per core or node.

Multitasking is a mean to increase the ability to deal with fluctuations in execution of the threads due to the fault handling or power management strategies. PGAS+multitasking is providing a programming model analogous with MPI+X.

In exascale system storage and communication hierarchies will be deeper than the current ones. Therefore it is expected that the hierarchical programming models should replace the current two level ones [1].

The one-side communication model enables programming in a shared-memory-like programming style. In MPI it is based on the concept of a communication window to which the MPI processes in a communicator statically attach contiguous segments of their local memory for exposure to other processes; the access to the window is granted by synchronization operations. The model separates the communication operations and synchronization for data consistency, allowing the programmer to delay and schedule the actual communications. However the model is criticized for being difficult to be used efficiently.

1.2.1. Innovative programming for heterogeneous computing systems

In recent years, heterogeneous systems have received a great amount of attention from the research community. Although several projects have been recently proposed to facilitate the programming of clusters with heterogeneous nodes [54–59, 68, 69], none of them combines support for high performance inter-node data transfer, support for a wide number of different devices and a simplified programming model.

Kim et al. [56] proposed the *SnuCL* framework that extends the original OpenCL semantics to heterogeneous cluster environments. SnuCL relies on the OpenCL language with few extensions to directly support collective patterns of MPI. Indeed, in SnuCL the programmer is responsible to take care of the efficient data transfers between nodes. In that sense, end users of the SnuCL platform need to have an understanding of MPI collective calls semantics in order to be able to write scalable programs.

Also other works have investigated the problem of extending the OpenCL semantics to access a cluster of nodes. The Many GPUs Package (*MGP*) [69] is a library and runtime system that using the MOSIX VCL layer enables unmodified OpenCL applications to be executed on clusters. *Hybrid OpenCL* [68] is based on the FOXC OpenCL runtime and extends it with a network layer that allows the access to devices in a distributed system. The *clOpenCL* [59] platform comprises a wrapper library and a set of user-level daemons. Every call to an OpenCL primitive is intercepted by the wrapper which redirects its execution to a specific daemon at a cluster node or to the local runtime. *dOpenCL* [55] extends the OpenCL standard, such that arbitrary compute devices installed on any node of a distributed system can be used together within a single application. *Distributed OpenCL* [54] is a framework that allows the distribution of computing processes to many resources connected via network using JSON RPC as a communication layer. *OpenCL Remote* [58] is a framework which extends both OpenCL's platform model and memory model with a network client-server paradigm. *Virtual OpenCL* [57], based on the OpenCL programming model, exposes physical GPUs as decoupled virtual resources that can be transparently managed independent of the application execution.

An innovative approach to program clusters of nodes composed of multi-core CPUs and GPUs has been introduced through *libWater* [71], a library-based extension of the OpenCL programming paradigm that simplifies the development of applications for distributed heterogeneous architectures.

libWater aims to improve both productivity and implementation efficiency when parallelizing an application targeting a heterogeneous platform by achieving two design goals: (*i*) transparent abstraction of the underlying distributed architecture, such that devices belonging to a remote node are accessible like a local device; (*ii*) access to performance-related details since it supports the OpenCL kernel logic. The *libWater* programming model extends the OpenCL standard by replacing the host code with a simplified interface. *libWater* also comes with a novel device query language (DQL) for OpenCL device management and discovery. A lightweight distributed runtime environment has been developed which dispatches the work between remote devices, based on asynchronous execution of both communications and OpenCL commands. *libWater* runtime also collects and arranges dependencies between commands in the form of a powerful representation called *command DAG*. The *command DAG* can be effectively exploited to improve the scalability. For this purpose a collective communication pattern recognition analysis and optimization has been introduced that matches multiple single point-to-point data transfers and dynamically replaces them with a more efficient collective operation (e.g. *scatter*, *gather* and *broadcast*) supported by MPI.

Besides OpenCL-based approaches, also CUDA solutions have been proposed to simplify distributed systems programming. *CUDASA* [66] is an extension of the CUDA programming language which extends parallelism to multi-GPU systems and GPU-cluster environments. *rCUDA* [61] is a distributed implementation of the CUDA API that enables shared remote GPGPU in HPC clusters. *cudaMPI* [62] is a message passing library for distributed-memory GPU clusters that extends the MPI interface to work with data stored on the GPU using the CUDA programming interface. All of these approaches are limited to devices that support CUDA, i.e. NVidia GPU accelerators, and therefore they cannot be used to address heterogeneous systems which combines CPUs and accelerators from different vendors.

Other projects have investigated how to simplify the OpenCL programming interface. Sun et. al [65], proposed a task queuing extension for OpenCL that provides a high-level API based on the concepts of work pools and work units. *Intel CLU* [75], *OCL-MLA* [53] and *SimpleOpencl* [70] are lightweight API designed to help programmers to rapidly prototype heterogeneous programs.

A more sophisticated approach was proposed in [67]. OmpSs relies on compiler technologies to generate host and kernel code from a sequential program annotated with pragmas. The runtime of OmpSs internally uses a DAG with the scope of scheduling. However, the DAG is not dynamically optimized as done by *libWater*.

2. Data-intensive programming and runtimes

The data intensity of scientific and engineering applications forces the expansion of exascale system. It puts a focus on architectures, programming models, runtime systems improvement on data intensive computing. A major challenge is to utilize the available technologies and large-scale computing resources effectively to tackle the scientific and societal challenges. This section describes the runtime requirements and scalable programming models for data-intensive applications, as well as new data access, communication, and processing operations for data-intensive applications.

2.1. Next generation MPI

MPI is the most widely used standard [10] in the current petascale systems, supporting among others the message passing model. It has proven high performance portability and scalability [9, 17], as well as stability over the last 20 years. MPI provides a (nearly) fixed number of statically scheduled processes with a local view of the data distributed across the system. Nevertheless, ultrascale systems are not going to be built scaling incrementally from current systems, which probably will have a high impact on all levels of the software stack [2, 18]. The international community agrees that changes need to be done in current software at all levels. Future parallel and distributed applications push to explore alternative scalable and reliable programming models [13].

Current HPC systems need to scale up by three orders of magnitude to meet exascale. While a sharp rise in the number of nodes of this magnitude is not expected, the critical growth will come from the intra-node capacities. Fat nodes with a large number of lightweight heterogeneous processing elements, including accelerators, will be common in the ultrascale platforms, mixing parallel and distributed systems. In addition, memory per core ratio is expected to decrease, while the number of NUMA nodes will grow to alleviate the problem of memory coherence between hundreds of cores [16]. On the software side, weak scaling of applications running on such platforms will demand more computation resources to manage huge volumes of data. Nowadays MPI applications, most of which are built using the *bulk-synchronous* synchronization model [11] as a sequence of communication and computation over the interchanged data stages, will continue to be important, but multi-physics and adaptive meshing applications, with multiple components implemented using different programming models, and with dynamic starting and finalization of such components, will become common in ultrascale. Apart from the most regular applications, this synchronization model is already a strangle point.

In this scenario, programming models need to face multiple challenges to efficiently exploit resources with a high level of programmability [14]: scalability and parallelism increase, energy efficient resource management and data movement, and I/O and resilience in applications among others.

MPI has successfully faced the scalability challenge at petascale with the so-called hybrid model, represented as MPI+X, meaning MPI to communicate between nodes and a shared memory programming model (e.g. OpenMP for shared memory and OpenACC for accelerators) inside a node. This scheme provides with load balancing and reduces the memory footprint and data copies in shared memory, and it is likely to continue in the future. Notwithstanding, increase in node scale, heterogeneity and complexity of integration of processing elements will demand improved techniques for balancing the computational load between potentially large number of processes running kernels composing the application [24].

Increasing imbalances in large-scale applications, aggravated by hardware power management, localized failures or system noise, require synchronization-avoiding algorithms, adaptable to dynamic changes in the hardware and the applications. An example is the collective algorithms based on a non-deterministic point-to-point communication pattern, and able to capture and deal with relevant network properties related to heterogeneity [23]. The MPI specification provides support to mitigate load imbalance issues through the one-sided communication model, non-blocking collectives, or the scalable neighbor collectives for communication along the virtual user defined topology. In the meanwhile, specification and implementation scalability issues have been detected [17]. They need to be either avoided, as the use of all to all inherently non-scalable collectives, or improved, as initialization or communicator creation, in exascale applications.

To support the hybrid programming model, MPI defines levels of thread-safety. Lower levels are suitable for bulk-synchronous applications, while higher levels require synchronization mechanisms, which lead current MPI libraries to a significant performance degradation. *Communication endpoints* [19] mechanism is a proposal to extend the MPI standard for reducing contention inside a single process by allowing to attach threads to different endpoints for sending and receiving messages.

Big data volumes and the power consumed in moving data across the system makes data management one of the main concerns in future systems. In the distributed view, the common methodology of reading data from a centralized filesystem, spreading it over the processing elements and writing the results is energy and performance inefficient, and failure prone. Data will be distributed across the system, and the placement of MPI processes in a virtual topology needs to adapt to the data layout to improve the performance, which will require better mapping algorithms. MPI addresses these challenges in shared memory by a programming model based on shared data windows accessible by processes in the same node, hence avoiding horizontal data movement inside the node. However, lack of data locality awareness leads to vertical movement of data across the memory hierarchy, which degrades performance. For instance, the communication buffers received by MPI processes and the access by the local OpenMP threads for computing on them will need smarter scheduling policies. Data-centric approaches are needed for describing data in the system and apply the computation where such data resides [12, 15].

Another critical challenge for MPI to support exascale systems is the resilience, a cross-cutting issue affecting the whole software stack. Current checkpointing/restart methods are insufficient for future systems under a failure ratio of a few hours and in the presence of silent errors, and traditional triple modular redundancy (TMR) is not affordable in an energy efficient manner. New techniques of resilient computing have been proposed and developed, also in the MPI context [5, 6]. One proposal for increasing resilience to node failures is to implement *malleable* applications, able to adapt their execution to the available resources in the presence of hardware errors, and avoiding the restart of the application [7].

Alternatives to MPI come from the Partitioned Global Address languages (PGAs) and High Productivity Computing Systems (HPCS) programming languages. PGAs programming models provide a global view of data with explicit communication as CAF [38] (Co-Array Fortran), or implicit communication as UPC [40] (Unified Parallel C). However, static scheduling and poor performance issues make them currently far from replacing the well established and successful MPI+X hybrid model. Moreover, OpenMP presents problems with nested parallelism and vertical locality, so the possibility of MPI+PGAs has been, and continues to be evaluated [8, 22] to provide a programming environment better suited to the future platforms. HPCS languages, such as a Chapel [63] and X10 [64], provide a global view of data and control. For instance, Chapel provides programming constructions at different levels of abstraction. It includes features for computation-centric parallelism based on tasks, as well as data-centric programming capabilities. For instance, the *locale* construction describes the compute nodes in the target architecture and allows to reasoning about locality and affinity, and to manage global views of distributed arrays.

2.2. Runtime requirements for data-intensive applications

Developing data-intensive applications over exascale platforms requires the availability of effective runtime systems. This subsection discusses which functional and non-functional requirements should be fulfilled by future runtime systems to support users in designing and executing complex data-intensive applications over large-scale parallel and distributed platforms in an effective and scalable way.

The functional requirements can be grouped into four areas: data management, tool management, design management, and execution management [39].

Data management. Data to be processed can be stored in different formats, such as relational databases, NoSQL databases, binary files, plain files, or semi-structured documents. The runtime system should provide mechanisms to store and access such data independently from their specific format. In addition, metadata formalisms should be provided to describe the relevant information associated with data (e.g., location, format, availability, available views), in order to enable their effective access, manipulation and processing.

Tool management. Data processing tools include programs and libraries for data selection, transformation, visualization, mining and evaluation. The runtime system should provide mechanisms to access and use such tools independently from their specific implementation. Also in this case metadata should be provided to describe the most important features of such tools (e.g., their function, location, usage).

Design management. From a design perspective, three main classes of data-intensive applications can be identified: single-task applications, in which a single sequential or parallel process task is performed on a given data set; parameter sweeping applications, in which data are analyzed using multiple instances of a data processing tool with different parameters; workflow-based applications, in which data-intensive applications are specified as possibly complex workflows.

A runtime system should provide an environment to effectively design all the above-mentioned classes of applications.

Execution management. The system should provide a parallel/distributed execution environment to support the efficient execution of data-intensive applications designed by the users. Since applications range from single tasks to complex workflows, the runtime system should cope with such a variety of applications. In particular, the execution environment should provide the following functionalities, which are related to the different phases of application execution: accessing the data to be processed; allocating the needed compute resources; running the application based on the user specifications, which may be expressed as a workflow; allowing users to monitor an applications execution.

The non-functional requirements can be defined at three levels: user, architecture, and infrastructure.

From a user point of view, non-functional requirements to be satisfied include:

- *Data protection.* The system should protect data from both unauthorized access and intentional/incidental losses.
- *Usability.* The system should be easy to use by users, without the need of undertaking any specialized training.

From an architecture perspective, the following principles should inspire system design:

- *Openness and extensibility.* The architecture should be open to the integration of new data processing tools and libraries; moreover, existing tools and libraries should be open for extension and modifications.
- *Independence from infrastructure.* The architecture should be designed to be as independent as possible from the underlying infrastructure; in other terms, the system services should be able to exploit the basic functionalities provided by different infrastructures.

Finally, from an infrastructure perspective, non-functional requirements include:

- *Heterogeneous/Distributed data support.* The infrastructure should be able to cope with very large and high dimensional data sets, stored in different formats in a single site, or geographically distributed across many sites.
- *Availability.* The infrastructure should be in a functioning condition even in the presence of failures that affect a subset of the hardware/software resources. Thus, effective mechanisms (e.g., redundancy) should be implemented to ensure dependable access to sensitive resources such as user data.
- *Scalability.* The infrastructure should be able to handle a growing workload (deriving from larger data to process or heavier algorithms to execute) in an efficient and effective way, by dynamically allocating the needed resources (processors, storage, network). Moreover, as soon as the workload decreases, the infrastructure should release the unneeded resources.
- *Efficiency.* The infrastructure should minimize resource consumption for a given task to execute. In the case of parallel/distributed tasks, efficient allocation of processing nodes should be guaranteed. Additionally, the infrastructure should be highly utilized so to provide efficient services.

Even though several research systems fulfilling most of these requirements have been developed, such as Pegasus [25], Taverna [26], Kepler [27], ClowdFlows [28], E-Science Central [29], and COMPSs [30], they are designed to work on conventional HPC platforms, such as clusters, Grids, and - in some cases - Clouds. Therefore, it is necessary to study novel architectures, environments and mechanisms to fulfill the requirements discussed above, so as to effectively support design and execution of data-intensive applications in future exascale systems.

2.3. Scalable programming models for data-intensive applications

Data-intensive applications often involve a large number of data processing tools that must be executed in a coordinated way to analyze huge amount of data. This section discusses the need for scalable programming models to support the effective design and execution of data-intensive applications on a massive number of processors.

Implementing efficient data-intensive applications is not trivial and requires skills of parallel and distributed programming. For instance, it is necessary to express the task dependencies and their parallelism, to use mechanisms of synchronization and load balancing, and to properly manage the memory and the communication among tasks. Moreover, the computing infrastructures are heterogeneous and require different libraries and tools to interact with them. To cope with all these problems, different *scalable programming models* have been proposed for writing data-intensive applications [31].

Scalable programming models may be categorized based on their level of abstraction (i.e., high-level and low-level scalable models) and based on how they allow programmers to create applications (i.e., visual or code-based formalisms).

Using *high-level scalable models*, the programmers define only the high-level logic of applications while hiding the low-level details that are not fundamental for application design, including infrastructure-dependent execution details. The programmer is helped in application definition and the application performance depends on the compiler that analyzes the application code and optimizes its execution on the underlying infrastructure. Instead, *low-level scalable models* allow the programmers to interact directly with computing and storage elements of the underlying infrastructure and thus to define the applications parallelism directly. Defining an application requires more skills and the application performance strongly depends on the quality of the code written by the programmer.

Data-intensive applications can be designed through *visual programming formalism*, which is a convenient design approach for high-level users, e.g. domain-expert analysts having a limited understanding of programming. In addition, a graphical representation of workflows intrinsically captures parallelism at the task level, without the need to make parallelism explicit through control structures [32]. *Code-based formalism* allows users to program complex applications more rapidly, in a more concise way, and with higher flexibility [33]. The code-base applications can be defined in different ways: i) with a language or an extension of language that allows to express parallel applications; ii) with some annotations in the application code that permits the compiler to understand which instructions will be executed in parallel; and iii) using a library in the application code that adds parallelism to application.

Given the variety of data-intensive applications (from single-task to workflow-based) and types of users (from end users to skilled programmers) that can be envisioned in future exascale systems, there will be a need for scalable programming models with different levels of abstractions (high-level and low-level) and different design formalisms (visual and code-based), according to the classification outlined above. Thus, the programming models should adapt to user needs by ensuring a good trade-off between ease in defining applications and efficiency of executing them on exascale architectures composed by a massive number of processors.

2.4. New data access, communication, and processing operations for data-intensive applications

This subsection discusses the need for new operations supporting data access, data exchange and data processing to enable scalable data-intensive applications on a large number of processing elements.

Data-intensive applications are software programs that have a significant need to process large volumes of data [21]. Such applications devote most of their processing time to run I/O operations and to exchange and move data among the processing elements of a parallel computing infrastructure. Parallel processing of data-intensive applications typically involves accessing, pre-processing, partitioning, aggregating, querying, and visualizing data which can be processed independently. These operations are executed using application programs running in parallel on a scalable computing platform that can be a large Cloud system or a massively parallel machine composed of many thousand processors. In particular, the main challenges for programming data-intensive applications on exascale computing systems come from the potential scalability and resilience of mechanisms and operations made available to developers for accessing and managing data. Indeed, processing very large data volumes requires operations and new algorithms able to scale in loading, storing, and processing massive amounts of data that generally must be partitioned in very small data grains on which analysis is done by thousands to millions of simple parallel operations.

Evolutionary models have been recently proposed that extend or adapt traditional parallel programming models like MPI, OpenMP, MapReduce (e.g., Pig Latin) to limit the communication overhead (in the case of message-passing models) or to limit the synchronization control (in the case of shared-models languages) [2]. On the other hand, new models, languages and APIs based on a revolutionary approach, such as X10, ECL, GA, SHMEM, UPC, and Chapel have been developed. In this case, novel parallel paradigms are devised to address the requirements of massive parallelism.

Languages such as X10, UPC, GA and Chapel are based on a partitioned global address space (PGAS) memory model that can be suited to implement data-intensive exascale applications because it uses private data structures and limits the amount of shared data among parallel threads. Together with different approaches (e.g., Pig Latin and ECL) those models must be further investigated and adapted for providing data-centered scalable programming models useful to support the efficient implementation of exascale data analysis applications composed of up to millions of computing units that process small data elements and exchange them with a very limited set of processing elements. A scalable programming model based on basic operations for data intensive/data-driven applications must include operations for parallel data access, data-driven local communication, data processing on limited groups of cores, near-data synchronization, in-memory querying, group-level data aggregation, and locality-based data selection and classification.

Supporting efficient data-intensive applications on exascale systems will require an accurate modeling of basic operations and of the programming languages/APIs that will include them. At the same time, a significant programming effort of developers will be needed to implement complex algorithms and data-driven applications such that used, for example, in big data analysis and distributed data mining. Programmers must be able to design and implement scalable algorithms by using the operations sketched above. To reach this goal, a coordinated effort between the operation/language designers and the application developers would be very fruitful.

3. Resilience

As exascale systems grow in computational power and scale, failure rates inevitably increase. Therefore, one of the major challenges in these systems is to effectively and efficiently maintain the system reliability. This requires to handle failures efficiently, so that the system can continue to operate with satisfactory performance. The timing constraints of the workload, as well as the heterogeneity of the system resources, constitute another critical issue that must be addressed by the scheduling strategy that is employed in such systems. Therefore, the next generation code will need to be *resistant to failures*. Advanced modeling and simulation techniques are the basic means of investigating fault tolerance in exascale systems, before performing the costly prototyping actions required for resilient code generation.

3.1. Modeling and simulation of failures in large-scale systems

Exascale computing provides a large-scale, heterogeneous distributed computing environment for the processing of demanding jobs. Resilience is one of the most important aspects of exascale systems. Due to the complexity of such systems, their performance is usually examined by *simulation* rather than by analytical techniques. Analytical modeling of complex systems is difficult and often requires several simplifying assumptions. Such assumptions might have an unpredictable impact on the results. For this reason, there have been many research efforts in developing tractable simulation models of large-scale systems.

In [34], simulation models are used to investigate performance issues in distributed systems where the processors are subject to failures. In this research, the author considers that failures are a Poisson process with a rate that reflects the failure probability of processors. Processor repair time has been considered as an exponentially distributed random variable with a mean value that reflects the average time required for the distributed processors to recover. The failure/ repair model of this paper can be used in combination with other models in the case of large-scale distributed processors.

3.2. Checkpointing in exascale systems

Application resilience is an important issue that must be addressed in order to realize the benefits of future systems. If a failure occurs, recovery can be handled by *checkpoint-restart (CPR)*, that is, by terminating the job and restarting it from its last stored checkpoint. There are views that this approach is not expected to scale efficiently to exascale, so different mechanisms are explored in the literature. Gamell et al. in [35] have implemented Fenix, a framework for enabling recovery from failures for MPI-based parallel applications in an online manner (i.e. without disrupting the job). This framework relies on *application-driven, diskless, implicitly-coordinated checkpointing*. Selective checkpoints are created at specific points within the application, guaranteeing global consistency without requiring a coordination protocol.

Zhao et al. in [36] investigate the suitability of a checkpointing mechanism for exascale computers, across both parallel and distributed filesystems. It is shown that a checkpointing mechanism on parallel filesystems is not suitable for exascale systems. However, the simulation results reveal that a *distributed filesystem* with local persistent storage could enable efficient checkpointing at exascale.

In [37], the authors define a model for future systems that faces the problem of *latent errors*, i.e. errors that go undetected for some time. They use their proposed model to derive optimal checkpoint intervals for systems with latent errors. The importance of a multi-version checkpointing system is explored. They conclude that a multi-version model outperforms a single checkpointing scheme in all cases, while for exascale scenarios, the multi-version model increases efficiency significantly.

Many applications in large-scale systems have an inherent need for fault tolerance and high-quality results within *strict timing constraints*. The scheduling algorithm employed in such cases must guarantee that every job will meet its deadline, while providing at the same time high-quality (i.e. precise) results. In [41], the authors study the scheduling of parallel jobs in a distributed real-time system with possible software faults. They model the system with a queuing network model and evaluate the performance of the scheduling algorithms with simulation techniques. For each scheduling policy they provide an alternative version which allows *imprecise computations*. They propose a performance metric which takes into account not only the number of jobs guaranteed, but also the precision of the results of each guaranteed job. Their simulation results reveal that the alternative versions of the algorithms outperform their respective counterparts. The authors employ the technique of imprecise computations, combined with checkpointing, in order to enhance fault tolerance in real-time systems. They consider monotone jobs that consist of a mandatory part, followed by an optional part. In order for a job to produce an acceptable result, it is required that at least the mandatory part of the job must be completed. The precision of the results is further increased, if the optional part is allowed to be executed longer. The aim is to guarantee that all jobs will complete at least their mandatory part before their deadline.

The authors employ *application-directed checkpointing*. When a software failure occurs during the execution of a job that has completed its mandatory part, there is no need to rollback and re-execute the job. In this case, the system accepts as its result the one produced by its mandatory part, assuming that a checkpoint takes place when a job completes its mandatory part. According to the research findings in [41], in large-scale systems where many software failures can occur, scheduling algorithms based on the technique of imprecise computations could be effectively employed for the fault-tolerant execution of parallel real-time jobs.

3.3. Alternative programming models for fault tolerance in exascale systems

However, programming models that enable more appropriate recovery strategies than CPR are required in exascale systems. Towards this direction, Heroux in [42] presents the following four programming models for developing new algorithms:

- ***Skeptical Programming (SkP):*** SkP requires that algorithm developers should expect that silent data corruption is possible, so that they can develop validation tests.
- ***Relaxed Bulk-synchronous Programming (RBSP):*** RBSP is possible with the introduction of MPI 3.0.
- ***Local Failure, Local Recovery (LFLR):*** LFLR provides programmers with the ability to recover locally and continue application execution when a process is lost. This model requires more support from the underlying system layers.
- ***Selective Reliability Programming (SRP):*** SRP provides the programmer with the ability to selectively declare the reliability of specific data and compute regions.

The User Level Failure Mitigation (ULFM) interface has been proposed to provide fault-tolerant semantics in MPI. In [43], the authors present their experiences on using ULFM in a case study to exploit the advantages and difficulties of this interface to program fault-tolerant

MPI applications. They found that ULFM is suitable for specific types of applications, but it provides few benefits for general MPI applications.

The issue of fault-tolerant MPI is also considered in [44]. Due to the fact that the system kills all the remaining processes and restarts the application from the last saved checkpoint when an MPI process is lost, it is expected that this approach will not work for future extreme scale systems. The authors address this scaling issue through the LFLR programming model. In order to achieve this model, they design and implement a software framework using a prototype MPI with ULFM (MPI-ULFM).

4. Code sustainability and other metrics

Software designers for supercomputers face new challenges. Their code must be efficient whatever the underlying platform is while not wasting computing time for crossing abstraction layers. Several tools presented in the previous sections provide designers and programmers with tools to abstract from the underlying hardware while achieving the maximum performance.

The clear goal to achieve is to increase the raw performance of supercomputers, but it is not anymore the simple *the faster, the better*. Two reasons show that taking care of raw performance is no more sufficient:

- Life of code is way longer than hardware life;
- Other metrics (energy, plasticity, scalability) become more and more important.

4.1. Life cycle of codes

HPC world is comprised of a few widely used codes that serve as base library and a majority of ad-hoc codes often mainly designed and programmed by non-computer scientists.

As an example for the first category, in the scientific computing domain, which aims at constructing mathematical models and numerical solution techniques for solving problems arising in science and engineering, Scalapack [51] is a largely used library of high-performance linear algebra routines for parallel distributed memory machines. It is a base of large-scale scientific codes and can run on nearly every classical supercomputers. It encompasses BLAS (Basic Linear Algebra Subprograms) and pBLAS (parallel BLAS) libraries. This package is comprised of old C and Fortran codes and was first released in 1979 from NASA [52] for BLAS and 1996 for Scalapack [51]. In the last version (version 2.0.2, May 2012) large parts of code are still dating from the first version of 1996, being raw computing code in Fortran or higher level code in C. This version also contains code from nearly every year from 1996 to 2012. This code was able to evolve up to present days due to the community of users behind it. Most other less used libraries or software did not have this chance. But even this library has several sustainability problems such as new hardware architectures: Several supercomputer projects are planning to use GPU [50] or ARM [49] processors instead of classical standard x86 ones.

Concerning the second category the difficulties are even higher, as a large number of codes has been tested and evaluated only on a handful of supercomputers. Their scalability is unknown on different networks or memory topologies for example. In this case, these codes are not sustainable as they require a major rewrite to run on new architecture.

Hence new programming paradigms such as skeletons [47], or YML [48] are needed to reach sustainable codes that run efficiently on the latest generation of supercomputers. Concerning exascale computing the situation is even more dire as the exact detail of these architectures is still cloudy.

4.2. New metrics

Further away from sustainability of the code itself, other metrics are important for designers and programmers. Power consumption of supercomputers is reaching thresholds that prevent them from continuing to grow like before [46]. The main three metrics that programmers have to confront are:

- *Raw power consumption*: Depending on the particular instructions, library, memory and network access patterns, application will consume different power consumption at particular time and different overall energy for the same work;
- *Scalability*: The capability of scaling up is key as future exascale systems will be composed of hundreds of thousands of cores;
- *Plasticity*: It is the capability of the software to adapt to the underlying hardware architecture (ARM/x86/GPU, network topology, memory hierarchy,...) but also to reconfigure itself by changing the number of allocated resources or migrating between architectures at runtime.

At the moment most tools to provide insight on the code to programmers are aiming toward raw computing performance or memory and network usage. Only a few tools exist to provide feedback to programmers on such needed metrics. Valgreen [45] offers to give insight on the power consumption of codes for example. But at the moment manual evaluation is needed in order to evaluate these metrics for any code.

Conclusion

In this article we explored and discussed programming models and runtimes required for scalable high performance computing systems that comprise a very large number of processors and threads. Currently no programming solutions exist that satisfy all the main requirements of such systems. Therefore, new programming models are required that support data locality, minimize data exchange and synchronization, while providing resilience and fault-tolerant mechanisms in order to tackle the increasing probability of failures in such large and complex systems.

New programming models and languages will be a key component of exascale systems. Their design and implementation is one of the pillars of the future exascale strategy that is based on the development of massively parallel hardware, small-grain parallel algorithms and scalable programming tools. All those components must be developed to make that strategy effective and useful. Furthermore, in order to reach actual sustainability, code must reinvent itself and be more independent of the underlying hardware.

One main element will be to create new communication channels between runtime software and development environment. Indeed the latter have all relevant high-level information concerning application structure and adaptation capabilities but they are usually lost when the time to actually run the application comes.

As exascale systems grow in computational power and scale, their resilience becomes increasingly important. Due to the complexity of such systems, fault tolerance must be achieved by employing more effective approaches than the traditional checkpointing scheme. Even though many alternative approaches have been proposed in the literature, further research is required towards this direction.

Finally, a way to provide higher abstraction from design time to execution time that will be investigated is to extend MPI standards to support this abstraction and to provide higher scalability support.

The work presented in this article has been partially supported by EU under the COST program Action IC1305, "Network for Sustainable Ultrascale Computing (NESUS)".

References

1. Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, Mara J. Garzarn, David Padua, and Christoph von Praun, Programming for parallelism and locality with hierarchically tiled arrays. In Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '06). ACM, 48-57, 2006. DOI: 10.1145/1122971.1122981.
2. William Gropp, Marc Snir. Programming for exascale computers. Computing in Science and Engineering, 15(6):27–35, 2013. DOI: 10.1109/mcse.2013.96.
3. John Jenkins, James Dinan, Pavan Balaji, Nagiza F. Samatova, and Rajeev Thakur. Enabling fast, noncontiguous GPU data movement in hybrid MPI+GPU environments. In IEEE International Conference on Cluster Computing (CLUSTER), pages 468–476, 2012. DOI: 10.1109/cluster.2012.72.
4. Jesper Larsson Träff, Antoine Rougier, and Sascha Hunold. Implementing a classic: Zero-copy all-to-all communication with MPI datatypes. In 28th ACM International Conference on Supercomputing (ICS), pages 135–144, 2014. DOI: 10.1145/2597652.2597662.
5. G. Bosilca, A. Bouteiller, and F. Cappello. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In ACM/IEEE Supercomputing Conference, page 29. IEEE, 2002. DOI: 10.1109/sc.2002.10048.
6. G. E. Fagg, A. Bukovsky, and J. J. Dongarra. HARNESS and fault tolerant MPI. Parallel Computing, 27(11):1479–1495, October 2001. DOI: 10.1016/s0167-8191(01)00100-4.
7. C. George and S. S. Vadhiyar. ADFT: An adaptive framework for fault tolerance on large scale systems using application malleability. Procedia Computer Science, 9:166–175, 2012. DOI: 10.1016/j.procs.2012.04.018.
8. J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, and R. Thakur. Hybrid parallel programming with MPI and unified parallel C. in Proceedings of the 7th ACM international conference on Computing frontiers, CF '10, 2010. DOI: 10.1145/1787275.1787323.
9. P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff, MPI on millions of cores. Parallel Processing Letters, vol. 21, no. 1, pp. 45–60, 2011. DOI: 10.1142/s0129626411000060.
10. MPI Forum: MPI: A Message-Passing Interface Standard. Version 3.0 (September 4 2012).

11. Jerry Eriksson, Radoslaw Januszewski, Olli-Pekka Lehto, Carlo Cavazzoni, Torsten Wilde and Jeanette Wilde. System Software and Application Development Environments. PRACE Second Implementation Phase Project, D11.2, 2014.

12. D. Unat, J. Shalf, T. Hoefler, T. Schulthess, A. Dubey et. al. Programming Abstractions for Data Locality. White Paper, PADAL Workshop, 28-29 April, 2014, Lugano Switzerland.

13. S. Amarasinghe, M. Hall, R. Lethin, K. Pingali, D. Quinlan, V. Sarkar, J. Shalf, R. Lucas, and K. Yelick. ASCR programming challenges for exascale computing. Report of the 2011 workshop on exascale programming challenges, University of Southern California, Information Sciences Institute, July 2011.

14. R. Lucas et. al. Top Ten Exascale Research Challenges. DOE ASCAC Subcommittee Report, February 2014.

15. C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice & Experience, 23(2):187–198, February 2011. DOI: 10.1002/cpe.1631.

16. Ang, J. A., Barrett, R. F., Benner, R. E., Burke, D., Chan, C., Cook, J., Donofrio, D., Hammond, S. D., Hemmert, K. S., Kelly, S. M., Le, H., Leung, V. J., Resnick, D. R., Rodrigues, A. F., Shalf, J., Stark, D., Unat, D. and Wright, N. J. Abstract Machine Models and Proxy Architectures for Exascale Computing. Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing. Co-HPC '14, New Orleans, Louisiana. IEEE Press 978-1-4799-7564-8, pages: 25-32. 2014. DOI: 10.1109/co-hpc.2014.4.

17. R. Thakur, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, J. Larsson Träff. MPI at Exascale. Department of Energy SciDAC workshop, Jul, 2010.

18. J. Dongarra et al. The International Exascale Software Project roadmap. International Journal of High Performance Computing Applications, 25:3–60, 2011.

19. J. Dinan, R. Grant, P. Balaji, D. Goodell, D. Miller, M. Snir, R. Thakur. Enabling communication concurrency through flexible MPI endpoints. International Journal of High Performance Computing Applications, volume 28, pages 390-405. 2014. DOI: 10.1177/1094342014548772.

20. J. Carretero, J. Garcia-Blas, D. Singh, F. Isaila, T. Fahringer, R. Prodan, G. Bosilca, A. Lastovetsky, C. Symeonidou, H. Perez-Sanchez, J. Cecilia. Optimizations to Enhance Sustainability of MPI Applications. Proceedings of the 21st European MPI Users' Group Meeting, EuroMPI/ASIA '14, Kyoto, Japan, 2014. DOI: 10.1145/2642769.2642797.

21. I. Gorton, P. Greenfield, A. Szalay, R. Williams. Data-intensive computing in the 21st century. IEEE Computer, 41 (4), 30-32. DOI: 10.1109/mc.2008.122.

22. European Commission EPiGRAM project (grant agreement no 610598). http://www.epigram-project.eu.

23. K. Dichev, F. Reid, A. Lastovetsky. Efficient and Reliable Network Tomography in Heterogeneous Networks Using BitTorrent Broadcasts and Clustering Algorithms. Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC12). Salt Lake City, Utah, USA. 2012. IEEE Computer Society Press, ISBN: 978-1-4673-0804-5, pp. 36:1–36:11. DOI: 10.1109/sc.2012.52.

24. Z. Zhong, V. Rychkov, A. Lastovetsky. Data Partitioning on Multicore and Multi-GPU Platforms Using Functional Performance Models. IEEE Transactions on Computers, PrePrints. DOI: 10.1109/TC.2014.2375202.

25. E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. Scientific Programming, 13(3):219–237, 2005. DOI: 10.1155/2005/128026.

26. K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M. P. Balcazar Vargas, S. Sufi, and C. Goble. The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. Nucleic Acids Research, 41(W1):W557–W561, July 2013. DOI: 10.1093/nar/gkt328.

27. B. Ludscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. Concurrency and Computation: Practice and Experience, 18(10):1039–1065, 2006. DOI: 10.1002/cpe.994.

28. J. Kranjc, V. Podpecan, and N. Lavrac. ClowdFlows: A Cloud Based Scientific Workflow Platform. In P. Flach, T. Bie, and N. Cristianini, editors, Machine Learning and Knowledge Discovery in Databases, LNCS 7524: 816–819. Springer, Heidelberg, Germany, 2012. DOI: 10.1007/978-3-642-33486-3_54.

29. H. Hiden, S.Woodman, P.Watson, and J. Cala. Developing cloud applications using the e-Science Central platform. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 371(1983), January 2013. DOI: 10.1098/rsta.2012.0085.

30. F. Lordan, E. Tejedor, J. Ejarque, R. Rafanell, J. Álvarez, F. Marozzo, D. Lezzi, R. Sirvent, D. Talia, and R. M. Badia. ServiceSs: An Interoperable Programming Framework for the Cloud. Journal of Grid Computing, vol. 12, n. 1, pp. 67-91, 2014. DOI: 10.1007/s10723-013-9272-5.

31. J. Diaz, C. Munoz-Caro and A. Nino. A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era, Parallel and Distributed Systems, IEEE Transactions on , vol.23, no.8, pp.1369-1386, Aug. 2012. DOI: 10.1109/tpds.2011.308.

32. K. Maheshwari and J. Montagnat. Scientific workflow development using both visual and script-based representation. In Proceedings of the 2010 6th World Congress on Services, SERVICES '10, pages 328–335, Washington, DC, USA, 2010. DOI: 10.1109/services.2010.14.

33. F. Marozzo, D. Talia, P. Trunfio, "JS4Cloud: Script-based Workflow Programming for Scalable Data Analysis on Cloud Platforms". Concurrency and Computation: Practice and Experience, Wiley InterScience, 2015. DOI: 10.1002/cpe.3563.

34. H. D. Karatza. Performance analysis of a distributed system under time-varying workload and processor failures. Proceedings of the 1st Balkan Conference on Informatics (BCI'03), Nov. 2003, Thessaloniki, Greece , pp. 502–516.

35. M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar. Exploring automatic, online failure recovery for scientific applications at extreme scales. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14), Nov. 2014, New Orleans, USA, pp. 895–906. DOI: 10.1109/sc.2014.78.

36. D. Zhao, D. Zhang, K. Wang, and I. Raicu. Exploring reliability of exascale systems through simulations. Proceedings of the High Performance Computing Symposium (HPC'13), Apr. 2013, San Diego, USA, pp. 1–9.

37. G. Lu, Z. Zheng, and A. A. Chien. When is multi-version checkpointing needed? Proceedings of the 3rd Workshop on Fault-tolerance for HPC at Extreme Scale (FTXS'13), Jun. 2013, New York, USA, pp. 49–56. DOI: 10.1145/2465813.2465821.

38. Numrich, Robert W., and John Reid. Co-Array Fortran for parallel programming. ACM Sigplan Fortran Forum. Vol. 17. No. 2. ACM, 1998. DOI: 10.1145/289918.289920.

39. F. Marozzo, D. Talia, P. Trunfio. Cloud Services for Distributed Knowledge Discovery. In: Encyclopedia of Cloud Computing, S. Murugesan, I. Bojanova (Editors), Wiley-IEEE, 2016.

40. El-Ghazawi, Tarek, and Lauren Smith. UPC: unified parallel C. Proceedings of the 2006 ACM/IEEE conference on Supercomputing. ACM, 2006. DOI: 10.1145/1188455.1188483.

41. G. L. Stavrinides and H. D. Karatza. Fault-tolerant gang scheduling in distributed real-time systems utilizing imprecise computations. Simulation: Transactions of the Society for Modeling and Simulation International, vol. 85, no. 8, 2009, pp. 525–536. DOI: 10.1177/0037549709340729.

42. M. A. Heroux. Toward resilient algorithms and applications. arXiv:1402.3809, March 2014.

43. I. Laguna, D. F. Richards, T. Gamblin, M. Schulz, and B. R. de Supinski. Evaluating user-level fault tolerance for MPI applications. Proceedings of the 21st European MPI Users' Group Meeting (EuroMPI/ASIA'14), Sep. 2014, Kyoto, Japan, pp. 57–62. DOI: 10.1145/2642769.2642775.

44. K. Teranishi and M. A. Heroux. Toward local failure local recovery resilience model using MPI-ULFM. Proceedings of the 21st European MPI Users' Group Meeting (EuroMPI/ASIA'14), Sep. 2014, Kyoto, Japan, pp. 51–56. DOI: 10.1145/2642769.2642774.

45. Leandro Fontoura Cupertino, Georges Da Costa, Jean-Marc Pierson. Towards a generic power estimator. In : Computer Science - Research and Development, Springer Berlin / Heidelberg, Special issue : Ena-HPC 2014, July 2014. DOI: 10.1007/s00450-014-0264-x.

46. Shalf, John, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. High Performance Computing for Computational Science–VECPAR 2010. Springer Berlin Heidelberg, 2011. 1-25. DOI: 10.1007/978-3-642-19328-6_1.

47. Cole, Murray. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel computing 30 (3), Elsevier 2004: pages 389-406. DOI: 10.1016/j.parco.2003.12.002.

48. Petiton, S., Sato, M., Emad, N., Calvin, C., Tsuji, M., and Dandouna, M. Multi level programming Paradigm for Extreme Computing. In SNA+ MC 2013-Joint International Conference on Supercomputing in Nuclear Applications+ Monte Carlo, 2014. EDP Sciences. DOI: 10.1051/snamc/201404305.

49. Ramirez, A. (2011). European scalable and power efficient HPC platform based on low-power embedded technology. On-Line. Access date: March/2012. URL: `http://www.eesi-project.eu/media/BarcelonaConference/Day2/13-Mont-Blanc_Overview.pdf`.

50. Nukada, Akira, Kento Sato, and Satoshi Matsuoka. Scalable multi-gpu 3-d fft for tsubame 2.0 supercomputer. Proceedings of the International Conference on High Performance

Computing, Networking, Storage and Analysis. IEEE Computer Society Press, 2012. DOI: 10.1109/sc.2012.100.

51. Choi, Jaeyoung, James Demmel, Inderjiit Dhillon, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, Ken Stanley, David Walker, and R. Clinton Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers—Design issues and performance. In Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science, pp. 95–106. Springer Berlin Heidelberg, 1996. DOI: 10.1007/3-540-60902-4_12.
52. Lawson, Chuck L., Richard J. Hanson, David R. Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for Fortran usage. ACM Transactions on Mathematical Software (TOMS) 5, no. 3 (1979): 308–323. DOI: 10.1145/355841.355847.
53. OCL-MLA, `http://tuxfan.github.com/ocl-mla/`.
54. Bariş Eskikaya and D Turgay Altilar, "Distributed OpenCL Distributing OpenCL Platform on Network Scale", IJCA 2012, pp. 26–30.
55. Philipp Kegel, Michel Steuwer and Sergei Gorlatch, "dOpenCL: Towards a Uniform Programming Approach for Distributed Heterogeneous Multi-/Many-Core Systems" IPDPS Workshops 2012. DOI: 10.1109/ipdpsw.2012.16.
56. Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo and Jaejin Lee, "SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters", ICS 2012. DOI: 10.1145/2304576.2304623.
57. Shucai Xiao and Wu-chun Feng, "Generalizing the Utility of GPUs in Large-Scale Heterogeneous Computing Systems", IPDPS Workshops, 2012. DOI: 10.1109/ipdpsw.2012.325.
58. Ridvan Özaydin and D. Turgay Altilar, "OpenCL Remote: Extending OpenCL Platform Model to Network Scale", HPCC-ICESS 2012. DOI: 10.1109/hpcc.2012.117.
59. Alves Albano, Rufino Jose, Pina Antonio and Santos Luis Paulo, "Enabling task-level scheduling on heterogeneous platforms", Workshop GPGPU, 2012.
60. Khronos OpenCL Working Group, "The OpenCL 1.2 specification", 2012, `http://www.khronos.org/opencl`.
61. José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo and Enrique S. Quintana-Ortí, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters", HPCS 2010. DOI: 10.1109/hpcs.2010.5547126.
62. Orion S. Lawlor, "Message passing for GPGPU clusters: CudaMPI", CLUSTER 2009. DOI: 10.1109/clustr.2009.5289129.
63. Chamberlain, Bradford L., David Callahan, and Hans P. Zima. Parallel programmability and the chapel language. International Journal of High Performance Computing Applications 21.3 (2007): 291-312. DOI: 10.1177/1094342007078442.
64. Charles, Philippe, et al. X10: an object-oriented approach to non-uniform cluster computing. ACM Sigplan Notices 40.10 (2005): 519-538. DOI: 10.1145/1103845.1094852.
65. Sun Enqiang, Schaa Dana, Bagley Richard, Rubin Norman and Kaeli David, "Enabling task-level scheduling on heterogeneous platforms", WORKSHOP GPGPU 2012. DOI: 10.1145/2159430.2159440.
66. Magnus Strengert, Christoph Müller, Carsten Dachsbacher and Thomas Ertl, "CUDASA: Compute Unified Device and Systems Architecture", EGPGV 2008.

67. Bueno Javier, Planas Judit, Duran Alejandro, Badia Rosa M., Martorell Xavier, Ayguade Eduard and Labarta Jesus, "Productive Programming of GPU Clusters with OmpSs", IPDPS 2012. DOI: 10.1109/ipdps.2012.58.
68. Ryo Aoki, Shuichi Oikawa, Takashi Nakamura and Satoshi Miki, "Hybrid OpenCL: Enhancing OpenCL for Distributed Processing", ISPA 2011. DOI: 10.1109/ispa.2011.28.
69. A. Barak, T. Ben-Nun, E. Levy and A. Shiloh, "A Package for OpenCL Based Heterogeneous Computing on Clusters with Many GPU Devices", Workshop PPAC 2010. DOI: 10.1109/clusterwksp.2010.5613086.
70. Simple-opencl `http://code.google.com/p/simple-opencl/`.
71. Ivan Grasso, Simone Pellegrini, Biagio Cosenza, Thomas Fahringer. "libwater: Heterogeneous Distributed Computing Made Easy", ACM International Conference on Supercomputing, Eugene, USA, 2013. DOI: 10.1145/2464996.2465008.
72. Nesus European Cost Action IC1305 `http://www.nesus.eu/`.
73. NCF, Peter Michielse, and Patrick Aerts NCF. "European Exascale Software Initiative".
74. Dongarra, Jack. "The international exascale software project roadmap". International Journal of High Performance Computing Applications (2011): 1094342010391989. APA.
75. Computing Language Utility, Intel Corporation, `http://software.intel.com/`.

Received February 27, 2015.