Review Article

Survey of Methodologies, Approaches, and Challenges in Parallel Programming Using High-Performance Computing Systems

Paweł Czarnul [ID],[1] Jerzy Proficz,[2] and Krzysztof Drypczewski [ID][2]

[1]*Dept. of Computer Architecture, Faculty of Electronics, Telecommunications and Informatics, Gdansk University of Technology, Gdańsk, Poland*
[2]*Centre of Informatics–Tricity Academic Supercomputer & Network (CI TASK), Gdansk University of Technology, Gdańsk, Poland*

Correspondence should be addressed to Paweł Czarnul; pczarnul@eti.pg.edu.pl

This paper provides a review of contemporary methodologies and APIs for parallel programming, with representative technologies selected in terms of target system type (shared memory, distributed, and hybrid), communication patterns (one-sided and two-sided), and programming abstraction level. We analyze representatives in terms of many aspects including programming model, languages, supported platforms, license, optimization goals, ease of programming, debugging, deployment, portability, level of parallelism, constructs enabling parallelism and synchronization, features introduced in recent versions indicating trends, support for hybridity in parallel execution, and disadvantages. Such detailed analysis has led us to the identification of trends in high-performance computing and of the challenges to be addressed in the near future. It can help to shape future versions of programming standards, select technologies best matching programmers' needs, and avoid potential difficulties while using high-performance computing systems.

1. Introduction

In today's high-performance computing (HPC) landscape, there are a variety of approaches to parallel computing that enable reaching the best out of available hardware systems. Multithreaded and multiprocess programming is necessary in order to make use of the growing computational power of such systems that is available mainly through the increase of the number of cores, cache memories, and interconnects such as Infiniband or NVLink [1]. However, existing approaches allow programming at various levels of abstraction that affects ease of programming, also through either one-sided or two-sided communication and synchronization modes, targeting shared or distributed memory HPC systems. In this work, we discuss state-of-the-art methodologies and approaches that are representative of these aspects. It should be noted that we describe and distinguish the approaches by programming methods, supported languages, supported platforms, license, ease of programming,

deployment, debugging, goals, parallelism levels, and constructs including synchronization. Then, based on detailed analysis, we present current trends and challenges for development of future solutions in contemporary HPC systems.

Section 2 motivates this paper and characterizes the considered APIs in terms of the aforementioned aspects. Subsequent sections present detailed discussion of APIs that belong to particular groups, i.e., multithreaded processing in Section 3, message passing in Section 4, Partitioned Global Address Space in Section 5, agent-based parallel processing in Section 6 and MapReduce in Section 7. Section 8 provides detailed classification of approaches. Section 9 discusses trends in the development of the APIs including latest updates and changes that correspond to development directions as well as support for hybrid processing, very common in contemporary systems. Based on our extensive analysis, we formulate challenges in the field in Section 10. Section 11 presents existing comparisons, especially

performance oriented, of subsets of the considered APIs for selected practical applications. Finally, summary and planned future work are included in Section 12.

2. Motivation

In this paper, we aim at identifying key processing paradigms and their representatives for high-performance computing and investigation of trends as well as challenges in this field for the near future. Specifically, we distinguish the approaches by the types of systems they target, i.e., shared memory, distributed memory, and hybrid ones. This aspect typically refers to workstation/server, clusters, and systems incorporating various types of compute devices, respectively.

Communication paradigms are request-response/two-sided vs one-sided communication models. This aspect defines the type of a parallel programming API.

Abstraction level in terms of detailed level of communication and synchronization routines invoked by components is executed in parallel. This aspect is related to potential performance vs ease of programming of a given solution, i.e., high performance at the cost of more difficult programming for low level vs lower performance with easier programming using high-level constructs. Specifically, this approach distinguishes the following groups: low-level communication (just basic communication API), APIs with interthread, interprocess synchronization routines (MPI, OpenMP, etc.) that still requires much knowledge and awareness of the environment as well as framework-level programming. The authors realize that the presented assessment of ease of programming is subjective; nevertheless, it is clear that aspects like the number of lines of code to achieve parallelization are correlated with technology abstraction level.

The considered approaches and technologies have been superimposed on a relevant diagram and shown in Figure 1. We realize that this is our subjective selection, with many other available technologies like C++11 thread.h library [2] or Threading Building Blocks [3] (TBBs), High Performance ParalleX [4] (HPX), and others. However, we believe that the above collection consists of representative technologies/APIs and can be used as a strong base for the further analysis. Moreover, selection of these solutions is justified by the existence of comparisons of subsets of these solutions presented in Section 11 and discussed in other studies.

Data visualization is an important part of any HPC system, and GPGPU technologies such as OpenGL and DirectX received a lot of attention in recent years [5]. Even though they can be used for general purpose computations [6], the authors do not perceive those approaches to become the main track of the HPC technology.

3. Multithreaded Processing

In the current landscape of popular parallel programming APIs aimed at multicore and many-core CPUs, accelerators such as GPUs, and hybrid systems, there are several popular solutions [1] and descriptions of the most important ones in the following.
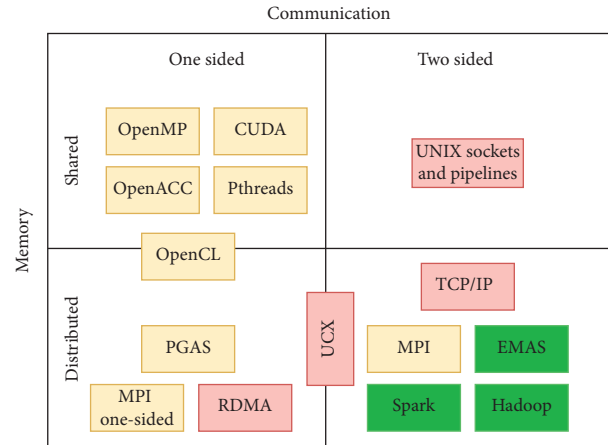

FIGURE 1: Abstraction level marked with colors: high, green; middle, yellow; low, red.

3.1. OpenMP. OpenMP [7] allows development and execution of multithreaded applications that can exploit multicore and many-core CPUs within a node. Latest OpenMP versions also allow offloading fragments of code to accelerators including GPUs. OpenMP allows relatively easy extension of sequential applications into a parallel application using two types of constructs: library functions that allow determination of the number of threads executing a region in parallel or thread ids and directives that instruct how to parallelize or synchronize execution of regions or lines of code. Mostly used directives include #pragma omp parallel spawning threads working in parallel in a given region as well as #pragma omp for allowing assignment of loop iterations to threads in a region for parallel processing. Various scheduling modes including static and dynamic with predefined chunk sizes with a guided mode with a decreasing chunk size are also available. It is also possible to find out the number of threads and unique thread ids in a region for fine-grained assignment of computations. OpenMP allows for synchronization through constructs such as critical sections, barrier, and atomic and reduction clauses. Latest versions of OpenMP support a task model in which a thread working in a parallel region can spawn tasks which are automatically assigned to available threads for parallel processing. A wait-directive imposing synchronization is also available [1].

3.2. CUDA. CUDA [8] allows development and execution of parallel applications running on 1 or more NVIDIA GPUs. Computations are launched as kernels that operate on and produce data. Synchronization of kernels and GPUs is performed through the host side. Parallel processing is executed by launching a grid of threads which are grouped into potentially many thread blocks. Threads within a block can be synchronized and can use faster albeit much smaller shared memory, compared to the global memory of a GPU. Shared memory can be used as a cache for intermediate storage of data as can be registers. When operating on data chunks, data can be fetched from global memory to registers

and from registers to shared memory to allow data prefetching. Similarly, RAM to GPU memory communication, computations within a kernel and GPU memory to RAM communication can be overlapped when operations are launched to separate CUDA streams. On the other hand, selection of the number of threads in a block can have an impact on performance as it affects the total block requirements for the number of registers and shared memory and considering limits on the numbers of registers and amount of shared memory per Streaming Multiprocessor (SM), it can affect the number of resident blocks and level of parallelization. Modern cards with high compute capability along with new CUDA toolkit versions allow for dynamic parallelism allowing launching a kernel from within a kernel as well as Unified Memory between the host and the GPU. CPU + GPU parallelization, similar to OpenCL, requires cooperation with another multithreading CPU API such as OpenMP or Pthreads. Multi-GPU systems can be handled with the API which allows to set a given active GPU which allows to do it from either one or many host threads to handle such systems. NVIDIA provides CUDA MPS for automatic overlapping and scheduling calls to even a single GPU from many host processes using e.g. MPI for interprocess communication.

3.3. OpenCL. OpenCL [9] allows development and execution of multithreaded applications that can exploit several compute devices within a computing platform such as a server with multiple multicore and many-core CPUs as well as GPUs. Computations are launched as kernels that operate on and produce data, within a so-called context defined for one or more compute devices. Work items within potentially many work groups execute the kernel in parallel. Memory objects are used to manage data within computations. OpenCL uses an analogous structure of an application to CUDA where work items correspond to threads and work groups to thread blocks. Similarly to CUDA, work items within a group can be synchronized. Since OpenCL extends the idea of running kernels on many and various (such as CPUs and GPUs) devices, it typically requires many more lines of device management code than a CUDA program. Similarly to CUDA streams, OpenCL uses the concept of command queues, into which commands such as data copy or kernel launches can be inserted. Many command queues can be used and execution can be synchronized by referring to events that are associated with commands. Additionally, the so-called local memory (similarly to what is called shared memory in CUDA) can be shared among work items within a single work group for fast access as cache-type memory. Shared virtual memory allows us to share complex data structures by the host and device sides.

3.4. Pthreads. Pthreads [1] allows development and execution of multithreaded applications on multicore and many-core CPUs. Pthreads allows a master thread to call a function that launches threads that execute code of a given function in parallel and then join the execution of the threads. The Pthreads API offers a wide array of functions, especially related to synchronization. Specifically, mutexes are mutual exclusion variables that can control access to a critical section in the case of several threads. Furthermore, the so-called condition variables along with mutexes allow a thread to wait on a condition variable if a condition is not met. Another thread that has changed the condition can wake up a thread or several threads waiting for the condition. This scheme allows implementation of, e.g., the producer-consumer pattern without busy waiting. In this respect, Pthreads allows expression of more complex synchronization patterns than, e.g., OpenMP.

3.5. OpenACC. OpenACC [10] allows development and execution of multithreaded applications that can exploit GPUs. OpenACC can be seen as similar to OpenMP [11] in terms of abstraction level but focused on parallelization on GPUs through directives instructing parallelization of specified regions of code, scoping of data, and synchronization as well as library function calls. The basic #pragma acc parallel directive specifies execution of the following block by one or more gangs, each of which may run one or more workers. Another level of parallelism includes vector lanes within a worker. A region can be marked as one that can be executed by a sequence of kernels done with #pragma acc kernels while parallel execution of loop iterations can be specified with #pragma acc loop. Directives such as #pragma acc data can be used for data management with specification of allocation copy and freeing space on a device. Reference counters to data are used. An atomic #pragma acc directive can be used for accessing data.

3.6. Java and Scala. Java [12] and Scala [13] are Java Virtual Machine- (JVM-) [14] based languages; both are translated into JVM byte codes and interpreted or further compiled into a specific hardware instruction set. Thus, it is natural that they share common mechanisms for supporting the concurrent program execution. They provide two abstraction levels of the concurrency, the lower one which is related directly to operating system/hardware-based threads of control and the higher one, where the parallelism is hidden by the executor classes, which are used to schedule and run user-defined tasks.

A Java thread can be used when direct control of the concurrency is necessary. Its life cycle is strictly controlled by the programmer; he or she can create and provide its content: the list instructions to be executed concurrently, monitor, interrupt, and finish. Additionally, API is provided that supports thread interactions, including synchronization and in-memory data exchange.

The higher level concurrency objects support parallel code execution in more complex applications, where the fine level of thread control is not necessary, but parallelization can be easily provided for larger groups of compute tasks. Concurrent collections can be used for parallel access to in-memory data, lock classes enable nonblocking access to synchronized code, atomic variables help to minimize synchronization and avoid consistency issues, and the executor classes manage thread pools for queuing and scheduling of compute tasks.

4. Message Passing Processing

4.1. Low-Level Communication Mechanisms. The low-level communication mechanisms are used by HPC frameworks and libraries to enable data transmission and synchronization control. From the network point of view, the typical approach is to provide a network stack, with the layers corresponding to different levels of abstraction. TCP/IP is a classical stack provided in the most modern systems, not only in HPC. Usually, its main goal is to provide means to exchange data with external systems, i.e., Internet access; however, it can be also used to support computations directly as a medium of data exchange.

Nowadays, TCP/IP [15] can be perceived as a reference network stack, although the ISO-OSI is still reminded to be used for this purpose [16]. Figure 2(a) presents the TCP/IP layer structure: link—the lowest one is responsible for handling hardware, IP—the second one provides simple routed transmission of the packages, transport—the third one is usually used by the communication frameworks or directly by the applications for either connection-based protocol: Transmission Control Protocol (TCP) or for connection-less datagram transmission: User Datagram Protocol (UDP).

The other, quite often application/framework API used in HPC, is Remote Direct Memory Access (RDMA) [17]. Similarly to the TCP/IP, its stack has layered structure and its lowest layer link is responsible for handling the hardware. Currently, two main hardware solutions are used: Infiniband, the intraconnecting network characterized by multicast support, high bandwidth, low latency, and an extended version of Ethernet, with RDMA over Converged Ethernet v1 (RoCEv1) protocol [18], where multicast transmission is supported in a local network (see Figure 2(b)). The test results presented in [19] showed performance advantages of a pure Infiniband solution; however, introduction of RoCE enabled great latency reduction in comparison with classical Ethernet.

Figure 2(c) presents RDMA over Converged Ethernet v2 (RoCEv2) [20], where RDMA is deployed over plain IP stack, on top of the UDP protocol. In this case, some additional requirements over the protocol implementation are introduced: ordering of the transmitted messages and some congestion control mechanism. Usage of UDP packets, which are routable, implies that the communication is not limited to one local network, and that is why RoCEv2 is sometimes called Routable RoCE (RRoCE).

The Unified Communication X (UCX) [21] is a network stack providing a collection of APIs dedicated to support different middleware frameworks: Message Passing Interface (MPI) implementations, Partitioned Global Address Space (PGAS) languages, task-based paradigms, and I/O bound applications. This initiative is a combined effort of the US national laboratories, industry, and academia. Figure 2(d) presents its architecture with link layer split into hardware and driver parts, where the former is responsible for physical connection and the latter provides vendor-specific functionality used by the higher layer, which is represented by two APIs: UC-T supporting low level hardware-transport functionality and UC-S with common utilities. Finally, the highest layer provides UC-P collections of protocols, where

specific platforms or even applications can find the proper communication and/or synchronization mechanisms.

The UCX reference implementation presented promising results in performed benchmarks, showing the measurements being very close to the underlying driver capabilities, as well as providing the highest publicly known bandwidth for a given hardware. The above results were confirmed by benchmarks executed on OpenSHMEM [22] PGAS platform, where, on the Cray XK, in most test cases, UCX implementation outperformed the one provided by the vendor [23]. In [24], comparison of performance of UC-P and UC-T on InfiniBand is presented. Even though UC-T was more efficient, optimizations proposed by Papadopoulou et al. suggest that there is a room to improve performance of higher level UC-P.

Finally, for the sake of completeness, we need to mention UNIX sockets and pipeline mechanisms [25], which are quite similar to TCP/IP ones; however, they work locally within a boundary of a single server/workstation, managed by the UNIX-based operating system. Usually, the sockets support stream and datagram messaging, similar to the TCP/IP approach, but since they work on the local machine, the data transfer is reliable and properly sequenced. The pipelines provide a convenient method for data tunneling between the local processes and usually correspond to the standard output and input streams.

4.2. MPI. The Message Passing Interface (MPI) [26] standard was created for enabling development and running parallel applications spanning several nodes of a cluster, a local network, or even distributed nodes in grid-aware MPI versions. MPI allows communication among processes or threads of an MPI application primarily by exchanging messages—message passing. MPI is a standard, and there are several popular implementations of the standard, examples of which are MPICH [27] and OpenMPI [28].

Key components of the standard, in the latest 3.1 version, define and include the following:

(1) Communication routines: point-to-point as well as collective (group) calls
(2) Process groups and topologies
(3) Data types including calls for definition of custom data types
(4) Communication contexts, intracommunicators, and intercommunicators for communication within a group or among groups of processes
(5) Creation of processes and management
(6) One-sided communication using memory windows
(7) Parallel I/O for parallel reading and writing from/to files by processes of a parallel application
(8) Bindings for C and Fortran

5. Partitioned Global Address Space

Partitioned Global Address Space (PGAS) is an approach to perform parallel computations using a system with

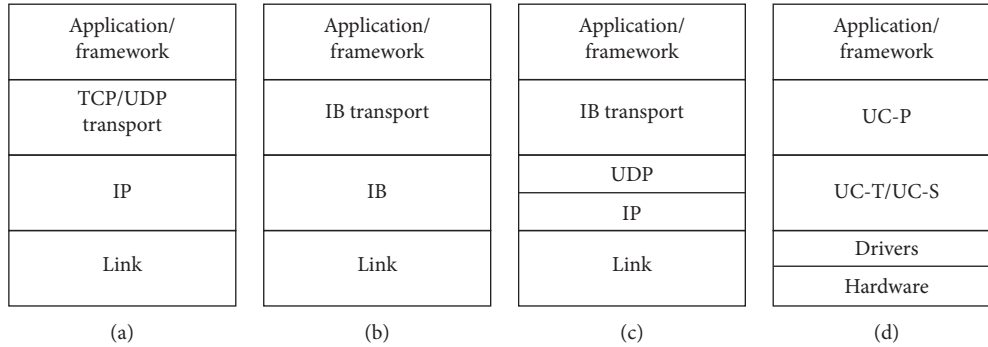| Application/ framework	Application/ framework	Application/ framework	Application/ framework
TCP/UDP transport	IB transport	IB transport	UC-P
IP	IB	UDP	UC-T/UC-S
		IP	
Link	Link	Link	Drivers
			Hardware
(a)	(b)	(c)	(d)

Figure 2: Main network stacks used in HPC: (a) TCP/IP, (b) RDMA over Infiniband/RoCEv1, (c) RDMA over RoCEv2, and (d) UCX.

potentially distributed memory. The access to the shared variables is possible by a special API, supported by a middleware implementing data transmission, synchronization, and possible optimization, e.g., data prefetch. Such a way of communication, when the data used by many processes are updated only by one, without activities taken by the other is called one-sided communication.

The classical example of PGAS realization is Open-SHMEM [22] specification, which provides a C/Fortran API for data exchange and process synchronization with distributed shared memory. Each process, potentially assigned to a different node, can read and modify a common pool of the variables as well as use a set of synchronization functions, e.g., invoking barrier before data access. This initiative is supported by a number of organizations including CRAY, HPE, Intel, Mellanox, US Department of Defense, and Stony Brook University. The latter one is responsible for an OpenSHMEM reference implementation.

Another notable PGAS implementation is Parallel Computing in Java [29] (PCJ), providing a library of functions and dedicated annotations for distributed memory access over an HPC cluster. The proposed solution uses Java language constructs like classes, interfaces, and annotations for storing and exchanging common data between the cooperating processes, potentially placed in different Java Virtual Machines on separated cluster nodes. There are other typical PGAS mechanisms like barrier synchronization or binomial tree-based vector reduction. The executed tests showed good performance of the proposed solution in comparison with an MPI counterpart.

In our opinion, the above selection consists of representative examples of PGAS frameworks; however, there are many more implementations of this paradigm, e.g., Chapel [30] and X10 [31] parallel programming languages, Unified Parallel C [32] (UPC), or C++ Standard Template Adaptive Parallel Library [33] (STAPL).

6. Agent-Based Parallel Processing

Soft computing is a computing paradigm that allows solving problems with an approach similar to the way a human mind reasons and provides good enough approximations instead of precise answers. Soft computing includes many computing techniques including machine learning, fuzzy logic, Bayesian networks, and genetic and evolutionary algorithms. A multiagent system (MAS) is a soft computing system that consists of an environment and a set of agents. Agents communicate, negotiate, and cooperate with each other and act in way that can change their own state or the state of the environment. MAS aims to provide solutions acquired from knowledge base acquired from evolutionary process. Kisiel-Dorohinicki et al. [34] distinguished the following complexity-based MAS types: (1) traditional model based of fuzzy logic in which evolution occurs on the agent level, (2) evolutionary multiagent systems (EMAS) in which evolution occurs on population level of homogeneous agents, and (3) MAS with heterogeneous agents that use different types of soft computing methods. Kisiel-Dorohinicki [35] proposed a decentralized EMAS model based on an M-Agent architecture. Agents have profiles that inform about actions taken. Profiles consist of knowledge about environment, acquired resources, goals, and strategies to be achieved. In EMAS, similarly to organic evolution, agents can reproduce (create new, sometimes changed agents) and die according to agent fitness and changes in the environment. Selection for reproduction and death is a nontrivial problem, since agents have their autonomy and there is no global knowledge. Agents obtain nonrenewable resource called life energy that is obtained as a reward or lost as a penalty. Energy level specifies actions that agents can perform.

Several general purpose agent modeling frameworks were proposed. Repast [36] is an open-source toolkit for agent simulation. It provides functionality for data analysis with special focus on agent storage, display, and behavior. Repast scheduler is responsible for running user-defined "actions," i.e., agent actions. The scheduler orders actions in tree structures that describe execution flow. This allows for dynamic scheduling during model tick, i.e., an action performed by an agent generates a new action in response [37]. Repast HPC aims to provide Repast functionality in an HPC environment. It uses a scheduler that sorts agent actions (timeline and relation between agent relations) and MPI to parallelize computations. Each process typically handles one or more agents and is responsible for executing local actions. Then, the scheduler aggregates information for the current tick and enables communication between related agents [38]. EUR-ACE is an agent-based system project that tries to model European Economy [39]. Agents communicate between each

other by sending messages. To reduce the amount of data exchanged between agents, it groups them into local groups. It leverages the idea that agents will typically communicate with a small number of other agents that should be processed as closely as possible (i.e., different processes on the same machine instead of different cluster nodes).

7. MapReduce Processing

7.1. Apache Hadoop. Apache Hadoop is a programming tool and framework allowing distributed data processing. It is an open source implementation of Google's MapReduce [40]. The Hadoop MapReduce programming model is dedicated for processing large amounts of data. Computation is split into small tasks that are executed in parallel in the machines of the cluster. Each task is responsible for processing only small part of data and thus reducing resource requirements. This approach is very scalable and can be used both on high end and commodity hardware. Hadoop handles all typical problems connected with data processing like fault tolerance (repeating computation that failed), data locality, scheduling, and resource management.

First Hadoop versions were designed and tailored for handling web crawler processing pipelines which provided some challenges for adoption of MapReduce for wider types of problems. Vavilapalli et al. [41] describe design and capabilities of Yet Another Resource Negotiator (YARN) that aims to disjoin resource management and programming model and provide extended scheduling settings. YARN moves from original architecture to match modern challenges such as better scalability, multiuser support ("multitenancy"), serviceability (ability to perform a "rolling upgrade"), locality awareness (moving computation to data), reliability, and security. YARN [42] also includes several types of basic mechanisms for handling resource requests: FAIR, FIFO, and capacity schedulers.

Apache Hadoop was deployed by Yahoo in early 2010s and achieved high utilization on a large cluster. Nevertheless, energy efficiency was not satisfactory, especially when heavy loads were not present. Leverich and Kozyrakis [43] pointed out that (due to its replication mechanism) Hadoop Distributed Files System (HDFS) precluded scaling down clusters. The authors proposed a solution in which a cluster subset must contain blocks of all required data, thus allowing processing only on this subset of nodes. Then, additional nodes can be added if needed and removed when load is reduced. This approach allowed for reducing power consumption even up to 50% but the achieved energy efficiency was accompanied with diminished performance.

Advancements in high-resolution imaging and decrease in cost of computing power and IoT sensors lead to substantial growth in the amount of generated spatial data. Aji et al. [44] presented Hadoop-based geographical information system (GIS) for warehousing large-scale spatial datasets that focuses on expressive and scalable querying. The proposed framework parallelizes spatial queries and maps them to MapReduce jobs. Hadoop GIS includes mechanism for boundary handling especially in context of data partitioning.

In recent years, several algorithms extending capabilities of Hadoop MapReduce were proposed [45]. Hadoop schedulers do not allow setting time constraints for job execution. Kc and Anyanwu [46] present a scheduling algorithm for meeting deadlines that ensures that only jobs that can finish in user-defined time frame are scheduled. The algorithm takes into consideration the number of available map and reduces task slots for a job that has to process the set amount of data and estimates if the deadline can be kept on the cluster of a predefined size. Ghodsi et al. [47] proposed the Domain Resource Fairness (DRF) allocation algorithm for providing fair share in a system with heterogeneous resources (e.g., two jobs may require similar memory, but different amount of CPU time). DRF aims to provide dominant share, i.e., demands weight which mostly depends on max-min fairness of dominant resource. Longest Approximate Time to End (LATE) [48] scheduling policy aims to offer better performance for heterogeneous clusters. LATE does not assume that that tasks progress linearly or that each machine in cluster has the same performance (which is important in virtualized environments). In case of tasks that perform slower than expected ("stragglers"), Hadoop runs duplicate ("speculative") task on different nodes to speed up processing. LATE improves the heuristics that recognize stragglers by taking into consideration not only the current progress of the task but also the progress rate.

7.2. Apache Spark. Hadoop MapReduce became a very popular platform for distributed data processing of large datasets. Even though its programming model is not suitable for several types of applications. An example of those would be interactive operations on data sets such as data mining or fast custom querying and iterative algorithms. In the first case, intermediate processing results could be saved in memory instead of being recomputed, thus improving performance. In the second case of input data, iterative map tasks read input data for each iteration, thus requiring repetitive, costly disk operations.

Apache Spark is a cluster computing framework designed to solve the aforementioned issues and allow MapReduce style operations on streams. It was proposed in 2010 [49] by AMPLab and later became Apache Foundation project. Similarly to MapReduce, the Spark programming model allows the user to provide directed acyclic graph of tasks which are executed on the machines of the cluster.

The most important part of Spark is the concept of a Resilient Distributed Dataset (RDD) [50], which represents an abstraction for a data collection that is distributed among cluster nodes. RDD provides strong typing and ability to use lazily evaluated lambda functions on the elements of the dataset.

The Apache Spark model is versatile enough to allow us to run diverse types of applications and many big data processing platforms run heterogeneous computing hardware. Despite that, most big data oriented schedulers expect to run in an homogeneous environment both in context of applications and hardware. Heterogeneity-Aware Task

Scheduler RUPAM [51] takes into consideration not only standard parameters like CPU, RAM, and data locality but also include parameters like disk type (HDD/SDD), availability of GPU or accelerator, and access to remote storage devices. RUPAM reduced execution time up to 3.4 times in the tested cluster.

Spark allows multiple tasks to be run on a machine in the cluster. To improve performance, the colocation strategy must take into account characteristics of task's resource requirements. For example, if a task receives more RAM that it requires, the cluster throughput is reduced. If a task does not receive enough memory, it will not be able to finish, thus also affecting total performance. Due to this, developers often overestimate their requirements from schedulers. The strategy used by typical colocation managers to overcome these problems requires detailed resource usage data for each task type provided in situ or gathered from statistical or analytical models. Marco et al. [52] suggest a different approach using memory aware task colocation. Using machine learning, the authors created an extensive model for different types of tasks. The model is used during task execution to estimate its behavior and future resource requirements. The proposed solution increases average system throughput over 8x.

Similarly to Hadoop MapReduce, Spark recognizes tasks for which execution times are longer that expected. To improve performance, Spark uses speculative task execution to launch duplicates of slower tasks so that job can finish in a timely manner. This algorithm does not recognize sluggers, i.e., machines that run slower than other nodes in the cluster. To solve this problem, Data-Based Multiple Phases Time Estimation [53] was proposed. It provides Spark with information about estimated time of tasks which allows speculative execution to avoid slower nodes and increase of execution time up to 10.5%.

8. Classification of Approaches

In order to structure the knowledge about the approaches and exemplary APIs representing the approaches, we provide classifications in three groups, by

(1) Abstraction level, programming model, language, supported platforms and license in Table 1—we can see that approaches at a lower level of abstraction support mainly C and Fortran, sometimes C++ while at a higher level distributed ones, Java/Scala

(2) Goals (performance, energy, etc.), ease of programming, debugging, and deployment as well as portability in Table 2—we can see that ease of programming, debugging, and deployment increase with the level of abstraction

(3) Level of parallelism, constructs expressing parallelism, and synchronization in Table 3—the latter ones are easily identified and are supported for all the presented approaches

We note that classification of the approaches and APIs in terms of target system types, distributed and shared memory systems, is shown in Figure 1. The APIs targeting accelerators are intentionally regarded as shared memory referring to the device memory.

9. Trends in Scientific Parallel Programming

There are several sources that observe changes in the HPC arena and discuss potential problems to be solved in the near future. In this section, we collect these observations and then, along with observations, build towards formulation of challenges for the future in the next section.

Jack Dongarra underlines the progress in HPC hardware that is expected to reach the EFlop/s barrier in 2020-2021 [56]. It can be observed that most of the computational power in today's systems is grouped in accelerators. At the same time, old benchmarks do not fully represent current loads. Furthermore, benchmarks such as HPCG obtain only a small fraction of peak performance of powerful HPC systems today.

HPC can now be accessed relatively easily in a cloud and GPUs and specialized processors like tensor processing units (TPU) addressing artificial intelligence (AI) applications have become the focus with edge computing, i.e., the need for processing near mobile users being important for the future [57].

Energy-aware HPC is one of the current trends that can be observed in both hardware development as well as software solutions, both at the scheduling and application levels [58]. When investigating performance vs energy consumption tradeoffs, it is possible to find nonobvious (i.e., nondefault) configurations using power capping (i.e., other than the default power limit) for both multicore CPUs [59] as well as GPUs [60]. However, optimal configurations can be very different, depending on both the CPU/GPU types as well as application profiles.

A high potential impact of nonvolatile RAM (NVRAM) on high-performance computing has been confirmed in several works. The evaluation in [61] shows its potential support for highly concurrent data-intensive applications that exhibit big memory footprints. Work [62] shows a potential for up to 27% savings in energy consumption. It has also been shown that parallel applications can benefit from NVRAM used as an underlying layer of MPI I/O and serving as a distributed cache memory, specifically for applications such as multiagent large-scale crowd simulations [63], parallel image processing [64], computing powers of matrices [65], or checkpointing [66].

Trends in high-performance computing in terms of software can also be observed by following recent changes to the considered APIs. Table 4 summarizes these changes for the key APIs along with version numbers and references to the literature where these updates are described.

In the network technologies, we can see strong competition in performance factors, where especially the bandwidth is always a hot topic. New hardware and standards for Ethernet speeds beats subsequent barriers: 100, 400, . . . GBps as well as the InfiniBand with its 100, 200, . . . GBps. Thus, this rapid development gives the programmers opportunities to introduce more and more parallel solutions, which are well scalable even for large sizes of the problems. On the other hand, such race does not have a great impact on

Table 1: Target/model classification of technologies.

Technology/ API	Abstraction level/group	Programming model	Programming language	Supported platforms/target parallel system	License/standard
OpenMP	Library	Multithreaded application	C/C++/Fortran	Heterogeneous system with CPU(s), accelerators including GPU(s) [54], supported by, e.g., gcc	OpenMP is a standard [7]
CUDA	Library	CUDA model, computations launched as kernels executed by multiple threads grouped into blocks, global, and shared memory on the GPU as well as host memory for data management	C	Server or workstation with 1 + NVIDIA GPU(s)	Proprietary NVIDIA solution, NVIDIA EULA [8][1]
OpenCL	Library	OpenCL model, computations launched as kernels executed by multiple work items grouped into work groups and memory objects for data management	C/C++	Heterogeneous platform including CPUs, GPUs from various vendors, FPGAs, etc., supported by, e.g., gcc	OpenCL is a standard [9]
Pthreads	Library	Multithreaded application, provides thread management routines, synchronization mechanisms including mutexes, conditional variables	C	Widely available in UNIX platforms, implementations, e.g., NPTL	Part of the POSIX standard
Open ACC	Library	Multithreaded application	C/C++/Fortran	Heterogeneous architectures, e.g., a server or workstation with x86/POWER + NVIDIA GPUs, support for compilers such as PGI, gcc, accULL, etc.	OpenACC is a standard [10]
Java Concurrency	JVM [14] specific	Multithreaded application	Java, scala	Server, workstation, mobile device	Open standards: [12, 13]
TCP/IP	Network stack	Multi-process	C, Fortran, C++, Java, and others	Cluster, server, workstation, mobile device, and others	TCP/IP [15] is a standard broadly implemented by OS developers
RDMA	Network stack	Multiprocess	C	Cluster	RDMA [17] is a standard implemented by over InfiniBand and converged Ethernet protocols
UCX	Network stack	Multiprocess, multithreaded	C, Java, Python	Cluster, server, workstation	UCX [21] is a set of network APIs with a reference implementation
MPI	Library	Multiprocess, also multithreaded if implementation supports	C/Fortran	Cluster, server, workstation	MPI is a standard [26], several implementations available, e.g., OpenMPI and MPICH
OpenSHMEM	Library	Multiprocess application	C, Fortran	Cluster	Open standard with reference implementation
PCJ	Java library	Multiprocess application	Java	Cluster	Open source Java library [29]

TABLE 1: Continued.

Technology/ API	Abstraction level/group	Programming model	Programming language	Supported platforms/target parallel system	License/standard
Apache Hadoop	Set of applications	YARN managed resource negotiation, multiprocess MapReduce tasks [41]	Core functionality in JAVA, also C, BASH, and others	Cluster, server, workstation	Open source implementation of Google's MapReduce [40], Apache software license-ASL 2.0
Apache Spark	Set of applications	Resource negotiation based on the selected resource manager (YARN, Spark Standalone, etc.), executors run workers in threads [49]	Scala	Cluster, server, workstation	Apache software license-ASL 2.0 [55]

[1]https://docs.nvidia.com/cuda/eula/index.html

TABLE 2: Technologies and goals.

Technology/ API	Goals (performance, energy, etc.)	Ease of programming	Ease of assessment, e.g., performance	Ease of deployment/ (auto) tuning	Portability (between hardware, for new hardware, etc.)
OpenMP	Performance, parallelization	Relatively easy, parallelization of a sequential program by addition of directives for parallelization of regions and optionally library calls for thread management, difficulty of implementing certain schemes, e.g., similar to those with Pthread's condition variables [1]	Execution times can be benchmarked easily, debugging relatively easy	Easy, thread number can be set using an environment variable, at the level of region or clause	Available for all major shared memory environments, e.g., in gcc
CUDA	Performance	Proprietary API, easy-to-use in a basic version for a default card, more difficult for optimized codes (requires stream handling, memory optimizations including shared memory–avoiding bank conflicts, global memory coalescing)	Can be performed using cuda-gdb or very powerful nvvp (NVIDIA visual Profiler) or text-based nvprof	Easy, requires CUDA drivers and software	Limited to NVIDIA cards, support for various features depends on hardware version-card's CUDA compute capability and software version
OpenCL	Performance	More difficult than CUDA or OpenMP since it requires much device and kernel management code, optimized code may require specialized kernels which somehow defies the idea of portability	Can be benchmarked at the level of kernels, queue management functions can be used for fencing benchmarked sections	Easy, requires proper drivers in the system	Portable across hybrid parallel systems, especially CPU + GPU
Pthreads	Performance	More difficult than OpenMP, flexibility to implement several multithreaded schemes, involving wait-notify, using condition variables for, e.g., producer-consumer	Easy, thread's code in designated functions, and can be benchmarked there	Easy, thread's code executed in designated functions	Available for all major shared memory environments

TABLE 2: Continued.

Technology/ API	Goals (performance, energy, etc.)	Ease of programming	Ease of assessment, e.g., performance	Ease of deployment/ (auto) tuning	Portability (between hardware, for new hardware, etc.)
OpenACC	Performance	Easy, similar to the OpenMP's directive-based model, however requires awareness of overheads and corresponding needs for optimization related to, e.g., data placement, copy overheads, etc.	Standard libraries can be used for performance assessment, gprof can be used	Requires a compiler supporting OpenACC, e.g., PGI's compiler, GCC, or accULL	Portable across compute devices supported by the software
Java Concurrency	Parallelization	Easy, two levels of abstraction	Easy debugging and profiling	Easy deployment for many OS	Portable over majority of hardware
TCP/IP	Standard network connectivity	Programming can be difficult, requires knowledge of low-level network mechanisms	Debugging can be difficult, available tools for time measurement	Usually already deployed with the OS	Portable over majority of hardware
RDMA	Performance	Programming can be difficult, requires knowledge of low-level network mechanisms	Debugging can be difficult, available tools for time measurement	Deployment can be difficult	Usually used with clusters
UCX	Performance	Programming can be difficult, it is library for frameworks	Debugging can be difficult, it is quite a new solution	Deployment can be difficult	Usually used with clusters
MPI	Performance, parallelization	Relatively easy, high-level, message passing paradigm	Measurement of execution time easy, difficult debugging, especially in a cluster environment	Deployment can require additional tools, e.g., drivers for advanced interconnects such as Infiniband or SLURM for an HPC queue system, tuning typically based on low-level profiling	Portable, implementations available on clusters, servers, workstations, typically used in Unix environments
OpenSHMEM	Performance, parallelization	Easy, needs attention for synchronized data access	No dedicated debugging and profiling tools	Fairly easy deployment in many environments	Portable, implementations available on clusters, servers, workstations, typically used in UNIX environments
PCJ	Performance, parallelization	Easy, classes and annotations used for object distribution	No dedicated debugging and profiling tools	Easy deployment for many OS	Portable over majority of hardware
Apache Hadoop	Performance, large datasets	Relatively easy, high level abstraction, requires good understanding of MapReduce programming model	Easy to acquire job performance overview (web UI and logs), moderately easy debugging, central logging can be used to streamline the process	Moderately easy basic deployment, tweaking performance, and security for entire hadoop ecosystem can be very difficult	Used in clusters, available for Unix and windows
Apache Spark	Performance, low disk, and high RAM usage, large datasets	Relatively easy, high-level abstraction, based on lambda functions on RDD and dataFrames	Easy to acquire job performance overview (web UI and logs), moderately easy debugging, central logging can be used to streamline the process	Easy Spark Standalone deployment, Spark on YARN deployment requires a functioning Hadoop ecosystem	Used in clusters, available for Unix and Windows

the APIs, protocols, and features provided to the programmers, so the legacy software based on the lowest layer services does not need to be updated often.

We can observe that the frameworks and libraries are continuously extended and updated. We can see that some converging tendencies have already been present for a long

time, e.g., an introduction of offload for accelerator support in OpenMP or multithreading support in OpenSHMEM or MPI. The message to the users is that their favorite API will finally support new features of the most popular hardware or at least will give easy way to use it in collaboration with other technologies (e.g., the case of complementing MPI and OpenMP).

Hybrid parallelism has also become mainstream in high-performance computing due to hardware developments and heterogeneity in terms of various compute devices within a node or a cluster (e.g., CPUs + GPUs). This forces programmers to use combinations of APIs for efficient parallel programming, e.g., MPI + CUDA, MPI + OpenCL, or MPI + OpenMP + CUDA [1]. Table 5 summarizes hybridity present in various considered technologies including potential shortcomings as well as disadvantages.

10. Challenges in Modern High-Performance Computing

Similar to discussing trends, we mention selected works discussing expected problems and issues in the HPC arena for the nearest future. We then identify more points for an even more complete picture, in terms of the aspects discussed in Section 2.

Dongarra mentions several problematic issues [56] such as minimization of synchronization and communication of algorithms, using mixed precision arithmetics for better performance (low precision is already used in deep learning [72], for instance), designing algorithms to survive failures, and autotuning of software to a given environment.

According to [73], one of the upcoming challenges in Exascale HPC era will be energy efficiency. Additionally, software issues in HPC are denoted as open issues in this context, e.g., memory overheads and scalability in MPI, thread creation overhead in OpenMP, and copy overheads. Fault tolerance and I/O overheads for large-scale processing are listed as difficulties.

Both the need for autotuning and progress in software for modern HPC systems have also been stated in [74], with an emphasis on the need for looking for better suited languages for HPC than the currently used C/C++ and Fortran.

Finally, apart from the aforementioned challenges and based on our analysis in this paper, we identify the following challenges for the types of parallel processing considered in this work:

(1) Difficulty of offering efficient APIs for hybrid parallel systems includes difficulty of automatic load balancing in hybrid systems. Currently, combinations of APIs with proper optimizations at various parallelization levels are required such as MPI + OpenMP and MPI + OpenMP + CUDA. This stems directly from Figure 1 where there is no single approach/API covering all the configurations in the diagram.

(2) Few programming environments oriented on several criteria apart from performance. Optimizations using performance and, e.g., energy consumption are performed at the level of algorithms or scheduling rather than embedded into programming environments or APIs. This suggests the lack of consideration of energy usage in APIs, especially APIs allowing us to obtain desired performance-energy goals for particular classes of applications and compute devices. This is shown in Table 3. This requires automatic tools for determination of performance vs energy profiles for various applications and compute devices.

(3) Lack of knowledge (of researchers) to integrate various APIs for hybrid systems; many researchers know only single APIs and are not proficient in using all options shown in Table 5.

(4) Need for benchmarking modern and representative applications on various HPC systems with new hardware features, e.g., latest features of new GPU families such as independent thread scheduling in NVIDIA Volta, memory oversubscription in NVIDIA Pascal + series cards, cooperative groups, etc. This stems from very fast developments in the APIs which is shown in Table 4.

(5) Convergence of APIs that target similar environments, e.g., OpenMP and OpenCL. OpenMP now allows offloading to GPUs, accelerators, etc., as shown in Table 3 and some of their applications overlap. This raises a question on whether both will follow in the same direction or diverge more for particular uses.

(6) Lack of automatic determination of application parameters run on complex parallel systems, especially hybrid systems, i.e., numbers of threads and thread affinity on CPUs, grid configurations on GPUs, load balancing among compute devices, etc. Some works [75] have attempted automation of this process but this field of autotuning in such environments, as also shown above, is relatively undeveloped yet.

(7) Difficulty in porting of specialized existing parallel programming environments and libraries to modern HPC systems when one wants to use the architectural benefits of the latest hardware. This is also related to the fast changes in the hardware architectures and APIs following these such as for the latest GPU generations and CUDA versions, but also for other APIs, as shown in Table 4.

(8) Problem of finding best hardware configuration for a given problem and its implementation (CPU/GPU/other accelerators/hybrid), considering relative performance of CPUs, GPUs, interconnects, etc. Certain environments such as MERPSYS [76] allow for simulation of parallel application execution using various hardwares including compute devices such as CPUs and GPUs but the process requires prior calibration on small systems and target applications.

(9) Lack of standardized APIs for new technologies such as NVRAM in parallel computing. This is related to the technology being very new and starting to be used for HPC, as shown in Section 9.

TABLE 3: Technologies and parallelism.

Tech/API	Level of parallelism	Parallelism constructs	Synchronization constructs
OpenMP	Thread teams executing some regions of an application	Directives that define that a certain region is to be executed in parallel, such as #pragma omp parallel, #pragma omp sections, etc.	Several constructs that allow synchronization such as #pragma omp barrier, constructs that denote that a part of code be executed by a certain thread, e.g., #pragma omp master, #pragma omp single, critical section #pragma omp critical, directives for data synchronization, e.g., #pragma omp atomic
CUDA	Threads executing kernels in parallel, threads are organized into a grid of blocks each of which consists a number of threads, both threads in a block and blocks in a grid can be organized in 1D, 2D, or 3D logical structures, kernel execution, host to device and device to host copying can be overlapped if issued into various CUDA streams	Invocation of a kernel function launches parallel computations by a grid of threads, possible execution on several GPUs in parallel	Execution of all grid's threads is synchronized after the kernel has completed; on the host side, execution of individual threads in a block is possible with a call to __syncthreads (), atomic functions available for accessing global memory
OpenCL	Work items executing kernels in parallel, work items are organized into an NDRange of work groups each of which consists a number of work items, both work items in a work group and work groups in an NDRange can be organized in 1D, 2D, or 3D logical structures, kernel execution, host to device and device to host copying can be overlapped if issued into various command queues	Invocation of a kernel function launches parallel computations by an NDRange of work items, OpenCL allows parallel execution of kernels on various compute devices such as CPUs and GPUs	Execution of all NDRange's work items is synchronized after the kernel has completed; on the host side, execution of individual work items in a block is possible with a call to barrier with indication whether a local or global memory variable that should be synchronized, synchronization using events is also possible, atomic operations available for synchronization of references to global or local memory
Pthreads	Threads are launched explicitly for execution of a particular function	a call to pthread_create () creates a thread for execution of a specific function for which a pointer is passed as a parameter	Threads can be synchronized by the thread that called pthread_create () by calling pthread_join (), there are mechanisms for synchronization of threads such as mutexes, condition variables with wait pthread_cond_wait () and notify routines, e.g., pthread _cond_signal (), barrier pthread_barrier*(), implicit memory view synchronization among threads upon invocation of selected functions
OpenACC	Three levels of parallelism available: execution of gangs, one or more workers within a gang, vector lanes within a worker	Parallel execution of code within a block marked with #pragma acc parallel, parallel execution of a loop can be specified with #pragma acc loop	For #pragma acc parallel, an implicit barrier is present at the end of the following block, if async is not present, atomic accesses possible with #pragma acc atomic according to documentation [10], the user should not attempt to implement barrier synchronization, critical sections or locks across any of gang, worker, or vector parallelism
Java Concurrency	Thread inside the same JVM	The main tread created during the JVM start in main () method is a root of other threads created dynamically using explicit, e.g., new thread (), or implicit constructs, e.g., thread pool	Typical shared memory mechanisms like synchronized sections or guarded blocks
TCP/IP	The whole network nodes	Managed manually by adding and configuring hardware	Using IP addresses and ports for distinguishing the connections/ destinations, no specific constructs

TABLE 3: Continued.

Tech/API	Level of parallelism	Parallelism constructs	Synchronization constructs
RDMA	The whole network nodes	Managed manually by adding and configuring hardware	Using remote access with the indicators of the accessed memory
UCX	The whole network nodes	Managed manually by adding and configuring hardware	Special APIs for message passing and memory access
MPI	Processes (+threads combined with a multithreaded API like OpenMP, Pthreads if MPI implementation supports required thread support level)	Processes created with mpirun at application launch + potentially processes created dynamically with a call to MPI_Comm_spawn or MPI_Comm_spawn _multiple	MPI collective routines: barrier, communication calls like MPI_Gather, MPI_Scatter, etc.
OpenSHMEM	Processes possibly on different compute nodes	Processes created with oshrun at application launch	OpenSHMEM synchronization and collective routines; barrier, broadcast, reduction, etc.
PCJ	The so-called nodes placed in possibly separated JVMs on different compute nodes	The node structure is created by a main Manager node at application launch	PCJ synchronization and collective routines; barrier, broadcast, etc.
Apache Hadoop	Task is a single process running inside a JVM	API to formulate MapReduce functions	Synchronization managed by YARN, API for data aggregation (reduce operation)
Apache Spark	Executors run worker threads	RDD and DataFrame API for managing distributed computations	Managed by built-in Spark Standalone or by external cluster manager: YARN, Mesos etc.

TABLE 4: Selected, important latest features and extensions in various technologies.

Tech/API	Description of latest features	Version	Literature
OpenMP	Support for controlling offloading behavior (it is possible to offload to GPUs as well), extensions regarding thread affinity information management (affinity added to the task construct), data mapping clarifications and extensions, extended support for various C++ and Fortran versions	5.0	[7]
CUDA	Improved the scalability of cudaFree in multi-GPU environments, support for cooperative group kernels with MPS, new cuBLASLt library has been added for general matrix GEMM operations, cuBLASLt now has support for FP16 matrix multiplies using tensor cores on volta and turing GPUs, improved performance of cuFFT on multi-GPU systems, some random generators in cuRAND	10.1	[8]
OpenCL	Minor changes in the latest 2.1 to 2.2 update, e.g., added calls to clSetProgramSpecializationConstant and clSetProgramReleaseCallback, major changes in 1.2 to 2.0 update including shared virtual memory, device queues used to enqueue kernels on a device, added the possibility for kernels enqueing kernels using a device queue	2.2	[9]
OpenACC	Reduction clause on in a compute construct assumes a copy for each reduction variable, arrays and composite variables are allowed in reduction clauses, local device defined	2.7	[10]
Java Conc.	An interoperable publish-subscribe framework with flow class and various other improvements	9	[67]
MPI	Introduction of nonblocking collective I/O routines, corrections in Fortran bindings	3.1	[26]
OpenSHMEM	Multithreading support, extended type support, C11 type-generic interfaces for point-to-point synchronization, additional functions and extensions to the existing ones	1.4	[68]
Apache Hadoop	Support for opportunistic containers, i.e., containers that are scheduled even if there is not enough resources to run them. Opportunistic containers wait for resource availability and since they have low priority, they are preempted if higher priority jobs are scheduled	3.0.3	[69]
Apache Spark	Built-in avro datasource, support for eager evaluation of DataFrames	2.4	[70]

11. Comparisons of Existing Parallel Programming Approaches for Practical Applications

In order to extend our systematic review of the approaches and APIs, in this section, we provide summary of selected existing comparisons of at least some subsets of approaches considered in this work for practical applications. This can be seen as a review that allows us to gather insights which APIs could be preferred in particular compute intensive applications.

In [77], ten benchmarks are used to compare CUDA and OpenACC performance. The authors measure execution times and speed of GPU data transfer for 19 kernels with different optimizations. Test results indicate that CUDA is slightly faster than OpenACC but requires more time to send

TABLE 5: Hybridity in various technologies.

Tech/API	Support for hybridity (description)	Potential disadvantages or shortcomings
OpenMP	Allows to run threads on multicore/many-core CPUs as well as offload and parallelize within devices, including GPUs	Not easy to set up for offloading to GPUs
CUDA	CUDA's API allows management of several GPUs, it is possible to manage computations on several GPUs from a single CPU thread, several streams may be used for sequences of commands onto one or more GPUs	Requires combination with some multithreaded APIs such as OpenMP or Pthreads for load balancing across CPU + GPU systems, with MPI for clusters, many host threads may be preferred for balancing among several GPUs
OpenCL	A universal model based on kernels for execution on several, potentially different, compute devices, command queues used for several streams of computations	Requires many more lines of code when used with hybrid CPU + GPU systems compared to, e.g., OpenMP + CUDA
OpenACC	Allows to manage computations across several devices within a node	While it is possible to balance computations among devices using OpenACC functions (similarly to CUDA), CPU threads (and correspondingly APIs allowing that) might be preferred for more efficient balancing strategies [71]
MPI	The standard allows a hybrid multiprocess + multithreaded model if implementation supports it (check with MPI_init_thread ()). An MPI implementation can be combined with multithreaded APIs such as OpenMP or Pthreads, a CUDA-aware MPI implementation allows using device pointers in MPI calls	Requires combining with APIs such as OpenCL or, e.g., OpenMP/CUDA to use efficiently with hybrid multicore/many-core CPUs and GPUs, such solutions are not always fully supported by every MPI implementations, e.g., CUDA features can be limited to some type of the operations, e.g., point-to-point
Apache Hadoop	Ability to manage computations with different processing paradigms: MapReduce, Spark, HiveQL, Tez, etc.	Easy basic installation but requires a lot of effort to provide production ready and secure cluster
Apache Spark	Barrier execution mode makes integration with machine learning pipelines much easier	Production ready solutions typically require external cluster manager

data to and from a GPU. Since both APIs are performed similarly, the authors suggest using multiplatform Open-ACC, especially because it provides an easier to use syntax.

The EasyWave [78] system receives data from seismic sensors and is used to predict characteristic (wave height, water fluxes etc.) of a tsunami. To improve processing speed, CUDA and OpenACC EasyWave implementations were compared, each tested on two differently configured machines with NVIDIA Tesla and Fermi GPU, respectively. CUDA single instruction multiple dispatch (SIMD) optimizations for grid point updates (computing value for element of the grid) achieved 2.15 and 10.77 for the aforementioned GPU. Parallel window extension with atomic instruction synchronization allowed for 13% and 46% speed up.

Cardiac electrophysiological simulations allow study of patient's heart behavior. Those simulations provide computationally heavy challenges since the nonlinear model requires numerical solutions of differential equations. In [79], the authors provide implementation of system solving partial and ordinary differential equations with discretization for high spatial resolutions. GPGPU solutions using CUDA, OpenACC, and OpenGL are compared to test the performance. Ordinary differential equations were best solved with OpenGL which achieved a speedup of 446 while parabolic partial equations where best solved using CUDA with a speedup of 8.

SYCL is a cross-platform solution that provides functionality similar to OpenCL and allows building parallel application for heterogeneous hardware. It uses standard C++, and its programming model allows providing kernel and host code in one file ("single-source programming"). In [80], the authors compare overall performance (number of API calls, memory usage, processing time) and easy of use of SYCL with OpenMP and OpenCL. Two benchmarks are provided: Common Midpoint (CMP) used in seismic processing and 27stencil which is one of the OpenACC benchmarks and is similar to algorithms for solving partial differential equations. The authors also compare results with previously published benchmarking results. Generally, results indicate that non-SYCL implementations are about two times faster (2.35 and 2.77 for OpenCL, 1.38 and 2.22 for OpenMP) than SYCL implementation. The authors point out that differences in processing time may be influenced by small differences in used hardware and compiler used. Comparisons with previous tests indicate that SYSCL is catching up with other programming models in context of performance.

In paper [81], the authors presented a comparison of the OpenACC and OpenCL related to the ease of the tunability. They distinguished four typical steps of the tuning process: (i) avoiding redundant host-device data transfer, (ii) data padding for 32, 64, 128 bytes segments read-write matching, (iii) kernel execution parameter tuning, and (iv) use of on-chip instead of global memory where possible. Furthermore, the additional barrier operation was proposed for OpenACC to introduce the possibility of explicit thread synchronization. Finally, the authors performed evaluation, using a nanopowder growth simulator as a benchmark, and implemented each optimization step. The results showed

similar speedups for both OpenCL and OpenACC implementations; however, the OpenACC one required fewer modifications for the two first optimization steps.

An interesting evaluation of OpenMP regarding its new features (ver. 4.5/5.0) was presented in [82]. The authors tested four different implementations of miniMD (a molecular dynamics benchmark from the Mantevo benchmark suite [83]): (i) orig: original, (ii) mxhMD: optimized for Intel Xeon architecture, (iii) Kokkos: based on Kokkos portability framework [84], and (iv) omp5: utilizing OpenMP 4.5 offload features. For the performance-portability assessment of each implementation, a self-developed Φ metric was used and the results showed the advantage of Kokkos for GPU and mxhMD for CPU hardware; however, for the productivity measured in SLOC, omp5 was on-par with Kokkos. The conclusion was that introduction of new features in OpenMP provides improvements for the programming process, but the portability frameworks (like Kokkos) are still viable approaches.

The paper [85] provides a survey of approaches and APIs supporting parallel programming for multicore and many-core high-performance systems, albeit already 7 years old. Specifically, the authors classified parallel processing models as pure (Pthreads, OpenMP, message passing), heterogeneous parallel programming models (CUDA, OpenCL, DirectCompute, etc.), Partitioned Global Address Space and hybrid programming (e.g., Pthreads + MPI, OpenMP + MPI, CUDA + Pthreads and CUDA + OpenMP, CUDA + MPI). The work presents support for parallelism within Java, HPF, Cilk, Erlang, etc., as well as summarizes distributed computing approaches such as grids, CORBA, DCOM, Web Services, etc.

Thouti and Sathe [86] present a comparison of OpenMP and OpenCL, also 7 years old already. The authors developed four benchmarking algorithms (matrix multiplication, N-Queens problem, image convolution, and string reversal) and describe achieved speedup. In general, OpenCL performed better when input data size increased. OpenMP performed better in the image convolution problem (speedup of 10) while (due to overhead work of kernel creation) OpenCL provided no improvement. The best speedup was achieved in the matrix multiplication solution (8 for OpenMP and 598 for OpenCL).

In [87], Memeti et al. explore performance of OpenMP, OpenCL, OpenACC, and CUDA. Programming productivity is measured subjectively (number of lines of code needed to achieve parallelization) while energy usage and processing speed are tested objectively. The authors used SPEC Accel suite and Rodinia for benchmarking aforementioned technologies in heterogeneous environments (two single-node configurations with 48 and 244 threads). In context of programming productivity, OpenCL was judged to be the least effective since it requires more effort than OpenACC (6.7x more) and CUDA (2x more effort). OpenMP requires less effort than CUDA (3.6x) and OpenCL (3.1x). CUDA and OpenCL had similar, application dependent, energy efficiency. In the context of processing speed, CUDA and OpenCL performed better than OpenMP and OpenCL was found to be faster than OpenACC.

Heat conduction problem solution, a mini-app called TeaLeaf, is used to showcase [88] code portability and compare performance of moderately new frameworks: Kokkos and RAJA with OpenACC, OpenMP 4.0, CUDA, and OpenCL. In general, RAJA and Kokkos provide satisfactory performance. Kokos was only 10% and 5% slower than OpenMP and CUDA while RAJA was found to be 20% slower than OpenMP. Results for OpenCL varied and did not allow for reliable comparison. Device tailored solutions were found performing better than platform-independent code. Nevertheless, Kokkos and RAJA provide rich lambda expressions, good performance and easy portability which means that if they reach maturity, they can become valuable frameworks.

In [89], Kang et al. presented a practical comparison between the shared memory (OpenMP), message-passing (MPI–MPICH), and MapReduce (Apache Hadoop) approaches. They selected two fairly simple problems (the all-pairs-shortest path in a graph, as a computational-intensive benchmark and two sources-data join as a data-intensive one). The results showed the advantage of the shared memory for computations and MapReduce for data-intensive processing. We can note that the experiments were performed only for two problems and only using one hardware setup (a set of workstations connected by 1 Gbps Ethernet).

Another MapReduce vs message-passing/shared memory comparison was presented in [90] showing that even for a typical big data problem (counting words in a text, with roughly 2 GB of data), the in-memory implementation can be much faster than a big-data solution. The experiments were executed in a typical cloud environment (Amazon AWS) using Apache Spark (which is usually faster than a typical Hadoop framework) in comparison with MPI/OpenMP implementation. The Spark results were an order of magnitude slower than OpenMP/MPI ones.

Asaadi et al. in [91] presented yet another MapReduce/message-passing/shared memory comparison using the following frameworks: Apache Hadoop and Apache Spark, with two versions: IP-over-Infiniband and RDMA directly (for shuffling only), OpenMPI with RDMA support, and OpenMP, using an unified hardware platform based on a typical HPC cluster with an InfiniBand interconnect. The following benchmarks were executed: sum reduction of vector of numbers (a computation performance micro-benchmark), parallel file reading from local file system (an I/O performance micro-benchmark), calculating average answer count for available questions using data from StackExchange website, and executing PageRank algorithm over a graph with 1,000,000 vertices. The discussion covered several quality factors: maintainability (where OpenMP was the leader), support for execution control flow (where MPI has the most fine-grained access), performance and scalability (where MPI showed the best results even for I/O-intensive processing), and fault tolerance (where Spark seems to be the best choice, however containing one single point of failure—a driver component).

In [92], Lu et al. proposed extension to a typical MPI implementation to provide Big Data related functionality: DataMPI. They proposed four supported processing modes:

Common, MapReduce, Iteration, and Streaming, corresponding to the typical data processing models. The proposed system was implemented in Java and provided an appropriate task scheduler, supporting data-computation locality and fault tolerance. The comparison to Apache Hadoop showed an advantage of the proposed solution in efficiency (31%–41% better performance), fault tolerance (21% improvement), and flexibility (more processing modes), as well as similar results in scalability (linear in both cases) and productivity (comparable coding complexity).

The evaluation of Apache Spark versus OpenMPI/OpenMP was presented in [93]. The authors performed tests using two machine learning algorithms: *K*-Nearest Neighbors (KNN) and Pegasus Support Vector Machines (SVM), for data related to physical particles' experiments (HIGGS Data Set [94]) with the size 11 of 28-dimension records, i.e., about 7 GB of disk space, thus they fit in the memory of a single compute node. The benchmarks were executed using a typical cloud environment (Google Cloud Platform), with different numbers of compute nodes and algorithm parameters. For this setup, with such a small data size, the performance results, i.e., execution times, showed that OpenMPI/OpenMP outperformed Spark by more than one order of magnitude; however, the authors clearly marked distinction in possible fault-tolerance and other aspects which are additionally supported by Spark.

The paper [95] provides performance comparison of OpenACC and CUDA languages used for programming an NVIDIA accelerator (Tesla K40c). The authors tried to evaluate data size sensitivity of both solutions, namely, their methodology uses Performance Ratio of Data Sensitivity (PRoDS) to check how the change of data size influences the performance of a given algorithm. The tests covering 10 benchmarks with 19 different kernels showed the advantage of CUDA in the case of optimized code; however, for implementation without the optimization, OpenACC is less sensitive to data changes. The overall conclusion was that OpenACC seems to be a good approach for nonexperienced developers.

12. Conclusions and Future Work

In this paper, we presented detailed analysis of state-of-the-art methodologies and solutions supporting development of parallel applications for modern high-performance computing systems. We distinguished shared vs distributed memory systems, one-sided or two-sided communication and synchronization APIs, and various programming abstraction levels. We discussed solutions using multithreaded programming, message passing, Partitioned Global Address Space, agent-based parallel processing, and MapReduce processing. For APIs, we presented, among others, supported programming languages, target environments, ease of programming, debugging and deployment, latest features, constructs allowing parallelism as well as synchronization, and hybrid processing. We identified current trends and challenges in parallel programming for HPC. Awareness of these can help standard committees shape new versions of parallel programming APIs.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

References

[1] P. Czarnul, *Parallel Programming for Modern High Performance Computing Systems*, Chapman and Hall/CRC Press, Boca Raton, FL, USA, 2018.
[2] C++ v.11 thread support library, 2019.
[3] Intel threading building blocks, 2019.
[4] High Performance paralleX (HPX), 2019.
[5] J. Nonaka, M. Matsuda, T. Shimizu et al., "A study on open source software for large-scale data visualization on sparc64fx based hpc systems," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, pp. 278–288, ACM, Chiyoda, Tokyo, Japan, January 2018.
[6] M. U. Ashraf and F. E. Eassa, "Opengl based testing tool architecture for exascale computing," *International Journal of Computer Science and Security (IJCSS)*, vol. 9, no. 5, p. 238, 2015.
[7] OpenMP Architecture Review Board, *OpenMP Application Programming Interface*, 2018.
[8] NVIDIA: CUDA toolkit documentation v10.1.243, 2019.
[9] Khronos OpenCL Working Group, "The openCL specification," 2019.
[10] OpenACC-Standard.org, *The OpenACC Application Programming Interface*, 2018.
[11] S. Wienke, C. Terboven, J. C. Beyer, and M. S. Müller, "A pattern-based comparison of openacc and openmp for accelerator computing," in *European Conference on Parallel Processing*, pp. 812–823, Springer, Berlin, Germany, 2014.
[12] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, and D. Smith, "The Java language specification," 2019.
[13] M. Odersky, P. Altherr, V. Cremet et al., "Scala language specification," 2019.
[14] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, and D. Smith, "The Java virtual machine specification," 2019.
[15] TCP/IP standard, 2019.
[16] A. L. Russell, "The internet that wasn't," *IEEE Spectrum*, vol. 50, no. 8, pp. 39–43, 2013.
[17] RDMA consortium, 2019.
[18] InfiniBand architecture specification release 1.2.1 Annex A16: RoCE, 2010.
[19] M. Beck and M. Kagan, "Performance evaluation of the RDMA over ethernet (RoCE) standard in enterprise data centers infrastructure," in *Proceedings of the 3rd Workshop on Data Center–Converged and Virtual Ethernet Switching*, Berkeley, CA, USA, September 2011.
[20] InfiniBand architecture specification release 1.2.1 Annex A17: RoCEv2, 2010.
[21] P. Shamis, M. G. Venkata, M. G. Lopez et al., "UCX: an open source framework for HPC network APIs and beyond," in *Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pp. 40–43, IEEE, Santa Clara, CA, USA, August 2015.
[22] B. Chapman, T. Curtis, S. Pophale et al., "Introducing openshmem: shmem for the pgas community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model, PGAS '10*, pp. 2.1–2.3, ACM, New York, NY, USA, October 2010.
[23] M. Baker, F. Aderholdt, M. G. Venkata, and P. Shamis, "OpenSHMEM-UCX: evaluation of UCX for implementing

OpenSHMEM programming model," in *OpenSHMEM and Related Technologies. Enhancing OpenSHMEM for Hybrid Environments*, pp. 114–130, Springer International Publishing, Berlin, Germany, 2016.

[24] N. Papadopoulou, L. Oden, and P. Balaji, "A performance study of ucx over infiniband," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid '17*, pp. 345–354, IEEE Press, Piscataway, NJ, USA, May 2017.

[25] R. Love, *Linux System Programming: Talking Directly to the Kernel and C Library*, O'Reilly Media, Inc., Newton. MA, USA, 2007.

[26] Message passing interface forum MPI: a message-passing interface standard, 2015.

[27] MPICH–a portable implementation of MPI, 2019.

[28] The Open MPI Project, "Open Mpi: open source high performance computing. A high performance message passing library," 2019.

[29] M. Nowicki and P. Bala, "Parallel computations in Java with PCJ library," in *Proceedings of the 2012 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 381–387, IEEE, Madrid, Spain, July 2012.

[30] The chapel parallel programming language, 2019.

[31] The X10 parallel programming language, 2019.

[32] Berkeley UPC—unified parallel C, 2019.

[33] A. A. Buss and H. Papadopoulos, "STAPL: standard template adaptive parallel library," *SYSTOR '10*, vol. 10, 2010.

[34] M. Kisiel-Dorohinicki, G. Dobrowolski, and E. Nawarecki, "Agent populations as computational intelligence," in *Neural Networks and Soft Computing*, L. Rutkowski and J. Kacprzyk, Eds., pp. 608–613, Physica-Verlag HD, Heidelberg, Germany, 2003.

[35] M. Kisiel-Dorohinicki, "Agent-oriented model of simulated evolution," 2002.

[36] M. J. North, T. R. Howe, N. T. Collier, and J. R. Vos, "The repast simphony runtime system," in *Proceedings of the Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms*, vol. 10, pp. 13–15, Citeseer, Chicago, IL, USA, October 2005.

[37] N. Collier, "Repast: an extensible framework for agent simulation," *The University of Chicago's Social Science Research*, vol. 36, p. 2003, 2003.

[38] N. Collier and M. North, "Repast hpc: a platform for large-scale agent-based modeling," *Large-Scale Computing*, vol. 10, pp. 81–109, 2012.

[39] S. Cincotti, M. Raberto, and A. Teglio, "Credit money and macroeconomic instability in the agent-based model and simulator eurace. Economics: the open-access," *Open-Assessment E-Journal*, vol. 4, 2010.

[40] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pp. 137–150, San Francisco, CA, USA, December 2004.

[41] V. Kumar Vavilapalli, A. Murthy, C. Douglas et al., "Apache hadoop yarn: yet another resource negotiator," 2013.

[42] T. White, *Hadoop: The Definitive Guide*, O'Reilly Media, Inc., Newton, MA, USA, 4th edition, 2015.

[43] J. Leverich and C. Kozyrakis, "On the energy (in)efficiency of hadoop clusters," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 1, pp. 61–65, 2010.

[44] A. Aji, F. Wang, H. Vo et al., "Hadoop-gis: a high performance spatial data warehousing system over mapreduce," *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, vol. 6, 2013.

[45] J. Alwidian and A. A. A. AlAhmad, "Hadoop mapreduce job scheduling algorithms survey and use cases," *Modern Applied Science*, vol. 13, 2019.

[46] K. Kc and K. Anyanwu, "Scheduling hadoop jobs to meet deadlines," in *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science, CLOUDCOM '10*, pp. 388–392, IEEE Computer Society, Washington, DC, USA, November 2010.

[47] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: fair allocation of multiple resource types," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, NSDI'11*, pp. 323–336, USENIX Association, Berkeley, CA, USA, June 2011.

[48] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08*, pp. 29–42, USENIX Association, Berkeley, CA, USA, December 2008.

[49] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10*, USENIX Association, Berkeley, CA, USA, June 2010.

[50] M. Zaharia, M. Chowdhury, T. Das et al., "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12*, USENIX Association, Berkeley, CA, USA, April 2012.

[51] L. Xu, R. Butt, A., S. H. Lim, and R. Kannan, "A heterogeneity-aware task scheduler for spark," 2018.

[52] V. S. Marco, B. Taylor, B. Porter, and Z. Wang, "Improving spark application throughput via memory aware task co-location: a mixture of experts approach," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, Middleware '17*, pp. 95–108, ACM, New York, NY, USA, December 2017.

[53] P. Zhang and Z. Guo, "An improved speculative strategy for heterogeneous spark cluster," 2018.

[54] S. McIntosh-Smith, M. Martineau, A. Poenaru, and P. Atkinson, *Programming Your Gpu with Openmp*, University of Bristol, Bristol, UK, 2018.

[55] H. Karau and R. Warren, *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*, O'Reilly Media, Inc., Sebastopol, CA, USA, 1st edition, 2017.

[56] J. Dongarra, "Current trends in high performance computing and challenges for the future," 2017.

[57] B. Trevino, "Five trends to watch in high performance computing," 2018.

[58] P. Czarnul, J. Proficz, and A. Krzywaniak, "Energy-aware high-performance computing: survey of state-of-the-art tools, techniques, and environments," *Scientific Programming*, vol. 2019, Article ID 8348791, 19 pages, 2019.

[59] A. Krzywaniak, J. Proficz, and P. Czarnul, "Analyzing energy/performance trade-offs with power capping for parallel applications on modern multi and many core processors," in *Proceedings of the 2018 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pp. 339–346, Poznań, Poland, September 2018.

[60] A. Krzywaniak and P. Czarnul, "Performance/energy aware optimization of parallel applications on gpus under power capping," *Parallel Processing and Applied Mathematics*, 2019.

[61] B. Van Essen, R. Pearce, S. Ames, and M. Gokhale, "On the role of nvram in data-intensive architectures: an evaluation," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 703–714, Shanghai, China, May 2012.

[62] D. Li, J. S. Vetter, G. Marin et al., "Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 945–956, Kolkata, India, May 2012.

[63] A. Malinowski and P. Czarnul, "Multi-agent large-scale parallel crowd simulation with nvram-based distributed cache," *Journal of Computational Science*, vol. 33, pp. 83–94, 2019.

[64] A. Malinowski and P. Czarnul, "A solution to image processing with parallel MPI I/O and distributed NVRAM cache," *Scalable Computing: Practice and Experience*, vol. 19, no. 1, pp. 1–14, 2018.

[65] A. Malinowski and P. Czarnul, "Distributed NVRAM cache–optimization and evaluation with power of adjacency matrix," in *Computer Information Systems and Industrial Management–16th IFIP TC8 International Conference, CISIM 2017, volume of 10244 of Lecture Notes in Computer Science*, K. Saeed, W. Homenda, and R. Chaki, Eds., pp. 15–26, Bialystok, Poland, 2017.

[66] P. Dorożyński, P. Czarnul, A. Malinowski et al., "Checkpointing of parallel mpi applications using mpi one-sided api with support for byte-addressable non-volatile ram," *Procedia Computer Science*, vol. 80, pp. 30–40, 2016.

[67] D. Lea, "JEP 266: more concurrency updates," 2019.

[68] Baker M. B., Boehm S., Bouteiller A., et al., Openshmem specification 1.4, 2017.

[69] K. Karanasos, A. Suresh, and C. Douglas, *Advancements in YARN Resource Manager*, Springer International Publishing, Berlin, Germany, 2018.

[70] J. Laskowski, "The internals of apache spark barrier execution mode," 2019.

[71] OpenACC-Standard.org, *OpenACC Programming and Best Practices Guide*, 2015.

[72] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning, ICML'15*, vol. 37, pp. 1737–1746, Lille, France, July 2015.

[73] C. A. Emerson, "Hpc architectures–past, present and emerging trends," 2017.

[74] M. B. Giles and I. Reguly, "Trends in high-performance computing for engineering calculations," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 372, 2014.

[75] P. Czarnul and P. Rościszewski, "Expert knowledge-based auto-tuning methodology for configuration and application parameters of hybrid cpu + gpu parallel systems," in *Proceedings of the 2019 International Conference on High Performance Computing & Simulation (HPCS 2019)*, Dublin, Ireland, July 2019.

[76] P. Czarnul, J. Kuchta, M. Matuszek et al., "MERPSYS: an environment for simulation of parallel application execution on large scale HPC systems," *Simulation Modelling Practice and Theory*, vol. 77, pp. 124–140, 2017.

[77] X. Li and P. C. Shih, "Performance comparison of cuda and openacc based on optimizations," in *Proceedings of the 2018 2Nd High Performance Computing and Cluster Technologies Conference, HPCCT 2018*, pp. 53–57, ACM, New York, NY, USA, June 2018.

[78] S. Christgau, J. Spazier, B. Schnor, M. Hammitzsch, A. Babeyko, and J. Waechter, "A comparison of cuda and openacc: accelerating the tsunami simulation easywave," in *Proceedings of the 2014 Workshop Proceedings on Architecture of Computing Systems (ARCS)*, pp. 1–5, Luebeck, Germany, February 2014.

[79] R. Sachetto Oliveira, B. M. Rocha, R. M. Amorim et al., "Comparing cuda, opencl and opengl implementations of the cardiac monodomain equations," in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waśniewski, Eds., pp. 111–120, Springer Berlin Heidelberg, Berlin, Germany, 2012.

[80] H. C. D. Silva, F. Pisani, and E. Borin, "A comparative study of sycl, opencl, and openmp," in *Proceedings of the 2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pp. 61–66, New York, NY, USA, December 2016.

[81] M. Sugawara, S. Hirasawa, K. Komatsu, H. Takizawa, and H. Kobayashi, "A comparison of performance tunabilities between opencl and openacc," in *Proceedings of the 2013 IEEE 7th International Symposium on Embedded Multicore Socs*, pp. 147–152, Tokyo, Japan, September 2013.

[82] S. J. Pennycook, J. D. Sewall, and J. R. Hammond, "Evaluating the impact of proposed openmp 5.0 features on performance, portability and productivity," in *Proceedings of the 2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pp. 37–46, Dallas, TX, USA, November 2018.

[83] M. A. Heroux, D. W. Doerfler, P. S. Crozier et al., "Improving performance via mini-applications. Sandia national laboratories," Technical Report SAND2009-5574 3, Sandia National Laboratories, Livemore, CA, USA, 2009.

[84] H. C. Edwards, C. R. Trott, D. Sunderland, and Kokkos, "Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[85] J. Diaz, C. Munoz-Caro, and A. Nino, "A survey of parallel programming models and tools in the multi and many-core era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 8, pp. 1369–1386, 2012.

[86] K. Thouti and S. R. Sathe, "Comparison of openmp & opencl parallel processing technologies," *International Journal of Advanced Computer Science and Applications*, vol. 3, no. 4, 2012.

[87] S. Memeti, L. Li, S. Pllana, J. Kolodziej, and C. Kessler, "Benchmarking opencl, openacc, openmp, and cuda: programming productivity, performance, and energy consumption," in *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing, ARMS-CC '17*, pp. 1–6, ACM, New York, NY, USA, July 2017.

[88] M. Martineau, S. McIntosh-Smith, M. Boulton, and W. Gaudin, "An evaluation of emerging many-core parallel programming models," in *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM'16*, pp. 1–10, ACM, New York, NY, USA, May 2016.

[89] S. J. Kang, S. Y. Lee, and K. M. Lee, "Performance comparison of openmp, mpi, and mapreduce in practical problems," *Advances in Multimedia*, vol. 2015, 2015.

[90] J. Li, "Comparing spark vs mpi/openmp on word count mapreduce," 2018.

[91] H. Asaadi, D. Khaldi, and B. Chapman, "A comparative survey of the hpc and big data paradigms: analysis and experiments," in *Proceedings of the 2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 423–432, Taipei, Taiwan, September 2016.

[92] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu, "Datampi: extending mpi to hadoop-like big data computing," in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 829–838, Minneapolis, MN, USA, May 2014.

[93] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita, "Big data analytics in the cloud: spark on hadoop vs mpi/openmp on beowulf," *Procedia Computer Science*, vol. 53, pp. 121–130, 2015.

[94] Whiteson, D.: HIGGS data set, 2019.

[95] X. Li and P. C. Shih, "An early performance comparison of cuda and openacc," in *Proceedings of the MATEC Web of Conferences, ICMIE*, vol. 208, Lille, France, July 2018.