Scientific Programming

Methodologies for Highly Scalable and Parallel Scientific Programming on High Performance Computing Platforms

Lead Guest Editor: Pedro Valero-Lara
Guest Editors: Antonio J. Peña and Vassil Alexandrov

Methodologies for Highly Scalable and Parallel Scientific Programming on High Performance Computing Platforms

Methodologies for Highly Scalable and Parallel Scientific Programming on High Performance Computing Platforms

Lead Guest Editor: Pedro Valero-Lara
Guest Editors: Antonio J. Peña and Vassil Alexandrov

Contents

Survey of Methodologies, Approaches, and Challenges in Parallel Programming Using High-Performance Computing Systems
Paweł Czarnul, Jerzy Proficz, and Krzysztof Drypczewski
Review Article (19 pages), Article ID 4176794, Volume 2020 (2020)

Analysis of a New MPI Process Distribution for the Weather Research and Forecasting (WRF) Model
R. Moreno, E. Arias, D. Cazorla, J. J. Pardo, A. Navarro, T. Rojo, and F. J. Tapiador
Research Article (13 pages), Article ID 8148373, Volume 2020 (2020)

Hindawi
Scientific Programming
Volume 2020, Article ID 4176794, 19 pages
https://doi.org/10.1155/2020/4176794

Review Article

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

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

Received 11 October 2019; Accepted 30 December 2019; Published 29 January 2020

Guest Editor: Pedro Valero-Lara

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.

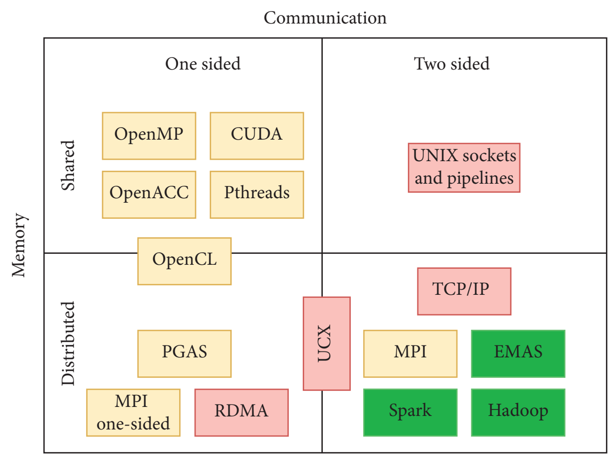

Figure 1: Abstraction level marked with colors: high, green; middle, yellow; low, red.

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

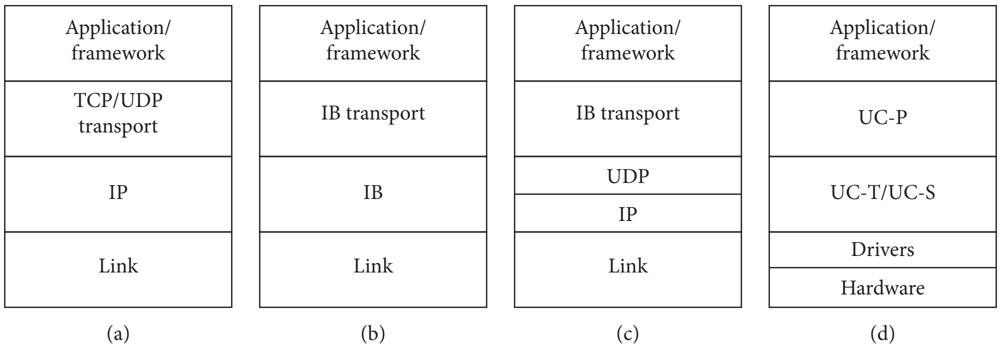

Figure 2: Main network stacks used in HPC: (a) TCP/IP, (b) RDMA over Infiniband/RoCEv1, (c) RDMA over RoCEv2, and (d) UCX.

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 OpenSHMEM [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]. EURACE 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 OpenACC, 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 manycore 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.

Hindawi
Scientific Programming
Volume 2020, Article ID 8148373, 13 pages
https://doi.org/10.1155/2020/8148373

Research Article

Analysis of a New MPI Process Distribution for the Weather Research and Forecasting (WRF) Model

R. Moreno, E. Arias, D. Cazorla, J. J. Pardo, A. Navarro, T. Rojo, and F. J. Tapiador

University of Castilla-La Mancha, Albacete, Spain

Correspondence should be addressed to J. J. Pardo; juanjose.pardo@uclm.es

Received 24 September 2019; Revised 28 November 2019; Accepted 14 December 2019; Published 17 January 2020

Academic Editor: Antonio J. Peña

The standard method used in the Weather Research and Forecasting (WRF) model for distributing MPI processes across the processors is not always optimal. This circumstance affects performance, i.e., execution times, but also energy consumption, especially if the application is to be extended to exascale. The authors found that the reason why the standard method for process distribution is not always optimal was an imbalance between the orthogonality of the communication and the proper cache usage, and this affects energy consumption. We present an improved MPI process distribution algorithm that increases the performance. Furthermore, scalability analyses for the new algorithm are presented and the energy use of the system is evaluated. A solution for balancing energy use with performance is also proposed for cases where the former is a concern.

1. Introduction

Weather forecasting is becoming increasingly important in people's everyday lives. It is as important for a person who wants to have a good weekend as it is for an agency that has to plan a world-class event like the Winter Olympics. Moreover, increasingly higher resolution forecasts are being demanded, which involves the need to increase the computational power used for these forecasts, even reaching exascale. From an operational point of view, performance is the most important computational aspect of systems providing weather forecasts, as these must be generated in a short period of time, without losing sight of the energy consumption of these computational resources. Thus, a forecast for the next 12 hours should be computed in less than an hour in order to be useful for operations. For this reason, the use of more computational resources is demanded. Increasing the number of processors used on a weather simulation allows the problem to be split into smaller subproblems, but at the cost of an increased communication throughput. This increase in computational resources cannot be sustained indefinitely; at some point, the computational workload of the subproblems will be so low that communications will become a bottleneck, avoiding further reduction in the computation times.

The Weather Research and Forecasting (WRF) model package is a well-known weather forecasting software used extensively around the world. WRF makes use of parallel computing, which has allowed it to be executed on many supercomputers. The authors used the Advanced Research WRF (WRF-ARW) to provide 24-hour-ahead forecasts every 12 hours for the interest points of the events of the Winter Olympics 2018.

WRF performs a domain-based distribution of the workload, using the Message Passing Interface (MPI) as the communication protocol, where the domain is the target region on Earth to be simulated. The domain is partitioned among the available MPI processes, distributed on the available processors. The processing complexity arises from two issues: (i) the simulation resolution, which governs the complexity of the integration step; and (ii) the grid/mesh dimensions, which determine the size of the subdomains.

Each of these questions is influenced by different properties of the underlying processing platform. For example, the processing capacity of every processing unit greatly determines the quality and performance of integration step (i). As the resolution increases, i.e., the mesh subdomains become smaller, the integration step is shortened causing an increase in the required total integration steps.

This paper focuses, on the one hand, on the second issue (ii), which establishes the number of subdomains, each subdomain having a fixed number of cells. WRF evenly splits the general domain into n subdomains, n being the number of available MPI processes, which are arranged in a 2D grid $(x \times y)$.

The process distribution parameters x and y greatly determine the overall performance of the simulation depending on the dimensions of the subdomain mesh, i.e., the dimensions $(x \times y)$ of the domain grid. For example, with 25 MPI processes, the domain can be decomposed into three possibilities: (25×1), (5×5), and (1×25). It may be thought that distributing processes in a (25×1) layout performs similarly to distributing them in a (1×25) layout, but we show that this was not the case with WRF. In fact, there was an enormous difference in performance between the two layouts. For this reason, we studied the impact of the different process distributions on the simulation times and the reasons for this impact and proposed a new distribution algorithm that works better than the one implemented by WRF.

On the other hand, we also studied how increasing the number of used processing resources decreases the wall time of the simulation at the cost of losing efficiency for computing the same workload, dramatically increasing the energy consumption. In consequence, we found a balance between overall performance and energy consumption, which indicated the best process distribution when energy consumption was also a factor. Note that exascale platforms will be composed of thousands of processors, which means that a slight reduction in the energy consumption of each of them would substantially reduce the energy consumption of the platform as a whole.

The rest of this paper presents an overview of the current state of the art in Section 2. All the methods used in this work are detailed in Section 3. In Section 4, we study the effects of different process distributions on the overall performance and propose a new method to distribute them in Section 5. In Section 6, we study how an increase of the available MPI processes, as well as the variability of the processing capacity of each of these processes, affects the efficiency of the distributed computations and the energy consumption, thinking on an implementation of WRF for exascale. Finally, we present our conclusions and future work in Section 7.

2. Precedents and Related Work

WRF performance may be affected by different factors. Some are related to the software used in the compilation and execution phases (C and Fortran compiler, MPI library, and use of threads) and others to the configuration of a certain case study (physical model used, resolution requested, domain mesh, . . .). In this paper, we will focus on three topics: how scalable WRF is when the number of available MPI processes increases; how the dimensions of the domain mesh affects performance; and how we can save energy while achieving good performance. In the literature, there are a number of papers addressing these topics.

Malakar et al. [1] focused on improving and analyzing the performance of nested domain simulations. They showed a significant reduction (up to 29%) in runtime via a combination of compiler optimizations, mapping of processes to physical topology, overlapping communication with computation, and parallel communications. They also concluded that high-resolution nested weather simulations are a challenge in terms of scaling to a large number of processors and considered it critical that practitioners choose a good nesting configuration.

Christidis [2] concluded that significant performance improvements, due to better cache utilization, can be obtained with a proper choice of the parameters nproc_x, nproc_y, and numtiles. Smaller contiguous arrays fit more efficiently in local caches, especially in the cases of "thin" decompositions (nproc_x < nproc_y) which allow computations with minimal cache misses.

In the same line, Johnsen et al. [3] investigated a "best fit" node placement scheme when using 2 OpenMP threads per MPI rank, 8 MPI ranks on each Cray XE6 "Blue Waters" node. By default, the XE6 job scheduler places MPI ranks in serial order on the machine, but halo exchange partners are not mapped this way in WRF. Using an alternate placement allows 3 communication partners to be obtained for most MPI ranks on the same node. At very high scales, this strategy improves overall WRF performance by 18% or more. The WRF grid is decomposed into rectangles with latitudes longer than longitudes for each subdomain. The optimized placement employed has the benefit of sending smaller east-west direction exchanges off-node and keeping as many larger north-south messages on-node as possible.

Shainer et al. [4] concluded that although interconnect type was the greatest determinant in improving WRF scalability, it was also observed that overall cluster productivity could be improved by up to 20% by running simultaneous jobs on the cluster rather than allocating the entire cluster to a single job. This increase in productivity was the result of two factors: (i) core and memory affinity, which reduces the remote memory access penalties and increases cache hits, and (ii) parallel jobs with smaller core counts help reduce the synchronization overhead for each application.

Kruse et al. [5] studied WRF scalability to several thousand cores on commodity supercomputers using Intel compilers and found that total time decreased between 512 and 2K cores and increased beyond 2K cores. While the computation time scaled well with increasing numbers of cores, the time to complete operations involving I/O increased, outweighing the gains in simulation speed at 2K cores and beyond.

For a very long time, computing performance was the only metric considered when launching a program. Scientists and users were only concerned about the time it took for a program to finish. Though still often true, the priority of many hardware architects and system administrators has shifted to an increasing concern for energy consumption. High-performance computing consumes ever-larger volumes of electricity, and the reduction of consumption saves an appreciable amount of money.

One group of methods to reduce energy consumption focuses on how to distribute the workload among the cores of the computer. In this area, Lagravière et al. [6] compared the performance and power efficiency of Unified Parallel C (UPC), MPI, and OpenMP by running a set of kernels from the NAS Benchmark. They focused on the Partitioned Global Address Space (PGAS) model, and their main conclusion is that UPC can compete with MPI and OpenMP in terms of both computation speed and energy efficiency, but the data show that OpenMP consumes less energy than the others.

Igumenov and Žilinskas [7, 8] measured the power consumption of multicore computers with different computing loads: when the computer is idle and when some cores are fully loaded. The mean power consumption per core decreases when the computing load increases. Hence, running computers with greater loads is preferable to distributing parallel tasks among separate multicore computers.

Aqib and Fouz [9] compared the time and energy consumption of different tasks using different parallel programming models (OpenMP, OpenMPI, and CUDA). Their results, which can be generalized, outline the effect of choosing a programming model on the efficiency and energy consumption when running different codes on different machines. The parallel programming models obviously only improve the efficiency and reduce the energy consumption if there are blocks of codes that can be parallelized. Their conclusion is that OpenMPI performs much better than the other parallel models considered.

To summarize, the first works mentioned in this section seek to improve performance by modifying different parameters, both software and hardware, but unlike our work, in no case have they presented an exhaustive analysis that provides a heuristic to determine a distribution of processes close to optimal. In all these works, there is no concern for energy consumption. In the following works, different techniques are presented that allow reducing the consumption of energy for certain established test benches. Although they show the way forward, we have carried it out through software in a real situation.

3. Materials and Methods

3.1. Application Case. This work was conducted within the ICE-POP 2018 project, where the collaborating agencies were tasked with supporting the Winter Olympics by providing different kinds of weather information. A mid-resolution simulation with WRF has approximately 1 to 4 km of resolution per cell, with an integration step in the order of seconds. For the ICE-POP 2018, a resolution of about 300 m was required, which imposed integration steps lower than a second. Figure 1 shows the three nested domains with different resolutions that were computed in every simulation over the Korean peninsula. Different parameters from WRF forecasts, such as the visibility or the humidity, were extracted for the points of interest of the Olympics. This kind of information was extremely useful for planning the events. We performed the same kind of simulations over the same geographical region for all our studies. WRF-ARW version 3.9.1 with customized configuration (UCLM-WRF) was used by the authors to obtain the weather forecasts. WRF is open-source, so the source code can be obtained from the National Center of Atmospheric Research (NCAR) website (http://www2.mmm.ucar.edu/wrf/users/download/get_source.html). The configuration made use of state-of-the-art P3 microphysics [10], the Rapid Radiative Transfer Model scheme [11] for radiation, and the Noah Land Surface Model [12] for the surface.

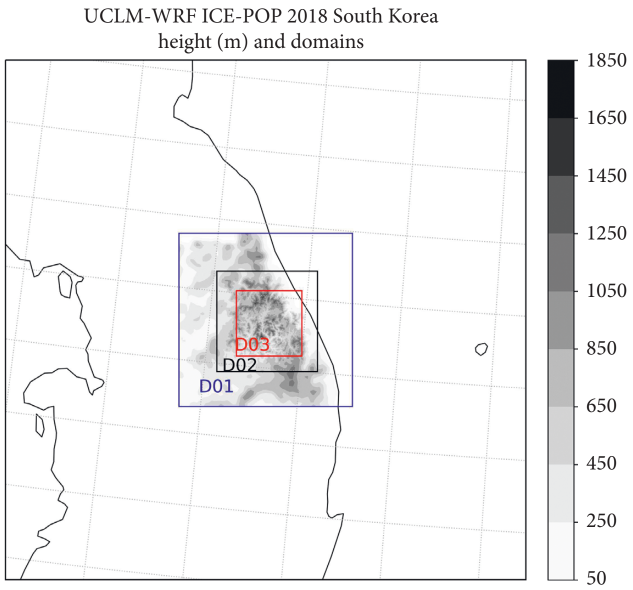

FIGURE 1: WRF domains used in our simulations over the Korean peninsula. Three nested domains D01, D02, and D03 with different resolutions. Terrain heights in meters are plotted inside the domains.

3.2. Test Platform. We used the GALGO Supercomputer to perform all the tests in this work. GALGO is located at the Albacete Research Institute of Informatics, Spain, and hosts all kinds of scientific research. GALGO is a cluster of approximately 1200 processing cores, half of which are provided by *Intel Xeon E5450 3.0 GHz* processors. Each processing node is dual-socket, mounting two processors with shared DRAM and a 40 Gb/s dual-port *Mellanox ConnectX-2 Infiniband* interface. We used up to 40 of GALGO's processing nodes (320 processing cores) for our tests. The topology of the network is depicted in Figure 2; it consists of a first level of 24-port DDR switches and a second level of one 36-port QDR switch, with link rate of 20 Gb/s. Sixteen computing nodes are connected to each 24-port switch.

3.3. Compilation Options. The choice of compiler, compilation options, and MPI implementation has a big influence on the runtime of a simulation. In this work, we used the

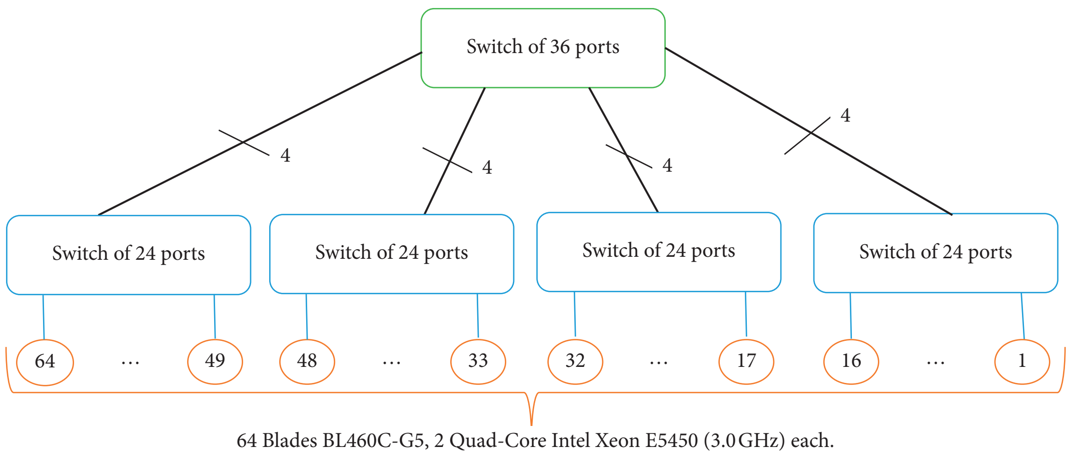

Figure 2: Topology of the network.

fastest binary codes, which are the best combination we can get, that is, the executable that allows us to run a simulation in the shortest possible time of all combinations. In our case, we used Intel compilers with Intel MPI libraries (ver. 2018.0.128) to compile all the required programs and dependencies. For best performance, we used the -O3 compilation option to activate the most aggressive optimizations available. We also used the -xHost compilation option to use the SIMD capabilities of the processor (SSE4). We empirically verified that the hybrid MPI-OpenMP (dm + sm) WRF compilation performed better than the other options and hence used this configuration in our tests.

3.4. Simulations. Our simulations covered the target region where the Winter Olympics were held, simulating 25 January 2018 from 06:00 to 06:30 for all the tests. A typical simulation for the Winter Olympics covered 24 simulated hours, but we used a reduced simulated time in our tests because of the large number of simulations these test involved. We used three one-way nested grids (see Figure 1) with sizes 199×199 for the inner grid (300 meters per cell), 103×103 for the intermediate grid (900 m), and 60×60 for the outer grid (2700 m), each of them with 70 vertical levels. Furthermore, the very high resolution of 300 meters per inner cell required substantial processing power per iteration. Numerical stability greatly depends on the resolution of the input geographical data, and therefore we used a high-resolution dataset of the Korean peninsula provided by the ICE-POP project instead of the default WRF Preprocessing System (WPS) Geographical Input Data.

3.5. Mesh Distribution. WRF performs an automatic distribution or layout of the simulation domain/grids among the available MPI processes, based on a Cartesian topology (MPI_Cart_create). As a result, it splits the domain into the most orthogonal coordinates possible $(x, y) \in \mathbb{N}$ or $(x \times y)$, assigning every coordinate to an MPI process. The x and y values can be overridden, so in order to check whether the most orthogonal layout is the best domain distribution, we performed additional simulations with all the possible $(x \times y)$ combinations for a set of n values, where n represents the number of processing nodes. In our experiments, the number of MPI processes matches the number of nodes n because we only used one MPI process per node. An easy way to change the x and y coordinates in WRF is through the edition of the namelist.input file needed to run WRF. In the namelist.input file, nproc_x controls the x coordinate and nproc_y controls the y coordinate.

3.6. Time Measurement. The basic measurement for our experiments is the average wall time, which is defined as

$$\mu_w = \sum_{i=1}^{z} \frac{W_i}{z}, \quad (1)$$

where W_i is the wall time for the i simulation and z the total number of iterations. Every test is composed of 10 iterations, i.e., executed 10 times, and then μ_w is obtained using (1).

3.7. Speedup and Efficiency. The average wall time μ_w is not appropriate to characterize how scalable parallel software is. For this reason, the analyses are usually done over the speedup (see Kumar et al. [13]) value:

$$S = \frac{\mu_{\text{ref}}}{\mu_{\text{faster}}}, \quad (2)$$

where μ_{ref} is the reference average wall time which depends on the study, e.g., for scalability, it is usually the time for the case with fewer processing units. μ_{faster} is the time of the case we are interested in analyzing. We can also measure the parallel efficiency E of a case as

$$E = \frac{S}{p}; \quad p = \frac{n_{\text{faster}}}{n_{\text{ref}}}, \quad (3)$$

where S is the speedup of the case, n_{faster} is the number of processes corresponding to μ_{faster}, n_{ref} is the number of processes corresponding to μ_{ref}, and p is how many times more processes the selected case has with respect to the reference case.

3.8. Energy Estimation. There is currently great interest in carrying out efficient implementations, both from the computational point of view (less execution time) and from the energy consumption perspective (less energy is needed). The power consumption of the entire platform is measured in Watts, and it is measured in Joules when the energy consumption is considered. In order to approximately estimate the power consumption, we have to know the amount of energy consumed in Watts by each processor. For the processors of the experimental platform, the vendor specifies this number as 80 W on average. In order to estimate the energy consumption in average from the average wall time μ_w, we defined

$$J = \mu_w \times n \times W_c, \tag{4}$$

where J represents the estimated Joules consumed by n nodes and W_c is the amount of Watts per processor according to vendor specifications. Therefore, W_c takes a value of 80 W when considering 2 or 4 cores (1 processor per node) and 160 W when considering 6 or 8 cores (two processors per node). We suppose that idle processors do not consume energy, which is not actually true, as seen in some studies such as Igumenov and Žilinskas [7]. We do this because we only have processors with the same model and a fixed number of cores, so we need to ascertain whether a processor with a reduced number of cores would be more energy efficient.

3.9. Experiment Setup. As a summary of the information shown in this section, the experiment setup is detailed below:

(i) Machines

  (1) Up to 40 of GALGO's processing nodes with Intel Xeon E5450 3.0 GHz processors. Each processing node is dual-socket, mounting two processors with 32 GB DRAM and a 40 Gb/s dual-port Mellanox ConnectX-2 Infiniband interface.

(ii) Software

  (1) Weather Research and Forecasting (WRF) version 3.9.1, compiled for hybrid MPI-OpenMP platforms (option "dm + sm" in WRF configuration).
  (2) Intel compilers with Intel MPI libraries (version 2018.0.128). Compilation options used: -O3 and -xHost.

(iii) Methodology

  (1) All WRF simulations were performed with the same parameters; the only change was the number of processes and the distribution of these processes over the nodes.
  (2) We performed experiments using different distributions of $n = 9$, 16, 25, and 36 nodes. These values of n were used because they allow us to use an exact orthogonal distribution, that is, (3×3), (4×4), (5×5), and (6×6), which is the default in WRF.
  (3) For a given number of nodes n, we considered all the different distributions of nodes in a 2D mesh.
  (4) Average wall time was obtained for n processing nodes and 8 cores per node with different domain distributions. Every domain distribution was executed 10 times.
  (5) Energy consumption is estimated from wall time and the amount of Watts per processor according to vendor specifications.

4. Analysis over WRF Process Distribution

All our WRF simulations were performed with the same parameters; the only change was the number of processes and the distribution of these processes over the processors. The WRF distribution algorithm assumes that the best layout for distributing the processes is the one that most preserves orthogonality, i.e., for n processes, the distribution is approximately $(\sqrt{n} \times \sqrt{n})$. We performed experiments using different distributions of $n = 9, 16, 25$, and 36 nodes. These values of n were used because they allow us to use an exact orthogonal distribution, that is, (3×3), (4×4), (5×5), and (6×6).

The question is do these orthogonal layouts provide the best times? In our experiments, we took into account all the different distributions of nodes in a 2D mesh with a fixed n. For instance, in the case of $n = 16$ nodes, the possibilities are (16×1), (8×2), the orthogonal (4×4), (2×8), and (1×16). In Table 1, we can see the layout effects on μ_w for $n = 9, 16, 25, 36$. The underlined values correspond to the automatic layouts used by WRF and the best values of μ_w are marked in bold.

The results in Table 1 show that even though the layouts chosen by WRF are usually "good enough," they are not the best. Therefore, based on the results, we state that the most orthogonal layouts are **not always** the best performing layouts. The results also corroborate the theory, that is, $(x \times y)$ is **not equal** to $(y \times x)$ in terms of performance due to the fact that different $(y \times x)$ process distribution involves different communication patterns.

4.1. WRF Communication Behavior. In order to explain the above finding, we looked at the communication behavior. It is known that MPI communications can be a bottleneck in programs with low computational workload and large number of communications among processes. We looked at two main factors that greatly influence the overhead introduced by MPI or any other message passing strategy: (i) the amount of data shared among the MPI processes and (ii) the number of transactions needed to share that amount of data. When these aforementioned factors exceed the capacity of the underlying platform, especially the interconnection network, the performance is greatly degraded.

Because the example of $n = 36$ has more distribution combinations than the others, we used this example to

Table 1: Average wall time μ_w when using n processing nodes and 8 processor cores with different domain distributions.

$(x \times y)$	$n = 9$	$(x \times y)$	$n = 16$	$(x \times y)$	$n = 25$	$(x \times y)$	$n = 36$
9×1	3443	16×1	2596	25×1	2117	36×1	—
3×3	*2403*	8×2	1714	5×5	**1017**	18×2	1152
1×9	**2327**	4×4	1458	1×25	1298	12×3	951
		2×8	**1396**			9×4	861
		1×16	1622			6×6	*824*
						4×9	**801**
						3×12	855
						2×18	908
						1×36	—

The automatic distributions chosen by WRF are underlined and the ones with the best μ_w are marked in bold. Some extreme distributions such as (36×1) crashed the simulations and could not be executed.

analyze the communications, using the statistics provided by the Intel MPI library. Therefore, we plotted the amount of data in MB injected into the intercommunication network (i) by the $n = 36$ distributions in Figure 3 and the number of transactions of every distribution (ii) in Figure 4.

Looking at the results, we can see that the more orthogonal combinations show the minimum amount of data transferred (i); however, they present a higher amount of transactions (ii). With WRF, we see that the amount of MB has a much greater impact on the wall time than the number of transactions. Moreover, if we suppose that communications are the most important factor for performance in our WRF simulations, $(x \times y)$ performance should be similar to $(y \times x)$ performance. If we look at Figure 5, we can see that the observed μ_w for every distribution of $n = 36$ is not the expected μ_w (approximation) in a situation where the communications are the most important factor.

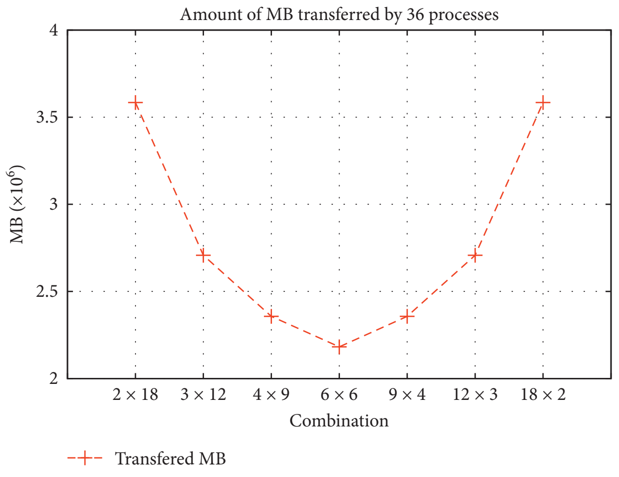

Figure 3: Total amount of data in MB injected into the interconnection network by the MPI processes. The distributions correspond to the case when $n = 36$.

4.2. WRF Integration Step Analysis. Consequently, communications were not the reason for the differences in performance between $(x \times y)$ and $(y \times x)$ distributions. As observed in Christidis [2] and Johnsen et al. [3], combinations where $x < y$ work better due to enhanced cache usage.

The WRF Fortran routine that is executed in every integration step is solve_em(), which is defined in solve_em.F source file. All the MPI and OpenMP functionality is used in this file. As previously stated, every subdomain of the $(x \times y)$ distribution is assigned to a MPI process. As an example, in Figure 6, we can see how WRF divided the inner domain $(199 \times 199 \times 70$ cells) of our simulations when using the (6×6) process distribution, where $x = 6$ and $y = 6$. For the sake of clarity, the cells of the domains are indexed by the i index (latitudes), the j index (longitudes), and the k index (vertical levels). This (6×6) distribution generated a subdomain for every MPI process with 33×33 cells $(i = 33, j = 33, k = 70)$, each of those cells having 70 vertical levels (see the subdomain of case 6×6 in Figure 6).

When using OpenMP, every one of these subdomains is divided into tiles, which can be automatically handled by WRF (usually, tiles = threads) or manually set through the numtiles parameter in the namelist. The solve_em routine contains many loops over the target subdomain of every MPI process, and every loop iterates the corresponding tiles of the subdomain using OpenMP threads (OMP PARALLEL DO). A pseudocode of the WRF integration step (solve_em) is presented in Algorithm 1.

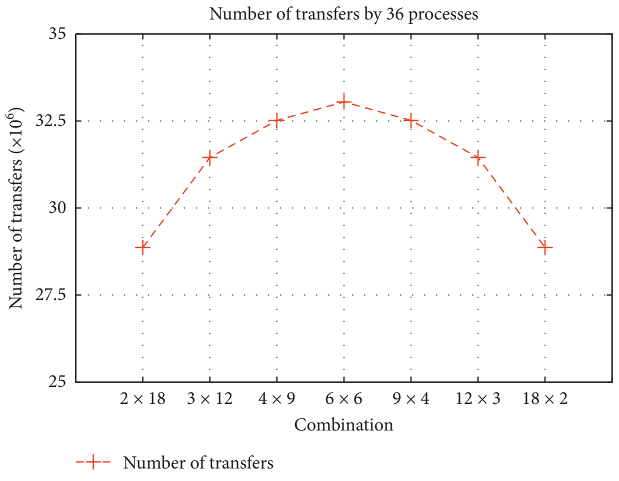

Figure 4: Total number of transactions performed by the MPI processes on the interconnection network. The distributions correspond to the case when $n = 36$.

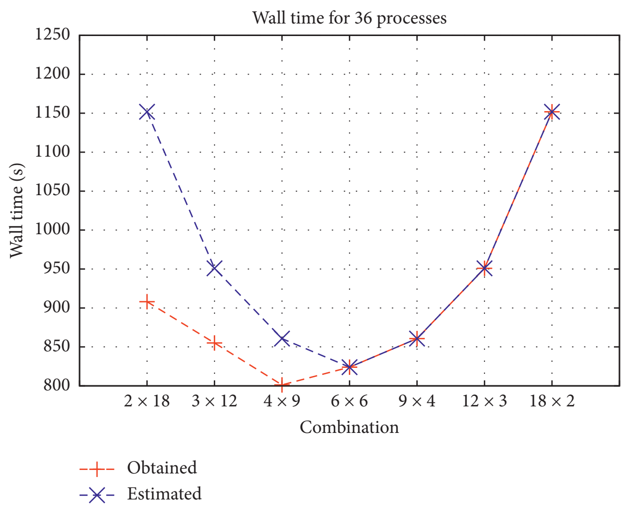

Figure 5: Execution time μ_w obtained on the tests and expected execution time for the different distributions of $n = 36$ processes.

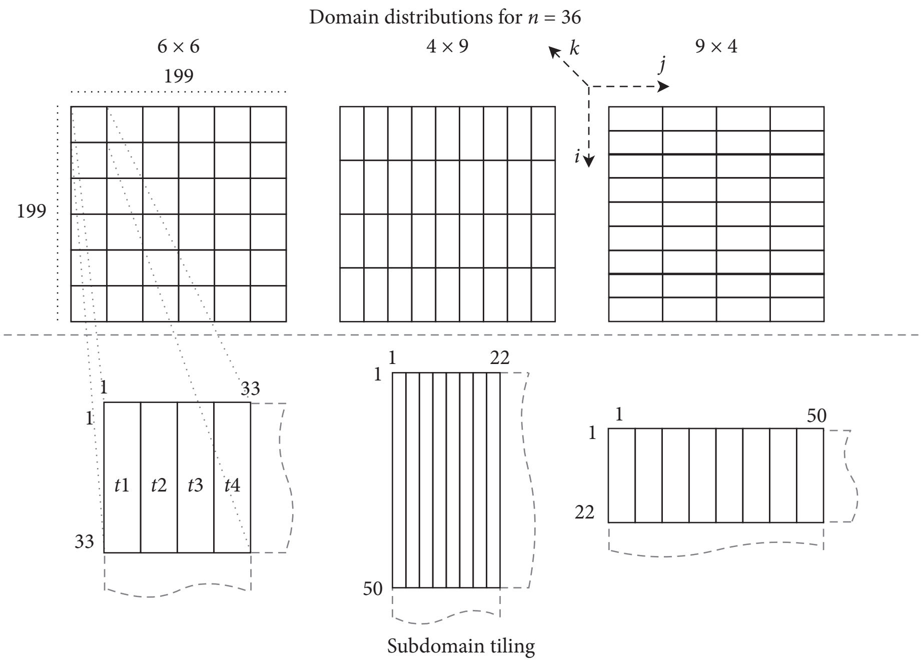

Figure 6: Process distributions over the inner domain when using $n = 36$ processes over the inner domain. The domains had 199×199 cells, each with 70 vertical levels. Every subdomain was divided in a number of tiles that was automatically chosen by WRF.

To be efficient with Fortran memory layout, every time the integration step computes anything over the target tile, it performs three nested loops in the right order (from inner to outer loop: i, k and then j). Thus, the problem lies elsewhere. In Figure 6, we can see the effects on the tiles when using three different distributions for $n = 36$. Remember that in terms of μ_w: $(4 \times 9) < (6 \times 6) < (9 \times 4)$, as we saw in Table 1. If we look at the three cases in Figure 6, the difference is the dimensions of i and j. When WRF computes every tile or slice of the corresponding subdomain, it performs better when the i dimension is large because of better cache usage.

In order to understand this effect, we need to look at how WRF maps the subdomains in memory. The i dimension is contiguous in memory, but not the k and j dimension because the subdomain is mapped on top of a larger memory layout where additional cells are allocated (e.g., the halos). These discontinuities in the three nested loops interfere with the performance of the cache when changing the k or j indexes, slowing down the computation.

Therefore, lowering the j dimension increases the i dimension, resulting in better cache usage and performance. Again, if we look at Figure 6, we can see that in the case of the (4×9) distribution, which is the fastest of the three, the i dimension is larger than in the other two cases, making it better performing than the others.

In the ideal case that the communications did not matter, the (1×36) distribution would be the fastest because a better cache usage ($i = 199, j = 5$ per subdomain). In

```
(1) function COMPUTE_SOMETHING(tile)
(2)   for $j = 1$ to $tile\_max\_y$ do                ▷ Iterate the tile
(3)     for $k = 1$ to $max\_vertical$ do            ▷ The iteration order is OK
(4)       for $i = 1$ to $tile\_max\_x$ do
(5)         tile[$i$][$k$][$j$] = . . .              ▷ Do something on the grid
(6)       end for
(7)     end for
(8)   end for
(9) end function
(10) procedure SOLVE_EM(subdomain)
(11)   . . .                                        ▷ Initialization
(12)   !$OMP PARALLEL DO                            ▷ One thread per tile
(13)   for $t = 1$ to numtiles do                   ▷ Iterate tiles
(14)     compute_something(subdomain[t])            ▷ Pass the tile to the function to compute something
(15)   end for
(16)   !$OMP END PARALLEL DO                        ▷ End of parallel region
(17)   if DM_PARALLEL then                          ▷ If MPI is being used
(18)     HALO_EM∗∗∗.inc        ▷ Send halos to other adjacent processes depending on the stencil
(19)   end if
(20)   . . .   ▷ Replicate the same loop structure tens of times for the different variables to compute on the subdomain
(21) end procedure
```

Algorithm 1: Integration step (solve_em) pseudocode obtained from WRF Fortran code.

practice, this is not the case because in these extreme cases, communications escalate the computing times (see Figures 3 and 5 and Table 1). As a conclusion, a balance between the next two factors must be found when using WRF:

(1) Communications perform better when the process distribution is orthogonal

(2) Cache performance improves when larger values of i (latitudes) are used in the tiles

5. Improved Distribution Algorithm for WRF

Drawing on the results described in Section 4, we propose an alternative algorithm to distribute the layout of processes based on an α value. We observed that with a lower x dimension than y dimension (better cache usage), performance is much better when the workload of every process is sufficient to negate the communication overhead.

Thus, we devised a method to obtain a better distribution from an α ratio applied to n processes, balancing good cache usage with an acceptable increase in the communication overhead. In fact, the ratio between the values of x and y seemed to be the same, which we defined as

$$\alpha = \frac{x}{y}. \tag{5}$$

Depending on α chosen, we can obtain a good value for x that can be used to obtain an optimal $(x \times y)$ distribution of n MPI processes. In order to derive x, we have the system of equations composed by (5) and

$$xy = n, \tag{6}$$

$$y = \frac{n}{x}. \tag{7}$$

Solving (5)–(7) we obtain

$$\begin{aligned}\alpha &= \frac{x}{y}\\ &= \frac{x}{n}\\ &= \frac{x^2}{n}.\end{aligned} \tag{8}$$

Then, we clear the x from (7) and (8):

$$x = \sqrt{\alpha n}. \tag{9}$$

Equation (9) provides a way to obtain a good x value from a specific n. We also define $f(x, n)$ as a function which returns the divisor x' of n which is the nearest integer divisor to the value of x. Having a specific n, we can use (9) to obtain a near-optimal distribution $(x' \times y')$:

$$(x' \times y') = \left(f(x,n) \times \frac{n}{f(x,n)}\right). \tag{10}$$

5.1. Deriving the Optimal Alpha. Equation (9) needs an α value to be useful, and this value should minimize the overhead generated by the communication and cache misses. We have a domain with $l \times l$ latitude/longitude dimensions, and we subdivide the domain into $x \times y = n$

subdomains. From this, we obtain that the latitude/longitude dimensions of every subdomain are $((l/x) \times (l/y))$.

The communication overhead for every subdomain can be calculated as the length of the perimeter multiplied by the communication overhead A for every point at the perimeter. The communication overhead for the subdomain, using a specific x, is therefore defined by the following formula:

$$o_{\text{comm}}(x) = 2A\left(\frac{l}{x} + \frac{l}{y}\right) = 2A\frac{l}{x} + 2A\frac{lx}{n}. \quad (11)$$

Then, we derive (11) and equate to zero:

$$o'_{\text{comm}}(x) = -2A\frac{l}{x^2} + 2A\frac{1}{n} = 0 \longrightarrow x = \sqrt{n}. \quad (12)$$

The last equation proves that the communications overhead is minimum when $x = \sqrt{n}$, as we already saw in Figure 4.

Following the same line of thought, we derived the overhead of the cache misses. The cache miss overhead could be defined by the dimension of the longitudes (l/y) multiplied by the overhead B introduced for every cache miss. From this, we obtain the cache miss overhead for a specific x as

$$o_{\text{miss}}(x) = B\frac{l}{y} = B\frac{lx}{n}. \quad (13)$$

Equation (13) represents a monotonic increasing function whose minimum value is obtained for $x = 1$. This result supports the finding that the lower the y dimension is, the better the cache usage is. From the previous results, we can obtain the combined overhead $o(x)$ of (11) and (13):

$$o(x) = o_{\text{comm}} + o_{\text{miss}}(x) = 2A\frac{l}{x} + 2A\frac{lx}{n} + B\frac{lx}{n}. \quad (14)$$

After deriving (14) to obtain the x value where the combined overhead is minimum, we obtained:

$$o'(x) = -2A\frac{l}{x^2} + 2A\frac{l}{n} + B\frac{l}{n} = 0 \longrightarrow x = \sqrt{\frac{2An}{2A+B}}. \quad (15)$$

Therefore, from (9) and (15), the optimal α where the combined overhead is minimum is

$$x = \sqrt{\frac{2An}{2A+B}} \longrightarrow \alpha = \frac{2A}{2A+B}. \quad (16)$$

This result clearly shows the influence of both overheads (communication and cache misses) on the final computing performance.

5.2. Obtaining an Alpha Value. Note that A and B are unknown constants in equation (16), which impeded us from calculating the optimal α. On the other hand, we still needed to assign a value to α in order to obtain any distribution using (10). The problem is that it is not trivial to theoretically (or even empirically) calculate these constants and so we attempted another empirical approach to obtain an approximation of the optimal α.

For this purpose, we first defined α_n values for each of our training cases. From Table 1, we obtained the $(x_n \times y_n)$ distributions with the best μ_w of every n. We then used the x_n and y_n from these distributions to define α_n as

$$\alpha_n = \frac{x_n}{y_n}. \quad (17)$$

In the same way, we obtain the values α_n for the distributions presented in Johnsen et al. [3] (Table 2). Finally, we fit a $\sqrt{\alpha \cdot n}$ curve to these data (α_n values) and obtain as result $\alpha = 0.43$.

5.3. Near-Optimal Process Distribution. After applying (10) and $\alpha = 0.43$ to our values $n = 9$, $n = 16$, $n = 25$, and $n = 36$, we obtained the optimal distributions of (1×9), (2×8), (5×5), and (4×9), respectively. In these cases, WRF picked the orthogonal distributions, which were suboptimal. Our process distribution implementation is presented in Algorithm 2. Algorithm 2 admits two call parameters, the number of nodes considered (n) and the alpha value, and returns the value of x which is divisor of n and which is closer to the one calculated by the formula $x = \sqrt{\alpha n}$. To do this, function $F(x,n)$ first looks for values smaller than x (decreasing a unit in each step) until it finds a divisor of n and then repeats the process with values greater than x. Finally, both obtained values are compared and the one closest to the initial x is chosen. Finally y is calculated as $y = n/x$.

We were unable to perform simulations with n higher than 40 MPI processes in our platform, but we applied our distribution algorithm to the n values in Johnsen et al. The resulting x values from our algorithm (marked with $\times$) are shown in Figure 7 along with the x values used by Johnsen et al. (marked with +). The top solid line represents $x = \sqrt{n}$ (orthogonal), whereas the bottom solid line represents the x values obtained by (9). Using our algorithm achieves a smoother adjustment of x while increasing n.

To perform our tests with different distributions, we used the outputs of Algorithm 2 to change the value of the nproc_x and nproc_y parameters in the WRF namelist file. Nevertheless, the WRF distribution code can be modified to implement our algorithm without the need to externally modify (via automatic scripting or manual way) the namelist.

6. WRF Scalability and Energy Analysis

The improved distribution algorithm allowed us to boost in performance for our simulations, but we wanted to determine how this performance could be further increased. The second part of this work consisted of a study of the scalability of WRF and its efficiency when increasing the available processing power from the perspective of performance and energy consumption. Our target was to increase performance without significantly increasing the energy consumption.

Table 2: Rounded average wall times μ_w in seconds when using n processing nodes and c processor cores per node.

$c\backslash n$	10	20	30	40
8	2024	1182	896	774
6	2288	1333	964	812
4	3049	1733	1220	952
2	3895	2316	1612	1239

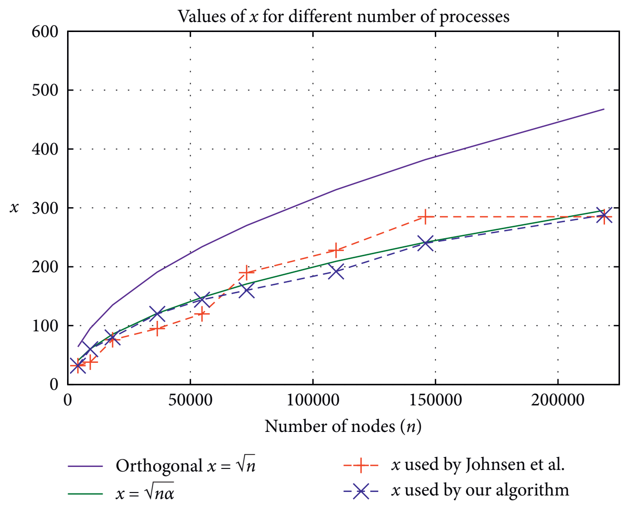

Figure 7: Final x values obtained from our distribution algorithm. The orthogonal $x = \sqrt{n}$ values (top solid line) and raw values from equation (9) (bottom solid line) are plotted. The x values obtained by our algorithm are very similar to those chosen by Johnsen et al.

```
Require: $n \in N$, $\alpha \in \mathbb{R}$, $n > 0$ and $\alpha > 0$
(1) function $F(x, n)$    ▷ $f(x, n)$ implementation
(2)   for $i = 0$ to $n$ do  ▷ Iterate possible decrements
(3)     $x_f \longleftarrow x - i$
(4)     $r \longleftarrow n \bmod x_f$
(5)     if $r == 0$ then
(6)       break ▷ Nearest divisor below $x$ has been found
(7)     end if
(8)   end for
(9)   for $i = 0$ to $n$ do ▷ Iterate possible increments
(10)    $x_c \longleftarrow x + i$
(11)    $r \longleftarrow n \bmod x_c$
(12)    if $r == 0$ then
(13)      break ▷ Nearest divisor over $x$ has been found
(14)    end if
(15)  end for
(16)  if $(x - x_f) < (x_c - x)$ then ▷ Return the nearest to $x$
(17)    return $x_f$
(18)  else
(19)    return $x_c$
(20)  end if
(21) end function
(22) procedure DISTRIBUTE$(n, \alpha)$ ▷ Distribution algorithm
(23)   $x \longleftarrow \sqrt{\alpha \cdot n}$     ▷ Get the first candidate
(24)   $x \longleftarrow f(x, n)$  ▷ Get the nearest divisor of $n$ to $x$
(25)   $y \longleftarrow n/x$
(26)   Distribute processes using the $(x \times y)$ distribution
(27) end procedure
```

Algorithm 2: New WRF process distribution algorithm and $f(x', n)$ implementation.

6.1. Scalability. For our scalability study, we tested different cases using a variable number of computing nodes n and processing cores c from these nodes. After performing 10 simulations of every combination and applying (1), we obtained the μ_w values shown in Table 2. The same results are presented in Figure 8, where we can see the enormous differences in performance when n is low. In contrast, when n is high, the four cases shorten the distance between each other and converge to the same values of μ_w.

The processing cores c were handled by OpenMP threads, which provided good performance and a reduced memory footprint. After applying (2) and (3) to the values in Table 2, we obtained the efficiency values e, presented in Table 3.

From the information in Tables 2 and 3, we inferred the following observations:

(i) As expected, the average time per simulation is reduced when increasing the total number of processing units used $(n \times c)$

(ii) Decreasing the number of c cores per node and increasing the number of nodes n greatly increase efficiency e

(iii) When the total number of processing units $(n \times c)$ is greater than 180, efficiency e plummets

From the first observation, we see that increasing the number of nodes n reduces the wall times in all the cases when the number of c used per node is constant. In the case of using all the cores of every node, efficiency is greatly undermined when increasing the number of nodes. However, when the number of used cores per node is four or less, the efficiency keeps stable. This circumstance is clearly observed in Table 3.

One clear example to see the impact of using more nodes with limited cores is when comparing the cases that used 80 processing units: $(n = 10, c = 8)$, $(n = 20, c = 4)$, and $(n = 40, c = 2)$. The $(n = 20, c = 4)$ case is $\approx 14\%$ faster than the $(n = 10, c = 8)$ case, even at the expense of an increase in the MPI communications. Again, we see that communications are not the critical factor for performance when using WRF. The gap is even larger in the case of $(n = 40, c = 2)$ where only 25% (2 cores) of the nodes' processing capacity is being used, reducing the wall times by $\approx 38\%$.

After profiling the processor performance when executing $(n = 10, c = 8)$ and $(n = 40, c = 2)$ cases, we found that the number of minor memory page faults in the second case was $\approx 75\%$ lower than that in the first one. Additionally, the number of cache misses was reduced by 6% in the second case. We therefore conclude that the reason for these differences in performance is the better cache usage because of the reduced size of the data structure.

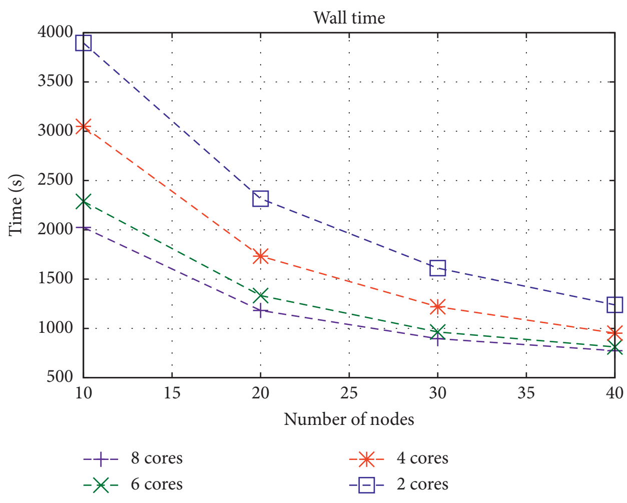

Figure 8: Execution time μ_w for different numbers of processing nodes n.

Table 3: Efficiency e for using n processing nodes and c processor cores, relative to the number of cores.

$c\backslash n$	10	20	30	40
8	1	0.86	0.75	0.65
6	1	0.86	0.79	0.70
4	1	0.88	0.83	0.80
2	1	0.84	0.81	0.79

When we combined the second observation with the conclusions of the first one, we saw that memory management greatly improves when the size of the data structures per node is low enough but also the number of processing cores c is not too high for that size. This is reflected in the efficiency e values obtained in Table 3 for the cases with less than 180 total processing units ($n \times c \leq 180$), which is also supported by the third observation. As a conclusion, we state that in order to maintain a high parallel efficiency, the size of the problem should be proportional to the used processing capacity. For the size of the problem in our simulations, 180 processing units are a good compromise between performance and efficiency. Increasing this number or the number of MPI processes n would divide the domains into small partitions that are too small and cannot be efficiently fed to the processors.

Looking at the previous observations, we conclude and recommend that WRF should be executed in computational resources that prioritize the size and latency of the cache memory more than the complexity and quantity of the computing cores. This recommendation is supported by the better efficiency observed in our simulations with the low values of c, which is in line with the results obtained in Shainer et al. [4]. Moreover, the available processing power should be appropriate to the size of the problem (resolution and domain dimensions), being neither too high nor too low.

Table 4: Estimated energy consumption J for using n processing nodes and c processor cores.

$c\backslash n$	10	20	30	40
8	3238400	3782400	4300800	4953600
6	3660800	4265600	4627200	5196800
4	2439200	2772800	2928000	3046400
2	3116000	3705600	3868800	3964800

6.2. Energy Efficiency. As we stated in the introduction section, when dealing with exascale platforms, a small reduction on performance could become a large reduction on energy consumption. This is why it is so important to balance the performance of the whole system when considering its energy consumption.

There is a limit at which increasing the number of computational resources barely improves performance, at the cost of skyrocketing energy consumption. Therefore, energy consumption is a factor when deciding how many computational resources are needed to satisfy the established requirements. Table 4 shows the estimated energy consumption J after applying (4) to the μ_w times presented in Table 2. The estimated energy consumption J is also presented in Figure 9, where we can see that the greatest energy savings corresponding to the use of 4 cores.

In consequence, it may be thought possible to find a good balance between the time spent on an execution and energy consumption. Evidently, if time is the most critical variable in an experiment, the energy consumption becomes irrelevant. For instance, if the simulations of this paper must be executed in less than 800 seconds, the only feasible configuration corresponds to 8 cores and 40 nodes. But, if the time to execute the simulation is represented by a limit, e.g., 1000 seconds, then we can play with other variables such as the energy consumption. We scatter plotted all the combinations in Figure 10 so we could choose a good combination for the last example. With 1000 seconds as a limit and looking at Figure 10, we have 4 options: $(n = 40, c = 4)$, $(n = 30, c = 8)$, $(n = 30, c = 6)$, and $(n = 40, c = 8)$. In this case, the best option is clearly $(n = 40, c = 4)$ because of the enormous difference in Joules between this and the other options.

In other cases, the best option is not so clear and depends on the priorities. For example, with a limit of 1900 seconds, the reader might agree with us that of all the possibilities, only three are feasible: $(n = 20, c = 4)$, $(n = 30, c = 4)$, and $(n = 40, c = 4)$. The $(n = 20, c = 4)$ case is the least energy consuming but much slower than the other two. $(n = 40, c = 4)$ consumes a little more than $(n = 20, c = 4)$ but, in contrast, is twice as fast. $(n = 30, c = 4)$ is a compromise between both. Depending on our priorities, we would choose

(i) $(n = 40, c = 4)$ for maximizing performance

(ii) $(n = 20, c = 4)$ for maximizing energy savings

(iii) $(n = 30, c = 4)$ if the priority is to harmonize performance and energy consumption

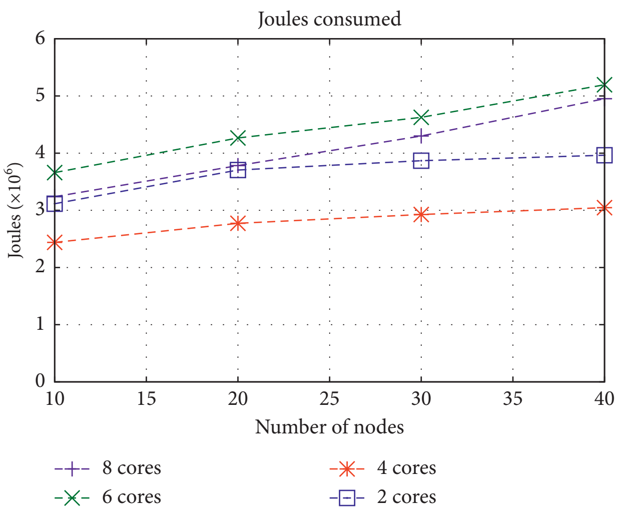

FIGURE 9: Estimated energy consumption J.

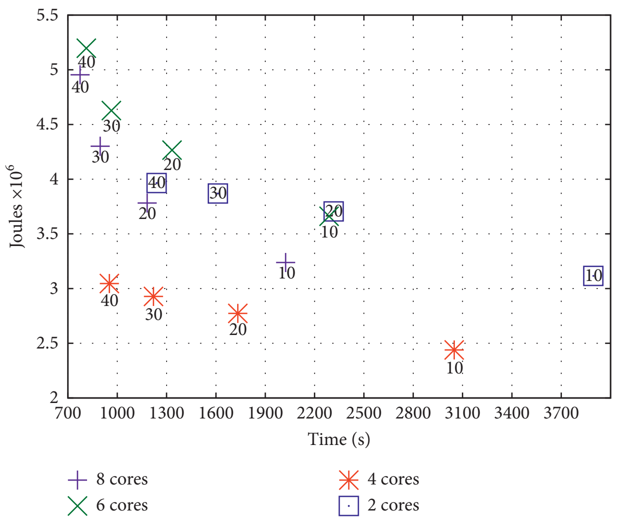

FIGURE 10: Scatter plot of time (ordinates) and energy consumption (abscissas) for the different combinations of n processes (below the points) and c cores.

In our simulations, we needed the fastest distribution possible, which was the $(n = 40, c = 8)$ distribution. The other option we considered was the $(n = 40, c = 4)$ distribution, which obtained a 38% energy saving with an increase of 23% in the wall time. The problem with this distribution was that the wall times were far from our target time requirements and therefore not a feasible choice.

7. Conclusions

This paper proposes a new distribution algorithm that works better than the one implemented by WRF. This algorithm was devised from the results of a performance study over different process distributions with a different number of total processes.

We also present a study of the performance of the WRF-ARW model in terms of three main variables: process distribution, execution time, and energy consumption. The authors had to comply with the execution time requirements imposed by the ICE-POP 2018 project, with regard to 2018 Winter Olympics held in Pyeongchang, but also taking into account the appropriate use of resources and energy consumption imposed by the authors' research institution. However, this study is essential for any research group working in this area, so this paper could be considered as a guideline.

Beyond this guideline, the following main contributions emerge from this work:

(i) Orthogonal distributions are optimal for communications and interprocessor performance.

(ii) Latitude dominant dimensions are optimal for cache usage and intraprocessor performance.

(iii) WRF performance therefore depends on the balance between orthogonality (communications) and efficient cache usage (longer latitudes per subdomain). We proposed an algorithm to obtain a balanced distribution.

(iv) In our platform, the WRF model works better when using less cores per node.

(v) WRF consumes less energy using less nodes, achieving a good execution time.

(vi) To execute WRF, it is thus recommended to use simple machines (not so many cores and more energy efficient than complex ones) with quality cache memories.

Related to the grid distribution used in the WRF software package, the authors evidence that for the platform used in this experiment, the best process distribution is not always the orthogonal one. To address this issue, the authors proposed a new distribution algorithm to calculate a better distribution layout than the default one implemented by WRF. As a future work, the authors propose to corroborate the results of this paper in other platforms completely different to that used in this work, especially platforms where the number of processes could be higher than 10,000. In this work, we obtained a good α value from near-optimal distributions used in other works, but we expect to find a better α value from the best process distributions when n is high enough. We also proved that the combined overhead of the communications and cache misses follows a relationship between two constants that we propose to determine in future works. These two constants would allow us to obtain the optimal process distribution for any number of MPI processes.

In order to test our algorithm in our simulations, we externally modified the WRF namelist's nproc_x and nproc_y parameters in our simulations, but the algorithm is easily implementable as an alternative distribution option inside WRF source code.

From the study of WRF source code, we think it is possible to optimize the data locality and hence remove the limitation of having latitude dominant dimensions to achieve the best performance. This could be achieved by modifying the details of the tiling processing code.

Furthermore, and from the previous contributions, we explored different ways of saving energy by changing the process locations. The authors used a graphical method to obtain the best configuration by plotting energy and time together. Depending on the priorities (performance or energy savings), different options could be chosen from this plot. In addition, in this work, we are not considering the use of accelerator due to the fact that the use of GPUs on WRF software is reduced to a short set of functions. Our purpose in this paper was to study the influence of process distribution both in terms of performance and energy consumption without comparing against other WRF implementations that consider GPUs. This comparison (with WRF and without using GPUs) could be interesting as a future work.

Data Availability

The data used to support the findings of this study are included within the article.

Disclosure

An earlier version of this manuscript was presented as a part of RM's PhD dissertation "Some critical HPC improvements in numerical weather prediction workflows."

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the Spanish Ministry of the Economy, Competitiveness, Science and Innovation (MINECO) (grant nos. CGL2013-48367-P and CGL2016-80609-R) and the KMA (Korea Meteorological Administration) (grant no. 1365002970/KMA2018-00721). RM acknowledges support from MINECO (grant no. FPI BES-2014-069430) for conducting his PhD. AN acknowledges support from MINECO (grant no. FPU 13/02798) for carrying out his PhD.

References

[1] P. Malakar, V. Saxena, T. George et al., "Performance evaluation and optimization of nested high resolution weather simulations," in *Euro-Par 2012 Parallel Processing*, C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, Eds., pp. 805–817, Springer, Berlin, Germany, 2012.

[2] Z. Christidis, "Performance and scaling of WRF on three different parallel supercomputers," in *High Performance Computing*, J. M. Kunkel and T. Ludwig, Eds., pp. 514–528, Springer, Berlin, Germany, 2015.

[3] P. Johnsen, M. Straka, M. Shapiro, A. Norton, and T. Galarneau, "Petascale WRF simulation of hurricane sandy: deployment of NCSA's cray XE6 blue waters," in *Proceedings of the 2013 SC-International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–7, Denver, CO, USA, November 2013.

[4] G. Shainer, T. Liu, J. Michalakes et al., "Weather research and forecast (WRF) model performance and profiling analysis on advanced multi-core HPC clusters," in *Proceedings of the 10th LCI International Conference on High-Performance Clustered Computing*, Boulder, CO, USA, 2009.

[5] C. Kruse, D. Del Vento, R. Montuoro, M. Lubin, and S. McMillan, "Evaluation of WRF scaling to several thousand cores on the yellowstone supercomputer," in *Proceedings of the Front Range Consortium for Research Computing 2013*, Boulder, CO, USA, August 2013.

[6] J. Lagravière, P. H. Ha, and X. Cai, "Evaluation of the power efficiency of UPC, OpenMP and MPI. Is PGAS ready for the challenge of energy efficiency? A study with the NAS benchmark," Tech. Rep., The Arctic University of Norway, Tromsø, Norway, 2015.

[7] A. Igumenov and J. Žilinskas, "Electrical energy aware parallel computing with MPI and CUDA," in *Proceedings of the 2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, pp. 531–536, IEEE, Compiegne, France, October 2013.

[8] A. Igumenov and J. Žilinskas, "Power consumption optimization with parallel computing," *Jaunųjų Mokslininkų Darbai*, vol. 4, pp. 119–122, 2011.

[9] M. Aqib and F. F. Fouz, "The effect of parallel programming languages on the performance and energy consumption of HPC applications," *International Journal of Advanced Computer Science and Applications*, vol. 7, no. 2, 2016.

[10] H. Morrison and J. A. Milbrandt, "Parameterization of cloud microphysics based on the prediction of bulk ice particle properties—part I: scheme description and idealized tests," *Journal of the Atmospheric Sciences*, vol. 72, no. 1, pp. 287–311, 2015.

[11] E. J. Mlawer, S. J. Taubman, P. D. Brown, M. J. Iacono, and S. A. Clough, "Radiative transfer for inhomogeneous atmospheres: RRTM, a validated correlated-k model for the longwave," *Journal of Geophysical Research: Atmospheres*, vol. 102, no. D14, pp. 16663–16682, 1997.

[12] G.-Y. Niu, Z.-L. Yang, K. E. Mitchell et al., "The community Noah land surface model with multiparameterization options (Noah-MP): 1—model description and evaluation with local-scale measurements," *Journal of Geophysical Research: Atmospheres*, vol. 116, no. D12, 2011.

[13] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Vol. 400, Benjamin Cummings, San Francisco, CA, USA, 1994.