Research Article

Accelerating Spark-Based Applications with MPI and OpenACC

Saeed Alshahrani ¹, **Waleed Al Shehri** ², **Jameel Almalki** ², **Ahmed M. Alghamdi** ³,
and **Abdullah M. Alammari** ⁴

¹Department of Information Systems, College Computing and Information Technology, Shaqra University, Shaqra, Saudi Arabia

²Department of Computer Science, College of Computer in Al-Leith, Umm Al-Qura University, Makkah, Saudi Arabia

³Department of Software Engineering, College of Computer Science and Engineering, University of Jeddah, Jeddah, Saudi Arabia

⁴Curriculums and Teaching Methods Department, Faculty of Education, Umm Al-Qura University, Makkah, Saudi Arabia

Correspondence should be addressed to Waleed Al Shehri; washehri@uqu.edu.sa

Received 25 March 2021; Accepted 10 July 2021; Published 21 July 2021

Academic Editor: Adil Mehmood Khan

Copyright © 2021 Saeed Alshahrani et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The amount of data produced in scientific and commercial fields is growing dramatically. Correspondingly, big data technologies, such as Hadoop and Spark, have emerged to tackle the challenges of collecting, processing, and storing such large-scale data. Unfortunately, big data applications usually have performance issues and do not fully exploit a hardware infrastructure. One reason is that applications are developed using high-level programming languages that do not provide low-level system control in terms of performance of highly parallel programming models like message passing interface (MPI). Moreover, big data is considered a barrier of parallel programming models or accelerators (e.g., CUDA and OpenCL). Therefore, the aim of this study is to investigate how the performance of big data applications can be enhanced without sacrificing the power consumption of a hardware infrastructure. A Hybrid Spark MPI OpenACC (HSMO) system is proposed for integrating Spark as a big data programming model, with MPI and OpenACC as parallel programming models. Such integration brings together the advantages of each programming model and provides greater effectiveness. To enhance performance without sacrificing power consumption, the integration approach needs to exploit the hardware infrastructure in an intelligent manner. For achieving this performance enhancement, a mapping technique is proposed that is built based on the application's virtual topology as well as the physical topology of the undelaying resources. To the best of our knowledge, there is no existing method in big data applications related to utilizing graphics processing units (GPUs), which are now an essential part of high-performance computing (HPC) as a powerful resource for fast computation.

1. Introduction

Recently, there has been drastic growth in the volume of data generated by many different scientific fields for all big data characteristics. Big data can be characterized using different Vs: variety, volume, velocity, veracity, and value [1]. Many applications and systems in the real world depend on collecting, storing, and analyzing such large-scale data, and this trend is predicted to overgrow [2]. Big data environments, such as Hadoop, MapReduce, and Spark, have emerged for supporting parallelism in processing huge dataset volumes [3]. With a future path leading toward Exascale computing with increasing the number of high-performance CPUs and

GPUs in each compute node [4], resource and job management techniques can be considered a keystone of any scalable computing system to support a high degree of parallelism, control system processes, and impact overall performance and system efficiency [5].

Currently, Apache Spark is the largest and perhaps the most comprehensive open-source framework dealing with big data [6]. However, due to its use of the Java Virtual Machine (JVM), its performance is not as desired, and there is still much room for improvement. A high-level implementation of Spark is unaware of the underlying hardware and how to best utilize it, focusing more on the optimization of the virtual machine on which it operates.

Existing High Performance Computing (HPC) tools, such as MPI-based tools, are not designed to manage big data applications [7]. This is a missed opportunity to use the rich set of resources developed over time for HPC that are intended to deliver high performance. There is no existing method in big data applications using GPUs, which are now an essential part of HPC, as a powerful resource for fast computation. Traditionally, GPUs have been utilized to solve complex algorithms, but they can be an effective tool in big data applications and help solve the convergence challenge.

In our previous research [8], we surveyed HPC systems that should support big data for high performance and resource utilization. The studied factors include load balancing and data locality, job scheduling strategies, topology-awareness, and data decomposition. It was found that parallel programming models are dataset specific and it lacks generality. While much of the research problems regarding HPC in big data have been elaborated, complete solutions are still lacking.

In this paper, our proposed solution builds upon three systems: Spark, MPI, and OpenACC. The basic idea is to use a best-of-both-worlds approach: take the big data management offered by Spark, integrate it with the optimal computational processing offered by MPI, and further accelerate the GPUs process by using OpenACC. However, simply integrating the three systems alone is not enough. Spark is a high-level framework implemented in Java and uses the JVM for execution, whereas MPI and OpenACC require low-level programming languages and libraries, such as C, C++, or Fortran. Our proposed approach requires the establishment of a bridging process.

Such bridging alone is also not enough; however, a complete resource management mechanism needs to be established, which defines all modules within the new system and any potential optimization that can be achieved to enhance the system further. To test and verify the effectiveness of the new system, a benchmark was established that, for defined parameters, it must outperform Spark. A case study was chosen with challenging big data problems, for which both Spark and the new system and its variants are tested under strenuous circumstances. The results of the test case provide in-depth information on the effectiveness of the proposed approach.

Our paper has been structured in the following manner: Section 2 will highlight various state-of-the-art techniques used for big data applications. Our proposed hybrid system is explained in Section 3, and its software architecture and modules are discussed in Section 4. In Section 5, we present the system implementation and its evaluation based on the results of a case study along with a discussion of those results. Finally, our conclusions and future work will be discussed in Section 8.

2. Background and Related Work

A computing system that consists of multiple nodes of multiple processors as a part of one unit is known as high-performance computing (HPC). These machines are capable of rapidly processing large amounts of data or information

by using several clusters of processing units within a single resource unit. The core methodology used within the framework of these fast processing units is parallel computing [3]. Briefly, HPC is fabulous for its fast-processing capacity to analyze or process data and information.

In today's modern world, technologies like big data processing, artificial intelligence (AI), and the Internet of Things (IoT) are evolving and changing the spectrum of modern life. Moreover, in these times, data are considered vital for organizations, and it is expanding exceptionally. Technologies like HPC are in demand to support information processing in real time and achieve desired goals and targets [9].

To develop the HPC architecture, a network of servers is created to form clusters so that the collection of computers can work together as a single unit and process the data as fast as possible. In some cases, such as financial markets and stock exchanges, data processing needs to occur in real-time. Different algorithms and software programs are used concurrently within the architecture of the clusters. These clusters are networked together with an output unit where all the information is collected or stored [10]. There is no denying the fact that these HPC clusters seamlessly complete diverse tasks.

Within the HPC clusters, it is vital to synchronize each component's pace with other servers to get the maximum performance from the machine. For instance, the computing servers' processed data should be transferred into the storage components at the same pace as it is processed. If the storage component is not storing the data or information as fast as the clusters have processed it, then its HPC infrastructure would not be able to perform as fast as it should. Therefore, all the components within the network must perform at the same pace.

Typically, there are hundreds of computing servers in HPC clusters that are connected to form a network. Each server within the network is known as a node. These nodes within the network perform parallel computing to enhance the processing speed to deliver the output in minimum time.

In the following subsections, several state-of-the-art mechanisms and techniques used for big data applications have been highlighted. Although the topic itself is vast, it attempts to cover many techniques and aspects to support big data applications.

2.1. Topology Awareness. The expanding complexity of computing platforms fueled the necessity of developers and programmers understanding hardware organization and adjusting their applications accordingly [11]. As a component of the overall optimization process, there is a great need to visualize a hardware platform model. Hardware locality (HWLOC) is the most famous programming tool used for revealing a consistent perspective on central processing unit (CPU) and memory topology. The paper published in [12] demonstrates how HWLOC accesses these computing assets by joining the I/O structure and providing ways to handle different hubs.

More recently, a particular type of application has been targeted by some work, such as MapReduce-based

applications. Topology-Aware Resource Adaptation (TARA) [13] uses an application description for resource allocation purposes. However, the work is designed for a particular set of applications and is not intended to address other hardware details.

Most open-source and proprietary resource and job management systems (RJMS) consider this kind of feature by using their underlying infrastructure topology characteristics. However, they have failed to consider the importance of application behavior while allocating resources. An approach named HTCondor, formerly Condor [14], is proposed to take advantage of the matchmaking method, which allows for matching the applications' requirements with hardware resource availability. However, this matchmaking method does not consider application behavior, and HTCondor applies to both connected workstations and clusters. In [15], an open-source topology-aware hierarchical unstructured mesh partitioning and load-balancing tool is proposed and named as TreePart tool. The framework can detect and build a hierarchical MPI topology for the underlying hardware at runtime. This topology information can help in partitioning load-balancing between shared and distributed parallel algorithms intelligently.

2.2. Load Balancing and Data Locality. Load balancing techniques such as work stealing are used for distributed task scheduling systems, where tasks are transferred from heavily loaded schedulers and assigned to the idle ones. However, the work-stealing technique could lead to poor data locality in data-intensive applications because tasks' execution depends on extensive data processing, resulting in data-transferring overhead.

The research in [16] proposed two methods for distributing and optimizing load balancing for processing big data. The proposed method aims to reduce idle node time, which minimizes task execution time. Results show a significant decrease in processing time. One attempt to enhance the work stealing is by organizing dedicated and shared queues [17] based on their location and data size. Computing multiple tasks could be distributed via techniques such as the MATRIX task scheduler. The Distributed Key-Value Store (DKVS) organizes the metadata and other data locality and task dependency elements. Research showed that the data-aware work-stealing technique exhibits adequate performance.

A fast and light-weight task execution framework (FALKON) [18] is a centralized task scheduler for supporting many-task computing (MTC) applications. FALKON adopts a data diffusion approach for scheduling data-intensive workloads via DKVS [17]. However, FALKON suffers from uncertain task execution times resulting in poor load balancing due to its hierarchical implementation. It is also prone to the issue of scaling in Petascale systems. Data diffusion acquires computing resources and storage dynamically. The process replicates data based on demand to schedule computations close to the data. In contrast to the distributed key-value store [17], FALKON suffers from poor scalability due to deploying a centralized index server for storing metadata.

Mesos platform shares resources among many cluster computing frameworks for scheduling tasks [19]. Therefore, it allows frameworks to achieve data locality via reading the data stored on each machine. When delay scheduling systems are deployed, it waits for a limited time to get data storing nodes. Therefore, the approach generates a significant waiting time for any task that is scheduled on time, especially for larger datasets.

The flexible and dynamic framework of Quincy [20] is a distributed concurrent job scheduling that adopts fine-grain resource sharing. However, the Quincy model takes substantial time to find the best graph structure solution based on both data locality and load balancing.

MPI-based scientific applications generally prefer a computer-centric architecture that runs on several nodes using a file system. However, the unpredictable growth of scientific data threatens the efficiency and high performance of these MPI-based applications. Data locality MPI (DL-MPI) was proposed to solve this problem by acquiring data information through a data locality API and a scheduling algorithm that tasks each node based on its capacity [21]. Similarly, the data locality API assesses the amount of unprocessed local data; thereby, the scheduling algorithm allocates processing to compute nodes. However, the algorithm scales down on a large number of nodes, and its implementation requires sophistication. It would seem that the data movement overhead obstructs the scaling of the system in its baseline.

It was found that the locality-aware scheduling and data distribution on many-core processors have not been discussed adequately in the literature [8]. It was claimed [8] that leveraging the locality awareness reduces high-performance parallel-systems energy waste via optimized power-efficient techniques. The reference explains two power efficiency symbiotic techniques, which are the intralocality power-driven power optimization and the interprocess locality-driven power optimizations. The intralocality technique has the flexibility and control to assign processor frequency and manage sleep states within the same process sequence. On the other hand, the interprocess technique uses the coscheduling and coplacement of jobs in a varied set of threads from a diverse set of processes executed simultaneously on an HPC cluster. Coscheduling and coplacement group threads and processes, based on the similarity of their affinity patterns and symbiosis, reduce as much as 25% in energy consumption. However, energy reduction efficiency depends heavily on the correct identification of computer memory functions and the CPU capacity, which should be used in perfect harmony to be more beneficial. Therefore, it is prudent to analyze the results of these techniques in a systematic framework.

2.3. Decomposition Techniques. While software application parallelization provides efficiency, it could be error inclined and exceptionally unoptimistic, specifically in terms of the data decomposition task [22, 23]. Therefore, careful attention should be taken when choosing a parallelization methodology that provides consequent conveyance of data

or task processing running on available cores [24, 25]. While several studies were conducted to understand software usage, debugging, testing, project organization, and tuning [26, 27], data decomposition research is a significant challenge in a parallel programming environment [28]. Up to this point, there has been insufficient empirical research in the HPC field that has focused on these issues.

2.4. Hybrid-Programming Models. Several studies showed that MPI-based applications outperform Spark or Hadoop-based big data applications by order of magnitude or more for several applications, e.g., k-nearest neighbors and support vector machines [29], k-means [30], graph analytics [31, 32], and large-scale matrix factorizations [7]. It was found that compute load was the primary performance bottleneck in some Spark applications, precisely the duration of serialization and deserialization [33].

Big data programming models can be improved by combining them with parallel programming models like MPI. This approach can be seen in [34], which showed how to enable the Spark environment using the MPI libraries. Even though this technique signposts remarkable speedups, it must use shared memory and other overheads, potentially drawbacks. Several MPI-based communication primitives were used to resolve the performance problem of HPC over big data, for example, replacing MapReduce communicators in Hadoop (Jetty) with an MPI derivative (DataMPI) [35]. However, the approach is not a drop-in replacement for Hadoop, and existing modules need to be recoded to use DataMPI. While machine learning libraries can interoperate with Spark, such as H₂O.ai [36] and deeplearning4j [37], they are limited. Such Java-based approaches do not provide the direct integration of existing native-code MPI-based libraries. One solution was spark with accelerated tasks (SWAT) [38] that creates OpenCL code from JVM code at runtime to improve Spark performance. As opposed to our work, SWAT is limited to single-node optimizations; it does not have access to the communication improvements made available through MPI. Alchemist [39] interfaces between MPI libraries and Spark and observes that such interfacing speeds up linear algebra routines. Improved performance comes from the comparative overhead of moving data over the network between Spark nodes and Alchemist, and there is still a benefit from working in the Spark environment.

3. The Proposed Hybrid System

Spark alone has serious deficiencies when handling computationally intensive tasks with big data, although it is excellent at managing big data itself. Alternatively, MPI-based implementations, due to their native approach, are considerably more computationally performant. However, MPI-based applications lack the efficiency to handle big data. Further, to enhance the computation process, GPU processors can be utilized if available. GPU programming standards, such as OpenACC, which support multivendor devices, are also low-level implementations supported in either C, C++, or Fortran and are compatible with MPI [40].

However, with the use of extra resources and GPUs, more energy is required. This creates a burden on the system, causing heating issues that can be a determinant in long-term computation.

Therefore, the two main factors that were considered while formulating the proposed technique are the following:

- (1) Combining the benefits of Spark data handling with the computationally intensive nature of MPI and OpenACC
- (2) Keeping power usage to a minimum by employing optimal integration techniques

3.1. The Simple Integration. In this section, the process of minimum integration is discussed while maintaining the two key principles explained previously. The three technologies—Spark, MPI, and OpenACC—will be integrated together to form the Hybrid Spark MPI OpenACC system (HSMO). Spark's core is based on the resilient distributed dataset (RDD), a special data structure that has the ability to handle and compute data in parallel. RDD is the best solution for managing big data. However, RDDs are created and managed by high-level frameworks that require the use of JVM [41]. There is currently no stable interface for low-level programming languages or frameworks to manage RDDs [39]. This obstacle creates a hurdle for the transfer of data for processing to MPI- and OpenACC-based implementation. Hence, any implementation will require a bridge between Spark and low-level implementations, in which Spark will handle the data and call upon MPI-based workers to execute the desired computations of the data.

Nevertheless, any transfer of data after initial partitioning will conflict with the second principle of integration and cause a spike in power usage. Hence, the data, once partitioned, must not leave the nodes at which it resides.

To accomplish this, repartitioning is disabled, and workers are called on specific nodes. As the technique evolves, further steps are taken to ensure data movement is curtailed as much as possible to reduce power consumption. The previous discussion created a rough sketch of the components of the system that are to be created. Data are to be gathered in Spark, that is the input of the system and the point receiving user interaction. Likewise, as a high-level system, it is logical to make it the interface for the entire system. It will behave as the controller of the system, responsible for input, output, task distribution, and system management. Workers, which will do the bulk of the computations, will be implemented in MPI + OpenACC. Henceforth, they will be referred to as MPI-Workers. The job of the MPI-Worker is to process data and return a result to the controller. In Simple Integration, the goal is to combine the three subsystems to formulate the hybrid system. However, simple integration is not sufficient, and additional steps are required in certain cases and at certain points to enhance the performance of the newly formed hybrid system. Figures 1 and 2 show an abstract-level view of the hybrid system and our simple integration of Spark, MPI, and OpenACC.

FIGURE 1: Abstract-level view of the hybrid system.

FIGURE 2: Simple integration of Spark, MPI, and OpenACC.

3.2. *The Extended Integration.* Once the basic integration of subsystems has been accomplished, it can be extended for the optimal performance of the parameters defined in the next chapter. The extension is based on three factors:

- (1) Physical topology generator.
- (2) The virtual topology generator of the application (e.g., big data application).
- (3) Mapping technique based on virtual and physical topologies.

In simple integration, the goal is to make all subsystems work together, building upon the same infrastructure. Additional components are added to the simple integration; these are as follows.

3.2.1. *Virtual Topology Generator.* A low-level component generates the topology within the MPI-worker. The grid generation can be accomplished by analyzing the application behavior to detect the number of processes/threads as well as the communication frequency between processes/threads for better mapping and task management. This can be accomplished by tracking every MPI sent and received and the messaging frequency between every two processes.

3.2.2. *Physical Topology Generator and Core Mapper.* A low-level component is responsible for two things: first, retrieve the core information for the entire system and second, bind the process/threads as per the virtual topology.

Figures 3 and 4 show the extended integration, in which C1, C2, and Cm1 are system cores, and g1, g2, and gp are the GPUs of the system. There are two forms of extended integration where the first one is extended integration with common GPU that many CPUs are sharing the same group of GPUs as shown in Figure 3. This can have less performance with less power consumption. In contrast, Figure 4 shows groups of CPUs sharing dedicated GPUs, and this

may enhance the performance with more power consumption.

3.3. *GPU Optimization.* In GPU optimization, regular cores are assigned for data handling, and partition numbers are increased, making it a function of the number of cores available across all GPUs in a combined system. While the bulk of computation is handled by GPUs, if any cores are available, they still handle processing as per the extended integration rules, because GPU integration is built upon extended integration and contains all features and characteristics of simple and extended integration. Figure 5 shows the GPU optimization variant.

4. Software Architecture and Modules of the Proposed Techniques

The high-level architecture of our hybrid system is shown in Figure 6; it is designed based on different software modules, resources, and interconnectivity components to create the overall system.

The data are provided by the user in Spark to the tri-model interface with the help of the metadata about the stored big data. The metadata could include useful information, such as the location, size, and type of the stored data. This can play a critical role in supporting big data computing in HPC, as the massive volume of data will be stored in many different nodes. The task here is distributed between Spark and MPI-workers, as the Spark source code is responsible for the management of the system, while the MPI-worker code processes the data. The Spark controller creates the partitions of the data based on the available HPC system resources in terms of regular CPU and GPU cores. In order to achieve this, it communicates with the HPC system and obtains the necessary system information. Based on this, partitions are created, and MPI-workers are called.

FIGURE 3: Extended integration with separate GPU grid.

FIGURE 4: Extended integration with common GPU grid.

The MPI-worker receives the data allocated to it for processing. By receiving this and analyzing the worker behavior, the virtual topology of the worker can be generated.

Once the topology is generated, threads are created accordingly and bound to cores in the nodes in which the partitions reside. In order to do so, the physical topology of the nodes containing a bitmap and logical core tree is obtained. In the following explanation, each module, along with its algorithm, is discussed.

4.1. Spark Controller Module. The Spark controller module is the nerve center of the proposed technique; from here, the entire system is controlled and managed in terms of partitioning the RDD data and gathering the system resources. The first step is for the system user to input and/or choose the data. This is supported by metadata that helps the user explore and choose the required data. Then, this increases the effectiveness and quality of the system service. In addition, caches, along with system cores and GPUs, need to be fetched for partitioning purposes to maximize GPU

FIGURE 5: GPU optimization variant.

FIGURE 6: High-level architecture of the proposed system.

utilization. In order to do so, the hardware infrastructure information is fetched, and, from there, the GPU cores are obtained. The number of partitions is then made dependent on the number of GPU cores. The more GPU cores there are, the more partitions there will be.

Regular system cores are responsible for managing the partitions. If any system cores are left or idle, they may be used for processing in MPI-workers. The last step is result collection, during which MPI-workers send data back to the Spark controller.

In Algorithm 1, we show the Spark controller module mechanism.

4.2. Virtual Topology Generator Module. This module is responsible for generating the virtual topology of the MPI workers. The generation parameters are mainly dependent on the analysis of the MPI-worker source code to detect the number of processes/threads as well as the

communication frequency between them in order to formalize the grid's size and the ordering of the processes/threads on the grid.

A grid structure is created for threads intended to be created with possible messaging among neighbors. The reordering option allows for reordering the topology during execution. Such an option can be used based on the communication frequency between processes, so the highest communication processes should be close together and, accordingly, map the neighboring processes on physical resources close to each other. This period option allows cycling at the end of the grid. This is set to true, as it is intended to introduce new threads onto cores from the ready queue as soon as any slot becomes available.

Once options are set, the `MPI_Cart_Create` [42] method is called, which creates the new topology and returns the communicator for the new topology. The communicator holds the information regarding the structure of the

INPUT: Data.

OUTPUT: Result.

START

- (1) Let M hold R 's metadata for reference.
- (2) Let R be the RDD holding data, input by the system user.
- (3) Let H be the set of system resources containing Cores C , Memory r , Cache Ca , GPU cores Gc , and Nodes N .
- (4) Let P_i be a partition of R , where $i = 1, 2, \dots, Gc$.
- (5) Call worker on Node N_p for each P_i , where N_p is the node holding data for partition P_i .
 - (a) Collect partial results R_i for P_i .
 - (b) Combine R_i to form complete result R as per the application requirement.
- (6) Out Result.

STOP

ALGORITHM 1: Spark controller.

topology and the communicator is used in subsequent modules.

Our virtual topology generator module is presented in Algorithm 2, and the data structure example of the virtual topology is shown in Figure 7.

4.3. MPI + OpenACC Implementer Module. The purpose of this MPI-worker module is to implement application-specific tasks and generate partial results. This module is the core workhorse in which all the computations take place. For running on GPU, the first step is to check whether OpenACC libraries are available on the system. If so—and a compatible GPU is available—it can be initialized. OpenACC is a directive-based implementation, the benefit of which is that the absence of either OpenACC or GPU allows for normal/alternate execution to proceed smoothly.

However, within a GPU, code is parallelized even further. This process is automated and not controlled within the scope of the proposed system. This obstacle is presented as future work.

4.4. Physical Topology Tree Gatherer Module. An important step within the implementation is core binding. In order to bind threads to cores, the physical topology of the cores is required, and, as explained earlier, workers are called on the respective nodes of the partition. Each node may have its unique hardware topology. In order to retrieve this information, the HWLOC (Hardware locality) library is used, which provides two main items: topology and bitmap.

4.4.1. Topology. It is the entire system architecture of the node, core block, internal and external caches, memory, and buses. Topology is needed for referencing when binding to cores.

4.4.2. Bitmap. This is specific to cores. It contains the core layout, the way they are arranged in blocks, and their number.

This information can help deduce the ability of a node to handle simultaneous threads. Hardware locality provides

methods for extracting the information, which is then used for the mapping of threads.

Algorithm 3 shows our physical topology tree gatherer approach.

4.5. Mapper Module. The purpose of this module is to bind threads to cores. The information provided from the “Physical Topology Tree Gatherer Module” and “Virtual Topology Generator Module” is the input. The “implementer” module provides the thread in the simple and extended cases. In the GPU optimization case, core-binding is not required.

From the bitmap, a single-core reference is extracted and, again, the HWLOC provides the necessary method. An active thread is bound to a core by following the principle of virtual topology, in which neighboring threads, by following the principle of virtual topology, must be bound on neighboring cores.

In Algorithm 4, we show our core mapper mechanism.

5. Implementation and Evaluation

This section discusses the system implementation and its evaluation based on the results of a case study along with a discussion of those results. The details of the testing environment are also presented along with the testbed setup.

Additionally, the parameters for which the results were evaluated are discussed. Each technique is then evaluated based on those parameters. Moreover, the benefits and shortcomings of the techniques are presented. Finally, a comparative study is presented with the closest competitors of the technique.

5.1. PageRank Algorithm as a Case Study. PageRank Algorithm was used as a test case for the proposed system. PageRank is an iterative algorithm [16] defined by the following:

$$\mathbf{PR}^{t+1}(\mathbf{v}) = \mathbf{r} + (1 - \mathbf{r}) \times \sum_{\mathbf{u} | (\mathbf{u}, \mathbf{v}) \in E} \frac{\mathbf{PR}^t(\mathbf{u})}{\text{degree}(\mathbf{u})}, \quad (1)$$

where “ E ” is the total number of nodes and “ $\mathbf{u}$ ” is the summation variable 1, 2, 3, ..., E . “ $\mathbf{v}$ ” is the function of the

INPUT: MPI-worker processes.
OUTPUT: virtual topology for MPI-worker.
START
 (1) Let Q be no. of Processes/Threads obtained using `MPI_comm_size`.
 (2) Let Dim be the dimensions of the Grid by calling the generator method.
 (3) Set options for virtual topology Reorder & Period True or False.
 (4) Call method `mpi_cart_create` with generated dimensions, and set options.
 (5) Out new communicator for virtual topology.
STOP

ALGORITHM 2: Virtual topology generator.

Partition #	X	Y	Is GPU available?
0	0	9	0
0	0	10	0
0	0	1	1
0	0	2	0
0	0	7	0
0	0	3	0
0	0	8	0
0	0	5	0
0	0	4	0
0	0	6	0
0	0	0	0

FIGURE 7: Example of virtual topology data structure.

equation to be calculated, since its iterative equation “ r ” is a damping factor equal to 0.85, and it is based on PageRank theory, in which the surfers who click on random links will stop at the end of clicking.

The equation can be parallelized and is widely used in big data case studies. The equation is implemented using fixed iterations. The implementation algorithm is provided in Figure 8.

5.2. Testing Environment. Table 1 and Figure 9 show the specifications of the target machine used for developing the HSMO prototype. Figure 9 depicts the hierarchical view of the hardware specifications that are extracted by `hwloc's lstopo` command.

5.3. Methodology of the Experiments. In order to test the PageRank Algorithm implementation, a large dataset is required. PageRank's input links between nodes. Each node can represent a web page, social media account, or any similar scenario. The link between the two nodes can be represented by many means. For instance, they can be represented as a single line, in which each line has two-node entries.

For this purpose, the dataset requires millions of such links to produce enough load in order to satisfy various test cases for the system against the defined parameters.

The specifications of the dataset used for experimentation are listed in Table 2 [43]. In order to test the system, the dataset was sliced into various ranges, and readings were taken for parameters explained in a later section. The

PageRank algorithm allows for such slicing due to the nature of the algorithm.

5.4. Experiments and Benchmarks. There are two main parameters for which experiments are conducted: processing time and power consumption. These parameters are discussed below. The benchmark for both parameters is that they must improve on a Spark-only implementation of PageRank for all defined inputs.

5.4.1. Processing Time. In order to measure processing time, elapsed time “ t_e ” is calculated, which is defined as the duration between an input being distributed among workers and the receipt of the overall results from all partitions. Equation (2) defines the elapsed time:

$$t_e = t_f - t_0, \quad (2)$$

where “ t_0 ” is the initial time recorded before the input is submitted to the workers and “ t_f ” is the time recorded after the results are received.

Elapsed time “ t_e ” was recorded for all techniques and is the key in determining the optimally beneficial choice of a technique for a data range. A technique is said to be more efficient for a dataset range if its elapsed time is less than that of other techniques.

5.4.2. Power Consumption. Power consumption is defined as the amount of power, in Watts, used over the execution time. The value is measured using the powerstat tool [44]. As per principle 2 of the proposed technique, power consumption is to be kept to a minimum. Hence, this parameter was essential for a complete analysis of the techniques.

Powerstat gives values in steps, while execution proceeds. The measurements need to be observed for idle state and spiked state when a task is submitted to the Apache Framework. Further, each step needs to be considered in the final calculation of the overall power consumption of a technique. Equation (3) gives the power consumption formula:

$$Pt = \sum_{k=1}^n (Pn - Pi), \quad (3)$$

where “ Pt ” is the total power consumption, n is the number of steps in powerstat, “ Pi ” is the idle state value, and “ P_k ” is the step value of the power.

INPUT: HPC node hardware infrastructure.

OUTPUT: Bitmap, logical core tree.

START

- (1) Let NCW be the HPC node for worker W.
- (2) Let TNCW be the hardware topology for NCW obtained using hwloc_topology_load.
- (3) Let BNCW be the bitmap for cores obtained from TNCW.
- (4) Let LNCW be the logical core tree obtained from BNCW.
- (5) OUT BNCW, LNCW.

STOP

ALGORITHM 3: Physical topology tree gatherer.

INPUT: bitmap, logical core tree, MPI thread, T , T_d .

OUTPUT: partial result.

START

- (1) Let T_d hold the depth information based on the system topology T containing logical core information using method, hwloc_get_type_or_below_depth.
- (2) Let PU be the logical core obtained from T & T_d using method hwloc_get_obj_by_depth.
- (3) Let PU_Cpuset hold PU core's cpuset.
- (4) Set PU_Cpuset to get only one logical processor in case of SMT/hyperthreading using method hwloc_bitmap_singlify.
- (5) Bind current mpi_thread to core using method hwloc_set_bind.
 - (a) If bind fails, report error.
 - (b) Otherwise calculate and return result R .
- (6) Out R .

STOP

ALGORITHM 4: Core mapper.

```
var PR = Array.fill (n) (1. 0)
val oldPR = Array.fill (n) (1. 0)
for (iter <- 0 until numIter) {
  swap (oldPR, PR)
  for (i <- 0 until n) {
    PR[i] = alpha + (1 - alpha) * inNbrs[i].map (j=> oldPR[j] / outDeg[j]).sum
  }
}
```

FIGURE 8: The PageRank algorithm implementation.

6. Discussion

Figure 10 shows the processing time comparison for the HSMO Simple and Extended variants with the Spark implementation. The first point to note is that, for the same set of resources, HSMO Simple is able to extend the range of data the system was able to handle. With HSMO Extended, the range was even further extended. It is noticeable that, as the data increased, HSMO Simple and Extended outperformed Spark. However, it is also observed that, for the smaller dataset, HSMO Simple and Extended performed almost identically, and, on some occasions, HSMO Simple performed better than HSMO Extended for smaller datasets. This can be attributed to the HSMO Extended version spending time creating a virtual topology and core

binding, whereas the simple variant used that time on computation. Only when the dataset was small could this overhead in HSMO Extended be noticed. However, these steps are necessary, since, as the dataset grows, system performance improves.

Figure 11 illustrates that power consumption is reduced in both the HSMO Simple and Extended variants in comparison with Spark, as the data size is increased. The abnormal spike in Spark's chart is due to the data movement that occurs due to the computational process being non-uniform in nature at any given stage.

The processing time comparison of HSMO Extended for various numbers of cores is shown in Figure 12. It can be seen that increasing the number of cores improves the processing time of the system, which emphasizes that the system is scalable. This is simply because the MPI-workers have more resources at their disposal.

The power consumption value comparison of HSMO Extended for various numbers of cores is displayed in Figure 13. It can be observed that, as more cores are introduced, less power is consumed. This is attributed to the fact that results are quickly calculated, resulting in lower runtime, and, hence, less power was consumed even though more resources were available for disposal. The abnormal spike in the middle of the chart can be attributed to the time required to create the virtual topology, so more time leads to more power usage.

Figure 14 shows the time and power statistics for HSMO GPU optimized. It can be observed that HSMO GPU optimized is able to outperform the Extended variant quite significantly in

TABLE 1: System specification.

OS	Ubuntu 18.04.1
Memory	Description: system memory Physical id: 3 b Slot: system board or motherboard Size: 16 GiB
Cache	*-cache:0 Description: L1 cache-size: 384 KiB *-cache:1 Description: L2 cache-size: 1536 KiB *-cache:2 Description: L3 cache-size: 9 MiB
CPU	Description: CPU-12 cores-product: Intel(R) Core (TM) i7-8750H CPU @ 2.20 GHz
GPU	NVIDIA GeForce GTX 1060

FIGURE 9: Hardware information of the target machine.

TABLE 2: Dataset specification.

Dataset property	Characteristics
Dataset nature	Social circles: Twitter
Nodes	81,306
Edges	1,768,149

terms of both time and power. This can be attributed to the fact that the capabilities of GPUS outperform CPUs performance significantly, and this clarifies that our system exploits such capabilities to achieve research objectives successfully.

It can also be realized from Figures 15 and 16 that HSMO GPU optimized is able to handle large datasets, with significant improvements in both time and power consumption. From Figure 15, we noticed that even after increasing the data size by

FIGURE 10: Processing time comparison for different approaches.

FIGURE 11: Power consumption comparison for different approaches.

FIGURE 12: HSMO_Extended/time comparison for different number of cores.

FIGURE 13: HSMO_Extended/power consumption comparison for different number of cores.

FIGURE 14: HSMO_Extended/GPU optimized—time-power consumption comparison.

FIGURE 15: Processing time comparison for maximum data size.

FIGURE 16: Power consumption comparison for maximum data size.

more than 20.000% (exactly 20.461%), the processing time has decreased by 42%, which improved the system performance. As we see in Figure 16, the power consumption has decreased by 69% even with the huge increase in the data size as we mentioned previously by more than 20,000%.

7. Comparative Study

The novelty of the proposed technique in this research can be seen in the tri-model that involves Spark, MPI, and OpenACC. In addition, and as far as we know, there is no work yet that employs this tri-model and considers both big data application behavior and the resource management of the hardware infrastructure. Furthermore, most of the related work focuses on enhancing performance—whether explicitly or implicitly. However, such enhancement is usually accompanied by an increase in power consumption. Targeting performance enhancement while controlling power consumption is a key challenge. This challenge arises clearly in the field of big data and HPC environments as the volume of data is enormous with high velocity and various varieties, and the underlying infrastructure has hundreds of different resources. Accordingly, it is important to consider parameters in both domains. Particularly the convergence of these two domains is going to be more necessary in many different scientific fields.

The two competing approaches, “bridging the gap between HPC and big data frameworks” [34] and Alchemist [39] integrate Spark with MPI. The first approach focuses on shared memory and optimizing Spark. Alchemist also fails to address the data locality question. Apart from not including OpenACC, it solely focuses on integration and not optimizing the integration. There is no mention of how MPI handles computations, a fault shared with [34].

The approach presented in “bridging the gap between HPC and big data frameworks” is more of an effort to optimize existing techniques rather than providing a new technique. It also fails to present how the actual integration module works. Alchemist

also suffers from data transfer overhead when data are on different nodes, indicating a lack of utilization of the Spark framework, as Spark is able to manage initial data transfers.

In comparison, HSMO placed an additional accelerant in the form of a GPU and presented variants for optimizing its use and also addresses the question of data locality. Furthermore, HSMO focuses on optimizing the hybrid system instead of the high-level system. This shows an understanding of the proven fact that high-level systems cannot compete in computational performance when compared with low-level systems. Moreover, HSMO has well-defined roles for each subsystem. User interactivity is not compromised, as the high-level module controls all user interactions. Moreover, performance is not compromised, as the low-level module controls all computational processes.

Comparing HSMO with Alchemist [34, 39] yields that it is a more dynamic technique that addresses practical questions of integration, specifically for MPI, which, by itself, lacks the ability to handle big data.

8. Conclusion and Future Work

This research demonstrated the performance gap challenge for big data applications on HPC clusters. The big data revolution is still moving at a high rate of growth in terms of the volume, velocity, and variety of data, which makes dealing with such data increasingly difficult. Accordingly, big data technologies keep emerging to face this drastic growth and overcome the difficulty of storing, transferring, and analyzing such enormous data to extract value and desired benefits.

In the same context, HPC, with its hardware infrastructure combining hundreds of thousands of computer nodes, packages, and parallel programming models, is a very attractive environment and a fertile field for supporting big data computing as well as many other types of research and experiments.

The convergence of HPC and big data is inevitable, which makes its progress subject to many accompanying

challenges, especially in the field of resource and job management and their effect on performance and power consumption.

Usually, there is a performance issue when deploying big data platforms on HPC clusters resulting from the underutilization of the capabilities afforded by HPC. This lack of performance is due to the difference between the two environments in terms of architectural design and the level of programming models that each support. The architectural design of big data platforms differs from that of HPC, which makes resource management and job scheduling very difficult. Furthermore, big data technologies are written in high-level programming languages that do not effectively support parallelism when compared with the lower-level programming languages supported by HPC environments. The large volume of big data may also hinder the application of a high level of parallelism when employing parallel programming models. As a result, big data needs to be tamed and optimized to maximize the potential from using HPC.

This was the focus of this paper, and it is achieved by developing the Hybrid Spark MPI OpenACC (HSMO) System that employs a novel resource management and mapping technique to stimulate high parallelism. HSMO takes into account decomposition and data locality strategies to schedule tasks according to the capabilities and availabilities of hardware infrastructure to enhance the performance while also controlling power consumption. Our future work can be summarized as the following:

- (1) GPU internally manages its resources—mainly cores. A mechanism should be developed, allowing access to those resources for an overall HSMO system.
- (2) When collecting results, data movement is required; this creates power consumption, and an alternate method may be investigated either to minimize the power consumption or to obtain alternative results.
- (3) Direct access to RDD partitions from MPI workers is required; this will entail the following:
The integrity of the original data for future reference.
A referencing mechanism within MPI-workers for threads. If achieved, this will further enhance both power consumption and processing time measures.
- (4) Currently, when data need to be transferred to workers, a copy is made in the system even, as elaborated in the discussion of the proposed technique, if that data do not leave the node of the partition. This is due to the limitation of a stable interface between the HDFS File system, which is the backbone of RDDs, and low-level programming languages such as C, C++, or Fortran.
- (5) A copy creates a burden on the system, eats up resources that may be utilized for computation, or extends the data range of the available HPC system.
- (6) The metadata extraction in the Spark controller can be extensive and linked with MPI-workers for better user interactivity.
- (7) HSMO needs to be tested on HPC clusters; this includes clusters of various configurations.

Data Availability

The Social circles: Twitter data used to support the findings of this study have been deposited in the Stanford Network Analysis Project repository (<https://doi.org/10.17616/R3XP7Q>).

Conflicts of Interest

The authors declare that they have no conflicts of interest to report regarding the present study.

References

- [1] S. Bergamaschi, C. Cavazzoni, A. Curioni, and G. Fox, "New opportunities in high performance data analytics (HPDA) and high performance computing (HPC)," in *Proceedings of 2014 International Conference on High Performance Computing & Simulation (HPCS)*, Bologna, Italy, July 2014.
- [2] K. M. Tolle, D. S. W. Tansley, and A. J. G. Hey, "The fourth paradigm: data-intensive scientific discovery [point of view]," *Proceedings of the IEEE*, vol. 99, no. 8, pp. 1334–1337, 2011.
- [3] S. Fiore, M. Bakhouya, and W. W. Smari, "On the road to exascale: advances in high performance computing and simulations—an overview and editorial," *Future Generation Computer Systems*, vol. 82, pp. 450–458, 2018.
- [4] L. DeRose, "The path to delivering programable exascale systems," in *Proceedings of 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, p. 727, Rio de Janeiro, Brazil, May 2019.
- [5] W. A. Shehri, M. Khemakhem, A. Basuhail, and F. E. Eassa, "A proposed architecture for parallel HPC-based resource management system for big data applications," *Advances in Science, Technology and Engineering Systems Journal*, vol. 4, no. 1, pp. 40–44, 2019.
- [6] A. Uta, A. L. Varbanescu, A. Musaafer, C. Lemaire, and A. Iosup, "Exploring HPC and big data convergence: a graph processing study on intel knights landing," in *Proceedings of 2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 66–77, Belfast, UK, 2018–September.
- [7] A. Gittens, A. Devarakonda, E. Racah et al., "Matrix factorizations at scale: a comparison of scientific data analytics in spark and C+MPI using three case studies," in *Proceedings of 2016 IEEE International Conference on Big Data (Big Data)*, pp. 204–213, Washington, DC, USA, December 2016.
- [8] W. Al Shehri, M. Khemakhem, F. Eassa, A. Basuhail, and F. E. Eassa, "Evaluation of high-performance computing techniques for big data applications," *Science International*, vol. 31, no. 1, pp. 149–163, 2019.
- [9] S. Usman, R. Mehmood, and I. Katib, *Big Data and HPC Convergence for Smart Infrastructures: A Review and Proposed Architecture*, Springer Science and Business Media Deutschland GmbH, Berlin, Germany, pp. 561–586, 2020.
- [10] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Communications of the ACM*, vol. 58, no. 7, pp. 56–68, 2015.
- [11] W. Al Shehri, M. Khemakhem, A. Basuhail, and F. E. Eassa, "A hybrid spark MPI OpenACC system," *International Journal of Computer Science and Network Society*, vol. 19, no. 5, pp. 81–86, 2019.
- [12] D. Unat, A. Dubey, T. Hoefler et al., "Trends in data locality abstractions for HPC systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 3007–3020, 2017.

- [13] G. Lee, N. Tolia, P. Ranganathan, and R. H. Katz, "Topology-aware resource allocation for data-intensive workloads," in *Proceedings of the First ACM Asia-Pacific Workshop on Workshop on Systems-APSys '10*, Hangzhou, China, August 2010.
- [14] R. Raman, M. Livny, and M. Solomon, "Matchmaking: distributed resource management for high throughput computing," in *Proceedings of the Seventh International Symposium on High Performance Distributed Computing*, pp. 140–146, Chicago, IL, USA, July 1998.
- [15] P. Mohanamurthy and G. Staffelbach, "Hardware locality-aware partitioning and dynamic load-balancing of unstructured meshes for large-scale scientific applications," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, Geneva, Switzerland, June 2020.
- [16] S. L. Mirtaheri and L. Grandinetti, "Optimized load balancing in high-performance computing for big data analytics," *Concurrency and Computation: Practice and Experience*, p. e6265, 2021, Inpress.
- [17] K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, and I. Raicu, "Optimizing load balancing and data-locality with data-aware scheduling," in *Proceedings of 2014 IEEE International Conference on Big Data (Big Data)*, pp. 119–128, Washington, DC, USA, October 2014.
- [18] C. N. Nguyen, J. Lee, S. Hwang, and J.-S. Kim, "On the role of message broker middleware for many-task computing on a big-data platform," *Cluster Computing*, vol. 22, no. S1, pp. 2527–2540, 2019.
- [19] A. Di Stefano, A. Di Stefano, G. Morana, and D. Zito, "Coope4M: a deployment framework for communication-intensive applications on mesos," in *Proceedings of 2018 IEEE 27th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pp. 36–41, Paris, France, June 2018.
- [20] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles-SOSP'09*, p. 261, Big Sky, MT, USA, October 2009.
- [21] J. Yin, A. Foran, and J. Wang, "DL-MPI: enabling data locality computation for MPI-based data-intensive applications," in *Proceedings of 2013 IEEE International Conference on Big Data*, pp. 506–511, Silicon, Valley, CA, USA, October 2013.
- [22] H. Vandierendonck and T. Mens, "Averting the next software crisis," *Computer*, vol. 44, no. 4, pp. 88–90, 2011.
- [23] J. D. Garcia, D. del Rio, M. Aldinucci et al., "Challenging the abstraction penalty in parallel patterns libraries," *The Journal of Supercomputing*, vol. 76, no. 7, pp. 5139–5159, 2020.
- [24] A. Johanson and W. Hasselbring, "Software engineering for computational science: past, present, future," *Computing in Science & Engineering*, vol. 20, no. 2, pp. 90–109, 2018.
- [25] N. Giacaman and O. Sinnen, "Preparing the software engineer for a modern multi-core world," *Journal of Parallel and Distributed Computing*, vol. 118, pp. 247–263, 2018.
- [26] T. Marrinan, G. Eisenhauer, M. Wolf, J. A. Insley, S. Rizzi, and M. E. Papka, "Parallel streaming between heterogeneous HPC resources for real-time analysis," *Journal of Computational Science*, vol. 31, pp. 163–171, 2019.
- [27] A. Mastoras and T. R. Gross, "Understanding parallelization tradeoffs for linear pipelines," in *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores*, pp. 1–10, Vienna, Austria, February 2018.
- [28] A. Meade, J. Buckley, and J. J. Collins, "Challenges of evolving sequential to parallel code," in *Proceedings of the 12th International Workshop and the 7th Annual ERCIM Workshop on Principles on Software Evolution and Software Evolution-IWPSE-EVOL'11*, Szeged, Hungary, September 2011.
- [29] 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.
- [30] S. Jha, J. Qiu, A. Luckow, P. Mantha, and G. C. Fox, "A tale of two data-intensive paradigms: applications, abstractions, and architectures," in *Proceedings of 2014 IEEE International Congress on Big Data*, pp. 645–652, Anchorage, AK, USA, June 2014.
- [31] N. Satish, N. Sundaram, M. M. A. Patwary et al., "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pp. 979–990, Snowbird, UT, USA, June 2014.
- [32] G. M. Slota, S. Rajamanickam, and K. Madduri, "A case study of complex graph analysis in distributed memory: implementation and optimization," in *Proceedings of 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 293–302, Chicago, IL, USA, May 2016.
- [33] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, pp. 293–307, Santa Clara, CA, USA, March 2015.
- [34] M. Anderson, S. Smith, N. Sundaram et al., "Bridging the gap between HPC and big data frameworks," *Proceedings of the VLDB Endowment*, vol. 10, no. 8, pp. 901–912, 2017.
- [35] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu, "DataMPI: extending MPI to hadoop-like big data computing," in *Proceedings of 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 829–838, Phoenix, AZ, USA, May 2014.
- [36] "H₂O.ai. Sparkling Water." <https://github.com/h2oai/sparkling-water> (accessed Aug. 14, 2018).
- [37] "deeplearning4j. Deep Learning for Java. Open-Source, Distributed, Deep Learning Library for the JVM." <https://deeplearning4j.org/> (accessed Aug. 14, 2018).
- [38] M. Grossman and V. Sarkar, "SWAT," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pp. 81–92, Kyoto, Japan, June 2016.
- [39] A. Gittens, K. Rothauge, S. Wang et al., "Accelerating large-scale data analysis by offloading to high-performance computing libraries using alchemist," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 293–301, London, UK, August 2018.
- [40] L. C. Platforms, P. Strategies, and Y. Yan, "Introduction to OpenACC Directives," 2012.
- [41] Databricks, "Intro to Apache Spark," 2015. https://stanford.edu/~rezab/sparkclass/slides/itas_workshop.pdf.
- [42] "MPI_Cart_create(MPI_Comm Comm_old, Int Ndims, Int *dims, Int *periods, Int Reorder, MPI_Comm *comm_cart) Function." https://mpi.deino.net/mpi_functions/MPI_Cart_create.html (accessed Oct. 18, 2019).
- [43] "SNAP: network datasets: social circles." <https://snap.stanford.edu/data/ego-Twitter.html> (accessed Dec. 01, 2019).
- [44] "Ubuntu Manpage: Powerstat-a tool to measure power consumption." <http://manpages.ubuntu.com/manpages/xenial/man8/powerstat.8.html> (accessed Oct. 18, 2019).