Cluster Programming using the OpenMP Accelerator Model

HERVÉ YVIQUEL, LAURO CRUZ, and GUIDO ARAUJO, Institute of Computing, University of Campinas (UNICAMP)

Computation offloading is a programming model in which program fragments (e.g., hot loops) are annotated so that their execution is performed in dedicated hardware or accelerator devices. Although offloading has been extensively used to move computation to GPUs, through directive-based annotation standards like OpenMP, offloading computation to very large computer clusters can become a complex and cumbersome task. It typically requires mixing programming models (e.g., OpenMP and MPI) and languages (e.g., C/C++ and Scala), dealing with various access control mechanisms from different cloud providers (e.g., AWS and Azure), and integrating all this into a single application. This article introduces computer cluster nodes as simple OpenMP offloading devices that can be used either from a local computer or from the cluster head-node. It proposes a methodology that transforms OpenMP directives to Spark runtime calls with fully integrated communication management, in a way that a cluster appears to the programmer as yet another accelerator device. Experiments using LLVM 3.8, OpenMP 4.5 on well known cloud infrastructures (Microsoft Azure and Amazon EC2) show the viability of the proposed approach, enable a thorough analysis of its performance, and make a comparison with an MPI implementation. The results show that although data transfers can impose overheads, cloud offloading from a local machine can still achieve promising speedups for larger granularity: up to 115× in 256 cores for the *2MM* benchmark using 1GB sparse matrices. In addition, the parallel implementation of a complex and relevant scientific application reveals a 80× speedup on a 320 core machine when executed directly from the headnode of the cluster.

CCS Concepts: • **Computing methodologies** → **Distributed programming languages**; • **Software and its engineering** → **Parallel programming languages**; **Distributed programming languages**; **Runtime environments**; *Source code generation*;

Additional Key Words and Phrases: OpenMP, LLVM, Apache Spark

ACM Reference format:
Hervé Yviquel, Lauro Cruz, and Guido Araujo. 2018. Cluster Programming using the OpenMP Accelerator Model. *ACM Trans. Archit. Code Optim.* 15, 3, Article 35 (August 2018), 23 pages.
https://doi.org/10.1145/3226112

Extension of Conference Paper: This article extends our previous work [1] by adding support for intra-cluster execution and dynamic loop scheduling. It also evaluates a complex application which resolves a relevant scientific problem on real dataset, and compares the results with a standard distributed MPI implementation.

This work is supported by CCES CEPID/FAPESP under Grants No. 2014/25694-8 and No. 2017/21339-7. The experiments are also supported by the Microsoft Azure for Research program and the AWS Cloud Credits for Research program.

Authors' addresses: H. Yviquel, L. Cruz, and G. Araujo, Institute of Computing, University of Campinas (UNICAMP), Av. Albert Einstein 1251, Cidade Universitária, Campinas, São Paulo, Brazil; emails: herve.yviquel@ic.unicamp.br, laurocruzsouza@gmail.com, guido@ic.unicamp.br.

1544-3566/2018/08-ART35 $15.00
https://doi.org/10.1145/3226112

1 INTRODUCTION

Parallelizing loops is a well-known research problem that has been extensively studied. Most of the approaches to this problem use DOALL [2], DOACROSS [3], DSWP [4], vectorization [5], data rearrangement [6], and algebraic and loop transformations [7] to improve program performance. However, the combination of large data-center clusters and Map Reduce-based techniques, like those found in Internet search engines [8], has opened up opportunities for cloud-based parallelization, which could eventually improve program performance.

Although there is a number of approaches to loop parallelization, in general-purpose computers this is typically achieved through message passing programming models like MPI [9] or multi-threading-based techniques such as those found in the OpenMP standard [10]. OpenMP is a directive-based programming model in which program fragments (e.g., hot loops) are annotated to ease the task of parallelizing code. The last version of the OpenMP standard [11] (Release 4.5) introduces new directives that enable the transfer of computation to heterogeneous computing devices (e.g., GPUs). From the programmer viewpoint, a program starts running on a typical processor host, and when an OpenMP annotated code fragment is reached, the code is transferred to the indicated device for execution, returning the control flow to the host after completion, a technique called *offloading*.

Although OpenMP offloading has been extensively used in combination with powerful computing devices like GPUs, parallelizing computation to very large clusters can become a complex and cumbersome task. It typically requires mixing different programming models (e.g., OpenMP and MPI) and languages (e.g., C/C++ and Scala), dealing with access control mechanisms from distinct cloud services, while integrating all this together into a single application. This task can become a major programming endeavor that can exclude programmers who are not parallel programming experts from using the huge computational resources available in the cloud [12].

To address such a problem, this article presents an open-source development framework, called OmpCloud,[1] which integrates OpenMP directives, Spark-based MapReduce parallelization, and remote communication management into a single OpenMP offloading device that can be seen by the programmer as available from its local computer. To achieve that, it relies on the OpenMP accelerator model to include the cluster as a new device target. The cluster resources (e.g., execution nodes, data storage) are identified using a specific configuration file and a runtime library allows the programmer to get rid of all glue code required for the interaction with the cloud infrastructure. The main contributions of this article are the following:

- It proposes a new distributed parallel programming model that combines traditional parallelization techniques with map-reduce-based computation to enable the generation of parallel distributed code;
- It introduces remote cluster nodes as OpenMP computing devices making the task of mapping local source code to local or remote clusters transparent to the programmer, a useful tool specifically for those programmers who do not master parallel programming skills and cloud computing;

The remainder of this article is organized as follows. First, we start by introducing the basic concepts involved in directive-based programing and cloud computing (Section 2). We then describe the proposed approach in Section 3. Section 4 presents and analyzes the experimental results, and Section 5 discusses the related works. Finally, we conclude in Section 6.

[1] OmpCloud is freely available at http://ompcloud.github.io.

```
1  void MatMul(float *A, float *B, float *C) {
2    // Offload code fragment to the cloud
3    #pragma omp target device(CLOUD)
4    #pragma omp target map(to: A[:N*N], B[:N*N]) map(from: C[:N*N])
5    // Parallelize loop iterations on the cluster
6    #pragma omp parallel for
7    for(int i=0; i < N; ++i) {
8      for (int j = 0; j < N; ++j) {
9        C[i * N + j] = 0;
10       for (int k = 0; k < N; ++k)
11         C[i * N + j] += A[i * N + k] * B[k * N + j];
12     }
13   } // Resulted matrix 'C' is available locally
14 }
```

Listing 1. Using the OpenMP accelerator model for executing matrix multiplication on a computer cluster.

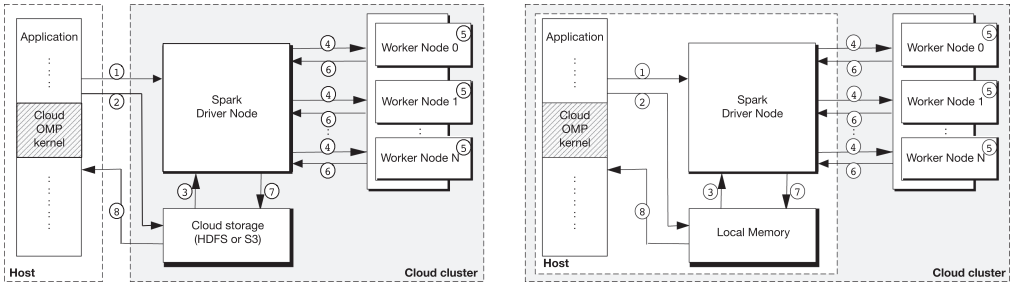
2 BACKGROUND

Cloud computing has been considered a promising platform that could free users from the need to buy and maintain expensive computer clusters, while enabling a flexible and pay-as-needed computational environment. Although it has been successfully used to handle the rise of social media and multimedia [13], all such systems have been designed using a programming abstraction that clearly separates the input/output of local data from cloud computation. This goes against a clear tendency in computing to ease the integration of data collection to the huge resources available in the cloud, as demanded by modern mobile devices and Internet-of-Things (IoT) networks. For example, by collecting data from a cellphone and transparently sending it to the cloud, one could use expensive Machine Learning (ML) algorithms to identify the best device parameters, thus tuning its operation to the user usage profile. Easing the integration of local code with the cloud is central to enable such type of applications. The OmpCloud approach proposed in this article aims to bridge this gap.

The MapReduce [8] programming model associated with the Hadoop Distributed File System (HDFS) [14] has become the *de facto* standard used to solve large problems in the cloud [15, 16]. A generalization of the MapReduce model, Spark [17] has enabled the design of many complex cloud-based applications and demonstrated very good performance numbers [18–21]. To achieve such performance, the Spark runtime relies on an innovative data structure, called *Resilient Distributed Dataset* (RDD) [17], which is used to store distributed data collections with the support of parallel access and fault-tolerance.

Although it has been used to solve many large-scale problems, the combination of MapReduce and HDFS has not become a programming model capable of turning the cloud into a computing device easily used by non-expert programmers. However, the emergence of multicore computing platforms has enabled the adoption of *directive-based programming models*, which have simplified the task of programming such architectures. In directive-based programming, traditional programming languages are extended with a set of directives (such as *C pragma*) that informs the compiler about the parallelism potential of certain portions of the code, usually loops but also parallel sections and pipeline fragments. OpenACC [22] and OpenMP [10] are two examples of such language extensions that rely on thread-level parallelism. Due to its simplicity and seamless mode, OpenMP is probably the most popular directive-based programming interface in use today.

Listing 1 presents a simple program loop describing a matrix multiplication that was annotated with OpenMP directives to offload the computation to an accelerator. In the OpenMP abstract



(a) Offloading mode (b) Intra-custer mode

Fig. 1. Applications can be run from the user computer or from the headnode of the cluster (in gray is the cluster).

accelerator model, the *target* pragma defines the portion of the program that will be executed by the target device. The *map* clause details the mapping of the data between the host and the target device: inputs (A and B) are mapped *to* the target, and the output (C) is mapped *from* the target. While typical target devices are DSP cores, GPUs, Xeon Phi accelerators, and so on [23, 24], this article introduces the cloud as yet another target device available from the local computer, giving the programmer the ability to quickly expand the computational power of its own computer to a large-scale cloud cluster. Using OmpCloud, the programmer can leverage on his/her basic OpenMP knowledge. He first uses standard annotations to validate a small computation in a local machine and then, after a few modifications, migrates the computation to the cloud for a more expensive execution.

3 USING OPENMP ACCELERATOR MODEL FOR DISTRIBUTED EXECUTION

Instead of parallelizing program fragments across heterogeneous cores within a single computer, our runtime automatically parallelizes loop iterations by offloading kernels across multiple machines of a computer cluster that can be available locally or in the cloud. To achieve that it uses the Apache Spark framework, while transparently providing desirable features like fault tolerance, data distribution, and workload balancing. Most cloud providers have made the deployment of Spark clusters pretty straightforward thanks to dedicated web interface and custom Linux distribution (e.g., Azure HDinsight or Amazon EMR). The OmpCloud user can easily create his own cluster with just a few clicks without knowing much about parallel programming and cloud computing.

As shown in Figure 1, the execution of the OpenMP annotated code on the cluster device can have two different operation modes: the *offloading* mode when the program is started from the user computer (Figure 1(a)) or the *intra-cluster* mode when the program is started from the head node of the cluster (Figure 1(b)). In both modes, the user needs to use a Spark cluster, which was previously deployed at his local cluster or through his favorite cloud provider. In *offloading* mode, the user must first store the credentials of the cluster into a specific configuration file (detailed later in Section 3.1). The program is then started by the user in its own local machine and runs locally until the OpenMP annotated code fragment is reached. A method is then called to initialize the cloud device ①. Offloading is done dynamically, and thus if the cloud is not available the computation is performed locally. The runtime sends the input data required by the kernel as binary files to a cloud storage device (e.g., AWS S3 or any HDFS server) ②. After all the input data has been transmitted, the runtime submits the job to the Spark cluster and blocks until the end of the job execution. The *driver node*, which is in charge of managing the cluster, reads the input data

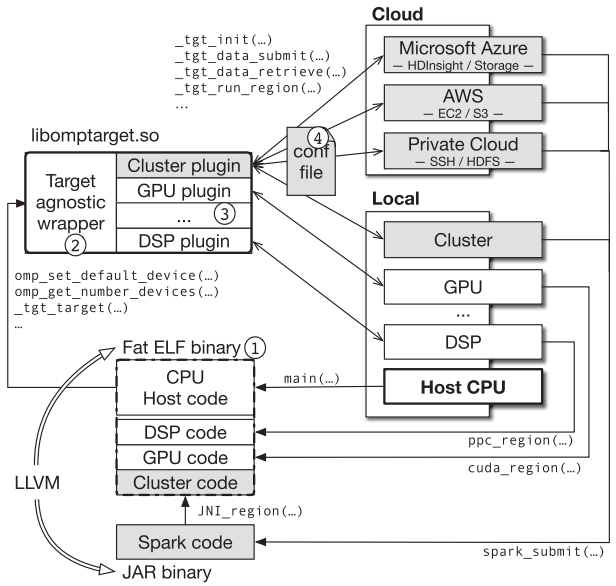


Fig. 2. Modular implementation of the OpenMP accelerator model; in gray is what we implemented to enable local or cloud clusters as novel devices.

from the cloud file system ③, transmits the input data and distributes the loop iterations across the Spark *worker nodes* ④, which are in charge of the computations. Next, the *worker nodes* run the mapping function that computes the loop body in parallel ⑤. The output of the loop is then collected, reconstructed by the driver ⑥ and stored into the cloud storage ⑦ to be transmitted back to the local program ⑧, which then continues the execution on the local machine. In *intra-cluster* mode, the behavior is similar except that the user runs its application directly from within the cluster headnode. Thus, the runtime does not copy input and output data to the cloud storage but directly accesses the local memory of the headnode computer. Hence, although the overhead induced by the offloading is removed, the user has connect to the cluster trough SSH and setup the headnode with the program and its data before running the execution. In fact, such an execution model is very similar to the one used by scientists who run batch jobs on clusters through SSH.

3.1 Offloading to Local and Cloud Clusters

Our workflow relies on a flexible implementation of the OpenMP accelerator model, presented in Figure 2. Such an implementation was developed by Jacob et al. within the OpenMP offloading library [25, 26] (known as *libomptarget*) and the LLVM compiler [27]. Their implementation relies on runtime calls made by the host device for the execution of the offloaded code on the target device. In the initial implementation, offloading was implemented only for typical devices like general-purpose processors (×86 and PowerPC) and NVIDIA GPUs (running CUDA code). In this article, we extended the LLVM compiler to generate code for Spark-based cloud devices and the *libomptarget* library to allow the offloading of data and code to such devices. The extensions proposed in this article are shown in gray in Figure 2.

To ease the implementation of new accelerators, Jacob et al. [25] decomposed their implementation in distinct components (see Figure 2):

① *Fat binary generated by LLVM*—which contains host and target codes. While host code (contained in the `main(...)` function) and target codes (such as function

`ppc_region(...)`) are typically embedded in the same fat binary using the ELF format, cluster devices require an additional file to be generated: the Scala code describing the Spark job (compiled to JAR binary). When submitting the job to the cluster, the *driver node* runs the Scala program and distributes the loop iterations among the *worker nodes*. Then, the workers natively run (in C/C++) the function describing the loop body (`JNI_region(...)`) through the Java Native Interface (JNI) to avoid the translation of C/C++ code to Scala. Obviously, this code had to be compiled to a binary format compatible with the architecture of the cluster nodes;

② *Target-agnostic offloading wrapper*—which is responsible for the detection of the available devices, the creation of devices' data environments, the execution of the right offloading function according to the device type. The wrapper implements a set of user-level runtime routines (such as `omp_get_num_devices(...)`) and compiler-level runtime routines (such as `_tgt_target(...)`), which allow the host code to be independent of the target device type;

③ *Target-specific offloading plug-ins*—which performs the direct interaction with the devices, according to their architecture and provides services such as the initialization and transmission of input and output data, and the execution of offloaded computation. In our case, the plugin is used to initialize the cluster, to compress and transmit the offloaded data through the cloud file storage (HDFS or S3), when using remote clusters, and to submit the Spark jobs locally or through SSH connection.

There are some major differences when using the cloud to offload computation when compared to other traditional target devices, such as GPUs. For instance, the host-target communication overhead might be reduced by compressing offloaded data, and transmitting them in parallel. Our cloud plugin automatically creates a new thread for transmitting each offloaded data (possibly after gzip compression [28, 29] if the data size is larger than a predefined minimal compression size). Additionally, the user can choose to print the log messages of Spark to the standard output of the host computer to check the current state of the computation. Another remarkable difference is that cloud devices cannot be detected automatically, since they are not physically hosted at the local computer. As a matter of fact, the user has to provide identification/authentication information (e.g., address and login) to allow the connection of the current application to the cloud service, which will be used for offloading (cluster and storage). Our plugin reads at runtime a configuration file ④ to properly set up the cluster device and to avoid the need to recompile the binary (assuming compatible instruction-sets). Besides the login information, the file might also contain a set of runtime configurations to allow more advanced users to fine-tune the execution (passing any supported options to the Spark runtime for example). In general, when the user is also the developer of the application, he is responsible for deploying the cluster and setting up the configuration file before running the application. However, for commercial use-cases when the user is not the developer, the cloud cluster could be provided by the developers to the end-user through a predefined configuration file.

To allow an easy portability over existing cloud services, our plugin was implemented as a modular infrastructure where the communication with the cloud can be customized for each existing cloud service by taking into account their specificities (e.g., storage services, security mechanisms, etc.). For now, our plugin supports computation offloading to Spark clusters running within a private cloud, Amazon Elastic Compute Cloud (EC2), or Microsoft Azure HDInsight. We also support data offloading to HDFS, Amazon Simple Storage Service (S3), and Microsoft Azure Storage. This approach can be easily extended to support other commercial cloud services, like Cloudera or Google Cloud. Moreover, during offloading our library is also able to (on-the-fly) start and stop

```
1  #pragma omp target device(CLOUD)
2  #pragma omp target map(to: A[:N*N], B[:N*N]) map(from: C[:N*N])
3  #pragma omp parallel for
4  for(int i=0; i < N; ++i) {
5  // Partitions A and C matrices among the worker nodes
6  #pragma omp target data map(to: A[i*N:(i+1)*N]) map(from: C[i*N:(i+1)*N])
7    for (int j = 0; j < N; ++j) {
8      C[i * N + j] = 0;
9      for (int k = 0; k < N; ++k)
10       C[i * N + j] += A[i * N + k] * B[k * N + j];
11   }
12 }
```

Listing 2. Extending OpenMP data map directive to enable dynamic data partitioning.

virtual machines from the EC2 service. In other words, the EC2 instance can be started when offloading the code and stopped after it ends its execution. As a result, the programmer can automatically control the usage of the cloud infrastructure, thus allowing him/her to pay for just the amount of computational resources used.

3.2 Extending OpenMP for Distributed Data Partitioning

One central issue in distributed parallel execution models is to enable a data partitioning mechanism that assigns a specific data block to the worker node containing the kernel code that will use it. By doing so, programs can considerably benefit from locality, thus reducing the overhead of moving data around interconnecting networks. Nevertheless, automatic data partitioning is a hard task that cannot typically be achieved solely by the compiler or runtime. In most applications the programmer knowledge is essential to enable an efficient data allocation. Unfortunately, the OpenMP standard does not have directives specifically designed for data partitioning within offloaded regions. As a result, the programming model proposed in this article extends the use of the OpenMP directive *target data map* to allow the programmer to express the data distribution to the cloud Spark nodes. No syntax modification was required in this directive; it was only used in a way that is mentioned as *undefined behavior* by the current OpenMP specification. To do that, the programmer should indicate after the *to/from* specifier of the *map* directive the first element of the partitioned data block followed by colon and the last element of the corresponding block.

Consider, for example, the code fragment in Listing 2 extracted from the matrix multiplication example in Listing 1. It is well-known that matrix multiplication $C = A \times B$ implies in multiplying the rows of A by the columns of B storing the result as elements of C. Hence, to improve locality the programmer can insert line 6 of Listing 2 to specify the partitioning of the matrices during the iterations of the parallel loop. For example, in line 6 of Listing 2 the rows of matrix A are indexed using variable `i`.[2] By using `map(to:i*N:(i+1)*N)` the programmer states that all elements of row `i` of A ranging from index `i*N` to `(i+1)*N` should be allocated into the same Spark node. Please notice that B is deliberately not partitioned, because its partition interval depends on the internal loop counter j (indexing the column). In our implementation, the partitioning that reduces the amount of data moving through the network is performed by the Spark driver node. But Spark only knows the values taken by the induction variable of the outer loop (i.e., the `parallel for` annotated loop). Partitioning B would require us to coalesce the internal loop into the external one, which would increase the number of iterations, thus reducing the granularity of the parallelization and increasing the scheduling and communication overheads. For this reason, each worker

[2]Matrices A, B, and C in Listing 2 are represented in their linearized forms.

node receives a full copy of B to perform its part of the computation. In reality, the communication overhead will be limited by the efficiency of BitTorrent protocol used by Spark to broadcast variables.

3.3 Matching Spark Execution Model

As said before and shown in Figure 2, Spark clusters are composed by one driver node associated with a set of worker nodes (or simply *workers*). The driver is in charge of communication with the outside world (i.e., host computer), resource allocation and task scheduling. The workers perform computation by applying operations, mostly *map* and *reduce*, in parallel on large datasets. In our programming model, given that the loop has been annotated using a *parallel for* pragma, the programmer assumes that it is a DOALL loop. Thus, the different iterations can be distributed and computed in parallel among cloud cores without any restriction, as no loop-carried dependence exists between iterations.

To achieve such a parallelism, Spark relies on a specific data structure called *Resilient Distributed Dataset* (RDD) [17]. An RDD is basically a collection of data that is partitioned among the workers that apply parallel operations to them. To allow the parallel execution of a DOALL loop with N iterations, we build an initial RDD, such that

$$RDD_{IN} = \bigcup_{i=0}^{N-1} \{(i, V_{IN}(i))\}, \tag{1}$$

$$V_{IN}(i) = \{V_{IN_0}(i), \ldots, V_{IN_{K-1}}(i)\}, \tag{2}$$

where $i \in \{0, \ldots, N-1\}$ are the values taken by the loop index during loop execution and $V_{IN}(i)$ is the set of K input variables (i.e., *r-values*) read during the execution of the loop body at iteration i. For any iteration i, each input $V_{IN_k}(i)$ with $k \in \{0, \ldots, K-1\}$ can be either a full variable or a portion of it depending if the programmer has described the partitioning as presented in Listing 2.

RDD_{IN} is divided automatically by the driver in equal parts and distributed among the workers $w \in \{0, 1, \ldots, W-1\}$, such that

$$RDD_{IN}(w) = \bigcup_{m=w*\lfloor N/W \rfloor}^{(w+1)*\lfloor N/W \rfloor - 1} \{(m, V_{IN}(m))\}. \tag{3}$$

A map operation (Equation (4)) is then applied to the RDD that passes the values taken by the loop index and the input variables through a function describing the loop body (Equation (5)) and returns a new RDD_{OUT}, such that

$$RDD_{OUT} = MAP(RDD_{IN}, loopbody), \tag{4}$$

$$V_{OUT}(i) = loopbody(i, V_{IN}(i)), \tag{5}$$

$$V_{OUT}(i) = \{V_{OUT_0}(i), \ldots, V_{OUT_{L-1}}(i)\}, \tag{6}$$

$$RDD_{OUT} = \bigcup_{i=0}^{N-1} \{(i, V_{OUT}(i))\}, \tag{7}$$

where $V_{OUT}(i)$ (Equation (6)) is the set of L output variables (i.e., *l-values*) produced at iteration i by the workers. In fact, each call to the *loopbody* function produces a partial value of the output variables V_{OUT}, since each iteration accesses a different part of the output variables in DOALL loops. Similar to the input variables, each output $V_{OUT_l}(i)$ with $l \in \{0, \ldots, L-1\}$ can be either a full variable or a portion of it depending if the programmer has annotated a partitioning. As a result, the *loopbody* function will only partially compute output variables at each call, even if we know they are not partitioned. Thus, we need to reconstruct the complete outputs V_{OUT} from

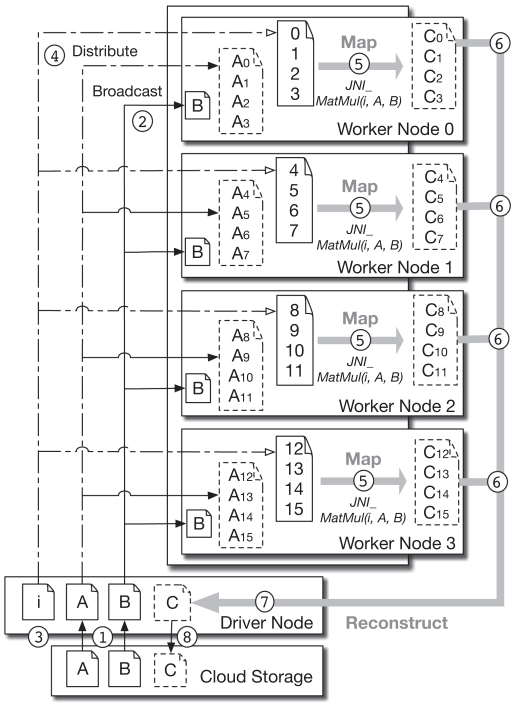


Fig. 3. Using map-reduce computational model to perform matrix multiplication $C = A \times B$ in Spark cluster.

those *partial* output values $V_{OUT}(i)$ as expected by the host computer. This is done according to the following equations:

$$V_{OUT_l} = \begin{cases} Reconstruct(RDD_{OUT}, l) \\ REDUCE(RDD_{OUT}, l, bitor) \end{cases}, \tag{8}$$

$$V'_{OUT_l}(u, v) = bitor(V_{OUT_l}(u), V_{OUT_l}(v)), \tag{9}$$

$$V_{OUT} = \{V_{OUT_0}, \ldots, V_{OUT_{L-1}}\}, \tag{10}$$

where $V'_{OUT_l}(u, v)$ is the partial output value obtained by combining $V_{OUT_l}(u)$ and $V_{OUT_l}(v)$ (Equation (9)). If the loop body has several outputs (Equation (10)), then RDD_{OUT} will simply be composed of tuples that are reconstructed separately before being written back in different binary files. Thus, we consider the set of all partial values of each output variable $V_{OUT_l}(i)$ as arrays of bytes. If the variable was partitioned, then the driver allocates the full variable and writes each value at the right index. If the programmer has not annotated the partitioning, then we simply apply a bitwise-or reduction (Equation (8)) to join them together. Additionally, if one of the outputs has been defined as a reduction variable by the OpenMP clause, Spark just performs the reduction using the predefined function instead of the bitwise-or.

To illustrate this process, let us consider the matrix multiplication $C = A \times B$ presented previously in Listing 1. As shown in Figure 3, the Spark driver node gets the files ① representing the input data from the cloud storage (HDFS or S3), and loads them as ByteArray objects. It then generates $RDD(I)$, which contains the successive values taken by the loop index i $(0, \ldots, 15$ in our case), splits A according to the partitioning bound defined by the user (as a function of i), distributes them equitably to the worker cores ④ while broadcasts B ② to the same cores. Notice that Spark automatically compresses all data transmitted through the network and use the BitTorrent

protocol for broadcasting efficiently. The driver orders a map transformation that applies the *MatMult* function corresponding to the loop body (through JNI) for each loop iteration using partitions of *A* and copies of *B*. The workers decompress the input data and perform in parallel the mapping tasks assigned to their cores ⑤. As a result, they produce an RDD containing sixteen versions of *C*, which needs to be collected to produce the final result for array *C*. To achieve that the workers compress and send the values to the driver ⑥. Then, the driver starts by allocating variable *C* to its full size, before sequentially writing at the right index the values contained by each piece of the array *C*, until obtaining its final format ⑦. The driver finally writes out the final values of *C* to a new file into the cloud storage ⑧ (line 20), after which the local computer is able to read them back and continue its execution.

3.4 OpenMP Loop Scheduling

An important feature of the OpenMP execution model is its ability to define loop scheduling strategies. Programmers can decide how the OpenMP runtime schedules the loop iterations to the processor cores by adding specific clauses to the `parallel for` directives. OpenMP scheduling can be: *static*, *dynamic* or *guided* for a given iteration chunk size. In our workflow, we need to ensure that the Spark runtime follows the choice made by programmers. Spark scheduling is controlled by the number and the size of partitions within RDDs. When performing the map operation, each partition is assigned to one core that performs all the operations related to the elements of the given partition. If the number of partitions is higher than the number of cores, then the later partitions will be dynamically assigned to cores as soon as they become available. As a result, the OmpCloud runtime generates the partitions according to the following loop scheduling strategies set by the programmer:

- Static scheduling: the runtime simply divides the RDD containing the loop iterations into *C* equal-sized partitions with *C* being the number of worker cores available in the cluster. The chunk size is always set to N/C, which is generally the best choice to benefit from data locality. However, parametrized chunk size could be supported using a custom Spark partitioner, which would attribute a partition $P(i)$ to a given index i according to the chunk size S such that $P(i) = \lfloor \frac{i}{S} \rfloor \mod C$. In fact, static scheduling does not benefit from custom chunk sizes because of data locality, except when the ordering is enforced (using the *ordered* clause) but this will not be supported by OmpCloud due to performance issues on distributed systems.
- Dynamic scheduling: the runtime splits the RDD in partitions of chunk size. Then, the internal task queue of the Spark driver automatically gives a partition (i.e., a chunk-sized block of loop iterations) to each worker core. When a core finishes, it retrieves the next partition from the task queue of the driver. By default, the chunk size is 1, but the programmer needs to be careful when using this scheduling policy because of the extra overhead involved with the communication between the driver and the worker nodes. For Spark, the most common advice is to define 3 to 5 tasks (or partitions) to each core.
- Guided scheduling is not yet supported but could be implemented using another custom Spark partitioner.

In addition, our compiler automatically adjusts the iteration number of the outer-loop according to the cluster size using loop tiling to reduce JNI overhead, such as presented in Algorithm 1. Indeed, since each iteration will require one call to JNI, the closer the number of iterations is to the number of partitions, the smaller will be the overhead. The chunk size *K* is passed as an argument when Spark is calling the map functions to avoid any recompilation when executing on different clusters. In case some of the input/output variables are partitioned, the lower and upper bounds

ALGORITHM 1: Reducing overhead with loop tiling.

```
1: // Original parallel for
2: for i = 0 to N − 1 do
3:     loopbody
4: end for
```

```
1: // Tiling the loop for a K chunk size
2: for ii = 0 to N − 1 by K do
3:     for i = ii to min(ii + K − 1, N − 1) do
4:         loopbody
5:     end for
6: end for
```

of the partitions will also be readjusted dynamically according to the tiling size, hence increasing their granularity.

3.5 Application Domain and Programming Model

Before moving into the details of the application domain, it is important to highlight that the goal of the programming model proposed herein is to make the resources of the cloud transparently available to the common non-expert programmer who uses a regular laptop and wants to run large workloads. A typical example is a user that locally collects a large amount of data from a scientific experiment or a mobile device and wants to perform some heavy computation on it [30–32]. Another good use-case for OmpCloud could be the rendering engine of 3D modeling software (e.g., Blender [33]), which requires large amounts of computation to render a big movie project [34]. The rendering could be offloaded to the cloud cluster while the resulting movie would be displayed at the designer desktop computer. Indeed, designers currently need to export their projects from the modeling software and import it by hand to rendering farms within the cloud. It is not a goal of this work to claim a programming model that can deliver HPC type of speedups for a complex specialized scientific application (e.g., ocean simulation): there is already a huge amount of tools and programming models (e.g., MPI) that work well in this scenario [35].

Of course, this specific solution does not match well the computation requirements of any kind of application. First, the problem to be solved has to be sufficiently complex to allow the application to take advantage of the large parallel processing capabilities of the cloud when compared to the overhead cost of the data offloading task. Nevertheless, one might run his application directly from the driver node of the Spark cluster, thus removing the overhead of host-target communication. Second, applications should be described in C/C++ and annotated using directives defined by the OpenMP accelerator model. While this article presents a matrix multiplication annotated with just one *target* pragma, one *target map* and one *parallel for*, our approach also supports more complex OpenMP constructs such as those using several *parallel for* loops within the same *target* region, as illustrated by the vector multiplications presented in Listing 3. This is implemented by performing successive map-reduce transformations within the Spark job. Moreover, similar techniques also allow one to implement the offloading of sequential code kernels or nested parallel loops. As an example, Listing 4 presents another vector multiplication where vector A is initialized within the target region. Please notice that A has been mapped using the *alloc* type. This allows programmers to define mapped variables with undefined initial values that can be allocated on the device without copying from/to the host memory.

Additionally, OpenMP allows programmers to add an *if* clause to *target* pragmas to condition the offloading according to a boolean expression (For example, the size of the vector in Listing 3).

```
#pragma omp target if(N > 1000)
    map(to: A[:N], B[:N], C[:N])
    map(from: D[:N], E[:N])
{
#pragma omp parallel for
  for(int i=0; i < N; ++i)
    D[i] = A[i] * B[i];
#pragma omp parallel for
  for(int i=0; i < N; ++i)
    E[i] = C[i] * D[i];
}
```

Listing 3. 2 parallel multiplications of vector in 1 target region.

```
#pragma omp target map(alloc: A[:N])
     map(to: B[:N]) map(from: C[:N])
{
  init(A);
  #pragma omp parallel for
  for(int i=0; i < N; ++i)
    C[i] = A[i] * B[i];
}
```

Listing 4. Serial code in target region.

In fact, the automatic decision of offloading to a computer cluster versus running on the programmer's processor is an interesting but complex problem that is not supported by our runtime in its current state but will certainly deserve additional research.

Finally, our cloud device does not support the synchronization constructs of the OpenMP model, since Spark relies on a distributed architecture. Thus, offloaded OpenMP regions that use *atomic*, *flush*, *barrier*, *critical*, or *master* directives are not supported. A full implementation of OpenMP on cloud clusters would require a distributed shared memory mechanism [36, 37], which has not yet been proved to be efficient and is incompatible with the map-reduce model. Alternatively, a dedicated programming model suited for distributed nodes could be employed, but we believe that the popularity of OpenMP makes it a better choice.

4 EXPERIMENTAL RESULTS

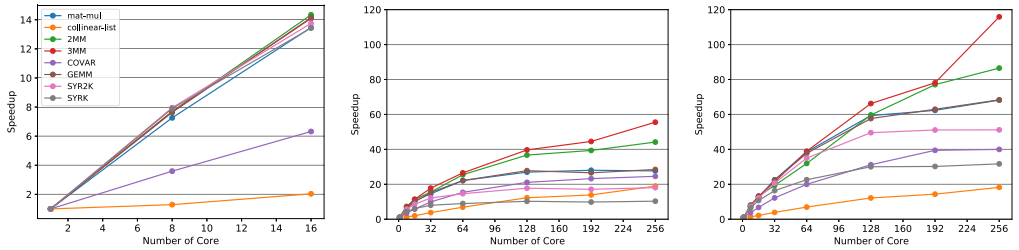
This section explores two sets of experiments that cover the *offloading* and *intra-cluster* modes described in Section 3: the offloading of a set of benchmarks to the cloud executed from the user's computer and the distributed parallelization of a scientific simulation directly from the head node of the cluster.

For all experiments, the applications have been compiled with OmpCloud v0.3.1, our custom fork of Clang/LLVM v3.8, using only the standard *O3* optimization flag. For this work, we chose not to use more advanced optimization techniques, like polyhedral optimizations, as we wanted to characterize a typical use-case and compilation scenario. However, for future work, we can definitely see advantages on using such compilation techniques [38, 39], as they might enable performance improvements by means of cluster-wide tiling and loop optimizations.

4.1 Cloud Offloading of Benchmarks

In our experiments, the local machine is a simple laptop (Intel Core i7 4750HQ and 16GB of RAM) with Ubuntu 16.04, which interacts with an AWS cluster through an Internet connection. Our experiments intend to be a realistic test-case where the client computer is far away from the cloud data-center. The cloud instances were acquired and configured using a third-party script called *cgcloud*[3]. This script allowed us to quickly instantiate a fully operational and highly customizable Spark cluster within AWS infrastructure. For now, the size of the cluster is predefined by the user when running the script but the parallel loop is tiled dynamically to use all instances with the minimal overhead. Our experimental Spark cluster was composed of 1 driver node and 16 worker nodes, all of them running Apache Spark 2.1.0 on top of Ubuntu 14.04. Each node of the cluster is

[3] *cgcloud* is freely available at http://github.com/BD2KGenomics/cgcloud/.



(a) Multi-threaded OpenMP (b) OmpCloud on dense matrices (c) OmpCloud on sparse matrices

Fig. 4. Speedup over single core execution for AWS cloud offloading and multi-threaded OpenMP.

an EC2 instance of type *c3.8xlarge*, which has 32 vCPU (executing on Intel Xeon E5-2680 v2) and 60GB of RAM. Each worker is configured to run one Java Virtual Machine (commonly called Spark executor) that manages all 32 vCPUs and a heap size of 40GB. Since each EC2 vCPU corresponds to one hyper-threaded core according to Amazon description (e.g., 1 dedicated CPU core corresponds 2 vCPUs), we configured Spark to assign two vCPUs to each map and reduce task that we need to run (*spark.task.cpus=2*). Thus, the following benchmark results are presented according to the number of dedicated CPU cores used by all workers (from 8 to 256 cores that are configured thanks to *spark.cores.max* and *spark.default.parallelism* parameters).

We used benchmarks from the *Polyhedral Benchmark* suite [40] and the *MgBench* suite [41], which were previously adapted for the OpenMP accelerator model. We selected for our experiments the set of benchmarks that contains only the supported OpenMP constructs and that could benefit the most of cloud offloading: *SYRK*, *SYR2K*, *COVAR*, *GEMM*, *2MM*, and *3MM* from *Polybench*; and *Mat-mul* and *Collinear-list* from *MgBench*. All data used in the benchmarks consisted of 32-bit floating point numbers (single precision). The dimension of the datasets used by the benchmarks has been scaled to benefit from the Spark distributed execution model. As an example, most matrices used by the benchmarks have been scaled to 16,000 × 16,000 (about 1GB). Moreover, to evaluate the impact of gzip compression [28, 29] on performance, we have deliberately executed the benchmarks using two types of input data: **sparse** matrices (most of the elements are zero) and **dense** matrices (the elements are randomly generated). As expected, sparse matrices are compressed faster with better compression rates (about 1,000:1 and 1.2:1, respectively). All benchmarks used in our experimentation used the classical for loop implementations without any advanced optimizations that would be written by the standard programmer. As a reference, the single-threaded OpenBLAS-based implementation of the matrix multiplication is executed in about 200s on the same architecture.

Figure 4 presents the execution speedups obtained with the benchmarks parallelized with traditional multi-threading of OpenMP (Figure 4(a)) and the cloud offloading of OmpCloud (Figures 4(b) and 4(c)). They were measured when running the benchmarks on both sparse and dense matrices to explore the impact of the data type on the performance, but include the whole offloading time (communication and computation). The multi-threaded OpenMP experiments used only 8 and 16 threads, since the largest AWS EC2 instance of type c3 has only 16 cores. Results with OpenMP show close-to-linear speedups except for *COVAR* and *collinear-list*, which, respectively, reach only 6× and 2× on 16 cores. Globally, all speedups of *OmpCloud* tend to increase with the number of cores: up to 55×/115× with 256 cores for 3MM on dense and sparse matrices, respectively. In fact, even *collinear-list*, which shows the worst performance with multi-threading, was able to reach almost 20× on 256 cores. Results show that the benchmarks having the biggest computing complexity, like *2MM* and *3MM*, benefits the most from larger cluster.

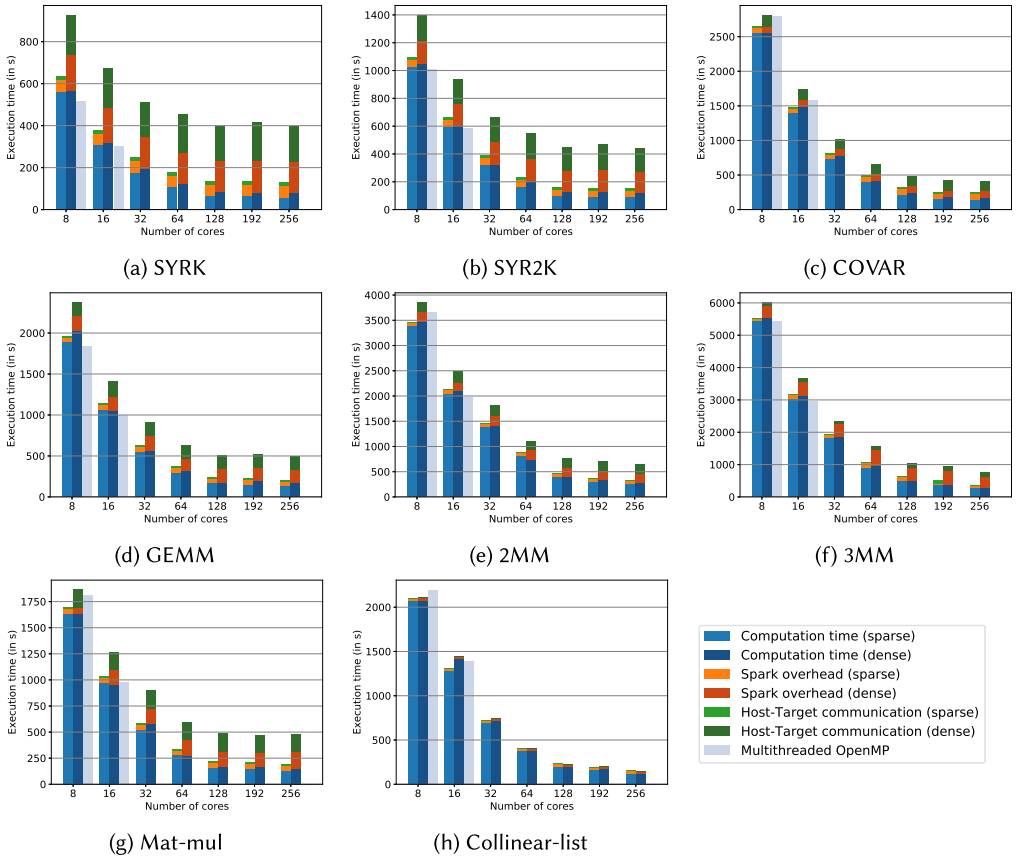


Fig. 5. Average load distribution of AWS cloud offloading according to the total number of worker cores and the data type.

All speedups presented here are interesting indicators to determine the effectiveness of the offloading, since they were computed against single-core execution performed locally (without offloading). However, the problem of determining the effectiveness of offloading depends not only on the size of the dataset and the computation but also on other parameters such as the computing power of the local platform, the connection bandwidth, and so on. This is a very interesting but complex problem that will require additional research.

The execution times of cloud offloading are presented in Figure 5 along with those for multi-threaded OpenMP on a single node. Results also show that: (a) 2 benchmarks are executed on 8 cores in between 10 and 25min; (b) 5 in between 30min and 1h; and (c) 1 in about 1h 30min. Although offloading to a larger cluster could probably benefit from even longer execution times, we were limited by the maximal size of the arrays supported by the Java Virtual Machine. As shown in Figure 5, the distribution of the execution time of all benchmarks were broken into three parts: *Host-target communication* including compression and transmission overhead between the local computer and the cloud device, *Spark overhead* including scheduling and communication within the cluster, and *computation time* (i.e., the execution of loop iterations in parallel through JNI). Such decomposition shows that while the computation time decreases as the number of cores increases, the overhead induced by cloud offloading and Spark distributed execution stays

constant. Moreover, both overheads increase substantially when processing dense matrices (in comparison with sparse ones) but the variation is negligible for the computation time. This demonstrates that the data type (and especially its compressibility) can have a huge impact on performance because of the host-target and intra-cluster communications. Additionally, results of *collinear-list* presented in Chart 5h show a negligible overhead of the communication and scheduling. In fact, *collinear-list* processes a much smaller amount of data than the other benchmarks, showing that cloud offloading scales well when the dataset size stays small according to the computation (i.e., high computation to communication ratio).

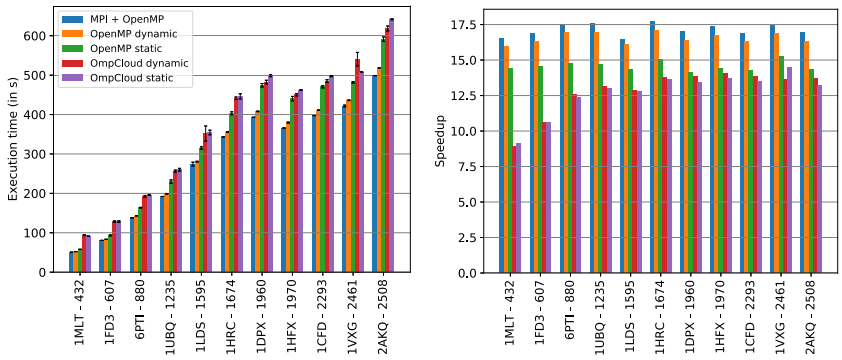
In addition, the comparison of the execution times of cloud offloading to those from multi-threaded OpenMP on 8 and 16 cores (i.e., one worker node) revealed small overheads: (a) just 1.8% when considering only the computation time, what confirms the efficiency of JNI to run native code kernels; (b) 8.8% when considering the spark overhead, what demonstrates the competitive performance of the Spark execution model with respect to multi-threading, even within a driver-worker infrastructure; and (c) 13.6% when considering the total execution time, what shows the limited cost of offloading data to the cloud.

4.2 Intra-Cluster Parallelization of Collision Cross-Section

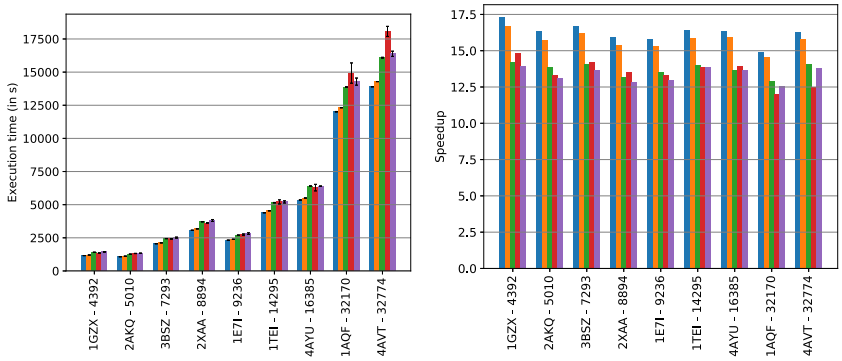
As discussed above, the OmpCloud programming model was not designed to speedup HPC workloads, but to enable the usage of large computing clusters by programmers who do not have access to these resources, or that are not parallel programming experts. However, HPC and its programming models (e.g., MPI) have always worked as an upper bound on the limitations of computation. Hence, in this section, we use a relevant scientific application to evaluate and compare OmpCloud to MPI.

Mass spectrometry is a vital tool for molecular characterization in many areas of Biochemistry analysis. Ion-Mobility Mass Spectrometry (IM-MS) is an analytical (experimental) technique used to separate and identify ionized molecules moving through a drifting gas based on their mobility. In such experiments, molecules move in a tube and bump into the gas molecules. As a result, gas molecules are reflected and the combination of the resulting collision angles is used to estimate the *Collision Cross Section (CCS)*, which depends on the molecule structure. CCS measurements have a number of applications such as determining the conformation of molecules. CCS can also be estimated using simulation of the collision process. The result helps to interpret the experimental CCS obtained from IM-MS. There are a number of methods to simulate CCS computation. The TM method [42, 43] is the most accurate, since it uses Monte Carlo Statistical simulation to handle lots of highly charged big molecules. It is also expensive as it depends on a large number of collisions to calculate the average of the collision angles. In this section, the various parallel implementations of CCS computation are called HPCCS [44].

Three different implementations of HPCCS have been tested during our experiments: a standard OpenMP implementation that relies on multithreading (i.e., on single node only), tested with static and dynamic loop scheduling; an OmpCloud implementation that we have especially adapted to our workflow using the OpenMP accelerator model, tested with static and dynamic loop scheduling; and a mixed MPI+OpenMP implementation, that allows the comparison with a more traditional distributed implementation. The MPI implementation used static loop distribution at cluster-level but dynamic loop scheduling inside each node, and has been experimented with MPICH v3.2. All implementations used the LLVM OpenMP runtime to allow for a fair performance comparison. Our experiments have been performed within the Microsoft Azure cloud using D15v2 dedicated virtual machines of which have 20 vCPUs (executing on Intel Xeon E5-2673 v3 at 2.4 GHz) and 140GB of RAM. We deployed Spark clusters running on top of CentOS 7.2 using MapR (v5.2) and MPI clusters running on top of Ubuntu 16.04 using Azure Resource Manager. The experiments have



(a) Smaller proteins (less than 3000 atoms)



(b) Larger proteins (more than 3000 atoms)

Fig. 6. Performance of the different implementations of HPCCS for various sized proteins running on a single node computer (with 20 cores).

been performed on two sets of variously sized protein samples taken from real experiments [44]. The datasets are sized from 80KB for the smallest protein to about 300KB for the largest one, which is much smaller than the benchmarks previously tested where each matrix was about 1GB.

Figure 6 presents the performance, execution times and speedups, obtained from the execution on a single computer. On the abscissa are the proteins sorted by their corresponding number of atoms. Results show that the application benefits from the dynamic loop scheduling of OpenMP runtime. Indeed, contrary to matrix multiplication, the computation time of HPCCS iterations is irregular. However, the performances of static and dynamic scheduling stay quite similar when using the OmpCloud runtime. Additionally, results shows the overhead induced by each distributed runtime: while MPI does not have any overhead on a single node, the overhead of OmpCloud is significant, from about 30s on small proteins to 30min on the larger ones, but the impact on the speedup reduces for larger proteins thanks to longer execution times. On smaller proteins, the overhead can be explained by the load of the Spark runtime, which could be reduced using advanced tools for dedicated Spark job submission, like Apache Livy [45] or Spark Job Server [46]. Those tools introduce submission through RESTful APIs and maintain a unique Spark context alive between all jobs, thus enabling lower initialization latency. However, we choose not to use them, since they are not yet integrated into common Spark distributions. However, the overhead is increasing significantly on larger proteins, so the load time, more or less constant, cannot

explain it entirely. Since there is no inter-nodes communication (single node), this demonstrates that OpenMP is more efficient than Spark for multithreading. As a result, future OmpCloud implementations could really benefit from automatically using the OpenMP runtime instead of Spark to parallelize within each cluster node, similarly to what is done manually in mixed MPI+OpenMP implementations.

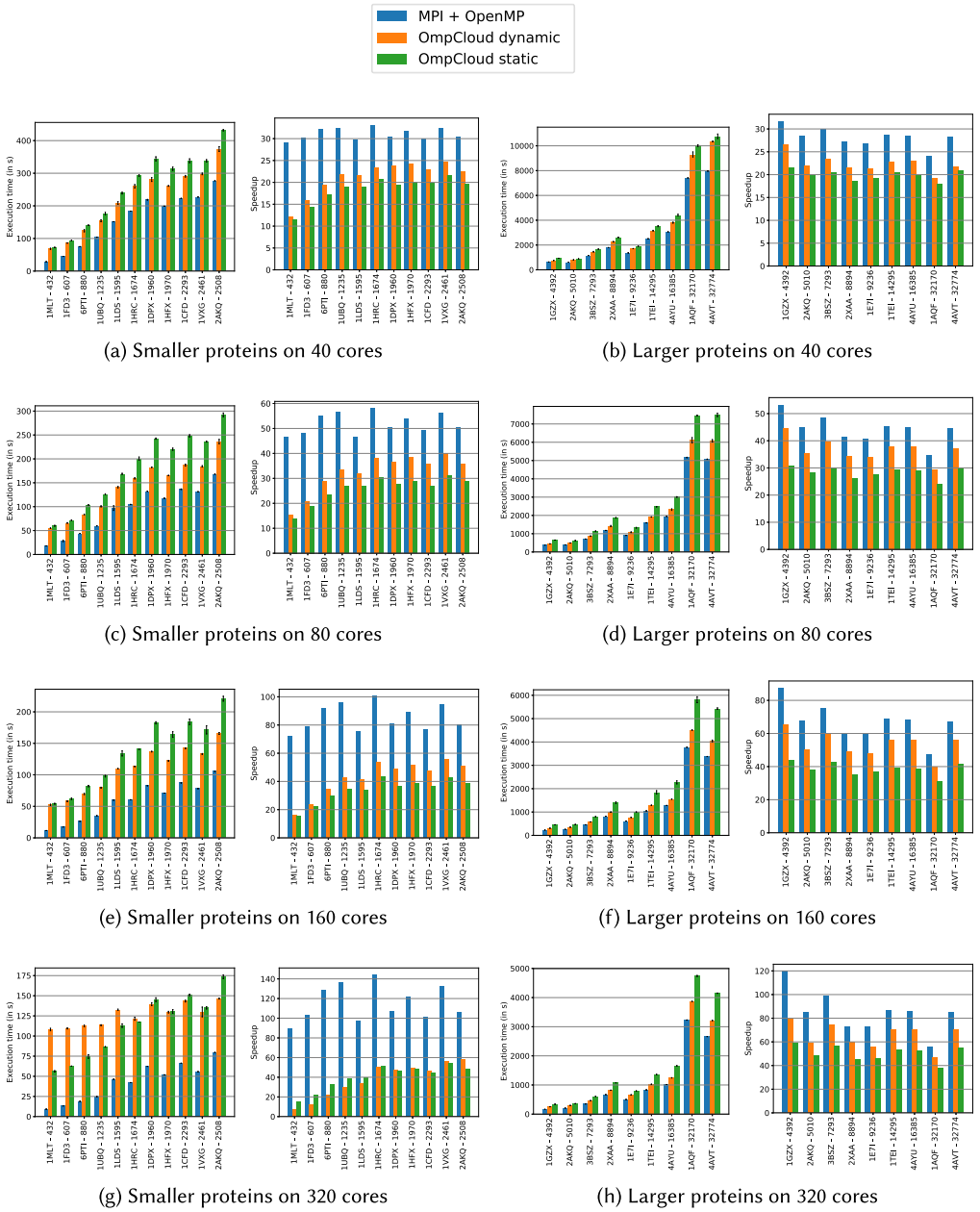
Figure 7 presents the execution times and the speedups obtained from the execution of OmpCloud and MPI implementations on a 16-node cluster. All executions have been performed 5 times and, as shown by the standard deviations presented on the bars, the performance variability of the cloud clusters is negligible. The execution time of OmpCloud implementations range from about 70s to almost 3h on 2 nodes and from 55s to 1.5h on 16 nodes. Knowing the largest protein of our dataset only requires 300KB of memory, we can affirm that the computation-to-communication ratio of HPCCS is largely superior to the ratio of the linear algebra kernels previously tested. As expected, the MPI implementation always obtains the best speedups, varying between 70× and 150× on 320 cores. The OmpCloud implementation is showing poor speedups with small proteins but its efficiency increases with the protein size. In fact, the performance of the smaller proteins are degrade on larger clusters, while they stay relatively stable for larger proteins. For example, we obtained less than 15× on 320 cores with the *1MLT* protein, which is only composed by 432 atoms but 80× on the same number of cores with the *1GZX* protein, which is much bigger (4,392 atoms). The dynamic scheduling of OmpCloud clearly shows better performance than the static one even if the results can be more mitigated on small proteins.

Figure 8 presents the parallel efficiency of the MPI and OmpCloud implementations for each protein. Results shows that the MPI implementation has a better efficiency on smaller proteins (Figure 8(a)) while the efficiency of OmpCloud implementation improves on larger proteins (Figure 8(b)). In fact, the efficiency of the two implementations looks very similar for larger proteins except for the first value of each curve (corresponding to the efficiency of the execution on a 20-core single node), which is around 0.8 for MPI and 0.7 for OmpCloud. This demonstrate that OmpCloud parallelization is competitive against MPI from the cluster perspective when the computation complexity is sufficient but confirms that OmpCloud multithreading within each node needs to be improved to reach the global efficiency of mixed MPI+OpenMP implementations. On the contrary, the efficiency of the MPI implementation seems to reduce when the size of the protein increases. This could be explained by the fact that loop iterations are statically distributed to the computer nodes in this MPI implementation.

5 RELATED WORKS

Cloud offloading has been largely studied for mobile computing to increase the computational capabilities of cellphones [47]. Some frameworks have been proposed to facilitate the development of mobile applications using cloud resources [48–50]. Contrary to our approach, which relies on C/C++, they mostly rely on .NET or Java, which are the most popular environments for mobile device. One of the key challenges of those offloading frameworks, not treated in this article, is how to dynamically determine if it is worth offloading the computation in terms of the communication overhead and energy consumption [51]. Some of those frameworks even explore the parallelization of the execution by providing multi-threading support, or virtual machine duplication, but this is a considerable programming effort for non-experts, and their results do not present very large speedups (up to 4×). Additionally, older works used a similar offloading execution model to accelerate spreadsheet processing in grid computing [52, 53].

Recent works have been proposed to port scientific applications from various domains to private and public clouds [54–57] by using a mixture of MPI and OpenMP; an approach that benefits from the communication efficiency of MPI primitives and the easy parallelization of



(a) Smaller proteins on 40 cores

(b) Larger proteins on 40 cores

(c) Smaller proteins on 80 cores

(d) Larger proteins on 80 cores

(e) Smaller proteins on 160 cores

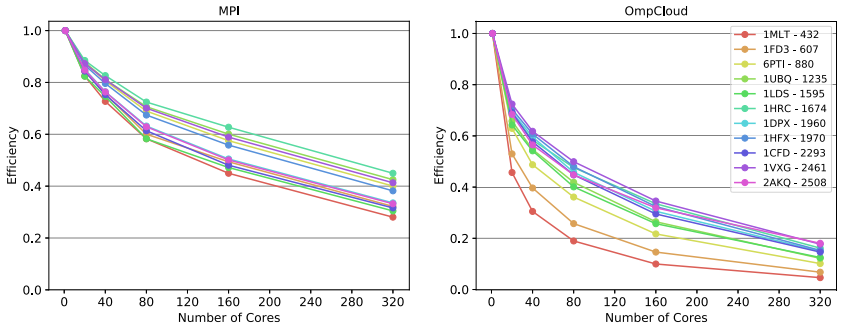
(f) Larger proteins on 160 cores

(g) Smaller proteins on 320 cores

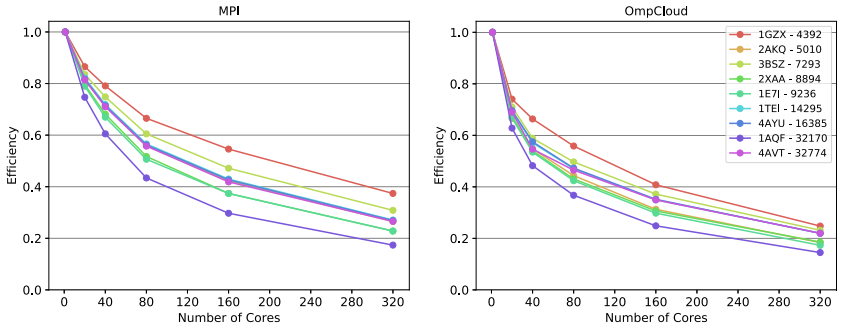
(h) Larger proteins on 320 cores

Fig. 7. Performance of the different implementations of HPCCS running on 16-node cluster for various sized proteins.

OpenMP annotations. Experimental results usually show good performance, but they also reveal the difficulties associated to MPI programming, which requires a level of expertise and platform knowledge that is far beyond the knowledge of typical programmers. Such drawbacks keep most programmers away from parallelizing their applications, constraining the computational power of the cloud cluster to a small set of expert programmers and specialized applications [12].



(a) MPI implementation is a clear winner for smaller proteins (less than 3000 atoms)



(b) OmpCloud implementation is competitive with MPI for larger proteins (more than 3000 atoms)

Fig. 8. Parallel efficiency of the different implementations of HPCCS running on 16-node cluster for various sized proteins.

Several other works have proposed to use directive-based programming for programming computer clusters. First, Nakao et al. proposed a new directive-based programming language similar to OpenMP but specialized to HPC clusters [58]. Their directives allow to micromanage parallelization and communication within the distributed architecture of the cluster; their custom compiler then translates pragmas into MPI calls. The OpenACC accelerator model is used to offload computation to a GPU within each node. If, on one hand, their work demonstrates very good scalability, close to handmade MPI implementation, then, on the other hand, their extensions do not follow the OpenMP standard and require information about the cluster architecture (e.g., the number of nodes), which reduces its portability. Second, Wottrich et al. [59] proposed a set of new OpenMP directives based on a Hadoop MapReduce framework to extend the standard towards cloud computing. Similarly to our work, they consider the offloading from local computer to the cloud. Although their approach was supported by a set of proofs-of-concept, their code transformation was performed by hand and did not include any evaluation of the communication overhead. Moreover, they defined a new syntax for mapping variables to the cloud, which is not compliant to the OpenMP and C standards. Next, Duran et al. introduced another directive-based programming language, called OmpSs [60], to program heterogeneous multi-core architectures. In fact, experiments was performed using OmpSs on multi-CPU (several sockets) / multi-GPU architectures [61]. OmpSs have also been extended to support distributed architecture [62, 63] using GASNet [64], a software infrastructure for Partitioned Global Address Space (PGAS) languages over

high-performance networks, generally used in supercomputing infrastructure. Their results show scalable performance quite on par with MPI. However, their methodology to support distributed systems with multiple address-spaces using multi-level task directives is programmatically complex. Finally, Jacob et al. introduced a new methodology relying on the OpenMP accelerator model to run applications on a cluster using the MPI infrastructure [25]: the host code is executed by the master node and the offloaded kernels by the worker nodes. Offloading to a set of worker nodes is then achieved by defining the offloaded kernel inside a *parallel loop* body. However, this requires various handmade modifications to applications, such as programmatically splitting (and merging) the offloaded data and defining nested DOALL loop to parallelize the execution on multicore worker nodes.

Most of the rest of the literature that strictly follows the OpenMP accelerator model studied GPU offloading and demonstrate good results [23, 24]. Nevertheless, some of them look for more untraditional targets like the Intel Xeon Phi platform or FPGAs [65]. Several interesting works also explore the usability of the programming model as well as the performance portability of the application over different platforms [66–68]. In particular, Hahnfeld et al. explore programming patterns to allow pipeline parallelism between host-device communication and computation [69].

Unlike previous works, our approach aims at enabling clusters as new OpenMP accelerators that can be accessed from the headnode of a cloud cluster or directly from the computer of the programmer, while respecting the OpenMP standard. We fully implemented our approach in existing tools allowing us to experiment it on a set of benchmarks and a real world application: we also analyzed the performance cost involved in cloud offloading and compare the performance with an MPI-based implementation. Last, contrary to most previous works, which rely on MPI, we rely on Spark a modern framework that has already been extensively used in the industry and is supported by a very dynamic community.

6 CONCLUSION

In this article, we addressed two problems: offloading computation to cloud infrastructures and programming distributed computer clusters to benefit from their quasi-unlimited parallel processing capabilities. We chose to base our methodology on a directive-based programming paradigm because of its simplicity, and especially on the OpenMP accelerator model, which is one of the most used parallel programming framework. To ease the utilization of the cloud infrastructure and computer clusters, we designed a runtime that offloads, maps and schedules computation automatically. Our approach allows portability over HPC clusters, commercial cloud services and private clouds. Indeed, by using a configuration file, our runtime is able to easily switch from one infrastructure to another without recompiling the program (assuming compatible instruction-sets). The communication with cloud storage services and the execution within the Spark cluster is handled automatically according to the given configuration. Experiments were performed on a set of benchmarks and a real scientific application described using the OpenMP accelerator model. Our results show that, because of data compression, the overhead induced by the offloading and the distributed computation heavily depends on the type of data processed by the application. However, promising performance was demonstrated by good speedups, reaching up to 115× on 256 cloud cores for the *2MM* benchmark using 1GB sparse matrices. Finally, our results show that OmpCloud implementation can be competitive in performance with MPI when running directly from the headnode of the cluster if there is sufficient computation workload. For example, on a more complex application resolving the Collision Cross-Section problem, our parallel implementation reached up to 80× on a 320 core cluster. In the future, we plan to implement data caching and pipeline parallelism (using OpenMP tasks) to limit the cost of host-target communications.

Additionally, we would like to explore the potential benefits of using polyhedral analysis and optimization within our workflow.

ACKNOWLEDGMENTS

The authors thank the anonymous referees for their valuable comments and helpful suggestions.

REFERENCES

[1] Hervé Yviquel and Guido Araujo. 2017. The cloud as an OpenMP offloading device. In *Proceedings of the 46th International Conference on Parallel Processing (ICPP'17)*.
[2] Leslie Lamport. 1974. The parallel execution of do loops. *Commun. ACM* 17, 2, 83–93.
[3] Ron Cytron. 1986. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the International Conference on Parallel Processing (ICPP'86)*.
[4] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. 2005. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th International Symposium on Microarchitecture (MICRO'05)*.
[5] John Randal Allen. 1983. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations*. PhD thesis, Rice University.
[6] Monica S. Lam and Michael E. Wolf. 2004. A data locality optimizing algorithm. *SIGPLAN Not.* 39, 4, 442–459.
[7] M. E. Wolf and M. S. Lam. 1991. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2, 4, 452–471.
[8] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplied data processing on large clusters. In *Proceedings of 6th Symposium on Operating Systems Design and Implementation*. 137–149.
[9] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. 1996. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Comput.* 22, 6, 789–828.
[10] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An industry standard API for shared-memory programming. *IEEE Comput. Sci. Eng.* 5, 1, 46–55.
[11] OpenMP. 2013. *OpenMP Application Program Interface*. Technical report.
[12] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2009. Above the clouds: A Berkeley view of cloud computing. Technical report, University of California, Berkeley.
[13] Ibrahim Hashem, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Ullah Khan. 2015. The rise of "big data" on cloud computing: Review and open research issues. *Info. Syst.* 47, 98–115.
[14] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*. 1–10.
[15] Cheng Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng, and Kunle Olukotun. 2007. Map-reduce for machine learning on multicore. *Adv. Neural Info. Process. Syst.* 19, 281.
[16] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly et al. 2010. The genome analysis toolkit: A mapreduce framework for analyzing next-generation DNA sequencing data. *Genome Res.* 20, 9, 1297–1303.
[17] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. 10.
[18] Timothy Hunter, Teodor Moldovan, Matei Zaharia, Samy Merzgui, Justin Ma, Michael J. Franklin, Pieter Abbeel, and Alexandre M. Bayen. 2011. Scaling the mobile millennium system in the cloud. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC'11)*. 1–8.
[19] Haitao Zhang, Jin Yan, and Yue Kou. 2016. Efficient online surveillance video processing based on spark framework. In *Proceedings of the Big Data Computing and Communications: Second International Conference (BigCom'16)*, Yu Wang, Ge Yu, Yanyong Zhang, Zhu Han, and Guoren Wang, Eds. 309–318. Springer International Publishing, Cham.
[20] Diego Teijeiro, Xoán C. Pardo, Patricia González, Julio R. Banga, and Ramón Doallo. 2016. Implementing Parallel Differential Evolution on Spark. In *Proceedings of the Applications of Evolutionary Computation: 19th European Conference (EvoApplications'16)*. Springer International Publishing, 75–90.
[21] Marek S. Wiewiorka, Antonio Messina, Alicja Pacholewska, Sergio Maffioletti, Piotr Gawrysiak, and Michal J. Okoniewski. 2014. SparkSeq: Fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision. *Bioinformatics* 30, 18, 2652–2653.
[22] OpenACC. 2013. The openacc application programming interface. Technical report.
[23] Marcio M. Pereira, Rafael C. F. Sousa, and Guido Araujo. 2017. Compiling and optimizing OpenMP 4.X programs to opencl and SPIR. In *Proceedings of the International Workshop on Scaling OpenMP for Exascale Performance and Portability*, vol. 10468, 48–61.

[24] Matt Martineau, Simon Mcintosh-smith, Carlo Bertolli, Arpith C. Jacob, Samuel F. Antao, Alexandre E. Eichenberger, Gheorghe-teodor Bercea, Tong Chen, Tian Jin, Kevin O. Brien, and Georgios Rokos. 2016. Performance analysis and optimization of clang's OpenMP 4.5 GPU support. In *Proceedings of the 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*.

[25] Arpith C. Jacob, Ravi Nair, Alexandre E. Eichenberger, Samuel F. Antao, Carlo Bertolli, Tong Chen, Zehra Sura, Kevin O'Brien, and Michael Wong. 2015. Exploiting fine- and coarse-grained parallelism using a directive-based approach. In *Lecture Notes in Computer Science*, vol. 9342, 30–41.

[26] Samuel F. Antao, Alexey Bataev, Arpith C. Jacob, Gheorghe-Teodor Bercea, Alexandre E. Eichenberger, Georgios Rokos, Matt Martineau, Tian Jin, Guray Ozen, Zehra Sura, Tong Chen, Hyojin Sung, Carlo Bertolli, and Kevin O'Brien. 2016. Offloading support for openMP in clang and LLVM. In *Proceedings of the 3rd Workshop on the LLVM Compiler Infrastructure in HPC*.

[27] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*. 75–86.

[28] L. Peter Deutsch. 1996. GZIP file format specification version 4.3.

[29] Jean-loup Gailly and Mark Adler. 2017. Zlib: A massively spiffy yet delicately unobtrusive compression library. http://zlib.net/.

[30] F. Pescador, M. Chavarrías, M. Garrido, J. Malagón, and C. Sanz. 2017. Real-time HEVC decoding with OpenHEVC and OpenMP. In *Proceedings of the IEEE International Conference on Consumer Electronics (ICCE'17)*. 370–371.

[31] Pedro Alonso, P. Vera-Candeas, Raquel Cortina, and José Ranilla. 2017. An efficient musical accompaniment parallel system for mobile devices. *J. Supercomput.* 73, 1, 343–353.

[32] Tyng Yeu Liang, Hung Fu Li, and Yu Chih Chen. 2016. An OpenMP programming environment on mobile devices. *Mobile Information Systems*, vol. 2016, Article ID 4513486, 24 pages.

[33] Blender: free and open source 3D creation suite. Retrieved 2018 from https://www.blender.org/.

[34] Milan Jaroš, Lubomír Říha, Petr Strakoš, Tomáš Karásek, Alena Vašatová, Marta Jarošová, and Tomáš Kozubek. 2015. Acceleration of blender cycles path-tracing engine using intel many integrated core architecture. *Lecture Notes Comput. Sci.* 9339, 86–97.

[35] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput.* 22, 6, 789–828.

[36] Jay P. Hoeflinger. 2006. Extending openmp to clusters. Intel Corporation white paper.

[37] Y. C. Hu, H. H. Lu, A. L. Cox, and W. Zwaenepoel. 2000. OpenMP for networks of SMPs. *J. Parallel Distrib. Comput.* 60, 12, 1512–1530.

[38] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2008. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. *Lecture Notes Comput. Sci.* 4959, 132–146.

[39] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Process. Lett.* 22, 4, 1250010.

[40] Louis-Noel Pouchet. 2015. PolyBench: The polyhedral benchmark suite. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

[41] Fernando Magno Quintao Pereira, Douglas do Couto Teixeira, Kézia Andrade, and Gleison Souza. 2015. MgBench: OpenACC benchmark suite. https://github.com/lashgar/ipmacc/tree/master/test-case/mgBench.

[42] M. F. Mesleh, J. M. Hunter, A. A. Shvartsburg, George C. Schatz, and M. F. Jarrold. 1996. Structural information from ion mobility measurements: Effects of the long-range potential. *J. Phys. Chem.* 100, 40, 16082–16086.

[43] Alexandre A. Shvartsburg and Martin F. Jarrold. 1996. An exact hard-spheres scattering model for the mobilities of polyatomic ions. *Chem. Phys. Lett.* 261, 1, 86–91.

[44] Leandro Zanotto, Gabriel Heerdt, Paulo C. T. Souza, Guido Araujo, and Munirv Skaf. High performance collision cross section calculation—HPCCS. *J. Comput. Chem.*

[45] Apache Livy: A REST Service for Apache Spark. Retrieved 2017 from https://livy.incubator.apache.org/.

[46] spark-jobserver: REST job server for Apache Spark. Retrieved 2018 from https://github.com/spark-jobserver/spark-jobserver.

[47] Karthik Kumar, Jibang Liu, Yung Hsiang Lu, and Bharat Bhargava. 2013. A survey of computation offloading for mobile systems. *Mobile Netw. Appl.* 18, 1, 129–140.

[48] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*. 49–62.

[49] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. 2012. ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. *Proceedings of the IEEE INFOCOM*. 945–953.

[50] M. S. Gordon, D. A. Jamshidi, and Scott Mahlke. 2012. COMET: Code offload by migrating execution transparently. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. 93–106.

[51] Marco V. Barbera, Sokol Kosta, Alessandro Mei, and Julinda Stefa. 2013. To offload or not to offload? the bandwidth and energy costs of mobile cloud computing. In *Proceedings of the IEEE INFOCOM*. 1285–1293.

[52] Krishna Nadiminti, Y. F. Chiu, Nick Teoh, Akshay Luther, Srikumar Venugopal, and Rajkumar Buyya. 2004. ExcelGrid: A.NET plug-in for outsourcing Excel spreadsheet workload to enterprise and global grids. In *Proceedings of the 12th International Conference on Advanced Computing and Communication*.

[53] David Abramson, Jack Dongarra, Eric Meek, Paul Roe, and Zhiao Shi. 2004. Simplified grid computing through spreadsheets and netsolve. In *Proceedings of the 7th International Conference on High Performance Computing and Grid in Asia Pacific Region (HPCAsia'04)*. 19–24.

[54] Milos Nikolić, Mroslav Hajduković, Dragan D. Milašinović, Danica Goleš, P. Marić, and Žarco Živanov. 2015. Hybrid MPI/OpenMP cloud parallelization of harmonic coupled finite strip method applied on reinforced concrete prismatic shell structure. *Adv. Eng. Softw.* 84, 55–67.

[55] Mroslav Hajduković, D. D. Milašinović, Danica Goleš, Milos Nikolić, P. Marić, Žarco Živanov, and P. S. Rakić. 2013. Cloud Computing-based MPI/OpenMP parallelization of the harmonic coupled finite strip method applied to large displacement stability analysis of prismatic shell structures. *Proceedings of the 3rd International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*. 1–21.

[56] Vojtech Nikl and Jiri Jaros. 2014. Parallelisation of the 3D fast fourier transform using the hybrid openmp/MPI decomposition. In *Mathematical and Engineering Methods in Computer Science*, vol. 8934, 100–112.

[57] Ronald D. Haynes and Benjamin W. Ong. 2014. MPI-OpenMP Algorithms for the parallel space-time solution of time dependent PDEs. In *Domain Decomposition Methods in Science and Engineering XXI*, 179–187.

[58] Masahiro Nakao, Hitoshi Murai, Takenori Shimosaka, Akihiro Tabuchi, Toshihiro Hanawa, Yuetsu Kodama, Taisuke Boku, and Mitsuhisa Sato. 2015. XcalableACC: Extension of XcalableMP PGAS language using OpenACC for accelerator clusters. *Proceedings of the 1st Workshop on Accelerator Programming Using Directives (WACCPD'14)*. 27–36.

[59] Rodolfo Wottrich, Rodolfo Azevedo, and Guido Araujo. 2014. Cloud-based OpenMP parallelization using a mapreduce runtime. In *Proceedings of the IEEE 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'14)*. 334–341.

[60] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. OmpSs: A proposal for programming heterogeneous multi-core architecture. *Parallel Process. Lett.* 21, 2, 173–193.

[61] Guray Ozen, Sergi Mateo, Eduard Ayguadé, Jesús Labarta, and James Beyer. 2016. Multiple target task sharing support for the OpenMP accelerator model. In *Proceedings of the 12th International Workshop on OpenMP: Memory, Devices, and Tasks (IWOMP'16)*. 268–280.

[62] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M. Badia, Eduard Ayguade, and Jesús Labarta. 2011. Productive cluster programming with OmpSs. In *Lecture Notes in Computer Science*, vol. 6852, 555–566.

[63] Javier Bueno, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. 2013. Implementing OmpSs support for regions of data in architectures with multiple address spaces. *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS'13)*. 359.

[64] Dan Bonachea and Jaein Jeong. 2002. Gasnet: A portable high-performance communication layer for global address-space languages. Technical report.

[65] Lukas Sommer, Jens Korinth, and Andreas Koch. 2017. OpenMP device offloading to FPGA accelerators. In *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*. 201–205.

[66] Leopold Grinberg, Carlo Bertolli, and Riyaz Haque. 2017. Hands on with OpenMP4.5 and unified memory: Developing applications for IBM's hybrid CPU+GPU systems (Part I). In *Proceedings of the International Workshop on Scaling OpenMP for Exascale Performance and Portability*. Vol. 10468, 3–16.

[67] Leopold Grinberg, Carlo Bertolli, and Riyaz Haque. 2017. Hands on with OpenMP4.5 and Unified Memory: Developing Applications for IBM's Hybrid CPU+GPU Systems (Part II). In *Proceedings of the International Workshop on Scaling OpenMP for Exascale Performance and Portability*. 17–29.

[68] Matt Martineau and Simon Mcintosh-smith. 2017. The productivity, portability and performance of openMP 4.5 for scientific applications targeting intel CPUs, IBM CPUs, and NVIDIA GPUs. In *Proceedings of the International Workshop on Scaling OpenMP for Exascale Performance and Portability*. Vol. 10468, 185–200.

[69] Jonas Hahnfeld, Tim Cramer, Michael Klemm, Christian Terboven, and S. M. Matthias. 2017. A Pattern for Overlapping Communication and Computation with OpenMP Target Directives. In *Proceedings of the International Workshop on Scaling OpenMP for Exascale Performance and Portability*. 325–337.

Received February 2018; revised May 2018; accepted May 2018