Efficient Execution of Dynamic Programming Algorithms on Apache Spark

Mohammad Mahdi Javanmard
Stony Brook University, USA
mjavanmard@cs.stonybrook.edu

Zafar Ahmad
Stony Brook University, USA
zafahmad@cs.stonybrook.edu

Jaroslaw Zola
University at Buffalo, USA
jzola@buffalo.edu

Louis-Noël Pouchet
Colorado State University, USA
pouchet@colostate.edu

Rezaul Chowdhury
Stony Brook University, USA
rezaul@cs.stonybrook.edu

Robert Harrison
Stony Brook University, USA
robert.harrison@stonybrook.edu

***Abstract*—One of the most important properties of distributed computing systems (e.g., Apache Spark, Apache Hadoop, etc) on clusters and computation clouds is the ability to *scale out* by adding more compute nodes to the cluster. This important feature can lead to performance gain provided *the computation (or the algorithm) itself can scale out.* In other words, the computation (or the algorithm) should be easily decomposable into smaller units of work to be distributed among the workers based on the hardware/software configuration of the cluster or the cloud. Additionally, on such clusters, there is an important trade-off between communication cost, parallelism, and memory requirement. Due to the scalability need as well as this trade-off, it is crucial to have a *well-decomposable, adaptive, tunable,* and *scalable* program. Tunability enables the programmer to find an optimal point in the trade-off spectrum to execute the program efficiently on a specific cluster. We design and implement well-decomposable and tunable dynamic programming algorithms from the Gaussian Elimination Paradigm (GEP), such as Floyd-Warshall's all-pairs shortest path and Gaussian elimination without pivoting, for execution on Apache Spark. Our implementations are based on *parametric multi-way recursive divide-&-conquer algorithms.* We explain how to map implementations of those grid-based parallel algorithms to the Spark framework. Finally, we provide experimental results illustrating the performance, scalability, and portability of our Spark programs. We show that offloading the computation to an OpenMP environment (by running parallel recursive kernels) within Spark is at least partially responsible for a $2-5\times$ speedup of the DP benchmarks.**

***Index Terms*—Dynamic Programming, Recursive Divide-&-Conquer, Distributed-memory Computing, Apache Spark, I/O-efficiency, Communication-efficiency, Polyhedral Compilation.**

I. Introduction

Dynamic programming (DP) is an algorithm design technique that uses memoization to improve the performance of the recursive solution to an optimization problem [1]–[4]. A DP algorithm outperforms the recursive solution by preventing the recomputation of overlapping subproblems. It computes each subproblem *exactly* once and stores the optimal solutions to the subproblems in a DP table to be reused for building an optimal solution for a larger problem. A DP algorithm can be viewed as trading off space for lower computation time [2].

DP is one of the core techniques used in solving a variety of combinatorial optimization problems [5], [6]. It has been widely used in many different applications, including economics [7], bioinformatics and computational biology [8], big data [9], [10] and machine learning [11], [12].

A wide class of DP algorithms can be represented in a general form shown in Fig. 1. It is called the Gaussian Elimination Paradigm (GEP) [13], [14]. In this form, a DP algorithm processes the underlying DP table whose entries, to be updated, are chosen from an arbitrary set S. The cell $c[i,j]$ gets updated based on the output of function $f: S\times S\times S\times S\rightarrow S$ over inputs $c[i,j]$, $c[i,k]$, $c[k,j]$, and $c[k,k]$. The set of updates the algorithm needs to perform is denoted by Σ_G. Floyd-Warshall's all-pairs shortest path (FW-APSP) [15], transitive closure [16], and Gaussian elimination without pivoting (GE) [3], [17] are among the most important examples of DP algorithms that fit into this form.

```
$GEP(c, 1, n)$

input: matrix $c[1..n, 1..n]$
1. for $k = 1$ to $n$ do
2.     for $i = 1$ to $n$ do
3.         for $j = 1$ to $n$ do
4.             if $\langle i, j, k\rangle \in \Sigma_G$
5.                 $c[i,j] = f(c[i,j], c[i,k], c[k,j], c[k,k])$
```

Fig. 1. The GEP form.

Apache Spark is a high-performance, scalable, general-purpose distributed computing system [18], [19]. It allows one to run batch, interactive and streaming jobs on a cluster using the *same* unified framework [19]. Through the high-level abstractions and easy-to-use APIs, it enables programmers to process large data sets, beyond what can fit into a single compute machine [20]. The abstractions and APIs are

This work is supported in part by NSF grants CCF-1725611, CCF-1750399, CMMI-1906344, OAC-1910539, CCF-1725428, CNS-1553510 and OAC-1931387.

978-1-7281-6677-3/20/$31.00 ©2020 IEEE
DOI 10.1109/CLUSTER49012.2020.00044

designed in a way that programmers can rapidly implement Spark applications that operate on datasets small enough to fit into one machine and then run the same applications on very large clusters and/or computational clouds without any significant modification [21].

Currently, Apache Spark is among the most active open-source projects in the Apache ecosystem [19]. Spark's in-memory execution model can lead to up to $100\times$ faster execution of the same job, compared to Hadoop's MapReduce [22], albeit with the cost of more memory consumption [23]. It has been shown that Spark implementations of Word Count, k-means and PageRank algorithms are $2.5\times$, $5\times$, and $5\times$ faster than their Hadoop's MapReduce counterparts, respectively [24]. This in-memory computing model can have huge impact on the performance of iterative algorithms that need to reuse data, including DP algorithms. As a result of improved efficiency of Apache Spark over Hadoop's MapReduce, researchers have been using Apache Spark as the core engine for solving problems in different areas, including machine learning and data mining [25], [26], big data and data science [27], [28], bioinformatics and computational biology [29]–[31].

Our Contributions. In this paper, for the class of DP problems in the GEP form (see Fig. 1), we provide *well-decomposable* and *easily tunable* implementations for execution on Apache Spark. Our implementations are based on *parametric multi-way recursive divide and conquer* (r-way $\mathcal{R}$-$\mathcal{DP}$), which has recursive hierarchical structure. It has already been shown that hierarchical algorithms are suitable for hierarchical architectures (including shared-memory as well as distributed-memory compute machines) [32]–[36]. In this paper, we show that the *parametric* and *recursive structure* of the r-way R-DP algorithms provides the tunability and adaptability needed for massively parallel execution on multi-level software stacks (e.g., Apache Spark), which is a common standard for execution of large-scale programs on clusters and computation clouds. Considering the state-of-the-art implementation [37] of FW-APSP algorithm in Apache Spark as a baseline for one of our benchmarks, we show that using r-way $\mathcal{R}$-$\mathcal{DP}$ kernels instead of iterative kernels and offloading the computation to an OpenMP environment (by running parallel recursive r-way $\mathcal{R}$-$\mathcal{DP}$ kernels) within Spark can lead to at least $2\times$ speedup.

The paper is organized as follows. Section II reviews the basic concepts in Apache Spark. Then Section III provides a literature survey on DP algorithms and their state-of-the-art implementations on different architectures such as shared-memory (multi- and manycores) and distributed-memory machines. It also provides a summary of existing research comparing HPC programming models and Big Data frameworks. Section IV summarizes design methodologies for r-way $\mathcal{R}$-$\mathcal{DP}$ algorithms [34]–[36], [38], and explains how to map the algorithms to the Spark framework. Section V provides detailed experimental results illustrating the performance, scalability, and portability of the Spark programs for different DP algorithms. Finally, Section VI concludes the paper.

II. Overview of Apache Spark

There are three important concepts in Apache Spark:

- **RDD (Resilient Distributed Datasets)**: RDD is the core data abstraction that represents lazily evaluated, compile-time type-safe, distributed collections [20]. RDD is further divided into several partitions and distributed among the executors. They are stored either in main memory or persistent storage. Each machine in the cluster contains one or more partitions. Spark keeps RDD partitions in memory on the compute nodes (executors) to reduce I/Os in repeated computations [20].
- **Transformation**: A transformation is a computation that takes *input* (or *parent*) RDDs and generates *output (or child)* RDDs. The RDD transformations are lazily evaluated, which means they get computed only when an action is triggered. From dependency perspective, transformations can be classified into two categories [39]:
  - **Transformation with narrow dependencies**: In such transformations, each partition of the input RDD is used by *at most one* partition of the output RDD. Thus, for executing them, there is no need for data shuffle across partitions and hence, they can be efficiently combined and executed together in one pass of the data.
  - **Transformation with wide dependencies**: In such transformations, multiple output partitions may depend on each partition in the input RDD. They are slow as they often require all or some data to be shuffled over the network[1]. Transformations with wide dependencies break down the Spark job into stages [20]. Each stage is a set of computations (i.e., a group of one or many transformations with narrow dependencies) that can be computed without a need for data shuffle. Corresponding to each partition of input RDDs the stage takes, there is one task. In Spark, task is considered as the smallest unit of execution. All such tasks which associate with one stage run the same code but on different partitions of the input RDDs [20]. The number of RDD partitions has a direct impact on the overall performance as it impacts the data distribution through the cluster and determines the number of tasks that will be executing RDD transformations [40]. The number of RDD partitions is configurable and by default, it is equal to the total number of cores on all executor nodes.
- **Action**: An action is a computation that takes input RDDs and generates the *final result*. Actions trigger the scheduler to build a directed acyclic graph (DAG). The edges of the DAG are determined by the dependencies between the transformations [20]. Spark uses this DAG to define a series of steps, called execution plan, to produce the final result. Lazy evaluation helps Spark to optimize the execution plan by grouping and fusing together the transformations to avoid doing multiple passes through the underlying data which leads to improved efficiency [20], [21].

[1]If Spark knows that the input RDD is already partitioned in a specific way, the transfomration might not cause data shuffle

III. Related Work

DP algorithms are specified using recurrence relations that determine how and in which order one should fill out the underlying DP table cells using the already-computed values of other cells. The most common approach to implement DP algorithms uses a loop-based program that populates the DP table cells iteratively. Such implementations often have good spatial locality[2] and benefit from prefetching optimizations. However, they do not perform efficiently due to the lack of temporal locality[3] [13], [41].

To prevail over the shortcomings of the loop-based DP algorithms, recursive divide-&-conquer DP algorithms (2-way $\mathcal{R}$-$\mathcal{DP}$) have been introduced [13], [14], [41]. The recursive structure of the algorithms prevents inefficient memory accesses and hence, provides the best opportunity to gain excellent temporal locality and consequently much better performance. Such algorithms execute iterative loop-based kernels after reaching sufficiently small base cases. Reordering the execution of instructions inside the kernels enables further optimizations such as vectorization, parallelization, etc. I/O efficiency can also be obtained by using an optimizing compiler transformation called tiling [42]. However, it has been shown that unlike the tiled programs, recursive divide-&-conquer DP algorithms are cache oblivious[4] [13], [43] and more importantly cache adaptive[5] [41], [44].

Although the standard 2-way (or any fixed r-way) $\mathcal{R}$-$\mathcal{DP}$ algorithms have several advantages over their simple loop-based counterparts, they have several shortcomings [34]–[36] that limit their suitability for distributed-memory heterogeneous clusters.

- They are not suitable for architectures that have limited support for recursion and require explicit memory instructions (either between the compute nodes, e.g., distributed-memory computation via MPI, or between different levels of the memory hierarchy, e.g., GPUs).
- One of the most important properties of distributed computing systems (e.g., Spark, Hadoop, etc.) on clusters and computation clouds is their ability to *scale out* by adding more compute nodes to the cluster. This important feature can lead to performance gain provided the computation itself can scale out. In other words, the computation should be easily decomposable into smaller units of work to be distributed among the workers based on the hardware/-software configuration of the cluster or the cloud. The standard 2-way (or any fixed r-way) $\mathcal{R}$-$\mathcal{DP}$ algorithms do such decompositions recursively with low precision often matching the required sizes only within a constant factor.
- In distributed cluster and cloud computing, a trade-off between communication cost, parallelism, and memory requirement is among the most important performance challenges. Due to the scalability need as well as this trade-off, it is crucial for the algorithm to be highly adaptive and tunable so that the programmer can find the optimal performance point (either on-the-fly by using adaptive runtime configuration selection or using estimates from hardware/software parameters using analytical models). Though standard 2-way (or any fixed r-way) $\mathcal{R}$-$\mathcal{DP}$ algorithms are known to be adaptive and can be tuned, we seek better precision and control.

The above limitations of 2-way $\mathcal{R}$-$\mathcal{DP}$ algorithms have led to the introduction and development of parametric r-way recursive divide-&-conquer DP algorithms (r-way $\mathcal{R}$-$\mathcal{DP}$) [34]–[36]. Section IV explains the design methodologies of r-way $\mathcal{R}$-$\mathcal{DP}$ algorithms which we use for our Spark implementations. In the rest of this section, we go over known research results on design and implementation of several DP algorithms for different architectures. We also include a brief literature survey on research comparing HPC programming models and Big Data frameworks.

GPU algorithms have been designed for solving several DP problems, e.g., Floyd-Warshall's APSP [45]–[49], the parenthesis problem family (including CYK algorithm, optimal polygon triangulation, RNA folding, etc.) [50]–[53], and sequence alignment [54]–[57]. Some results take advantage of the fact that Floyd-Warshall's APSP problem can be reduced to a matrix-matrix multiplication kernel, on a semiring, e.g., R-Kleene's algorithm [48], [58], [59].

Several researchers have introduced high performance distributed-memory graph algorithms [60]–[62] as well as dynamic programming algorithms in various application areas [61]–[69]. Wang et al. have introduced DPX10, a framework for generating distributed implementations of iterative DP programs [70]. DPX10 exploits node-level, and thread-level parallelism. Hegde et al. have extended AutoGen [41] to produce MPI-based distributed-memory implementations of DP algorithms [71]. A runtime system dynamically executes the 2-way recursive algorithm until the subproblem sizes become small enough to be assigned to other processes for execution. Graph algorithms on MapReduce have been introduced by several researchers [72]–[75]. Apache Spark provides frameworks such as GraphX [76] and GraphFrame [77] which can compute multi-source shortest-paths.

Schoeneman and Zola [37] have recently introduced an efficient implementation of the blocked all-pairs shortest-paths (FW-APSP) algorithm [78] for Apache Spark. They have shown the importance of tuning performance factors (e.g., block decomposition parameter, RDD partitioning granularity, etc). In addition, their experimental results indicate that a communication-efficient distributed-memory MPI implementation [62] outperforms their Spark implementation which reflects Spark's trade-off between programmer's productivity and scalability. Our work in the current paper improves over their FW-APSP solver by using r-way $\mathcal{R}$-$\mathcal{DP}$ algorithms

[2]i.e., whenever a cache block is brought into the cache, it contains as much useful data as possible.

[3]i.e., whenever a cache block is brought into the cache, as much useful work as possible is performed on this data before removing the block from the cache.

[4]i.e., they are I/O efficient among *every* two consecutive levels of the memory hierarchy without any specific tuning of the algorithm

[5]i.e., they can passively self-adapt to the changes in the size of available space in a cache shared with other threads and processes

[34]–[36], [38] as kernels instead of using iterative kernels, and additionally extends their solution, as a general framework, to a wider class of DP problems in GEP form.

Jha et al. [79] and Asaadi et al. [80] have presented a general comparison of the current HPC programming models (e.g., MPI, OpenMP, etc) and Big Data frameworks (e.g., Hadoop, Spark, etc). For several benchmarks from distributed graph algorithms and machine learning applications, including Latent Dirichlet Allocation, PageRank, Single Source Shortest Path, and Canonical Polyadic Decomposition, Anderson et al. have shown that offloading the computation to an MPI environment provides $3.1 - 17.7\times$ speedups [81]. Similarly, we show that offloading the computation to an OpenMP environment (by running parallel recursive r-way $\mathcal{R}$-$\mathcal{DP}$ kernels) within Spark is at least partially responsible for a $2 - 5\times$ speedup of the DP benchmarks.

IV. Parametric Multi-way Recursive Divide-&-Conquer DP (r-way $\mathcal{R}$-$\mathcal{DP}$) Algorithms

In this section, first we review how to transfrom a standard 2-way $\mathcal{R}$-$\mathcal{DP}$ into a parametric r-way $\mathcal{R}$-$\mathcal{DP}$ using an *inline and optimize* approach [34]–[36]. As an alternative approach, we also review how to *systematically* obtain an r-way $\mathcal{R}$-$\mathcal{DP}$ algorithm from the loop-based specification of a DP algorithm by using *polyhedral compiler transformations* [36], [38]. For the rest of the paper, we consider Gaussian Elimination without pivoting (GE) [3], [17] as a running example. Although GE is not usually considered a DP algorithm, it fits into the GEP framework. The GE algorithm is used in Linear Algebra to solve systems of linear equations [3], [17]. It is also used for solving LU decomposition of symmetric positive-definite or diagonally dominant real matrices [3], [17]. In the GE problem, a system of $(n-1)$ equations in $(n-1)$ unknowns $(x_1, x_2, ..., x_{n-1})$ is represented by an $n \times n$ matrix X, where each row represents an equation. For example, the pth row, represents the equation $\sum_{j=1}^{n-1}(X[p,j] \times x_j) = X[p,n]$. A loop-based implementation of GE is provided in Fig. 2.

```
LOOP_GE(X)

1. for $k = 1$ to $(n-1)$ do
2.   for $i = (k+1)$ to $n$ do
3.     for $j = k$ to $n$ do
4.       $X[i,j] -= (X[i,k] \times X[k,j])/X[k,k]$
```

Fig. 2. Iterative algorithm for Gaussian Elimination w/o pivoting (GE).

A. First design methodology: inline and optimize

In this section, we review the first methodology [34]–[36] for designing r-way $\mathcal{R}$-$\mathcal{DP}$ algorithms for several DP problems in the *fractal-DP* class [41]. The procedure starts by using either AutoGen [41] or Bellmania [82] to automatically obtain the standard 2-way $\mathcal{R}$-$\mathcal{DP}$ for the given DP problem. The 2-way algorithm is, indeed, a special case of the general 2^t-way algorithm, where $t = 1$. To generalize the algorithm to r-way $\mathcal{R}$-$\mathcal{DP}$ (where $r = 2^t$), starting from $t = 1$, the approach repeatedly applies the following two refinement steps until the general pattern representing the algorithm in a compact form is identifiable. These steps are used to derive the $2^{(t+1)}$-way algorithm from 2^t-way:

1) Take the 2^t-way $\mathcal{R}$-$\mathcal{DP}$ algorithm and inline each recursive function call by one level of recursion based on the 2-way $\mathcal{R}$-$\mathcal{DP}$ specification of the function. The *output* is an *inefficient* $2^{(t+1)}$-way $\mathcal{R}$-$\mathcal{DP}$ algorithm.
2) To execute the algorithm *as efficiently as possible* (i.e., in as few parallel stages as possible), move each function call to the lowest possible stage (i.e., the earliest time) without violating the following dependency constraints, where for function F, $W(F)$ denotes the DP subtable F writes to and $R(F)$ denotes the set of DP subtables F reads from:
   - If $W(F_1) \neq W(F_2)$ and $W(F_1) \in R(F_2)$, then F_1 must be executed before F_2 (denoted by $F_1 \rightarrow F_2$).
   - If $W(F_1) = W(F_2)$ and only F_1 is flexible, i.e., $W(F_1) \notin R(F_1)$, then F_1 must be executed before F_2 (denoted by $F_1 \rightarrow F_2$).
   - If $W(F_1) = W(F_2)$ and both F_1 and F_2 are flexible, then they can be executed in any order but not in parallel (denoted by $F_1 \longleftrightarrow F_2$)
   - If F_1 and F_2 do not satisfy any of the above rules, then they can be executed in parallel (denoted by $F_1 || F_2$).

Fig. 3 shows part of refining the 2-way GE algorithm and reducing the stages of parallelism. After some level of refinements, the pattern of function calls becomes apparent and the algorithm in a compact form can be represented. Fig. 4 shows the r-way $\mathcal{R}$-$\mathcal{DP}$ algorithm for GE.

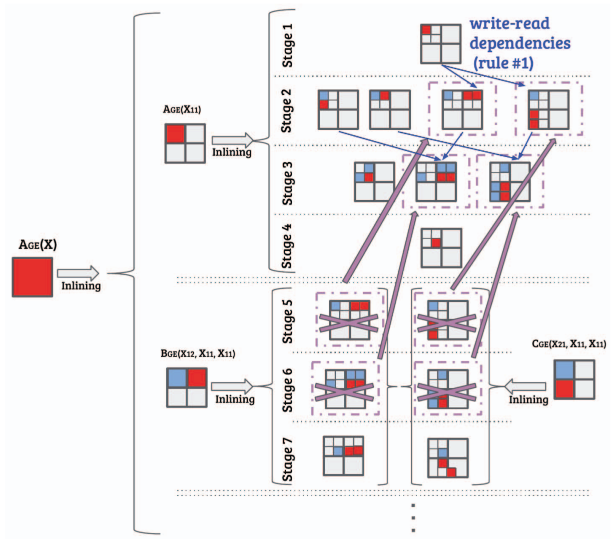

Fig. 3. Refining 2-way $\mathcal{R}$-$\mathcal{DP}$ of A_{GE} by one level. The functions in stages 5 and 6 are moved to stages 2 and 3 respectively.

It is worth mentioning that in the r-way $\mathcal{R}$-$\mathcal{DP}$, the problem size is always assumed to be divisible by the parameter r. In cases where it is not divisible, virtual padding can be applied to increase the problem size to the next smallest value which is divisible by r. Javanmard et al. [34]–[36] have

```
$A_{GE}(X, r)$
1. if X is a small matrix then loop-based $GE(X)$
2. else
3.   for $k \leftarrow 0$ to $(r-1)$ do
4.     $A_{GE}(X_{kk}, r)$
5.     parallel_for $i \leftarrow k+1$ to $(r-1)$ do
6.       $B_{GE}(X_{ki}, X_{kk}, X_{kk}, r)$, $C_{GE}(X_{ik}, X_{kk}, X_{kk}, r)$
7.     $D_{GE}(X_{lm}, X_{lk}, X_{km}, X_{kk}, r)$, par_for $l, m \in$ [k+1,r-1]
```

```
$D_{GE}(X, U, V, W, r)$
1. if X is a small matrix then loop-based $GE(X, U, V, W)$
2. else
3.   for $k \leftarrow 0$ to $(r-1)$ do
4.     $D_{GE}(X_{ij}, X_{ik}, V_{kj}, W_{kk}, r)$, par_for $i \in$ [0,r-1], $j \in$ [0,r-1]
```

```
$B_{GE}(X, U, W, r)$
1. if X is a small matrix then loop-based $GE(X, U, W)$
2. else
3.  for $k \leftarrow 0$ to $(r-1)$ do
4.    $B_{GE}(X_{kj}, U_{kk}, W_{kk}, r)$, par_for $j \in$ [0,r-1]
5.    $D_{GE}(X_{ij}, U_{ik}, X_{kj}, W_{kk}, r)$, par_for $i \in$ [k+1,r-1], $j \in$ [0,r-1]
```

```
$C_{GE}(X, V, W, r)$
1. if X is a small matrix then loop-based $GE(X, V, W)$
2. else
3.   for $k \leftarrow 0$ to $(r-1)$ do
4.     $C_{GE}(X_{ik}, V_{kk}, W_{kk}, r)$, par_for $i \in$ [0,r-1]
5.     $D_{GE}(X_{ij}, X_{ik}, V_{kj}, W_{kk}, r)$, par_for $j \in$ [k+1,r-1], $i \in$ [0,r-1]
```

Fig. 4. An r-way $\mathcal{R}$-$\mathcal{DP}$ for Gaussian elimination without pivoting (GE).

shown that these algorithms are theoretically and practically efficient on various architectures including multicores, many-cores and distribute-memory machines. They have provided experimental results for GPU and MPI-based distributed-memory algorithms as well as detailed discussions on how to choose an optimal value of r and how to map the algorithm to GPUs and distributed-memory machines.

B. Second design methodology: using polyhedral compiler transformations

In this section, we review the second methodology for designing r-way $\mathcal{R}$-$\mathcal{DP}$ algorithms using polyhedral compiler transformations [36], [38]. The procedure first applies a mono-parametric tiling transformation [83]. It has been proven that a mono-parametrically tiled program is a polyhedral program and hence standard polyhedral representation and analysis can be applied to it [84]. Such a program consists of a sequence of nested for loops, iterating over the tiles (called inter-tile loop nests) and inside each tile, iterating over the points inside the tile (called intra-tile loop nests). As the second step, the program is transformed into a recursive form [85], by (1) replacing the intra-tile loop nest with a recursive function call with proper loop bounds and recursion variables (e.g., base case size and the parameter r) and (2) including recursion base case. After applying the first two transformations, a single-function recursive program is generated which reads one or more input tiles and updates the output tile. However, by further analysis of the recursive function calls, we can identify the cases (i.e., the function calls) where the input tiles can fully overlap with or be entirely disjoint from the output tile.

As the third step, using index set splitting transformation [86] with the criteria of disjointness of the output tile from the input tiles, the inter-tile loop is decomposed into different cases with different degree of overlap or disjointness among the input and output tiles. For each such case, a new recursive function is introduced, with the purpose of providing further opportunity for the next transformation to produce a more efficient algorithm from the parallelism perspective: the more disjoint the input and output tiles are, the more relaxed the data dependencies among the tiles and hence, more parallelism across calls can be exploited. Each of the newly introduced recursive functions is processed in the same manner. The final step is applying data dependence analysis among the function calls [87] (whether there exists a loop-carried data dependence, and between each consecutive function calls whether there is any data dependence) and emitting a (doall and docross) parallel implementation.

C. Apache Spark r-way $\mathcal{R}$-$\mathcal{DP}$ Implementations

By carefully analyzing the r-way $\mathcal{R}$-$\mathcal{DP}$ algorithm, we find a general *iterative* pattern of recursive function calls (e.g., in Fig. 4 function A_{GE} first calls A_{GE} recursively, then calls functions B_{GE}, and C_{GE}, etc). This pattern indicates how to map them to the Apache Spark framework. Considering the top level recursive function (e.g., A_{GE}), we design an algorithm which executes the same sequence of recursive function calls. In Spark implementations, those functions are `ARecGE`, `BRecGE`, `CRecGE`, and `DRecGE`, which correspond to A_{GE}, B_{GE}, C_{GE}, and D_{GE}, respectively, in Fig. 4. Each iteration of the algorithm is further decomposed into stages and each stage consists of some function calls executing concurrently. In our implementations, the underlying DP table is decomposed into $r \times r$ tiles, where, for achieving better performance, the decomposition parameter r can be tuned, either on-the-fly by using adaptive runtime configuration selection or using estimates from hardware/software parameters based on analytical models. We use pair RDD [21] to represent the underlying DP, which is a mapping, whose key (i, j) (where $i, j \in [0, r-1]$) is the coordinate of the tile and whose value is the corresponding tile $DP(i, j)$ in Spark. It is worth mentioning that the grid-based DP table decomposition is different from the Spark's RDD partitioning scheme. Single RDD partition can have multiple DP blocks. For our benchmarks introduced in Section V-A, irrespective of whether the function kernels are recursive or iterative, we provide two implementations [37]:

- **In-Memory (IM) Implementation:** In each iteration of r-way $\mathcal{R}$-$\mathcal{DP}$ algorithms, since there are data dependencies between function kernels in one stage and the previous stage, each function kernel is responsible for making an updated output tile/block *as well as* making copies of the updated tile/block required for the function kernels in the next stage. For example, in an r-way GE algorithm, in iteration `k` (for `k<r`), function `ARecGE` updates the block with key `(k,k)` and makes $2 \times (r-k-1) + (r-k-1)^2$ copies of it to be used by functions `BRecGE` (which updates the block `DP(k,j)` using the input tile `DP(k,k)` for `j>k`), `CRecGE` (which updates the block `DP(i,k)` using the input tile `DP(k,k)` for `i>k`), and `DRecGE` (which updates the block `DP(i,j)` using the input tiles `DP(i,k)`, `DP(k,j)`, and `DP(k,k)` for `i>k` and `j>k`) in the next stages of the current iteration of the algorithm. These copies are, indeed, a mapping from (i,k) or (k,j) or (i,j) to the updated block `DP(k,k)`. By using a transformation `combineByKey()`, they are coupled with the block $DP(i,k)$ or $DP(k,j)$ or $DP(i,j)$ for executing functions `BRecGE` or `CRecGE` or `DRecGE`, respectively (e.g., the transformation `combineByKey()` makes a mapping from (i,k) to $[DP(k,k), DP(i,k)]$).

Although this implementation is completely based on Apache Spark features, because of using communication-expensive transformations with wide dependencies (e.g., `combineByKey()`), it has two *major* drawbacks, which in some benchmarks (with complicated and heavy communication patterns), can lead to a very poor performance:

1) Data shuffles can incur high communication cost.
2) It is also constrained by the size of the underlying SSDs [37]. This constraint is due to the fact that for executing transformations with wide dependencies, the intermediate data should be staged in the local SSDs before shuffling and hence, for large inputs or small inputs with many replicates, it can lead to a failure in the execution of the algorithm.

Listing 1 shows the In-Memory Spark implementation of the GE algorithm.

```
for k in range(0, r):
  # Stage 1: calling function A
  aBlocks = DP.filter(FilterA(k)).flatMap(ARecGE)
              .partitionBy(p,partitioner)
  # Stage 2: calling functions B and C
  abcBlocks = DP.filter(FilterA(k) or FilterB(k) or
                        FilterC(k))
                .union(aBlocks).combineByKey(..)
                .flatMap(BCRecGE)
                .partitionBy(p,partitioner)
  # Stage 3: calling functions D
  abcdBlocks = DP.filter(FilterD(k)).union(abcBlocks)
                 .combineByKey(..)
                 .mapPartitions(DRecGE)
                 .partitionBy(p,partitioner)
  # Preparation for the next iteration
  # prevBlocks: blocks not touched in this iteration
  prevBlocks = DP.filter(not(FilterA(k) or
                             FilterB(k) or
                             FilterC(k) or
                             FilterD(k)))
  DP = sc.union([prevBlocks,abcdBlocks])
         .partitionBy(p,partitioner)
```

Listing 1. In-Memory implementation of GE R-way R-DP algorithm.

- **Collect-Broadcast (CB) Implementation:** Unlike the IM implementation, instead of using transformations with wide dependencies, such as `combineByKey()`, we alleviate the costly data shuffle by using `collect()` which brings the required block(s) to the Spark's driver node. Then the driver node distributes the block(s) to the executors via shared persistent storage. So, the function kernels (e.g., `ARecGE`, etc) do not make any copy after making an updated output tile/block. In this way, we trade efficiency at the expense of using auxiliary storage. Listing 2 shows the Collect-Broadcast Spark implementation of GE.

In the Spark implementations, we use predicate functions `FilterX` (`X` $\in$ {`A`, `B`, `C`, `D`}) to extract the correct set of blocks to be updated. They are defined based on the lower bounds and upper bounds of the for loops in the function A_{GE} in Fig. 4. For example, `FilterD` is defined as `FilterD[((l,m),DP(l,m)),k]: return (l>k&&l<r&&m>k&&m<r)`. Functions `ARecGE`, `BCRecGE`, and `DRecGE` can be implemented iteratively or recursively using the *I/O efficient* r-way $\mathcal{R}$-$\mathcal{DP}$ algorithms [34]–[36]. As a result, we have two important parameters to tune: (1) the block decomposition parameter r used in the top level Spark program (e.g., in line 1 of Lst. 1, and Lst. 2), and (2) r_{shared} which is used for recursive kernels to execute r_{shared}-way recursive $\mathcal{R}$-$\mathcal{DP}$ in executors.

```
1  for k in range(0, r):
2    # Stage 1: calling function A
3    aBlock = DP.filter(FilterA(k)).map(ARecGE)
4    aBlock.collect().tofile()
5    # Stage 2: calling functions B and C
6    bcBlocks = DP.filter(FilterB(k) or FilterC(k))
7                 .map(BCRecGE)
8    bcBlockscollect = bcBlocks.collect()
9    for block in bcBlocks: block.tofile()
10   # Stage 3: calling functions D
11   dBlocks = DP.filter(FilterD(k)).map(DEecGE)
12   # Preparation for the next iteration
13   # prevBlocks: blocks not touched in this iteration
14   prevBlocks = DP.filter(not(FilterA(k) or
15                              FilterB(k) or
16                              FilterC(k) or
17                              FilterD(k)))
18   DP = sc.union([prevBlocks,aBlock,bcBlocks,dBlocks])
19          .partitionBy(p,partitioner)
```

Listing 2. Collect-Broadcast implementation of GE R-way R-DP algorithm.

V. Experimental Results

In this section, we present the performance, scalability and portability plots for our Spark implementations. Specifically, for the all-pairs shortest path problem [15], we compare our implementations with the state-of-the-art Spark implementation by Schoeneman and Zola [37] which contains only iterative kernels and works on undirected graphs. We extended their implementation to the more general case of directed graphs. Additionally, our implementation has the flexibility of having either iterative kernels or recursive (r_{shared}-way recursive $\mathcal{R}$-$\mathcal{DP}$ for different values of r_{shared}) kernels implemented in OpenMP. Our experimental results indicate that optimized I/O-efficient recursive kernels employed by Spark executors outperform the iterative kernels (i.e., the baselines). We also illustrate how the tunable parameters – (1) the block decomposition parameter r used in the top level Spark program, and (2) r_{shared} used inside the executors for recursive kernels – impact the overall performance.

OMP_NUM_THREADS \ executor-cores	32	16	8	4	2	1
2	381	387	425	461	771	1302
4	264	262	288	324	534	944
8	213	211	280	262	421	741
16	292	285	429	330	407	696
32	581	601	752	656	668	829

TABLE I
COMPARING PERFORMANCE OF GE BENCHMARK (IN SECONDS) FOR DIFFERENT COMBINATIONS OF EXECUTOR-CORES AND OMP_NUM_THREADS

OMP_NUM_THREADS \ executor-cores	32	16	8	4	2	1
2	339	347	451	696	1209	2233
4	310	310	334	508	864	1608
8	302	303	321	403	688	1274
16	318	323	342	410	591	1084
32	363	360	382	446	605	977

TABLE II
COMPARING PERFORMANCE OF FW-APSP BENCHMARK (IN SECONDS) FOR DIFFERENT COMBINATIONS OF EXECUTOR-CORES AND OMP_NUM_THREADS

A. Benchmarks

From the GEP class of DP algorithms, we consider the following two algorithms – one solving a graph problem and the other solving a problem from linear algebra:

- **Floyd-Warshall's All-Pairs Shortest Path (FW-APSP).** Floyd [15] introduced a DP algorithm that finds the shortest path between every pair of vertices in a directed graph. Warshall introduced an algorithm for finding transitive closure [16]. Later, Aho et al. introduced an algebraic structure, known as closed semiring, as a general framework for solving path problems in directed graphs [88]. They have introduced an algorithm for finding a set of all paths between each pair of vertices in a directed graph. Indeed, the algorithms introduced by Floyd and Warshall are specific cases of the problem in the closed semiring introduced by Aho et al. A directed graph is represented as $G = (V, E)$, where $V = \{v_1, v_2, ..., v_n\}$ contains the vertices of the graph and each edge (v_i, v_j) is labeled by an element $l(v_i, v_j)$ of some closed semiring $(S, \bigoplus, \bigodot, 0, 1)$. For $i, j \in [1, n]$ and $k \in [0, n]$, let $d^{(k)}[i, j]$ represent the *smallest* cost path from v_i to v_j with intermediate vertices from the set $\{v_1, v_2, ..., v_k\}$. Then $d^{(n)}[i, j]$ is the cost of the *shortest path* from v_i to v_j. Assuming $d^0[i, j]$ = $l(v_i, v_j), i \neq j$ and $d^0[i, i] = 1$, the following recurrence computes all $d^{(k)}[i, j]$ for all $k > 0$:
$$d^{(k)}[i, j] = d^{(k-1)}[i, j] \bigoplus (d^{(k-1)}[i, k] \bigodot d^{(k-1)}[k, j])$$
The FW-APSP algorithm computes over a particular closed semiring $(\mathbb{R}, min, +, +\infty, 0)$. Fig. 5 represents the loop-based iterative implementation of FW-APSP.

```
LOOP_FW − APSP(X)

1. for $k = 1$ to $n$ do
2.   for $i = 1$ to $n$ do
3.     for $j = 1$ to $n$ do
4.       $d[i, j] = d[i, j] \bigoplus (d[i, k] \bigodot [k, j])$
```

Fig. 5. Iterative algorithm for FW-APSP problem.

FW-APSP is among the most fundamental graph algorithms. It has applications in various scientific and engineering areas, including logic programming [89], optimizing compilers [90], verification and model-checking tools [91], software-defined computer networks [92], transportation research [93]–[95], wireless sensor networks [96], etc.

- **Gaussian Elimination without Pivoting (GE).** GE is among the most important linear algebra algorithms which is used to solve systems of linear equations and LU decomposition of symmetric positive-definite or diagonally dominant real matrices [3], [17]. Further explanation of GE algorithm was provided in Section IV.

B. Experimental Setup

For each of the above benchmarks, we have four different implementations: (1) IM implementation with the iterative kernel, (2) IM implementation with recursive r-way $\mathcal{R}$-$\mathcal{DP}$ kernel, (3) CB implementation with the iterative kernel, and (4) CB implementation with recursive r-way $\mathcal{R}$-$\mathcal{DP}$ kernel.

We have used Spark 2.2.0 pySpark and Python 2.7. The iterative kernels are implemented using Python and Numba v0.47 JIT compiler [97]. The iterative kernels are offloaded to bare-metal via Numpy 1.16.6 and SciPy. The recursive kernels are implemented in C and OpenMP [98]. They are compiled by gcc 7.3.0 with `-Ofast` and `--march=native` optimization flags to generate shared objects and called by the executors using Cython 0.29.14 [99], [100].

The experiments were carried out on a standalone Apache Spark cluster with 16 compute nodes, which are interconnected through GbE network. For weak scalability plots (Fig. 9), we used a cluster with 1, 8, or 64 compute nodes. Each node in the cluster had two 16-core Intel Skylake (Intel Xeon Gold 6130 2.10GHz) processors with 32KB L1 and 1024KB L2 caches, and 192GB of RAM. Each node had one standard 1TB SSD drive. We allocated one executor per compute node. Additionally, inside each compute node, the Spark executor used all the cores. The implementation is available at [101]

C. Performance Results

Irrespective of the type of kernels (iterative or recursive) used, for distributed-memory parallel computations we solely rely on the capabilities of Apache Spark. For parallelism inside each compute node (i.e., shared-memory parallelism), (1) we have one executor per node and set *multiple* tasks (or *logical* cores) to run in parallel inside *each* executor (by setting the parameter `executor-cores` to the total number of *physical* cores in each compute node), and (2) we set the total number of RDD partitions to a value *larger* than the total number of *physical* cores in the cluster

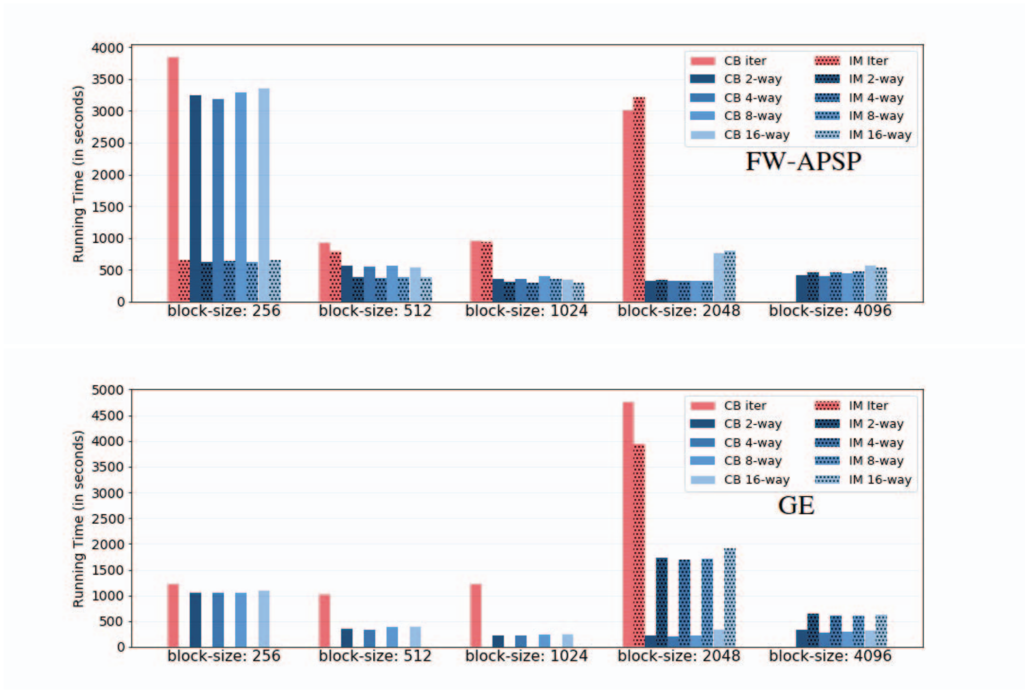

Fig. 6. Various Spark implementation of our benchmarks.

(based on Spark guidelines [102], $2\times$ to $4\times$ the total number of cores, and we chose to have $2\times$ the total number of cores, similar to [37]). However, for recursive kernels, since we used OpenMP, we have one more key performance factor: `OMP_NUM_THREADS`. As a result, for shared-memory parallelism we had to check various combinations of `executor-cores` and `OMP_NUM_THREADS`.

Table I shows excerpt of execution of GE (on a 16-node cluster), for problem size $32K \times 32K$ with blocks of size $1K \times 1K$, using a CB implementation with recursive 4-way $\mathcal{R}$-$\mathcal{DP}$ kernels. Table II shows the same for FW-APSP but using an IM implementation with recursive 16-way $\mathcal{R}$-$\mathcal{DP}$ kernels. These results confirm that for any specific value of `executor-cores`, increasing the value of `OMP_NUM_THREADS` can improve the overall performance up to some point but increasing further results in thread oversubscription (similar to [103]) which leads to drastic degradation of performance. As a result, For the benchmarks with parallel recursive kernels, to ensure that we have obtained the best performance to report, we set the `executor-cores` to the total number of *physical* cores in each compute node but executed the benchmarks with various values for `OMP_NUM_THREADS` and reported the one producing the best performance.

In addition to the performance factors explained above, there are other important factors to explore, which are: (1) different types of kernels (iterative and recursive), (2) the block decomposition parameter r used in the top level Spark program (which determines the block size executors are processing), and (3) r_{shared} used inside the executors for recursive kernels (which determines the recursive fan-out and hence coarse-grained parallelism as well as the sub-block size for the recursive kernels).

For each of the benchmarks, we ran several experiments for block sizes in $\{256, 512, 1024, 2048, 4096\}$ (i.e., for the problem size $32K \times 32K$, having blocking factor $r \in \{64, 32, 16, 8\}$). For r_{shared}-way $\mathcal{R}$-$\mathcal{DP}$ recursive kernels, we tried r_{shared} values from $\{2, 4, 8, 16\}$ (with various `OMP_NUM_THREADS`). We fixed the following parameters:

- We set the problem size to $32K \times 32K$, i.e., the underlying DP table has size $32K \times 32K$.
- We set the number of executors to 16 (we ran the experiments on a cluster with 16 compute nodes).
- We set the executor/driver memory to 160GB.
- We set the number of RDD partitions to 1024 which is $2\times$ total number of cores $= 2\times$ number of compute nodes $\times$ (number of cores in each compute node) $= 2 \times 16 \times 32 = 1024$. To minimize the frequency of Spark shuffles, blocks to be processed should be assigned to the same partitions, and partitions should be evenly distributed among executors [102]. However, due to the probabilistic nature of the default partitioner, there is no such guarantee. As a result, to have a better load balance, the ratio between the total number of RDD partitions and the number of available cores should be more than one. That's why, we decided to have $2\times$ the total number of cores as the number of RDD partitions.
- We used Spark's default partitioner.

Fig. 6 compares various Spark implementations of our benchmarks (in seconds). The missing bars (e.g., IM 16-way for FW-APSP) indicate experiments taking more than

8 hours to run[6]. In our FW-APSP experiments, unlike the GE benchmark, IM implementations outperformed CB implementations in most of the cases. There are two main reasons for such behavior:

- In FW-APSP, the dependencies among the kernels are much lighter and simpler than the GE benchmark and hence, in IM implementations, making multiple copies of the blocks and distributing them among the consumer kernels (in the next stage, i.e, kernels B and C) is not a bottleneck for the overall computation (see Fig. 7). However, in the GE benchmark, as kernel A has to copy the block it just updated to almost all other kernels B, C, and D and redistributes it, the dependency pattern among the kernels are much heavier and more complicated. Hence, making multiple copies of the blocks and distributing them becomes a huge performance bottleneck for the overall performance in IM implementations for small block sizes (the smaller the block size, the more blocks exist and hence, more time-consuming the block copy and distribution process). A CB implementation of the GE benchmark alleviates this problem at the expense of using auxiliary storage.
- In an IM implementation, there are multiple executions of the partitioner, unlike in a CB implementation which has one execution of the partitioner at the end of each iteration of the outermost loop which can lead to a potentially better load-balanced execution (see Lst. 1 and Lst. 2).

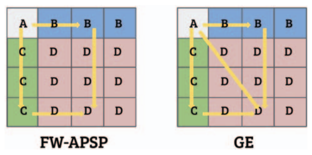

Fig. 7. Data dependencies among kernels are shown with arrows.

In the FW-APSP benchmark, for the iterative kernel, the best running time of 651 seconds is obtained by the IM execution with block size 256. However, the best running time with the recursive kernel is for the IM execution with 16-way recursive kernel with block size 1024 and 8 OMP_NUM_THREADS, which is 302 seconds (2.1$\times$ faster).

For the GE benchmark, as explained above, CB implementations outperformed the IM implementations. Additionally, for iterative kernels, the best running time of 1032 seconds is obtained by the CB execution with block size 512. However, the best running time with the recursive kernel is obtained by the CB execution with a 4-way recursive kernel with block size 2048 and OMP_NUM_THREADS= 16, which is 204 seconds (5$\times$ faster). Looking at the running time patterns for the iterative kernels (in both benchmarks), we observe that the block size directly impacts the level of available parallelism, and the level of data movement. Too large a block

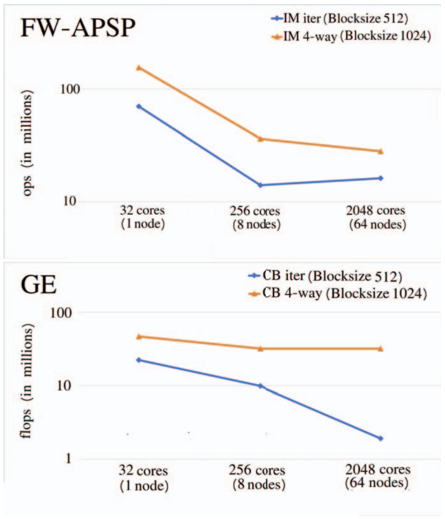

Fig. 9. Weak scaling of benchmarks FW-APSP and GE.

size may serialize the Spark execution or make staging data blocks in the auxiliary storage a performance bottleneck. On the other hand, very small blocks lead to huge overheads due to task scheduling and data shuffling. As a result, the block size should be selected carefully. Another important takeaway from the figure is that for small block sizes (i.e., 512) since the blocks fit in the L2 cache, performance of iterative kernels and recursive kernels are similar. However, for larger block sizes (i.e., 1024 and beyond), since the blocks do not fit in L2, the recursive kernels significantly outperform the iterative kernels.

To generate weak scalability plots, for FW-APSP, we set work per compute node (i.e., $\frac{N^3}{p}$) to the fixed value of $4K$ and for the GE benchmark, we set work per compute node to the fixed value of $8K$. We chose some of the efficient configurations for both kernels. For FW-APSP, the first configuration was an IM execution with an iterative kernel using block size 512, and the second was an IM execution with 4-way recursive kernel with block size 1024 and OMP_NUM_THREADS= 8. For GE, the first configuration was a CB execution with an iterative kernel using block size 512, and the second configuration was a CB execution with a 4-way recursive kernel using block size 1024 and OMP_NUM_THREADS= 8. Fig. 9 illustrates weak scalability of various kernels. As the Figure shows, the 4-way recursive kernel CB execution of GE scales better than its iterative kernel CB execution.

Additionally, in order to evaluate performance portability and robustness of our implementations, we used another standalone Apache Spark cluster with 16 compute nodes and repeated the FW-APSP experiments. In this cluster, each node had dual 10-core Intel Haswell (Xeon E5-2650v3 2.30GHz) processors and 64GB of RAM. All nodes were interconnected via a standard GbE network. The local storage in each node was deployed on 7500rpm SATA *spinning hard drives*. For this cluster, we fixed the following parameters:

- We set the number of executors to 16.
- We set the problem size to $32K \times 32K$, i.e., the underlying

[6]For the readability of the plots, we intentionally omit the bars for the iterative kernels with block size 4096. For the FW-APSP benchmark, the IM implementation took 14530 seconds and the CB implementation took 14480 seconds. For the GE benchmark, the IM and the CB implementations took 11344 seconds 15548 seconds, respectively.

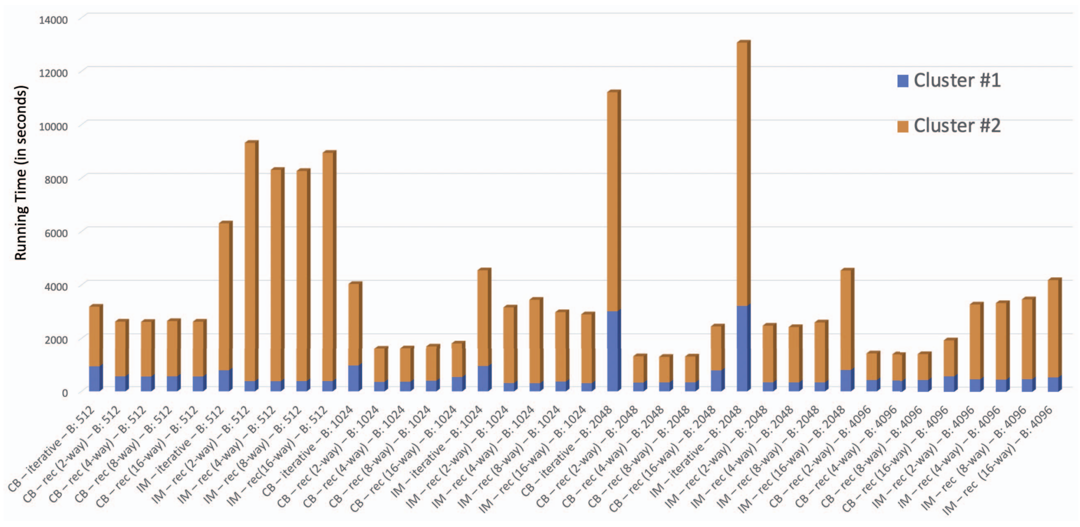

Fig. 8. Comparing performance of FW-APSP benchmark in two different clusters

DP table has size $32K \times 32K$.

- We set the number of RDD partitions to 640, which is $2\times$ the total number of cores = $2 \times 16 \times 20 = 640$.
- We set executor/driver memory to 60GB.

Fig. 8 provides the performance comparison between the two clusters. The missing bars (e.g., CB and IM iterative kernel executions for block size 4096) indicate time-out, mainly for the weaker cluster which is cluster #2, as our experiments were constrained to terminate within 8 hours. Results in Fig. 8 confirm that for the Spark implementations, if block decomposition factor r as well as r_{shared} for recursive kernels are chosen independent of the system configuration, the resulting implementation can be very inefficient. For example, on the first cluster, the IM implementation with 4-way recursive kernel with block size 1024 runs in 302 seconds (almost the best performance obtained in the first cluster). However, the same combination of parameters on the second cluster takes 3144 seconds which is $3.3\times$ slower than the best running time obtained with the second cluster (951 seconds). As a result, to run efficiently on a given cluster, the value of these two important parameters may need to be different based on the hardware/software configuration (including the number of compute nodes, cores per compute node available, and the sizes of memory/caches at different levels of the memory hierarchy).

VI. Conclusion

In this paper, we proposed and implemented r-way $\mathcal{R}$-$\mathcal{DP}$ algorithms from the Gaussian Elimination Paradigm using the Apache Spark framework. We summarized different design methodologies to obtain the algorithms. We provided several implementations for each of the benchmarks including IM and CB implementations with iterative and recursive kernels. We learned that the dependency structure among the kernels determines whether the IM or the CB implementation performs better. For shared-memory parallel implementations, in order to obtain high-performance, we emphasized the importance of finding the right combination of `executor-cores` and `OMP_NUM_THREADS`, and the value of r_{shared} for parallel recursive r_{shared}-way $\mathcal{R}$-$\mathcal{DP}$ kernels. Additionally, we illustrated how the tunable block decomposition parameter r used in the top level Spark program impacts the overall performance. Experiments on two different clusters suggest that the recursive kernels are more robust than iterative kernels under changes in the amount of available memory. Such experiments also reveal the importance of tuning the algorithmic (i.e., r and r_{shared}) and runtime parameters (`executor-cores` and `OMP_NUM_THREADS`) to find the high-performance execution. In summary, experimental results illustrate the importance of scalability, tunability and adaptability of r-way $\mathcal{R}$-$\mathcal{DP}$ algorithms which make them great candidates for heterogeneous supercomputers, clusters and computation clouds.

Some potential future directions extending our current work are as follows.

- We would like to extend the framework to include other data-intensive DP algorithms (beyond GEP) and later, beyond DP algorithms, which can benefit from the recursive divide-and-conquer structure of r-way $\mathcal{R}$-$\mathcal{DP}$ algorithms.
- In this work, we have used Apache Spark's default partitioner. However, the dependency structure among the kernels provides an opportunity to design and implement highly-efficient custom partitioners. We will design and implement such partitioners next.

Acknowledgments. The authors would like to acknowledge support provided by the Center for Computational Research at the University at Buffalo. The authors would like to thank anonymous reviewers for valuable comments that have significantly improved the paper.

References

[1] R. Bellman *et al.*, “The theory of dynamic programming,” *Bulletin of the American Mathematical Society*, vol. 60, no. 6, pp. 503–515, 1954.

[2] S. S. Skiena, *The algorithm design manual: Text.* Springer, 1998, vol. 1.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[4] M. Sniedovich, *Dynamic programming: foundations and principles.* CRC press, 2010.

[5] V. T. Paschos, *Concepts of Combinatorial Optimization.* John Wiley & Sons, 2014.

[6] D. A. Pierre, *Optimization theory with applications.* Courier Corporation, 1986.

[7] J. Rust, “Numerical dynamic programming in economics,” *Handbook of Computational Economics*, vol. 1, pp. 619–729, 1996.

[8] D. Gusfield, *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge university press, 1997.

[9] K. Gai, M. Qiu, M. Liu, and Z. Xiong, “In-memory big data analytics under space constraints using dynamic programming,” *Future Generation Computer Systems*, vol. 83, pp. 219–227, 2018.

[10] H. Zhao, M. Qiu, M. Chen, and K. Gai, “Cost-aware optimal data allocations for multiple dimensional heterogeneous memories using dynamic programming in big data,” *Journal of Computational Science*, vol. 26, pp. 402–408, 2018.

[11] D. Eaton and K. Murphy, “Bayesian structure learning using dynamic programming and mcmc,” *arXiv preprint arXiv:1206.5247*, 2012.

[12] A. P. Singh and A. W. Moore, “Finding optimal Bayesian networks by dynamic programming,” 2004.

[13] R. A. Chowdhury and V. Ramachandran, “Cache-oblivious dynamic programming,” in *SODA’06*, 2006, pp. 591–600.

[14] ——, “The cache-oblivious gaussian elimination paradigm: theoretical framework and experimental evaluation,” in *Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures.* ACM, 2006, pp. 236–236.

[15] R. W. Floyd, “Algorithm 97: shortest path,” *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[16] S. Warshall, “A theorem on boolean matrices,” in *Journal of the ACM.* Citeseer, 1962.

[17] R. A. Chowdhury and V. Ramachandran, “The cache-oblivious Gaussian Elimination Paradigm: theoretical framework, parallelization and experimental evaluation,” *TCS*, vol. 47, no. 4, pp. 878–919, 2010.

[18] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[19] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, “Apache Spark: a unified engine for big data processing,” *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[20] H. Karau and R. Warren, *High performance Spark: best practices for scaling and optimizing Apache Spark.* ” O’Reilly Media, Inc.”, 2017.

[21] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning Spark: lightning-fast big data analysis.* ” O’Reilly Media, Inc.”, 2015.

[22] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, “Shark: Sql and rich analytics at scale,” in *SIGMOD’13.* ACM, 2013, pp. 13–24.

[23] L. Gu and H. Li, “Memory or time: Performance evaluation for iterative operation on Hadoop and Spark,” in *HPCC’13 and EUC’13.* IEEE, 2013, pp. 721–727.

[24] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan, “Clash of the titans: Mapreduce vs. Spark for large scale data analytics,” *VLDB Endowment*, vol. 8, no. 13, pp. 2110–2121, 2015.

[25] S. Ramírez-Gallego, H. Mouriño-Talín, D. Martínez-Rego, V. Bolón-Canedo, J. M. Benítez, A. Alonso-Betanzos, and F. Herrera, “An information theory-based feature selection framework for big data under Apache Spark,” *IEEE TSMC: Systems*, vol. 48, no. 9, pp. 1441–1453, 2017.

[26] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, “Mllib: Machine learning in Apache Spark,” *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[27] J. G. Shanahan and L. Dai, “Large scale distributed data science using Apache Spark,” in *KDD’15.* ACM, 2015, pp. 2323–2324.

[28] S. Salloum, R. Dautov, X. Chen, P. X. Peng, and J. Z. Huang, “Big data analytics on Apache Spark,” *International Journal of Data Science and Analytics*, vol. 1, no. 3-4, pp. 145–164, 2016.

[29] A. Bahmani, A. B. Sibley, M. Parsian, K. Owzar, and F. Mueller, “Sparkscore: leveraging Apache Spark for distributed genomic inference,” in *IPDPSW’16.* IEEE, 2016, pp. 435–442.

[30] B. Xu, C. Li, H. Zhuang, J. Wang, Q. Wang, and X. Zhou, “Efficient distributed smith-waterman algorithm based on Apache Spark,” in *CLOUD’17.* IEEE, 2017, pp. 608–615.

[31] F. Jiang, C. K. Leung, O. A. Sarumi, and C. Y. Zhang, “Mining sequential patterns from uncertain big DNA in the Spark framework,” in *BIBM’16.* IEEE, 2016, pp. 874–881.

[32] D. E. Keyes, H. Ltaief, and G. Turkiyyah, “Hierarchical algorithms on hierarchical architectures,” *Philosophical Transactions of the Royal Society A*, vol. 378, no. 2166, p. 20190055, 2020.

[33] R. Carratalá-Sáez, M. Faverge, G. Pichon, G. Sylvand, and E. Quintana-Ortí, “Tiled algorithms for efficient task-parallel h-matrix solvers,” in *21st IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2020)*, 2020.

[34] M. M. Javanmard, P. Ganapathr, R. Das, Z. Ahmad, S. Tschudi, and R. Chowdhury, “Toward efficient architecture-independent algorithms for dynamic programs: poster,” in *PPoPP’19.* ACM, 2019, pp. 413–414.

[35] M. M. Javanmard, P. Ganapathi, R. Das, Z. Ahmad, S. Tschudi, and R. Chowdhury, “Toward efficient architecture-independent algorithms for dynamic programs,” in *ISC HPC’19.* Springer, 2019, pp. 143–164.

[36] M. M. Javanmard, “Parametric multi-way recursive divide-and-conquer algorithms for dynamic programs,” Ph.D. dissertation, State University of New York at Stony Brook, 2020.

[37] F. Schoeneman and J. Zola, “Solving all-pairs shortest-paths problem in large graphs using Apache Spark,” in *ICPP’19.* ACM, 2019, p. 9.

[38] M. M. Javanmard, Z. Ahmad, M. Kong, L.-N. Pouchet, R. Chowdhury, and R. Harrison, “Deriving parametric multi-way recursive divide-and-conquer dynamic programming algorithms using polyhedral compilers,” in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020, pp. 317–329.

[39] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *NSDI’12.* USENIX Association, 2012, pp. 2–2.

[40] P. Zecevic and M. Bonaci, *Spark in Action.* Manning Publications Co., 2016.

[41] R. Chowdhury, P. Ganapathi, J. J. Tithi, C. Bachmeier, B. C. Kuszmaul, C. E. Leiserson, A. Solar-Lezama, and Y. Tang, “Autogen: Automatic discovery of cache-oblivious parallel recursive algorithms for solving dynamic programs,” in *ACM SIGPLAN Notices*, vol. 51, no. 8, p. 10.

[42] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, “A practical automatic polyhedral parallelizer and locality optimizer,” in *PLDI’08*, vol. 43, no. 6. ACM, 2008.

[43] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, “Cache-oblivious algorithms,” in *FOCS’1999.* IEEE, 1999.

[44] M. A. Bender, R. Ebrahimi, J. T. Fineman, G. Ghasemiesfeh, R. Johnson, and S. McCauley, “Cache-adaptive algorithms,” in *SODA’14*, 2014.

[45] P. Harish and P. Narayanan, “Accelerating large graph algorithms on the GPU using CUDA,” in *HiPC’07*, 2007, pp. 197–208.

[46] G. J. Katz and J. T. Kider Jr, “All-pairs shortest-paths for large graphs on the GPU,” in *GH08*, 2008, pp. 47–55.

[47] K. Matsumoto, N. Nakasato, and S. G. Sedukhin, “Blocked all-pairs shortest paths algorithm for hybrid CPU-GPU system,” in *HPCC’11*, 2011, pp. 145–152.

[48] A. Buluç, J. R. Gilbert, and C. Budak, “Solving path problems on the GPU,” *Parallel Computing*, vol. 36, no. 5, pp. 241–253, 2010.

[49] B. Lund and J. W. Smith, “A multi-stage cuda kernel for floyd-warshall,” *arXiv preprint arXiv:1001.4108*, 2010.

[50] P. Steffen, R. Giegerich, and M. Giraud, “GPU parallelization of algebraic dynamic programming,” in *PPAM’09*, 2009, pp. 290–299.

[51] G. Rizk and D. Lavenier, “GPU accelerated RNA folding algorithm,” in *ICCS’09*, 2009.

[52] S. Solomon and P. Thulasiraman, “Performance study of mapping irregular computations on GPUs,” in *IPDPSW’10*, 2010, pp. 1–8.

[53] K. Nishida, K. Nakano, and Y. Ito, “Accelerating the dynamic programming for the optimal polygon triangulation on the gpu,” in *ICA3PP’12*, 2012, pp. 1–15.

[54] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig, “Bio-sequence database scanning on a GPU,” in *IPDPS’06*, 2006, pp. 8–pp.

[55] W. Liu, B. Schmidt, G. Voss, and W. Muller-Wittig, "Streaming algorithms for biological sequence alignment on GPUs," *TPDS'07*, vol. 18, no. 9, pp. 1270–1281, 2007.

[56] G. M. Striemer and A. Akoglu, "Sequence alignment with GPU: Performance and design challenges," in *IPDPS'09*, 2009, pp. 1–10.

[57] S. Xiao, A. M. Aji, and W.-c. Feng, "On the robust mapping of dynamic programming onto a graphics processing unit," in *ICPADS'09*, 2009, pp. 26–33.

[58] J. F. Sibeyn, "External matrix multiplication and all-pairs shortest path," *IPL'04*, vol. 91, no. 2, 2004.

[59] P. D'Alberto and A. Nicolau, "R-Kleene: A high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks," *Algorithmica*, vol. 47, no. 2, 2007.

[60] H. Meyerhenke, P. Sanders, and C. Schulz, "Parallel graph partitioning for complex networks," *TPDS'17*, vol. 28, no. 9, pp. 2625–2638, 2017.

[61] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing: design and analysis of algorithms.* Benjamin/Cummings Redwood City, 1994, vol. 400.

[62] E. Solomonik, A. Buluc, and J. Demmel, "Minimizing communication in all-pairs shortest paths," in *IPDPS'13.* IEEE, 2013, pp. 548–559.

[63] J.-F. Jenq and S. Sahni, "All pairs shortest paths on a hypercube multiprocessor," 1987.

[64] V. Kumar and V. Singh, "Scalability of parallel algorithms for the all-pairs shortest-path problem," *Journal of Parallel and Distributed Computing*, vol. 13, no. 2, pp. 124–138, 1991.

[65] M. B. Habbal, H. N. Koutsopoulos, and S. R. Lerman, "A decomposition algorithm for the all-pairs shortest path problem on massively parallel computer architectures," *Transportation Science*, vol. 28, no. 4, pp. 292–308, 1994.

[66] S. Holzer and R. Wattenhofer, "Optimal distributed all pairs shortest paths and applications," in *PODC'12.* ACM, 2012.

[67] P. Krusche and A. Tiskin, "Efficient longest common subsequence computation using bulk-synchronous parallelism," in *ICCSA'06.* Springer, 2006, pp. 165–174.

[68] S. Im, B. Moseley, and X. Sun, "Efficient massively parallel methods for dynamic programming," in *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, 2017, pp. 798–811.

[69] J. Zola, X. Yang, A. Rospondek, and S. Aluru, "Parallel-tcoffee: A parallel multiple sequence aligner." *ISCA-PDCS'07*, vol. 7, pp. 248–253, 2007.

[70] C. Wang, C. Yu, S. Tang, J. Xiao, J. Sun, and X. Meng, "A general and fast distributed system for large-scale dynamic programming applications," *Parallel Computing*, vol. 60, pp. 1–21, 2016.

[71] N. Hegde, Q. Chang, and M. Kulkarni, "D2p: Automatically creating distributed dynamic programming codes," 2018.

[72] J. Lin and M. Schatz, "Design patterns for efficient graph algorithms in mapreduce," in *Eighth Workshop on Mining and Learning with Graphs.* ACM, 2010, pp. 78–85.

[73] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii, "Filtering: a method for solving graph problems in mapreduce," in *SPAA11.* ACM, 2011, pp. 85–94.

[74] M. F. Husain, P. Doshi, L. Khan, and B. Thuraisingham, "Storage and retrieval of large rdf graph using hadoop and mapreduce," in *CLOUD'09.* Springer, 2009, pp. 680–686.

[75] L. Qin, J. X. Yu, L. Chang, H. Cheng, C. Zhang, and X. Lin, "Scalable big graph processing in mapreduce," in *SIGMOD'14.* ACM, 2014, pp. 827–838.

[76] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on Spark," in *GRADES'13.* ACM, 2013, p. 2.

[77] A. Dave, A. Jindal, L. E. Li, R. Xin, J. Gonzalez, and M. Zaharia, "Graphframes: an integrated api for mixing graph and relational queries," in *GRADES'16.* ACM, 2016, p. 2.

[78] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya, "A blocked all-pairs shortest-paths algorithm," *JEA*, vol. 8, pp. 2–2, 2003.

[79] S. Jha, J. Qiu, A. Luckow, P. Mantha, and G. C. Fox, "A tale of two data-intensive paradigms: Applications, abstractions, and architectures," in *BigData Congress*, 2014, pp. 645–652.

[80] H. Asaadi, D. Khaldi, and B. Chapman, "A comparative survey of the hpc and big data paradigms: Analysis and experiments," in *CLUSTER'16.* IEEE, 2016, pp. 423–432.

[81] M. Anderson, S. Smith, N. Sundaram, M. Capotă, Z. Zhao, S. Dulloor, N. Satish, and T. L. Willke, "Bridging the gap between hpc and big data frameworks," *Proceedings of the VLDB Endowment*, vol. 10, no. 8, pp. 901–912, 2017.

[82] S. Itzhaky, R. Singh, A. Solar-Lezama, K. Yessenov, Y. Lu, C. Leiserson, and R. Chowdhury, "Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations," in *ACM SIGPLAN Notices*, vol. 51, no. 10. ACM, 2016, pp. 145–164.

[83] M. M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan, "Parameterized tiling revisited," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization.* ACM, 2010, pp. 200–209.

[84] G. Iooss, S. Rajopadhye, C. Alias, and Y. Zou, "Mono-parametric tiling is a polyhedral transformation," Ph.D. dissertation, INRIA Grenoble-Rhône-Alpes; CNRS, 2015.

[85] Q. Yi, V. Adve, and K. Kennedy, "Transforming loops to recursion for multi-level memory hierarchies," *ACM Sigplan Notices*, vol. 35, no. 5, pp. 169–181, 2000.

[86] M. Griebl, P. Feautrier, and C. Lengauer, "Index set splitting," *International Journal of Parallel Programming*, vol. 28, no. 6, pp. 607–631, 2000.

[87] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.

[88] A. V. Aho and J. E. Hopcroft, *The design and analysis of computer algorithms.* Pearson Education India, 1974.

[89] C. Papadimitriou and M. Sideri, "On the floyd–warshall algorithm for logic programs," *The journal of logic programming*, vol. 41, no. 1, pp. 129–137, 1999.

[90] W. Bielecki, K. Kraska, and T. Klimek, "Using basis dependence distance vectors in the modified floyd–warshall algorithm," *Journal of Combinatorial Optimization*, vol. 30, no. 2, pp. 253–275, 2015.

[91] L. Ridi, J. Torrini, and E. Vicario, "Developing a scheduler with difference-bound matrices and the floyd-warshall algorithm," *IEEE software*, vol. 29, no. 1, pp. 76–83, 2011.

[92] A. Ojo, N.-W. Ma, and I. Woungang, "Modified floyd-warshall algorithm for equal cost multipath in software-defined data center," in *2015 IEEE International Conference on Communication Workshop (ICCW).* IEEE, 2015, pp. 346–351.

[93] M. G. Bell, "Alternatives to dial's logit assignment algorithm," *Transportation Research Part B: Methodological*, vol. 29, no. 4, pp. 287–295, 1995.

[94] A. Pradhan and G. Mahinthakumar, "Finding all-pairs shortest path for a large-scale transportation network using parallel floyd-warshall and parallel dijkstra algorithms," *Journal of Computing in Civil Engineering*, vol. 27, no. 3, pp. 263–273, 2013.

[95] I. K. L. D. Pandika, B. Irawan, and C. Setianingsih, "Apllication of optimization heavy traffic path with floyd-warshall algorithm," in *2018 International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC).* IEEE, 2018, pp. 57–62.

[96] P. Khan, G. Konar, and N. Chakraborty, "Modification of floyd-warshall's algorithm for shortest path routing in wireless sensor networks," in *2014 Annual IEEE India Conference (INDICON).* IEEE, 2014, pp. 1–6.

[97] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *LLVM'15.* ACM, 2015, p. 7.

[98] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming.* MIT press, 2008, vol. 10.

[99] I. M. Wilbers, H. P. Langtangen, and Å. Ødegård, "Using cython to speed up numerical python programs," *Proceedings of MekIT*, vol. 9, pp. 495–512, 2009.

[100] K. W. Smith, *Cython: A Guide for Python Programmers.* " O'Reilly Media, Inc.", 2015.

[101] "DP Spark implementation," https://github.com/TEAlab/DPSpark, 2020.

[102] "Tuning Spark," https://spark.apache.org/docs/latest/tuning.html.

[103] S. Cadambi, G. Coviello, C.-H. Li, R. Phull, K. Rao, M. Sankaradass, and S. Chakradhar, "Cosmic: middleware for high performance and reliable multiprocessing on xeon phi coprocessors," in *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, 2013, pp. 215–226.