2013 IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing

Domain-based MapReduce Programming Model for Complex Scientific Applications

Min Li, Xin Yang, and Xiaolin Li
Scalable Software Systems Laboratory
University of Florida
{minli, xinyang}@ufl.edu; andyli@ece.ufl.edu

***Abstract*—The MapReduce programming model has introduced simple interfaces to a large class of applications. Its easy-to-use APIs and autonomic parallelization are attracting attentions from scientific community. However, current MapReduce-style scientific frameworks focus more on the most popular MapReduce applications that can be easily partitioned and involve little communication across map or reduce tasks. They typically lack adequate support for more complex applications that involve iterative communication and dynamic domain partitioning. In this paper, via abstraction of numerical domains of many applications, we present a domain-based MapReduce programming model for iterative and dynamic scientific applications. Using real-world applications, we introduce a general methodology to adapt APIs of legacy scientific codes into the more developer-friendly MapReduce-like programming model.**

I. Introduction

Programming scientific applications running on High Performance Computing (HPC) systems in parallel always associates with a high degree of complexity. A wide range of concerns including programming models (message passing or shared memory), system architectures (e.g., multicore, SIMD), and runtime managements should be handled. Software packages or solvers are developed through joint efforts of scientists and system developers. But significant details such as runtime configurations for parallelization and complex data structures are exposed to users, resulting in unfriendly programming interfaces.

The MapReduce programming model [1] is designed to provide easy-to-use interfaces for programming data intensive applications on commodity clusters. It relieves users from the heavy handling work in distributed programming and automatically deals with parallelism, scalability and fault-tolerance. The proliferation of MapReduce and the associated systems, e.g., Hadoop [2], Twister [3], Haloop [4], are attracting attentions from the scientific computing community [5], [6], [7], [8]. However, the intrinsic characteristics of the current MapReduce programming model and systems imply that only applications with easy-to-partition workload and simple communication patterns can be migrated.

In this paper, we focus on providing a user-friendly programming model for *domain-based scientific applications*. The computation of these applications are often mathematically modeled in a 2D or 3D grid (or domain). The computation unit is a point in the domain and multiple iterations are regular for one execution. The communications during the execution involve two aspects: temporal (values from past iterations are needed for current computation) and spatial (values of neighboring points are needed). Adaptive and dynamic load balancing are used during the execution to fully leverage the parallelism. A wide range of scientific applications, including sparse matrix decompositions [9], [10], [11], adaptive mesh partitioning [12], [13], [14], image rendering [15] and particle-in-cell simulation [16], [17] fall in the category of domain-based scientific applications. Inspired by the original model, we propose the adapted MapReduce programming model for domain-based scientific applications after investigating their computation and communication patterns. The proposed programming model retains the *map* and *reduce* primitives in MapReduce but introduces some modifications. With this model, scientists are able to program the algorithms for processing a single point in map as if they have the data of the entire grid locally, and easily handle the communication issues according to the stencils in reduce. There is no need to explicitly specify how to acquire the data of neighbor points from other processors or propagate results to others. Considerations for parallelization such as the balanced workload partitioning at runtime are hidden behind the simple yet highly expressive APIs.

The proposed model essentially separates the programming interface apart from the computation framework. This helps framework developers focus on the data movement and communication details under the logic of scientific applications. The system optimization can be performed transparently to the end users. We use a real-world PDE solver, GrACE [18], and three applications to show the effectiveness of the model. However, the proposed MapReduce programming model is not limited to a certain software package or solver. We introduce a general methodology to adapt legacy codes into this user-friendly MapReduce-like programming model.

In summary, our contributions in this paper are:

1) We analyze the real-world domain-based scientific applications and dig out the ultimate requirements for users to program them against a specific computation engine.
2) We propose the MapReduce-like programming model for domain-based scientific applications. This model preserves the easy-to-use programming interface and fits in the needs of complex scientific applications. It

The work presented in this paper is supported in part by National Science Foundation (grants CCF-1128805, OCI-0904938, and CNS-0709329).

978-0-7695-5088-6/13 $31.00 © 2013 IEEE
DOI 10.1109/HPCC.and.EUC.2013.87

simplifies the programming work for scientists.

3) We evaluate our domain-based MapReduce programming model by adapting it onto GrACE, a computational engine infrastructure for Structured Adaptive Mesh Refinement (SAMR) problems, and show that the domain-based MapReduce is easy-to-use and quite effective.

The rest of the paper is organized as follows. In Section II, we introduce the MapReduce programming model, its runtime system and the execution data flow. In Section III, we discuss domain-based applications and their key features. We also propose the MapReduce programming model for domain-based applications. In Section IV, we verify the functionality of the model by implementing it upon the original GrACE. In Section V and Section VI we discuss the related work and conclude our paper.

II. MapReduce

MapReduce [1] and its associated runtime implementation are introduced by Google in 2004. It was inspired by the two general primitives, map and reduce, from Lisp and many other functional programming languages. Programs written in the specified functional style are automatically parallelized and executed upon target commodity machine clusters. In this section, we present a brief introduction about the programming model, the underlying runtime system, and the data flow of MapReduce.

A. Programming Model

The MapReduce framework takes in a set of input key/value pairs, processes them and generates a set of output key/value pairs. Computations are expressed through the two functions, map and reduce, with the interfaces as shown below:

map	$(K1, V1)$	$\longrightarrow list < K2, V2 >$
reduce	$(K2, list < V2 >)$	$\longrightarrow list < V2 >$

The map function, defined by users, takes and processes an input key/value pair($< K1, V1 >$) at a time, generating a set of intermediate key/value pairs ($list < K2, V2 >$). These intermediate results are partitioned by the intermediate keys and values associated with the same key are grouped together forming into one piece of key/value pair ($K2, list < V2 >$) for reduce. Each of these pairs is fed to the reduce function. The reduce function, also implemented by the users, iteratively handles each of the grouped intermediate results. It normally aggregates the values associated with the same key, producing the final output key/value pair ($< K2, V2 >$).

B. Runtime System and Data Flow

Google's MapReduce runtime system runs on top of Google File System (GFS) [19]. Many copies of programs are started after the submission of a job. Mapper and Reducer are programs running the corresponding user defined functions. Master is a unique program coordinates the job execution. Before the start of a MapReduce job, the input data is divided into a set of M splits by the GFS. Mappers are automatically invoked across the machines with regards to the location of the splits. Each split normally contains multiple records of input pairs and is iteratively processed by the map function. The intermediate results are partitioned into a set of R pieces by a partitioning function (e.g., hash on key and mod R) and each part is assigned to a reducer by the master. Reducers are notified by the master after the completion of mappers and pull the intermediate results from mappers. They group the results to combine values with the same key and iteratively aggregate the values for each key to generate the final results. The runtime system also utilizes backup and restart mechanism for fault-tolerance and a central scheduler for job assignments.

III. Domain-based Applications: Features and General Programming Model Design

Many scientific applications can be classified as domain-based application. These scientific applications exhibit some specific features in computation and dependency patterns. These features help us abstract and adapt the MapReduce programming model for domain-based applications. In this section, we discuss the domain-based applications and their features. Through the exploration of these features, we develop a common mapreduce-like programming model for this kind of applications.

A. Domain-based Application and its Features

A large amount of scientific applications focus on exploring the physical properties of certain phenomena within a target area, or simulating such complicated physical processes. Normally, the targeting area is modeled into a standardized geometric 2D or 3D domain ($\mathbf{D}$, or dom), discretized into grids, and represented by a set of meta information, e.g. size, resolution and etc. Each point (P, or $P_{i,j}$ for point at coordinate $< i, j >$ in 2D domain) within this domain is assigned with a set of values corresponding to different physical properties. Calculations are performed on each point, pushing the whole domain into the next time step. This kind of applications are what we called the *domain-based applications*.

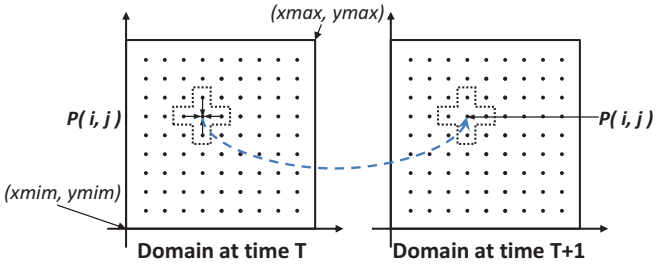

Fig. 1: Dependency relations between points for Jacobi Iteration

Domain-based applications are commonly associated with iterative algorithms and values from different time steps might be needed for computation. This leads to the fact that multiple values of the domain from different time steps coexist at a certain time. The kernel function (usually defined by the

scientists) of a specific application implicitly defines the underlying dependency relations between points of different time steps and different locations within the targeting domain. For example, Fig. 1 shows the dependency in Jacobi Iteration [20]. In this application, the $P_{i,j}$ at time step $T+1$ depends on the values of its adjacent neighbors from time step T. In general, we abstract two relations that defines these dependencies:

Spatial Relation. This relation defines the location dependency and can be expressed by a function, with the form as $P_{i,j}.spatialR(Q_{m,n})$, which returns $True$ only if value at location $< m, n >$ is needed for new value of $< i, j >$. For example in Jacobi Iteration, we can define the function to return $True$ when $distance(P_{i,j}, Q_{m,n}) = 1$. In this way, the direct neighbors as shown within the dash line in Fig. 1 are considered to be location dependents of $P_{i,j}$.

Temporal Relation. This relation defines the time dependency lies between points within the domain during consecutive time steps. It can be expressed by an integer T_{gap}, meaning that values from time step $(T+1) - T_{gap}$ till T are needed for calculation of domain at time step $(T+1)$. For example in Jacobi Iteration, this T_{gap} equals to 1, meaning that the values of the domain at time step $T+1$ only depend on the values from time step T.

The two relations can be combined to locate the dependents of a certain point. We call the point *result point* and the set of dependents as its *argument point set*. The result points form a *result domain* and all the argument point sets make up the *argument domain set*. We should also note that values representing different physical properties may exhibit different spatial relations (e.g. different physical properties in weather application have different calculation kernel). Even if they are of the same property, but from different time steps, the spatial relation pattern between the same result point and them may vary (e.g. Leapfrog scheme for PDE based scientific applications). We classify these points according to the same spatial relation pattern and same time step. And the target area can then be represented by layers of domains, including the result domain and the argument domains.

B. Point-based Algorithm for Domain-based applications in MapReduce

The unit for calculation in domain-based applications is each point within the domain. From the dependency analysis above, we are able to generalize a common compute model for point-based calculation style of domain-based applications:

$$P_{i,j}^{T+1} \;=\; f(\mathbb{S}_{P_{i,j}}^{t}) \qquad \text{(III.1)}$$

, where

$$\begin{cases} t \in [(T+1) - T_{gap}, T]; \\ \mathbb{S}_{P_{i,j}}^{t} = \{\ Q_{m,n}^{t} \mid P_{i,j}^{T}.spatialR(Q_{m,n}^{t})\ \}; \end{cases}$$

The point set $\mathbb{S}_{P_{i,j}}^{t}$ is the argument point set of result point $P_{i,j}^{T+1}$, which are needed for evolvement from $P_{i,j}^{T}$ to $P_{i,j}^{T+1}$. From the Equation III.1, we may infer that there are two phases for point-based algorithm: argument points collecting and the aggregating calculation. It is natural that we use each point's coordinate as the input key and the point's value as its input value and utilize the MapReduce model for processing. For domain-based application, we propose a point-based algorithm as shown in Algorithm 1.

Algorithm 1 Point based algorithm

Input: key // i, the sequential order number of split $\mathbb{S}_i^t$
$value$ // the argument domain set $\mathbb{S}_i^t$
Output: $key1$ // the coordinate $< i, j >$ of target point $P_{i,j}^T$
$value1$ // value and information associated with point $P_{i,j}^T$
1: **map function:**
2: **begin**:
3: **for** each point $Q_{m,n}^t \in \mathbb{S}_i^t$ **do**
4: **for** each point $P_{i,j}^T$, $P_{i,j}^T.spatialR(Q_{m,n}^t) = True$ **do**
5: $key1 \leftarrow < i, j >$
6: $value1.info \leftarrow relative\ position\ and\ other\ info\ between\ P_{i,j}^T\ and\ Q_{m,n}^t$
7: $value1.value \leftarrow value\ of\ Q_{m,n}^t$
8: emit_intermediate($key1$, $value1$)
9: **end for**
10: **end for**
11: **end**

Input: $key1$ // the coordinate $< i, j >$ of the point $P_{i,j}^T$
$value1\_list$ // the list of intermediate values associated
// with the same point $P_{i,j}^T$, which is $\mathbb{S}_{P_{i,j}}^t$
Output: $key2$ // coordinate of point $P_{i,j}^{T+1}$
$value2$ // value of point $P_{i,j}^{T+1}$
1: **reduce function:**
2: **begin**:
3: // use specific numerical method to calculate
4: // the value for next time step
5: **for** each $< key1, value1\_list >$ pair **do**
6: $value \leftarrow f(value1\_list, \mathbb{C})$
7: emit($key1$, $value$)
8: **end for**
9: **end**

The map stage in the point-based algorithm is a preparation phase. The $spatialR()$ function in the map stage relies on the calculation stencil of the application to express the location dependency pattern. It is used to generate the argument point set for each result point. Inside the double loops, each point is propagated to its corresponding result points. The outer loop iterates on every point of the input split for this mapper and the inner loop emits the point for every result point that requires it. The intermediate key type $key1$ is indexed by the coordinate as well, while the intermediate value $value1$ is a complex type. Except for the real value stored on the point, $value1$ might contain information about the relative position between the point and its result point, as $value1.info$ in the algorithm. This information is needed because the calculation function $f()$ of domain-based applications may not be commutative or associative with respect to each element in the argument point set.

The reduce stage is the actual calculation phase. The reduce workers in MapReduce frameworks conduct the sort and group procedures after they get all the intermediate key/value pairs. The combined intermediate results are then passed to and consumed by the user defined reduce function. Upon each value-list associated with the same key, the function $f()$ will be performed, generating the new value of the point for next time step.

C. Domain-based Algorithm for Domain-based application in MapReduce

While point-based algorithm is straightforward and can handle domain-based applications neatly, there are still some disadvantages that cannot be neglected. First, it does not truly explore the *'domain features'* lying under these applications. The spatial relation essentially results in the localized computation pattern; on the hand, the map/reduce phases in point-based algorithm smash the whole domain, shuffle the points through the network and recompose argument point set on the reducer side. The $value1.info$ part is actually the duplicated information and overhead of shuffling the points of the domain. Second, the point-based algorithm generates multiple intermediate key-value pairs for just one input pair, which, consequently, inflates the scale of intermediate results and triggers more communication as well as other handling overheads. Last, range partition is commonly used in domain-based applications for load partitioning, distributing across the cluster, and balancing. The point-based algorithm fails to utilize this good technique.

Traditional range partitioning divide the domain into subdomains (dom_i) with concern of load balance between processes. Instead of regarding each point as an calculation unit, we can consider the subdomains as the calculation unit. In this way, though computation is still performed upon each point within the subdomain, we are able to preserve the localized computation of domain-based applications. These subdomains are essentially the same compared with the original domain in terms of calculation execution. The general compute model for domain-based calculation style is as follows:

$$dom_i^{T+1} \;=\; f(set_i^t) \qquad \text{(III.2)}$$

, where

$$\begin{cases} t \in [(T+1) - T_{gap}, T]; \\ set_i^t = \{\, dom_k^t \mid dom_k^t \in \; argument\, domain\, set\, of\, dom_i^{T+1} \,\}. \end{cases}$$

Following the domain-based calculation style, we first use the same range partition method on all the layers of domains representing the target area into subdomain sets (set_i^t). Each subdomain set consists of the result subdomain and its argument subdomains, and is considered as the computation unit. These sets contain almost all information they need to evolve into the next time step according to the localized computation pattern. The only exception lies along the boundary parts as showing by a simple example at the left part in Fig. 2. The shadowed parts need information beyond their nesting subdomains. To ensure that every point in each subdomain has enough information when performing calculation, we utilize the *'ghost region'* concept. Except for the points within the range, each subdomain holds some extra points of the adjacent subdomains whose values are needed when performing calculations on the boundary points, as specified by the spatial relation. These parts are the ghost regions for the subdomain.

Algorithm 2 Domain-based algorithm

Input: key // i, the sequential order number of set_i^t
$value$ // values for subdomain set set_i^t
Output: $key1$ // same as input key
$value1$ // ghost region for update
1: **map function:**
2: **begin**:
3: // subdomain dom_i is the result subdomain in set_i^t
4: **for** each point $P_{m,n} \in dom_i$ **do**
5: // $\mathbb{S}^t_{P_{m,n}} \subset set_i^t$
6: $P_{m,n}^{T+1} \leftarrow f(\mathbb{S}^t_{P_{m,n}}, \mathbb{C})$
7: **end for**
8: // emit ghost regions for update in reduce phase
9: **for** each neighboring subdomain dom_j^{T+1} **do**
10: emit_intermediate($j,\ ghost_{i\to j}$)
11: **end for**
12: **end**
Input: $key1$ // i, the sequential order number of dom_i^{T+1}
$value1\_list$ // corresponding ghost regions for update
Output: $key2$ // i, the sequential order number of dom_i^{T+1}
$value2$ // updated dom_i^{T+1}
1: **begin**:
2: // update the ghost regions for each dom_i^{T+1}
3: merge(dom_i^{T+1}, $value1\_list$)
4: emit($i,\ dom_i^{T+1}$)
5: **end**

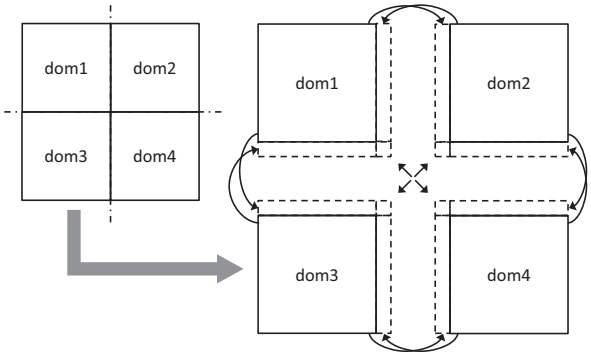

Fig. 2: Boundary and Ghost region value exchanges

Fig. 2 also shows the boundary and ghost region value exchange schema. On the right part of the figure, the shadows around each subdomain are the ghost regions for each of them. The ghost regions consist of parts from all neighboring subdomains, as shown by the pattern of the shadows and the arrows. With these extra data, the function $f()$ in Equation III.2 is able to process all the points within the range of each subdomain. In

order to keep the ghost regions consistent with the subdomain through multiple iterations, the new values should be propagate to these neighbors. This calculate-update model leads to the domain-based algorithm as shown in Algorithm 2.

The algorithm also contains two phases: the calculation phase, which is the map stage; and the ghost regions update phase, which is the reduce stage. In map stage, though we still iteratively perform calculation on each point (map function, line4-7), the temporal and spatial relation features are preserved within the subdomain. After the dom_i^T has evolved to dom_i^{T+1}, the ghost regions that lie within dom_i are emitted to the corresponding subdomains. The ghost regions are collected and merged in the reduce, keeping the subdomain up-to-date and ready for the next iteration.

The domain-based algorithm addresses all the disadvantages mentioned at the beginning of this section. It preserves the domain features by putting a single point into the neighborhood where all the dependencies exist. And this spares the domain-based algorithm with additional position information as the $value.info$ required in point-based algorithm. The only duplicated data is the ghost regions and is only a considerable small part compared with the whole domain. This greatly reduces the message in the network as well as in the memory. The traditional range partition method can be utilized in handling the input for map, achieving load balance without users notice.

IV. Case Study: MapReduce interface for GrACE

In this section, we discuss how our domain-based MapReduce programming model cooperates with the runtime engine for PDE-based applications with SAMR technique, GrACE. We first go through some representative applications and develop the MR-GrACE upon the original runtime. We then generalize a common methodology for adapting MapReduce programming model for domain-based applications. The evaluation of this model is given at the end of this section.

A. GrACE and general adaptation

GrACE is a general computing engine for PDE-based applications. There are three levels for the logical design of GrACE. At the bottom is the data management part which deals with the basic underlying data structure and provides interfaces for the upper level to use. Functions associated with parallelism, e.g. communication and load partition, are also defined on this level. The middle level is the programming model part, where logical geometrical data abstraction is built upon the underlying basic data structures. This part provides interfaces directly used by scientists to implement their applications. At the top is the application layer, where users exploit the provided interfaces for application implementations.

We add a MapReduce level upon the middle level to equip GrACE with domain-based MapReduce programming model. The partitioner in the data management level is used to divide the whole domain into subdomains and these subdomains are distributed to processors with consideration of load balance. To iteratively perform calculation on each point within the subdomain, the meta information of this subdomain should be passed to the map function. In GrACE, the ghost region exchanges only require the user to define the width and should be automatically performed. Thus, the reduce function is implemented and called implicitly when needed. With the considerations above, we design the classes needed for domain-based MapReduce model as shown in Table I.

TABLE I: Classes for MapReduce Model

Class Name	Descriptions
Map	General Map class that contains a pure virtual function map();users inherit this class and instantiate their own specific map() function.
Domain	Stores the meta information for a subdomain, including lower/upper bound and the resolution of an coordinate; passed to user in map function for usage in calculation.
Data	The value of the points in the subdomains in the format of array, e.g. dom[x][y] for value at point $P_{i,j}$ for 2D application.
Job	The main driver for GrACE jobs; utilizes interfaces provided by GrACE for hierarchical data structure setup and other organization/management work; opens API (Job::registerMap(Map mapper)) for user to register their customized Map class; and APIs to specify the ghost region width and T_{gap} for temporal relation.

We intuitively define some important APIs in these classes with the interfaces as shown in Table II.The Domain object specifies the evolvement range, resolution information and also takes on the rule as key for input. The $argument$ is the argument subdomain. $result$ stores values for the new subdomain. In the following subsections we discuss how we gradually adapt this intuitive interface into the final form according to the needs of different applications.

TABLE II: APIs for MapReduce Model

```
void Map :: map(Domain dom, Data argument, Data result);
void Job :: registerMap(Map mapper);
void Job :: setTemporalFactor(int T_gap);
void Job :: setSpatialFactor(int width);
```

B. Representative Applications

Buckley-Leverett. This application follows the Buckley-Leverett model and focuses on the oil-water flow simulation (OWFS). It is used for simulation of hydrocarbon pollution in aquifers. The first-order upwind scheme [21] is applied in this application. Calculations for the values in the next time step only need values of the adjacent neighboring points and their own from the current time step.

The intuitive interface suits this application well. The width of the ghost regions is set to be 1, meaning only the adjacent points should be replicated. The temporal relation factor is set to be 1, requiring the subdomains at the current time step to be the argument subdomains for the next time step.

Wave. This is a 2D wave equation and is solved using the leapfrog scheme [21]. The difference between the leapfrog scheme and the first-order upwind scheme is that the former one requires values from time step $T-1$ as well as time step T when calculating values at $T+1$. This implies that the temporal relation factor for this application is 2.The dom^T and dom^{T-1} should be considered as different layers of domains representing the target area as the spatial relations between them and the next time step differs.

We extend the parameter part from single Data argument to an array of argument and each Data object should be associated with its own spatial relation registration. We utilize GrACE data structure management interfaces to dynamically assemble the Data array in time sequential order according to the temporal relation factor. In this way, we keep the argument domain set consistent with the applications requirement.

Transport. This application solves the 2D transport equation using the MacCormack method [21]. This method has a major difference from the above upwind and leapfrog schemes. The upwind and leapfrog schemes can be viewed as 'atomic' computation with respect to a specific subdomain: each iteration requires only one computing traverse following by the synchronization and value exchanges. However, the two-step computation style of MacCormack method depends on two map-like functions for a complete update.

To address this 'non-atomic' update problem, we extend the Map class registered in one job object to an *action list*. The action list is a series of Map classes implemented and registered by the users. Each of the Map class represents a certain computation action and they are sequentially performed according to the registration order. Reduces are performed implicitly after each individual action. If reduce is not needed, the two actions can be merged into one and are essentially atomic. After all the Map classes in the action list are conducted upon the subdomain, a successful update completes. For this transport application, there are two user implemented Map classes: PredictMap and CorrectMap.

C. Generalization

We make a few changes in the APIs according to the analysis of the three representative applications above, as shown in Table III. We combine the argument and result data together forming a single data array according to the time order. This is quite natural for the MapReduce model: in each computation iteration, the user defined map function makes multiple layers of domains to proceed one time step forward. The function to set spatial factor also adds a time relevant parameter to support multiple argument domains with different spatial relations.

To make use of the domain-based MapReduce programming model, there are generally three steps that framework developers need to follow. First, utilizing the existing interfaces to generate the domain meta information class (Domain) and domain value access interfaces (Data). Second, provide the Map/Reduce classes for user to instantiate the calculation kernels for specific domain-based scientific applications. Third, create APIs for users to express the corresponding temporal and spatial relations. Expend original framework to use these APIs and automatically assemble the domain and data parameters used in map function. Generally, the domain-based MapReduce programming model can be applied to different domain-based runtime systems following the same procedure as that of GrACE. The differences lies in the different APIs to automatically support the above three parts and provide the APIs for temporal and spatial relation as shown in Table II and Table III.

TABLE III: Changes for MR-GrACE

```
void Map :: map ( Domain dom, Data ddom[] );
void Job :: setSpatialFactor(int time, int width);
```

D. Model Evaluation

To test the effectiveness of the domain-based MapReduce programming model, we implement MR-GrACE upon the original GrACE, with domain-based MapReduce programming interfaces. We build the above three applications using MR-GrACE. The effective length of codes for the original GrACE drivers and the new drivers utilizing MapReduce programming model are shown in Table IV. Although the length of code can not entirely represent the simplicity of the programming interface, we can still infer from the considerable difference that the MapReduce programming interface is able to relieve the scientists from heavy programming work.

TABLE IV: Code length comparison between original GrACE and MR GrACE

	Buckley-Leverett	Wave	Transport
Original GrACE	692	1020	669
MR GrACE	131	185	110

We run applications of MR-GrACE on the HPC system at University of Florida. Each node in the cluster has 2 Quad-core Intel E5462 (2.8GHz) processors and 64GB memory. The cluster is connected using the infiniband. As we are validating the effectiveness of the model, we only use a relatively small configuration. Fig. 3 and Fig. 4 show the average aggregated time for the map phase, the reduce phase and total execution for Buckley-Leverett and Transport applications, respectively (The Wava application exhibits similar behavior and we do not show it here due to the space limitation). The curves for original GrACE are identical to MR-GrACE (not shown in the figure because they are overlapped) due to the negligible overhead of the extra programming interface. We can see from the results that MR-GrACE preserves the scalability of the original GrACE. Except for map and reduce, the computation process also contains a lot of other time-consuming procedures, e.g. periodically recomposition of the hierarchical data structure. These are the parts that should be hidden from the users and these parts also accounts for the observation that the

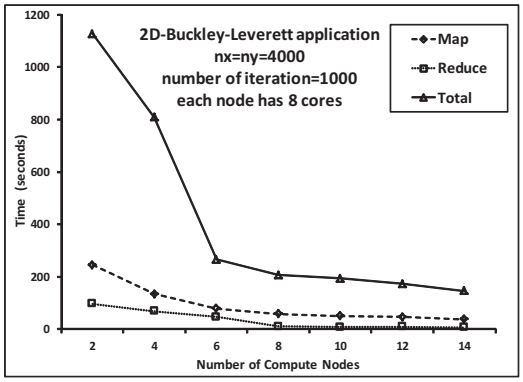

Fig. 3: Results for Buckley-Leverett application of MR-GrACE

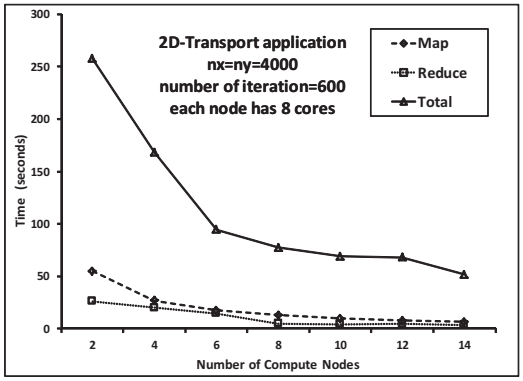

Fig. 4: Results for Transport application of MR-GrACE

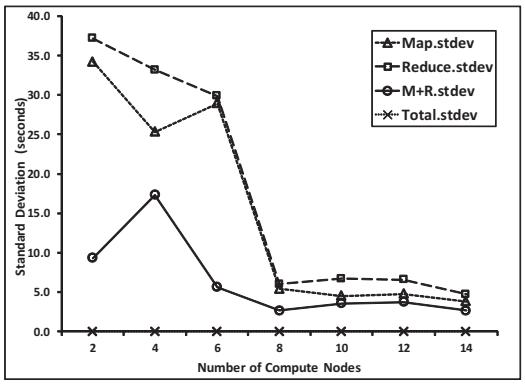

Fig. 5: Standard deviations for Transport application

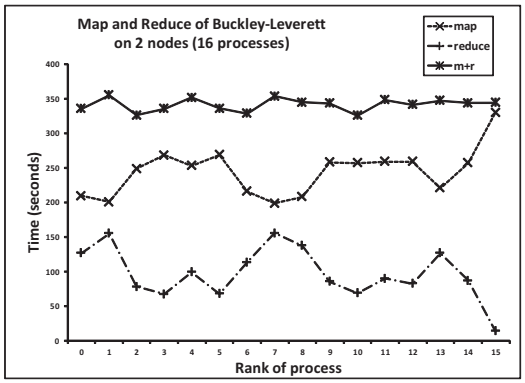

Fig. 6: Map and Reduce time variance for Transport on 2 nodes

total execution time is much longer than the sum of map and reduce.

Fig. 5 shows the standard deviations of map, reduce, sum of map and reduce, and total for different number of nodes. This result mainly shows the effectiveness of the partitioner used in MR-GrACE, which is the default composite partitioner in GrACE. The result shows the standard deviations of map, reduce, and their summation drop with the increase of nodes, which matches the partitioner. GrACE uses the synchronous calculation model, which accounts for the negligible deviations of total iteration time. The variances for summation are smaller than that of map or reduce, which can also be attributed to the synchronous model. Details for 2-node results are shown in Fig. 6. We can see that map with small load completes quickly, but needs more time for ghost region exchanges (long reduce time).

V. Related Work

Researchers never stopped the efforts for simplifying the programming in parallel and distributed systems. Libraries like MPI [22], OpenMP [23] and etc. provide APIs for easy communication and memory sharing. Yet, they are low level APIs and rarely have application aware considerations. As a result, they are often preferred tools for framework builders rather than end users.

MapReduce aims at running data-intensive parallel applications on large clusters of commodity machines. And Hadoop [24] is a mature open source implementation of MapReduce widely used in industry and academic. The simple and easy programming languages has been explored on other platforms. Phoenix [25] is a MapReduce implementation for multi-core and multiprocessor systems. Work like [26], [27], [28] discusses the experience in utilizing MapReduce on GPUs. Enhancements for the original MapReduce framework have also been a hot topic in recent years. [3], [4], [29], [30], [31] improve the MapReduce framework for iterative operations. YC Kwon et al. presented how they mitigate the skew in MapReduce through [7], [32]. The programming model and data flow pattern of MapReduce also inspired good parallel processing frameworks like Dryad [33], Pregel [34], Piccolo [35] etc.

MapReduce frameworks have also been applied to scientific data analyses. [5], [36], [6] focus on utilizing MapReduce-like frameworks for data-intensive scientific analyses. [37], [7] introduced MapReduce in some specific scientific area. However, our proposed MapReduce programming model focuses on domain-based applications with consideration and application awareness about their computation and dependency patterns. The model can also be capped onto existing computation frameworks without much efforts or performance

degradation.

VI. Conclusion

We presented a domain-based MapReduce programming model for dynamic iterative scientific applications. Using two APIs to specify dependency requirements of each point in a domain, we successfully abstracted the general procedure for map and reduce functions.This domain-based programming model is naturally aligned with the domain view of general scientific applications, making it developer-friendly. This domain-based abstraction also considers common computation and dependency patterns in these applications. We believe such simpler model with application-awareness dramatically lowers the barrier to entry for scientists to develop scientific codes following the natural domain-based views. Through creating the domain-based MapReduce programming model for our computational engine GrACE, we verified the functionality and effectiveness of the domain-based MapReduce programming model. We also generalize the common methodology for adapting this model to other scientific computing frameworks.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] "Apache Hadoop Homepage," http://hadoop.apache.org/.

[3] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, 2010, pp. 810–818.

[4] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[5] J. Ekanayake, S. Pallickara, and G. Fox, "Mapreduce for data intensive scientific analyses," in *eScience, 2008. eScience'08. IEEE Fourth International Conference on*. IEEE, 2008, pp. 277–284.

[6] J. Wang, D. Crawl, and I. Altintas, "Kepler+ hadoop: a general architecture facilitating data-intensive applications in scientific workflow systems," in *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*. ACM, 2009, p. 12.

[7] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skew-resistant parallel processing of feature-extracting scientific user-defined functions," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 75–86.

[8] X. Yang, Z. Yu, M. Li, and X. Li, "Mammoth: autonomic data processing framework for scientific state-transition applications," in *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*. ACM, 2013, p. 13.

[9] B. Vastenhouw and R. Bisseling, "A two-dimensional data distribution method for parallel sparse matrix-vector multiplication," *SIAM review*, pp. 67–95, 2005.

[10] A. Pinar and C. Aykanat, "Sparse matrix decomposition with optimal load balancing," in *High-Performance Computing, 1997. Proceedings. Fourth International Conference on*. IEEE, 1997, pp. 224–229.

[11] M. Ujaldon, S. Sharma, E. Zapata, and J. Saltz, "Experimental evaluation of efficient sparse matrix distributions," in *Proceedings of the 10th international conference on Supercomputing*. ACM, 1996, pp. 78–85.

[12] M. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations," *Journal of computational Physics*, vol. 53, no. 3, pp. 484–512, 1984.

[13] D. Bai and A. Brandt, "Local mesh refinement multilevel techniques," *SIAM journal on scientific and statistical computing*, vol. 8, p. 109, 1987.

[14] R. Bank and A. Sherman, "Algorithmic aspects of the multi-level solution of finite element equations," *Duff-Stewart [1]*, pp. 62–89, 1979.

[15] H. Kutluca, T. Kurç, and C. Aykanat, "Image-space decomposition algorithms for sort-first parallel volume rendering of unstructured grids," *The Journal of Supercomputing*, vol. 15, no. 1, pp. 51–93, 2000.

[16] S. Plimpton, D. Seidel, M. Pasik, R. Coats, and G. Montry, "A load-balancing algorithm for a parallel electromagnetic particle-in-cell code," *Computer physics communications*, vol. 152, no. 3, pp. 227–241, 2003.

[17] H. Karimabadi, H. Vu, D. Krauss-Varban, Y. Omelchenko, and J. Raeder, "Global hybrid simulations of the earth magnetosphere: Nuts and bolts," *Bulletin of the American Physical Society*, 2005.

[18] M. Parashar and X. Li, "Grace: Grid adaptive computational engine for parallel structured amr applications," *Advanced computational infrastructures for parallel and distributed adaptive applications*, pp. 249–263, 2010.

[19] S. Ghemawat, H. Gobioff, and S. Leung, "The google file system," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[20] R. Barrett, *Templates for the solution of linear systems: building blocks for iterative methods*. Society for Industrial Mathematics, 1994, no. 43.

[21] J. Tannehill, D. Anderson, and R. Pletcher, *Computational fluid mechanics and heat transfer*. Taylor & Francis Group, 1997.

[22] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface, seconde édition*. the MIT Press, 1999.

[23] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[24] A. Bialecki, M. Cafarella, D. Cutting, and O. OMALLEY, "Hadoop: a framework for running applications on large clusters built of commodity hardware," *Wiki at http://lucene. apache. org/hadoop*, vol. 11, 2005.

[25] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*. Ieee, 2007, pp. 13–24.

[26] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 260–269.

[27] F. Ji and X. Ma, "Using shared memory to accelerate mapreduce on graphics processing units," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 805–816.

[28] B. Catanzaro, N. Sundaram, and K. Keutzer, "A map reduce framework for programming graphics processors," in *Workshop on Software Tools for MultiCore Systems*, 2008.

[29] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "imapreduce: A distributed computing framework for iterative computation," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 1112–1121.

[30] P. Bhatotia, A. Wieder, R. Rodrigues, U. Acar, and R. Pasquin, "Incoop: Mapreduce for incremental computations," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 7.

[31] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. USENIX Association, 2010, pp. 10–10.

[32] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: mitigating skew in mapreduce applications," in *Proceedings of the 2012 international conference on Management of Data*. ACM, 2012, pp. 25–36.

[33] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 59–72, 2007.

[34] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 international conference on Management of data*. ACM, 2010, pp. 135–146.

[35] R. Power and J. Li, "Piccolo: building fast, distributed programs with partitioned tables," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 2010, pp. 1–14.

[36] J. Ekanayake, T. Gunarathne, G. Fox, A. Balkir, C. Poulain, N. Araujo, and R. Barga, "Dryadlinq for scientific analyses," in *e-Science, 2009. e-Science'09. Fifth IEEE International Conference on*. IEEE, 2009, pp. 329–336.

[37] A. Verma, X. Llora, D. Goldberg, and R. Campbell, "Scaling genetic algorithms using mapreduce," in *Intelligent Systems Design and Applications, 2009. ISDA'09. Ninth International Conference on*. IEEE, 2009, pp. 13–18.