Space-Time is the Key: For Expressing Spatial-Temporal Computing

E. Shook[1]

[1]Department of Geography, Environment, and Society, University of Minnesota

*Email: eshook@umn.edu

Abstract

This paper outlines a new spatial-temporal programming model. The model builds on key-value programming models such as MapReduce by enabling bounded space and time to be keys and a collection of spatial-temporal data to be values. This reconceptualization of key-value programming exploits spatial-temporal characteristics in data to facilitate the parallelization of spatial methods and models. This paper outlines the new space-time key-collection programming model and a proof-of-concept implementation For Expressing Spatial-Temporal computation in parallel, called ForEST. Three use cases for the programming model and ForEST language are outlined. First, as a platform to advance research in geospatial computing and algorithm development for spatial problems. Second, as a teaching tool to help learners understand the complexities around handling data, expressing computation, and executing code in spatial-temporal applications. Third, as a language to help process, mine, and analyze spatial-temporal data in a number of fields including GeoComputation, Geographic Information Science, and Spatial Data Science.

Keywords: geospatial computing, domain-specific programming language, big data.

1 Introduction

MapReduce revolutionized distributed computing by building on two classic primitives from functional programming, namely *Map* and *Reduce* (Dean and Ghemawat, 2008). Traditionally, map applies a function to each element in a list producing a new list, and reduce applies a reducer function such as sum to each element in a list and produces a scalar value. MapReduce is a generic programming model that is applicable to a broad set of problems, hides the complexity of parallelism from the application developer, and enables automatic parallelism the so-called "holy grail" of parallel computing. The novelty of MapReduce was not in the methods—Map and Reduce—which were around for decades, but rather how they were elegantly linked together using key-value pairs. MapReduce achieved automatic parallelism by separating the data model (key-value pairs), expression of the computation (mappers and reducers), and execution of the code (MapReduce system). This separation both hid parallelization complexities from users while also enabling optimizations and tuning at a system level. MapReduce also made it clear that many, often simple map and reduce tasks can be combined to create complex workflows and still achieve remarkable performance. This model served as a basis for a wide range of follow-on implementations that have driven big data processing in industry and academia. However, the key-value programming model can be difficult to apply to spatial problems.

This paper reenvisions the generic key-value programming model to use space-time as the key and a collection of spatial-temporal data as the value. This paper's key contributions include: (1) a space-time key-collection programming model, and (2) a proof-of-concept implementation of the model called $ForEST$ (For Expressing Spatial-Temporal computing). The programming model and associated implementation are capable of expressing myriad methods and models in geocomputation, geographic information science, spatial analysis, spatial data mining, and spatial data science.

The rest of this article is organized as follows. Section 2 reviews MapReduce and related work. Section 3 describes the space-time key-collection programming model. Section 4 describes a proof-of-concept implementation. Section 5 draws conclusions and discusses future applications.

2 MapReduce

MapReduce and MapReduce-like variants including Map-Reduce-Merge (Yang et al., 2007) have both advantages and disadvantages. They can effectively leverage massive amounts of cheap commodity hardware for high-throughput computing, are fault-tolerant through data replication, and support automatic parallelization through a simple model based on functional programming primitives (Map and Reduce) (White, 2012; Dean and Ghemawat, 2008; Yang et al., 2007). However, MapReduce has several key drawbacks that limit its application to spatial problems.

1. MapReduce tends to work best on homogeneous data (e.g., webpage text, log files, etc). An extension called Map-Reduce-Merge partially resolves this limitation by introducing a *Merge* task that follows a Reduce task and "merges" data from separate sources.

2. Partitioning (spatial) data occurs outside of the MapReduce model. This simplifies execution, because all data partitioning is handled "outside of the execution model" in theory. However, in practice users are still required to split their data. Usually this occurs when data is stored in a distributed file system (e.g., Google File System, Hadoop Distributed File System) or memory system (e.g., Spark Resilient Distributed Datasets). Unfortunately, support for splitting spatial data is lacking. Spatial MapReduce extensions including SpatialHadoop, Hadoop GIS, or GeoSpark have provided excellent support for spatial operations (Eldawy and Mokbel, 2014; Yu et al., 2015; Aji et al., 2013; Eldawy and Mokbel, 2015). While they add spatial support for code inside map and reduce tasks, they do not incorporate spatial support for the programming model itself.

3. The key-value model is designed to apply an operation to each data element. However, many spatial operations use neighboring data as part of the calculations for each data element (e.g., convolution, windowed operations, focal operations). Without significant data duplication the key-value model does not provide a suitable way to capture spatial neighborhoods as part of a calculation. Common parallelization techniques such as using shared memory and memory pointers that reference overlapping data elements are not supported in the key-value model. While this limitation simplifies the implementation of key-value programming, it also hampers the ability for spatial operations to share and/or reuse data. This is perhaps the biggest limitation of MapReduce.

In short, MapReduce has been very successful in many application domains. However, for spatial-temporal applications MapReduce has several limitations.

3 Programming Model

The *Space-Time Key-Collection* model represents a novel programming model for geospatial computing. It is not an extension of MapReduce nor a spatial database management system (DBMS), and it is not generic or suitable for all domains. Rather, it is a new conceptualization of key-value programming that uses bounded space-time as a "key" and a collection of spatial-temporal data that are situated within the space-time bounds as a "value". In other words, space and time are first-class citizens in this programming model. Due to the use of space-time the programming model can leverage topological models such as the 9-intersection model (Egenhofer et al., 1993) for examining spatial relationships between collections.

3.1 Data Structure: Space-Time Key-Collections

Bounded-space time forms the "key" and spatial-temporal data forms the "value" in a Space-Time Key-Collection (*stkc*). They minimally consist of an origin point in space-time (y, x, z, t), spatial-temporal bounds ([h]eight, [w]idth, [v]ertical height, [d]uration) to form a space-time key (*stk*), and spatial-temporal data comprising a [c]ollection of individual spatial-temporal features $[f_1, f_2, ..., f_n]$. Just as MapReduce is agnostic to value's data type in a key-value pair, this model is agnostic to the type of spatial-temporal data in a collection. The construction of a *stkc* is illustrated below.

$$\begin{aligned} stk\text{:} &\quad ((y, x, z, t), (h, w, v, d)) \\ [c]: &\quad [f_1, f_2, ..., f_n] \\ stkc\text{:} &\quad (stk, [c]) \end{aligned}$$

The programming model assumes that all features are bounded in space and time (i.e., there are no infinitely sized features in any ST dimension). So features can be bounded using an origin point (y, x, z, t) represented as the minimum value in each dimension, and a set of bounds (h, w, v, d) represented as the difference between the minimum and maximum value in each dimension.

Using bounded space-time as a key provides several benefits. First, it provides a straightforward mechanism to organize data that are spatially and/or temporally proximate, which will often improve data locality in spatial-temporal operations. For example, to support data query and retrieval data can be organized in collections based on R-tree or quad-tree so they perfectly align with a spatial index. Second, it can be an easy way to reason about parallel data processing if space-time keys control how data are partitioned and processed in parallel (Figure 1). Third, it decouples the type of spatial-temporal data (e.g., raster versus vector versus space-time cube) from the computational system enabling generic support of spatial-temporal computation.

3.2 Data Computation: Sweep Tasks

Sweep tasks are the foundation of data computation in the *stkc* programming model. They accept one or more *stkc*, sweep over the features in a collection applying analytical, modeling, or other computational processing capabilities, and output one or more *stkc*. Unlike MapReduce, keys are bounded space-time so there is no need to have separate map and reduce tasks. Reduce-like sweep tasks simply output a single feature in a *stkc* and Map-like sweep tasks output many features in a *stkc*.

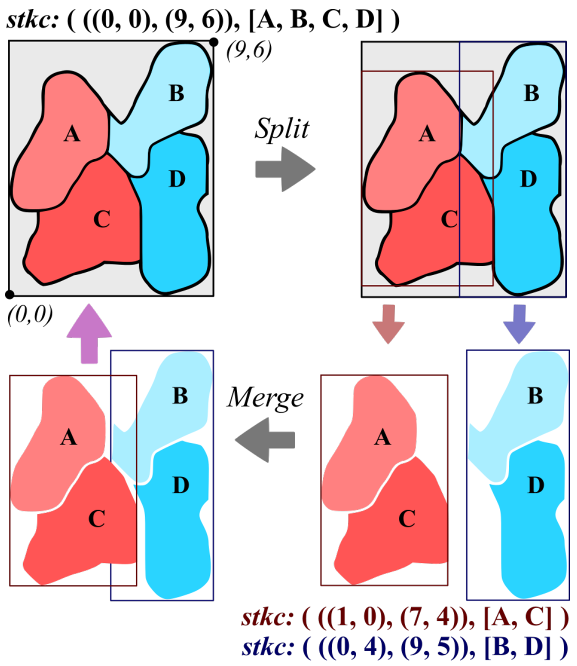
stkc: (((0, 0), (9, 6)), [A, B, C, D])
(9,6)
(0,0)
Split
Merge
A
B
C
D
stkc: (((1, 0), (7, 4)), [A, C])
stkc: (((0, 4), (9, 5)), [B, D])

Figure 1: Two-Dimensional Space-Time Key-Collection (*stkc*) example demonstrating four features in a *stkc* (upper left), which are split into two *stkc* (lower right) that each contain two features. Then, the two *stkc* being merged back into the original *stkc*.

For illustration purposes the *stkc* model is compared to Map, Reduce, and Merge tasks. Map tasks take a key-value pair and output a list of key-value pairs. The keys and values can be different (hence k_1 to k_2). The system combines all values with the same key to a list $(k_2, [v_2])$. Reduce tasks take a key and list of values and output a list of values. Merge tasks take the original key from reduce (k_2) and the list of values $[v_3]$ from multiple reduce tasks enabling the system to handle heterogeneous data. Notice map, reduce, and merge each accept or produce a different type of data (i.e., a key-value pair, a list of key-value pairs, a key and list of values, or a list of values). Sweep tasks, on the other hand, accept and output *stkc* collections (i.e., a space-time key and a collection of features).

$$\begin{aligned}
\text{map:} \quad & (k_1, v_1) \rightarrow [(k_2, v_2)] \\
\text{reduce:} \quad & (k_2, [v_2]) \rightarrow [v_3] \\
\text{merge:} \quad & ((k_2, [v_3]), (k_3, [v_4])) \rightarrow [(k_4, v_5)] \\
\textbf{sweep:} \quad & ((stk_1, [c_1]), (stk_2, [c_2])) \rightarrow [(stk_3, [c_3])]
\end{aligned}$$

The *sktc* model aligns with existing spatial models such as cartographic modeling (Tomlin, 2013) and its parallel extension—parallel cartographic modeling (Shook et al., 2016)—which are powerful spatial data processing frameworks that support many spatial methods and models. In these two frameworks, the elemental computing tasks input and output spatial data layers so there is a direct translation to sweep tasks and *stkc*. However, the drawback of a generic sweep task is that the system cannot make assumptions about what to do next unlike for map and reduce tasks. For example, after a map task, the MapReduce system knows to gather and create a list of values for each key. Instead, we must introduce a way to express how *stkc* flow between sweep tasks.

3.3 Data Flow: Postfix Expressions

An expression represents a workflow (or data flow) for a spatial-temporal method or model. Formally, expressions use postfix notation in which operators follow their operands (e.g., $12+$ rather than $1 + 2$). In the case of the *stkc* model, operators are compute functions (i.e., sweep tasks) and operands are data load functions (i.e., data read tasks).

Postfix notation has several advantages for a workflow syntax. First, order of evaluation is always left-to-right and there is no ambiguity in terms of order of precedence. Second, operators are applied to values immediately to the left of themselves. This follows intuition in terms of the output of one operator becomes the input of another operator in a workflow system. Finally, implementing postfix notation is straightforward. This not only simplifies development, but also makes it easy to learn in an education setting.

Data flow in the *stkc* model is handled as a data stack. Each data load operator (e.g., file read) pushes a *stkc* onto the stack. Each sweep task pops one or more *stkc* off of the stack (depending on the number of parameters). Sweep tasks produce one or more *stkc*, which are pushed on the stack. Data store operators (e.g., file write) pop a *stkc* off the stack and write the data to disk.

4 Implementation

ForEST is a domain-specific programming language **For E**xpressing **S**patial-**T**emporal computing. Domain-specific programming languages provide expressive power for particular domains and have been used to parallelize existing spatial data processing frameworks (Shook et al., 2016). Forest is a proof-of-concept implementation of the space-time key-collection programming model. The fundamental data structure is the spatial-temporal Bounding Object (*Bob*), which implements a *stkc*. The fundamental compute function is a *Primitive*, which generally accept one or more Bobs as input and produce one or more Bobs as output. *Primitives* serve as sweep tasks. Users can combine primitive tasks into *Patterns* using common workflow patterns (van Der Aalst, 2009). Patterns implement postfix data flow in the *stkc* model. Importantly, Patterns allow users to declare the start and end of parallelism using a simple syntax. This new syntax facilitates declarative parallel execution of Patterns. Many spatial-temporal methods or models can be constructed as a pattern in a single line of ForEST code. In addition to the language, the ForEST system supports multiple *Engines*, which accept a Pattern as input, and manages their parallel execution.

4.1 Bobs

A *Bob*—spatial-temporal Bounding OBject—is the basic data structure in ForEST. Bobs implement *stkc* and have a space-time key and a list of features. Using a list of features as a collection allows ForEST to support raster data as a list of cells, vector data as a list of points, lines, or polygons, as well as a multitude of temporal and spatial-temporal data structures. Depending on the data type additional metadata and indices can be optionally incorporated. Further, since each feature has a spatial and temporal attribute the ForEST system can split a Bob into multiple Bobs or merge multiple Bobs into a single Bob. This feature enables the system to organize features spatially and temporally, which often preserves data locality and thus improves computational performance.

ForEST can take advantage of point set theory and topological models such as the 9-Intersection Model (9IM) (Egenhofer et al., 1993) to provide a powerful foundation for geometric operations comparing interior, boundary, and exterior properties of features (and subsequently Bobs) to derive spatial relationships (e.g., disjoint, within, touches, overlaps). The 9IM can allow the ForEST system to reason about Bobs and manage spatial-temporal data in much the same way MapReduce systems can reason about and manage generic data such as grouping values that share the same key into a list. The current implementation includes basic support, but this is an area of future exploration.

There are three key points worth noting. First, Bobs do not need to be *minimum* bounding boxes (also called the minimum bounding rectangle, MBR). While oftentimes beneficial for optimization of computation, the system does not require Bobs to minimally bound their features. This allows systems to align with common decomposition and indexing schemes such as row decomposition and quadtree. Forcing MBR on the system would break the assumptions of these approaches. Second, Bobs can be empty. Again, so ForEST can align with approaches such as quadtrees. Third, it is possible to have two or more Bobs with the same STKs. Recall, Bobs are merely containers of features so it does not cause problems if the containers spatially and/or temporally overlap.

4.2 Primitives

A *Primitive* task is the basic compute function in ForEST. Primitive tasks are executed serially and take a set of Bobs as input and produce a set of Bobs as output. Primitives are intended to be the building blocks of methods and models rather than an entire method or model. This allows for flexible design and development as well as the ability of the ForEST system to parallelize spatial-temporal computation. Similar to MapReduce, the computation within a Primitive task is abstracted from the system so Primitives are free to be as simple or complex as a developer needs. To demonstrate the proof-of-concept, the author along with several students have implemented a suite of Primitives in the development version of ForEST ranging from raster, vector, space-time cube, and agent-based modeling operations (Shook, 2018).

4.3 Patterns

A *Pattern* is an expression of a spatial-temporal method or model. Patterns use a custom syntax that can be used to declare the start and end of data parallelism. Importantly, Patterns give Engines flexibility in *how* or *even if* parallelism is invoked for spatial methods or models. Equally important, Patterns remove the need for developers to program in parallel. Ultimately, the goal of a pattern is to express a parallel spatial-temporal computation without specifying how it will be parallelized. This is similar to other more familiar declarative languages such as SQL in which queries define *what* data is needed, but not *how* to retrieve it.

In between each primitive is a workflow *Connector*. ForEST currently supports three common workflow connectors: split, sequence, and merge. It is straightforward to support additional connectors, but these three cover many spatial-temporal applications.

Name	Connector	Description
split:	<	partition Bobs, begin parallelism
sequence:	==	pass Bob to the next primitive, if enabled maintain parallelism
merge:	>	gather Bobs, end parallelism

Take the following hypothetical example (A * (B+C)) using the read, add, multiply, and write primitives:

read(A) == read(B) == read(C) < add == multiply > write(out)

In this case the first three operations (reading A, B, and C) can execute at the same time. The system would have to preserve the order of the stack (C is above B, which is above A). Then, the split connector enables parallel computation. This will split the Bobs in the stack and (depending on the Engine) will run the add task on multiple processing cores. The top two Bobs (i.e., B and C) are popped off the stack(s) and multiplied. The output Bob (hidden) is pushed back onto the stack. The multiply primitive then multiplies the two Bobs (i.e. the hidden Bob and A) and pushes the final result onto the stack. The merge connector triggers the Bobs to be merged and ends parallelism. Finally, the results are written out using the write primitive by popping off the final Bob on the stack and writing the data to disk as out. Notice, add and multiply are executed in parallel, but the developer did not have to specify how the data would be split or how parallelism would be achieved. Instead they declared their desire for parallelism and the ForEST system handles the execution details.

Each connector is summarized below:

1. **Split** declares the start of parallel processing, partitions Bobs, and distributes the partitioned Bobs to various processing cores. A split operator will also change the spatial extent of each primitive from the entire spatial datasets (global bounds including all data features) to the features in the split Bob. Users can pass parameters to change split (e.g., decomposition method, the granularity of decomposition, etc.), however it is important to note that users are declaring parallelism and it is up to the compute Engine to actually invoke parallelism.

2. **Sequence** passes the output of one primitive to be the input of the next primitive. Sequence can be conceptualized as a UNIX pipe (|) where data is passed from one operation to the next. Users can avoid parallelism by replacing split and merge operators with sequences. In this case, the primitives will process all features. This simple change is useful when debugging codes.

3. **Merge** declares the end of parallel processing and gathers Bobs to a single core. A merge operator will also return the spatial extent of the Bob to the entire spatial dataset. Similar to Split users can pass parameters to Merge for performance tuning.

4.4 Further Details

ForEST is an open source project and available on GitHub (Shook, 2018). It is implemented in the Python programming language. Preliminary implementations have shown that ForEST can support multiple computing engines including one non-parallel engine that tiles data to reduce memory load and parallel engines using shared memory (multiprocessing library), Spark, and Graphics Processing Unit (GPU) through PyCUDA. Preliminary testing in social-environmental applications using

IPUMS-Terra (formerly TerraPop) (Haynes et al., 2017) and on cyberinfrastructure projects such as the Extreme Science and Engineering Discovery Environment (Towns et al., 2014; GISandbox, 2018) show promise. Due to space limitations I cannot go into detail on the implementation or experiments, but I encourage readers to examine the early-stage source code and use GitHub to ask questions, submit feature requests, or report bugs.

5 Conclusions and Future Work

The space-time key-collection programming model is designed for geospatial computing. This new programming language rethinks the key-value programming model to use space-time itself as a key and collections of spatial-temporal data as values. An open source proof-of-concept implementation called ForEST (Shook, 2018) demonstrates that the model works in practice. Further work is required to solidify ForEST into a production-level system and examine the theoretical capabilities and limitations of the programming model. Real world applications such as the Paleoscape Model (Shook et al., 2015) will be used to further test ForEST. Importantly, future work will test ForEST's ability to implement analytics methods as well as geospatial models in the language. This combined capability of expressing both spatial-temporal analytical methods and models will help it contribute to the field of Geocomputation and its combined focus on geospatial analytics and modeling.

6 Acknowledgements

The author would like to acknowledge the Minnesota Population Center (MPC) for providing support to conduct this research. The author also wishes to thank the graduate students in the CyberGIS Seminar courses taught in Spring 2017 and Spring 2019 at the University of Minnesota as well as Dr. Jayakrishnan Ajayakumar for their insights and contributions toward the development of ForEST. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. The author thanks Davide Del Vento and Brian Vanderwende for their assistance with supporting the GISandbox that was during development and testing, which was made possible through the XSEDE Extended Collaborative Support Service (ECSS) program.

7 References

Aji, A., F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz
2013. Hadoop GIS: a high performance spatial data warehousing system over mapreduce. *Proceedings of the VLDB Endowment*, 6(11):1009–1020.

Dean, J. and S. Ghemawat
2008. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Egenhofer, M. J., J. Sharma, D. M. Mark, et al.
1993. A critical comparison of the 4-intersection and 9-intersection models for spatial relations: formal analysis. In *Proceedings of the AutoCarto'11 Conference*. ASPRS.

Eldawy, A. and M. F. Mokbel
2014. Pigeon: A spatial mapreduce language. In *2014 IEEE 30th International Conference on Data Engineering*, Pp. 1242–1245. IEEE.

Eldawy, A. and M. F. Mokbel
2015. Spatialhadoop: A mapreduce framework for spatial data. In *2015 IEEE 31st international Conference on Data Engineering*, Pp. 1352–1363. IEEE.

GISandbox
2018. GISandbox Homepage. `http://gisandbox.org`.

Haynes, D., S. Manson, and E. Shook
2017. Terra Populus architecture for integrated big geospatial services. *Transactions in GIS*, 21(3):546–559.

Shook, E.
2018. ForEST Programming Language. `https://github.com/eshook/Forest`.

Shook, E., M. E. Hodgson, S. Wang, B. Behzad, K. Soltani, A. Hiscox, and J. Ajayakumar
2016. Parallel cartographic modeling: a methodology for parallelizing spatial data processing. *International Journal of Geographical Information Science*, 30(12):2355–2376.

Shook, E., C. Wren, C. W. Marean, A. J. Potts, J. Franklin, F. Engelbrecht, D. O'Neal, M. Janssen, E. Fisher, K. Hill, et al.
2015. Paleoscape model of coastal South Africa during modern human origins: progress in scaling and coupling climate, vegetation, and agent-based models on XSEDE. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, P. 2. ACM.

Tomlin, C. D.
2013. *GIS and cartographic modeling*. Esri Press.

Towns, J., T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, et al.
2014. XSEDE: accelerating scientific discovery. *Computing in Science & Engineering*, 16(5):62–74.

van Der Aalst, W. M.
2009. Workflow patterns. *Encyclopedia of Database Systems*, Pp. 3557–3558.

White, T.
2012. *Hadoop: The definitive guide*. O'Reilly Media, Inc.

Yang, H.-c., A. Dasdan, R.-L. Hsiao, and D. S. Parker
2007. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, Pp. 1029–1040. ACM.

Yu, J., J. Wu, and M. Sarwat
2015. Geospark: A cluster computing framework for processing large-scale spatial data. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, P. 70. ACM.