# A Multi-Agent Parallel Approach to Analyzing Large Climate Data Sets

Jason Woodring*, Matthew Sell*, Munehiro Fukuda*, Hazeline Asuncion* and Eric Salathé†
* Division of Computing and Software Systems
† Division of Physical Sciences
University of Washington Bothell
18115 NE Campus Way, Bothell, WA 98011
{jman5000, mrsell, mfukuda, hazeline, salathe}@uw.edu

***Abstract*—**
**Despite various cloud technologies that have parallelized and scaled up big data analysis, they target data mostly in texts which are easy to partition and thus easy to map over a cluster system. Therefore, their parallelization do not necessarily cover scientific structured data such as NetCDF or need additional, user-provided tools to convert the original data into specific formats. To facilitate user-intuitive parallelization of such scientific data analysis, this paper presents an agent-based approach that instantiates distributed arrays over a cluster system, maintains structured scientific data in these arrays, deploys many mobile agents over the arrays to perform computational actions on data, and collects necessary results. To demonstrate the practicability of our agent-based approach, we focused on climate change research and implemented a web-interfaced climate analysis, using the MASS (multi-agent spatial simulation) library. In this paper, we show practical advantages of, performance improvements by, and challenges for our agent-based approach in structured data analysis.**

## I. Introduction

While most open-source software used in cloud computing such as MapReduce [1], Spark [2], and Storm [3] facilitates data-processing power on text data such as key/value pairs, CVS, and SQL schemas, some applications in scientific data analysis need to analyze binary or multi-dimensional structured data such as NetCDF [4]. Three approaches to addressing this requirement can be considered: (1) relying on utilities customized for data analysis such as CDO and NCL [5], (2) using platform-aware libraries for parallelization such as MPI [6] and GlobalArray [7], and (3) converting structured data into MapReduce/Spark-readable formats and feeding them to SciHadoop [8] and SciSpark [9]. Of importance is that they are not necessarily computing specialists who understand underlying parallel-programming techniques and distributed-computing architectures. Therefore, these customized software tools, parallelization-aware libraries, or user-provided data conversion can be still programming barriers to scientists.

To ease parallelization of scientific data analysis, we take an agent-based approach that maintains scientific data in distributed arrays over a cluster system, deploys many mobile agents over the arrays to perform arithmetic actions on data, and lets them collect computational results. We have applied to this approach our parallelization library for multi-agent spatial simulation (MASS) [10]. To demonstrate the MASS library's practicability in data analysis, we focused on climate change research that has been conducted at University of Washington [11]. The analysis handles a collection of NetCDF files and examines the time series of climate data, which performs repetitive iterations of computation over such resilient distributed datasets. The analysis includes discoveries of the future time of emergence (ToE) [12].

The contribution of this paper is two-fold: (1) demonstrating the practicability of agent-based approach to scientific data analysis, in particular focusing on structured data and (2) confirming performance improvements of data analysis using different agent migration algorithms. The rest of the paper is organized as follows: Section II identifies computational and programming requirements in climate analysis and proposes the use of the MASS library for this data analysis; Section III explains the UWCA (University of Washington Climate Analysis) system that implements a web-interfaced climate analysis with MASS; Section IV demonstrates the MASS library's practicability in climate analysis; Section V discusses the performance improvements by and challenges for our agent-based approach to climate analysis; and section VI states our conclusions and future work.

## II. Background

This section first identifies computational and programming requirements in climate analysis, thereafter introduces the MASS library as our agent-based data-analyzing tool, and differentiates MASS from related work.

### A. Requirements in Climate Analysis

In an effort to understand the coming climate changes and to warn humanity, climate scientists have taken measures to predict the future state of the earth's climate. Several different climate models are produced by climate science facilities in the form of common data sets such as NetCDF files which contain grid-like data suitable for analysis. Unidata has several different software packages available for working with NetCDF data from different software environments such as C++ and Java [4].

A calculation of interest for climate scientists is Time of Emergence (ToE). ToE is the time at which certain climate properties become apparent. In other words, when a certain climate attribute such as temperature starts consistently rising above a certain threshold, then it could be considered ToE for that property [12]. ToE is important to predict, because it is the perceived indicator that could be a warning that extreme weather events could be increasing in frequency, such as storms or floods.

There are challenges with analyzing this climate model data however. The amount of data can be very large and consist of several files. Any given climate model may include up to half a dozen files of 5 - 10GB size. Special computing hardware and advanced computer science concepts may sometimes be necessary to process and analyze the data without significant performance problems. Some of the techniques employed to speed up reading and processing of this data may include distributed file reading, incremental reading and processing of data on a single computing node, or potentially distributed computing clusters to keep the entire data set in memory during analysis. While large climate science facilities may have special computing resources to perform these analyses, many climate researchers are without these resources or skills [13]. For these reasons, climate scientists tend to rely on tools which are familiar to them and have a lower barrier to entry such as the NCAR Command Language (NCL) / Climate Data Operators (CDO) [5]. Often climate scientists will create one-time use scripts which have problems of their own. Because climate scientists are not necessarily software-focused, they will often create NCL code which may suit a particular purpose for one time use, rather than solving a problem in a more re-usable software workflow format [13]. Also the NCL scripting language has limitations of its own such as not being able to take advantage of distributed systems or multi-threaded mechanisms which more modern programming languages enjoy for performance improvement.

## B. *MASS Library*

We use MASS: a parallel-computing library for multi-agent spatial simulation to parallelize climate analysis. MASS abstracts out many difficult computer science concepts such as parallelization and distributed computing. With MASS, users can easily declare a large-size grid data structure that will be distributed out among many computing nodes. The users can then deploy many agents onto the distributed data structure for applying computational operations to the data and computing necessary results.

As shown in Fig. 1, the MASS library forms a collection of communicating multithreaded processes, each maintaining a different portion of distributed arrays and exchanging mobile agents with other processes. The library hides such underlying parallelization and computing platforms from the user's viewpoint. Users are given the MASS programming framework that abstracts distributed arrays and mobile agents with *Places* and *Agents*. Each *Place* element is automatically mapped to one of computing nodes, is located with a logical array index, and is capable of invoking a given function in parallel as well as exchanging data with other elements. On the other hand, each *Agent* object can autonomously migrate from one to another place, spawn children, and interact with the current place where it resides. A user designs a data-analyzing program by extending the Place and Agent base classes and specifying some behavior. Actual computation is performed between MASS.init() and MASS.finish(), using the following major methods, each performed in parallel.

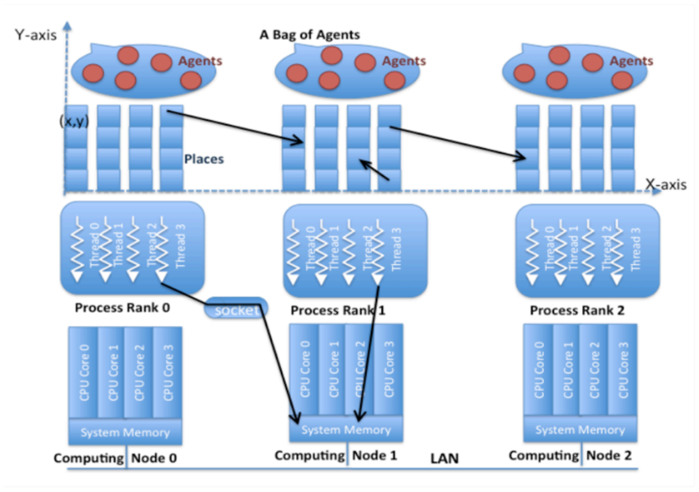


Fig. 1. The MASS architecture

***Places Class***

| |
|---|
| ***public Places(int handle, String className, int size...)*** instantiates a shared array with *size* from *className*. |
| ***public Object[] callAll(int functionId, Object[] arguments)*** calls the method specified with *functionId* of all elements as passing *arguments[i]* to element[i], and receives a return value into *Object[i]*. |
| ***public void exchangeAll(int handle, int functionId, Vector<int[]> destinations)*** calls from each element to a given method of all neighbors, each indexed with a *Vector* element, and exchanges data among the elements. |

***Place Class***

| |
|---|
| ***private size[]; private index[]*** maintains the size of the shared array that each element belongs to and the index of each array element. |

***Agents Class***

| |
|---|
| ***public Agents(int handle, String className, Places places)*** populates agents from *className* onto a given places. |
| ***public Object callAll(int functionId, Object[] arguments)*** is the same as *Places.callAll()*. |
| ***public void manageAll()*** updates each agent's status, based on its latest calls of *migrate()*, *spawn()*, and *kill()*. These methods are invoked within *callAll()*. |

***Agent Class***

| |
|---|
| ***migrate(int[] index...); spawn(int nChildren); kill( )*** moves a calling *Agent* to a place specified with *index*, spawns children, and terminates the agent respectively. |

## C. *Other Potential Parallelization Tools*

Table I compares the MASS library with the major software libraries that have enabled cloud-based or cluster-based data analysis: MapReduce [1], Spark [2], Storm [3], GlobalArray [7], and mobile agents [14]. Note that Table I also shows how SciHadoop [8] and SciSpark [9] address manipulations of structured data on top of Hadoop and Spark respectively.

MapReduce [1] provides users with a simple programming framework of map() and reduce(): the former performs parallel computation onto each data item of distributed files and the latter then collects results from the computation. In the lambda architecture [15], MapReduce serves at the batch layer to create various batch views in expectation of their future uses by the service layer, and therefore its main goal is background processing rather than high-speed analysis.

Spark [2] serves at both batch and speed layers not only to create batch views but also to incorporate new data into real time views, and thus focuses on high-speed analysis. Spark reads a dataset into a cluster system's distributed memory using micro-batch streaming, transforms data into another resilient distributed dataset, and performs computational actions to the in-memory data using various parallel-commutating primitives. Although Spark implements flexible data analysis with lambda expressions, it does not introduce or dynamically link new programs to the runtime data analysis.

Storm [3] facilitates a real-time data streaming framework based on a directed acyclic graph that can be described with its topology builder's setSpout() and setBolt() methods and executed with a fault-tolerant queue-worker model. This actually means that Storm can apply multiple different programs to multiple data (MPMD), whereas MapReduce and Spark apply the same single program to multiple data (SPMD). It supports cumulative computation as guaranteeing at-least-once semantics. However, due to its FIFO-based nature of data analysis, it cannot reverse data streaming or roll back data that have already been processed, which is not suitable to spatial analysis.

Focusing on distributed transparency, all of MapReduce, Spark, and Storm are based on only interpretive language platforms such as Java, Python, and/or Scala. They support holistic data measurement. One of their drawbacks is in handling an entire dataset as a collection of uniform primitive elements. To address this problem, SciHadoop [8] automatically partitions a structured file (e.g., NetCDF) into small chunks and groups them such that MapReduce can process structured data as unstructured partitioned blocks. However, they do not construct the original data structures in memory, have difficulty in understanding spatial relationships and patterns, and do not keep track of particular data items during the course of their analysis. Similarly, SciSpark [9] converts structured files (including not only NetCDF but also HDF) into a collection of Spark-readable data frames named *sciTensors*, each including key/value pairs and array data. Therefore, Spark can repetitively manipulate multiple array datasets at runtime. Needless to say, such data conversion must be materialized by user-provided partitioning and file-loader functions.

GlobalArray facilitates a native-mode data analysis that instantiates distributed arrays on top of MPI. The library provides one-sided access to data and parallel math operations. Although GlobalArray would be the fastest execution environment to support multi-dimensional structured datasets, users must have a substantial knowledge of parallel programming, (e.g. data synchronization) and be aware of their underlying platforms such as MPI ranks.

Another approach to scientific data analysis is to use a database as a distributed array. SciDB [16] maintains a large-scale array-based database using multiple disks. The array is multi-dimensional and capable of storing multiple data types as well as accepting arithmetical operations and relational queries in parallel. SciDB not only strictly divides an array into sub-arrays but also appends to each sub-array its neighbors' boundary elements as overlapping chucks, so that each computing node can access *ghost space*s without remote disk accesses. However, when implementing climate data, SciDB has the following two drawbacks: (1) data items are normally maintained in secondary storage while repeatedly accessed items are cached in main memory, and (2) the immutable semantics is used to create a new array every time when a series of arithmetic operations and queries is applied to the original array. These drawbacks slow down the computation.

Ideas of applying multi-agents or mobile agents for information retrieval or data analysis is not brand-new. Their main objective is to dispatch agents to and let them autonomously

| Features | MASS | MapReduce (SciHadoop) | Spark (SciSpark) |
|---|---|---|---|
| Data streaming | Parallel /tmp accesses | Hadoop [17] | Micro-batch streaming |
| File format | ***Structured: e.g. NetCDF*** | Text, key/value (Struct data flattened) | Text, CSV, key/value (Struct data converted to sciTensor) |
| Data structure | Arrays | Key/value | SQL |
| Data processing | Batch, Runtime | Batch | Micro-batch, Runtime |
| Execution model | MPMD, ***reactive agents*** | SPMD map/reduce | SPMD master-workers |
| Major functions | callAll(), exchangeAll(), manageAll() | map(), reduce(), | count(), reduce(), join(), map() |
| Runtime analysis | ***Dynamic linking*** | No | Lambda expression |
| Platforms | Java, C++, CUDA | Java | Java, Python, Scala |
| Execution speed | ***Interpretive & native exec.*** | Interpretive execution | 100x faster than MapReduce |

| Features | Storm | GlobalArray | Mobile agents |
|---|---|---|---|
| Data streaming | Micro-batch streaming | MPI/IO | Conventional file I/Os |
| File format | Text | Structured: MPI file view | Text |
| Data structure | Tuples | Arrays | Serializable objects |
| Data processing | Micro-batch, runtime | Batch | Batch |
| Execution model | MPMD queues-workers | SPMD master-workers | Cognitive agents |
| Major functions | setSpout and setBolt | get(), put(), pre-defined matrix methods | dispatch() clone() |
| Runtime analysis | No | No | Plug & play |
| Platforms | Java Python | C/C++, Fortran etc. | Java, Tcl/Tk Prolog |
| Execution speed | Interpretive execution | Fastest native execution | Interpretive execution |

TABLE I
TOOLS FOR PARALLEL AND DISTRIBUTED DATA ANALYSIS

interact with remote servers that maintain data of interest. This form of remote analysis relieved scientists from numerous interactions with data servers or from fine controls of remote objects to survey. For instance, D'Agents demonstrated their potential parallelism to retrieve remote server data in a distributed manner [14]. Remote Agents in NASA Research have integrated task planning, scheduling, and robust execution into agents that can autonomously perform remote operations in space [18]. However, these conventional agent systems handle independent, coarse-grained, and cognitive agents. While they are multithreaded and even run in a distributed environment, they are not focusing on analyzing the same data set in collaboration of many reactive agents that run over a cluster system.

As emphasized in bold in Table I, the MASS library has a combination of the following four advantages over the other software tools: (1) handling structured scientific data, (2) applying a collective group behavior of reactive agents for data analysis, (3) supporting runtime analysis with introducing new agents, and (4) facilitating both interpretive and native executions in the same programming model.

We understand that scientific workflow frameworks such as Anduril [19] and OnlineHPC [20] orchestrate a collection of data-analyzing tools, each addressing a different type of scientific data. Rather than competing with them, the MASS library can be included as one of their plug-ins and facilitate fast parallel computation.

To demonstrate the effectiveness and practicability of the MASS-based data-analyzing features, we have implemented the UWCA (University of Washington Climate Analysis) system that analyzes historical climate data in NetCDF. The remainder of this paper focuses on UWCA.

## III. Implementation

UWCA is a web-interfaced climate analyzing system that parallelizes ToE computation over a given historical climate data, using the MASS library. The following explains its functional overview, process architecture, and agent-based ToE computation.

### A. Functional Overview

UWCA was created with the needs of climate science stakeholders in mind, along with computing resource constraints. The main idea of the application is to provide an easy way for stakeholders in the climate science domain to be able to run ToE calculations without having to know the underlying logic or implementation details of the calculation. This is done by providing a simple GUI with three main features to allow a user to be able to submit calculation jobs, view the status of those jobs, and retrieve the results through file downloads. The provenance features provided allow users to review the details of the job submitted in terms of what data was used, how it was processed, and how long that took.

A normal use case of UWCA would be an Environmental Protection Agency (EPA) worker wanting to know about future temperature trends to determine whether or not to commit to funding to a university climate science department. By opening the UWCA web page, the EPA worker can select the temperature-based ToE variable, and then select the input climate model data set to use for the calculation. The user is also able to set the parameters for the calculation, such as temperature threshold (which is explained in more detail in the GUI section). The user can then submit the job which is placed into a work queue. Because the calculations are computing resource intensive, one job is executed at a time on the computing cluster. Fortunately the user can see the status of the job submitted on the web page at any time. Once the job starts running, the user can download the provenance file at any time to view what steps of the calculation have already been done, how long those took, and the parameters used. Once the calculation is complete, the user can download and save all of the files for viewing and later referencing.

The above use case is a good example of how UWCA accomplishes its goal of giving climate science stakeholders access to complicated ToE algorithms that normally would be out of their reach due to technical challenges.

*1) Graphical User Interface:* The GUI for UWCA was designed to be simple, while still providing all the features for submitting ToE jobs, viewing the GUI status of those jobs, and being able to download the files produced at any time.

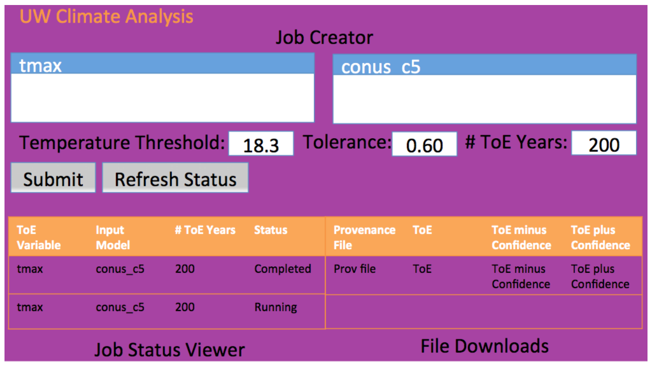
Fig. 2 shows the GUI which consists of three main parts:

1) Job Creator: The left section of this area allows for selection of the ToE variable to be calculated. Once this variable is selected, the climate models and parameters selections become available and may be adjusted to suit the desired ToE calculation. Once the user is satisfied with the selections, they may hit the submit button to begin execution of the job.
2) Job Status Viewer: Gives the user a view into which ToE calculations are or have been calculated on the computing cluster. It shows which ToE variable was selected, which climate model was used, what other parameters were given specifically for the calculation, and what the status is of the job to be executed.
3) File Downloads: This area provides the links to the provenance log file and the output ToE files which are a result of the calculation performed.

The ToE variable that UWCA currently analyzes is tasmax. Tasmax is a temperature-based variable used in climate analysis. The three parameters available for the tasmax calculation are:

1) Temperature Threshold: the value to compare the day temperature value to, in order to discover if that day was over the threshold.
2) Tolerance: the range used to select minimum and maximum days over temperature values based on historical periods.
3) Number of ToE years: the amount of years to project into the future when performing the final ToE calculation.

All values will be defaulted to acceptable ranges and recorded in the provenance log if the user fails to enter valid



UW Climate Analysis

Job Creator

tmax | conus c5

Temperature Threshold: 18.3 Tolerance: 0.60 # ToE Years: 200

| ToE Variable | Input Model | # ToE Years | Status | Provenance File | ToE | ToE minus Confidence | ToE plus Confidence |
|---|---|---|---|---|---|---|---|
| tmax | conus_c5 | 200 | Completed | Prov file | ToE | ToE minus Confidence | ToE plus Confidence |
| tmax | conus_c5 | 200 | Running | | | | |

Job Status Viewer

File Downloads

Fig. 2. The UWCA graphical user interface

values.

*2) Provenance Features:* UWCA features a simple provenance collection mechanism which logs many of the events which happen within the application. The following information is collected from the application and logged to a provenance file:

- Timestamps for each event
- Climate model input files used for the ToE calculation
- Parameters used
- Steps executed for the ToE calculation
- Output files
- Overall execution time in seconds

The file serves the purpose of recording what happened when. As long as a user is familiar with the ToE calculation steps, the provenance log file is understandable.

*3) Data Visualization:* The downloaded ToE NetCDF files can be viewed using several different free pieces of software such as Panoply Viewer or ncBrowse. They each have their own way of visualizing the data which is meaningful. Fig. 3 shows an example output of a ToE file using the ncBrowse software. The colors indicate the year in which each latitude and longitude coordinates hit ToE.

## *B. UWCA Process Architecture*

Fig. 4 depicts the process architecture that consists of two main modules: the UWCA web server and MASS program.

The web server is designed of Servlet and Enterprise Java Bean (EJB). The servlets simply handle GUI operations and pass a request from a client, (i.e., a climatologist) off to the EJB module to be processed by the Job Runner class. The Job Runner class runs in a separate thread, handles the dispatching of the user-requested job to the MASS library, waits until the MASS library has completed the requested operations, and then updates the Job Manager with the output of the calculation. The Job Runner then requests a new job to be processed by the MASS library from the Job Manager, if one is available.

The UWCA MASS program deploys MASS processes, each running at a different cluster node, creates *Places* distributed arrays over the cluster system, reads into the arrays NetCDF files either from an NFS server or each node's */tmp* local disk,

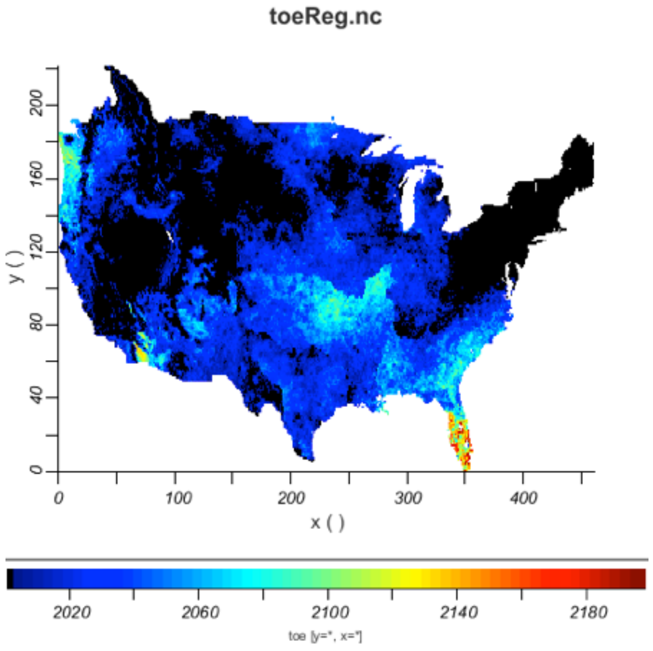


Fig. 3. An example of UWCA computation outputs

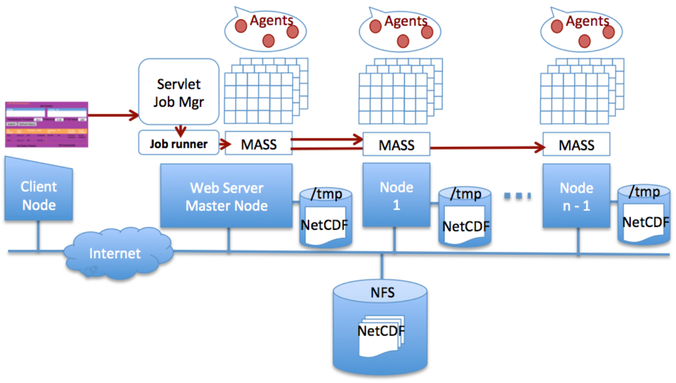


Fig. 4. The UWCA process architecture

depending on a user-provided option, and finally populates *Agents* on the *Places*. *Agents* transverse the arrays to compute and return a ToE value to the MASS main process that passes the value to the Job Runner.

## *C. Time of Emergence*

Although many different climate properties can be analyzed using the ToE calculation such as precipitation and humidity, for this example we will perform the ToE calculation to analyze future temperatures. Agents were used extensively in this calculation for a few reasons:

- Agents provide the functionality to move from place to place regardless of what computing node the place is on, while collecting data necessary to calculate different sub variables which are necessary for the ToE calculation.
- Ease of functional understanding of the algorithm is increased by understanding agent movement across the

```
1  import MASS.*; // MASS Library
2  public class UWCAMain {
3    public void main(Sting args[]) {
4      MASS.init(args);
5      Places dataset = new Places(1, "ToE", 222, 462, 150);
6      Agents crawler = new Agents(2, "Crawer", 1, 102564);
7      for (int time=0; time<150; time++) {
8        crawler.callAll(crawl_);
9        crawler.manageAll();
10     }
11     MASS.finish();
12 } }
13 public class Crawler extends Agent {
14   // data members
15   private int days;
16   private float temperature;
17   // functions
18   public Object callMethod(int funcID, Object args) {
19      switch(funcID) {
20      case crawl_: return crawl(args);
21      }
22   }
23   public Object crawl(Object args) { ...; }
24 }
```

Fig. 5. MASS agent framework

data grid during certain ToE steps.

- The MASS library makes agent programming very easy to accomplish through its well documented API's.

Fig. 5 presents our MASS agent framework. The UWCA Job Runner invokes *UWCAMain.main()* upon receiving a new job (line 3). *MASS.Init()* forks remote processes on Node 1 through to $n-1$ (line 4). Thereafter, the *main* function loads a climate data set in *Places* (line 5) which are distributed over the cluster. *Agents* are spawned on the places (line 6) and repeat migrating over the data set (lines 7-9). The actual agent code is shown in lines 13-24. Every time *main()* invokes *crawler.callAll* (line 8), the MASS library picks up each agent and calls its *callMethod* (line 18), so that the agent calls *crawl()* (line 20) to decide where to move. The actual agent migration is carried out by *crawler.manageAll()* at once (line 9).

The details of agent-based ToE calculation are given in the following steps:

*1) Finding days over threshold:* In this step the input climate model data is transformed from the day-based temperature values, into z dimension grid cells which represent a particular year (from 1950 - 2099), the values in those cells, and the amount of days over a specified temperature threshold (user-defined parameter). Approximately 365-6 time dimension elements are transformed into one element representing the amount of days over threshold.

*2) Finding historical tolerances:* Step 2 analyzes a 50-year historical period to analyze minimum and maximum values. For this calculation, $x \times y$ MASS Agents are spawned at the z[0] dimension (see Fig. 6) which represents the year 1950, and travel down to the z[49] dimension representing the year 1999, collecting days over threshold values. The collected values are analyzed and the maximum value is multiplied by a user-defined percentage such as 90% to find the maximum value. The minimum value is found in a similar manner. For example, if the calculation was decided to be done with

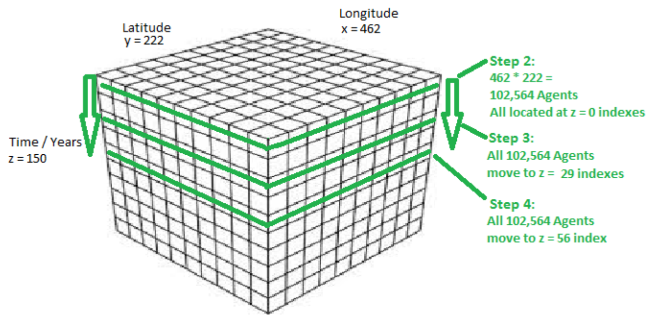


Fig. 6. Agents marching through a three-dimensional climate dataset

an 80% min/max range, the minimum value was 0, and the maximum was 100, then the calculated minimum value would be 10%, and the maximum value would be 90%.

*3) Finding climatology:* Step 3 moves the same agents from the old position, to a new z-index position representing year 1980. The agents then travel along the z dimension, collecting 30 years of values, adding them together. The total is then divided by 30 to get the average. This average represents the climatology.

*4) Computing Least Squared Regression:* Step 4 moves the same agents from the ending step-3 positions to new z-index positions representing year 2006. The agents travel all the way down to 2099 gathering days over threshold values. For each latitude and longitude coordinate, the slope and confidence intervals are calculated for the set of values collected by the agents. Step 4 results in three 2-dimensional arrays

- Slopes for each latitude and longitude x and y coordinates
- Slopes + Confidence Interval for each latitude and longitude x and y coordinates
- Slopes - Confidence Interval for each latitude and longitude x and y coordinates

*5) Finding ToE:* For step 5, three 3-dimensional arrays are created using the values derived from previous steps (see Fig. 7). For each of the arrays, the z[0] index becomes the climatology value derived for that latitude and longitude coordinate. Beyond the z[0] element the following pattern is used:

The amount of z-dimension elements created is determined by a user parameter, but is usually 200. The continual adding of the slopes and confidence intervals results in a positive or negative trend which, when projected out far enough into the future, will cross the minimum or maximum values determined in step 2. The element (which represents a year in the future) at which the value exceeds the minimum or maximum values represents the ToE year. The final output of the ToE calculation will be three 2-dimensional arrays which will be the same $x \times y$ dimensions as above, but will contain the year at which that grid cell exceeded the minimum or maximum values.

The functional implementation of the calculations in UWCA is an advancement in the area of ToE calculations within the climate science domain for several reasons. The performance increase over the comparable CDO/NCL scripts is incredible.

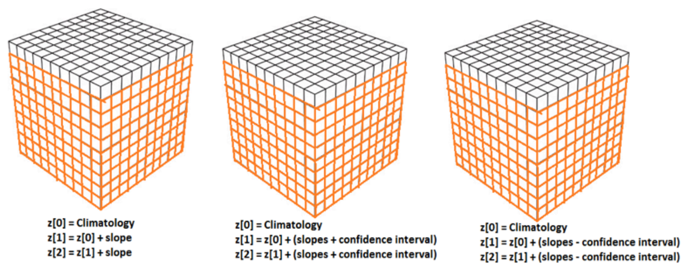

z[0] = Climatology
z[1] = z[0] + slope
z[2] = z[1] + slope
z[0] = Climatology
z[1] = z[0] + (slopes + confidence interval)
z[2] = z[1] + (slopes + confidence interval)
z[0] = Climatology
z[1] = z[0] + (slopes - confidence interval)
z[2] = z[1] + (slopes - confidence interval)

Fig. 7. Three-dimensional ToE arrays

Also the architecture of UWCA allows for extensibility of new ToE calculations without major redevelopment efforts. And most importantly it allows more casual climate science stakeholders and less technically experienced people to easily perform ToE calculations.

## IV. Practicability Analysis

This section qualitatively evaluates how the MASS library can be practically used in climate change analysis. This practicability evaluation considers the following four items that we claimed as the MASS advantages in Section II-C: (1) handling structured scientific data, (2) taking an agent-based approach, (3) supporting runtime analysis, and (4) facilitating both interpretive and native executions.

### A. Handling Structured Scientific Data

MASS and GlobalArray can maintain multi-dimensional data sets such as NetCDF data in their distributed arrays. In contrast, MapReduce mainly handles text files and key/value pairs. As an extension to manipulate structured data, Sci-Hadoop [8] automatically partitions a structured file into small chunks, which allows MapReduce to still handle unstructured data. However, this partitioning and regrouping must be repeated for each operation. Spark can process CVS and SQL schemas but not three or more dimensional arrays. Similarly, Storm focuses on streamed data but not structured data.

The biggest difference between MASS and GlobalArray in structured data is their inter-element communication. Although GlobalArray provides one-sided communication operations such as put/get, accumulate, and read-and-increment, as shown below, a user has to code a parallelization-aware program to let each computing node retrieve a different portion of array elements (lines 2-7), process each data item (lines 8-10), and save updates back to the original elements (line 11).

```
1   GA::GlobalArray *array = GA::SERVICES.createGA(...);
2   myId = GA_Nodeid;()                    // each computing node id
3   int low[2];                            // this node's lower bound
4   int hi[2];                             // this node's upper bound
5   low[0] = myId * N; low[1] = 0;
6   hi[0] = (myId + 1) * N - 1; hi[1] = N - 1;
7   array->get( lo, hi, data, ... );// retrieve data
8   for ( int i = 0; i < N; i++ )    // process each data item
9      for ( int j = 0; j < N; j++ )
10         data[i][j] = func( data[i][j] );
11   array->put( lo, hi, buf, ... ); // save updates
```

On the other hand, MASS invokes a given function call at each array element in parallel, only using a single callAll() statement.

```
1   Places *array = new Places( 1, "MyArray" );
2   array->callAll( MyArray.func_ );
```

Besides multi-dimensional arrays, we may also consider graphs as structured data (although they are seldom used in climate analyses). As MapReduce deals with social networks as one of its targets, many MapReduce programs have been introduced to solve graph algorithms such as graph path planning and page rank [21]. Their programming style uses map() to compute the state of each vertex, relays the update to all its neighboring vertices, (thus uses combiner() as graph links), and collects all updates at each destination vertex in reduce(). Iterative MapReduce must be used to flood such updates entirely over a graph. Spark has enabled graph computation, using GraphX [22] in the form of Pregel [23] that partitions a given graph into subgraphs, each allocated to a different worker machine. Each machine invokes the compute() function at all vertices within the given subgraph to update their states. Similarly to MapReduce, each vertex must exchange messages with their neighbors.

Contrary to these vertex-oriented approaches, MASS takes a flow-oriented approach where agents migrate over a graph represented in an adjacency matrix, (i.e., a two-dimensional MASS places). We believe that this approach would work more intuitively for graphs, where scientists write their programs from a car driver's viewpoint, in other words: as if computations drive from one to another vertex [24].

### B. Taking an Agent-Based Approach

In MASS, computation flows along time-series data as an agent migrates from one to another data item. Contrary to that, MapReduce, Spark, and GlobalArray processes all data items in a batch. For instance, consider to compute the total sum of data items. As shown in the following code, MapReduce simply retrieves all data items in map() (lines 3-4), and thereafter sums up all the items in reduce() (lines 10-11).

```
1  class Mapper {
2     method map( docid a, dataset d ) {
3        for ( item t : dataset d ) {// retrieve all data
4           collect( t, t.data );      // pass data to reduce
5     }
6  }
7  class Reducer {
8     method reduce( item t, counts [c1, c2, ...] ) {
9        int sum = 0;
10        for ( int c : counts [c1, c2, ...] ) {
11           sum += c;                      // add each data to sum
12        }
13     }
14     collect( t, sum );                   // write the final sum in disk
15  }
```

However, it is challenging for MapReduce to stop this computation in the middle when the intermediate sum reaches a given threshold. On the other hand, as shown below, MASS walks a group of agents, each migrating from one to another place (line 16) as summing up its data item (line 10) and eventually terminating itself upon reaching the threshold (lines 11-14). This code mimics a scientist's behavior that skims over a dataset from top to bottom, which thus facilitates intuitive programming. In our ToE computation, agents can go back and forth through a given three-dimensional climate dataset frequently, which would not be concisely implemented in other software tools.

```
Places array = new Places( 1, "MyArray" );
Agents agents = new Agents( 2, "Crawler", 1, ...);
wihle ( agents.getPopulation( ) > 0 ) {// until any agents exist
   agents.callAll( Crawler.sum_ );// repeat their migration.
   agents.manageAll( );
}
class Crawler extends Agent {
   private mySum = 0;
   public void sum( ) {
       mySum += place.data;         // add data to my sum
       if ( mySum >= threshold ) {// reaching the threshold
          MASS_log( "[" + place.index[0] + ","
                    + place.index[1] + "]" );
          kill( );                  // print out the result. I'm done.
       } else
          migrate( );               visit a next place
 }  }
```

## C. Supporting Runtime Analysis

Since Spark can run on top of interactive script languages such as Python, it naturally facilitates runtime analysis. A user can upload a Resilient Distributed Dataset (RDD) anytime (line 3 in the code below), invoke a built-in analyzing function (line 4), and even execute a lambda expression on the fly (lines 5-6). However, when it comes to Java-based cluster computing, a user must compile his/her Java programs *a priori* and submit byte code as a batched job.

```
1 # A modified sample code from http://spark.apache.org/docs/latest/quick-start.html
2 ./bin/pyspark
3 >>> textFile = sc.textFile("ToE.md")
4 >>> textFile.count()
5 >>> textFile.map(lambd line: len(line.split())).
6 ...    reduce(lambda a, b: a if (a > b) else b)
```

Contrary to that, the MASS library supports runtime analysis even on a cluster system. It separates the main program from Place and Agent definitions. The main program behaves as a framework to instantiate Places and Agents and to orchestrate their method calls. Since MASS dynamically links necessary Place and Agent code to the main program, main() does not even have to assume any specific scenario of data analysis. In the code below, the main program includes a loop that keeps receiving the name of a new agent class (line 7), so that a user can repetitively inject new instances to the same in-memory datasets anytime during his/her analysis (line 10). This in turn means that users can change their ensemble data analysis at run time.

```
1  import MASS.*;
2  public class Analysis {
3    public void main(String args[]) {
4      MASS.init(args);
5      Places dataset = new Place(1, "ToE", 222, 464, 150);
6      Scanner keyboard = new Scanner(System.in);
7      while (keyboard.hasNext()) {// read a user input such as:
8        String aName = keyboard.next(); // new agent class name
9        int nAgents = keyboard.nextInt();// #agents to populate
10       Agents agents = new Agents(2, aName, 1, nAgents);
11       while (population > 0) { // keep walking agents until
12         agents.callAll(crawl_);// they are done.
13         agents.manageAll();
14 } } } }
```

In fact, our web-based GUI supports this runtime refinement of data analysis by adapting various agents with respect to climate analysis. MASS can speed up the ToE computation with its parallelization, and therefore time spent on recording/capturing provenance search with agents is quite acceptable for real-time analysis.

## D. Facilitating Interpretive and Native Executions

Both interpretive and native executions in a single programming paradigm makes MASS attractive for fast mock-up and gradual performance tune-up. At present, MASS facilitates the same programming framework in the two different languages: Java and C++. Therefore, users can quickly mock up their scenario of data analysis in Java and thereafter gradually tune up the execution performance of their data analysis in C++. On the other hand, Spark and MapReduce are available only at the interpretive language level: Java, Python, or Stella. If users want to speed up their data analysis, they have to choose a different programming model such as MPI and GlobalArray, which requires additional time to re-write their analysis.

## V. Performance Analysis

To evaluate the performance of MASS-parallelized ToE computation, we used two computing systems: (1) a stand-alone machine of 32GB RAM, 1TB HDD, 8 CPU (*Intel Xeon*) cores, each with 2 hardware threads running at 1.6GHz, and (2) a Giga-Ethernet cluster of 16 computing nodes, each with a 16GB RAM, 500GB HDD 4-core CPU (*Intel i7-3770*) running at 3.40Hz. The former was used as the cluster head for measuring the overall execution of centralized file-reading and in-memory ToE analysis, whereas the latter evaluated the performance of decentralized file-reading and memory-only ToE analysis. The Java environment used was Java 1.7.0_60 with Java HotSpot 64-Bit Server VM 24.60. Java runtime flags were set to increase the starting and maximum heap size to 1GB and 8GB respectively (using *-Xms1g* and *-Xmx8g*).

In the following, we will discuss the performance improvements by and challenges for the MASS library in structured data analysis.

## A. Performance Improvements by MASS

Fig. 8 compares the overall performance of file-reading and in-memory ToE analysis among (1) the original CDO/NCL script, (2) the corresponding MASS-multiprocessed ToE computation over a cluster system, and (3) MASS-multithreaded version at a stand-alone machine of 32GB RAM. CDO/NCL is only specialized to handle NetCDF files in sequential. Therefore, its actual ToE computation was intolerably slow for real-time analysis (which took 21 minutes). On the other hand, MASS-multiprocessed ToE completed an entire execution within eight minutes or shorter. Since this version uses a cluster of computing nodes, each with only 16GB RAM, it cannot load all 22GB NetCDF data onto a single node's memory. Furthermore, the multiprocessed ToE needs to distribute NetCDF files to each cluster node's */tmp* disk whose speed is 3Gbps, twice slower than the stand-alone

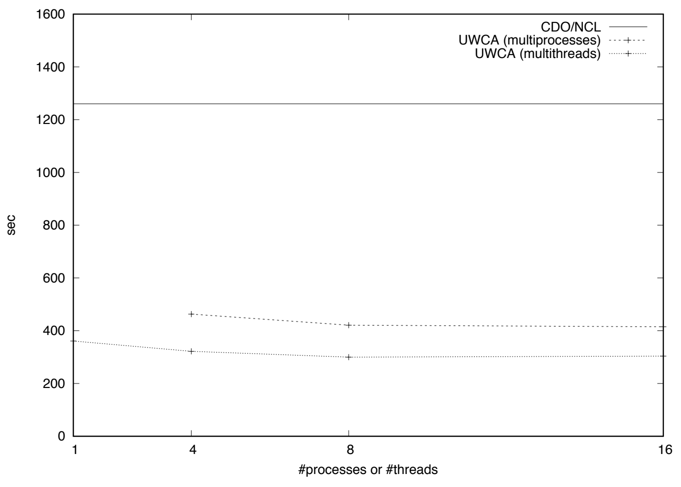


Fig. 8. The overall execution performance of file-reading and in-memory ToE analysis

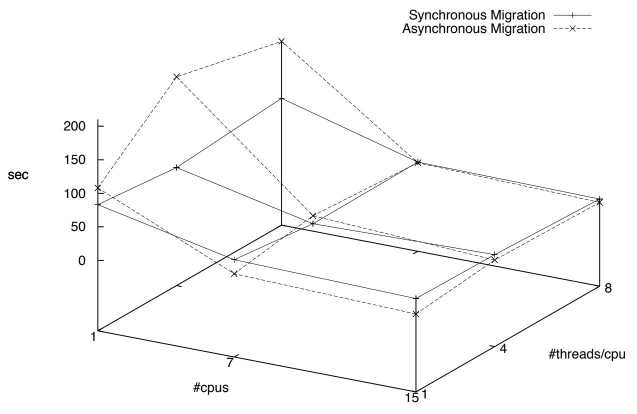


Fig. 9. The execution performance of memory-only ToE analysis

machine, which results in considerable file-reading overheads. Therefore, we also ran the MASS-multithreaded ToE version that demonstrated the fastest execution within six minutes or shorter. However, the multithreaded execution did not yet remarkably show scalable performance. This is because the entire execution was still bound up to the 6Gbps disk performance. Although this file-reading operations are considered as a one-time ramp-up before the predominant body of repetitive in-memory analyses, they make the main program a bottleneck of data streaming.

Fig. 9 excluded this bottleneck problem for the present and focused on the performance of memory-only ToE parallelization. We measured the execution as increasing the number of cluster nodes as well as the degree of multithreading. For this evaluation, we used two different strategies of agent migration: (1) synchronous and (2) asynchronous migration. Synchronous migration is the original MASS implementation where Agents.manageAll() performs migration of all agents in a batch. It works better for moving a large number of agents to a different computing node at once, but incurs master-slave communication overheads every time manageAll() is invoked. In contrast, asynchronous migration needs no invocation of manageAll() so that agents can migrate to another place at any time. It obviously mitigates manageAll-incurred master-slave communication while it must detect so-called distributed termination of agents, which is expensive in particular when using only a few computing nodes. Fig. 9 verifies our expectation and demonstrates the scalable performance of asynchronous migration. Synchronous migration performed 1.3 times faster than asynchronous migration with a single-threaded single computing node, however their performance is reversed with seven or more computing nodes. Eventually, asynchronous migration with 15 computing nodes completes a ToE computation in 11 seconds, thus performing 7.5 times faster than synchronous migration with a single node (in 83 seconds).

## B. *Performance Challenges for MASS*

There are still performance challenges for the MASS-parallelized ToE computation to become even more scalable. As shown in Fig. 10, MASS-multiprocessed ToE spent 75% of the entire execution time for distributed file-reading operations. In addition, agents needed 1.5GB memory for the last 90-second computation. To address these challenges, we are planning to implement the following three performance-improvement solutions in the MASS library:

1) *Removing overheads of file read into memory:* For each computing node, we will have the first place read the node-allocated data into memory so that the other places on the same node can access their data quickly without competing file reads.
2) *Significant heap consideration:* Memory usage can be improved by pooling agents to alleviate actual agent creations/terminations.
3) *Optimizing the number of active agents:* Rather than create $x \times y$ agents, we should automatically spawn the same number of agents as the underlying CPU cores to reduce agent management overheads.

## VI. Conclusions

We have applied an agent-based approach to parallelizing the analysis of structured scientific datasets. For the practicability verification, we parallelized discoveries of future time of emergence, using the MASS library. We demonstrated that our agent-based approach has the four practicability advantages including (1) simple code of agent-based structured data analysis, (2) intuitive data analysis by walking agents freely

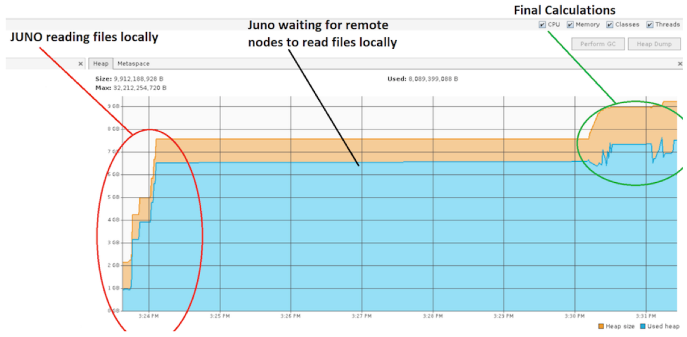


Fig. 10. MASS overheads in file reading

over datasets, (3) runtime analysis by injecting new agents to datasets, and (4) fast mock-up in Java and gradual performance tune-up in C++. We have also confirmed that the MASS execution performance is scalable with up to 15 computing nodes but has the following challenges to address: (1) parallel file reading, (2) pool of agents and (3) optimization of the number of active agents. These improvement plans will allow us to scale up the problem size of climate analysis, using more computing nodes. In addition to these performance improvement tasks, we are also working on the MASS library's debugger that allows users to visualize on-going computation and to modify the states of active agents and array data. We also plan to incorporate many of the provenance techniques developed in previous work [13].

In this paper, we particularly focused on temperature-based ToE computation. This is because their variables are easy to work with as they generally have a trend in the positive direction for all grid cells. For further parallelization of climate change analysis, we are planning to examine precipitation and other hydro-climate variables that show spatial heterogeneity and temporal variability. This will make the analysis and programming more difficult. However, extreme precipitation amounts based on historical thresholds is noteworthy to climatologists. Therefore, it is worthwhile continuing our work on enhancing the UWCA system.

## Acknowledgment

We are very grateful to Mr. Hung Ho, a UWB graduate student for assisting us in improving the MASS library's performance with automatic, asynchronous agent migration. We also thank Mr. David Guenther and Mr. Rafael Silva, CSS advanced system engineers for setting up and maintaining our Linux machines to measure the MASS execution performance. This work is supported by NSF CAREER ACI 1350724: "iProvenance: Integrating Data Provenance with Software Traceability".

## References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proc. of the 6th Symposium on Operating System Design and Implementation - OSDI'4*. San Francisco, CA: Publisher, December 2004, pp. 137–150.
[2] Apache Spark - Lightning-Fast Cluster Computing, "http://spark.apache.org/."
[3] Apache Storm, distributed and fault-tolerant realtime computation, "http://storm.apache.org/."
[4] Unidata | NetCDF, "http://www.unidata.ucar.edu/software/netcdf/."
[5] NCAR Command Language, "http://www.ncl.ucar.edu/."
[6] Message Passing Interface, "https://computing.llnl.gov/tutorials/mpi/."
[7] J. Nieplocha, B. Palmer, V. Tipparaju, H. Trease, and E. Apra, "Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit," *International Journal of High Performance Computing Applications*, vol. Vol.20, no. No.2, pp. 203–231, 2006.
[8] J. B. Buck, N. Wtkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt, "SciHadoop: Array-based query processing in hadoop," in *SC'11 Proc. of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. Seattle, WA: IEEE, November 2011, p. Articl# 66.
[9] P. Palamuttam, R. M. Mogrovejo, C. Mattmann, B. Wilson, K. Whitehall, R. Verma, L. McGibbney, and P. Ramirez, "SciSpark: Applying Inmemory Distributed Computing to Weather Event Detection and Tracking," in *Proc. of 2015 IEEE International Conference on Big Data*, Santa Clara, CA, November 2015, pp. 1959–1965.
[10] T. Chuang and M. Fukuda, "A Parallel Multi-Agent Spatial Simulation Environment for Cluster Systems," in *Proc. 16th IEEE International Conference on Computational Science and Engineering - CSE2013*. Sydney, Australia: IEEE CS, December 2013, pp. 140–153.
[11] E. P. Salathé Jr., A. F. Hamlet, C. F. Mass, S.-Y. Lee, M. Stumbaugh, and R. Steed, "Estimates of Twenty-First-Century Flood Risk in the Pacific Northwest Based on Regional Climate Model Simulations," *Journal of Hydrometeorology*, vol. Vol.15, no. Issue 5, pp. 1881–1899, October 2014.
[12] E. Hawkins and R. Sutton, "Time of emergence of climate signals," *Geophysical Research Letters*, vol. Vol.39, no. No.1, January 2012.
[13] B. Yasutake, N. Simonson, J. Woodring, N. Duncan, W. Pfeffer, H. Asuncion, M. Fukuda, and E. Salathé, "Supporting Provenance in Climate Science Research," in *Proc. 7the International Conference on Information, Process, and Knowledge Management - eKnow 2015*, Lisbon, Portugal, February 22-27 2015.
[14] R. S. Gray, G. Cybenko, D. Kotz, R. A. Peterson, and D. Rus, "D'Agents: applications and performance of a mobile-agent system," *Software – Practice and Experience*, vol. Vol.32, no. No.6, pp. 543–573, May 2002.
[15] N. Marz and J. Warren, *Big Data*. Mannig, 2015.
[16] P. G. Brown, "Overview of SciDB: Large Scale Array Storage, Processing and Analysis," in *Proc. of the 2010 ACM SIGMOD International Conference on Mangment of Data*. Indianapolis, IN: ACM, June 2010, pp. 963–968.
[17] Apache Hadoop, "http://hadoop.apache.org/."
[18] N. Muscettola, P. P. Nayak, B. Pell, and B. C. Williams, "Remote Agent: to boldly go where no AI system has gone before," *Artificial Intelligence*, vol. Vol.103, no. 1-2, pp. 5–47, April 2001.
[19] ANDURIL Workflow Platform, "http://www.anduril.org/anduril/site/."
[20] Workspace, "https://research.csiro.au/workspace/."
[21] J. Lin and C. Dyer, *Data-Intensive Text Processing with MapReduce*. Williston, VT: Morgan & Claypool Publishers, 2010.
[22] Apache Spark GraphX, "http://spark.apache.org/graphx/."
[23] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," in *Proc. of SIGMOD'10*. Indianapolis, IN: ACM, June 2010, pp. 135–145.
[24] M. Kipps, W. Kim, and M. Fukuda, "Agent and Spatial Based Parallelization of Biological Network Motif Search," in *17th IEEE International Conference on High Performance Computing and Communications - HPCC 2015*, New York, August 24-26 2015.