Chapter xxx

SCIENTIFIC SERVICES ON THE CLOUD

David Chapman, Karuna P Joshi, Yelena Yesha, Milt Halem, Yaacov Yesha and Phuong Nguyen
Computer Science and Electrical Engineering Department
University of Maryland, Baltimore County, MD, USA

1. Introduction

Scientific Computing was one of the first every applications for parallel and distributed computation. To this date, scientific applications remain some of the most compute intensive, and have inspired creation of petaflop compute infrastructure such as the Oak Ridge Jaguar and Los Alamos RoadRunner. Large dedicated hardware infrastructure has become both a blessing and a curse to the scientific community. Scientists are interested in cloud computing for much the same reason as businesses and other professionals. The hardware is provided, maintained, and administrated by a third party. Software abstraction and virtualization provide reliability, and fault tolerance. Graduated fees allow for multi-scale prototyping and execution. Cloud computing resources are only a few clicks away, and by far the easiest high performance distributed platform to gain access to. There may still be dedicated infrastructure for ultra-scale science, but the cloud can easily play a major part of the scientific computing initiative.

Scientific cloud computing is an intricate waltz of compute abstract programming models, scientific algorithms, and virtualized services. On one end, highly compute intensive scientific data algorithms are implemented upon cloud programming platforms such as Map Reduce and Dryad, while on the other, service discovery and execution implement the bigger picture with data product dependencies, service chaining, and virtualization.

The cloud of science services is very tightly knit. It is difficult to make meaningful scientific discoveries from only a single data product. Yet even individual data products are produced from other products, which in turn require even more different products for calibration. Service chaining is essential to the scientific cloud, just as much as with the business cloud. However, scientific cloud computing's distinctive feature is data processing and experimentation; a compute elephant hiding underneath the service oriented architecture. Both the service lifecycle, and the processing platforms are key ingredients to a successful scientific cloud

computation. We discuss both fronts from the perspective of an atmospheric cloud computing system Service Oriented Atmospheric Radiances (SOAR). We also make sure to touch on many related cloud technologies, even if they were not necessarily the best fit for our SOAR system.

1.1. Outline

There are two ends to a scientific cloud, the back end and the front end. We briefly describe the *Service Oriented Atmospheric Radiances (SOAR)* system in the next section. The third section on *Scientific Programming Paradigms (back end)* describes how programming platforms affect the scientific algorithms. The fourth section discusses the *Scientific Computing Services* that form the front end, describing in detail how service virtualization affects scientific repositories.
We have found Map Reduce and Dryad to be highly effective platforms for more than our own algorithms. We summarize our own and others' work to apply these paradigms to science related problems.
We also describe a five phase service lifecycle, but in the perspective of scientific applications, and address some of the unique challenges that set science apart from other service domains.

2. Service Oriented Atmospheric Radiances (SOAR)

SOAR, a joint project between NASA, NOAA, and UMBC, is a scalable web service set of tools that provides complex gridding services on-demand for atmospheric radiance data sets from multiple temperature and moisture sounding sensors. SOAR accepts input through an online Graphical User Interface (GUI), or directly from other programs. The server queues these requests for a variety of complex science data services in a database tracking the various requested workflows. It uses large data sets collected by NASA, NOAA and DOD. These datasets contain satellite readings for temperature and moisture from the last three decades. SOAR uses the cloud Bluegrit at University of Maryland Baltimore County (UMBC) to apply data transformations such as gridding, sampling, subsetting and convolving in order to generate derived data sets from diverse atmospheric radiances. [15]

Satellite remote sensing instruments orbit the Earth sun-synchronously to observe temperature, moisture, and other atmospheric structure and properties. SOAR facilitates climate oriented experiments by providing geospatial computations and transformations. This puts SOAR in a unique position, as it must chain with remote servers to acquire data, but as a facilitator, would be well placed within an even deeper chain for complicated scientific experiments.

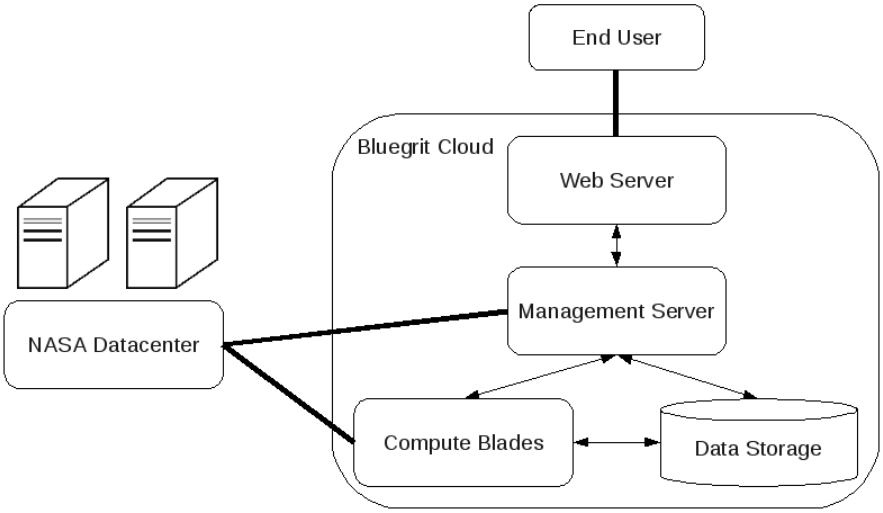

Figure 1: SOAR deployment system.

Figure 1 is a diagram of the SOAR deployment system. It makes use of the compute and data resources of the Bluegrit cloud. End users could be individual scientists, or other data service centers, and could make use of our graphical or SOAP interface provided by Bluegrit's Web server. Service requests encapsulate various climate related experiments such as tracking easterly cloud motion, or generating high resolution planetary images.

Management Server is a task driver for the various compute subsystems. It is responsible for scheduling tasks on the various compute blades. These tasks include precisely geolocated gridding, and singular value decomposition.

The input data for various service computations may not be locally available at the time of request. Management Server and Compute Blades must interact with various NASA data centers to acquire various data products for the requested scientific computations. Additionally Management Server routinely schedules jobs to compute and cache generic intermediate results, such as daily gridded average radiances.

3. Scientific Programming Paradigms

One of the biggest hurdle to unleashing the cloud onto science, is understanding its compute paradigms. The cloud provides a layer abstraction above and beyond the bare system configuration. Cloud abstraction typically arises from distributed middleware and centralized task scheduling. Programming paradigms empower the middleware, and change the way that we program; they force us to think in

parallel. The remainder of this chapter is designed to bend our minds into understanding how to program the cloud for scientific applications. We discuss two programming strategies, MapReduce, and Dryad, and various scientific related problems, and how they could be implemented in a cloud environment.

Map Reduce is a simple programming paradigm for distributed cloud computing. Google begat Map Reduce as a parallel processing solution for its indexing pipeline, and quickly realized that Map Reduce was useful for many more parallel processing chores within the scope of Internet data retrieval and Google now hosts thousands of Map Reduce applications. Although Map Reduce's intended purpose is text analysis and machine learning, it is also useful for many scientific computations, provided however, that they follow certain conditions [7]. This makes Map Reduce a very sharp topic for scientific computing, because it makes easy problems easier, but can potentially make hard problems even harder, if the problem does not fit the paradigm.

Dryad is a flexible programming model based on Directed Acyclic Graphs (DAGs). The nodes represent computation and the edges represent data flow direction. Dryad was developed by Microsoft as a generic paradigm for cloud computing problems, and as an alternative to Google's MapReduce. The Microsoft developers quickly found that Dryad was much more flexible than MapReduce, and as evidence were able to implement relational database, Map and Reduce, and many other software paradigms all completely encapsulated within Dryad's framework. Dryad is well rounded, and perfectly suitable for compute intensive, data intensive, dense, sparse, coupled, and uncoupled tasks. Dryad provides a very flexible solution, and is a good alternative for problems that might not fit well in simpler paradigms such as Map Reduce. On the flip side, Dryad is relatively complicated, and may not be necessary when easier solutions are possible.

3.1. Map Reduce

The original Map Reduce paper, by Dean and Ghemawat in 2004 [3] describes the programming paradigm of Map Reduce very concisely as follows.

"The computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the MapReduce library expresses the computation as two functions: Map and Reduce. Map, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function.
The Reduce function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per Reduce invocation. The intermediate values are supplied to the user's reduce

function via an iterator. This allows us to handle lists of values that are too large to fit in memory."

Other stages can be added to extend this paradigm. As one can see, the paradigm has only two user specified functions: Map and Reduce. A great way to become more familiar with Map Reduce is by example. Word counting is the canonical example, with pseudocode given from Dean and Ghemawat. [33]

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

"The map function emits each word plus an associated count of occurrences (just `1' in this simple example). The reduce function sums together all counts emitted for a particular word.
Thus, by performing Map, to report the occurrence of each word, followed by reduce to sum the number of occurrences for each word, the result of this example is the word count for each distinct word in the document."

By constructing new and different Map and Reduce functions, Map Reduce can be used to solve many problems in addition to word counting. The processing can be performed in parallel, because both the Map and Reduce functions can be performed in parallel. Map acts in parallel on each input element. Reduce acts in parallel on separate KV groups for each distinct key.

3.1.1 Map Reduce Merge

Yang, Dasdan, Lung-Hsiao, and Parker [17] have introduced an improvement to Map-Reduce called Map-Reduce-Merge. This improvement enables better handling of joins on multiple heterogeneous databases, compared with using Map-Reduce. The authors point out that Map-Reduce is good for homogeneous databases. They discuss the problem in performing joins on multiple heterogeneous

databases efficiently and mention that Pike et al. [18] point out that there is quite a lack of fit between Map-Reduce and such joins. In [17], the authors also mention the importance of database operations in search engines. They also described how Map-Reduce-Merge can be applied to relational data processing.

Map-Reduce [3] is described in [17] as follows:
"For each (key,value) pair, Map produces a list of pairs of the form (key',value').Then Reduce is applied to the pairs created by Map as follows:For every key key'' that appears in the output of Map as a key, Reduce applies user defined logic to all the values value'' such that (key'',value'') is one of those pairs, and creates a list of values value'''."

Map-Reduce-Merge is described in [17] in the context of lineages. In Map-Reduce-Merge, Map is modified to operate on each lineage separately. Reduce is modified to operate on each lineage separately, and further modified to create a list of pairs (*key'', value'''*) rather than a list of values value'''.
Also, Merge is added as a third step. Merge is applied to the output of Reduce in two lineages. From the list of values associated with a key key'' in one lineage, and and the list of values associated with a key key''' in another lineage, Reduce creates a list of pairs of the form (key'''',value''''). All the pairs created by Reduce form a new lineage.

3.2. Dryad

Dryad is a programming paradigm and software framework designed around the ideas of task scheduling and data flow. The programmer must create a Directed Acyclic Graph (DAG) that represents the processing task. The graph nodes are compute kernels that run on various processors, and the graph edges represent the data flow dependence. Each graph node becomes available for computation as soon as all input data is available. A centralized job manager schedules available graph nodes onto idle machines. The machine executes the kernel computation, and upon completion, the node passes its output down to its children, and the machine becomes idle once again. The child graph nodes become available for computation as soon as all input data is available from its deceased parents. The computation continues in this manner until the entire DAG is executed and the program terminates.
Isard et. al describe their Job scheduling system with a concise diagram. [8]

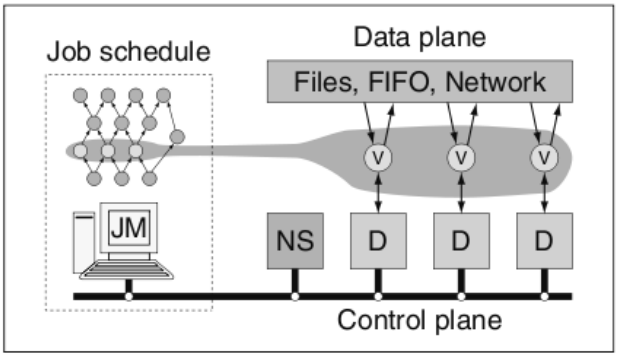

Figure 1: Dryad

The job manager (JM) consults the name server (NS) to discover the list of available computers. It maintains the job graph and schedules running vertices (V) as computers become available using the daemon (D) as a proxy. Vertices exchange data through files, TCP pipes, or shared-memory channels. The shaded bar indicates the vertices in the job that are currently running.

A common Dryad dilemma is that there is often more than one DAG that will satisfy computation of a particular problem. Which DAG is the fastest? Should one implement a very fine DAG with a high degree of parallelism, or a coarse DAG with low scheduling overhead? Sometimes the choice is clear, such as scheduling one node per machine, but often the choice is much more difficult to understand at first glance. Sometimes the best DAG depends on the design of your compute cluster; network and IO hardware design may play a critical role in determining potential bottlenecks in your data flow DAG. These low level issues may begin to contrast the philosophy that the cloud should be completely abstracted from its underlying hardware. Dryad makes it relatively easy for programmers to play around with the structure of the DAG, until they design one that runs efficiently on their target machine.

Additionally, Microsoft has attacked the graph tweaking dilemma head on with a number of automatic graph pruning and optimization algorithms. These techniques execute at runtime on the job scheduler, and thus can make decisions based on up to date profiles and resource availability. One such algorithm can make the DAG coarser by encapsulating a smaller subgraph to within a single node with serial execution. Although encapsulation makes the system less parallel, it can greatly improve performance in situations where the graph was designed too finely. Another technique is to automatically make data reductions hierarchical. This can greatly improve performance, by reducing the data volume before sending packets to other machines and across racks.

3.3. Remote Sensing Geo-reprojection

Atmospheric gridding and geo-reprojection is a great example of a single pass scientific problem, which is well suited for the Map Reduce, and Dryad programming paradigms. Satellite remote sensing instruments measure blackbody radiation from various regions on earth, to determine weather and climate related forecasting, as well as supply atmospheric models with raw data for assimilation. The geo-reprojection algorithm is one of the first major compute steps along the chain of any satellite atmospheric prediction. The satellite observes a surface temperature, 316K (43C ~109F) that's hot! But where is it? New Mexico? Libya? Geo-reprojection solves this task by producing a gridded map of Earth with average observed temperatures or radiances.

The measured region on Earth is a function of the instrument's position, and the direction that it is observing. Figure 2 is a diagram of the NASA Atmospheric Infrared Sounder (AIRS) satellite instrument. The satellite has a sun synchronous orbit while the Earth is rotating, so on a Lat Lon, projection, the joint scan pattern is illustrated by the blue striping on the left. The instrument measures many observations during flight, as the sensor quickly oscillates between -48.95 and 48.95 degrees taking 90 observations every 2.7 seconds.

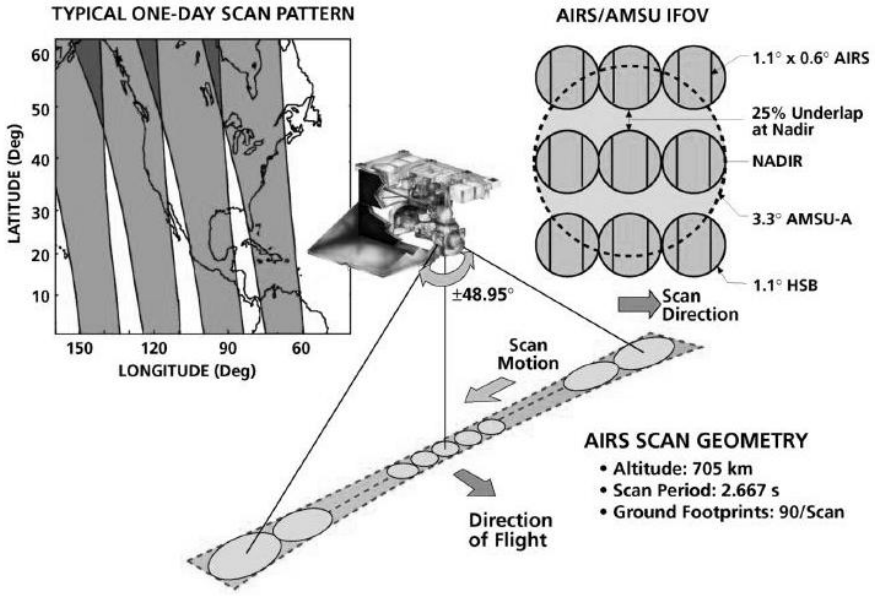

Figure 2: Geo-reprojection

The map of Earth is uniformly divided into a number of regions, or grid cells. The goal is to have a measured temperature or radiance for each cell on the grid. When the satellite measures the same grid cell more than once, the resulting temperatures are averaged together.

3.3.1. Remote Sensing Geo-reprojection with Map Reduce

The Map Reduce program for Geo-reprojection is similar in structure to the canonical word counting problem, provided we ignore details about computational geometry and sensor optics. Rather than counting words, we are averaging grid cells. Averaging is only slightly harder than summing, which in turn is only slightly harder than counting. The program has unchanged structure but novel details.

```
map(int timestamp, Measurement measurements):
  // key: timestamp
  // value: a set of instrument observations
  for each measurement m in measurements
    Determine region r containing measurement m
    EmitIntermediate(r, m.value);

reduce(int region, Iterator measurements):
  // key: a region ID
  // value: a set of measurement
  //       values contained in that region
  double result = 0.0;
  for each m in measurements:
    result += m;
  //divide by the total number of measurements
  result = result / measurements.size;
  Emit(result);
```

All of the sensor optics and geometry to determining the appropriate region are glossed over in the above pseudocode, with the line "Determine region r containing measurement m". For more detail see Wolf et. al. [9]. In this simple example, we assume there is only one region per measurement. However, in more realistic re projections, the observation may overlap multiple regions. In such an event, the Map would need to emit partial measurements for each region, and reduce would remain unchanged. Notice, that the final major step of reduce is to divide by the number of measurements (measurements.size). This division transforms the distributed summation to a distributed average, to derive the average measurement of the region.

3.3.2. Remote Sensing Geo-reprojection with Dryad

With Dryad, the remote geo-reprojection task may be computed somewhat differently than with Map Reduce. We will assume that the output grid is comparably of lower resolution than the input data set. This assumption is usually valid for the problem, because the sounding instrument typically observes overlapping regions multiple times within hours to days of observation. Also, for climate related applications, fine grain grids are often not required, allowing for even further data reduction.

Problems that greatly reduce the volume of data are typically well described by a reduction type of graph [8]. The basic generic reduction graph is shown in Figure 3.

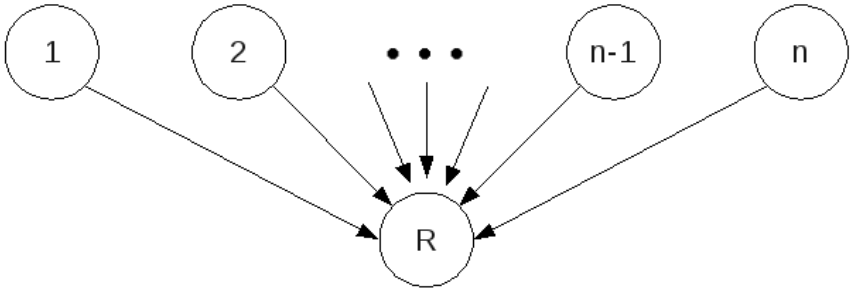

Figure 3: Generic reduction graph

Notice, that there are many reduction graphs that all produce the same result as the one listed above. An example is the one listed in Figure 4, which features a two level hierarchy. Partial reduction nodes r enumerate partial centroids an then pass the result onto the final reduction node R. This approach is more parallel, because there are more independent nodes working to do part of the reduction. Unfortunately, there is also more overhead in this hierarchical approach, because there are more nodes that need to be scheduled.

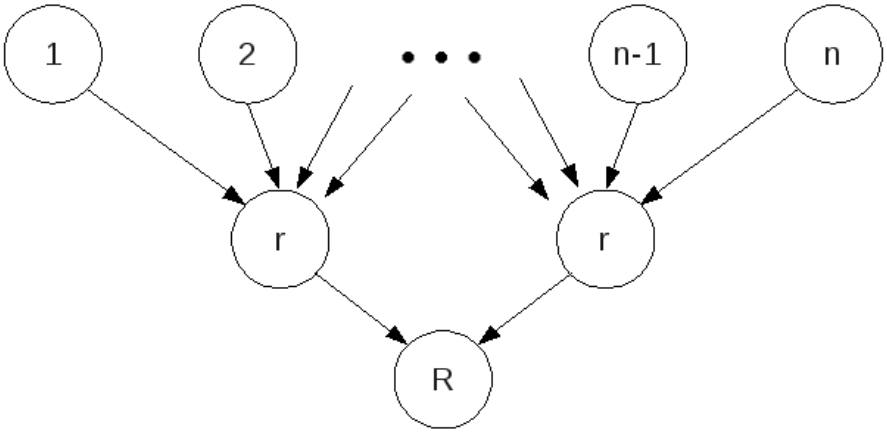

Figure 4: Reduction graph with two level hierarchy.

The following pseudocode could be used for the reduction DAGs described in this section. Start() represents nodes *1-n*, and reduce() represents nodes *r* and *R*.

```
start():
  // input0: instrument measurements
  Measurement []measurements = ReadChannel(0);
  // make an empty array of gridcell regions
  Region []regions = new EmptyRegions();
  //put each measurement in the region
  for each measurement m in measurements
    Determine region r containing measurement m
    r.result += m;
    r.count  += 1;
  //write the region array out to the channel zero
  WriteChannel(newCentroids, 0);

reduce():
  // input0-n: region arrays
  Region [][]regions;
  for every input channel i
    regions[i] = ReadChannel(i);
  //a single region array for the results
  Region []results = new EmptyRegions();
  //accumulate all of the regions together
  for every input channel i
    for every region j in regions[i]
```

```
    results[j].result += regions[i][j].result
    results[j].count  += regions[i][j].count
//divide, to perform the averaging
for every region j in results
  results[j].result /= results[j].count
  //don't double divide if we reduce multiple times
  results[j].count = 1
//We're done, write results to output channel 0
WriteChannel(results, 0);
```

3.4. K-Means Clustering

Clustering is an essential component of many scientific computations, including gene classification [10], and N body physics simulations [11]. The goal of clustering is to separate a number of multidimensional data points into N groups, based on their positions relative to the other points in the dataset. K-Means clustering uses the concept of the centroid, or average position of all of the points in this group, to define the cluster. Initially, the points are grouped randomly into clusters. K-Means iteratively refines these clusters until it converges to a stable clustering.

K-Means uses the cluster centroid (average position), to determine the cluster grouping. A single iteration of K-Means is as follows:

1. Compute cluster centroid (average all points) for each cluster
2. Reassign all points to the cluster with the closest centroid
3. Test for convergence

3.4.1. K-Means Clustering with Map Reduce

K-Means clustering is an iterative process that is a good candidate for Map Reduce. Map Reduce would be used for the centroid and clustering computation performed within each iteration. One would call Map Reduce inside "if a" loop (until convergence), in order to compute iterative methods such as K-Means clustering.

The primary reason why K-Means is a reasonable Map Reduce candidate, is because it displays a vast amount of data independence. The centroid computation is essentially distributed average, which is a small variation on the distributed summation, as exemplified by canonical word counting. Distributed summations require straightforward list reduction operations. The reassignment of points to clus-

ters, requires only the current point and all cluster centroids; the points can be assigned independently of one another. Below is pseudo-code for K-Means clustering using Map Reduce.

The Map function takes as input a number of points, and the list of centroids, and from this it produces a list of partial centroids. These partial centroids are aggregated in the reduce function, and used in the next iteration of the Map Reduce

```
map(void, {Centroid []centroids, Point []datapoints}):
  // key: not important
  // value: list of centroids and datapoints
  Centroid []newCentroids;
  Initialize newCentroids to zero
  for each point p in datapoints
    Determine centroid centroids[idx] closest to p
    //accumulate the point to the new centroid
    newCentroids[idx].position += p;
    //we added a point, so remember the
    //total for averaging
    newCentroids[idx].total += 1;
  for each centroid newCentroids[idx] in newCentroids
    //send the intermediate centroids for accumulation
    EmitIntermediate(idx, newCentroids[idx]);

reduce(int index, Centroid []newCentroids):
  // key: centroid index
  // value: set of partial centroids
  Centroid result.position = 0;
  //accumulate the position and total for a grand total
  for each centroid c in newCentroids
    result.position += c.position;
    result.total += c.total;
  //Divide position by total to compute
  // the average centroid
  result.position = result.position / result.total
  Emit(result);
```

3.4.2. K-Means Clustering with Dryad

K-Means can equally well be implemented in the Dryad paradigm and mindset. The main task of programming with the Dryad paradigm is to understand the data

flow of the system. A keen observations about K-Means, is that typically, each cluster has many points. In other words, there are far fewer clusters than there are points. Thus on each iteration the total amount of information is greatly reduced, when determining centroids from the set of points.

A reduction DAG is a good choice for K-Means, because the data volume is reduced. The following pseudocode would be used to perform the graph reduction operations for the K-Means algorithm using Dryad. In the graph reduction diagrams given the section "Remote Sensing Geo-reprojection with Dryad", start() is the function for nodes *1*-*n*, and reduce() is the function for nodes *r* and *R*.

```
start():
  // input0: Complete list of centroids from the prior run
  Centroid []centroids = ReadChannel(0);
  // input1: Partial list of datapoints
  Point []datapoints   = ReadChannel(1);
  // make a new list of centroids
  Centroid []newCentroids;
  Initialize newCentroids to zero
  for each point p in datapoints
    Determine centroid centroids[idx] closest to p
    //accumulate the point to the new centroid
    newCentroids[idx].position += p;
    //we added a point, so remember the
    //total for averaging
    newCentroids[idx].total += 1;
  //send the intermediate centroids for accumulation
  //channel zero is the only output channel we have
  WriteChannel(newCentroids, 0);

reduce():
  // input0-n: list of intermediate centroids
  Centroid [][]newCentroids;
  for every input channel i
    newCentroids[i] = ReadChannel(i);
  // make a list of result centroids
  Centroid []results;
  for every input channel i
    results[i].position = 0;
    //accumulate the position and total for a grand total
    for each centroid c in newCentroids[i]
```

```
    result[i].position += c.position;
    result[i].total += c.total;
  //Divide position by total to compute
  // the average centroid
  results.position = result.position / result.total
 //don't double divide if we reduce multiple times
  results.total = 1
//write our the results to channel 0
WriteChannel(results, 0);
```

3.5. Singular Value Decomposition

Singular value decomposition (SVD) can also be parallelized with cloud computing paradigms. The goal of SVD is very similar to matrix diagonalization. One must describe how a matrix, M, can be represented as the product of three matrices under the conditions as described below:

$$M = U \Sigma V^T$$

Where M is the original m-by-n matrix, U is an m-by-m orthogonal matrix, V^T is a n-by-n orthogonal matrix, and Σ is a m-by-n diagonal matrix.

The idea behind the Jacobi method is to start with the identity $M = I M I$ and attempt to slowly transform this formula into $M = U \Sigma V^T$ by a series of rotations designed to zero out the off-diagonal elements one at a time. Unfortunately, zeroing out one element may un-zero another. However, if this approach is repeated sufficiently, matrix M will converge to the diagonal matrix Σ.

Since the focus of this book is on cloud computing and not matrix algebra, we will not go into detail about the formulas required by the Jacobi and related methods. For further reading, refer to [12][13][14].

The One Sided JRS and Jacobi algorithms, to zero out a single element require modification of two rows within the matrix. A single sweep, for each pair of rows in the matrix, one must compute the dot product with the other rows, and use this value to modify both of the rows. For an n-by-n matrix, there are n(n-1)/2 such pairs of rows. [13]

It is thus natural to partition the matrix into rows for a parallel implementation. Rajasekaran and Song [14] propose a round robin approach where each machine stores two blocks, computes all pairs of rows within each block, and then computes all pairs of rows between the two blocks. Then, the blocks are shuffled to other machines as illustrated in Figure 5.

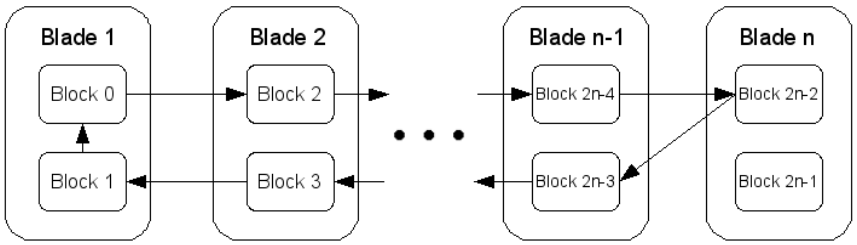

Figure 5: Round robin SVD block communication pattern

Although the aforementioned data access pattern can be implemented straightforwardly on grid systems with message passing, it is equally straightforward to implement with cloud computing paradigms Map Reduce and Dryad. The cloud paradigms still provide additional benefits such as fault tolerance and data abstraction.

3.5.1. Singular Value Decomposition with Map Reduce

The round robin block data pattern described in Figure 5 can be implemented with Map Reduce, but with a single caveat. The difference is that Map Reduce prefers to control how data is distributed based on the key/value pair of the block. Thus the key can be used as a *virtual* machine ID, rather than a *physical* ID. Each block is a key value pair. The `reduce` operation accepts two key value pairs (blocks) modifies them, and emits both of them back as results. This map reduce procedure must be performed iteratively until convergence.

```
map(int blockPos, Block block):
  // key: the current position of the block
  // value: 2D block
  int newBlockPos;
  if (blockPos == 0)
    newBlockPos = 1;
  else if (blockPos = 2n-1)
    newBlockPos = blockPos;
  else if (blockPos == 2n-2)
    newBlockPos = 2n-3;
  else if (blockPos % 2 == 1)
    newBlockPos += 2;
  else // (blockPos % 2 == 0)
    newBlockPos -= 2;
  //key must be equal to the virtual machine ID.
  //however, the slot (top or bottom) is also necessary
```

```
  //to disambiguate the block slots
  int machineID = floor(newBlockPos/2);
  BlockValue blockVal;
  blockVal.block = block;
  blockVal.slot  = newBlockPos % 2;
  EmitIntermediate(machineID, blockVal);

reduce(int machineID, BlockValue [2]blockVals):
  // key: virtual machine ID
  // value: structure with the slot (top or bottom)
  //       and the block data
  //use a convention arrang the blockvals by slot
  if (blockVals[0].slot == 1)
    swap (blockVals[0], blockVals[1]);
  //perform rotations in slot 0
  Block block0 = blockVals[0];
  for i=0 to block0.numRows-1
    for j=i to block0.numRows-1
      rotate(block0.row[i], block0.row[j]);
  //perform rotations in slot 1
  Block block1 = blockVals[1];
  for i=0 to block1.numRows-1
    for j=i to block1.numRows-1
      rotate(block1.row[i], block1.row[j]);
  //perform rotations across both slots
  for i=0 to block0.numRows-1
    for j=0 to block1.numRows-1
      rotate(block0.row[i], block1.row[j]);
  //remember the block position for the next iteration
  int blockPos0 = 2*machineID;
  int blockPos1 = 2*machineID + 1;
  Emit(blockPos0, block0);
  Emit(blockPos1, block1);
```

3.5.2. Singular Value Decomposition with Dryad

Much like Map Reduce, Dryad likes to control how data is distributed via task scheduling. For this reason, it is equally important to use *virtual* machine IDs for the round robin SVD data access pattern. For Dryad, the nodes in the DAG represent virtual machines, and the edges represent the data distribution.

The graph in Figure 5 is *cyclic*. It must be made *acyclic* for use with Dryad. To do so, we must unroll the graph. Unfortunately, the resulting graph would be of infinite length, as one does not know how much iteration must be performed before convergence falls below some error threshold. Fortunately, it is sufficient to make a large finite graph, and simply terminate early upon convergence.

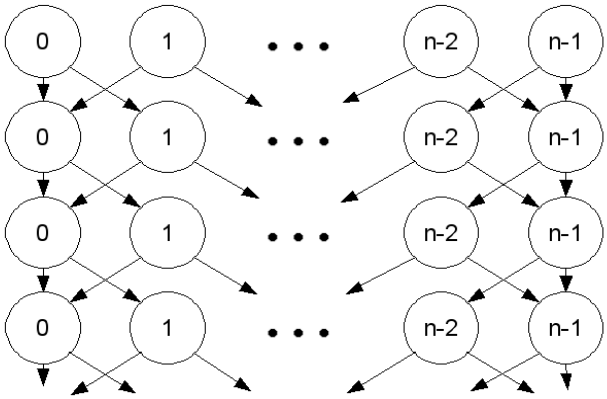

Figure 6: Acyclic SVD round robin block communication pattern

The connectivity of the graph is somewhat intricate in order to achieve the pattern shown in Figure 6. Notice how in Figure 6, every node has two blocks: one on top, and one on bottom. We define that channel 0 reads input to the top block, and input channel 1 reads input from the bottom block. We also define that output channel 0 writes output from the top block, and output channel 1 writes output from the bottom block. At every timestep, both blocks are read, modified, and written to their appropriate channels.

There are several corner cases for the connectivity of the graph. These cases would be handled in graph construction, and are thus not listed in the pseudocode of this section. Tables 1 and 2 show the specific rules of connectivity.

From		To	
Node	Chan	Node	Chan
x	0	x+1	0
x	1	x-1	1

Table 1: Standard cases for connectivity

From		To	
Node	Chan	Node	Chan
0	1	0	0
n-1	0	n-2	1
n-1	1	n-1	1

Table 2: Corner cases for connectivity

The pseudocode within a node to perform the block rotations is listed below.

```
node():
  // input0: Block for top slot
  Block block0 = ReadChannel(0);
  // input1: Block for bottom slot
  Block block1 = ReadChannel(1);
  //perform rotations in slot 0
  for i=0 to block0.numRows-1
    for j=i to block0.numRows-1
      rotate(block0.row[i], block0.row[j]);
  //perform rotations in slot 1
  for i=0 to block1.numRows-1
    for j=i to block1.numRows-1
      rotate(block1.row[i], block1.row[j]);
  //perform rotations across both slots
  for i=0 to block0.numRows-1
    for j=0 to block1.numRows-1
      rotate(block0.row[i], block1.row[j]);
  WriteChannel(block0, 0);
  WriteChannel(block1, 1);
```

4. Delivering Scientific Computing services on the cloud

Extant methodologies for service development do not account for a cloud environment, which includes services composed on demand at short notice. Currently, the service providers decide how the services are bundled together and delivered to service consumers. This is typically done statically by a manual process. There is a need to develop reusable, user-centric mechanisms that will allow the service consumer to specify their desired security or quality related constraints, and have automatic systems at the providers end control the selection, configuration and composition of services. This should be without requiring the consumer to understand the technical aspects of services and service composition.

Service Oriented Atmospheric Radiances (SOAR), demonstrates many examples of the forewarned paradox. Climate scientists want to study the earth's atmospheric profile, and they need satellite observations of sufficient quality for the experiments. It would be futile for them to learn every data processing step required, down to the algorithm version numbers, and the compute architectures used to produce every datum they require. Yet, they care that such a toolchain is well documented somewhere. If a colleague were to disagree with his findings years later, when all of the old data, algorithms and hardware have been upgraded, does the scientist even know the toolchain that produced his old experiments? It has been said that science without reproducibility is not science, yet in a world where data and computations are passed around the intricate cloud, provenance is all to easy to lose track of.

We have proposed a methodology for delivering virtualized services via the cloud [55]. We divide the IT service lifecycle on the cloud into five phases. In sequential order of execution they are requirements, discovery, negotiation, composition, and consumption. Figure 7 illustrates our proposed service lifecycle. Detailed lifecycle illustrating the sub-phases and is available at [55]. We have also developed ontology in OWL for the service lifecycle which can be accessed at [1616].

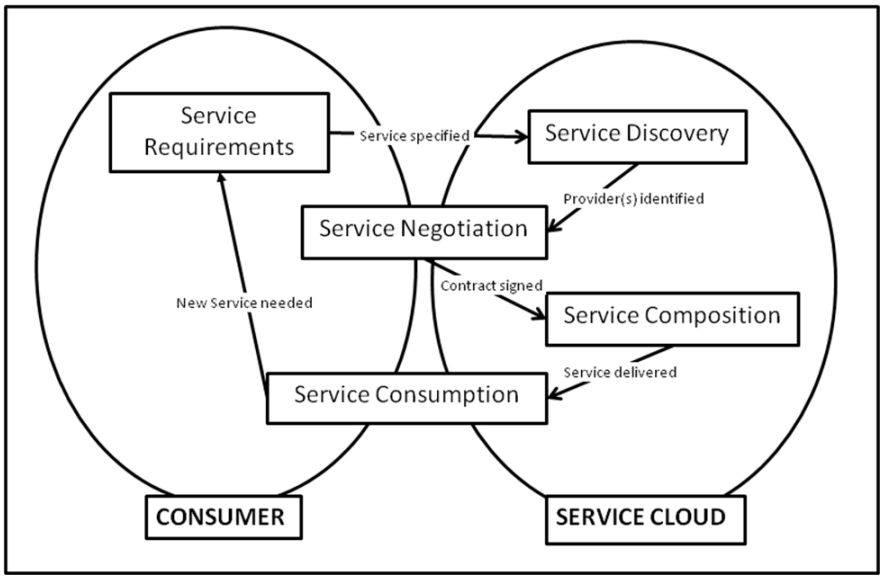

Figure 7: The Service lifecycle on a scientific cloud comprises five phases: requirements, discovery, negotiation, composition and consumption.

4.1. Service Requirements

In the service requirements phase the consumer details the technical and functional specifications that a scientific service needs to fulfill. While defining the service requirements, the consumer specifies not just the functionality, but also non-functional attributes such as constraints and preferences on data quality, service compliance and required security policies for the service. Depending on the service cost and availability, a consumer may be amenable to compromise on the service data quality. For example, a simple service providing images of the Earth might deliver data as images of varying resolution quality. Depending on their requirements, service consumers may be interested in the high resolution images (higher quality) or might be fine with lower image resolution if it results in lower service cost.

Such explicit descriptions are of use not just for the consumer of the service, but also the provider. For instance, the cost of maintaining the service data quality can be optimized depending on the type of data quality requested in the service. The advantage for the service provider is that they will not need to maintain the lower quality data with the same efficiency as the higher quality data; but they would still be able to find consumers for the data. They can separate the data into various databases and make those databases available on demand. As maintainability is a key measure of quality, low maintenance need of the service data will result in improved quality of the service.

The service requirement phase consists of two main sub-phases: service specification and request for service.

Service Specification: In this sub-phase, the consumer identifies the detailed functional and technical specifications of the service needed. Functional specification describe in detail what functions/tasks should a service help automate and the acceptable performance levels of the service software and the service agent (i.e. the human providing the service). The technical specifications lay down the hardware, operating system, application standards and language support policies that a service should adhere to. Specifications also list acceptable security levels, data quality and performance levels of the service agent and the service software. Service compliance details like required certifications, standards to be adhered to etc. are also identified. Depending on the requirements, specifications can be as short or as detailed as needed.

Figure 8 shows all of the Quality of Services (QoS) parameters available to the user for the SOAR Gridded Average Brightness Temperature web service. GUIs make a good human readable compliment to the machine readable description languages used by SOAP and REST style web services. User-centric QoS parameters include variable resolution, and result type (jpg image, or hdf and binary

datafiles). The service providers make their own decisions about the compute resources and related parameters. For example, a single service job may be distributed across a number of nodes, but this type of distributed computing is not necessary for small services that complete very quickly. It is up to the provider to decide how the task is scheduled, and what resources to designate.

Figure 8: Screenshot of the SOAR system GUI for the Gridded Average Brightness Temperature Web Service.

Request for Service: Once the consumers have identified and classified their service needs, they will issue a "Request for Service" (RFS). This request could be made by directly contacting the service providers, such as the "Submit Request" button on Figure 8. This direct approach bypasses any sort of discovery mechanism, but fails when the consumer is unaware of the service provider. Alternative-

ly, consumers can utilize a service search engine on the cloud to procure the desired service, as long as the service is registered with some discovery engine.

Service requirement is a critical phase in service lifecycle as it defines the "what" of the service. It is a combination of the "planning" and "requirements gathering" phases in a traditional software development lifecycle. The consumers will spend the maximum effort in this phase and so it has been depicted entirely in the consumer's area in the lifecycle diagram. The consumer could outsource compilation of technical and functional specifications to another vendor, but the responsibility of successful completion of this phase resides with the consumer and not the service cloud. Once the RFS has been issued, we enter the discovery phase of the service lifecycle.

4.2. Service Discovery

In this phase, service providers that offer the services matching the specifications detailed in the RFS are searched (or discovered) in the cloud. The discovery is constrained by functional and technical attributes defined, and also by the budgetary, security, data quality and agent policies of the consumer.

If the consumer elects the option to search the cloud instead of sending the RFS to a limited set of providers, then the discovery of services is done by using a services search/discovery engine. This engine runs a query against the services registered with a central registry or governing body and matches the domain, data type, functional and technical specifications and returns the result with the service providers matching the maximum number of requirements listed at the top.

The discovery phase may not provide successful results to the consumers and so they will need to either change the specifications or alter their in-house processes to be able to consume a service that meets their needs the most.

If the consumers find the exact scientific service within the budget that they are looking for, they can begin consuming the service. However, often the consumers will get a list of providers who will need to compose a service to meet the consumer's specifications. The consumer will then have to begun negotiations with the service providers which is the next phase of the lifecycle. Each search result will also return the primary provider who will be negotiating with the consumer. It will usually be the provider whose service meets most of the requirement specifications.

4.3. Service Negotiation

Service negotiation phase covers the discussion and agreement that the service provider and consumer have regarding the service delivered and its acceptance criteria. The service delivered is determined by the specifications laid down in the RFS. Service acceptance is usually guided by the Service Level Agreements (SLA) that the service provider and consumer agree upon. SLAs define the service data, delivery mode, agent details, compliance policy, quality and cost of the service. While negotiating the service level with potential service providers, consumers can explicitly specify service quality constraints (data quality, cost, security, response time, etc.) that they require.

Of course, scientists love to work with the most accurate and precise data available, until they run in to practical problems with it. The fine dataset might take too much storage, computation may take too much time, or perhaps out of place algorithms such as principal component analysis may require impractical amounts of RAM. These sorts of problems force scientists to reconsider the level of quality that they actually need for their desired experiment. SLAs will help in determining all such constraints and preferences and will be part of the service contract between the service provider and consumer.

Negotiation requires feedback, and the server must be able to estimate the cost involved with the service and deliver these statistics to the consumer before the service is executed. Although it is possible to negotiate automatically using hill-climbing and related optimization algorithms, SOAR was a service system designed primarily for human end users. Depending on the service and the QoS, the desired SOAR transaction could take seconds, minutes, or hours to compute. It is not necessary to give a precise time estimate, but at least a rough guess must be presented. For example, a warning "This transaction could take hours to compute" would be very helpful.

In addition to warning users about high cost services, the provider must also explain some strategy for reducing the cost. Of course, reducing QoS will reduce cost, but the degree may vary. Some service parameters may have a dramatic effect on the service cost, whereas others have little effect at all. For example, in the SOAR Gridded Average Brightness Temperature service, "resolution" has a tremendous effect on performance, whereas "type" (format) does not affect performance very much. Unfortunately, if the consumer doesn't know which parameters to vary, he may find himself toggling options randomly, or give up on the service entirely out of frustration. Although algorithms can happily toggle options systematically, humans would prefer to know what they are doing. SOAR documents the best known ways to cut cost. Preemptive GUIs may even be possible, that flag a user as soon as he selects a high cost option.

At times, the service provider will need to combine a set of services or compose a service from various components delivered by distinct service providers in order to meet the consumer's requirements. For example, SOAR services often need to interact with NASA datacenters to obtain higher resolution data for processing. The negotiation phase also includes the discussions that the main service provider has with the other component providers. When the services are provided by multiple providers (composite service), the primary provider interfacing with the consumer is responsible for composition of the service. The primary provider will also have to negotiate the Quality of Service (QoS) with the secondary providers to ensure that SLA metrics are met.

Thus the negotiation phase comprises of two critical sub phases, the negotiation of SLAs and negotiation of QoS. If there is a need for composite service, iterative discussions takes place between the consumer and primary provider and the primary provider and component providers. The final product of the negotiation phase is the service contract between the consumer and primary provider and between the primary provider and the component (or secondary) providers. Once the service contract is approved, the lifecycle goes to the composition phase where the service software is compiled and assembled.

4.4. Service Composition

In this phase one or more services provided by one or more providers are combined and delivered as a single service. Service orchestration determines the sequence of the service components. Often the composition may not be a static endeavor, and may even depend on the QoS parameters used in the RFS.

The SOAR gridded average brightness temperature service is a great example of a dynamic composition workflow. Raw daily observations are cached at coarse resolutions, so only local parallel processing is necessary to service such requests. However, if the data specified is not available in cache, then it must be obtained from NASA's Goddard and MODAPS datacenters. Such is an example of service chaining with foreign NASA entities. Depending on the request, this remote execution could take hours to compute, and thus necessitates asynchronous execution. However, by comparison, the coarse local services could be completed in seconds. This is a great example of how the QoS and RFS specification could have such a huge impact on the resulting workflow, performance, and resources.

Many times what is advertised as a single service by a provider could in turn be a virtualized composed service consisting of various components delivered by different providers. The consumer needs to know that the service is composite for accounting purposes only. The provider will have to monitor all the other services that it is dependent on (like database services, network services etc.) to ensure that

the SLAs defined in the previous phase are adhered to, and recorded for scientific reproducibility.

Once the service is composed, it is ready to be delivered to the consumer. The lifecycle then enters the final phase of service consumption.

4.5. Service Consumption and Monitoring

The service is delivered to the consumer based on the delivery mode (synchronous/asynchronous, real-time, batch mode etc.) agreed upon in the negotiation phase. After the service is delivered to the consumer, payment is made for the same. The consumer then begins consuming the service. An important part of the consumption phase includes performance monitoring using automated tools.

In this phase, consumer will require tools that enable quality monitoring and service termination if needed. This will involve alerts to humans or automatic termination based on policies defined using the quality related ontologies that need to be developed. The service monitor sub-phase measures the service quality and compares it with the quality levels defined in the SLA. This phase spans both the consumer and cloud areas as performance monitoring is a joint responsibility. If the consumer is not satisfied with the service quality, s/he should have the option to terminate the service and stop service payment. If the service is terminated, the consumer will have to restart the service lifecycle by again defining the requirements and issuing a RFS.

The performance monitoring tool used in SOAR is shown in Figure 9. The tool not terribly complicated, but is very effective. It shows not only completed tasks, but also those currently in progress. The tool provides a timestamp for submission, so it is easy to monitor how long the request is taking. The user can select a task and remove it, even if it is currently in progress. This allows the user to cancel tasks that are taking too much time. Task cancellation is also accessible via SOAP, and can be used by a machine interface as well as a human interface.

An example gridded brightness temperature image is displayed in Figure 10, this is an example of a simple result generated from the SOAR system. This is a brightness temperature image by the Atmospheric Infrared Sounder (AIRS) of the Indian ocean on Jan 1st 2005. The striping on display is due to the sun synchronous polar orbit, and would be resolved if the data was collected over a longer time period. The warmest regions are in South Africa and Ethiopia. The blue dots around Madagascar, New Guinea and other locations are due to clouds. These types of plots, which show clouds very clearly, can be used in Hovmoller (running mean) diagrams to track cloud movement over time and monitor processes such as the Madden Julian Oscillation, as shown in Figure 11.

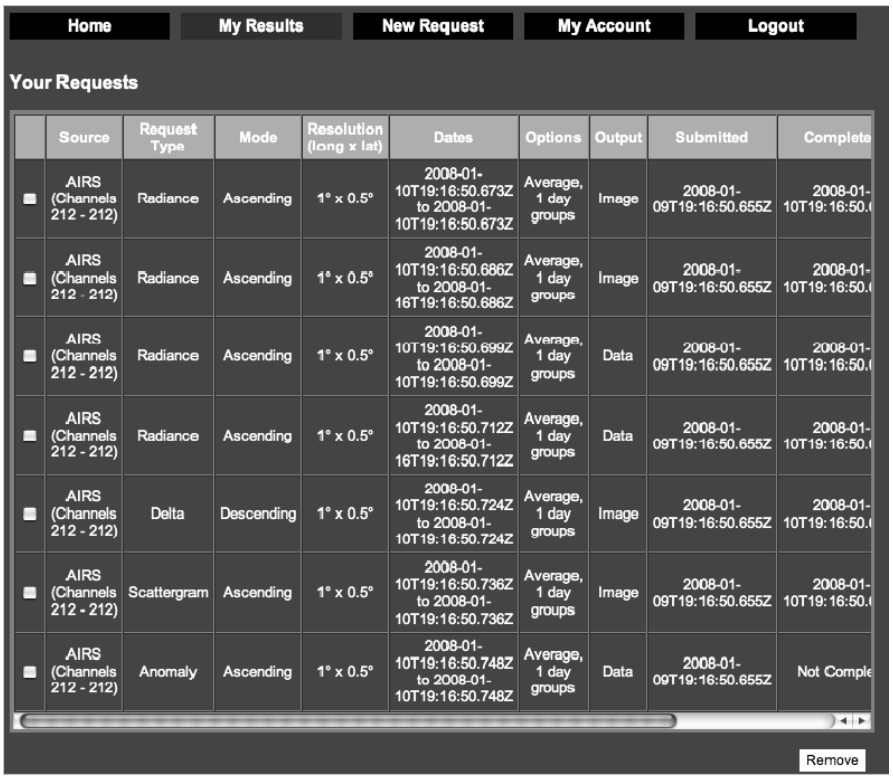

Figure 9: Your Requests tool provided by SOAR

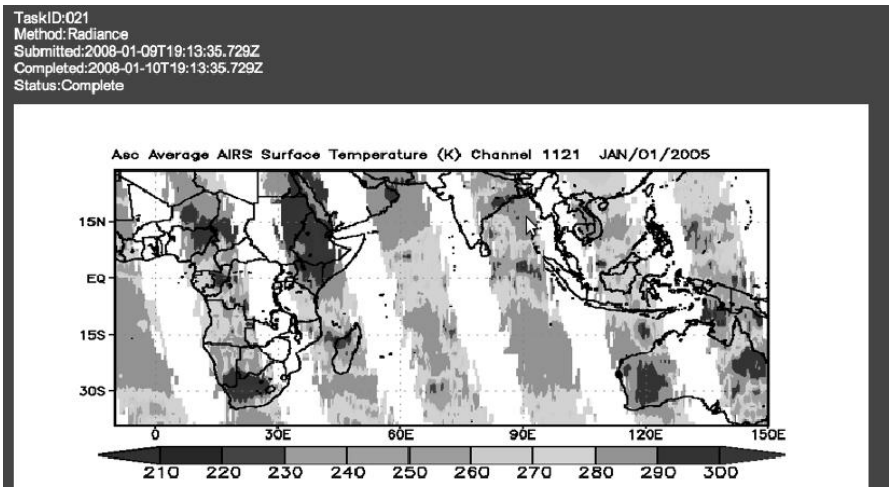

Figure 10: SOAR Generated Image

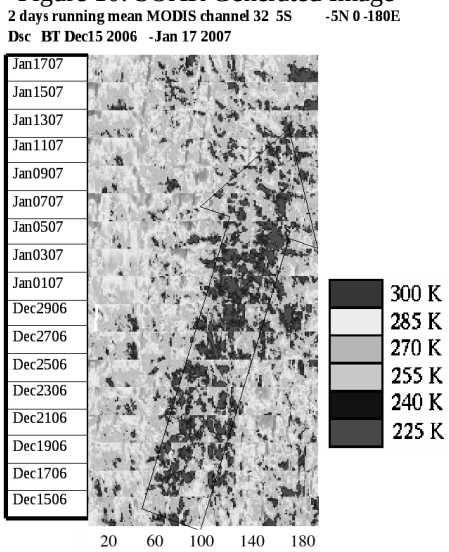

Figure 11: Hovmoller showing easterly cloud movement over the Pacific from Dec 15 2006 to Jan 17 2007

5. Summary / Conclusions

In summary, we believe that the cloud is a new and exciting platform for scientific computing. The compute and services paradigms at first glance may appear counterintuitive. However, once mastered they unleash the benefits cloud provides

flexible compute resources, virtualized web services, software abstraction, and fault tolerance. Map Reduce and Dryad are paradigms that can aid with scientific data processing on the back end, while service discovery and delivery provide not only a front end, but an entire a compute work flow including data selection and quality of service. SOAR is a complete scientific cloud application, and we hope that the lessons we have learned from this and other systems can help others in their scientific ventures.

References

1. U. Alon, N. Barkai, D. A. Notterman, K. Gish, S. Ybarra, D. Mack, and A. J. Levine, “Broad patterns of gene expression revealed by clustering analysis of tumor and normal colon tissues probed by oligonucleotide arrays”, Proceedings of the National Academy of Sciences of the United States of America Vol 96, Issue 12, June 1999, pp 6745-6750
2. Y. M. Marzouk and A. F. Ghoniem, “K-means clustering for optimal partitioning and dynamic load balancing of parallel hierarchical N-body simulations”, Journal of Computational Physics. Volume 207, Issue 2, 10 August 2005, Pages 493-528
3. J. Dean and S Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, J– Proceedings of OSDI, 2004, pp 137-150
4. M. Isard, M. Budiu, Y. Yu, A. Birrell and D. Fetterly, “Dryad: distributed data-parallel programs from sequential building blocks”, Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007.
5. K.P. Joshi, T. Finin, and Y. Yesha, "Integrated Lifecycle of IT Services in a Cloud Environment", Proceedings of The Third International Conference on the Virtual Computing Initiative (ICVCI 2009), October 2009.
6. S Ran, “A model for web services discovery with QoS”, ACM SIGecom Exchanges, Vol 4, Issue 1, 2003, pp 1-10, 2003
7. J. Ekanayake, S. Pallickara, and G. Fox, "Map-Reduce for Data Intensive Scientific Analysis", Proceedings of the IEEE International Conference on e-Science, December 2008.
8. M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks", Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, 2007.
9. R. Wolfe, D. Roy, E. Vermote, "MODIS land data storage, gridding, and compositing methodology: Level 2 grid", IEEE Transactions on Geoscience and Remote Sensing, Vol 36 Issue 4, 1998, pp 1324-1338, 1998
10. U. Alon, N. Barkai, D. Notterman, K. Gish, S. Ybarra, D. Mack, A. Levine, "Broad patterns of gene expression revealed by clustering analysis of tumor and normal colon tissues probed by oligonucleotide arrays", Proceedings of the National Academy of Sciences of the United States of America, June 1999

11. Y. Marzouk, A. Ghoniem, "K-means clustering for optimal partitioning and dynamic load balancing of parallel hierarchical N-body simulations", Journal of Computational Physics. Vol 207, Issue 2, pp 493-528, August 2005
12. V. Klema, A. Laub, "The Singular Value Decomposition: Its Computation and Some Applications.", IEEE Transactions on Automatic Control, Vol 25, Issue 2, April 1980
13. M. Soliman, S. Rajasekaran, R. Ammar, "A Block JRS Algorithm for Highly Parallel Computation of SVDs", High Performance Computing and Communications, Vol 4782, Issue 1, pp 346-357, September 2007
14. S. Rajasekaran, M. Song, "A Novel Scheme for the Parallel Computation of SVDs.", Proceedings of High Performance Computing and Communications, Vol 4208, Issue 1, pp 129–137, 2006
15. M. Halem, N. Most, C. Tilmes, K. Stewart, Y. Yesha, D. Chapmam, P. Nguyen, "Service Oriented Atmospheric Radiances (SOAR): Gridding and Analysis Services for Multi-Sensor Aqua IR Radiance Data for Climate Studies", IEEE Transactions on Geoscience and Remote Sensing, Vol. 47, Issue 1, pp 114-122, 2009
16. K. Joshi, OWL Ontology for Lifecycle of IT Services on the Cloud, http://www.cs.umbc.edu/~kjoshi1/IT_Service_ Ontology.owl
17. Hung-chih Yang, Ali Dasdan, Ruey-Lung-Hsiao and D. Scott Parker, Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters, Proceedings of the 2007 ACM SIGMOD international conference on Management of Data, Beijing, China, June 11-14, 2007, pages 1029-1040.
18. R. Pike, S. Dorward, R. Griesemer,and S. Quinlan, interpreting the Data: Parallel Analysis with Sawzall. Scientific Programming Journal, 2005, pages 227-298.

Index terms (alphabetically):

Dryad
Components
Geo-reprojection
K-Means Clustering
Map Reduce
Remote Sensing
Services
Singular Value Decomposition (SVD)
SOAR