Scalable Aggregation Service for Satellite Remote Sensing Data

Jianwu Wang[1](✉), Xin Huang[1], Jianyu Zheng[2], Chamara Rajapakshe[2], Savio Kay[1], Lakshmi Kandoor[1], Thomas Maxwell[3], and Zhibo Zhang[2]

[1] Department of Information Systems, University of Maryland, Baltimore County, Baltimore, MD 21250, USA
{jianwu,xinh1,savio1,k170}@umbc.edu

[2] Department of Physics, University of Maryland, Baltimore County, Baltimore, MD 21250, USA
{jzheng3,charaj1,zhibo.zhang}@umbc.edu

[3] Goddard Space Flight Center, NASA, Greenbelt, MD 20771, USA
thomas.maxwell@nasa.gov

Abstract. With the advances of satellite remote sensing techniques, we are receiving huge amount of satellite observation data for the Earth. While the data greatly helps Earth scientists on their research, conducting data processing and analytics from the data is getting more and more time consuming and complicated. One common data processing task is to aggregate satellite observation data from original pixel level to latitude-longitude grid level to easily obtain global information and work with global climate models. This paper focuses on how to best aggregate NASA MODIS satellite data products from pixel level to grid level in a distributed environment and provision the aggregation capability as a service for Earth scientists to use easily. We propose three different approaches of parallel data aggregation and employ three parallel platforms (Spark, Dask and MPI) to implement the approaches. We run extensive experiments based on these parallel approaches and platforms on a local cluster to benchmark their differences in execution performance and discuss key factors to achieve good speedup. We also study how to make the provisioned service adaptable to different service libraries and protocols via a unified framework.

Keywords: Big data · Data aggregation · Remote sensing · Servicelization · Benchmark

1 Introduction

The advances in climate study in recent years have resulted in astronomical growth of available climate data. There are two main sources for climate data: climate simulation model and satellite remote sensing. The paper [15] from *Science* magazine estimates the total worldwide available climate data will increase from less than 50 PB in 2015 to about 350 PB in 2030. Among projected data size in 2030, climate model simulation results and satellite remote sensing data consist of about 188 PB (54% percent) and 155 PB (44% percent) respectively.

A basic data processing task in climate study is to aggregate satellite observation data from original pixel level to latitude-longitude grid level to easily obtain global information and work with global climate simulation models. It is because most global climate simulation models, such as the climate simulation models in Phase 6 of the Coupled Model Intercomparison Project, known as CMIP6 [12], conduct simulation by dividing the Earth into 3-dimensional (latitude, longitude, altitude) grids, and solving physics equations (including mass and energy transfer) within each grid and its interactions (including radiant exchange) with neighboring grids. By aggregating satellite observation data from pixel level to grid level, Earth scientists can conduct many studies with climate model simulation data and aggregated satellite observation data since they have the same granularity. For instance, a climate model can be evaluated via the comparison between its simulation results with satellite observation data.

This paper addresses two specific challenges in satellite data aggregation: 1) how to efficiently aggregate data from pixel level to grid level in a distributed environment, 2) how to provision data aggregation as services so Earth scientists can achieve data aggregation without downloading data to local machine. To address these two challenges, we discuss different approaches of parallel data aggregation and how different factors such as big data platform and sampling ratio affect the execution performance and aggregation results. We also discuss how to make provisioned services adaptable to different service libraries and protocols. The software implementations of our work is open-sourced at [2].

The contributions of this paper are fourfold. First, we propose three parallel data aggregation approaches and discuss their differences. All approaches can show good scalability in our experiments with 738 GB input data with the best speedup ratio as 97.03 when running on 10 distributed compute nodes. Second, we apply the above data aggregation approaches with three popular parallel platforms/techniques: Spark [3,9], Dask [4,18] and MPI [16]. We benchmark and analyze their performance differences through our experiments. Third, we apply sampling techniques in our data aggregation approaches to understand how sampling affects execution speedup and the correctness of the results. Our experiments show sampling only has less than 1% data loss and its affects on execution speedup is mixed. Fourth, to work with different service protocols/libraries such as REST [17] and ZeroMQ [7], we adopt the Stratus framework [6] for servicelization of the data aggregation capability. Users only need to change one parameter to switch from one service protocol/library to another.

The rest of the paper is organized as follows. The background is introduced in Sect. 2. The data aggregation logic is explained in Sect. 3. Section 4 contains our proposed three scalable data aggregation approaches. Section 5 describes the experiments on different scalable aggregation approaches, different parallel platforms, and different sampling ratios. The servicelization of our data aggregation capability is discussed in Sect. 6. Finally, we discuss related work in Sect. 7 and conclude our paper in Sect. 8.

2 Background

2.1 MODIS Satellite Remote Sensing Data

The MODIS (Moderate Resolution Imaging Spectroradiometer) is a key instrument on board NASA's Terra (launched in 1999) and Aqua (launched in 2002) satellite missions as part of the larger Earth Observation System (EOS). MODIS measures the reflection and emission by the Earth-Atmosphere system in 36 spectral bands from the visible to thermal infrared with near daily global coverage and high-spatial resolution (250 m to 1 km at nadir). These measurements provide a critical observational basis for understanding global dynamics and processes occurring on the land, in the oceans, and in the lower atmosphere. MODIS is playing a vital role in the development of validated, global, interactive Earth system models that are able to predict global change accurately enough to assist policy makers in making sound decisions concerning the protection of our environment.

MODIS atmosphere properties products are processed into three levels, i.e., Level 1 (L1), Level 2 (L2) and Level 3 (L3). The Level 1 products contain geolocation and the raw reflectance and radiance measurements for all 36 MODIS spectral bands, at 250 m, 500 m, or 1 km spatial resolutions. The Level 2 products contain the geophysical properties, such as cloud mask, cloud and aerosol optical thickness, retrieved from the Level 1 products. The retrieval process is usually based on sophisticated algorithms developed by the MODIS science teams. Because Level 2 products are derived from the Level 1 products, they usually have the same or similar spatial resolution. For example, Level 2 MODIS cloud properties products (product name "MOD06" for Terra and "MYD06" for Aqua) have a nominal spatial resolution of 1 km. The Level 3 processing produces Earth-gridded geophysical parameter statistics, which have been averaged (e.g., daily or monthly), gridded (e.g., $1^\circ \times 1^\circ$ degree), or otherwise rectified or composited in time and space. The Level 3 MODIS Cloud Properties products contain hundreds of $1^\circ \times 1^\circ$ global gridded Scientific Data Sets (SDSs) or statistics derived from the Leval 2 products. The Level 1 and Level 2 products are often called pixel products and the Level 3 products are often called gridded products. Many atmospheric/climate research studies are done using Level 3 data. All three levels of MODIS data are publicly available at [5].

2.2 Parallel Platforms

Spark [3] is one of the most popular big data platform. By following and extending the MapReduce paradigm [10], Spark embeds computation using high-level functions like Map, Reduce, CoGroup and Cross, and achieve parallelism by distributing input data among many parallel tasks of the same function. Spark works well with Hadoop distributed file system (HDFS) to achieve parallel computation on partitioned data. It supports multiple programming languages including Scala, Java and Python. For job scheduling, Spark employs a master process communicating with parallel worker processes, maintains a task queue and distributes the next task in the queue to a worker process after the worker process finishes its current assigned task.

Dask [4,18] is another scalable platform targeted for big data processing and analytics. Similar to Spark, a Dask application is composed as a task graph which can be distributed within one computer or a distributed computing environment. Dask employs a similar master-worker framework for task scheduling. Because Dask is implemented in Python, it is native to work with other Python libraries and packages such as Numpy, XArray, Pandas.

Message Passing Interface (MPI) [16] defines message-passing standard to achieve parallel computing within a distributed computing environment such as a cluster. As a dominant model used in high-performance computing, MPI defines how to distribute tasks to multiple compute nodes and CPU cores and communicate among the parallel processes on synchronization. Unlike Dask and Spark, MPI does not use master-worker architecture. Instead, each process is assigned with a rank number, and the communication and synchronization are done by sending/receiving messages to/from different ranked processes. MPI supports multiple programming languages including C, Fortran and Python.

3 MODIS Data Aggregation Logic

MODIS data aggregation from Level 2 to Level 3 requires collecting all relevant Level 2 data files and calculating corresponding values based on the target variable and the spatial and temporal ranges. In this paper, we focus on monthly global aggregation of one atmospheric variable, called Cloud Fraction. In this section, we will first explain how to do it for a single Level 2 cloud properties product file, namely MYD06/MOD06, then how to combine results from all relevant Level 2 files.

As shown in Fig. 1, the process of generating Level 3 data from Level 2 file involves four main steps. In the first step, it reads one file from MYD06/MOD06 and its corresponding file from MYD03/MOD03 and produces grid-level counts for both cloud pixels and total pixels.

In its first step, it reads 'Cloud_Mask_1km' variable from the MYD06/MOD06 file, and reads 'Latitude' and 'Longitude' variables from the MYD03/MOD03 file. Based on MYD/MOD manual and HDF file convention, each of these three variables is a 2030×1354 2D array. Also, their values of three arrays are 1-to-1 mapped, namely the corresponding longitude and latitude value of each cloud mask value in Cloud_Mask array can be found at Latitude array and Longitude array respectively using the same array index.

The second step of the process is sampling, which only takes partial data from the original 2D (2030×1354) arrays. Based on Earth's geographic coordinates, each latitude grid is about 111 km in distance and each longitude grid is about 0–111 km in distance depends on the longitude's value (0 km for $\pm 90°$ and 111 km for $0°$). It means each (lat, lon) grid covers up to 111 km $\times$ 111 km area (over 12k pixels). Also based on the nature of cloud coverage, if a pixel is cloudy, its surrounding pixels are also likely to be cloudy. To reduce the computing load, we

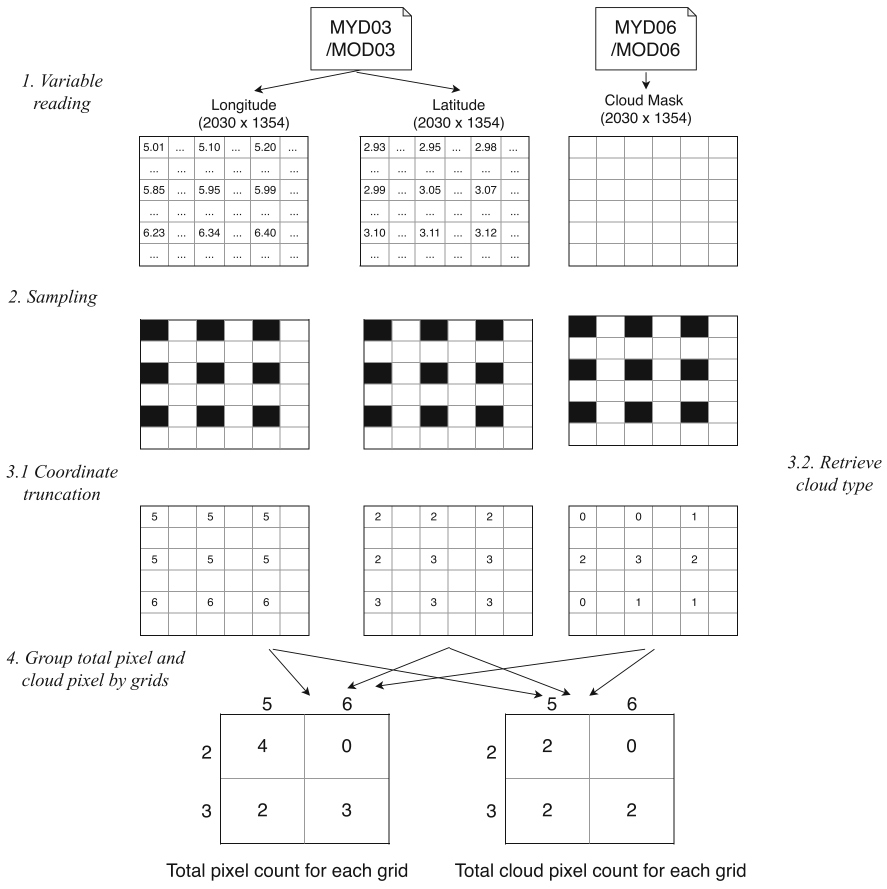

Fig. 1. Illustration of MODIS data aggregation from Level 2 to Level 3 for a single file.

could use different sampling ratios to only aggregate a portion of pixels at 1 km resolution. As shown in the figure, by using sampling ratio 2 for both longitude and latitude, the number of pixels to be aggregated is only 1/4 of original pixels.

The third step is to convert coordinate resolution from Level 2 to Level 3. The original latitude and longitude values in MOD03/MYD03 have precision as 5, while we only need to have integral latitude and longitudes in Level 3 data. So we need to convert coordinate resolution to integer by removing each value's floating part. For instance 25.12345° in Level 2 will be just 25° in Level 3.

The fourth step is to retrieve cloud condition information by doing bit operation. Based on the manual of MOD06/MYD06, there are four types of cloud condition which are encoded as two digits of the values in binary. So by doing bit operation, we can retrieve the values for cloud condition information. Out of the four possible values (0, 1, 2 and 3), only 0 value means the pixel is cloudy. The last step of the process is to calculate the total pixel count and cloud pixel count for each grid. It first group pixels into corresponding grids. Then it counts all pixels within each grid to get the total pixel table and only counts zero value pixels within each grid to get the cloud pixel table.

To get global aggregation results for a month, we need to do the same processing in Fig. 1 for all relevant files and eventually get results for every grid of global 180 × 360 longitude/latitude grids. Every 5 min, the satellite remote sensing retrieval algorithm obtains a snapshot data file, called *granule*, of the area the satellite covers at the time. Each variable measured by the satellite is saved as a 2D pixel array in the granule file. The 2D array size of the MODIS data we work with is 2030 × 1354. Because MODIS Level 2 product contains such as granule files every 5 min, the pixel number for a full day is about 800 million (2030 × 1354 × 288) and the number for a full month is 24.5 billion. The aggregation process will group these 24.5 billion values into 64,800 (180 × 360) grid bins based on their geographic locations and conduct corresponding calculations for the pixels within a grid bin. On average, each bin aggregates values from about 378,700 pixels.

4 Three Scalable MODIS Aggregation Approaches

In this section, we propose three scalable approaches for MODIS data aggregation, each in one subsection. Approach illustrations are in Figs. 2, 3 and 4. The main differences are task granularity and file/record count, where the numbers in parentheses, e.g., (2 × 288 × 31), are the file/record number for the step.

4.1 File Level Scalable MODIS Aggregation

The overall logic of our file level scalable MODIS aggregation approach is shown in Fig. 2. The implementations of the same approach in Spark, Dask and MPI are slightly different because of their programming model difference.

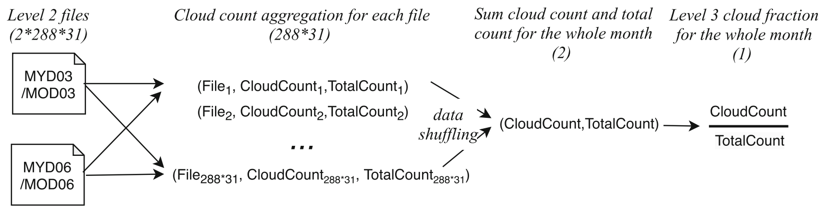
Level 2 files (2*288*31)
Cloud count aggregation for each file (288*31)
Sum cloud count and total count for the whole month (2)
Level 3 cloud fraction for the whole month (1)
MYD03 /MOD03
MYD06 /MOD06
(File1, CloudCount1,TotalCount1)
(File2, CloudCount2,TotalCount2)
...
(File288*31, CloudCount288*31, TotalCount288*31)
data shuffling
(CloudCount,TotalCount)
CloudCount / TotalCount

Fig. 2. Illustration of file level parallel aggregation.

We employed MapReduce model in our Spark based file level scalable MODIS aggregation. Each Map task calls the function shown in Algorithm 1. The function takes one MOD03/MYD03 file and one MOD06/MYD06 file as inputs and generates a 2D (180 × 360) array which which element contains the aggregated cloud pixel count and total pixel count for each grid. In the algorithm, output arrays are initialized in line 1, and data are read into three 2D (2030 × 1354) arrays in lines 2–4. Then line 5 applies the same sampling for all three variables. Line 6 operates on the two longitude/latitude arrays by truncating the floating part of the values. Then in the first for loop (lines 7–9), we can find the integral longitude-latitude grid indices for each pixel and increment the total pixel count by 1 for the grid. Then we check which pixel is cloudy in line 10 and use the second for loop (lines 11–14) to update cloudy pixel count based its grid location.

In this approach, the total (MOD03/MYD03, MOD06/MYD06) file pair number is 8928 (288 × 31). By first creating a list of 8928 file pairs and setting partition number to be 8928, Spark will generate 8928 Map tasks and one task for each file pair. These 8928 Map tasks run in parallel on distributed nodes. After receiving outputs from the Map phase, the two 2D (180 × 360) arrays of cloudy pixel count and total pixel count are simply added to two final 2D (180 × 360) arrays via a Reduce sum function. The cloud fraction ratio is calculated via dividing cloudy pixel counts by total pixel counts for each grid.

Algorithm 1: Data aggregation for each file: *aggregateOneFileData()*

```
Input: MYD06/MOD06 file path: M06_file; MYD03/MOD03 file path: M03_file
Output: 2D (180x360) array for cloud pixel count and total pixel count of each grid:
        cloud_pixel_count, total_pixel_count
 1: Initialize cloud_pixel_count and total_pixel_count to all zero 2D (180x360) array
 2: Read Cloud_Mask_1km variable from M06_file file and extract its cloud phase values
    to a 2D (2030x1354) array: 2D_pixel_array_cloud_mask
 3: Read Latitude variable values from M03_file file to a 2D (2030x1354) array:
    2D_pixel_array_lat
 4: Read Longitude variable values from M03_file file to a 2D (2030x1354) array:
    2D_pixel_array_lon
 5: Apply the same sampling ratio for all above three variables
 6: Convert floating-point numbers in 2D_pixel_array_lat and 2D_pixel_array_lon to
    integral numbers
 7: for each grid location in (2D_pixel_array_lat, 2D_pixel_array_lon) do
 8:     Increment total_pixel_count by 1 for the latitude-longitude grid
 9: end for
10: Retrieve element indices in 2D_pixel_array_cloud_mask if the element's value shows
    the pixel is cloudy: cloud_indices.
11: for each index in cloud_indices do
12:     Get integral latitude and longitude values for the index
13:     Increment cloud_pixel_count by 1 for the latitude-longitude grid
14: end for
15: Output (cloud_pixel_count, total_pixel_count)
```

In our Dask and MPI implementations, we used similar Map function so that all 8928 tasks from all file pair combinations can be executed in parallel. After the tasks are done, results are integrated via a for loop.

There is a major difference among the above implementations using Spark, Dask and MPI. Both Spark and Dask support dynamic scheduling of tasks by distributing tasks in the queue to available worker processes. It is particularly useful for our aggregation application because the tasks' execution times can vary from 1 s to 15 s. Dynamic scheduling can achieve good load balance among worker processes. For MPI, we had to programmatically assign tasks to processes without the knowledge of loads for each process.

4.2 Day Level Scalable MODIS Aggregation

The overall logic of day level scalable MODIS aggregation approach is shown in Fig. 3. In day level scalable MODIS aggregation, each function is similar with the one in Algorithm 1 except each function processes one day data (288 files) via an additional for loop. So there will be 31 tasks to be processed in total. Our tests show the execution times for the tasks are also very different, varying from 380 s to 685 s. The implementations in Spark, Dask and MPI are similar with their file level implementations.

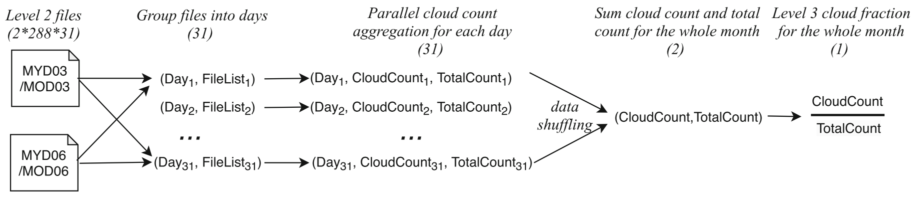
Level 2 files (2*288*31)
Group files into days (31)
Parallel cloud count aggregation for each day (31)
Sum cloud count and total count for the whole month (2)
Level 3 cloud fraction for the whole month (1)
MYD03 /MOD03
MYD06 /MOD06
(Day1, FileList1)
(Day2, FileList2)
...
(Day31, FileList31)
(Day1, CloudCount1, TotalCount1)
(Day2, CloudCount2, TotalCount2)
...
(Day31, CloudCount31, TotalCount31)
data shuffling
(CloudCount,TotalCount)
CloudCount / TotalCount

Fig. 3. Illustration of day level parallel aggregation.

4.3 Pixel Level Scalable MODIS Aggregation

The overall logic of pixel level scalable MODIS aggregation approach is shown in Fig. 4. In this approach, each function still processes one file pair like Algorithm 1. The difference lies in how to generate outputs. In stead of generating two 2D (180×360) arrays like the first two approaches, it outputs a list of all pixels in the input file. By looping through all pixels in the input, each pixel outputs a tuple of key-value pair. The key element is a tuple of grid information: (latitude, longitude) and the value element is a tuple of cloudy pixel count. If a pixel is cloudy, the element is (1, 1), otherwise it is (0, 1). Because each input file contains a 2D (2030×1354) array, the output list has 2,748,620 ($=2030 \times 1354$) elements.

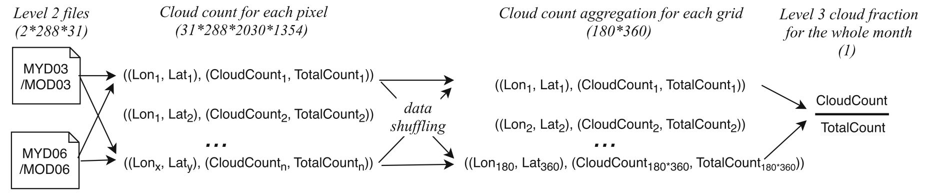
Level 2 files (2*288*31)
Cloud count for each pixel (31*288*2030*1354)
Cloud count aggregation for each grid (180*360)
Level 3 cloud fraction for the whole month (1)
MYD03 /MOD03
MYD06 /MOD06
((Lon1, Lat1), (CloudCount1, TotalCount1))
((Lon1, Lat2), (CloudCount2, TotalCount2))
((Lonx, Laty), (CloudCountn, TotalCountn))
data shuffling
((Lon1, Lat1), (CloudCount1, TotalCount1))
((Lon2, Lat2), (CloudCount2, TotalCount2))
((Lon180, Lat360), (CloudCount180*360, TotalCount180*360))
CloudCount
TotalCount

Fig. 4. Illustration of pixel level parallel aggregation.

In our Spark based implementation, we used *flatMap* to wrap the above function so that the elements of the output listed are flattened and shuffled based on their keys, and *reduceByKey* to group the results based on their grid information. We do not have implementations in Dask and MPI because they do not have similar higher-level functions like flatMap and reduceByKey.

5 Experiments

In this section, we explain the experiments we conducted to benchmark and evaluate the differences of using different parallel approaches, parallel platforms and sampling ratios. We choose January 2008 MODIS Level 2 cloud properties data products from NASA Aqua satellite for our aggregation inputs, which are 8928 MYD03 and MYD06 files. The files are in HDF format and the total data size is 738 GB. The files are located on a centralized data node and accessed via network file system (NFS). For software, we used Python (version 3.6.8), Spark (version 2.4), Dask (version 1.1.4), and MPI (version 1.4.1).

All experiments were done in a local High Performance Computing (HPC) cluster. Each computer node has two 18-core Intel Xeon Gold 6140 Skylake CPUs (36 cores in total) and 384 GB of memory. To make fair comparison, we allocate 2 CPU cores and 20 GB memory for each parallel process. By running on 2, 4, 6, 8, 10 and 12 nodes, we can have 36, 72, 108, 144, 180 and 216 parallel processes respectively.

5.1 Comparison Between Different Parallel Approaches

We listed the exact execution times in Table 1 and the speedup in Fig. 5. The bold numbers in the table are the shortest execution times for each distributed environment size. The speedups are calculated via dividing the execution times of serial versions of each approach by the execution times in Table 1. We note our serial version of file level aggregation approach is also used in calculating speedup of pixel level aggregation because it is difficult to implement serial version of key-based data shuffling. The table and figure show the performances of file level parallel aggregation are the best in most cases. We believe a big reason is the task granularity. By having 8928 tasks and each task processing one pair of files, file level parallel aggregation approach can keep all processes running in parallel with assigned tasks for distributed environments in our experiments. Day level parallel aggregation approach shows the worst performance. We believe one reason is many processes will be idling once the available process number is greater than day number (31). Another reason could be the difficulty to achieve load balance among parallel processes because its task granularity is much higher than file level approach. The performance of our spark pixel level parallel aggregation approach is close to spark file level approach. The biggest difference between these two approaches is the data record number to be shuffled after Map/flatMap phase: 8928 records in spark file level approach and 800 million records in pixel file level approach. We believe the reason that the huge data record number in the pixel level approach did not slow down the execution is Spark supports local partial aggregation at Map phase before data is shuffled across network if the following phase is reduceByKey function, which is similar to the Combiner in Hadoop.

Table 1. Execution time results (in Seconds) for scalability evaluation.

Node number	Spark			Dask		MPI	
	File level	Day level	Pixel level	File level	Day level	File level	Day level
2	**256.30**	803.86	279.81	769.82	890.84	788.10	306.51
4	**225.07**	603.66	269.81	423.76	635.48	479.56	428.45
6	**214.07**	729.08	245.80	422.94	595.03	326.90	434.04
8	197.07	464.00	**152.82**	335.16	593.53	205.74	329.25
10	181.68	690.66	208.77	462.80	632.09	**135.10**	418.66
12	164.30	507.79	216.07	364.17	571.87	**161.03**	303.57

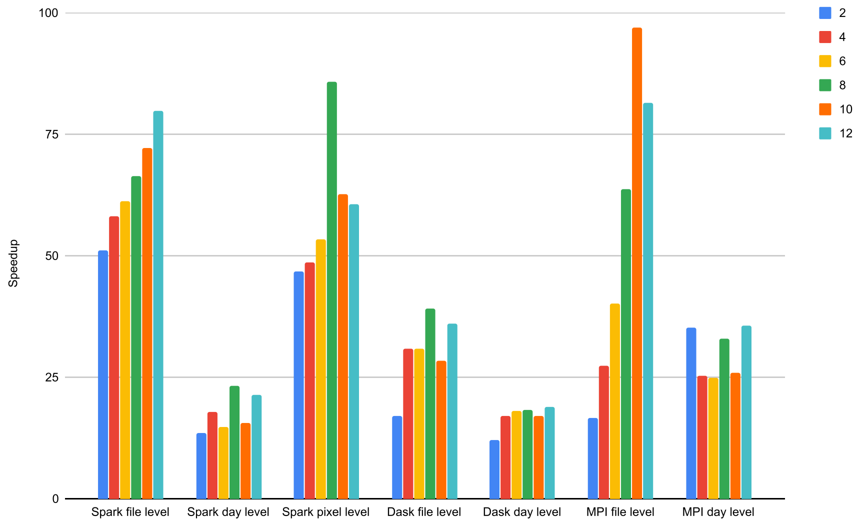

Fig. 5. Execution Speedup for scalability evaluation.

5.2 Comparison Between Different Parallel Platforms

From Table 1 and Fig. 5, we can see Spark achieves the best speedup on average among all three parallel platforms, especially for the file level approach. We believe the reason is Spark can manage task scheduling more efficiently and achieve better load balancing among parallel processes. For the comparison between MPI and Dask, the speedups done via MPI are better than those via Dask in most cases. We think it is due to less coordination overhead for MPI based parallelization. We also notice the best speed up is achieved by file level parallelization via MPI on its execution 10 nodes. It shows the advantage of MPI because of its low coordination overhead, especially if the static scheduling via the MPI approach happens to achieve relatively balanced work loads among different nodes.

5.3 Comparison Between Different Sampling Ratios

We further evaluated how different sampling ratios will affect the execution times and aggregation result quality. Because we conduct sampling for both longitude and latitude direction, sampling ratio of n means only reading 1 out of n^2 pixels. We tested our file level parallelization with Spark implementation on multiple nodes. Table 2 shows the execution times of different sampling ratios on different nodes where the baseline row is done without parallelization and Spark, and the no sampling column is for experiments without sampling. The bold numbers in the table are the shortest execution times for each environment size. From the table, we can see the execution times increase from no sampling to sampling, then decrease with higher sampling ratios. We believe the reason is the additional time for sampling operation could be longer than the time saved for downstream operations with less data, especially when sampling ratio is low. Further, the table shows the executions achieve good scalability for all sampling ratios. For sampling ratio as 5, all parallel executions take less time than corresponding no sampling executions and the largest speed up ratio is 2.586 when running on 4 nodes.

We also calculated data loss percentage for different sampling ratios based on Formula 1. It first calculates the percentage of absolute cloud fraction value difference for each grid, then computes the average for all grids. By using absolute cloud fraction value difference, not actual value difference, in the formula, we can avoid offsetting between positive value differences and negative differences. Our experiments show the data loss percentages are 0.1801%, 0.3021%, 0.4535% and 0.6167% when sampling ratios are 2, 3, 4 and 5, respectively. It shows higher sampling ratio causes a little more data loss percentage, but all data loss percentages are quite small (below 1%). Further, our experiments show the data losses are the same when running for different compute nodes, which verifies the correctness of our approach does not change with compute node numbers.

$$\frac{1}{180 * 360} \sum_{(-90,-180)}^{(90,180)} \left| \frac{CF(i,j)_{orig} - CF(i,j)_{samp}}{CF(i,j)_{orig}} \right| \tag{1}$$

Table 2. Execution time results (in Seconds) for different sampling ratios.

	No sampling	Sampling ratio			
		2	3	4	5
Baseline	**42,817.87**	133,830.33	267,660.87	86,225.44	55,484.36
2 nodes	1,254.83	5,395.07	2,509.40	1,510.31	**1,008.71**
4 nodes	1,460.57	3,312.09	1,272.82	1,045.13	**564.79**
6 nodes	598.71	2019.24	994.65	590.83	**424.14**
8 nodes	791.23	506.44	461.81	465.77	**415.51**
10 nodes	371.49	269.63	450.24	447.98	**320.76**
12 nodes	259.57	316.91	373.20	474.62	**254.23**

6 Big Data Service for MODIS Aggregation

To simplify on-demand MODIS data aggregation by users, we further provision the above scalable data aggregation capability as services. In this way, users do not need to have a distributed environment with proper back-end libraries installed and download large-scale MODIS data for aggregation. One challenge we face is how to make the services work with different service libraries/protocols, such as REST [17] and ZeroMQ [7]. To achieve this flexibility for our services, we employ an open-source framework called Stratus (Synchronization Technology Relating Analytic Transparently Unified Services) [6] developed by co-author Maxwell. We will explain the Stratus framework, and how our data aggregation services are implemented via Stratus.

The Stratus framework provides a workflow orchestration approach for incorporating Earth data analytic services as a unified solution. It defines a common set of API for workflow and request/response. It consists of a set of orchestration nodes and each implements a particular composition strategy on a particular technology and is designed to interface with other Stratus nodes. Then, an integration framework can be constructed by combining orchestration nodes. Currently available Stratus service handlers include endpoint, ZeroMQ [7], OpenAPI [1] and REST [17]. Stratus can support them by having a separate implementation of the same unified API for each specific library/protocol.

In order to expose a capability within the Stratus framework, that capability must be wrapped as a Stratus endpoint. A typical Stratus based analytics service architecture includes: 1) client application used by users to connect to remote server, 2) server application which accepts connections from the client, 3) endpoint which is a server-side operation for a certain task. In this architecture, a common request language can be established across all the supported endpoints. A common server is used by the client to submit service requests to any endpoint. Coordinating workflows composed of services developed by disparate teams requires the combination of multiple orchestration strategies, e.g., fan-out, publish-subscribe, distributed task queue, and request-reply.

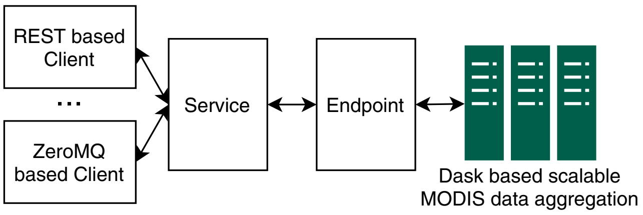

Fig. 6. Flexible MODIS data aggregation service.

Our implementation of MODIS aggregation servicelization using Stratus is illustrated in Fig. 6 and its endpoint and client side code are illustrated in Listings 1 and 2 respectively. For endpoint code, we only need to implement function *operation* to define an operation, and function *execute* to call the operation based on client-side inputs. For client-side code, request specifications are defined through a Python dictionary object. To switch from one service library/protocol to another, we only need to change the value for *type* parameter.

```
class XaOpsExecutable(Executable):
  #Definition of the operation
  def operate(self, M03_dir, M06_dir):
    if self.request['operation'][0]['name']=="
   cloudFraction":
      cf = modis.calculateCloudFraction(M03_dir, M06_dir)
    return cf
  #Executes the operation.
  def execute(self, **kwargs) -> TaskResult:
    inputSpec = self.request.get('input', [])
    cf = self.operate(inputSpec['path1'], inputSpec['path2
   '])
    result = xarray.DataArray(cf.tolist(), name='test')
    return TaskResult(kwargs, [result])
```

Listing 1. Endpoint for MODIS cloud fraction aggregation.

```
if __name__ == "__main__":
  settings = dict(stratus=dict(type="rest"))
  stratus = StratusCore(settings)
  client = stratus.getClient()
  requestSpec = dict(input=dict(path1="MYD03", path2="
   MYD06", operation=[dict(name="cloudFraction")])
  # Submit the request to the server and wait for the
   result
  task: TaskHandle = client.request(requestSpec)
  result: Optional[TaskResult] = task.getResult(block=True
   )
```

Listing 2. Client for MODIS cloud fraction aggregation.

7 Related Work

Benchmarking for Big Data Analytics. There have been many studies on benchmarking for big data analytics including [13,21]. Study [14] compared Spark and Dask in a deep learning satellite image application and their experiments show Spark achieves better performance. The experiments in [11] show no significant execution performance differences between Spark and Dask for their neuroimaging pipeline applications. The work at [8] compares OpenMP, OpenMP + MPI, and Spark for K-means clustering on distributed environment. Our study complements these studies by comparing not only different platforms but also different scalable data aggregation approaches, and analyzing the reasons for the differences.

Climate Analytics as Services. With the rapid increase of climate data, researchers have been studying how to apply service oriented architecture in climate analytics to achieve better efficiency and flexibility than the traditional way of downloading data to a local machine and then analyzing it. Book [20] collected recent studies on how to apply cloud computing for Ocean and Atmospheric Sciences. Among them, the Climate Analytics-as-a-Service (CAaaS) framework [19] is most similar to our work by addressing both service challenge and big data challenge. It differentiates two types of services for big climate data: 1) Analytic Services that run MapReduce style analytic jobs/workflows and 2) Persistent Services that manage the storage and access of data produced by Analytic Services. Our work in this paper falls in the Analytic Service category by supporting on-demand aggregation of satellite data. Our work further addresses the service library/protocol variety challenge by employing the Stratus framework so the same service side implementation can support different service libraries/protocols with simple configuration change at client side.

8 Conclusions

With astronomical growth of available climate data, we believe big data techniques and service techniques are promising to achieve scalable and on-demand analytics for climate data. In this paper, we study how to integrate these techniques for a fundamental climate data aggregation application. We proposed three different aggregation approaches and compared their performance in three different platforms. From the experiments, we conclude that, while we can achieve speedup using all approaches and platforms, defining proper task granularity and dynamic scheduling are key factors to enable good scalability and load balance. The best speed up ratio we can achieve is close to 100. Our experiments also show proper sampling ratio design could achieve execution speedup with little data loss. Last, we discussed how to leverage the Stratus framework to easily support different service libraries/protocols.

For future work, we will extend our satellite data aggregation capability for more climate variables (such as cloud top height), more statistics (such as standard deviation and histogram), more flexible spatial area selection. We also plan to expose the service on public cloud environments, such as Amazon Web Service, for users to use without requiring a distributed environment.

Acknowledgment. This work is supported by an NASA CMAC grant (80NSSC18K 0796) and an NSF CyberTraining grant (OAC-1730250). The execution environment is provided through the High Performance Computing Facility at UMBC.

References

1. OpenAPI Initiative (OAI). https://www.openapis.org. Accessed 28 May 2020
2. Scalable MODIS Data Aggregation Platform. https://github.com/big-data-lab-umbc/MODIS_Aggregation. Accessed 28 May 2020
3. Apache Spark Project (2020). http://spark.apache.org. Accessed 28 May 2020
4. Dask: Scalable analytics in Python (2020). https://dask.org/. Accessed 28 May 2020
5. NASA LAADS Distributed Active Archive Center (2020). https://ladsweb.modaps.eosdis.nasa.gov/search/. Accessed 28 May 2020
6. Stratus: Synchronization Technology Relating Analytic Transparently Unified Services (2020). https://github.com/nasa-nccs-cds/stratus/. Accessed 28 May 2020
7. ZeroMQ: An open-source universal messaging library (2020). https://zeromq.org/. Accessed 28 May 2020
8. Barajas, C., et al.: Benchmarking parallel k-means cloud type clustering from satellite data. In: Zheng, C., Zhan, J. (eds.) Bench 2018. LNCS, vol. 11459, pp. 248–260. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32813-9_20
9. Chambers, B., Zaharia, M.: Spark: The Definitive Guide: Big Data Processing Made Simple. O'Reilly Media Inc., Sebastopol (2018)
10. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
11. Dugré, M., Hayot-Sasson, V., Glatard, T.: A performance comparison of dask and apache spark for data-intensive neuroimaging pipelines. In: 2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS), pp. 40–49. IEEE (2019)
12. Eyring, V., et al.: Overview of the coupled model intercomparison project phase 6 (cmip6) experimental design and organization. Geoscientific Model Development (Online), 9(LLNL-JRNL-736881) (2016)
13. Han, R., Lu, X., Xu, J.: On big data benchmarking. In: Zhan, J., Han, R., Weng, C. (eds.) BPOE 2014. LNCS, vol. 8807, pp. 3–18. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13021-7_1
14. Nguyen, M.H., Li, J., Crawl, D., Block, J., Altintas, I.: Scaling deep learning-based analysis of high-resolution satellite imagery with distributed processing. In: 2019 IEEE International Conference on Big Data (Big Data), pp. 5437–5443. IEEE (2019)
15. Overpeck, J.T., Meehl, G.A., Bony, S., Easterling, D.R.: Climate data challenges in the 21st century. Science **331**(6018), 700–702 (2011)
16. Pacheco, P.: Parallel Programming with MPI. Morgan Kaufmann, San Francisco (1997)
17. Pautasso, C., Wilde, E., Alarcon, R.: REST: Advanced Research Topics and Practical Applications. Springer, New York (2013). https://doi.org/10.1007/978-1-4614-9299-3

18. Rocklin, M.: Dask: parallel computation with blocked algorithms and task scheduling. In: Proceedings of the 14th Python in Science Conference, no. 130–136. Citeseer (2015)
19. Schnase, J.L.: Climate analytics as a service. In: Cloud Computing in Ocean and Atmospheric Sciences, pp. 187–219. Elsevier (2016)
20. Vance, T.C. , Merati, N., Yang, C., Yuan, M.: Cloud computing for ocean and atmospheric science. IEEE (2016)
21. Wang, L., et al. Bigdatabench: a big data benchmark suite from internet services. In: 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), pp. 488–499. IEEE (2014)