# Evaluating the suitability of MapReduce for surface temperature analysis codes

Vinay Sudhakaran
EPCC, University of Edinburgh
JCMB, Mayfield Road
Edinburgh, EH16 5AH, U.K.

vinaysudhakaran@gmail.com

Neil Chue Hong
EPCC, University of Edinburgh
JCMB, Mayfield Road
Edinburgh, EH16 5AH, U.K.
+44 131 650 5957

N.ChueHong@epcc.ed.ac.uk

## ABSTRACT
Processing large volumes of scientific data requires an efficient and scalable parallel computing framework to obtain meaningful information quickly. In this paper, we evaluate a scientific application from the environmental sciences for its suitability to use the MapReduce framework. We consider cccgistemp – a Python reimplementation of the original NASA GISS model for estimating global temperature change – which takes land and ocean temperature records from different sites, removes duplicate records, and adjusts for urbanisation effects before calculating the 12 month running mean global temperature. The application consists of several stages, each displaying differing characteristics, and three stages have been ported to use Hadoop with the mrjob library. We note performance bottlenecks encountered while porting and suggest possible solutions, including modification of data access patterns to overcome uneven distribution of input data.

## Keywords
Data-intensive, MapReduce, Hadoop, environmental sciences.

## 1. INTRODUCTION
Advancing the state-of-the-art research in computational science requires analysis of large volumes of data collected from numerous scientific instruments and experiments conducted around the globe. Petabyte data sets are already becoming increasingly common in many High End Computing (HEC) applications from a diverse range of scientific disciplines [15], and this is only expected to grow in the near future. This necessitates the need for providing abstraction at multiple levels for acquiring, managing and processing of data [11], thus enabling the scientific community to focus on science rather than disentangling the complexities involved in setting up and maintaining the cyber-infrastructure required to facilitate data intensive computing.

The main focus of the work described in this paper is to evaluate if MapReduce [4], specifically the Hadoop [1] implementation of MapReduce, can provide the necessary high level programming abstraction that is required for parallelising data and compute intensive tasks of a scientific application.

## 2. GLOBAL TEMPERATURE ANALYSIS
Analyses of surface air temperature and ocean surface temperature changes are carried out by several groups, including the Goddard institute of space studies (GISS) [8] and the National Climatic Data Center [19] based on the data available from a large number of land based weather stations and ship data, which forms an instrumental source of measurement of global climate change. Uncertainties in the collected data from both land and ocean, with respect to their quality and uniformity, force analysis of both the land based station data and the combined data to estimate the global temperature change.

Estimating long term global temperature change has significant advantages over restricting the temperature analysis to regions with dense station coverage, providing a much better ability to identify phenomenon that influence the global climate change, such as increasing atmospheric CO2 [10]. This has been the primary goal of GISS analysis, and an area with potential to made more efficient through use of MapReduce to improve throughput.

### 2.1 Data Sources
The current GISS analysis obtains the monthly mean station temperatures from the Global Historical Climatology Network (GHCN), available for download from the NCDC website[1]. GHCN maintains data from about 7000 stations out of which only those stations that have a period of overlap with neighbouring stations (within 1200 km) of at least 20 years are considered [9]. Effectively, only 6300 stations are available for GISS analysis after this reduction. No data adjustments are done on the original GHCN data for clarity. The GHCN land-based temperature records are supplemented by monthly data for the Antarctic region from the SCAR Reference Antarctic Data for Environmental Research project[2].

The ocean surface temperature measurement is an integration of the data from Met Office Hadley Centre analysis of sea surface temperatures (HadISST1) for the period 1880-1981, which was ship based during that interval, and satellite measurements of sea surface temperature for 1982 to the present (Optimum Interpolation Sea Surface Temperature version 2 (OISST.v2). The satellite measurements are calibrated with the help of ship and buoy data [20].

### 2.2 Urbanisation Correction
Urbanisation, which includes human-made structures and energy sources, can significantly impact the accuracy of temperature measured by stations located in or near urban areas. This has been a major concern in the analysis of global temperature change. The homogeneity adjustment procedure [8] changes the long-term temperature trend of an urban station to make it agree with the

[1] http://www.ncdc.noaa.gov/oa/ncdc.html

[2] http://www.antarctica.ac.uk/met/READER/

mean trend of nearby rural stations. The current analysis uses satellite observed nightlights [9] to identify land based weather stations in extreme darkness and perform urban adjustments for non-climatic factors, such that urban effects on the analysed global temperature change are small.

## 2.3 GISTEMP and ccc-gistemp

The *GISS Surface Temperature Analysis (GISTEMP)*[3] is an open-source model from the environmental sciences for estimating the global temperature change, implemented by the NASA Goddard Institute of Space Studies (GISS). *GISTEMP* is written in FORTRAN and processes the current GHCN, USHCN and SCAR files in two stages. In the first stage, redundant multiple records are combined, and in the second stage the urban adjustments are performed so that their long-term trend matches that of the mean of neighbouring rural stations. Urban stations without a sufficient number of rural stations in their vicinity are removed from further analysis.

*ccc-gistemp* is a part of the Clear Climate Code (CCC) project[4] from Climate Code Foundation[5] to re-implement the original *GISTEMP* algorithm in Python for clarity and maintainability. The combination of FORTRAN code and shell scripts in the original GISS software, for both core GISS algorithms and supporting libraries, makes it hard to perceive the code flow and the underlying algorithm. By isolating the core GISS algorithms from the supporting functions (algorithms for reading input and writing output) and providing a single interface specifying parameters and initiating the analysis, ccc-gistemp seeks to improve the readability and maintainability of the software.

In particular, the temperature “Series” anomaly, which is the primary data object in most GISS algorithms, has differing representations in different parts of the code making it hard for new users to comprehend the code. This has been modified in ccc-gistemp to have a single representation throughout the code.

Although there have been a significant modifications to the structure of the original code, the ccc-gistemp developers have ensured that the ported algorithms are identical in function to the original by providing tests which compare the output at every stage with the expected results. For this reason, we have chosen to use the ccc-gistemp code rather than the original GISTEMP codes for this study.

## 3. PORTING ccc-gistemp

## 3.1 MapReduce

MapReduce is a programming model introduced by Google Inc. to support distributed computing on large datasets [4]. The application is implemented as a sequence of MapReduce operations, each consisting of a Map phase and a Reduce phase. In its basic form, the user specifies a ‘map’ function that processes a key/value pair to generate a set of intermediate key/value pairs, and a ‘reduce’ function that merges all intermediate values associated with the same intermediate key. It provides a master-worker mechanism to improve fault-tolerance that enables tasks on failed nodes to be rescheduled, and attempts dynamic load balancing and task reassignment based on the performance of the nodes.

Data locality – collocation of data and the node that performs computation – is a characteristic feature of MapReduce that facilitates data-intensive computing. The MapReduce master acquires information of the location of the input file from the distributed file-system and attempts to assign processing on the machine that actually contains the data. If this results in a failure, then the master reassigns the processing on a machine that is as close as possible to the input data. This has the effect of moving code to the data, improving the overall network utilisation by avoiding unnecessary data transfers. Experiments [3][22] indicate that data locality is a determining factor for MapReduce performance particularly in heterogeneous environments, a factor which is further influenced by the irregular data access patterns often found in scientific applications.

## 3.2 Hadoop and mrjob

Hadoop [1] is the Apache Software Foundation open-source implementation of the MapReduce framework in Java. It provides tools for processing data using the MapReduce framework and implements a distributed file system called Hadoop Distributed File System (HDFS). Although the Hadoop framework is implemented in Java, it is not required that MapReduce functions be written in Java. Hadoop streaming is a utility that allows programmers to create and run MapReduce jobs with executables (map and/or reduce function) written in any programming language that can read standard input and write to standard output. It uses UNIX standard streams as an interface between Hadoop framework in Java and the user program.

The fundamental idea of having a distributed file system is to divide user data (usually of the order of few gigabytes to a few terabytes) into blocks and replicate those blocks across the local disks of nodes in the cluster [14] such that it is easier to assign MapReduce job locally. HDFS is designed based on this principle. Additionally, data-intensive computing using MapReduce is dominated by long streaming reads and large sequential writes. As a result, the time to read the whole dataset is more important than the latency in reading the first record [21].

*mrjob*[6] is a Python package that aids in the development and execution of Hadoop streaming jobs. Features that are useful when porting scientific applications include the ability to write multi-step jobs (one map-reduce step feeds into the next), custom switches which can be added to the map-reduce jobs, including file options, and the ability to quickly switch between running locally, on a Hadoop cluster or on Amazon’s Elastic MapReduce. mrjob provides a simple abstraction for writing MapReduce jobs in Python by defining steps for specifying ‘mapper’ and ‘reducer’ functions, input and output file format (protocol) and paths. Additionally, it provides APIs for setting necessary parameters in the Hadoop MapReduce JobConf configuration file.

To facilitate transfer of complex data structures, Python provides a powerful interface called the Pickle module[7]. It is primarily used for serialising and de-serialising python object structures. Pickling results in data objects being converted into byte stream so that they can be transferred easily through a data pipe such as a network. Un-pickling results in the reverse operation where a byte stream is converted back into a data object. mrjob implements the more efficient cPickle module (written in C) to provide the communication protocol. The key/value pairs are represented as two string-escaped pickles separated by a tab.

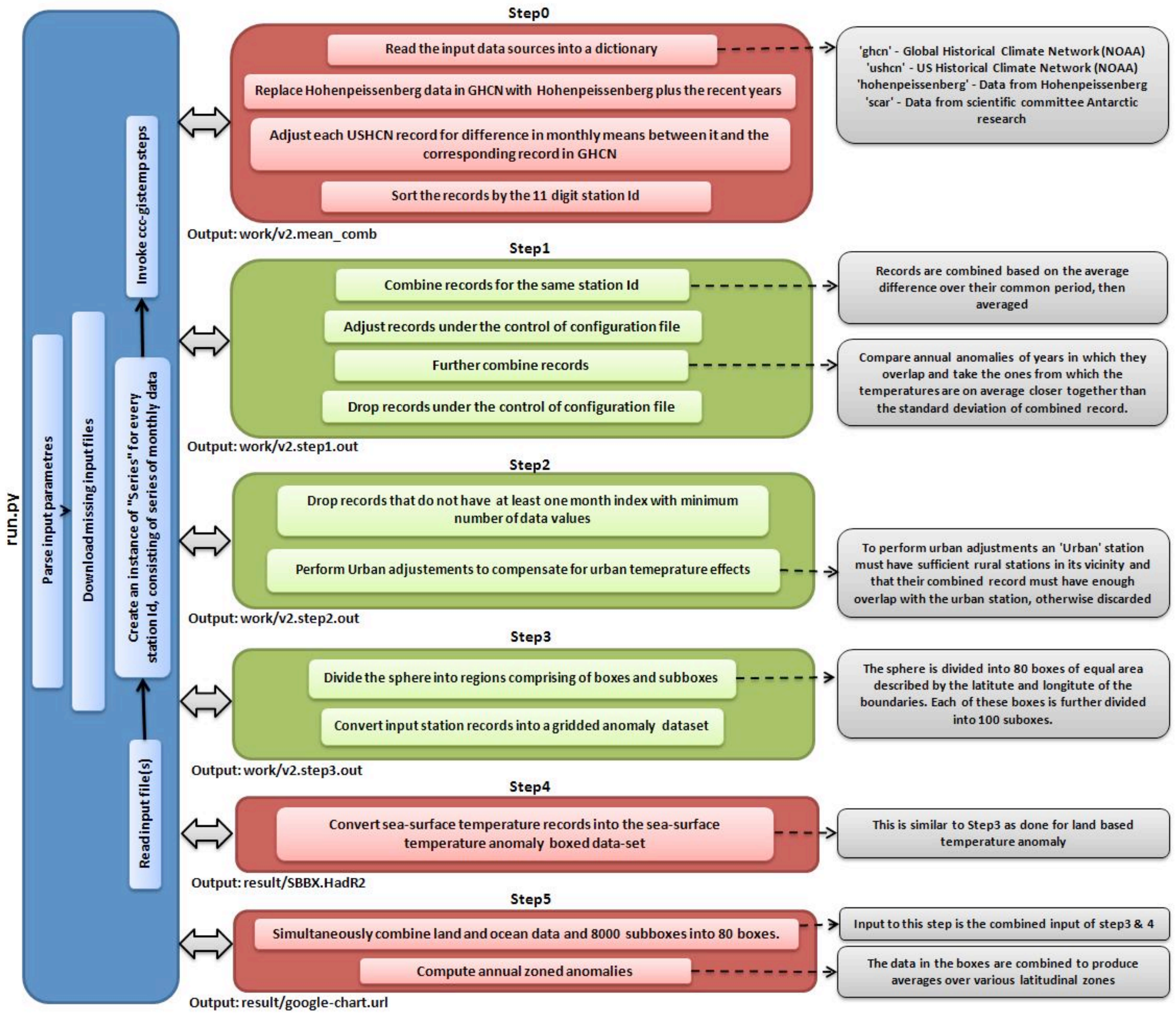
[3] http://www.giss.nasa.gov/

[4] http://code.google.com/p/ccc-gistemp/

[5] http://www.climatecode.org/

[6] http://pypi.python.org/pypi/mrjob/0.2.6

[7] http://docs.python.org/library/pickle.html



run.py
Parse input parametres
Download missing input files
Create an instance of "Series" for every station Id, consisting of series of monthly data
Read input file(s)
Invoke ccc-gistemp steps
Step0
Read the input data sources into a dictionary
Replace Hohenpeissenberg data in GHCN with Hohenpeissenberg plus the recent years
Adjust each USHCN record for difference in monthly means between it and the corresponding record in GHCN
Sort the records by the 11 digit station Id
'ghcn' - Global Historical Climate Network (NOAA) 'ushcn' - US Historical Climate Network (NOAA) 'hohenpeissenberg' - Data from Hohenpeissenberg 'scar' - Data from scientific committee Antarctic research
Output: work/v2.mean_comb
Step1
Combine records for the same station Id
Adjust records under the control of configuration file
Further combine records
Drop records under the control of configuration file
Records are combined based on the average difference over their common period, then averaged
Compare annual anomalies of years in which they overlap and take the ones from which the temperatures are on average closer together than the standard deviation of combined record.
Output: work/v2.step1.out
Step2
Drop records that do not have at least one month index with minimum number of data values
Perform Urban adjustements to compensate for urban temeprature effects
To perform urban adjustments an 'Urban' station must have sufficient rural stations in its vicinity and that their combined record must have enough overlap with the urban station, otherwise discarded
Output: work/v2.step2.out
Step3
Divide the sphere into regions comprising of boxes and subboxes
Convert input station records into a gridded anomaly dataset
The sphere is divided into 80 boxes of equal area described by the latitute and longitute of the boundaries. Each of these boxes is further divided into 100 suboxes.
Output: work/v2.step3.out
Step4
Convert sea-surface temperature records into the sea-surface temperature anomaly boxed data-set
This is similar to Step3 as done for land based temperature anomaly
Output: result/SBBX.HadR2
Step5
Simultaneously combine land and ocean data and 8000 subboxes into 80 boxes.
Compute annual zoned anomalies
Input to this step is the combined input of step3 & 4
The data in the boxes are combined to produce averages over various latitudinal zones
Output: result/google-chart.url

**Figure 1: Flow diagram of the original ccc-gistemp code before modification**

## 3.3 Analysing ccc-gistemp

The ccc-gistemp code can be considered to have one pre-processing stage (Step 0) followed by five processing stages (Steps 1-5), all coordinated by *run.py* (see Figure 1). The code was analysed to understand its execution pattern, whilst also identify areas in the code that could be suitable for porting to MapReduce. Profiling the code (see Figure 2) shows that the runtime is dominated by Step 3, where the input station records are converted into gridded anomaly data-sets represented as a box obtained by dividing the global surface (sphere) into 80 boxes of equal area. An improvement in performance could be obtained by parallelising Step 1 as well.

Each of these steps takes a data object as its input and produces a data object as its output. Ordinarily the data objects are iterators, either produced from the previous step or an iterator that feeds from an input file.  An instance of “Series”, the monthly temperature series for every year starting from the base year (set to 1880 by default, but can be changed to any value), uniquely identified by a 12-digit id is created for every stations data. Multiple series can exist for a single station and hence a 12-digit id is chosen to uniquely identify the records, comprising of 11-digit station id and 1 digit series identifier. This unique id is used as the ‘key’ in Step 1 and Step 2 *map/reduce* functions. Step 3 however has a different key/value combination. Figure 3 is a diagrammatic representation of an instance of “Series” – the primary data object used in all the aforementioned steps for computation.

### *3.3.1 Step 0*

Step 0 reads the input data sources into a dictionary which primarily consists of station data, land and sea surface temperatures from GHCN and USHCN, Antarctic temperature readings from SCAR and the Hohenpeissenberg data. In the first part of this step, the Hohenpeissenberg data in the GHCN is replaced with the correct values from the actual Hohenpeissenberg data. In the second part, the USHCN records are adjusted for the difference in monthly means between them and the corresponding record in GHCN. Once adjusted, the corresponding record in GHCN is removed. Finally, the adjusted records in USHCN, remaining records in GHCN and the original SCAR records are joined together and sorted to generate the final output of Step 0.

Whilst Step 0 appears to be be ideal for parallelisation, it may not fit well into the MapReduce programming model. A global synchronisation across all reducer nodes will be required to combine copies of either GHCN or USHCN records (depending on the logic). Global synchronisation with a compare-merge operation could be very expensive and does not fit into the MapReduce programming model. The MapReduce programming model is designed to perform operations on input data stream mapped as key/value pairs. Having to simultaneously operate on two independent input sources does not fit well into this model. A workaround for this problem would be to load the contents of GHCN file into an external key/value store and have each reduce task concurrently access GHCN records from the store. Records that are adjusted can be removed from the store. This approach is examined in more detail in Section 3.3.3.

In addition, joining of independent data sources (USHCN, GHCN and SCAR) cannot be performed within the mrjob framework. mrjob API does not offer support for operations outside the MapReduce programming paradigm. Workarounds and code hacks significantly deteriorate performance and hence does not form part of a good design. Bearing these design nuances in mind, step0 was not ported to MapReduce framework.

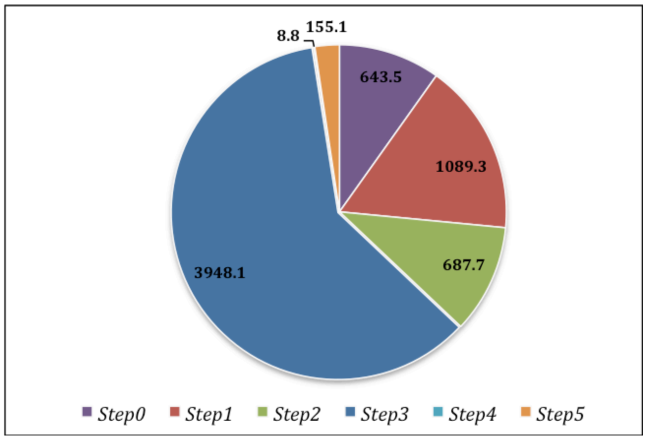


**Figure 2: Profiling original ccc-gistemp code (time to complete each step in pipeline in seconds)**

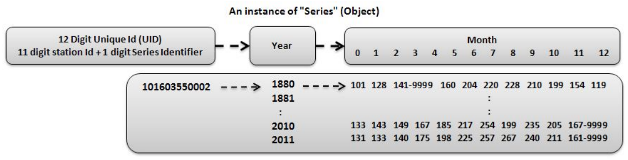


**Figure 3: An instance of "Series" - The primary data structure containing Station id, Year and the monthly temperature series for all years**

### *3.3.2 Step 1*

In this step records from the same station (11 digit station id) are combined in a two stage process. In the first stage records are combined by offsetting based on the average difference over their common period, then averaged. In the second stage, the records are further combined by comparing the annual temperature anomalies of years in which they do overlap, and finding the ones for which the temperatures are on average closer together than the standard deviation of the combined records. Finally, depending on parameters in the configuration files, a few station records are modified by adding a 'delta' to every datum for that station and a few station records are dropped from further analysis.

As the existing algorithms are written to process stations records in groups, these can be ported to MapReduce framework directly without much code changes. The input records are mapped as key/value pairs with 'key' being the 12 digit station id and 'value' being the "Series" temperature anomaly for each year. An intermediate reduce stage is used to construct the "Series" object from the input key/value pairs. This intermediate reduce stage yields the 11-digit station Id and the "Series" object, naturally resulting in data-grouping as required by the combining steps described above. The algorithms for combining records are then ported directly to the second stage reduce function.

### *3.3.3 Step 2*

Step 2 performs a cleaning of input station records by dropping records that do not have at least one month in a year with minimum number of data values prior to performing urban adjustments as mentioned in Section 2.2. The cleanup stage is data-intensive whilst the urban adjustment is mostly compute-intensive.

The data-cleanup step is ideally suited for the MapReduce programming paradigm where the input station records are grouped by their 12 digit station id and processed independently by the available reducers. However, the step following the cleanup operation would require all records processed by the individual reduce tasks to be combined, so as to generate 'rural' and 'urban' classification of records. If there is no global synchronisation at this point then every reduce task will have their own copy of 'urban' and 'rural' classification for the records that were initially assigned to it. This causes issues when performing the urban adjustment.

From Figure 4 it is evident that each urban station will need access to complete rural station records in order to identify rural stations in its vicinity. This dependency between the records contained in each of the reduce tasks is not ideal for the MapReduce framework. Additionally, the use of single reduce task to achieve synchronisation can have severe impacts on performance and scalability. With Step 2 being both data and compute intensive, having a global synchronisation (sequential execution in the MapReduce programming model) must be avoided. One option is to split the set of tasks performed in Step 2 into two separate stages.

The first stage takes the input file and generates key/value pairs, with the 12-digit station id as the key and value being "Series" temperature anomaly for each year. The initial cleanup operation is also performed in this stage. The output of the first stage is a stream of "Series" instances generated from the cleaned up records (urban and rural combined).

The second stage map tasks generate the 'rural' and 'urban' classification of records for the input key/value pairs. These records, generated independently across all map tasks, are stored temporarily in an external key/value store. The use of this external store is necessitated by the fact that the MapReduce model does not provide any natural interface to store shared variables that are required for such an implementation. Additionally, it is useful to use a key/value store as the 'urban' records are directly referred to by their 'key' in the algorithm for adjusting urban stations.

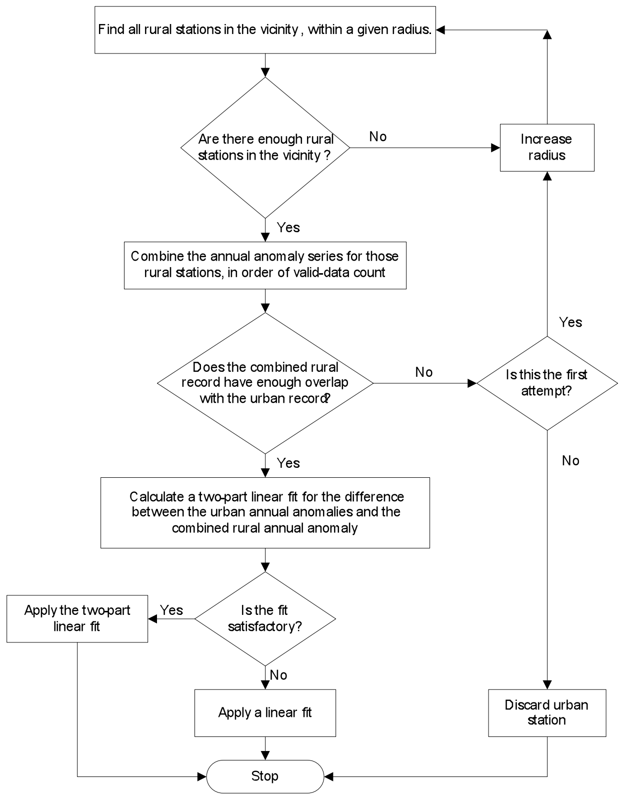

Find all rural stations in the vicinity, within a given radius.
Are there enough rural stations in the vicinity ?
No
Increase radius
Yes
Combine the annual anomaly series for those rural stations, in order of valid-data count
Does the combined rural record have enough overlap with the urban record?
No
Is this the first attempt?
Yes
Yes
No
Calculate a two-part linear fit for the difference between the urban annual anomalies and the combined rural annual anomaly
Is the fit satisfactory?
Yes
Apply the two-part linear fit
No
Apply a linear fit
Discard urban station
Stop

**Figure 4: Flowchart for urbanisation corrections**

Several key/value stores (HBase, PostgreSQL, Voldemort and Redis) were evaluated for use in this project however for reasons of space we omit the full analysis. Redis was chosen primarily because of its Python client and good documentation.

Changes to the data access pattern in the existing ccc-gistemp code were done to accommodate the use of key/value store. The original ccc-gistemp code used the "Series" object of urban stations as the key and its annotated object as the value to represent urban stations internally in a dictionary. This was inappropriate to be used with the key/value store. Pickling and un-picking the "Series" object to be used as 'key' in the external store is very expensive and inefficient in terms of memory consumption. Instead, the code was changed to have the 12-digit station id of urban stations as the 'key' and their annotated object as the 'value'. All the annotated objects of stations classified as 'rural' were appended to a single list on the key/value store.

Key/value stores can thus be used to share data across available reduce tasks without the need for global synchronisation with a single reduce task. However, in situations where the algorithm forces global aggregation [24], global synchronisation is inevitable with the current implementation of MapReduce.

### *3.3.4 Step 3*

In Step 3, the input station records are converted into gridded anomaly data-sets represented as a box obtained by dividing the global surface (sphere) into 80 boxes of equal. These boxes are described by a 4-tuple of its boundaries (fractional degrees of latitude for northern and southern boundaries and longitude for western and eastern boundaries). Each of these 80 boxes is further sub-divided into 100 subboxes described by the same 4-tuple latitude/longitude representation. The input station records are assigned to the box/subbox in which they belong. The station records that belong to a grid cell are called contributors. The number of contributing records varies significantly from one region to the other. A subbox series (similar to station record "Series") consisting of monthly temperature anomaly is created for all records and returned.

An initial approach for porting to MapReduce would be to map the input station records as key/value pairs, as done previously in Steps 1 and 2. However, this type of mapping has severe drawbacks. The original code is written to parse the input station records and assign then to the correct box and then subbox. If the input records are split across available reducers, each of them would process their own subset of the original records and assign them to grids created within each reducer. At the end of this step every reducer will have its own copy of the gridded anomaly data-set. There are two issues with this approach. Firstly, each of the subbox "Series" objects are created with only the partial data available within each reduce function and secondly, there is no way of combining these independent subbox "Series" objects for the same station as the objects are already fully constructed within each reducer. Writing methods to mutate the read-only objects of "Series" would be a serious design flaw.

The more suitable approach would be to split the regions (boxes) across available reducers and have each reduce function independently read the input station records and assign records that belong to its region (box). However, this can be viewed more as a parallelisation strategy for compute-intensive step rather than data-intensive computing using MapReduce. The 'key' is selected from one of the 4-tuples (latitude/longitude representation) and the 'value' is a tuple consisting of the region (box) and the subboxes within that region. Each of available reducers will compute the contributors for the region that was assigned to them, identified by the 4-tuples representation and yield the gridded anomaly dataset.

It should be noted that using this approach, no input was directly specified to the MapReduce job. Instead, the regions were read from within the map function and converted into key/value pairs consisting of one of the 4-tuples (latitude/longitude representation) as the 'key' and a tuple of region (box) and subboxes within that region as 'value'. All regions associated with a 'key' will be processed by the same reducer yielding the gridded anomaly data-sets.

Additionally, it is observed that grouping by longitude will result in many more unique 'keys' than by latitude, which has just 8 unique numbers. As we already know that MapReduce assigns all 'values' associated with the same 'key' to a single reduce task, using latitude as the key will result in a maximum of 8 reduce tasks, severely impacting the scalability of the implementation. Hence we choose the 'key' to be one of either eastern or western longitude. The results of benchmarking with both the combinations of keys are presented in Section 4.

### *3.3.5 Step 4*

Step 4 converts the recent sea-surface temperature records into the sea-surface temperature anomaly boxed dataset. The initial steps are I/O intensive but the overall execution time is extremely small compared to the other stages (see Figure 2). The Hadoop implementation of MapReduce incurs considerable start-up costs that are usually amortised when processing large amounts of data in parallel across available nodes. However, if the dataset is small, these initial start-up costs dominate even when executed on large number of nodes. As this step is neither data-intensive nor compute-intensive, it was not ported to MapReduce.

### *3.3.6 Step 5*

The output files from Step 3 (land data) and Step 4 (ocean data) forms the input to Step 5. Step 5 then assigns weights to the records – a process known as masking – and then combines the land and ocean series in each of the subboxes and combines the subboxes into boxes. The box data for each of the 80 boxes is processed to produce temperature averages over 14 latitudinal zones including northern hemisphere, southern hemisphere and global.

By altering the sequence of operations slightly, and taking care to ensure storage of intermediate results, several stages in Step 5 can be grouped together for the MapReduce framework leaving the I/O stages to be performed at the end. Splitting the input land and ocean records across the available reduce tasks has drawbacks as already mentioned in Step 3 analysis (see 3.3.4) for the MapReduce programming model. Instead, the regions (boxes) could be split across available reducers with each reduce function independently read the land and ocean records and assign records that belong to its region (box). Additionally, with this approach each reducer will have the Subbox metadata that is required for processing all gridded anomaly dataset.

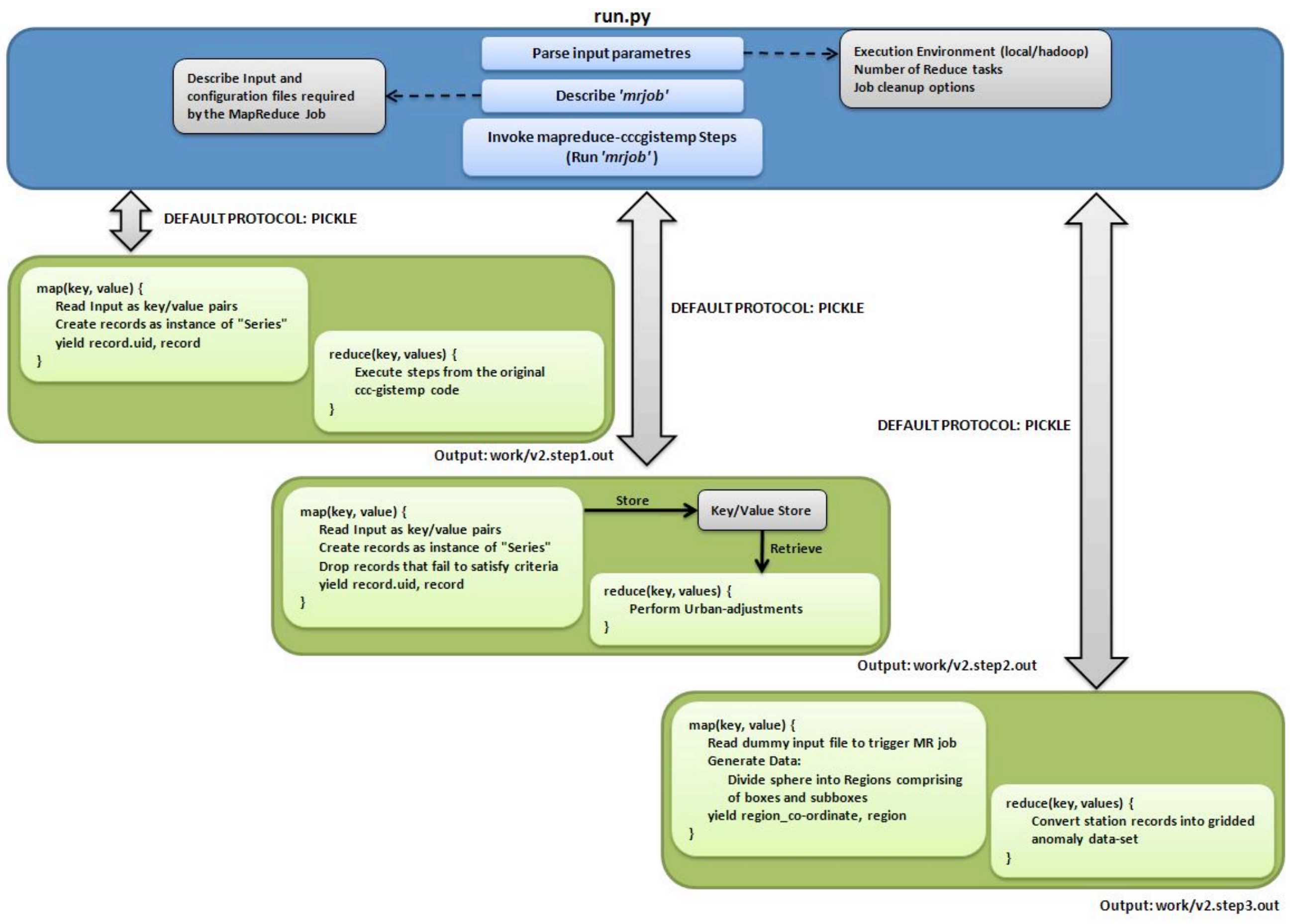
It must be noted that the input dataset will always be a tuple of land and ocean records consisting of 8000 lines each irrespective of the number of stations considered initially in the Step 0 input. Hence, Step 5 output is not scalable in terms of the input dataset. The only gain in performance obtained is by parallelising the operations across available reduce tasks. This parallelisation can however be achieved in a manner similar to that of Step 3. Since the primary goal of this work was to investigate the different approaches to using MapReduce on scientific application codes, rather than a complete parallelisation and optimisation of ccc-gistemp, Step 5 was not ported but left in its original state.

## 3.4 MapReduce-ccc-gistemp

The complete modified code is available from:

http://code.google.com/p/mapreduce-cccgistemp/downloads/list

Figure 5 shows the revised flow through the different steps of the application.


run.py
Parse input parametres
Execution Environment (local/hadoop)
Number of Reduce tasks
Job cleanup options
Describe Input and configuration files required by the MapReduce Job
Describe 'mrjob'
Invoke mapreduce-cccgistemp Steps (Run 'mrjob')
DEFAULT PROTOCOL: PICKLE
DEFAULT PROTOCOL: PICKLE
DEFAULT PROTOCOL: PICKLE
Output: work/v2.step1.out
Store
Key/Value Store
Retrieve
Output: work/v2.step2.out
Output: work/v2.step3.out

```
map(key, value) {
    Read Input as key/value pairs
    Create records as instance of "Series"
    yield record.uid, record
}

reduce(key, values) {
    Execute steps from the original
    ccc-gistemp code
}
```

```
map(key, value) {
    Read Input as key/value pairs
    Create records as instance of "Series"
    Drop records that fail to satisfy criteria
    yield record.uid, record
}

reduce(key, values) {
    Perform Urban-adjustments
}
```

```
map(key, value) {
    Read dummy input file to trigger MR job
    Generate Data:
        Divide sphere into Regions comprising
        of boxes and subboxes
    yield region_co-ordinate, region
}

reduce(key, values) {
    Convert station records into gridded
    anomaly data-set
}
```

**Figure 5: Flow diagram of the original ccc-gistemp code before modification**

## 4. RESULTS AND ANALYSIS

### 4.1 Benchmarking

Benchmarking of the modified ccc-gistemp code was performed on EDIM1[8], a cluster of commodity machines jointly funded by the Edinburgh Parallel Computing Center (EPCC) and the University of Edinburgh School of Informatics, primarily intended for Data-intensive research. The cluster is build from relatively inexpensive hardware with a dual core Intel ATOM processor on each node, which is comparatively slower to the current day high end processors. However, this machine has several fast disks connected directly to each of the 120 available nodes (distributed equally across three racks), ideally suited for data-intensive computing and research owing to its low latency and high I/O bandwidth. The Hadoop cluster setup on EDIM1 machine is based on the Cloudera distribution (CDH3) of Hadoop[9]. Table 1 provides details of the hardware and its configuration on the EDIM1 machine. Performance evaluations have been done on a dedicated subset of this machine configured as a 16 node cluster (one master node, one job tracker and fourteen slave nodes i.e. 28 cores), averaging over consecutive executions (observed variation between consecutive runs was always less than +/-1%).

| Category | Configuration |
|---|---|
| Number of Nodes | 120 (3 racks of 40 nodes each) |
| Processors/Node | Dual-Core Intel 1.6 GHz ATOM[10] processor |
| Disk Storage/Node | 1 x 256 MB Solid State Drive (SSD) + 3 x 2TB HDD |
| Network | 10 Gigabit Ethernet |
| OS | Rocks (Clustered Linux Distribution based on CENTOS)[11]<br>Linux Kernel Version 2.6.37 |
| JVM | 1.6.0_16 |
| Hadoop | 0.20.2<br>Cloudera Distribution version 3 (CDH3) |

Table 1: Hardware configuration of EDIM1 machine

Benchmarking of Step 1 (see Figure 6) for different samplings of the datasets show that the overall execution time decreases with the increase in the number of processing units. It also shows that the time required to perform I/O operations remains near constant for a given dataset and dominates as the number of cores increase.

Figure 7 shows the results from both an initial port of MapReduce of Step 2 and an optimised version. It was observed that the overall run time of MapReduce task was dominated by a single reduce task. By reviewing the input dataset it was identified that grouping values (station records) using the first two characters of the key (12-digit station id) created a severe imbalance in the number of records processed by each reduce task (recall that MapReduce assigns all values associated with the same key to a single reduce task). Further investigations revealed that the number of records associated with the station id beginning with '42' (USA), particularly '42572#######', were very large compared to other station ids causing this imbalance. The original ported code was then modified to account for this unequal distribution of values showing a much better scaling, though also hitting a limit constrained by the dataset size as in Step 1.

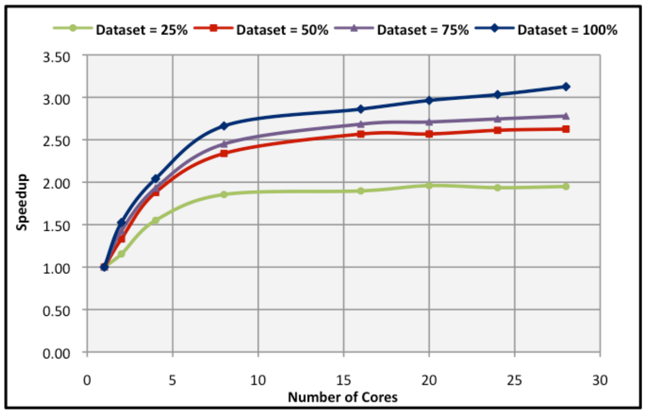


Figure 6: Plot of speedup for Step 1 for input dataset 100%, 75%, 50% and 25% respectively.

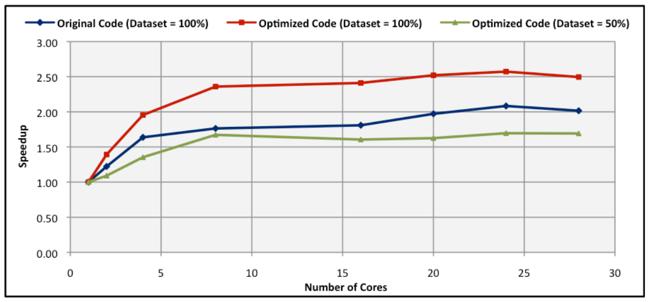


Figure 7: Plot of speedup for Step 2 for original and optimised code with dataset=100% and optimised code with dataset=50%.

It can be observed that a significant gain in performance can be obtained by avoiding the intermediate storage and retrieval. An important point to note from this study is the fact the I/O operations are performance inhibitors to a scalable system like MapReduce and must be minimised as much as possible. Figure 8 shows the effect of combining Steps 1 and 2.

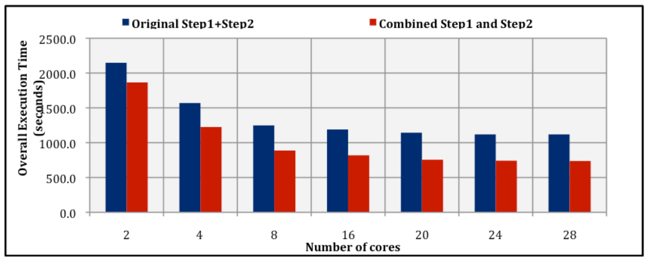


Figure 8: Performance improvement obtained by combining steps 1 and 2.

The speedup for Step 3 (see Figure 9) shows more interesting behaviour, increasing with the number of cores up to 20 cores, and then diminishing again. The timings for 50% and 100% of the dataset indicate that as with Steps 1 and 2, the MapReduce implementation of Step 3 is scalable with the input data size, up to a point where startup costs dominate. It can be observed from the plots that the choice of 'key' has an impact on performance. As already mentioned, any of the two coordinates (fractional degrees of longitude for eastern or western boundaries) can be used as the 'key'. Changing the 'key' results in regions being grouped

[8] http://www.epcc.ed.ac.uk/projects/research/dataintensive

[9] http://www.cloudera.com/hadoop/

[10] http://en.wikipedia.org/wiki/Intel_Atom

[11] http://www.rocksclusters.org/rocks-documentation/4.2/

differently to be processed by the reduce task, which in turn alters the amount of computation performed by each of the reduce tasks.

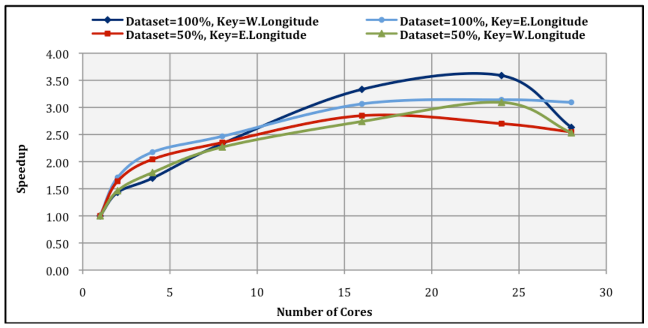


**Figure 9: Plot of speedup for Step 3 for keys W. Longitude and E. Longitude at 100% and 50% dataset respectively.**

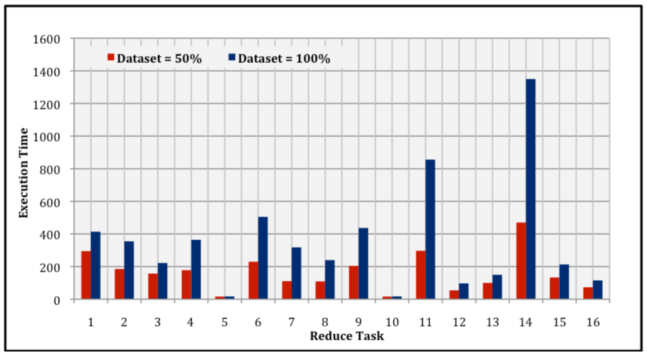


**Figure 10: Runtime distribution of reduce tasks in Step 3 of mapreduce-cccgistemp**

There are three reasons attributable to the diminishing speedup when using over 20 cores:

1. The number of unique keys in Step 3 is limited by the longitudes dividing the sphere. Thus, scaling beyond the maximum number of reduce tasks that can be created causes a significant decline in performance due to the presence of idle processing units. The start up costs associated with the MapReduce programming model can only be amortised when all the processing nodes are busy performing nearly the same amount of work all the time.

2. Uneven distribution of workload due to processing of uneven number of contributing stations by each reduce task and the Hadoop scheduler assuming that the amount of work done by each reduce task is roughly the same. Figure 10 indicates the runtime distribution of reduce tasks in Step 3, dominated by a single reduce task (task 14 in this example). The plot also indicates unequal workload distribution across all available reduce tasks.

3. The current implementation of the Hadoop load balancing strategy does not distribute workload based on the granularity of 'values' associated with a 'key', creating imbalance in the task execution times.

The dynamic load balancing strategy of MapReduce distributes tasks (map or reduce) to nodes as and when they finish processing the task at hand. This strategy ensures that the regions, grouped by the 'key' are processed continuously and concurrently with no idle time. However, assumptions made by the scheduler [23] sometimes causes unequal work load distribution which is particularly prominent in this example as the amount of computation required within a region depends on the number of contributing stations within that region. Currently, there is no way for the scheduler to obtain this piece of information while scheduling reduce tasks. Hence, the last set of tasks may finish at unequal times depending on the workload, causing an overall reduction is performance due to this 'slow' task. Thus it can be concluded that MapReduce is ideally suited for jobs that are large, but can be divided into smaller units of nearly equal size. A single large task can slow the overall performance.

There have been studies [7][13] of the impact of variable task runtimes in MapReduce applications. Fine partitioning of the reduce tasks such that there are more partitions than the number of available reducers (currently, the number of partitions is equal to the number of reducers) can distribute chunks of complex reduce tasks evenly among the available reducers, significantly minimizing the impact of variable task runtimes. However this project has shown that but redesigning a complex compute intensive algorithm for the MapReduce framework, including the choice of keys, requires domain expertise.

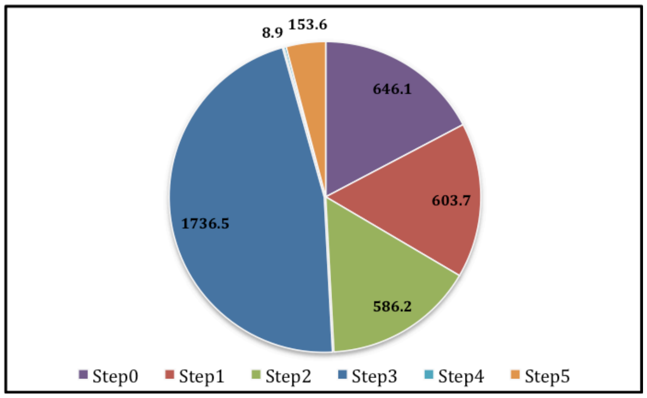


**Figure 11: Profiling ported ccc-gistemp code (seconds)**

Figure 11 shows the profiling of the ported *mapreduce-cccgistemp*, where the MapReduce Steps 1, 2 and 3 are parallelised across 16 cores. The CPU bound Steps 1 and 3 have found significant improvement in performance by distributing the compute-intensive tasks across 16 cores when compared to the original *ccc-gistemp* profiling chart in Figure 2. The improvement in performance of Step 2 is not as significant as that of Steps 1 and 3, for reasons already explained in the benchmarking analysis.

The final series output of the application can be seen in Figure 12.

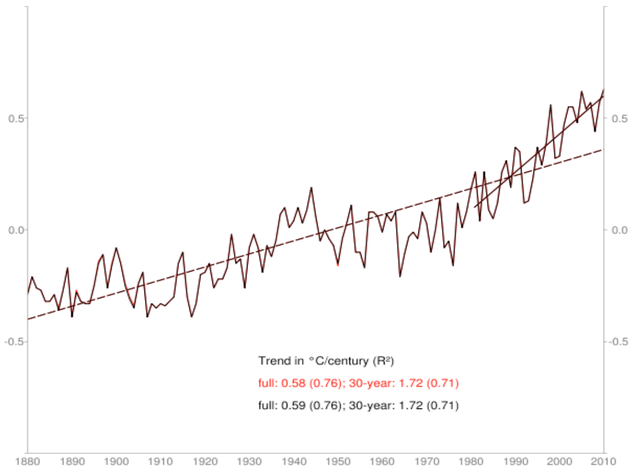


**Figure 12: Graph comparing the global temperature anomaly of original ccc-gistemp code (black) and ported mapreduce-cccgistemp (red)**

## 5. CONCLUSIONS AND FURTHER WORK

### 5.1 Impacts on Scalability

Scalable algorithms are highly desirable in both compute-intensive and data-intensive applications. Scalability along two particular dimensions is ideally applicable for data-intensive computing [14]. First in terms of data: given twice the amount of data, the same algorithm should take at most twice as long to run. Second, in terms of computing resource: given a cluster twice the size, the same algorithm should take no more than half as long to run. From the work we have done, the Step 1 and Step 3 analyses show that the MapReduce programming model is efficient and scalable across processing units and data sizes for CPU-intensive scientific tasks. Increasing the data and/or computation negates the impact of overheads induced by MapReduce programming model, thereby improving the overall speedup.

Hadoop MapReduce uses block scheduling scheme for assigning input data to the available nodes for processing, dynamically at runtime. This runtime scheduling strategy enables MapReduce to offer elasticity and remain fault tolerant by dynamically adjusting resources (adding nodes for scalability and removing failed nodes for fault tolerance) during job execution. However, it introduces runtime overheads that may slow down the execution of MapReduce job, and may not be suitable for scientific applications where there is an uneven distribution of data and processing. Skewed data in compute intensive processing can have significant impact on the overall performance. Improved load balancing strategies can mitigate the impacts of skew, thus enabling MapReduce to provide an ideal programming abstraction for processing data and compute intensive scientific applications.

### 5.2 Time and Ease of Porting

In distributed memory architectures, parallelising sequential code with MPI would require a significant amount of time to alter the existing code structure to use the MPI library. In this work, it was observed during the porting exercise that it is not essential to comprehend the entire algorithm to be able to port to MapReduce, with the framework handling the details of parallelisation, distribution of computation, load balancing, task management and fault tolerance. However, it is essential to understand the data-access patterns within the algorithm to be able to modify the algorithm to operate on key/value pairs, and lessen the need for global synchronisation across all reduce tasks.

Additionally, it was observed that algorithms designed to operate on groups of data are easier to port to MapReduce. These datasets can easily be mapped as key/value pairs with values associated with the same key processed by algorithms ported to the reduce function. In some situations the existing logic may not be directly portable to MapReduce but with small changes in the data access pattern, data-intensive algorithms can be ported. An example of such scenario is discussed in Section 3.3.4.

Algorithms that introduce a dependency between tasks while processing are harder to port to MapReduce. Since the MapReduce framework does not provide any direct interface to share data between dependent tasks, alternate techniques such as synchronisation with a single reduce task and use of external key/value store for shared data can be incorporated to overcome this limitation.

Thus it can be concluded from this porting exercise that the time and effort required to port the code when compared to the scalability obtained is quite low, when compared to other parallelisation techniques like MPI. Nevertheless, an understanding of the basics of parallel programming techniques can greatly help when determining changes required to the data access patterns.

### 5.3 Further work

The main area of focus in this project was to evaluate the applicability of MapReduce to particular data and compute intensive tasks of the ccc-gistemp code.

At present, whilst results show scaling after porting to MapReduce, a further study which utilises larger datasets would better test the scalability. At present, the issue is that this project used the actual dataset and domain knowledge is required to create a larger synthetic dataset.

Porting of ccc-gistemp to other scalable systems intended for data-intensive computing such as Dryad [12], All-Pairs [17] and Pregel [16] would provide a comparative study of the various programming abstractions that are suitable. Likewise implementations of MapReduce which use existing high-performance shared filesystems are now available (e.g. MARIANE [6]) which might improve the performance of the ported ccc-gistemp, particularly Step 2.

A key-value store based MapReduce framework has been implemented [18] which might overcome some of the limitations imposed by the current implementation of MapReduce. This new implementation is particularly aimed at improving the performance of HPC applications intended to use the MapReduce framework. Further work could determine if this new implementation addresses the limitation is associated with the sharing of data between map and reduce tasks during execution of MapReduce jobs for the ccc-gistemp code.

## 6. RELATED WORK

MapReduce is extensively used within Google for processing large volumes of raw data such as crawled documents and web request logs [4]. With its widespread adoption via an open source implementation called Hadoop [14], primarily for data-intensive computing, there have been many evaluations of this programming model using large volumes of web and textual data. However, there have been few evaluations with scientific data.

Zhu et al. [24] evaluated the feasibility of porting two applications (Water Spatial and Radix Sort) from the Stanford SPLASH-28 suite to the Hadoop implementation of MapReduce. Performance bottlenecks with porting were identified and suggestions provided for enhancing the MapReduce framework to suite these applications, in particular to reduce the overhead introduced from shared data synchronisation.

The main attributes of the implementation strategy that were considered in porting these applications were the data access patterns and computational steps. It was identified that most scientific applications require shared data and hence synchronisation was a major source of overhead. Additionally, the probability of scientific applications using matrices and multi-dimensional arrays for their processing was much higher than simple data-structures.

Global synchronisation across all reduce tasks in a MapReduce job was achieved with a single reduce task. Suggestions to provide better support for distributing array and matrices within the HDFS to reduce communication overheads were made. Also, the advantages of directly dumping the output of first stage to the second in a multi-stage job, without the need for intermediate HDFS store were highlighted to reduce I/O overheads.

Ekanayake et al. [5] evaluated the Hadoop implementation of MapReduce with High Energy Physics data analysis. The analyses were conducted on a collection of data files produced by high-energy physics experiments, which is both data and compute intensive. As an outcome of this porting, it was observed that scientific data analysis that has some form of SPMD (Single-Program Multiple Data) algorithm is more likely to benefit from MapReduce when compared to others. However, the use of iterative algorithms required by many scientific applications were seen as a limitation to the existing MapReduce implementations. It was suggested that support for directly accessing data in binary format could benefit many scientific applications which would otherwise need some form of data transformation, reducing performance.

## 7. ACKNOWLEDGMENTS

Our thanks to EPCC and the School of Physics MSc in High Performance Computing programme which supported this work. We also thank Paolo Besana, Adam Carter and Gareth Francis for their assistance with the use of the EDIM1 machine; Derek Wang, Adam Barker and Alex Voss for their assistance with the use of the St. Andrews STACC cloud used for initial development; and the mrjob and ccc-gistemp communities for their advice and support throughout the project.

## 8. REFERENCES

[1] Apache Hadoop framework. 2008. http://hadoop.apache.org/ (accessed June 27, 2011).

[2] Barroso, Luiz André, and Urs Hölzle. The Datacenteras a Computer: An Introduction to the Design of Warehouse-Scale Machines. Morgan and Claypool Publishers, 2009.

[3] Butt, Ali R., Prashant Pandey, Karan Gupta, and Gunaying Wang. "A Simulation Approach to Evaluating Design Decisions in MapReduce Steps." IEEE International Symposium on Modeling Analysis Simulation of Computer and Telecommunication Systems. 2009: IEEE, 2009. 1-11.

[4] Dean, Jeffrey, and Sanjay Ghemawat. "Mapreduce: Simplified Data Processing on Large Clusters." OSDI'04. 2004. 137-150.

[5] Ekanayake, Jaliya, Shrideep Pallickara, and Geoffrey Fox. "MapReduce for Data Intensive Scientific Analyses." IEEE Fourth International Conference on eScience. IEEE, 2008. 277- 284.

[6] Fadika, Zacharia, Elif Dede, Madhusudhan Govindaraju and Lavanya Ramakrishnan. "MARIANE: MApReduce Implementation Adapted for HPC Environments." 12th IEEE/ACM International Conference on Grid Computing. 2011.

[7] Gufler, Benjamin, Nikolaus Augsten, Angelika Reiser, and Alfons Kemper. "Handling Data Skew in MapReduce." CLOSER 2011 - International Conference on Cloud Computing and Services Science. 2011.

[8] Hansen, J., R. Ruedy, J. Glascoe, and M. Sato. "GISS analysis of surface temperature change." J. Geophys. Res.,104, 1999: 30,997-31,022.

[9] Hansen, J., R. Ruedy, M. Sato, and K. Lo. "Global Surface Temperature Change." J. Geophys. Res, 48, 2010: 1-29.

[10] Hansen, James, and Sergej Lebedeff. "Global Trends of Measured Surface Air Temperature." J. Geophys. Res., 92, 1987: 13,345-13,372.

[11] Hey, Tony, Stewart Tansley, and Kristin Tolle. The Fourth Paradigm: Data-Intensive Scientific Discovery. Redmond, Washington: Microsoft Research, 2009.

[12] Isard, Michael, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: distributed data-parallel programsfrom sequential building blocks." ACM SIGOPS Operating Systems Review. 2007. Volume: 41, Issue: 3, Pages: 59.

[13] Kwon, YongChul, Magdalena Balazinska, and Bill Howe. "A Study of Skew in MapReduce Applications." Open Cirrus Summit 2011. Russia, 2011.

[14] Lin, Jimmy, and Chris Dyer. "Data Intensive Text Processing with MapReduce."

[15] Mackey, Grant, Saba Sehrish, John Bent, Julio Lopez, Salman Habib, and Jun Wang. "Introducing map-reduce to high end computing." 3rd Petascale Data Storage Workshop. IEEE, 2008. 1-6.

[16] Malewicz, Grzegorz, et al. "Pregel : A System for Large-Scale Graph Processing." 28th ACM Symposium on Principles of Distributed Computing (PODC 2009). Calgary, Alberta, Canada: ACM, 2009. Volume: 9, Pages: 6-6.

[17] Moretti, Christopher, Hoang Bui, Karen Hollingsworth, Brandon Rich, Patrick Flynn, and Douglas Thain. "All-Pairs: An Abstraction for Data-Intensive Computing on Campus Grids." IEEE Transactions on Parallel and Distributed Systems. IEEE, 2010. Volume: 21, Issue: 1, Pages: 33-46.

[18] Ogawa, Hirotaka, Hidemoto Nakada, Ryousei Takano, and Tomohiro Kudoh. "An Implementation of Key-value Store based MapReduce Framework." 2010 IEEE Second International Conference on Cloud Computing Technology and Science. IEEE, 2010. 754- 761.

[19] Peterson, T.C., T.R. Karl, P.F. Jamason, R. Knight, and D.R. Easterling. "First difference method: Maximizing station density for the calculation of long-term global temperature change." J. Geophys. Res.,103, 1998: 25,967-25,974.

[20] Reynolds, R. W., N. A. Rayner, T. M. Smith, D. C. Stokes, and W. Wang. "An improved in situ and satellite SST analysis for climate." J. Clim., 15, 2002: 1609-1625.

[21] White, Tom. Hadoop: The Definitive Guide, Second Edition. O'Reilly, 2010.

[22] Xie, Jiong, Shu Yin, Xiaojun Ruan, Zhiyang Ding, and Yun Tian. "Improving MapReduce performance through data placement in heterogeneous Hadoop clusters." 2010 IEEE International Symposium on Parallel Distributed Processing Workshops. IEEE, 2010. 1-9.

[23] Zaharia, Matei, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. "Improving MapReduce Performance in Heterogeneous Environments." 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008. San Diego, 2008. 29-42.

[24] Zhu, Shengkai, Zhiwei Xiao, Haibo Chen, Rong Chen, Weihua Zhang, and Binyu Zang. "Evaluating SPLASH-2 Applications Using MapReduce." APPT09. 2009. 452-464.