Large-Scale Parallel Statistical Forecasting Computations in R

Murray Stokely* Farzan Rohani* Eric Tassone*

Abstract

We demonstrate the utility of massively parallel computational infrastructure for statistical computing using the MapReduce paradigm for R. This framework allows users to write computations in a high-level language that are then broken up and distributed to worker tasks in Google datacenters. Results are collected in a scalable, distributed data store and returned to the interactive user session. We apply our approach to a forecasting application that fits a variety of models, prohibiting an analytical description of the statistical uncertainty associated with the overall forecast. To overcome this, we generate simulation-based uncertainty bands, which necessitates a large number of computationally intensive realizations. Our technique cut total run time by a factor of 300. Distributing the computation across many machines permits analysts to focus on statistical issues while answering questions that would be intractable without significant parallel computational infrastructure. We present real-world performance characteristics from our application to allow practitioners to better understand the nature of massively parallel statistical simulations in R.

Key Words: Statistical Computing, R, forecasting, timeseries, parallelism

1. Introduction

Large-scale statistical computing has become widespread at Internet companies in recent years, and the rapid growth of available data has increased the importance of scaling the tools for data analysis. Significant progress has been made in designing distributed systems to take advantage of massive clusters of shared machines for long-running batch jobs, but the development of higher-level abstractions and tools for interactive statistical analysis using this infrastructure has lagged. It is particularly vital that analysts are able to iterate quickly when engaged in data exploration, model fitting, and visualization on these very large data sets.

Supporting interactive analysis of data sets that are far larger than available memory and disk space on a single machine requires a high degree of parallelism. At Google, parallelism is implemented using shared clusters of commodity machines [5]. This paper describes a statistical computing framework built on top of Google's distributed infrastructure and illustrates an example use case based on statistical forecasting of large numbers of time series.

The rest of this paper is structured as follows. In Section 2 we provide background information. Then, in Sections 3 and 4, we explain the design and implementation of a set of R [21] packages to take advantage of the distributed systems available at Google for high-level statistical computing tasks. After that, in Sections 5 and 6, we offer an application of this infrastructure for forecasting large numbers of time series, describing the parallel algorithms and providing experimental results from our clusters. We end with overall conclusions in Section 7.

*Google, Inc.

2. Background

2.1 Data Analysis in R

Split-apply-combine [26] is a common strategy for data analysis in R. The strategy involves splitting up the data into manageable chunks, applying a function or transformation on those chunks, and then combining the results. Such techniques map closely to the MapReduce [11] programming model for large compute clusters. As a vector language, R includes built-in functions for *map* or *apply*, and many functions that take lists of inputs act as reduce functions. The MapReduce paradigm extends the strategy to multiple nodes by sending a subset of the input data to Mappers running on different nodes. There are a number of parallel apply packages available in R [10, 24] that allow a user to do a parallel Map or Apply step as long as the results can fit in memory on the calling R instance, to essentially implement a MapReduce with only a single Reducer.

2.2 Related Work

Other parallel R implementations are surveyed in [22]. These implementations depend on technologies such as MPI or TCP/IP sockets relying on shared NFS storage for small clusters of workstations. In addition, they require manual pre-configuration of R and needed libraries on worker nodes. We work with much larger clusters that may write to other non-POSIX parallel filesystems such as GFS [15] or Bigtable [7]. The scale of these shared clusters precludes manual pre-staging of R, and thus we are not able to use these frameworks. The pR system [18] is a framework that transparently parallelizes R code through the use of runtime static analysis, but works only for non-interactive batch jobs and sequential processing tasks where the data set fits fully in memory on a single machine.

Our approach is most similar to the RHIPE package [16], which implements a more complete MapReduce environment with user-provided Map and Reduce functions written in R that run on multiple nodes with Hadoop. In contrast to RHIPE, though, we instead focus on larger scale clusters where more automated node setup is essential. Furthermore, in our system all definitions in the calling environment are serialized to disk and distributed to worker tasks, allowing the workers to reference functions, classes, and variables using the lexical scoping rules expected in the R language. We also take advantage of [25] to speed up the computations on individual multi-core nodes. However, our interest here is exclusively on scaling beyond multiple nodes and so we assume our applications have already been tuned to take advantage of the number of cores per machine.

3. Map: Parallel Apply

In this section we describe the high-level design and implementation details for a series of R packages facilitating the use of Google datacenters for executing massively parallel R code.

3.1 Design Goals

Our design goals were based on observations of how the use of R at Google has evolved over the past several years. In particular, these goals included:

- Facilitate parallelism of computations on up to thousands of machines without access to shared NFS filesystems.

- Make distribution of code and required resources as seamless as possible for analysts to minimize code modifications required to enable parallelism.

- No setup or pre-installation of R or specific libraries should be required on the machines in the cluster. A virtual machine for the workers should be created dynamically based on the global environment and available libraries of the caller.

- Return results of parallel computations in list form directly back to the calling interactive session, as with *lapply* in R.

- Allow the parallel functions to be used recursively, so that MapReduce workers can in turn spawn additional MapReduces.

3.2 Implementation Overview

Figure 1 shows an overview of the basic implementation of our Parallel Map framework. The three main steps of the process are described below.

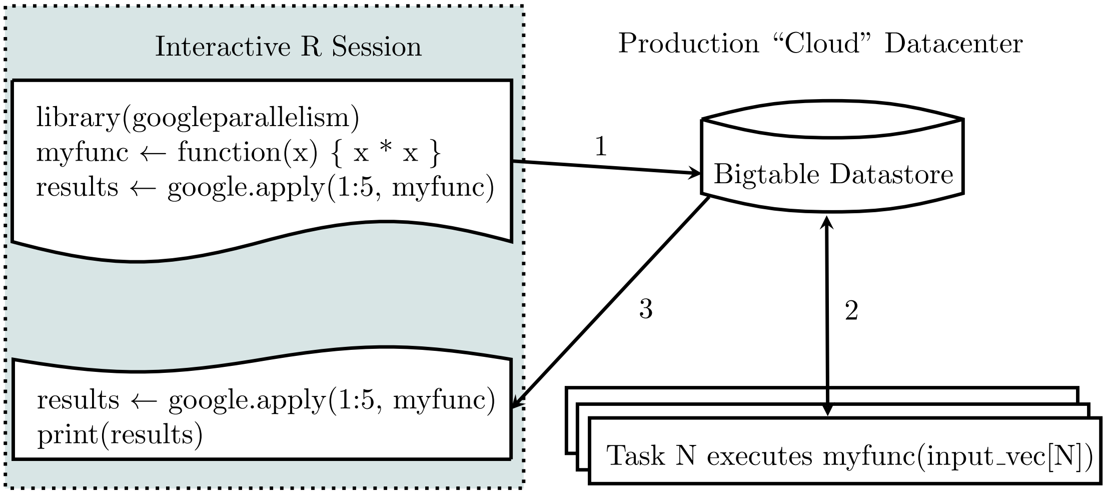

Interactive R Session
library(googleparallelism)
myfunc ← function(x) { x * x }
results ← google.apply(1:5, myfunc)
results ← google.apply(1:5, myfunc)
print(results)
Production "Cloud" Datacenter
Bigtable Datastore
1
2
3
Task N executes myfunc(input_vec[N])

Figure 1: Parallel Map

1. First, the user's code calls *google.apply()* with a list of inputs and a provided function, *FUN*. An archive is dynamically created on the client including R and all of the needed libraries and then staged to the cluster management system in a datacenter with available resources. *FUN* and its environment are serialized and written out to a Bigtable in that datacenter.

2. Second, workers tasks are spawned using the dynamically generated virtual machines providing access to all of the R packages that were loaded in the calling instance's R session. These workers read in the serialized environment from the Bigtable, execute the provided function over a unique element of the input list, and write out the serialized results to a different column of the Bigtable.

3. Third, and finally, the calling R instance reads back in the serialized return values from each worker task, performs the necessary error handling, and returns the computed list to the *google.apply()* caller.

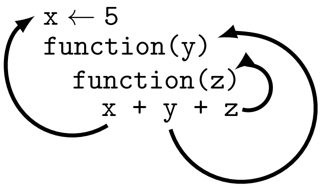
.

The next three subsections provide more detail about each of these three steps.

3.3 Lexical Scoping and Serialization in R

To reduce the effort of utilizing Google's parallel infrastructure for statistical computations, we opted to automatically serialize the calling environment and distribute it to the parallel workers. This allows users to reference visible objects from their calling frame in a way that is consistent with the R language, and without requiring cumbersome manual *source()* calls of distributed .R files on the worker tasks.

The R language's lexical scoping rules require that free variables in a function be resolved in the environment that was active when the function was defined [14]. Figure 2 shows a brief example of a function definition and the bindings of the variables to different calling environments. Serialization of a function thus requires the serialization of the calling environments all the way up to the global environment to ensure that any possible variable references or function calls used in the innermost functions are available when the serialized environment is loaded on another machine. The default R serialization mechanism described in [23] handles these details to allow us to stream the complete calling environment from the interactive or batch R instance and the spawned worker tasks.



```
x ← 5
function(y)
  function(z)
    x + y + z
```

Figure 2: Lexical Scoping

Algorithm 1 runs on the calling R instance and shows how the input list is split up, and the input function is serialized with the calling environment and distributed to worker tasks. One caveat with this approach is that package namespaces are serialized by name. This means that all loaded packages must be packaged up inside the virtual machine that is dynamically generated and distributed to the spawned worker tasks. Furthermore, any mutable objects, such as user-defined environments, that are hidden within a package namespace will not be serialized—a very rare occurrence, in practice, that is the price of not requiring any manual setup of worker nodes.

3.4 Worker Scheduling

In shared cluster environments with thousands of machines the runtime of long-term statistical computations will be affected by failures, upgrades, workload changes, task pre-emptions by higher priority jobs, and other factors. To deal with these events, individual worker tasks that fail will need to be restarted or migrated to separate machines. In some cases, backup workers may need to be scheduled if some particular workers are taking longer than others due to hardware, network,

Algorithm 1 Task distribution

```
# Simple case, we have 1 worker for each list element :
if (length(x) <= max.workers) {
  assign(".G.INPUT", x, env=.GlobalEnv)
  assign(".G.FUNCTION",
         function(x) { FUN(x, ...) }, env=.GlobalEnv)
} else {
  warning("length(x) > max.workers, some worker tasks will ",
          "execute over more than 1 input.")
  new.input = InputSplit(x, max.workers)
  assign(".G.INPUT", new.input, env=.GlobalEnv)
  assign(".G.FUNCTION",
         function(x) { lapply(x, FUN, ...) }, env=.GlobalEnv)
}

# Step 2.  Save the environment of the calling session
shared.env <- tempfile(".Rdata")
save(list = ls(envir = .GlobalEnv, all.names = TRUE),
     file = shared.env, envir = .GlobalEnv)

# Add the .Rdata file, R, and packages to stage in our VM.
packages <- list(VMPKG(files=shared.env))
packages <- c(packages, VMPKG(files=GetRFiles()))

# Get Bigtable rowkey where results should be written.
key <- GetBigtableKey()

# Launch the tasks with the created VM.
LaunchRVMs(max.workers, packages, key)
```

or contention from other shared workload jobs on that machine. In other cases, we may be able to return when 95% of the workers have completed to provide most of the accuracy of our computation at a fraction of the runtime cost compared to waiting for all workers to complete.

There are two parameters that we expose to the callers for scheduling their R worker tasks: one, the total number of failures we will tolerate from an individual worker task; and two, the total number of worker failures across all tasks. The first parameter should scale with the total runtime of the job, and is set to a reasonable default since we do not typically know the runtime of a job before first execution. The second parameter should scale with the total number of parallel tasks that were launched. We also provide deadlines and other scheduling parameters to give users greater control over the worker tasks. Dealing with stragglers and scheduling is an active area of research in MapReduce [2].

3.5 Error Handling and Return Values

When the worker tasks have completed, the calling R instance reads in the serialized results from the Bigtable, unserializes the result for each worker, and returns R language results. Depending on the scheduling parameters in use, all of the workers may have completed successfully, some may have failed to run completely because of resource constraints on the scheduling system, or may have run but reached an exception in the R language code executed on the workers. In all cases we seek to examine the results and promote errors from any of the workers to the attention of the caller. By default, the worker code is wrapped in a *try()* so the calling instance examines the returned output after unserializing it from the Bigtable and issues a *warning()* with the task number and exact error message from any *try-error*s encountered by any of the workers. If all of the workers returned a *try-error*, then these warnings are promoted to a *stop* error.

So far, we have described a massively parallel approach to the common Split-Apply-Combine data analysis paradigm, but we have not fully taken advantage of MapReduce because the results from all Mappers return to the calling R instance—essentially a MapReduce with a single reducer. The next section describes the extensions necessary for statistical computations where the aggregate of the outputs from the machines running the Map function is far too large for the memory of a single machine.

4. Reduce

In the traditional MapReduce programming model [11], the *Map* function produces a set of intermediate key/value pairs which are grouped together by intermediate key and then passed to a *Reduce* function. The reduce function is passed an iterator over the intermediate inputs, so it can process more records than will necessarily fit inside memory on a single Reduce worker.

There is limited support for streaming statistical computations in R, and so we have taken a hybrid approach for MapReduce-like statistical computations inside Google. This approach involves using a scalable query processing system directly over the intermediate outputs to implement the types of aggregations typically performed in a *Reduce*. Since our parallelism implementation allows individual *Map* workers to in turn generate separate parallel R applications, possibly running in a different datacenter, we can chain together a series of computations at the R level and then perform the final aggregation step with a distributed query system.

4.1 Distributed Result Aggregation in Dremel

Individual *Map* functions written in R can write out intermediate results in a variety of formats. We store Protocol Buffer outputs in the nested column-striped storage representation described in [19]. R data.frames and lists are written directly to this format from the *Map* functions in R code. When the *Map* functions complete, the resulting columnar data files are queried directly using an R-language interface to the Dremel scalable ad-hoc query system. In contrast to Pig [20], Hive [1], or Tenzing [8], these queries execute immediately against the data in-place, without having to launch separate MapReduce jobs over the data.

In the past two Sections (3 and 4) we have described the design and implementation of R packages that take advantage of the distributed systems available at Google for high-level statistical computing tasks. In the next two Sections (5 and 6) we see these R packages in action in an illustrative example based on statistical forecasting of a large number of time series.

5. Application to Forecasting

5.1 Forecasting at Google

At Google we use forecasting for numerous purposes, including evaluating performance and anomaly detection. We forecast many quantities (such as queries, revenue, number of users, etc.) for many services (such as web search, YouTube, etc.) and many geographic locations (such as global, continents, countries, etc.), which involves forecasting thousands of time series every day.

These time series exhibit a variety of trends, seasonalities, and holiday effects. For example, the number of Google searches for the query "pizza" grows with a different rate compared to the query "car insurance"[1]. Figure 3 shows that the two queries also differ in their behavior during the end-of-year holiday season, when "pizza" queries spike while "car insurance" queries dip. Consequently, we may need to use different models to forecast "pizza" and "car insurance" queries.

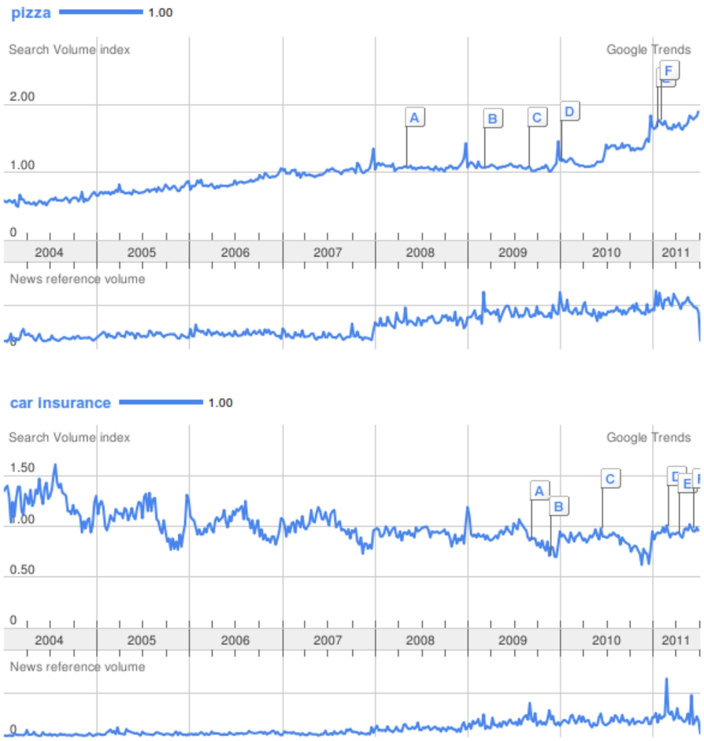


Figure 3: Trend and seasonality for "pizza" and "car insurance" queries from Google Trends.

Building and updating forecasting models individually for thousands of different time series is expensive and impractical, and requires a considerable amount of human intervention—highlighting the need for a generic forecasting methodology that is robust and provides an adequately accurate forecast for each time series. In this section, we focus on how the *googleparallelism* package in conjunction with Google's infrastructure can be a useful, practical, and inexpensive method for building, evaluating, and engineering such a forecasting methodology in the R programming

[1] All data sets used herein are publicly available from Google Trends, `google.com/trends`.

language. A high-level overview of our forecasting methodology is provided in the next sub-section, but further details are beyond the scope of this paper.

5.2 Brief Overview of Forecasting Methodology

As opposed to fine-tuning a single model, we generate forecasts by averaging ensembles of forecasts from different models [3, 4, 9, 17]. The idea is to reduce the variance and gain robustness by averaging out the various errors from individual models. Figure 4 shows the ensemble of forecasts for weekly "pizza" searches. The black curve in the middle is the trimmed mean[2] of individual forecasts at each point in time. This forecasting methodology does not provide the best forecast for every single case, but serves us well in large-scale forecasting, where it consistently produces adequate forecasts with minimal human intervention.

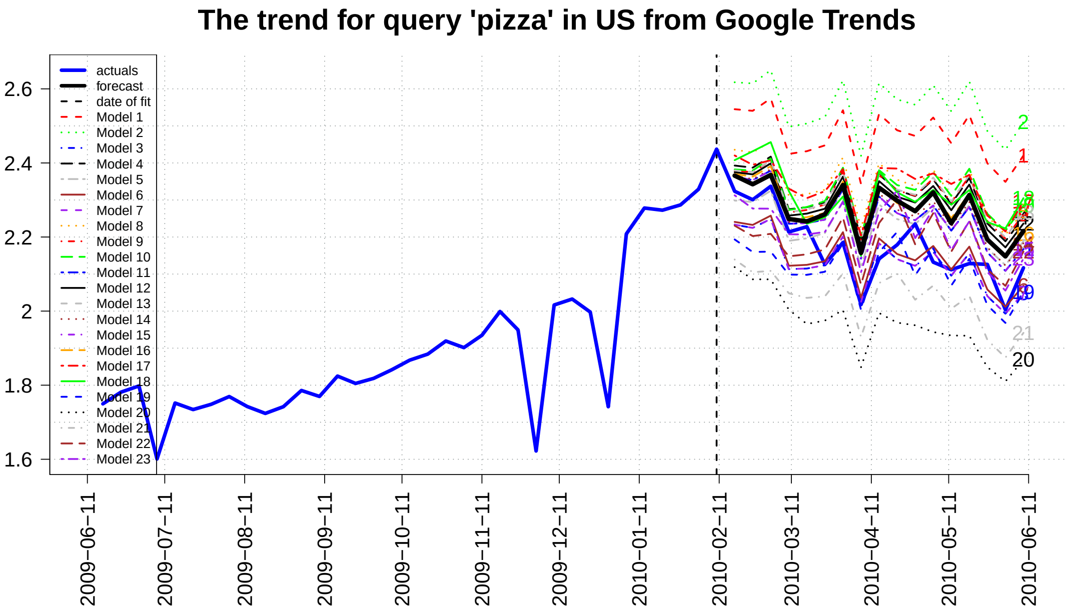


Figure 4: The ensemble of forecasts for US "pizza" queries

The robustness of ensemble averaging and the convenience of using R come at a price. Combining multiple models makes it difficult to quantify the uncertainty associated with the forecast process—that is, we cannot build confidence intervals or perform statistical inference. Using simulation-based methods is a typical solution to the problem, but these methods are computationally intensive, particularly on the scale at which we seek to operate. In the next sub-section we describe how the *googleparallelism* package can help us in building forecast confidence intervals using parallel simulations.

5.3 Forecast Confidence Intervals

Forecasts inevitably differ from the realized outcomes or *actuals*. Discrepancies between forecasts and actuals reflect forecast uncertainty or true differences. Because our ensemble methodology does not provide a measure of statistical uncertainty, we generate simulation-based confidence intervals, which necessitates a large number

[2] The top and bottom 20% of individual forecasts are trimmed at each point in time.

of computationally intensive realizations. We use the general framework described in the previous sections to generate these confidence intervals over many more time series than was previously possible.

We use a computationally intensive simulation method called “bootstrapping” [12, 13] to project a sample of trajectory paths for an arbitrary number of periods into the future and extract the distribution of simulated traffic at each time. This simulation-based method enables us to compute the uncertainty associated with different attributes of the time series, such as year-over-year growth values and daily, weekly, and quarterly totals.

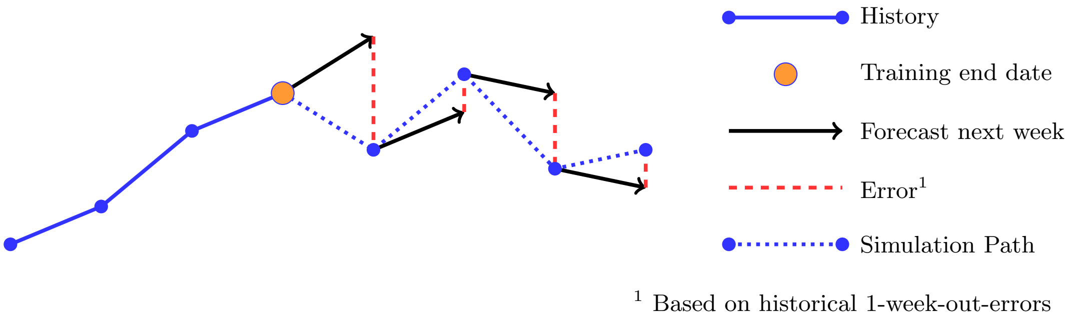
5.4 Iterative Forecasts and R Map Function

We use *iterative forecasts* to simulate future values as depicted in Figure 5. For one realization of values in the next n weeks, we repeat the following three steps n times:

1. At the training end date, we forecast the next week’s value.

2. We adjust the forecast value in Step 1, multiplying by an adjusting factor (a randomly generated number based on the distribution of historical one-week-out forecasting errors).

3. We add the adjusted value in Step 2 to the history as a new actual and move the training end date to the next week.



[1] Based on historical 1-week-out-errors

Figure 5: Iterative Forecasts.

Figure 6 depicts one thousand realizations for the normalized number of weekly “pizza” searches for 13 weeks starting at our training end date, 16 February 2010. At each time t, we take the $\alpha/2$ and $(1-\alpha/2)$ quantiles of the realizations as the lower and upper bounds of the $(1-\alpha)\times 100\%$ forecast confidence interval. These intervals are point-wise, and for 95% confidence regions we expect 5% of the actuals fall out of the bands.

To get a thousand realizations for the next year (52 weeks), we need to run the forecasting code $1000 \times 52 = 52,000$ times. A single run of the forecasting code takes about 5 seconds, so computing a one-year-long confidence region would take $5 \times 52,000$ seconds ≈ 72 hours on a single workstation, which is impractical for our purposes.

For forecast simulations, we can only parallelize the between-realization forecasts, while within-realizations forecasts must be run on the same machine due to

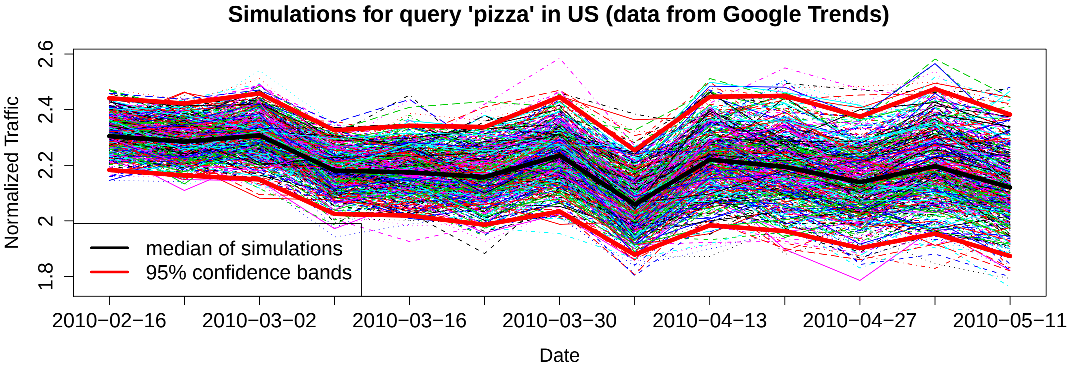


Figure 6: One thousand realizations of "pizza" traffic trajectories.

the iterative nature of the method—each forecast for the same realization uses the output of the previous forecast in the chain as an input. Overall, the R package reduced the running time in the above example to 15 minutes (about 300 times faster). Experimental measurements of the task setup costs and runtime distribution of the tasks is presented in Section 6.

5.5 Forecast Evaluation and R MapReduce

When the goal is to have a robust forecasting method, we must make sure that proposed changes to the forecasting models and parameters would improve forecasting accuracy in general and not only for particular time series—which means we need a comprehensive performance evaluation for our forecasting models over the large set of time series we forecast. As we expand the scope of Google forecasting project to handle more cases, changes in our forecasting codebase become more frequent. Therefore, any method we use for evaluating the performance of the forecast needs to be fast and efficient.

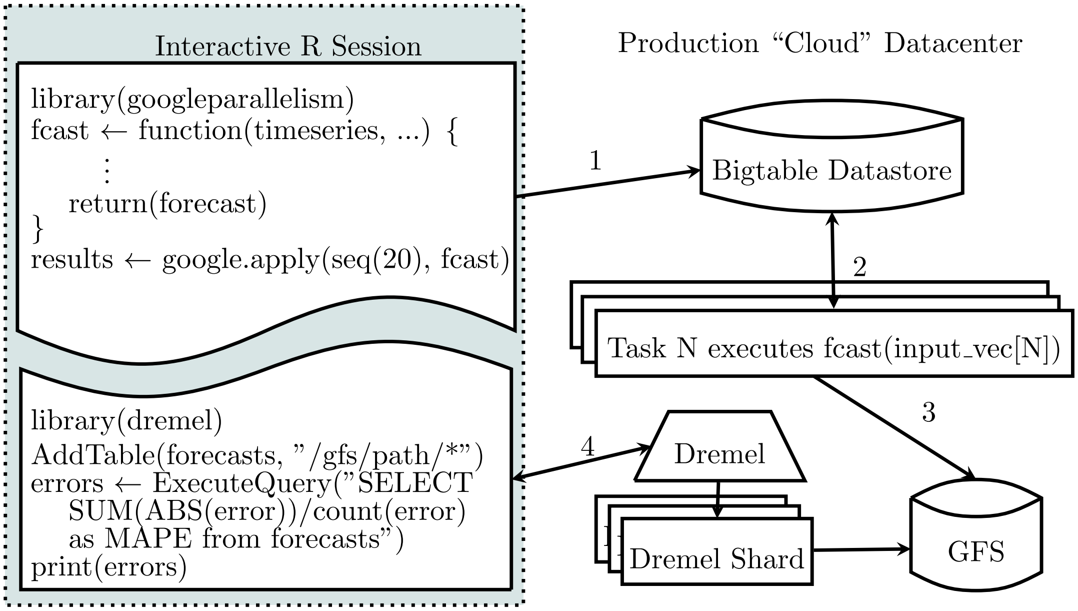
We used Google's R MapReduce to build an efficient forecast evaluation system. After trying out a change in our forecasting code, we use the output of this system to help decide whether or not the proposed change should be implemented. Our evaluation system consists of four major parts which are depicted in Figure 7:

1. A Google *datastore* (Bigtable) that stores an inclusive set of time series we forecast.

2. A *forecast mapper* that parallelizes the current and the updated forecast on a Google data center for the time series in the datastore (Part 1) at different prespecified training end dates. The output of each forecast is an R data.frame with columns specifying the time series, the forecasting model, the training end date, the forecast/actual date, the length of forecast horizon and the forecasting error, $(forecast - actual)/actual$.

3. The intermediate data.frames are saved on GFS in the nested column-striped format explained in Section 4.

4. A *forecast reducer* that uses the Dremel query system to aggregate the results of the forecast mapper and provides information regarding the forecast performance (e.g., MAPE metrics).



```
library(googleparallelism)
fcast ← function(timeseries, ...) {
        ⋮
    return(forecast)
}
results ← google.apply(seq(20), fcast)
```

```
library(dremel)
AddTable(forecasts, "/gfs/path/*")
errors ← ExecuteQuery("SELECT
    SUM(ABS(error))/count(error)
    as MAPE from forecasts")
print(errors)
```

Interactive R Session
Production "Cloud" Datacenter
1
Bigtable Datastore
2
Task N executes fcast(input_vec[N])
3
4
Dremel
Dremel Shard
GFS

Figure 7: Parallel Map

On a single computer, it would take weeks to generate historical forecasts at the scale of Google data. The forecast mapper uses the *googleparallelism* package to regenerate hundreds of thousands of historical forecasts in a matter of hours (100,000 forecasts would take less than two hours on 1,000 computers). Also, the output of forecast mapper contains millions of data.frame rows which makes the aggregation step very slow using standard R data manipulation (for 1,000 time series in the datastore and 100 different training end dates, the output of forecast mapper would have more than 20 million rows). Using the R dremel package, we can perform basic aggregations over this 20-million row data set in seconds. For example, we can easily compute Mean Absolute Percentage Error (MAPE) for different forecasting models and for a particular forecast horizon (like one-year-out forecasts) in only a few seconds.

6. Experimental Results

This section provides empirical results of the runtimes for the iterative forecast simulations. Table 1 shows the mean and 95th percentile runtimes for the five parallel jobs used to generate the results in Section 5. Each task generates one realization of traffic for the next 15 weeks (from the training end date) using iterative forecasts (explained in Section 5.4 and depicted in Figure 5).

These results demonstrate the motivation for some of the scheduling parameters described in Section 3.4. The long tail of straggler jobs is responsible for a disproportionate amount of the total runtime. On a large shared cluster the exact cause of the runtime differences could be due to workload differences, hardware capability differences, network congestion, or hardware failures. The effect is much more pronounced for longer running jobs and is one of the reasons that setting a deadline or scheduling duplicate tasks for the stragglers can help improve total runtime performance, as is suggested by Figure 8.

Table 1: Runtime characteristics of various parallel simulations.

Simulation Run	Startup Time (s)		Run Time (s)	
	Mean	95%	Mean	95%
1	48.3	88	249.4	266.1
2	40.8	67	260.7	283.1
3	43.3	74	312.6	343
4	37.4	61	283.5	304
5	32.3	44	249.5	264

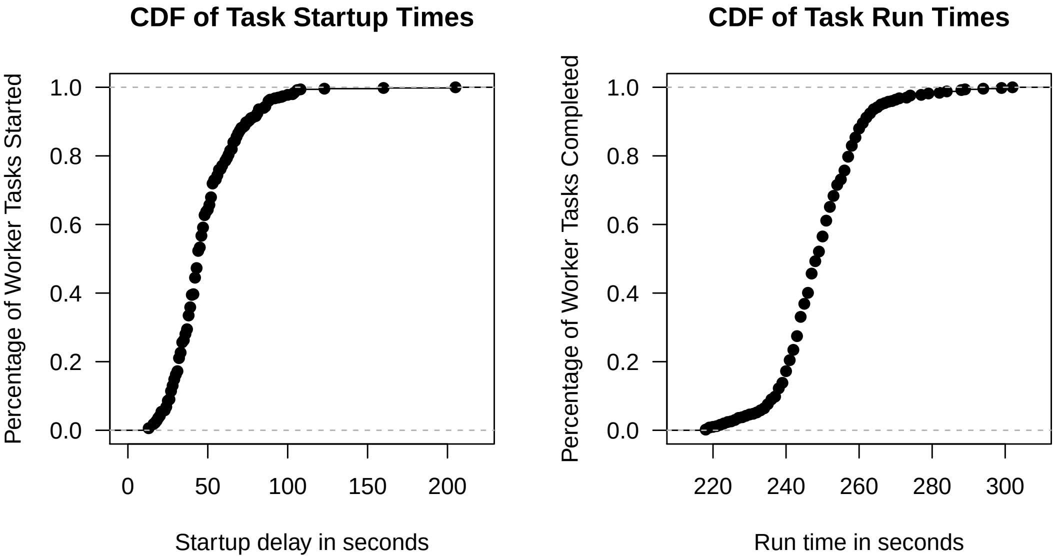


Figure 8: CDF of Worker Startup Time (left) and Run Time (right)

7. Conclusions

In addition to the traffic forecasting application described here, the *googleparallelism* R package has been applied to a variety of problems at Google requiring large-scale statistical analysis [6]. Since the initial development of the package, analyst teams have launched over 64,000 parallel statistical jobs using an average of 180 machines.

Importantly, this parallelism is available to analysts without any experience with Google's engineering infrastructure, dramatically expanding the set of people who can take advantage of the system—and allowing analysts to direct their creativity toward their problem-domain without worrying about the infrastructure.

References

[1] Hive. `https://cwiki.apache.org/confluence/display/Hive/Home`, 2011.

[2] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[3] J. Scott Armstrong. Combining forecasts. *Priniples of forecasting: A handbook for researchesr and practitioners*, pages 417–439.

[4] J. Scott Armstrong. Combining forecasts: The end of the beginning or the beginning of the end? *International Journal of Forecasting*, 5:585–588.

[5] Luiz André Barroso and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[6] David X. Chan, Yuan Yuan, Jim Koehler, and Deepak Kumar. Incremental clicks impact of search advertising. *Google Technical Report*.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 205–218, Nov. 2006.

[8] Biswapesh Chattopadhyay, Liang Lin, Weiran Liu, Sagar Mittal, Prathyusha Aragonda, Vera Lychagina, Yonghee Kwon, and Michael Wong. Tenzing: A sql implementation on the mapreduce framework. *Proc. VLDB Endow.*, September 2011.

[9] R.T. Clemen. Combining forecasts: A review and annotated bibliography. *International Journal of Forecasting*, 5:559–583.

[10] Duane Currie. *papply: Parallel apply function using MPI*, 2010. R package version 0.1-3.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[12] Bradley Efron. Better bootstrap confidence intervals. *Journal of the American Statistical Association*, 82(397):pp. 171–185, 1987.

[13] Bradley Efron. *An introduction to the bootstrap.* Chapman and Hall, New York, 1994.

[14] Robert Gentleman and Ross Ihaka. Lexical scope and statistical computing. *Journal of Computational and Graphical Statistics*, 9(3):pp. 491–508, 2000.

[15] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 29–43, Oct. 2003.

[16] Saptarshi Guha. *RHIPE: A Distributed Environment for the Analysis of Large and Complex Datasets*, 2010.

[17] Simon Haykin. Neural networks : a comprehensive foundation. *Upper Saddle River.*

[18] Jiangtian Li, Xiaosong Ma, Srikanth Yoginath, Guruprasad Kora, and Nagiza F. Samatova. Transparent runtime parallelization of the r scripting language. *J. Parallel Distrib. Comput.*, 71:157–168, February 2011.

[19] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3:330–339, September 2010.

[20] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[21] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2011. ISBN 3-900051-07-0.

[22] Markus Schmidberger, Martin Morgan, Dirk Eddelbuettel, Hao Yu, Luke Tierney, and Ulrich Mansmann. State of the art in parallel computing with r. *Journal of Statistical Software*, 31(1):1–27, 8 2009.

[23] Luke Tierney. A new serialization mechanism for r, 2003.

[24] Luke Tierney, A. J. Rossini, Na Li, and H. Sevcikova. *snow: Simple Network of Workstations*. R package version 0.3-3.

[25] Simon Urbanek. *multicore: Parallel processing of R code on machines with multiple cores or CPUs*, 2010. R package version 0.1-3.

[26] Hadley Wickham. The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, 40(1):1–29, 4 2011.