Streaming Big Data with Self-Adjusting Computation

Umut A. Acar
Carnegie Mellon University
umut@cs.cmu.edu

Yan Chen
Max Planck Institute for Software Systems
chenyan@mpi-sws.org

Abstract

Many big data computations involve processing data that changes incrementally or dynamically over time. Using existing techniques, such computations quickly become impractical. For example, computing the frequency of words in the first ten thousand paragraphs of a publicly available Wikipedia data set in a streaming fashion using MapReduce can take as much as a full day. In this paper, we propose an approach based on self-adjusting computation that can dramatically improve the efficiency of such computations. As an example, we can perform the aforementioned streaming computation in just a couple of minutes.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Self-adjusting computation; incremental MapReduce

1. Introduction

Due to pervasive use of computer systems to collect, communicate, and process data, techniques for computing with large amounts of data that do not fit into traditional databases, a.k.a. *big data*, has become increasingly important. Since such computations require relatively sophisticated systems support, approaches inspired by functional programming, where many different computations can be specified with the use of higher-order features, have become particularly attractive and seem promising. For example, MapReduce [Dean and Ghemawat], which is the most widely used system for big data, is inspired by the higher-order `map` and `reduce` functions commonly used in functional programming.

Essentially, any interesting big-data computation requires asymptotically linear time (or more) in the size of the data examined, because it examines each and every piece of data. This constitutes an important problem in computing with big data sets, because data sets change frequently over time, often as new data is accumulated. For example, when collecting information about real-time transactions (e.g., financial trades, email messages, phone calls, sensor data tracking), data sets grow monotonically over time as more transactions take place. Ideally, for computations on big data to remain consistent and relevant, the results must be updated every time the data changes. However, such *live* updates are difficult to perform in practice, because of their high computational complexity, which can be quadratic in the size of the data. Specifically, performing n linear-time updates on some data set of size n in a streaming fashion requires $\Theta(n^2)$ work (time using a single CPU), which is known not to scale beyond small n.

In this paper, we propose an approach to the problem of live updates that reduces the computational complexity, by a near-linear factor, $\Theta(n/\log n)$ more specifically, where n is the input size. Our proposal is to update computations incrementally by taking advantage of the similarity of data between successive runs. More precisely, let $D_0, D_1, D_2, \ldots$ be the data sets at time zero, one, two, etc. In many cases, the difference between D_{i-1} and D_i is small, because the latter is derived from the former by a small change. For example, when a new piece of data arrives from a server, we may expand our data set with that piece of data and update our computation.

One way to perform such updates is to design specific *dynamic algorithms* or *incremental algorithms* for each specific application of interest. Instead, we use self-adjusting computation, a general-purpose technique that applies to all functional programs, to make it possible to perform such updates automatically and efficiently. The idea behind self-adjusting computation [Acar et al. 2006, 2009] is to track the dependencies between the different parts of the computation so that when the computation data changes, the output can be updated by re-evaluating only the affected parts of the computation and re-building only the pieces of data affected by the change. While prior work on self-adjusting computation required making reasonably substantial changes to the program code, recent developments allow the types of a program to be annotated with just a few keywords and employ a type-directed translation algorithm to generate the necessary changes to the code automatically [Chen et al. 2011, 2012]. In this paper, we use this *implicit* approach.

In its more general setting, self-adjusting computation can be applied to any functional program.[1] Thus, it seems possible to incrementalize any big-data computation expressed as purely functional program. Our ultimate goal is to achieve exactly that, but in this paper, we restrict ourselves to the MapReduce paradigm. We implement a single-node (sequential, non-distributed) version of MapReduce in the implicit self-adjusting language that we proposed in a recent paper [Chen et al. 2012]. Our implementation allows us to read data from a file system into the memory and change the data incrementally, while performing automatic incremental updates after each change. Using our implementation, we report experimental results on a standard MapReduce benchmark, computing the frequency of words in a Wikipedia datasets, a big html file. We consider the case in which each successive line of the Wikipedia data, which roughly corresponds to a paragraph of text, arrives successively as a stream. In order to facilitate experimental evaluation, we consider the first 1M of the data. Using standard MapReduce, the streaming word-count computation takes 12.5 hours. Using the self-adjusting version of the MapReduce that we propose, the runtime is about 2 minutes, nearly 500 times faster.

DDFP'13, January 22, 2013, Rome, Italy.

[1] Self-adjusting computation is also applicable to imperative programs but for most data-driven computations, this does not seem beneficial.

```
signature MAPREDUCE =
sig
  type pair
  val mapper : α → pair
  val reducer : pair * (pair * pair → pair)

  val mapreduce : α list → pair list
end
```

Figure 1. Signature for MapReduce in SML

2. Self-Adjusting MapReduce

We implemented a proof-of-concept prototype to test the feasibility of incrementalizing large-scale data-driven applications. Specifically, we developed a MapReduce framework in Standard ML.

2.1 User Interface

Figure 1 shows the type signature for MapReduce in the Standard ML language. As is standard in MapReduce systems, the `MapReduce` module takes three arguments: the type of key-value pair, a mapper function and a reducer, which contains an initial key-value pair and a reduction function. The `mapreduce` function then takes an input list and applies the user defined mapper and reducer to produce the final key-value pair list.

As an example, consider computing the frequency of words in a text file. The type of key-value pair will be `string * int`. The mapper function converts string s to $(s, 1)$. The reducer will add the value in the input key-value pairs. With the mapper and reducer, we can read the text file into a list of strings, and execute the `mapreduce` function to compute word frequency.

2.2 Internals

We present a brief overview of self-adjusting computation and how we implement the signature in Figure 1 as a self-adjusting program that can respond to streaming live data automatically and efficiently.

2.2.1 Self-Adjusting Computation

Overview. The key concept behind self-adjusting computation is the notion of a *modifiable (reference)*, which stores *changeable* values that can change over time [Acar et al. 2006]. The programmer operates on modifiables with `mod`, `read`, and `write` primitive constructs to create, read from, and write into modifiables, respectively. The run-time system of a self-adjusting language uses these primitives to represent the execution as a dependency graph, enabling a *change propagation* algorithm that utilizes a particular form of memoization techniques [Acar et al. 2009, 2008].

For example, given a self-adjusting `map` function, we can run it in much the same way as running the conventional version. Such a complete run takes asymptotically as long as the conventional program but incurs some constant-factor overhead in practice. After a complete run, we can change any or all of the elements and update the output by performing a change propagation algorithm. As an example, consider inserting one cell into the input list and performing change propagation. This propagation will only trigger the application of the newly inserted cell, and all the rest of the results remain unaffected. Change propagation takes $\Theta(1)$ time to update the result, while a complete re-execution will require $\Theta(n)$ time. In writing the self-adjusting `map` function, we realized this efficiency without designing and implementing an incremental algorithm, but we nevertheless had to make significant changes to the code in order to use the primitives for modifiable references correctly.

Level Types. To help reduce the need for making pervasive changes to the code by explicitly using primitives for modifiable references, recent work proposed an implicit approach where such changes can be inferred from relatively straightforward type annotations [Chen et al. 2011]. We refer to the type annotations as *levels* and the resulting types as *level types*. With level types, the programmer only needs to identify changeable data, whose values can be changed, in the program, and mark their types with a `$C` level. All other types represent stable data, whose value cannot be changed. For example, the type `int $C` is a changeable integer. It indicates that its value can change across different executions. The type `int list $C` is a changeable list with stable integers. It indicates that the programmer can only insert and delete elements in the list, but is not allowed to mutate the content of the elements. Using level types, the programmer writes essentially "ordinary" Standard ML code, annotates the data types with levels as necessary, and the compiler type-checks the level annotations to make sure they are consistent, and generates self-adjusting executables automatically [Chen et al. 2012].

```
functor MapReduce
  (structure Pair : KEY_VALUE_PAIR
   val mapper : α → Pair.t
   val reducer : Pair.t * (Pair.t * Pair.t → Pair.t))
 : MAPREDUCE = struct

 type pair = Pair.t
 datatype α list $C =  Nil | Cons of α * α list $C

 val map : (α → pair) → α list $C → pair list $C
 val groupby : pair list $C → (pair list $C) list $C
 val reduce : pair * (pair * pair → pair) →
              pair list $C → pair $C

 fun mapreduce (input : α list $C) : pair $C list $C =
  let
   val pairs = map mapper input
   val merged = groupby pairs
  in
   map (fn block ⇒ reduce reducer block) merged
  end
end
```

Figure 2. MapReduce in SML with level type annotations

2.2.2 Implementation

Figure 2 shows our single-node MapReduce implementation in SML extended with level types [Chen et al. 2012]. Our `mapreduce` function contains three phases. First, the `map` function converts the input list into a list of key-value pairs. Second, the `groupby` function groups the pairs with the same key into one sub-list. We use merge sort and a scan to implement the groupby phase. Third, we apply the `reduce` function to each sub-list generated by the `groupby`. The `reduce` uses a divide-and-conquer algorithm, and assumes the reducer operation is associative. Overall, assuming the mapper and the reducer take constant time, the whole `mapreduce` function will require $\Theta(n \log n)$ time with input list of size n.

To make the MapReduce framework incremental, we need to put type annotations in the code. Since all our operations are based on list, in order to allow changes to the list, we annotate the list datatype as a changeable list with a polymorphic type. This list data type allows inserting and deleting cells. Whether the elements in the list are changeable or not depends on the instantiation of the polymorphic variable α. In the map phase, we usually pass in a changeable list with stable elements, because we need to be able to insert and delete input data. In the reduce phase, each sub-list returned by the `groupby` function is reduced to a changeable key-value pair, therefore we derive a changeable list of changeable

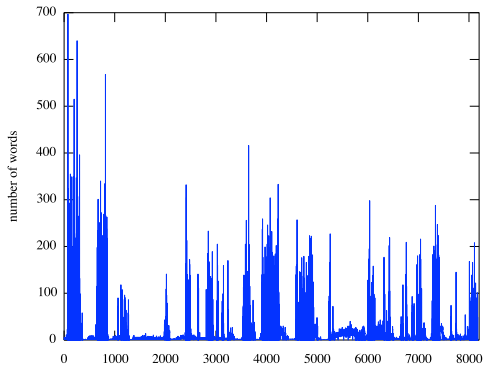

Figure 3. Number of words for each line (x-axis) in the dataset.

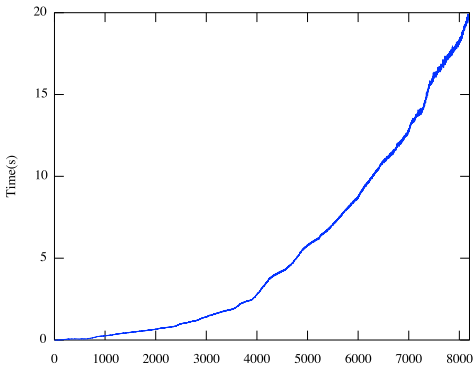

Figure 4. Update time per line (x-axis) for wordcount with non-self-adjusting ML implementation.

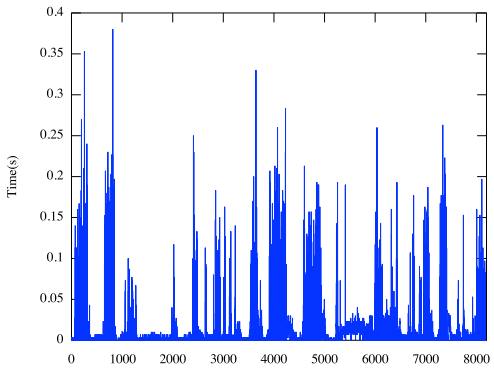

Figure 5. Update time per line (x-axis) with the self-adjusting `mapreduce`.

key-value pairs as the final return type. As a result, when we insert a new input cell, the result can be updated by changing the result of the reducer operating on the keys of the new cell, without requiring structural changes to the output. Using level types, we only annotate the type signature as shown in Figure 2 and make no further changes to the purely functional implementation of MapReduce. Our compiler [Chen et al. 2012] derives the self-adjusting executable.

To ensure efficient updates under dynamically changing data, our implementation uses stable algorithms [Acar 2005; Ley-Wild et al. 2009] to implement the `map`, `groupby`, and `reduce` functions. In a stable algorithm, changing a small fraction of the input causes only small changes to the dependency structure of the computation created by applying the algorithm to some input data. The classic (i.e., iterative) and the parallel implementations of `map` are both stable. For groupby, we use merge-sort, a divide-and-conquer sorting algorithm, followed by a simple scan, which are both stable. For `reduce`, we use the standard divide-and-conquer algorithm, which is stable. More generally, many algorithms, including parallel, divide-and-conquer, and iterative algorithms have been found to be stable (e.g., [Acar 2005; Ley-Wild et al. 2009]).

3. Evaluation

To evaluate the effectiveness of our approach, we consider the classic `wordcount` as the benchmark, which determines the frequency of words in a document.

We use a publicly available dataset with the contents of Wikipedia[2] as the input for `wordcount`. To ensure reasonable execution times, we consider a small prefix of the dataset, specifically 1M of data, consisting of 8173 lines, and 123026 words. The dataset and the prefix that we consider is an html file where each line corresponds roughly to a paragraph of text. Figure 3 shows the number of words on each line in the dataset. We take this dataset as a stream, and feed it into `wordcount` line by line. The benchmark updates the word frequencies after the arrival of each line.

For our measurements, we used a 2 GHz Intel Xeon with 64 GB memory, 0.25MB L2 cache, and 18MB L3 cache, and used the Linux operating system. The machine has multiple CPUs (32 cores partitioned between 4 nodes), but our benchmark is sequential. We compile our benchmark using the MLton [MLton] compiler for Standard ML extended to support self-adjusting computation with implicit type annotations.

As a baseline, we use two implementations of `wordcount`: the conventional, non-self-adjusting ML implementation of MapReduce, and the example `wordcount` code in Hadoop. Figure 4 shows the update time with the non-self-adjusting ML implementation after the arrival of each line in our dataset. As expected, the running time grows linearly as the input size increases, because `wordcount` has to re-process the whole data set after the arrival of each line. The slope of the running time remains constant, except a change around line 4000. The sudden slope change is due to the data filling up the fast cache. The whole computation takes 12.5 hours to finish.

To check the correctness and efficiency of our implementation, we also ran the same experiment with a single-node Hadoop, using the system-provided `wordcount` example program. For processing the whole dataset, Hadoop takes 31 seconds, while our ML implementation needs 20 seconds. Since Hadoop is designed as a batch processing system, it appears to have more overhead compared to our implementation. We therefore do not include the detailed update time for Hadoop in our evaluation.

Figure 5 shows the update time for each line using our self-adjusting `mapreduce`. The update time for the self-adjusting program ranges from 0 to 0.4 seconds depending on the number of words per line. Comparing the update time to the words-per-line distribution shown in Figure 3, we see that the update time is proportional to the number of words per update: the more words, the longer the update takes. This is in contrast to the non-self-adjusting implementation, where the update time is proportional to the size of the whole data, rather than the size of the "delta" change in the input. Performing the whole experiment with the self-adjusting ver-

[2] Wikipedia data-set: `http://wiki.dbpedia.org/`

sion takes 1.7 minutes compared to the 12.5 hours with the non-self-adjusting version. This is a 440-fold speedup.

This massive speedup comes at the cost of a significant, 45-fold increase in memory usage. Since self-adjusting `mapreduce` records the dependencies in the computation as a dynamic dependency graph, it takes 16GB of memory, while the ordinary ML version requires 350MB. This shows that in future work it will be important to reduce the memory consumption of the self-adjusting version.

4. Related Work

The problem of incremental computation, efficiently updating results of computation as the data that they depend on change over time, has been studied in several communities including in the algorithms, programming-languages, and software systems communities.

Algorithms and Programming Languages. In the algorithms community, researchers design *dynamic algorithms* that permit changes to their input and efficiently update their output when such changes occur. This vast research area (e.g., [Demetrescu et al. 2005]) shows that dynamic algorithms can be asymptotically more efficient than their conventional counterparts. Dynamic algorithms can, however, be difficult to develop and implement even for simple problems; some problems took years of research to solve and many remain open. In the programming languages community, researchers developed incremental computation techniques to achieve automatic incrementalization for a broad range of computations. The large body of research on incremental computation showed that it can be possible to achieve some degree of generality and efficiency (e.g. [Ramalingam and Reps 1993]). Recent advances on self-adjusting computation developed techniques that can achieve efficiency and full generality simultaneously (e.g., [Acar et al. 2006, 2009; Hammer et al. 2009]). Techniques for parallel self-adjusting computation have also been proposed [Hammer et al. 2007; Burckhardt et al. 2011].

Systems. There has also been recent interest in incremental big-data computations in the software systems community. Some systems such as Google's Percolator [Peng and Dabek] require the programmer to write a program in an event-driven programming model. Similarly, continuous bulk processing (CBP) [Logothetis et al.] proposes a new data-parallel programming model, which offers primitives to store and reuse prior state for incremental processing. Unfortunately, these approaches require the programmer to implement the incremental update algorithm, which essentially amounts to designing dynamic algorithms on a problem-specific basis. Other systems such as DryadInc [Popa et al.], Nectar [Gunda et al.], and Incoop [Bhatotia et al.] rely on caching of prior results to improve efficiency. Incoop additionally uses self-adjusting computations's stability notion to improve the response time by organizing MapReduce computations. Stream processing systems such as Comet [He et al.] and NOVA [Olston and et al 2011] provide techniques for re-using computations in a specified set of queries as the data arrives as part of a stream. None of these approaches use self-adjusting computation's change propagation technique which enables not just re-using previous computations but also actively pushing changes into computations that otherwise cannot be reused.

5. Conclusion and Future Work

We presented our preliminary results for investigating the use of self-adjusting computation for computing with dynamically changing live, big data. We implemented a single node, self-adjusting version of the MapReduce system using our approach based on implicit programming with type annotations, and presented experimental results that show that the self-adjusting version is about 500-fold faster than (non-self-adjusting) MapReduce. Our results show, however, that this speedup comes at the cost of increased memory usage, by about 50 fold. In future work, we plan to reduce this memory overhead by increasing the granularity of dependencies tracked by the underlying self-adjusting-computation system, and investigate generalizing our results to the distributed setting, as well as to programs beyond the MapReduce model.

References

U. A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.

U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Trans. Prog. Lang. Sys.*, 28(6):990–1034, 2006.

U. A. Acar, A. Ahmed, and M. Blume. Imperative self-adjusting computation. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*, 2008.

U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. *ACM Trans. Prog. Lang. Sys.*, 32(1):3:1–53, 2009.

P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: MapReduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 7:1–7:14.

S. Burckhardt, D. Leijen, C. Sadowski, J. Yi, and T. Ball. Two for the price of one: A model for parallel and incremental computation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2011.

Y. Chen, J. Dunfield, M. A. Hammer, and U. A. Acar. Implicit self-adjusting computation for purely functional programs. In *Int'l Conference on Functional Programming (ICFP '11)*, pages 129–141, Sept. 2011.

Y. Chen, J. Dunfield, and U. A. Acar. Type-directed automatic incrementalization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun 2012.

J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. 6th Symposium on Operating Systems Design and Implementation (OSDI'04)*.

C. Demetrescu, I. Finocchi, and G. Italiano. *Handbook on Data Structures and Applications*, chapter 36: Dynamic Graphs. CRC Press, 2005.

P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in data centers. In *Proc. 9th Symp. Operating Systems Design and Implementation (OSDI'10)*.

M. Hammer, U. A. Acar, M. Rajagopalan, and A. Ghuloum. A proposal for parallel self-adjusting computation. In *DAMP '07: Declarative Aspects of Multicore Programming*, 2007.

M. A. Hammer, U. A. Acar, and Y. Chen. CEAL: a C-based language for self-adjusting computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: batched stream processing for data intensive distributed computing. In *Proc. 1st Symposium on Cloud computing (SoCC'10)*.

R. Ley-Wild, U. A. Acar, and M. Fluet. A cost semantics for self-adjusting computation. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages*, 2009.

D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *1st Symp. on Cloud computing(SoCC'10)*.

MLton. MLton web site. `http://www.mlton.org`.

C. Olston and et al. Nova: continuous Pig/Hadoop workflows. In *Proceedings of the International Conference on Management of Data*, 2011.

D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proc. 9th Symposium on Operating Systems Design and Implementation (OSDI'10)*.

L. Popa, M. Budiu, Y. Yu, and M. Isard. DryadInc: Reusing work in large-scale computations. In *Worksh. on Hot Topics in Cloud Computing (HotCloud'09)*.

G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Principles of Programming Languages*, pages 502–510, 1993.