Scalable Language Processing Algorithms for the Masses: A Case Study in Computing Word Co-occurrence Matrices with MapReduce

Jimmy Lin
The iSchool, University of Maryland
National Center for Biotechnology Information, National Library of Medicine
`jimmylin@umd.edu`

Tech Report, HCIL-2008-28
Human-Computer Interaction Laboratory, University of Maryland
June 30, 2008

Abstract

This paper explores the challenge of scaling up language processing algorithms to increasingly large datasets. While cluster computing has been available in industrial environments for several years, academic researchers have fallen behind in their ability to work on large datasets. We discuss two challenges contributing to this problem: lack of a suitable programming model for managing concurrency and difficulty in obtaining access to hardware. Hadoop, an open-source implementation of Google's MapReduce framework, provides a compelling solution to both issues. Its simple programming model hides system-level details from the developer, and its ability to run on commodity hardware puts cluster computing within reach of many academic research groups. This paper illustrates these points with a case study on building word co-occurrence matrices from large corpora. We conclude with an analysis of an alternative computing model based on renting instead of buying computer clusters.

1 Introduction

Over the past couple of decades, the field of computational linguistics (and more broadly, human language technologies) has seen the emergence and later dominance of empirical techniques and data-driven research. Concomitant with this trend is a coherent research thread that focuses on exploiting increasingly-large datasets. Banko and Brill (2001) were among the first to demonstrate the importance of dataset size as a significant factor governing prediction accuracy in a supervised machine learning task. In fact, they argued that size of training set was perhaps more important than the choice of machine learning algorithm itself. Similarly, experiments in question answering have shown the effectiveness of simple pattern-matching techniques when applied to large quantities of data (Brill et al., 2001). More recently, this line of argumentation has been echoed in experiments with large-scale language models. Brants et al. (2007) showed that for statistical machine translation, a simple smoothing technique (dubbed *Stupid Backoff*) approaches the quality of the Kneser-Ney algorithm as the amount of training data increases, and with the simple method one can process significantly more data.

Challenges in scaling algorithms to increasingly-large datasets have become a serious issue for researchers. It is clear that datasets readily available today and the types of analyses that researchers wish to conduct have outgrown the capabilities of individual computers. The only practical recourse is to distribute the computation across multiple cores, processors, or machines. The consequences of failing to scale include misleading generalizations on artificially small datasets and ad-hoc approximations, both of which are undesirable.

This paper focuses on two barriers to developing scalable language processing algorithms: challenges associated with parallel programming and access to hardware. We argue that Google's MapReduce framework provides an attractive programming model for developing scalable algorithms, and with the release Hadoop, an open-source implementation of MapReduce, cost-effective cluster computing is within the reach of most academic research groups.

It is emphasized that this work focuses on large-data algorithms from the perspective of academia—our colleagues in industrial environments have long enjoyed the advantages of cluster computing. However, it is only recently that such capabilities have become *practical* for academic research groups. These points are illustrated by a case study in building large word co-occurrence matrices, a simple task that underlies many NLP algorithms.

We proceed as follows: the next section overviews the MapReduce framework and why it provides a compelling solution to the issues sketched above. Section 3 introduces task of building word co-occurrence matrices, which provides an illustrative case study: two separate algorithms are presented in Section 4. Our experimental setup is described in Section 5, followed by presentation of results in Section 6. We discuss implications and generalizations following that. Before concluding, we explore an alternative model of computing based on renting instead of buying hardware, which makes cluster computing practical for *everyone*.

2 MapReduce

The only practical solution to large-data challenges today is to distribute the computation across multiple cores, processors, or machines. The development of parallel algorithms involves a number of tradeoffs. First is that of cost: a decision must be made between "exotic" hardware (e.g., large shared memory machines, InfiniBand interconnect) and commodity hardware. There is significant evidence (Barroso et al., 2003) that solutions based on the latter are more cost effective—and for resource-constrained academic NLP groups, commodity hardware is the only practical route.

Given appropriate hardware, researchers must still contend with the challenge of developing software. Quite simply, parallel programming is difficult. Due to communication and synchronization issues, concurrent operations are notoriously challenging to reason about. Reliability and fault tolerance become important design considerations on clusters containing large numbers of commodity machines. With traditional parallel programming models (e.g., MPI), the developer shoulders the burden of explicitly managing concurrency. As a result, a significant amount of the programmer's attention is devoted to system-level details, leaving less time for thinking about the actual problem.

Recently, Google's MapReduce framework (Dean and Ghemawat, 2004) has emerged as an attractive alternative to existing parallel programming models. The MapReduce abstraction shields the programmer from having to explicitly worry about system-level issues such as synchronization, inter-process communication, and fault tolerance. The runtime is able to transparently distribute computations across large clusters of commodity hardware with good scaling characteristics. This frees the programmer to focus on actually solving the problem at hand.

MapReduce builds on the observation that many information processing tasks have the same basic structure: a computation is applied over a large number of records (e.g., Web pages, bitext pairs, or nodes in a graph) to generate partial results, which are then aggregated in some fashion. Naturally, the per-record computation and aggregation function vary according to task, but the basic structure remains fixed. Taking inspiration from higher-order functions in functional programming, MapReduce provides an abstraction at the point of these two operations. Specifically, the programmer defines a "mapper" and a "reducer" with the following signatures:

map: $(k_1, v_1) \rightarrow [(k_2, v_2)]$
reduce: $(k_2, [v_2]) \rightarrow [(k_3, v_3)]$

Key-value pairs form the basic data structure in MapReduce. The mapper is applied to every input key-value pair to generate an arbitrary number of intermediate key-value pairs (we adopt the convention of $[\ldots]$ to denote a list). The reducer is applied to all values associated with the same intermediate key to generate output key-value pairs. This two-stage processing structure is illustrated in Figure 1.

Under the framework, a programmer need only provide implementations of the mapper and reducer. On top of a distributed file system (Ghemawat et al., 2003), the runtime transparently handles all other aspects of execution, on clusters ranging from a few to a few thousand nodes. The runtime is responsible for scheduling map and reduce workers on commodity hardware assumed to be unreliable, and thus is tolerant to various faults through a number of error recovery mechanisms. The runtime also

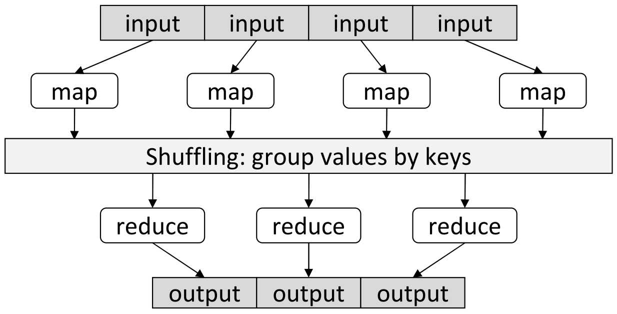

Figure 1: Illustration of the MapReduce framework: the "mapper" is applied to all input records, which generates results that are aggregated by the "reducer". The runtime groups together values by keys.

manages data distribution, including splitting the input across multiple map workers and the potentially very large sorting problem between the map and reduce phases whereby intermediate key-value pairs must be grouped by key.

As an optimization, the MapReduce model allows the use of "combiners", which are similar to reducers except that they operate directly on the output of mappers (in memory, before intermediate output is written to disk). Combiners operate in isolation on each node in the cluster and cannot use partial results from another node. Since the output of mappers (i.e., the key-value pairs) must ultimately be shuffled to the appropriate reducer over a network, combiners allow a programmer to aggregate partial results, thus reducing network traffic. In cases where an operation is both associative and commutative, reducers can directly serve as combiners.

Google's proprietary implementation of MapReduce is in C++ and not available to the public. However, the existence of Hadoop, an open-source implementation in Java, allows any programmer to take advantage of MapReduce. The growing popularity of this technology has stimulated a flurry of recent work, on applications in machine learning (Chu et al., 2006), machine translation (Dyer et al., 2008), and document retrieval (Elsayed et al., 2008).

3 Word Co-occurrence Matrices

To illustrate the arguments presented above, we present a case study using MapReduce to build word co-occurrence matrices from large corpora, a common task in natural language processing. Formally, the co-occurrence matrix of a corpus is a square $N \times N$ matrix where N corresponds to the number of unique words in the corpus. A cell m_{ij} contains the number of times word w_i co-occurs with word w_j within a certain window of m words (usually an application-dependent parameter). Note that the upper and lower triangles of the matrix are identical since co-occurrence is a symmetric relation.

This task is quite common in corpus linguistics and provides the starting point to many other algorithms, e.g., for computing statistics such as pointwise mutual information (Church and Hanks, 1990), for unsupervised sense clustering (Schütze, 1998), and more generally, a large body of work in lexical semantics based on distributional profiles, dating back to Firth (1957) and Harris (1968). The task also has applications in information retrieval, e.g., (Schütze and Pedersen, 1998; Xu and Croft, 1998), and other related fields as well. More generally, this problem relates to the task of estimating distributions of discrete events from a large number of observations (more on this in Section 7).

It is obvious that the space requirement for this problem is $O(N^2)$, where N is the size of the vocabulary, which for real-world English corpora can be hundreds of thousands of words. The computation of the word co-occurrence matrix is quite simple if the entire matrix fits into memory—however, in the case where the matrix is too big to fit in memory, a naive implementation can be very slow as memory is paged to disk. For large corpora, one needs to optimize disk access and avoid costly seeks. As we illustrate in the next section, MapReduce handles exactly these issues transparently, allowing the programmer to express the algorithm in a straightforward manner.

Before moving on, we note that in many applications building the complete word co-occurrence matrix may not actually be necessary. For example, Schütze (1998) discusses feature selection techniques in defining context vectors; Mohammad and Hirst (2006) present evidence that conceptual distance is better captured via distributional profiles mediated by thesaurus categories. These objections, however, miss the point—the focus of this paper is on practical cluster computing for academic researchers; this particular task serves merely as an illustrative example.

```
1: procedure $\text{MAP}_1(n, d)$
2:     for all $w \in d$ do
3:         for all $u \in \text{NEIGHBORS}(w)$ do
4:             $\text{EMIT}((w, u), 1)$
```

```
1: procedure $\text{REDUCE}_1(p, [v_1, v_2, \ldots])$
2:     for all $v \in [v_1, v_2, \ldots]$ do
3:         $sum \leftarrow sum + v$
4:     $\text{EMIT}(p, sum)$
```

Figure 2: Algorithm 1 ("pairs") for computing word co-occurrence matrices.

4 MapReduce Implementation

We present two MapReduce algorithms for building word co-occurrence matrices for large corpora. This section demonstrates how the problem can be concisely captured in the MapReduce programming model, and how the runtime hides many of the system-level details associated with distributed computing. Pseudo-code for the first, more straightforward, algorithm is shown in Figure 2. Unique document ids and the corresponding texts make up the input key-value pairs. The mapper takes each input document and emits intermediate key-value pairs with each co-occurring word pair as the key and the integer one as the value. In the pseudo-code, EMIT denotes the generation of a key-value pair that is collected (and appropriately sorted) by the MapReduce runtime. The reducer simply sums up all the values associated with the same co-occurring word pair, arriving at the absolute counts of the joint event in the corpus (corresponding to each cell in the co-occurrence matrix).

For convenience, we refer to this algorithm as the "pairs" approach. Note that since co-occurrence is a symmetric relation, we actually only need to compute half of the matrix—however, for conceptual clarity in presenting the algorithms and to generalize to instances where the relation may not be symmetrical, our algorithms compute the entire matrix.

The Java implementation of this algorithm is quite concise—less than fifty lines long. Notice the MapReduce runtime guarantees that all values associated with the same key will be gathered together at the reduce stage. Thus, the programmer does not need to explicitly manage the collection and distribution of partial results across a potentially large cluster. In addition, the programmer does not need to explicitly partition the input data and schedule workers. This example shows the extent to which distributed processing can be dominated by system issues, and how an appropriate abstraction can significantly simplify a solution.

```
1: procedure $\text{MAP}_2(n, d)$
2:     $\text{INITIALIZE}(H)$
3:     for all $w \in d$ do
4:         for all $u \in \text{NEIGHBORS}(w)$ do
5:             $H\{u\} \leftarrow H\{u\} + 1$
6:     $\text{EMIT}(w, H)$
```

```
1: procedure $\text{REDUCE}_2(w, [H_1, H_2, H_3, \ldots])$
2:     $\text{INITIALIZE}(H_f)$
3:     for all $H \in [H_1, H_2, H_3, \ldots]$ do
4:         $\text{MERGE}(H_f, H)$
5:     $\text{EMIT}(w, H_f)$
```

Figure 3: Algorithm 2 ("stripes") for computing word co-occurrence matrices.

It is immediately obvious that Algorithm 1 generates an immense number of key-value pairs. Although this can be mitigated with the use of a combiner (since addition is commutative and associative), the approach still results in a large amount of network traffic. An alternative approach is presented in Figure 3. The only difference is that we first store in an associative array (H) the counts of all the words that co-occur with a particular word. The output of the mapper is a number of key-value pairs with words as keys and the corresponding associative arrays (properly serialized) as the values. In the reducer, counts corresponding to the words are summed across all the associative arrays (denoted by the function MERGE). Once again, a combiner can be used to cut down on the network traffic by merging partial results. In the final output, each key-value pair corresponds to a row in the word co-occurrence matrix. For convenience, we refer to this as the "stripes" approach; this is similar to the technique recently introduced by Dyer et al. (2008).

Compared to the "pairs" approach, the "stripes" approach results in far fewer intermediate key-value pairs, although each is significantly larger (and there is overhead in serializing and deserializing associative arrays). A critical assumption of the "stripes" approach is that at any point in time, each associative array is small enough to fit into memory. In our case, this does turn out to be true, since the size of the associative array is bounded by the vocabulary size (on the order of hundreds of thousands of words). In Section 6, we compare the efficiency of both algorithms.

5 Experimental Setup

Our experiments used the English Gigaword corpus (version 3),[1] which consists of newswire documents from six separate sources, totally 7.15 million documents (6.8 GB compressed, 19.4 GB uncompressed). For some experiments we used only the documents from the Associated Press Worldstream (APW), which contains 2.27 million documents (1.8 GB compressed, 5.7 GB uncompressed). By LDC's count, the entire collection contains approximately 2.97 billion words.

Prior to working with Hadoop, we preprocessed the collection. All XML markup was removed, followed by tokenization and stopword removal using standard tools from the Lucene search engine. All tokens were replaced with unique integers for a more efficient encoding. The data was then packed into a Hadoop-specific binary file format. The entire Gigaword corpus took up 4.69 GB in this format; the APW sub-corpus, 1.32 GB.

For our experiments, we used Hadoop version 0.16.0, running on a 20-machine cluster (1 master, 19 slaves). Each machine has two single-core processors (running at either 2.4 GHz or 2.8 GHz), 4 GB memory (map and reduce tasks were limited to 768 MB). The cluster has an aggregate storage capacity of 1.7 TB. We report experimental results using this cluster, but discuss an alternative model of computing based on "renting cycles" in Section 8.

6 Results

First, we compared the running time of the "pairs" and "stripes" approaches discussed in Section 4. Running times on our cluster are shown in Figure 4 for the APW section of the Gigaword corpus: on the *x*-axis we plot different percentages of the sub-corpus, and on the *y*-axis running time in seconds is shown. For these experiments, the co-occurrence

[1]LDC catalog number LDC2007T07

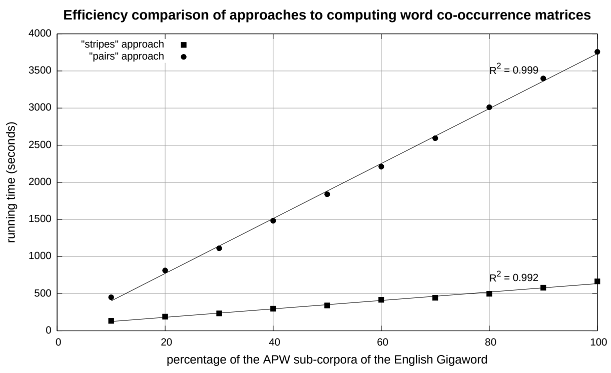

Figure 4: Running time of the two algorithms ("stripes" vs. "pairs") on the APW section of the Gigaword corpus.

window was set to two, i.e., w_i is said to co-occur w_j if they are no more than two words apart.

Results demonstrate that the stripes approach is far more efficient than the pairs approach: 666 seconds (11m 6s) compared to 3758 seconds (62m 38s) for the entire APW sub-corpus (improvement by a factor of 5.66). On the entire sub-corpus, the mappers in the pairs approach generated 2.6 billion intermediate key-value pairs totally 31.2 GB. After the combiners, this was reduced to 1.1 billion key-value pairs, which roughly quantifies the amount of data involved in the shuffling and sorting of the keys. On the other hand, the mappers in the stripes approach generated 653 million intermediate key-value pairs totally 48.1 GB; after the combiners, only 28.8 million key-value pairs were left. Although associative arrays can be less space efficient, the stripes approach provides more opportunities for combiners to aggregate intermediate results, thus greatly reducing network traffic in the sort and shuffle phase.

Figure 4 also shows that both algorithms exhibit highly desirable scaling characteristics—linear in the corpus size. This is confirmed by a linear regression applied to the running time data, which yields R^2 values close to one. Given that we have empirically shown the stripes algorithm to be more efficient than the pairs algorithm, we use the stripes approach for the remainder of our experiments.

With a window size of two, computing the word co-occurrence matrix for the complete Gigaword corpus takes 37m 11s on our cluster. Figure 5 shows the running time as a function of window size. With a window of six words, running time on the com-

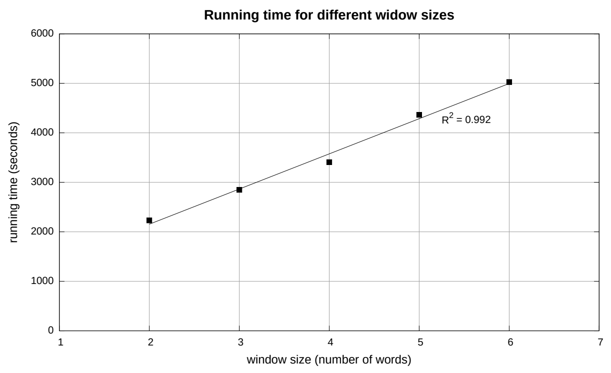

Figure 5: Running time on the entire Gigaword corpus, with varying window sizes.

plete Gigaword corpus is 1h 23m 45s. Once again, the algorithm exhibits the highly desirable characteristic of linear scaling in terms of window size, as confirmed by the linear regression with an R^2 value very close to one.

7 Discussion

The simplicity of the programming model and good scaling characteristics of resulting implementations make MapReduce a compelling tool for a variety of text processing tasks. In fact, MapReduce excels at a large class of problems in computational linguistics that involves estimating probability distributions of discrete events from a large number of observations according to the maximum likelihood criterion:

$$P_{MLE}(B|A) = \frac{c(A,B)}{c(A)} = \frac{c(A,B)}{\sum_{B'} c(A,B')} \quad (1)$$

In practice, it matters little whether these events are words, syntactic categories, word alignment links, or any construct of interest to researchers. Absolute counts in the "stripes" algorithm presented in Section 4 can be easily converted into conditional probabilities by final normalization step. Recently, Dyer et al. (2008) used similar techniques for phrase extraction and word alignment in statistical machine translation. Of course, many applications require smoothing of the estimated distributions—but this problem also has a nice solution in MapReduce (Brants et al., 2007).

Synchronization is perhaps the single largest bottleneck in distributed computing. In MapReduce, this is handled in the shuffling and sorting of key-value pairs between the map and reduce phases. Development of efficient MapReduce algorithms critically depends on careful control of intermediate output. Since the network link between different nodes in a cluster is by far the component with the largest latency, any reduction in the size of intermediate output or a reduction in the number of key-value pairs will have significant impact on efficiency.

8 Computing on Demand

The central theme of this paper is practical cluster computing for NLP researchers in the academic environment. We have identified two key aspects of what it means to be "practical": the first is an appropriate programming model for simplifying concurrency management; the second is access to hardware resources. The Hadoop implementation of MapReduce addresses the first point and to a large extent the second point as well. We note that the cluster used in the Section 6 experiments is modest by today's standards and within the capabilities of many academic research groups. It is not even a requirement for machines to be rack-mounted units in a machine room (although that is clearly preferable); there are plenty of descriptions on the Web about Hadoop clusters built from a handful of desktop machines connected by gigabit Ethernet.

Even without access to hardware, cluster computing remains within the reach of resource-constrained academics. "Utility computing" is an emerging concept whereby researchers could provision clusters on demand from a third-party provider. Instead of upfront capital investment to acquire a cluster and reoccurring maintenance and administration costs, researchers could "rent" computing cycles as they are needed. One such service is provided by Amazon, called Elastic Compute Cloud (EC2)[2]; with EC2, researchers could dynamically create a Hadoop cluster on-the-fly and tear down the cluster once experiments are complete. To demonstrate the use of this technology, we replicated our experiments on EC2 to provide a case study on this emerging model of computing.

Virtualized computation units in EC2 are called instances; the basic instance offers, according to

[2]http://www.amazon.com/ec2

Amazon, 1.7 GB of memory, 1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit), and 160 GB of instance storage. As of this writing, the current pricing is \$0.10 (all prices given in USD) per instance-hour. Computational resources are simply charged by the instance-hour, so that a ten-instance cluster for ten hours costs the same as a hundred-instance cluster for one hour (both \$10)—the Amazon infrastructure allows one to dynamically provision and release resources as necessary. This is attractive for researchers, who could on a limited basis allocate clusters much larger than they could otherwise afford if forced to purchase the hardware outright. Through virtualization technology, Amazon is able to parcel out allotments of processor cycles while maintaining high overall utilization across a data center and exploiting economies of scale.

Using EC2, we built word co-occurrence matrices from the entire English Gigaword corpus (window of two) on clusters of various sizes, ranging from 20 slave instances all the way up to 80 slave instances. The entire cluster consists of the slave instances plus a master controller instance that serves as the job submission queue; our clusters ran Hadoop version 0.17.0 (as of this writing, the latest release). The running times for our experiments are shown in Figure 6 (solid squares), with varying cluster sizes on the *x*-axis. Each data point is annotated with the cost of running the complete experiment.[3] We see that computing the complete word co-occurrence matrix costs, quite literally, a couple of dollars—certainly affordable by any academic researcher without access to hardware.

The alternate set of axes in Figure 6 shows the scaling characteristics of various cluster sizes. The circles plot the relative size and speedup of the EC2 experiments, with respect to the 20-slave cluster. We see highly desirable linear scaling characteristics, at least for the sizes of clusters we explored.

The above figures include only the cost of running the instances. One must additionally pay for bandwidth when transferring data in and out of of EC2. As of this writing Amazon charges \$0.10 per GB for data transferred in and \$0.17 per GB for data transferred out. To complement EC2, Amazon offers per-

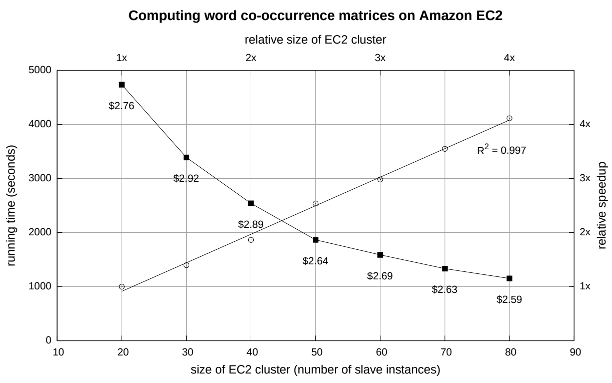

Figure 6: Running time analysis on Amazon EC2 with various cluster sizes; solid squares are annotated with the cost of each experiment. Alternate axes (circles) plot scaling characteristics in terms increasing cluster size.

sistent storage via the Simple Storage Service (S3),[4] at a cost of \$0.15 per GB per month. Amazon does not charge for transfers between EC2 and S3 to encourage use of the storage. The availability of this service means that one can choose between paying for data transfer or paying for persistent storage on a cyclic basis—the tradeoff naturally depends on the amount of data and its permanence.

The cost analysis presented above assumes optimally-efficient use of Amazon's services; end-to-end cost might better quantify real-world usage conditions. In total, the experiments reported in this section resulted in a bill of approximately thirty dollars. The figure includes all costs associated with instance usage and transfer costs. It also includes time taken to learn the Amazon tools (we previously had no experience with either EC2 or S3) and to run preliminary experiments on smaller datasets (before running on the complete English Gigaword corpus). The lack of fractional accounting on instance-hours contributed to the larger-than-expected costs, but such wastage would naturally be reduced with more experiments and higher sustained use. Overall, these cost appear to be very reasonable, considering that the largest cluster in our experiments (1 master + 80 slave instances) might be too expensive for most academic research groups to own and maintain.

Consider another example that illustrates the possibilities of a service like EC2. Brants et al. (2007) describe experiments on building language models

[3]Note that Amazon in actuality bills in whole instance-hour increments; these figures assume fractional accounting.

[4]http://www.amazon.com/s3

with increasingly-large corpora using MapReduce. Their paper reported experiments on a corpus containing 31 billion tokens (about an order of magnitude larger than the English Gigaword): on 400 machines, the model estimation took 8 hours.[5] With EC2, such an experiment would cost a few hundred dollars—sufficiently affordable that availability of data becomes the limiting factor, not computational resources themselves.

The availability of "computing-on-demand" services such as EC2 and S3, coupled with Hadoop, make cluster computing practical for academic researchers. Although Amazon is currently the most prominent provider of such services, they are not the sole player in an emerging market—in the future we foresee a vibrant field with many competing providers. Considering the tradeoffs between "buying" and "renting", we would recommend the following model for an academic research group: to purchase a modest cluster for development and for running smaller experiments, and to use a computing-on-demand service for scaling out and for running larger experiments (since it would be more difficult to economically justify a large cluster if it does not receive high sustained utilization).

Finally, we note that if the concept of utility computing takes hold, it would have a significant impact on computer science research in general: algorithms would not only be analyzed in traditional terms such as asymptotic complexity, but also in terms of the approximate cost for running experiments on different datasets and clusters of different sizes. The case can be made that cost is a more direct and practical measure of algorithmic efficiency.

9 Conclusion

This paper address two challenges faced by academic research groups in scaling up text processing algorithms to large corpora: the lack of an appropriate programming model for expressing the problem and the difficulty in getting access to hardware. With our case study in building word co-occurrence matrices from large corpora, we demonstrate that MapReduce, via the open source Hadoop implementation, provides a compelling solution. A large class of algorithms in computational linguistics can be readily expressed in MapReduce, and the resulting code can be transparently distributed across commodity clusters. Finally, the "cycle-renting" model of computing makes access to large clusters affordable to researchers with limited resources. Together, these developments dramatically lower the entry barrier for academic researchers who wish to tackle large-data issues.

[5] Brants et al. were affiliated with Google, so access to hardware was not an issue.

Acknowledgments

This work was supported by the Intramural Research Program of the NIH, National Library of Medicine. I would like to thank IBM and Google for hardware support via the Academic Cloud Computing Initiative, as well as Amazon support for EC2 and S3. I am, as always, grateful to Esther and Kiri for their kind support.

References

Michele Banko and Eric Brill. 2001. Scaling to very very large corpora for natural language disambiguation. In *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics (ACL 2001)*, pages 26–33, Toulouse, France.

Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. 2003. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28.

Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, and Jeffrey Dean. 2007. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 858–867, Prague, Czech Republic.

Eric Brill, Jimmy Lin, Michele Banko, Susan Dumais, and Andrew Ng. 2001. Data-intensive question answering. In *Proceedings of the Tenth Text REtrieval Conference (TREC 2001)*, pages 393–400, Gaithersburg, Maryland.

Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Ng, and Kunle Olukotun. 2006. Map-Reduce for machine learning on multicore. In *Advances in Neural Information Processing Systems 19 (NIPS 2006)*, pages 281–288, Vancouver, British Columbia, Canada.

Kenneth W. Church and Patrick Hanks. 1990. Word association norms, mutual information, and lexicography. *Computational Linguistics*, 16(1):22–29.

Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004)*, pages 137–150, San Francisco, California.

Chris Dyer, Aaron Cordova, Alex Mont, and Jimmy Lin. 2008. Fast, easy, and cheap: Construction of statistical machine translation models with MapReduce. In *Proceedings of the Third Workshop on Statistical Machine Translation at ACL 2008*, pages 199–207, Columbus, Ohio.

Tamer Elsayed, Jimmy Lin, and Douglas Oard. 2008. Pairwise document similarity in large collections with MapReduce. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics (ACL 2008), Companion Volume*, pages 265–268, Columbus, Ohio.

John R. Firth. 1957. A synopsis of linguistic theory 1930–55. In *Studies in Linguistic Analysis, Special Volume of the Philological Society*, pages 1–32. Blackwell, Oxford.

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP-03)*, pages 29–43, Bolton Landing, New York.

Zelig S. Harris. 1968. *Mathematical Structures of Language*. Wiley, New York.

Saif Mohammad and Graeme Hirst. 2006. Distributional measures of concept-distance: A task-oriented evaluation. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing (EMNLP 2006)*, pages 35–43, Sydney, Australia.

Hinrich Schütze and Jan O. Pedersen. 1998. A cooccurrence-based thesaurus and two applications to information retrieval. *Information Processing and Management*, 33(3):307–318.

Hinrich Schütze. 1998. Automatic word sense discrimination. *Computational Linguistics*, 24(1):97–123.

Jinxi Xu and W. Bruce Croft. 1998. Corpus-based stemming using cooccurrence of word variants. *ACM Transactions on Information Systems*, 16(1):61–81.