Parallelizing large-scale data processing applications with data skew: a case study in product-offer matching

Ekaterina Gonina*
University of California, Berkeley
Berkeley, CA
egonina@eecs.berkeley.edu

Anitha Kannan, John Shafer,
Mihai Budiu
Microsoft Research, Mountain View, CA
ankannan, jshafer,
mbudiu@microsoft.com

ABSTRACT

The last decade has seen a surge of interest in large-scale data-parallel processing engines. While these engines share many features in common with parallel databases, they make a set of different trade-offs. In consequence many of the lessons learned for programming parallel databases have to be re-learned in the new environment. In this paper we show a case study of parallelizing an example large-scale application (offer matching, a core part of online shopping) on an example MapReduce-based distributed computation engine (DryadLINQ). We focus on the challenges raised by the nature of large data sets and data skew and show how they can be addressed effectively within this computation framework by optimizing the computation to adapt to the nature of the data. In particular we describe three different strategies for performing distributed joins and show how the platform language allows us to implement optimization strategies at the application level, without system support. We show that this flexibility in the programming model allows for a highly effective system, providing a measured speedup of more than 100 on 64 machines (256 cores), and an estimated speedup of 200 on 1280 machines (5120 cores)of matching 4 million offers.

1. INTRODUCTION

Recent exponential growth in internet data and the commoditization of parallel and cluster computing has enabled development of large-scale data-parallel processing engines [4, 6, 8, 14, 1, 12, 26, 21, 5, 22] and large-scale data processing applications [2, 27]. Efficient parallelization of these applications presents many challenges to programmers. The data sets for web-scale applications are very large and often skewed. For example [2] addressed the data skew problem in computing cube materialization of large web-scale datasets, and [27] faced the data skew problem in large-scale botnet detection. The database community has also focused on automatically addressing the persistent data skew problem in joining large datasets in work such as [9, 25, 23] and [13]. While we can borrow some mechanisms and techniques from parallel databases, many techniques need to be reworked to be efficient in the new large-scale web application setting. In this paper, we investigate mechanisms for overcoming these challenges using a MapReduce framework (using DryadLINQ [26] distributed computation engine as an example) to efficiently parallelize an example web-scale data processing application of offer matching - matching product offers from online merchants to products in a catalog owned by a search engine. We show that this flexibility in the MapReduce programming model allows for high speedups of general large-scale web applications with significant data skew.

A comprehensive product catalog is a pre-requisite for an effective e-commerce search engine. Such a catalog contains structured descriptions of products in a form of attribute ⟨name, value⟩ pairs. To maintain the product catalog, the search engine need to match product offers (textual product descriptions) that it receives from online merchants to the structured records in the product catalog. Matching offers to products is a problem of complexity and scale. The product information is obtained from multiple product aggregators (*e.g.,* CNET, PriceGrabber). Hence, they are typically quite rich, with products being represented by dozens of attribute-value pairs such as the weight of a laptop computer or the screen size of a TV. The offer information comes from many more merchants (*e.g.,* buy.com, gadgettown.com) and is very poor in comparison, often consisting only of short textual descriptions of the product being sold. In addition, there is also the issue of scale that derives from the significantly large number of products and offers that comprise a typical e-commerce catalog. It is not unusual for an aggregator to receive tens of millions of offers every day that need to be matched against a catalog of several million products. Manually creating rule systems to this scale is outright impossible. Furthermore, the data of the offers and products is very skewed across categories - with some categories having millions of offers and products, while others only having a couple.

In this paper we describe a parallel implementation of the offer matching algorithm on a cluster of multi-core processors using the DryadLINQ distributed computing engine as an example data-parallel programming framework. We focus on efficiently handling the challenges raised by the nature

*work done while interning at Microsoft Research

Structured Record (Product)

Attribute Name	Attribute Value
category	digital camera
brand	Panasonic
product line	Panasonic Lumix
model	DMC-FX07
sensor resolution	7 megapixel
color	silver
weight	132 g
width	9.4 cm
height	5.1 cm
depth	2.4 cm
display: type	LCD display
display: technology	TFT active matrix
display: diagonal size	2.5 in
audio input type	none
flash memory: form factor	memory stick
flash memory: storage capacity	8 MB
video input: still image format	JPEG
video input: digital video format	MPEG-1
lens system: optical zoom	3.6
...	

Unstructured Text (Offer-1)

Panasonic Lumix DMC-FX07 digital camera [7.2 megapixel, 2.5", 3.6x optical zoom, LCD monitor]

Unstructured Text (Offer-2)

Panasonic DMC-FX07EB digital camera silver

Unstructured Text (Offer-3)

Lumix FX07EB-S, 7.2 MP

Figure 1: Structured product record for 'Panasonic DMC-FX07 digital camera' and textual descriptions from three matching offers.

of large data sets and data skew and show how they can be addressed effectively within the computation framework by optimizing the computation to adapt to the nature of the data. In particular we describe three different strategies for performing distributed joins and show how they can all be implemented at the application level, without system support. We also describe a dynamic data-dependent repartitioning strategy to handle data skew that allows for better platform utilization and load balanced execution. Finally, we show how nested parallelism can be leveraged to fully exploit the potential of a multi-core system. We show that by allowing for this flexibility in the programming framework, we can create a scalable matching system capable of yielding over 100× speedup on millions of offers, taking matching time from days down to minutes.

2. OFFER MATCHING

With the growth of Internet usage, how people shop for goods has fundamentally changed. According to a recent Nielsen study [20], over 77% of the U.S. population is now online and 80% expected to make an online purchase in the next six months. U.S. online retail spending surpassed $140B in 2010 growing by 10% since 2009 [11] and according to Forrester [10] is on track to surpass $250B by 2014. 80% of online purchases start with search [10], representing a key gateway into this huge market. Indeed, most major search engines (and many e-commerce aggregators such as PriceGrabber.com and Epinions.com) target this lucrative traffic by providing e-commerce search services that allow users to sift through extensive catalogs of product information and compare sales offers from various merchants selling those products. The success of these services is directly tied to the quality and comprehensiveness of this catalog. As search results are typically represented by products, not offers, it is critical that offers be matched correctly to the corresponding products in order for users to see them. Products are often the primary, if not the only, entry point into revenue-generating offers, and every unmatched or mismatched offer represents lost revenue potential.

The product information gathered from online merchants consists of various attributes and their corresponding values, stored in a structured record comprised of attribute ⟨name, value⟩ pairs. Similarly, offers come from multiple merchants and generally have very little structure. Fig. 1 shows part of the structured record for Panasonic DMC-FX07 digital camera as well as three merchant offers for this product. The figure illustrates the complexity of offer-to-product matching. Product offers differ in detail and the attribute information they provide. This impedance mismatch between offers and products is difficult to tackle algorithmically, and the industry often resorts to manually-created rule sets that are difficult to maintain and brittle in execution. However, there has been work that leverages the *implicit* structure in offer descriptions to perform this matching algorithmically [16]. The matching is robust in regards to data richness, and it avoids domain-specific features that, in part, make rule systems difficult to maintain. In this paper we implement a scalable parallel version of the algorithm presented in [16]. We first describe the offer matching algorithm in the following Sections 2.1 - 2.3 and then describe the details of the parallel implementation in Section 3, followed by performance results in Section 4.

2.1 Offer matching algorithm overview

The offer-matching algorithm consists of two phases - (1) offline training phase where the matching functions are learned, and (2) the online matching phase where the offers are matched to products using the matching functions learned in the offline phase. We describe these two phases below:

Offline training phase: Figure 2 schematically shows the offline training phase. In this phase, we first obtain a training set of offers O and products P from merchants and online aggregators respectively. The offers $o \in O$ are parsed and annotated with product attributes and then paired with products $p \in P$ in the *Label* phase. Then the model learning algorithm (*Match & Learn Models* phase) receives a small labeled training set of unstructured offers $\delta(O)$ and matching products $\delta(P)$ and also mismatched products from $\delta(P)$, one for every $o \in O$ as a set of negative examples. The similarity feature vectors are computed to be used as features for matching o to p. Along with their label (matched or mismatched), the feature vectors are used to train probabilistic scorer. Since sequential training (performed once every 6 months) is efficient, in this paper, we do not consider the problem of parallelizing this phase.

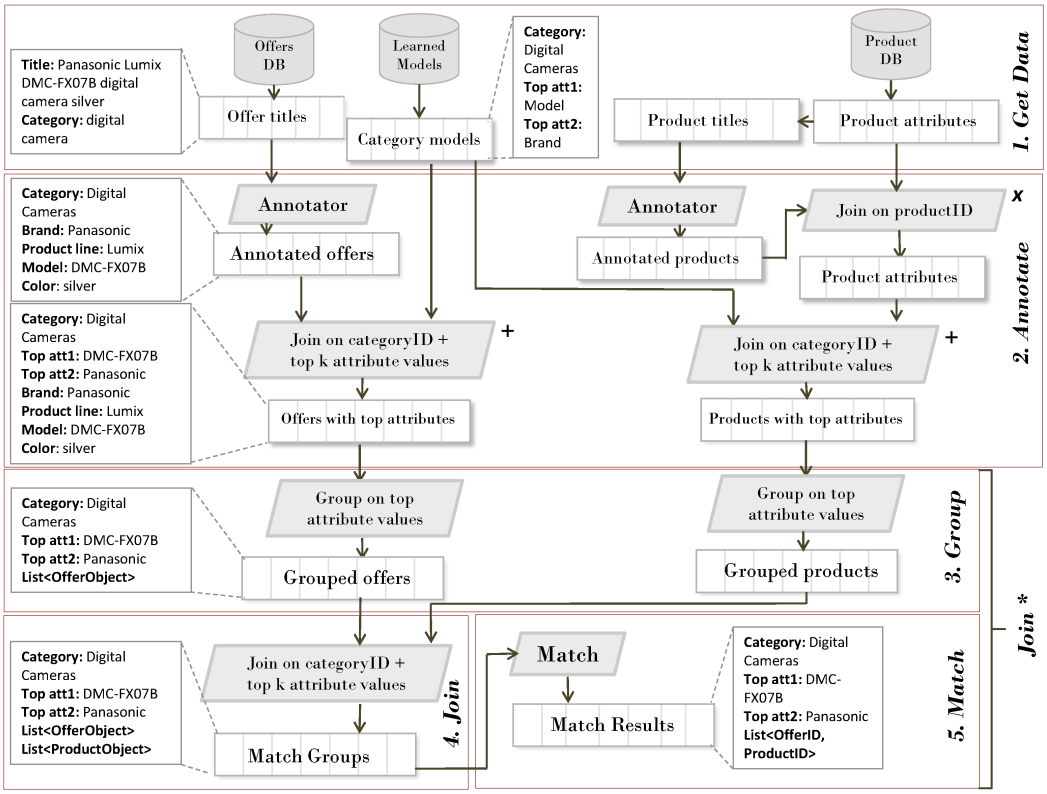

Figure 3: Online matching algorithm outline.

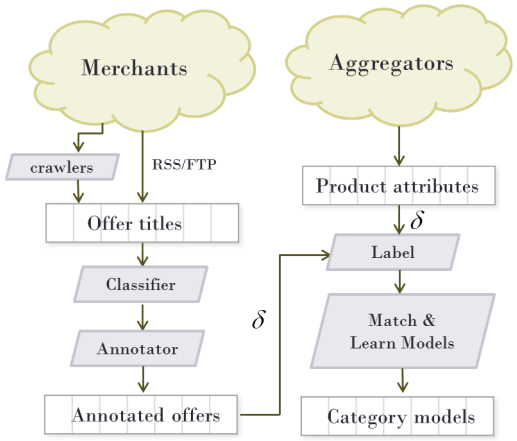

Figure 2: Offline training phase of the category matching models. δ corresponds to a subset of offers/products used for learning the matching models.

Online matching phase: During the online phase, we are given a set of previously unseen offers O', and the goal is to identify the best matching products $p \in P$ for each $o \in O'$. The scoring function learned during the offline phase provides the probability of match for a pair $\langle o, p \rangle$. Naively, we can find the best match by pairing $o \in O'$ with *every* $p \in P$, calculating the pair match score, and choosing the p^* that results in the highest score. However, such naive pairing will cost $O(|P| \times |O|)$ operations. Instead, [16], proposed a staged blocking strategy. This involves first categorizing the offer into a category node in the taxonomy, and then further reducing the *consideration set* of potential match products to those that agree on the value of the attribute that contributes the largest weight, as learned by the matcher for that category. We build on this blocking notion as described in the next section. Even with this pruning strategy the number of products and offers that need to be matched is skewed across categories, so we need to further optimize for this data skew.

2.2 Data skew in offer matching

During the matching process, offer strings need to be scored against potential products that could match the offer (the offer's *consideration set* as mentioned in the previous section). The baseline approach, selects potential products that can match an offer to be all products in the category the offer was classified to. However, this presents a lot of wasteful work, as the number of products in a category can be very large (10s of thousands), but the number of relevant products is usually much smaller (10s or 100s). In addition, there

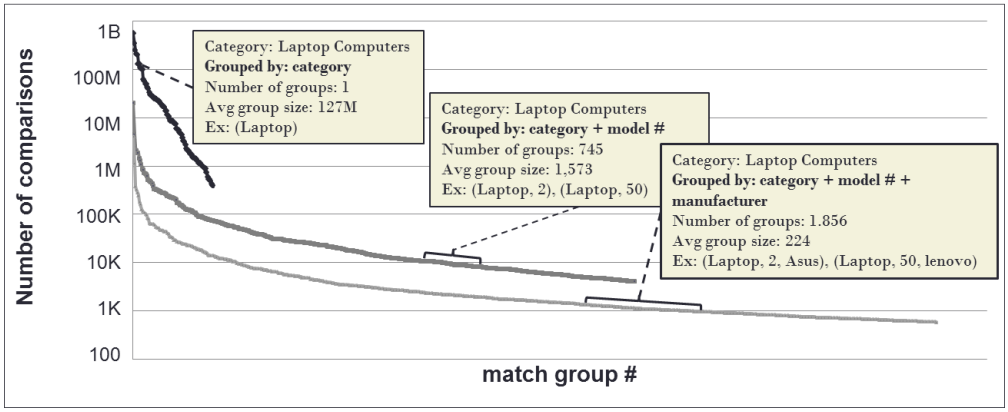

Figure 4: Matching job size distribution when grouping by category (black line), category and one top attribute (dark gray line) and category and two top attributes (light gray line). The captions show an example for refining matching jobs in the "Laptop Computers" category.

is a large skew in the distribution of the products and offers across the categories as shown in Figure 4.

First, we define *important attributes* for a category to refer to attributes that are used in the matching functions for that category that are learned in the offline matching phase (Figure 2). We also define *top attributes* as the set of attributes with the highest weight out of the important attributes for each category.

In order to create a more efficient matching strategy, we first need to select only those products for a particular offer that have common top attribute values with the offer. This set of attributes is category-specific. Referring to the example in Figure 1 for an offer in the "Digital Cameras" category with a title "Panasonic DMC-FX07B digital camera silver", the consideration set for this offer should consist of products with brand name "Panasonic" because brand name is the top attribute for the "Digital Cameras" category and not "Nikon" or "Sony". We can therefore group all the offers with brand name "Panasonic" with all the products with brand name "Panasonic" as potential match pairs and do matching only on these subsets. Note that offers with the same values for top attributes have the same consideration set.

As shown in Figure 4, the sizes of the matching jobs decrease with grouping by one and two top attributes as expected; however, we still see a large skew in the distribution of the matching job size for different groups. Thus, grouping by top attributes helps reduce the matching task size but in some cases still presents a challenge for efficient parallel implementation due to the data skew. Section 3 discusses how we mitigated the data skew problem in our parallel matching algorithm implementation. Also, by creating finer-grained groups by grouping offers and products on two or more top attributes we require top attribute value equality between offers and products. With noisy dataset, this strategy results in loss in recall. The granularity of grouping is a tunable parameter that can trade off between application performance and accuracy and is also further discussed in Section 3.

2.3 Offer matching algorithm steps

Before describing the parallelization of the offer matching algorithm, we first present the algorithm steps. The five steps in the parallel matching algorithm are *Get Data*, *Annotate*, *Group*, *Join* and *Match*. The steps are shown in Figure 3 and described in detail below.

Get Data Phase *- Retrieve the offer, product, and category attribute data (Figure 3(1)).*

First step in the matching process is to read the offer and product data from databases (data from aggregators and merchants is collected into databases independently from the offer-matching application). The input to this step is a list of categories we wish to do the matching on. The result is a set of collections of offer titles, product titles and attributes, and category attributes.

Annotate Phase *- annotate the offers (i.e. get the attribute values) from the offer and product titles, augment product attributes from database with attributes from product title (Figure 3(2)).*

After obtaining the data we need for matching, we annotate the offer and product titles for attribute values. In order to only extract the important attributes for the offer from the Get Data phase we consult the category attribute collection when annotating the offer titles to only keep values for important attributes in that category. For details of offer title annotation, please see [16].

To annotate products, we similarly extract important attribute values for product titles and then augment the product attribute collection with the attributes extracted from the titles by joining the two tables on the productID key (shown on the right in Figure 3(2)). For attributes that have both a value extracted from the title and the product attribute database we keep the value with higher priority. In our implementation, we can specify this priority depending on the cleanliness of the product data.

The next two steps ***Group, Join*** in the original sequential algorithm correspond to a single Join() operation. We join

the annotated offers and products on top attributes. Figure 3(3,4) show the parallel implementation of this phase and Section 3 describes the implementation and the need for separate Group() and Join() steps in detail.

Match Phase *- match the offers and products within each joined group (Figure 3(5)).*

Finally, for each offer in each match group we compute a matching score between the offer and all the products in the consideration set and pick the maximum scoring product(s). For details in the scoring function computation please refer to [16]. This is the most time-intensive step of the algorithm and we further optimize this step by utilizing the nested on-chip parallelism of the cluster, as discussed in Section 3.

3. PARALLELIZATION OF MATCHING

Referring to the algorithm shown in Section 2 we describe our parallelization stategy for each algorithm phase.

The input data (*GetData*, step 1) is currently obtained by opening a connection to a SQL Server database and extracting the data as a set of typed records on the client machine. Since this is a sequential process, after parallelization this remains the most expensive step. This step could also be sped-up by storing the data in the cluster using a parallel file system; we have not done so because we do not control the input database. We do not include the cost of extraction in our measurements.

Annotate, step 2, is completely parallel in the offer titles; there are millions of such records. The records are distributed by using hash-partitioning, a well-known randomized load-balancing technique pioneered by parallel databases. For instance, DryadLINQ provides a primitive HashPartition operator for this purpose, which spreads a dataset into a specified number of partitions.

Three different Join computations are performed in this step. Two of them in the Annotate step shown (left side of Figure 3(2), marked with a +)) join a big table of annotated offers and products with a small table of category attributes, so they are implemented by broadcasting the category attribute table to all partitions of the large offer and product tables. This pattern is sometimes called Map-Join.

The third Join in the Annotate step (right side in Figure 3(2), marked by a $\times$) is implemented using the regular Join operator. This operation generates a distributed Join implementation by hash-partitioning both input sets. This is similar to the Join implementation used in Pig and other MapReduce engines.

In *Group*, step 3, shown in Figure 3(3), we use a Group Join operator to build pairs of lists: (offers, products). All such pairs have distinct keys, so they can be processed independently. This operation is essentially equivalent with the Pig CoGroup operator; the outputs are groups of elements sharing the same key of top attribute values.

The *Join*, step 4, shown in Figure 3(4), performs a standard distributed Join (using deterministic hash-distribution) between the groups computed in step 4; the output is composed of pairs of groups.

Steps 3-4 together constitute the (hand-optimized) implementation of a third type of Join operation (bottom right of Figure 3, marked by a $*$). This is a Join operation between two large datasets with significant skew. The number of offers and products per group is very skewed as discussed in Section 2.2. These group Join operations could collectively be replaced with a single Join call. However, due to the data skew using the default Join implementation would cause a handful of machines to compute for a long time, keeping all other machines idle therefore severely underutilizing the hardware resource. Figure 5(a) shows this computation schedule.

The *Match*, step 5, shown in Figure 3(5), is composed of three substeps: estimate data skew, dynamically repartition data and compute the cartesian products. A Map operation is used to estimate the work required for each pair of groups. The large pairs (where $|O_i| \times |P_i| > t$, for some threshold t, say $1M$) are dynamically repartitioned by splitting the offer list O_i into several smaller lists. This additional dynamic data-dependent Map computation incurs a small additional overhead (contributing to roughly 10% of the running time), but leads to much better load balancing in the matching phase, as we show in the next section. Figure 5(b) shows the schedule of the load-balanced matching.

The Cartesian products of offers-to-product matches, are computed and aggregated using an associative function ($max()$ across all product scores for each offer). In consequence they can be computed in a streaming fashion, without materializing all the $|O_i| \times |P_i|$ partial results. Finally, in order to utilize the on-chip parallelism of the cluster, we further parallelize the matching step by using the .NET threading library; this is done by splitting each O_i list into K disjoint lists, one for each core. Figure 6 illustrates this data distribution used by the Matching algorithm.

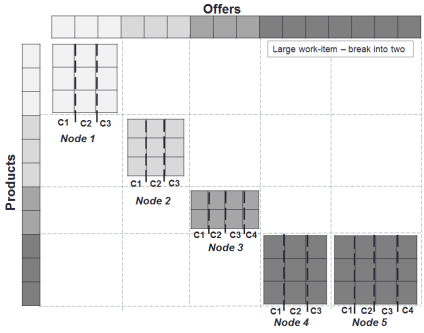

Figure 6: Parallelization of the matching tasks across nodes and cores. C* stands for core *. Different shades of gray correspond to different match groups.

As discussed in Section 2.2 there is an interesting trade-off between the work performed and the completeness of the results. This trade-off can be controlled by the number of top attributes used for matching. In terms of parallel performance, using more attributes performs a finer-grained partitioning of the work. Unfortunately, because the input data is not clean, many attribute values can be incorrect or miss-

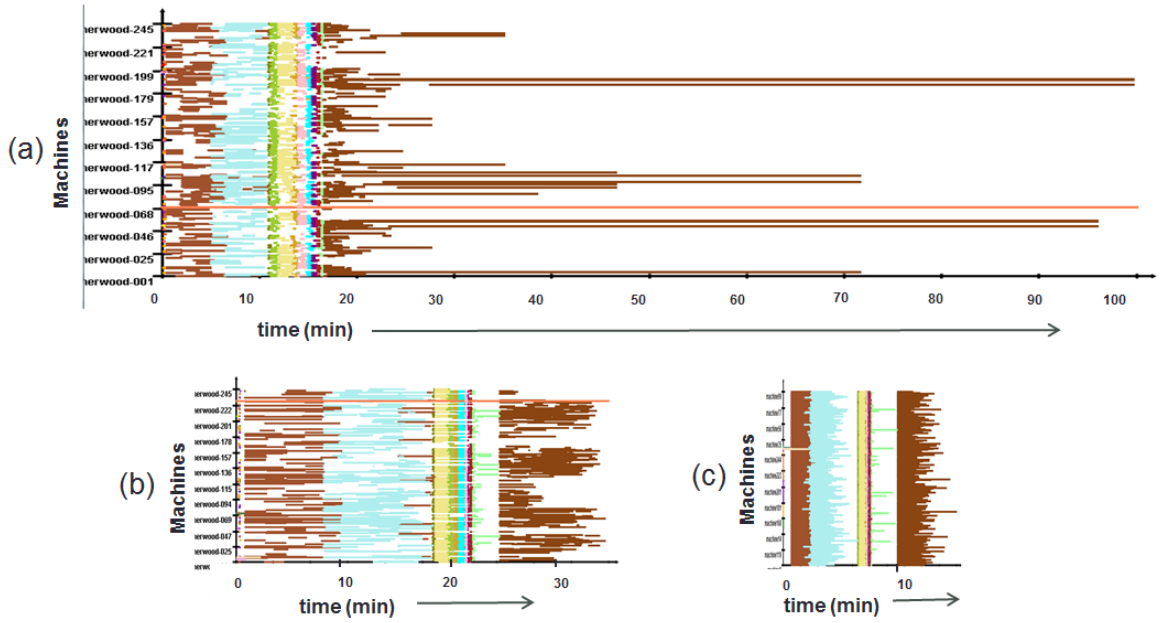

Figure 5: Impact of application-level load balancing and scheduling. (a) shows unbalanced execution. The execution time of the Matching phase is substantially reduced from (a)(no refinement) to (b)(splitting large Cartesian products). (c) shows the ideal schedule of (b) with no cluster sharing.

ing, and using their values for partitioning can decrease the recall of our algorithm (i.e. the more attributes we group by, the more we're enforcing the strict equality between product and offer attribute values thereby losing recall). However, often the dropped offers have weaker matches originally, since they do not agree on top attribute values. Thus, this loss in recall may be tolerable for cleaner input data sets.

4. PERFORMANCE RESULTS

We have implemented the distributed version of our matching algorithm using the DryadLINQ system. DryadLINQ [26] is a compiler which parallelizes LINQ programs to run on large computer clusters. The LINQ language (Language-Integrated Query) is integrated within other .NET languages (such as C# and Visual Basic); it provides to the programmer a set of data-parallel operators for manipulating data collections. LINQ is similar to the SQL database language, the LINQ collections being equivalent with SQL tables and views. DryadLINQ generates code for the Dryad [14] distributed execution engine. Dryad offers a generalization of the MapReduce programming model, supporting arbitrary directed-acycling dataflow graphs. Furthermore, DryadLINQ enables the programmers to use nested parallelism, by parallelizing the application both across the machines of a cluster and across the cores of a machine. In this work we have used DryadLINQ functionality which allows the user to override the default multi-core parallelization, and we wrote custom multi-threaded code. The core functionality of DryadLINQ has been emulated using basic MapReduce as a runtime layer, in projects such as Pig [21] and FlumeJava [5].

We have evaluated our parallelized algorithm on a cluster of 240 dual-CPU dual-core machines. Each of the machines was running Windows Server 2003 64-bit. The machines were equiped each with two 2.6GHz dual-core AMD Opteron 2218 HE CPUs, 16 GBytes of DDR2 random access memory, and four 750 GByte SATA hard drives.

Our cluster is a shared resource cluster for running DryadLINQ jobs; resource requests of multiple competing jobs are arbitrated by a fair-scheduler [15]. The cluster was quite heavily used during the period of measurements. The number of machines granted to our jobs varied dynamically during job execution, depending on the number of competing jobs, and thus the running times exhibited large variances. In order to report reproducible results we have built a tool which allows us to factor out most of the influence of other jobs. This tool receives as input the details of a job execution on the cluster (e.g., information about the dataflow graph and process running times) and then estimates the amount of time the job would take if the cluster ran no other competing jobs. This estimation is performed by essentially computing an off-line greedy schedule of the job. All the cluster running times reported in this paper are computed in this way. Let us notice that we are not reporting idealized running times: our schedules are still penalized by slow machines, failures and include data transfer times and time lost to network contention; and the work performed is the same in both instances. The only thing that we are idealizing is the absence of competing jobs.

Figures 5(b) and (c) show two example schedules. The X axis denotes time, and the Y axis denotes the machines in the cluster. Each horizontal line shows a machine being utilized. The color of the line denotes the phase of the computation performed by the machine. Figure 5(b) shows the actual schedule of a computation on a shared cluster; many computations are delayed by machines being assigned to other

jobs. Figure 5(c) shows the schedule of the same computation run on an idle cluster, where free machines are available as soon as requested. Our tool produces the schedule in (c) by "compacting" the schedule from (b).

Speedup

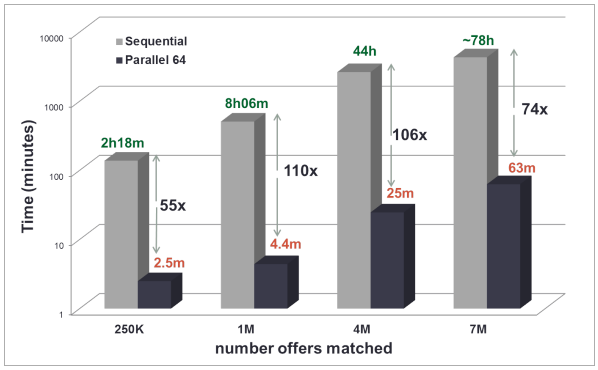

Figure 7: Speedup of the parallel offer matching over sequential code for different data sizes

Figure 7 shows the speedup of the parallel implementation over the serial code for various problem sizes. The parallel runs use all 240 machines. The time for the largest sequential job (7M offers) is only approximated, since our resources did not allow for such long-running jobs. The largest speedup is for 1 million and 4 million offers (the $1M$ and $4M$ data points in Figure 7) of over 100× over the sequential implementation, reducing the matching time from 8 hours to 4 minutes and from 44 hours to 25 minutes respectively.

Scaling

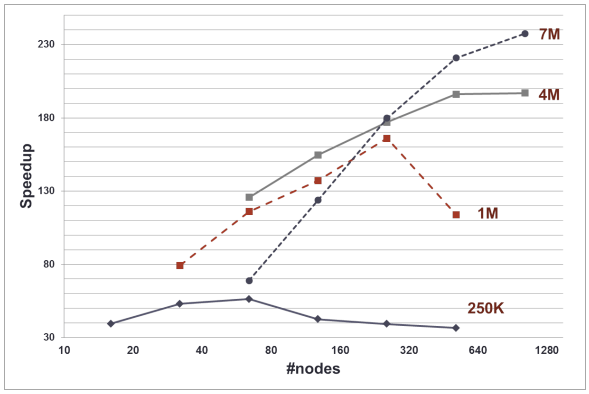

Figure 8: Scaling of the parallel offer matching code. 1 node is one dual-CPU dual-core node.

In Figure 8 we estimate the efficiency of our parallelization by estimating the running time on clusters with variable number of machines. For smaller data sets (250K and 1M offers) the speedup decreases with increasing number of nodes, since not all machines can be kept busy. Large problem sizes (4M and 7M), provide good scaling up to 1280 machines (5120 cores).

Load balancing

The schedule in Figure 5(a) shows the impact of skew in the dataset on the cost of the matching step. Notice that the last computation stage is dominated by a few machines processing large Cartesian products. Using our load-balancing optimization which splits large products into several independent products we obtain the schedule shown in Figure 5(b). The additional cost of the splitting stage (10%) is more than compensated by the reduced cost of matching (which reduces the overall running time by 65%).

Multi-threading

In our experiments, the multi-threaded implementation of the matching phase for one matching job yields a 3.5× improvement in performance using 8 threads for optimal utilization of the processors. For each matching job, we fork threads to compute subsets of offer-product matchings; each node is assigned a set of matching jobs, thus we fork and join the threads for each element in the matching job set. The fork-join overhead reduces the overall performance gain from multi-threading to 1.6×. This overhead could be reduced by using a fixed set of threads pulling from a task queue.

5. RELATED WORK

There has been a lot of interest in distributed data parallel systems [4, 6, 8, 14, 1, 12, 26, 21, 5, 22] based on the MapReduce paradigm as well as parallel databases. In this paper we use the DryadLINQ engine to effectively parallelize web-scale data processing application of offer matching. The core functionality of DryadLINQ has been emulated using basic MapReduce as a runtime layer, in projects such as Pig [21] and FlumeJava [5].

Efficient handling of data skew when performing large-dataset joins has been addressed in the database community by [9, 25, 23, 13]. In data-parallel MapReduce frameworks, [17] implemented data skew handling on top of Hadoop [1]. Many large-scale applications exhibit large data skew (for example [2] and [27]) and their efficient performance depends highly on efficiently handling the skew.

There has been much work done to create efficient algorithms in the field of record matching. For instance [18] describes a simple classification strategy for matching structured records to structured records by first clustering the records on common attributes. Our paper presents parallelization techniques described that would also apply to this structured record matching technique if the data exhibited significant skew.

6. CONCLUSION

This paper presents a detailed implementation of a large-scale data processing application using an example MapReduce framewok of DryadLINQ. While we have focused on the e-commerce problem of offer matching, we believe that the lessons learned are applicable in a wide variety of applications that exhibit large data skew such as data cube computation and botnet detection ([2] and [27]) as well as other offer matching approaches such as [19, 7, 3, 24, 18].

In particular, we have learned that an important feature of the programming language used for programming large-scale systems is to contain a mix of both high-level and low-level computation and data manipulation primitives. In our implementation we used both high-level primitives to express our computation and also hand-optimized the application using low-level primitives to implement dynamic data repartitioning and multi-core threading to mitigate the data skew of our input dataset and to fully utilize the parallel hardware of the cluster. Without the low-level flexible programming primitives to handle data skew in our application, the performance of the offer matching application would suffer from a very unbalanced execution and would result in at least a 3-fold increase in compute time.

In the future we expect that some of the techniques for automatically handling large data skew we have deployed are incorporated into automatic optimizations. In the meantime, using a MapReduce framework with both high-level and low-level programming primitives is invaluable in dealing with common issues such as data skew in large-scale data processing applications.

7. REFERENCES

[1] Apache Hadoop.

[2] P. B. R. R. Arnab Nandi, Cong Yu. Distributed cube materialization on holistic measures. Hanover, Germany, 2011.

[3] M. Bilenko, R. Mooney, W. Cohen, P. Ravikumar, and S. Fienberg. Adaptive name matching in information integration. *IEEE Intelligent Systems*, 18:16–23, 2003.

[4] R. Chaiken, P.-øA. L. Bob Jenkins, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. February 2008.

[5] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and Nathan. FlumeJava: Easy, efficient data-parallel pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[6] L. Chu, H. Tang, T. Yang, and K. Shen. Optimizing data aggregation for cluster-based internet services. In *Symposium on Principles and practice of parallel programming (PPoPP)*, pages 119–130. ACM, 2003.

[7] W. W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. pages 201–212, 1998.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. San Francisco, CA, December 2004.

[9] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 27–40, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[10] I. Forrester Research. `http://techcrunch.com/2010/03/08/forrester-forecast-online-retail-sales-will-grow-to-250-billion-by-2014/`, 2011.

[11] G. Fulgoni. The 2010 u.s. digital year in review. Technical report, Comscore, 2010.

[12] Y. Gu and R. Grossman. Sector and Sphere: The design and implementation of a high performance data cloud. *Philosophical Transactions of the Royal Society*, 367(1897):2429–2445, June 2009.

[13] K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, pages 525–535, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*, pages 59–72, Lisbon, Portugal, March 21-23 2007. also as MSR-TR-2006-140.

[15] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In J. N. Matthews and T. E. Anderson, editors, *ACM Symposium on Operating Systems Principles (SOSP)*, pages 261–276, Big Sky, Montana, USA, 2009.

[16] A. Kannan, I. E.Givoni, R. Agrawal, and A. Fuxman. Matching unstructured product offers to structured product descriptions. Technical Report MSR-TR-2010-172, Microsoft.

[17] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 75–86, New York, NY, USA, 2010. ACM.

[18] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '00, pages 169–178, New York, NY, USA, 2000. ACM.

[19] A. Monge and C. Elkan. The field matching problem: Algorithms and applications. In *In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, pages 267–270, 1996.

[20] Nielsen. Global trends in online shopping: A nielsen global consumer report. Technical report, June 2010.

[21] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *ACM SIGMOD International Conference on Management of Data (Industrial Track) (SIGMOD)*, Vancouver, Canada, June 2008.

[22] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive – a petabyte scale data warehouse using Hadoop. In *International Conference on Data Engineering (ICDE)*, pages 996–1005, Long Beach, California, March 1-6, 2010 2010.

[23] C. B. Walton, A. G. Dale, and R. M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, pages 537–548, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[24] W. E. Winkler, W. E. Winkler, and N. P. Overview of record linkage and current research directions. Technical report, Bureau of the Census, 2006.

[25] J. L. Wolf, D. M. Dias, P. S. Yu, and J. Turek. An effective algorithm for parallelizing hash joins in the presence of data skew. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 200–209, Washington, DC, USA, 1991. IEEE Computer Society.

[26] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. page 14, San Diego, CA, December 8-10 2008.

[27] Y. Zhao, Y. Xie, F. Yu, Q. Ke, Y. Yu, Y. Chen, and E. Gillum. Botgraph: large scale spamming botnet detection. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 321–334, Berkeley, CA, USA, 2009. USENIX Association.