Winner Determination in Huge Elections with MapReduce

Theresa Csar[1] and **Martin Lackner**[2] and **Reinhard Pichler**[1] and **Emanuel Sallinger**[2]
[1] TU Wien, Austria
[2] University of Oxford, UK
{csar, pichler}@dbai.tuwien.ac.at
{martin.lackner, emanuel.sallinger}@cs.ox.ac.uk

Abstract

In computational social choice, we are concerned with the development of methods for joint decision making. A central problem in this field is the winner determination problem, which aims at identifying the most preferred alternative(s). With the rise of modern e-business platforms, processing of huge amounts of preference data has become an issue. In this work, we apply the MapReduce framework – which has been specifically designed for dealing with big data – to various versions of the winner determination problem. We obtain efficient and highly parallel algorithms and provide a theoretical analysis and experimental evaluation.

1 Introduction

Winner determination is a central problem in social choice. In recent years, developing algorithms for winner determination has become an active research topic in the AI community, in particular in computational social choice. For many of the voting rules and scenarios important in the area, efficient algorithms have been devised (Dwork et al. 2001; Sandholm 2002; Betzler, Guo, and Niedermeier 2010; Bachrach, Betzler, and Faliszewski 2010; Lang et al. 2012; Caragiannis et al. 2014).

The classical example of a winner determination problem is a political election. For political elections, the number of votes may be large, but the number of candidates is typically small. Yet, we are facing ever increasing volumes of preference data coming from different sources: Whether a user watches or rates a movie on Netflix, buys or reviews a book at Amazon or clicks a link in a listing of search results, preferences of some alternatives over others are constantly expressed throughout a user's digital presence.

Unlike voting in a political election where ballots are cast, preference data in a digital environment may be generated without the user's knowledge: For example, when using a typical e-commerce site, the choice of clicking on a particular product in a list of search results is often interpreted as a preference for that product relative to the others the user did not access. The increasing use of sensors (e.g. through a user's "smart" phone or watch) further facilitates the collection of data that can be interpreted as preferences.

The reason for the huge size of preference datasets may vary, sometimes stemming from a huge number of candidates (such as in the case of search engines) or a huge number of votes (such as in the case of preferences generated by sensors). A growth in either dimension poses challenges to the design of parallel algorithms; sequential algorithms and systems for winner determination cannot be expected to handle huge preference datasets of this sort.

The most successful framework to design algorithms for handling huge amounts of data is the MapReduce framework (Dean and Ghemawat 2008), originally introduced by Google and since then adopted by many other companies and projects for processing "big data" in parallel – in clusters or in the cloud. The standout characteristic of the MapReduce framework is that it is both widely deployed in practice, but also well-studied in terms of its theoretical properties.

The goal of this paper is to design and analyse algorithms for winner determination that are able to deal with huge datasets. To this end, we shall adopt the MapReduce framework as the foundation of our algorithms.

Organization and main results. We start with brief introductions to the MapReduce framework and to computational social choice in Sections 2 and 3, respectively. Our main contributions, given mainly in Sections 4 – 6, are:

- We present MapReduce algorithms for four concrete voting rules (scoring rules, Schwartz, Smith, and Copeland) and investigate their theoretical properties.
- The most involved of our MapReduce algorithms (namely the one for computing the Schwartz set) has been implemented and evaluated using the Amazon Elastic MapReduce (EMR) platform. We report on these results.
- Finally, we also show limits of parallelizability, by proving that determining whether a given candidate wins an election subject to the single-transferable voting (STV) rule is P-complete. Thus, it is unlikely that there exists a highly parallelizable algorithm for this problem.

With this work, we demonstrate the potential of the MapReduce framework for designing efficient, highly parallel algorithms for a central computational problem in social choice.

2 Basic Principles of MapReduce

MapReduce, originally developed at Google (Dean and Ghemawat 2008), has evolved into a popular framework for large

scale data analytics in the cloud. A MapReduce algorithm consists of three phases: *map*, *shuffle*, and *reduce* phase. In the map phase, the input is converted into a collection of key-value pairs. The `key` determines which reduce task will receive the `value`. In the shuffle phase, the system sends the key-value pairs to the respective reducers. Each reduce task is responsible for one key and performs a simple calculation over all values it receives.

Introduction by example. We illustrate the main ideas of MapReduce by applying it to the winner determination problem of the Borda scoring rule (see Figure 1). Every voter provides a ranking of the candidates. Let us assume that the candidates are $\{a, b, c\}$ (e.g., a vote could be $a > b > c$). The candidate ranked first receives 2 points, the second 1 point and the last 0 points. The candidate is used as the *key* and the points are the corresponding *values* so that we obtain key-value pairs of the form (candidate, points). Each reducer sums up the points for one candidate. In a final, non-parallel step all candidates with the highest score are determined.

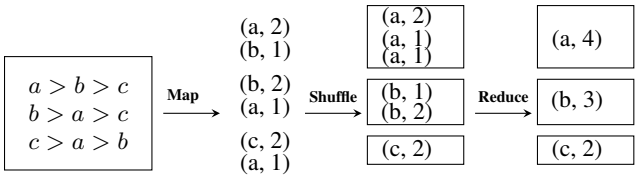

Figure 1: Calculation of scores for the Borda scoring rule.

Analysis of MapReduce algorithms. Various parameters are used to analyze the performance of MapReduce algorithms (Afrati et al. 2013; Leskovec, Rajaraman, and Ullman 2014). An important cost factor is the *total communication cost* (denoted tcc), which is the total number of input/output actions performed by the map and reduce tasks (recognizable as emit() and return() statements in our algorithms). It is common practice to ignore the input to the first map task (i.e., the problem instance) and the output of the overall result, since they do not depend on the chosen algorithm. Moreover, if a data item is sent by one task and received by another task, we only count this as one input/output action.

Closely related to tcc is the *replication rate* (denoted rr), which is defined as the ratio between the amount of data sent to all reducers and the original size of the input. This corresponds to the mean number of reduce tasks a value is sent to. Another parameter of interest is the *number of MapReduce rounds* (referred to as #rounds), which tells us how many map-reduce iterations are performed. This parameter is an essential indicator of how well the problem is parallelizable. We also consider the number of keys (referred to as #keys); for multi-round MapReduce algorithms, we use the maximum number of keys in any round. Note that #keys is a measure for the maximal possible parallelization, which would correspond to every key being assigned to a single (physical) machine. Finally, we will also study the *wall clock time* (wct), which measures the maximum time consumed by a single computation path in the parallel execution of the algorithm (assuming that all keys are processed in parallel). Since the predominant cost factor is the input/output, we identify the wall clock time with the maximum number of input/output data items in any computation path. For the input data we refer to the number of candidates as m and to the number of votes as n. Analysing the simple MapReduce algorithm described above (which immediately generalises to arbitrary scoring rules) gives the following result:

Proposition 1. *The set of winners for scoring rules can be computed in MapReduce with the following characteristics:* $\mathsf{rr} = 1$, $\#\mathsf{rounds} = 1$, $\#\mathsf{keys} = m$, $\mathsf{wct} \leq n + 1$, *and* $\mathsf{tcc} \leq m(n + 1)$.

3 Elections and Winner Determination

The goal of this paper is to establish MapReduce algorithms for the winner determination problem: given a set of candidates C and a list of votes, compute the set of all winners according to a given voting rule. There are two relevant dimensions that can make a problem instance "huge": the number of candidates (m) and the number of votes (n).

We assume that votes are partial orders, i.e., reflexive, antisymmetric and transitive binary relations. As an input data model we consider sets of total orders on subsets of candidates (e.g., $\{a > c > d > b, a > e\}$), which we refer to as *preflists*. They allow one to succinctly encode total orders (i.e., a full ranking) or top orders (a partial ranking with all remaining candidates ranked below); in both cases, preflists require $\mathcal{O}(m)$ space. For arbitrary partial orders they require $\mathcal{O}(m^2)$ space. This is in contrast to, e.g., $(0, 1)$-matrices, which require $\mathcal{O}(m^2)$ space independent of the given votes.

A candidate a strictly dominates b, written $a \succ b$, if there are more votes preferring a over b than the other way around. Similarly, a candidate a weakly dominates b, written $a \succeq b$, if the number of votes preferring a to b is greater or equal than the other way around. Given a list of votes, it is generally not clear what candidates should be selected as *choice sets*, i.e., should be chosen as winners. Probably the most natural approach is to consider pairwise comparisons and to declare a candidate to be the winner if it strictly dominates all other candidates. Such a candidate is a *Condorcet winner* – but a Condorcet winner might not exist. Hence a large number of extensions of this concept have been proposed, three of which we study in this paper: the Copeland set, the Smith set and the Schwartz set. All these sets contain only the Condorcet winner if it exists and are guaranteed to select at least one winner. We have selected these three choice sets since the complexity of computing them was shown by Brandt, Fischer, and Harrenstein [2009] to lie in the complexity classes TC_0, AC_0 and NL, respectively, and problems in these classes are considered as highly parallelizable (Johnson 1990).

The Copeland set is based on *Copeland scores*. The Copeland score of candidate a is defined as $|\{b \in C : a \succ b\}| - |\{b \in C : b \succ a\}|$. The Copeland set is the set of candidates that have the maximum Copeland score. The Smith set is the (unique) smallest set of candidates that dominate all outside candidates. The Schwartz set is the union of minimal sets that are not dominated by outside candidates. The Smith set is always a subset of the Schwartz set (Brandt, Fischer, and

Harrenstein 2009). We refer the reader to (Brandt, Brill, and Harrenstein 2014) and to the handbook chapter of Brandt, Brill, and Harrenstein (2016) for an overview on these and other choice sets.

In addition to the aforementioned rules, we consider the Single Transferable Vote (STV) rule, which is not based on the dominance relation. We introduce it in Section 6.

4 MapReduce Algorithms

In this section, we present algorithms for determining the Schwartz, Smith, and Copeland sets. Central to these three voting rules is the (strict or weak) dominance graph, for which we thus present a MapReduce algorithm first.

4.1 Computation of the Dominance Graph

The (strict or weak) dominance graph contains the candidates as vertices and has an edge between vertices a and b if a (strictly or weakly) dominates b. We use $D_{\succ}$ (resp. $D_{\succeq}$) to denote the strict (resp. weak) dominance graph or simply D if the variant of the dominance graph is clear from the context or not important to the specific case. We consider the representation of the dominance graph by an adjacency matrix. Alternative representations (e.g., by a table of edges) would yield similar results. For simplicity, we shall speak about the (weak or strict) "dominance matrix" as a shorthand for the "adjacency matrix of the dominance graph".

A straightforward MapReduce algorithm for computing the (strict or weak) dominance graph needs one map-reduce round (with processes MapPreflists and ReduceSum) and assembles the dominance matrix in a post-processing step.

MapPreflists takes as input the votes in the form of preflists. It emits key-value pairs where the key (i, j) is a pair of candidates i and j and the value is either 1 or -1 to indicate that in some vote candidate i is preferred to j (1) or vice versa (-1). To this end, MapPreflists processes every total order in every preflist and, for any two candidates i, j occurring in this total order, either emits $((i, j), 1)$ and $((j, i), -1)$ (if i is preferred to j) or $((i, j), -1)$ and $((j, i), 1)$ (otherwise).

Each ReduceSum process receives for one pair (i, j) a list of values 1 or -1. It returns the pair (i, j) together with the sum of these values. In the post-processing step, we assemble the strict (resp. weak) dominance matrix: If the value received for the pair (i, j) is greater than (resp. greater or equal to) 0, then we enter value 1 at position (i, j) in the matrix. Otherwise, we enter value 0.

Proposition 2. *The MapReduce algorithm for computing the (weak or strict) dominance graph has the following characteristics:* $\mathsf{rr} \leq m$, $\#\mathsf{rounds} = 1$, $\#\mathsf{keys} = m^2$, $\mathsf{wct} \leq n + 1$, *and* $\mathsf{tcc} \leq m^2(n + 1)$.

While the original input has size $\mathcal{O}(nm^2)$, the size of the resulting dominance matrix is $\mathcal{O}(m^2)$. If n is huge but m^2 is rather small, then one can clearly use a conventional sequential algorithm to compute choice sets based on this matrix. In contrast, if m^2 is still huge, we have to further rely on parallelization. Our MapReduce algorithms in the subsections below refer to this case.

4.2 Computation of the Schwartz Set

For computing the Schwartz set, we use the following alternative characterisation based on the strict dominance graph: candidate a is in the Schwartz set if and only if for every candidate b, there is a path (in the strict dominance graph) from a to b whenever there is a path from b to a (Brandt, Fischer, and Harrenstein 2009, Lemma 4.5). In Algorithm 1, we present the overall structure of a MapReduce algorithm for this computation.

Algorithm 1 Schwartz Set

```
FirstMap;
ReduceVertex;
while there exists a vertex with new ≠ ∅ do
    MapVertex;
    ReduceVertex;
ComputeSchwartzSet;
```

The central datatype in this algorithm is VertexWritable. It consists of three sets of vertices storing information on incoming and outgoing edges for a given vertex a as follows:

- the set *old* stores all vertices that have been found previously to be reachable from a;
- the set *new* stores all vertices that have been found in the last map-reduce round to be reachable from a;
- the set *reachedBy* stores all vertices known to reach a;

First map-reduce round. The FirstMap process handles each row of the strict dominance matrix as follows. The key-value pairs emitted consist of a vertex as key and a value of type VertexSetWritable. This datatype consists of a set of vertices plus a *mode*, i.e., a value from {'old', 'new', 'reachedBy'}. More precisely, for the i-th row $D[i, -]$, FirstMap emits a pair $(i, (set, \text{'new'}))$ with $set = \{j \mid D[i, j] = 1\}$. Moreover, for every j with $D[i, j] = 1$, the mapper also emits a key-value pair $(j, (\{i\}, \text{'reachedBy'}))$.

In ReduceVertex, the data structure of type VertexWritable is computed for the vertex specified by the input key. In the first round, the set *old* is thus assigned the empty set; the set *new* is assigned the unique input of type VertexSetWritable with mode 'new'; and the set *reachedBy* is assigned the union of all the singletons with mode 'reachedBy'.

Map-reduce rounds in the while-loop. After k iterations of the loop, the VertexWritable data structure for each vertex i contains in the set *new* all vertices reachable from i via a path of length $\leq 2^k$ but not via a path of length $\leq 2^{k-1}$. The vertices reachable via a path of length $\leq 2^{k-1}$ are stored in set *old*. Finally, set *reachedBy* contains all vertices j, such that i can be reached from j via a path of length $\leq 2^k$. Analogously to FirstMap, also MapVertex (Algorithm 2) outputs key value pairs where the key is a vertex and the value is of type VertexSetWritable, i.e., a set of vertices together with a mode from {'old', 'new', 'reachedBy'}. Each MapVertex process handles the VertexWritable data structure for one vertex, say i. With key i, MapVertex emits both sets, *old* and *new* with mode 'old', since the newly found vertices of the

previous iteration of the while-loop are old in the next iteration.

For every vertex r in *reachedBy* as key, MapVertex emits the whole set *new* with mode 'new'. Likewise, for every vertex n in *new* as key, MapVertex emits the set *reachedBy* with mode 'reachedBy'. In other words, the MapVertex process handling vertex i combines the incoming paths and the outgoing paths for vertex i to get paths of length $\leq 2^{k+1}$.

ReduceVertex (Algorithm 3) receives values of type VertexSetWritable for a given vertex i and updates the VertexWritable data structure for i: the union of all input sets with mode 'reachedBy' is assigned to the set *reachedBy*. The unique input set with mode 'old' is assigned to the set *old*. Some care is required with the set *new*, since an input set with mode 'new' may contain vertices j from *old*. This happens if a path from i to j of length ℓ with $2^k < \ell \leq 2^{k+1}$ has been newly discovered, but there also exists a path from i to j of length $\leq 2^k$, which was already detected before. Hence, we may provisionally assign to set *new* the union of all input sets with mode 'new'. But then we set *new* = *new* $-$ *old*.

Algorithm 2 MapVertex

```
input: vertex i; VertexWritable v;
emit(i,VertexSetWritable(v.new, mode='new'));
emit(i,VertexSetWritable(v.old, mode='old'));
emit(i,VertexSetWritable(v.reachedBy,mode='reachedBy'));
for r in v.reachedBy do
    emit (r, VertexSetWritable(v.new, mode='new'));
for n in v.new do
    emit(n,VertexSetWritable(v.reachedBy, mode='reachedBy'));
```

Algorithm 3 ReduceVertex

```
input: key i, list of VertexSetWritable;
new = ∅;
old = ∅;
reachedBy = ∅;
for (set, mode) in input-list do
    if mode = 'old' then old = set;
    if mode = 'new' then new = new ∪ set;
    if mode = 'reachedBy' then reachedBy = reachedBy ∪ set;
new = new \ old;
return (i, VertexWritable(old, new, reachedBy));
```

Example. Suppose there are five candidates (A,B,C,D,E) and the strict dominance graph resulting from the votes is a chain, as shown in Figure 2. When computing the Schwartz set the strict dominance matrix is the input to the first MapReduce round and the reduce tasks create the initial *VertexWritables* as output. These initial VertexWritables form the input to the second Mapreduce round and are shown in the first row of Figure 3.

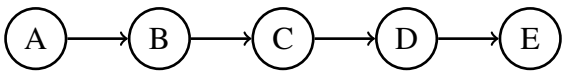

Figure 2: Dominance graph in the example.

The only vertex that can be reached within one step starting from A is B, and therefore the set *new* of candidate A is only containing B. Candidate A cannot be reached by any other vertex (i.e. it is not dominanted by any other candidate) and so the set *reachedBy* is empty. (In the graphs *reachedBy* is abbreviated to *rB*.) Similarly the set *new* of candidate B contains C – the only candidate reachable within one step – and the set *reachedBy* contains A, since $A \succ B$.

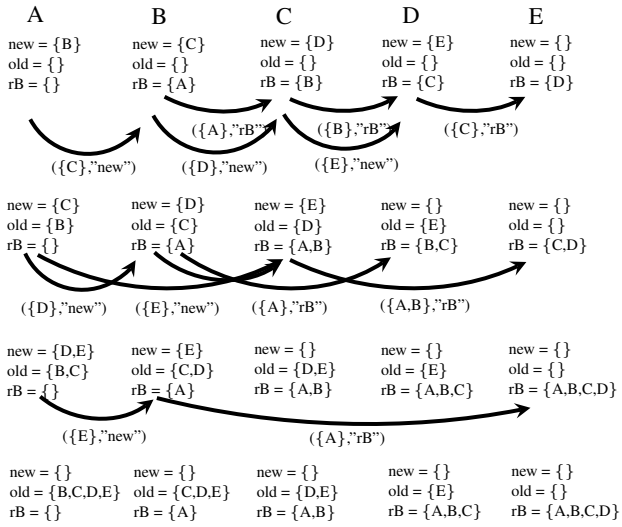

Figure 3: Schwartz set algorithm by example.

In Figure 4 the second MapReduce round is illustrated in more detail. First the map task(s) read the input files containing *VertexWritables*. (The number of map tasks depends on the number and the size of the input files.) For each *VertexWritable* the map tasks emit key-value pairs. The output created by the map task after reading vertex A and B is shown in Figure 4. Vertex A results in emitting only one key-value pair, namely $(A, (\{B\}, "old"))$, since the previous *new* set is considered to be already known (*old*) in the following MapReduce round. Equivalently vertex B emits the already known dominated vertex *new* as *old* $(B, (\{C\}, "old"))$ and passes on the set of *reachedBy* values $(B, (\{A\}, "reachedBy"))$. Additionally Vertex B emits the information that A can reach C, it does this by emitting the following key-value pairs: $(C, (\{A\}, "rB"))$ and $(A, (\{C\}, "new"))$.

In the reduce phase (see Figure 4) there is one reduce task per candidate. Each reduce task receives all key-value pairs with the same key. The reduce task than combines all the received values and creates a new *VertexWritable* as output. All those *VertexWritables* are written to files and used as input to the next round. In Figure 3 the development of the *VertexWritables* during the whole computation is illustrated. The arrows illustrate emitted key-value pairs, but in favor of readability not all of them are shown. In particular, only key-value pairs with differing destination and source vertices are shown. The first row contains the *VertexWritables* created by the first reduce phase and are used as input to the second map phase. Then the second row is the output of the

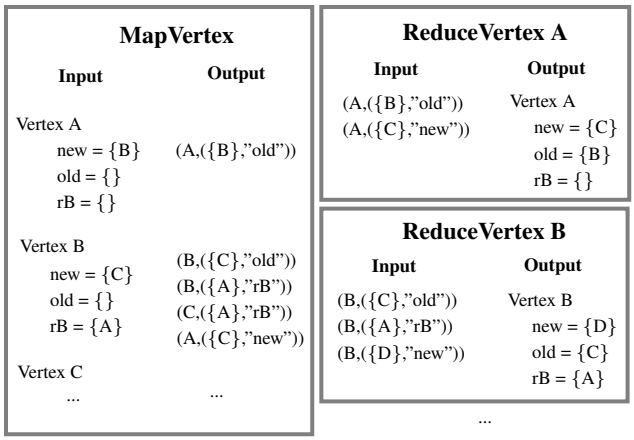

Figure 4: Input and output of the second MapReduce round.

second reduce phase and input to the third map phase and so on. A vertex is considered as inactive as soon as the set *new* is empty. The algorithm terminates when all vertices are inactive (last row in Figure 3). ♢

Post-processing. The ComputeSchwartzSet procedure inspects the VertexWritable data structure for every vertex i and writes i to the output if *reachedBy* $\subseteq$ *old* holds. Of course, this step can be done in parallel for all vertices.

Example (continued). In the last row of Figure 3 the final state of all Vertices is shown. Only vertex A is satisfying the requirement *reachedBy* $\subseteq$ *old* and hence A is the only vertex contained in the Schwartz set. ♢

Proposition 3. *Algorithm 1 for computing the Schwartz set has these characteristics:* $\mathsf{rr} \leq 2m+1$, $\#\mathsf{rounds} = \lceil \log_2 m \rceil + 1$, $\#\mathsf{keys} = m$, $\mathsf{wct} = (2m^2+3m)(\lceil \log_2 m \rceil + 1)$, *and* $\mathsf{tcc} \leq 2m^3 + 3m^2(\lceil \log_2 m \rceil + 1)$.

Note that we have an upper bound $2m^2 + m$ on the input received by each reducer, because each of the m reducers may receive one vertex-set *old* and linearly many sets *new* and *reachedBy*. The linearly many sets may arise, if new paths from r to n via different "mid-points" i have been detected. However, this does not mean that the reducer has to handle data of size $\mathcal{O}(m^2)$ in memory. Instead, the reducer can iterate through these sets one by one and compute their union with only a linearly big data structure in memory. Moreover, tcc has upper bound $\mathcal{O}(m^3)$, which is smaller than $m^2 \cdot \#\mathsf{keys} \cdot \#\mathsf{rounds}$, because the total size of all sets *reachedBy* or *new* that may ever be emitted is bounded by the number of possible combinations $(r, i, n) \in \{1, \ldots, m\}^3$.

4.3 Smith Set and Copeland Set

For the computation of the Smith set, we use the following characterization by Brandt, Fischer, and Harrenstein (2009): candidate a is in the Smith set if and only if for every candidate b there is a path from a to b in the weak dominance graph. A naive algorithm for computing the Smith set would thus first compute the transitive closure of the weak dominance graph with logarithmically many map-reduce rounds as in Algorithm 1. However, we can do better by applying the following property: let $D^k_{\succeq}(v)$ denote the set of vertices reachable from vertex v by a path of length $\leq k$. Brandt, Fischer, and Harrenstein (2009) show that in the weak dominance graph a vertex t is not reachable from a vertex s if and only if there exists a vertex v such that $D^2_{\succeq}(v) = D^3_{\succeq}(v)$, $s \in D^2_{\succeq}(v)$, and $t \notin D^2_{\succeq}(v)$.

A high-level description of a MapReduce algorithm for the Smith set is given in Algorithm 4. The algorithm consists of three map-reduce rounds plus a post-processing step realized by procedure ComputeSmithSet.

Algorithm 4 Smith Set

```
FirstMap;
ReduceVertex;
MapVertex;
ReduceVertex;
MapVertex;
ReduceComplement;
ComputeSmithSet;
```

The first two rounds as well as the map phase in the third round are precisely as in the Schwartz-Set algorithm. The ReduceComplement process takes the same input as ReduceVertex, i.e., values of type VertexSetWritable for a given vertex i. ReduceComplement consists of two phases: first, we check if i is a vertex of kind v in the Smith set characterization recalled above, i.e., the input set with mode 'old' must not contain all vertices and all input sets with mode 'new' must be subsets of the one with mode 'old'. Strictly speaking, we thus check if $D^2_{\succeq}(v) = D^4_{\succeq}(v)$ holds. Of course, this is equivalent to checking $D^2_{\succeq}(v) = D^3_{\succeq}(v)$.

If i indeed is a vertex of kind v, then ReduceComplement outputs all vertices in the set with mode 'old', i.e., all vertices reachable from v. Otherwise, ReduceComplement outputs nothing. In other words, ReduceComplement outputs all vertices which (by the above criterion) have been found to be *not* in the Smith set. The procedure ComputeSmithSet then computes the Smith set as the complement of the union of the sets of vertices thus received.

Proposition 4. *Algorithm 4 for computing the Smith set has the following characteristics:* $\mathsf{rr} \leq 2m+1$, $\#\mathsf{rounds} = 3$, $\#\mathsf{keys} = m$, $\mathsf{wct} \leq 6m^2 + 8m$ *and* $\mathsf{tcc} \leq 6m^3 + 8m^2$.

We conclude this section by briefly discussing the computation of the Copeland set. It just needs one map-reduce round, where each reducer is responsible for computing the Copeland score of one candidate. In the map phase, we send the row and column relevant to a candidate to the corresponding reducer. The entries in the column sent to the reducer are multiplied by -1 as they correspond to the candidates that dominate the candidate under consideration. Each reducer then simply sums up the values of the row and the (negative) values of the column to get the Copeland score of the corresponding candidate. Finally, in a simple post-processing step, the maximum value of the Copeland scores is computed and the candidates with maximum Copeland score are returned as the Copeland set.

Proposition 5. *The Copeland set can be computed by a MapReduce algorithm with the following characteristics:* $\mathsf{rr} = 2$, $\#\mathsf{rounds} = 1$, $\#\mathsf{keys} = m$, $\mathsf{wct} \leq 2m + 2$, *and* $\mathsf{tcc} \leq 2m^2 + 2m$.

5 Experimental Evaluation

We implemented our approach to winner determination in Java using the Amazon Elastic MapReduce (EMR) platform. Amazon EMR provides a managed Hadoop framework, an open-source implementation of MapReduce. For this evaluation, we chose to focus on the Schwartz set, as it is the most complex of the algorithms described in this paper, and the theoretical analysis in the previous section showed that it is the hardest. (We also implemented the Smith and Copeland algorithms as well as dataset generators suited for all our winner determination algorithms.) The goal of our evaluation is to demonstrate that (i) the algorithm for computing the Schwartz set is practicable and, importantly, (ii) that it scales in practice, i.e., that given larger and larger problem instances, one can achieve reasonable times for determining the winners of these elections by using an appropriate number of computation nodes. Our implementation is available as open-source software[1], as is the code for generating the datasets and running the code on EMR.

Setup. In our experiments, we utilize Amazon EMR which uses the Amazon Elastic Compute Cloud (EC2) to provide computation resources. The nodes in an EC2 cluster are called instances, of which Amazon EC2 offers various kinds. In the results reported here, we utilize `m3.xlarge` instances, but note that tests with other types of instances did not yield substantially different results with regard to scalability. We use up to 128 of such instances. The times we report in this section are measured from the start of the first MapReduce round to the end of the final MapReduce round.

Datasets. As our main goal of the evaluation is to show practicability and scalability, we utilize synthetic datasets. For a desired number m of candidates, we randomly generate dominance graphs with a certain number of edges utilizing DigraphGenerator (Sedgewick and Wayne 2016): We use two kinds of datasets, sparser ones with $10m$ edges and denser ones with $m^2/10$ edges. In the times we report in this section, for each number of candidates, we generate five different instances for the $10m$ edges graphs, and report the average of the times measured to generate the winner set. For the $m^2/10$ edge graphs, we generate only one dataset per number of candidates to limit the cost incurred by the experiments. The datasets used to generate our results will be provided as open data.

Schwartz set. Run times for computing the Schwartz set (as described in Section 4.2) are shown in Figure 5 for the sparser ($10m$ edges) datasets, with the number of candidates ranging from 1000 to 7000 and with $1+2$ to $1+32$ EC2 instances (1 control instance and x worker instances). A timeout of 60 minutes was used for this experiment. We see that the time incurred falls below 20 minutes once $1+16$ EC2 instances are used, and below 15 minutes for $1+32$ instances.

[1] `https://github.com/theresacsar/BigVoting`

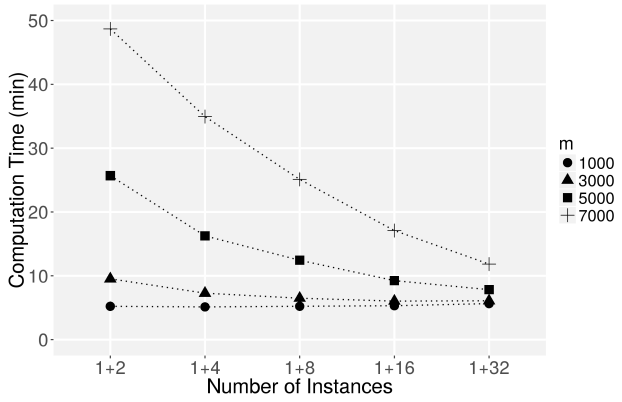

Figure 5: Time for the computation of the Schwartz Set with $10m$ edges using up to $1 + 32$ instances.

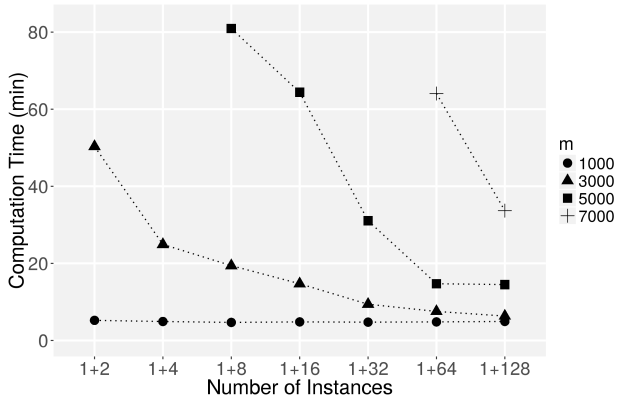

Figure 6: Time for the computation of the Schwartz Set with $m^2/10$ edges using up to $1 + 128$ instances.

The times for the denser ($m^2/10$ edges) graphs is shown in Figure 6, also between 1000 to 7000 candidates, but with up to $1 + 128$ EC2 instances. As Figure 6 shows, the runtimes are higher for the denser graphs, but the general picture remains the same. A timeout of 90 minutes was used for this experiment. We see that the time incurred falls below 40 minutes for most inputs once $1 + 64$ EC2 instances are used and below 20 minutes once $1 + 128$ EC2 instances are used.

Most importantly, we see that the implementation scales: We see that utilizing a larger number of EC2 instances significantly decreases the computation time. Note that the main memory consumption used in the EC2 instances remains $\mathcal{O}(m)$, i.e., relative to the number of candidates, and not $\mathcal{O}(m^2)$. This is essential for scalability, as otherwise main memory size would become a hidden limit for scalability, not obvious in Figures 5 and 6.

Discussion. It is to be noted that the size of problem instances under consideration (with $m \leq 7000$) is not yet in the usual order of magnitude of "big data"; our instances have a size of megabytes rather than gigabytes. Further work is therefore necessary to improve our algorithms and implementations to push the practical boundaries of our approach.

Nonetheless, we demonstrate with our experiments that our algorithms indeed benefit from an increase in parallelization, i.e., a significant decrease in the run time can be observed when the number of processors (instances) is increased.

6 Limits of Parallelizability

In this section we study the Single Transferable Vote (STV) rule as an example of a voting rule that allows for only limited parallelization. STV is defined as follows: Every voter provides a total order of all m candidates. In each round the candidate that is ranked first in the least number of votes is removed from each vote. The remaining candidate after $m-1$ rounds is the winner. In case of ties we assume that a tie-breaking order is given. In the following we will show that STV is in general difficult to parallelize effectively. Indeed, the decision problem STV-WINNER, asking whether a given candidate is the winner, is P-complete and therefore considered as inherently sequential (Johnson 1990).

Theorem 6. STV-WINNER *is* P*-complete.*

Idea. Since this proof requires an intricate construction, we can only provide the main idea. P-hardness is shown by reduction from the BOOLEAN CIRCUIT EVALUATION problem (Greenlaw, Hoover, and Ruzzo 1995). Given a Boolean circuit with m gates $g_1, \ldots, g_m$, where g_m is the output gate, we construct an instance of STV-WINNER with $2m$ candidates $C = \{c_1, \bar{c}_1, c_2, \bar{c}_2, \ldots, c_m, \bar{c}_m\}$. The set of votes is defined such that, in the first m rounds, one of c_i and $\bar{c}_i$ is eliminated for every $i \in \{1, \ldots, m\}$, namely: c_i is retained if and only if gate g_i evaluates to true. In the next $m-1$ rounds, the remaining candidates are eliminated in ascending order of their indices. Hence, c_m is the STV-winner if and only if gate g_m (and hence the circuit) evaluates to true. □

The next theorem shows that only the number m of candidates is the source of P-completeness; the number n of voters is not an obstacle to parallelization.

Theorem 7. STV-WINNER *can be solved in* $\mathcal{O}(m+\log(n))$ *space.*

Proof. We require the following variables to be kept in memory: the current vote under consideration ($i \in \{1, \ldots, n\}$), the current candidate c_j ($j \in \{1, \ldots, m\}$), the current score of candidate c_j ($s \in \{1, \ldots, n\}$), the minimum score of the preceding candidates $c_1, \ldots, c_{j-1}$ ($t \in \{1, \ldots, n\}$), the candidate having this minimum score $c_{j'}$ ($j' \in \{1, \ldots, n\}$) and the set of candidates that have already been removed $A \subseteq C$. The algorithm starts with $i = 1$, $j = 1$, $j' = 0$, $s = 0$, $t = +\infty$ and $A = \emptyset$. We repeat the following steps $m-1$ times: For every candidate $c_j \notin A$ we compute the score of c_j by verifying for each vote V_i whether c_j is the highest ranking candidate not contained in A; if yes, we increase s by 1. If s is smaller than t (that is, we assume lexicographic tie-breaking), we set $t \leftarrow s$ and $j' \leftarrow j$. Once this has been done for every candidate, we add candidate $c_{j'}$ to A, and set $i = 1$, $j = 1$, $j' = 0$, $s = 0$ and $t = +\infty$. The remaining candidate after $m-1$ iterations is the winner. The space requirements of this algorithm are $\log(n)$ for the variables i, s, t, $\log(m)$ for the variables j, j' and m for the set A. □

From the perspective of classical complexity theory, we have shown that STV-WINNER is contained in L (i.e., it can be solved with logarithmic space), if we fix m to a constant. Membership in L can be seen as evidence that a problem is parallelizable as it can be computed in $\log^2$ time with a polynomial number of parallel processors. Theorem 7 can also be seen from the perspective of parameterized space complexity (Elberfeld, Stockhusen, and Tantau 2014). Our result translates to a para-L membership proof for STV-WINNER with parameter m, which requires that the problem can be solved in $\mathcal{O}(f(m) + \log(n))$ space for some computable function f. Note that para-L containment is a stronger result than L membership for fixed m since the latter would also hold, for instance, for a space complexity of $\mathcal{O}(m \cdot \log(n))$.

To support Theorem 7, we briefly sketch and analyze a MapReduce algorithm for STV winner. The basic idea is to use $m-1$ rounds and exclude one candidate per round. Each reducer is responsible for one candidate. During the map phase, the highest-ranking not-yet-excluded candidate of each vote is sent to the corresponding reducer, which simply counts the number of received messages. The next round starts with the exclusion list extended by the lowest scoring candidate (subject to tie-breaking). Clearly, this algorithm is impractical for large m as it requires $m-1$ rounds. However, for small m, this algorithm can be considered feasible – which matches exactly the claims of Theorems 6 and 7.

Proposition 8. *For computing STV, we obtain the following characteristics:* $\mathsf{rr} = 1$, $\#\mathsf{rounds} = m-1$, $\#\mathsf{keys} \leq m$, $\mathsf{wct} \leq (m-1)(n+1)$, *and* $\mathsf{tcc} \leq \frac{(m+2)(m-1)}{2} \cdot (n+1)$.

7 Conclusion

This paper presents parallel algorithms for winner determination problems using the popular MapReduce framework, which are particularly useful in a situation where m^2 is "large" (e.g., the full dominance graph cannot be stored and processed in memory by a single machine), but m is still manageable, which is a reasonable assumption for most huge datasets. The main characteristics of our algorithms are summarized in Table 1. Our experimental results reported in Section 5 are promising: the current data clearly shows speed ups when increasing the number of machines. Also, the speed up rate increases for larger instances, i.e., parallelization is more beneficial when it is needed most.

To the best of our knowledge, this paper is the first to apply MapReduce or related techniques to problems from computational social choice. As a consequence, many directions for future research are left to be explored. First, there are many more voting rules to be investigated for their parallelizability. For some of them, such as the Kemeny rule, winner determination is NP-hard (Bartholdi III, Tovey, and Trick 1989; Hemaspaandra, Spakowski, and Vogel 2005) and thus is unlikely to allow for practical parallel computation. However, heuristic algorithms (Dwork et al. 2001; Davenport and Kalagnanam 2004) might be parallelizable.

Problem	Input	Input Size	#keys	rr	#rounds	wct	tcc
Scoring rules	total orders	$\mathcal{O}(mn)$	m	1	1	$n+1$	$m(n+1)$
Dom. graph	partial orders	$\mathcal{O}(nm^2)$	m^2	m	1	$n+1$	$m^2(n+1)$
Schwartz set	dom. graph	$\mathcal{O}(m^2)$	m	$2m+1$	$\lceil \log_2 m \rceil + 1$	$\mathcal{O}(m^2 \log m)$	$\mathcal{O}(m^3)$
Smith set	dom. graph	$\mathcal{O}(m^2)$	m	$2m+1$	3	$6m^2+8m$	$6m^3+8m^2$
Copeland set	dom. graph	$\mathcal{O}(m^2)$	m	2	1	$2m+2$	$2m^2+2m$
STV	total orders	$\mathcal{O}(mn)$	m	1	$m-1$	$(m-1)(n+1)$	$\frac{(m+2)(m-1)}{2} \cdot (n+1)$

Table 1: Summary of performance characteristics of our MapReduce algorithms.

Winner determination is a central but not the only algorithmic problem considered in computational social choice. Further topics include committee selection, judgment aggregation and problems of fair division. On the other hand, MapReduce is only one of many frameworks proposed for parallel computation. Another research direction is to explore the applicability of other cloud computing technologies, in particular Pregel, as well as the flexibility offered by Apache Spark and GraphX.

Acknowledgments

This work was supported by the Vienna Science and Technology Fund (WWTF) through project ICT12-015, by the Austrian Science Fund projects (FWF):P25207-N23, (FWF):P25518-N23 and (FWF):Y698, by the European Research Council (ERC) under grant number 639945 (AC-CORD) and by the EPSRC programme grant EP/M025268/1.

References

Afrati, F. N.; Sarma, A. D.; Salihoglu, S.; and Ullman, J. D. 2013. Upper and lower bounds on the cost of a map-reduce computation. *Proceedings of the VLDB Endowment* 6(4):277–288.

Bachrach, Y.; Betzler, N.; and Faliszewski, P. 2010. Probabilistic possible winner determination. In *Proceedings of AAAI-10*. AAAI Press.

Bartholdi III, J.; Tovey, C. A.; and Trick, M. A. 1989. Voting schemes for which it can be difficult to tell who won the election. *Social Choice and Welfare* 6(2):157–165.

Betzler, N.; Guo, J.; and Niedermeier, R. 2010. Parameterized computational complexity of dodgson and young elections. *Information and Computation* 208(2):165–177.

Brandt, F.; Brill, M.; and Harrenstein, P. 2014. Extending tournament solutions. In Brodley, C. E., and Stone, P., eds., *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, 580–586. AAAI Press.

Brandt, F.; Brill, M.; and Harrenstein, P. 2016. Tournament solutions. In Brandt, F.; Conitzer, V.; Endriss, U.; Lang, J.; and Procaccia, A., eds., *Handbook of Computational Social Choice*. Cambridge University Press.

Brandt, F.; Fischer, F.; and Harrenstein, P. 2009. The computational complexity of choice sets. *Mathematical Logic Quarterly* 55(4):444–459.

Caragiannis, I.; Kaklamanis, C.; Karanikolas, N.; and Procaccia, A. D. 2014. Socially desirable approximations for Dodgson's voting rule. *ACM Transactions on Algorithms (TALG)* 10(2):6.

Davenport, A., and Kalagnanam, J. 2004. A computational study of the kemeny rule for preference aggregation. In *Proceedings of AAAI-04*, volume 4, 697–702.

Dean, J., and Ghemawat, S. 2008. Mapreduce: simplified data processing on large clusters. *Communications of the ACM* 51(1):107–113.

Dwork, C.; Kumar, R.; Naor, M.; and Sivakumar, D. 2001. Rank aggregation methods for the web. In *Proceedings of WWW-01*, 613–622. ACM Press.

Elberfeld, M.; Stockhusen, C.; and Tantau, T. 2014. On the space and circuit complexity of parameterized problems: Classes and completeness. *Algorithmica* 71(3):661–701.

Greenlaw, R.; Hoover, H. J.; and Ruzzo, W. L. 1995. *Limits to parallel computation: P-completeness theory*. Oxford University Press.

Hemaspaandra, E.; Spakowski, H.; and Vogel, J. 2005. The complexity of Kemeny elections. *Theoretical Computer Science* 349(3):382–391.

Johnson, D. S. 1990. A catalog of complexity classes. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*. 67–161.

Lang, J.; Pini, M. S.; Rossi, F.; Salvagnin, D.; Venable, K. B.; and Walsh, T. 2012. Winner determination in voting trees with incomplete preferences and weighted votes. *Autonomous Agents and Multi-Agent Systems* 25(1):130–157.

Leskovec, J.; Rajaraman, A.; and Ullman, J. D. 2014. *Mining of Massive Datasets, 2nd Ed*. Cambridge University Press.

Sandholm, T. 2002. Algorithm for optimal winner determination in combinatorial auctions. *Artificial intelligence* 135(1):1–54.

Sedgewick, R., and Wayne, K. 2016. *Algorithms (Fourth edition deluxe)*. Addison-Wesley.