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. Our main result are efficient and highly parallel algorithms together with a performance analysis for this problem.

1 Introduction

Winner determination is a central problem in the field of social choice. In recent years, developing algorithms for winner determination has become an active 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 *et al.*, 2010; Bachrach *et al.*, 2010; Lang *et al.*, 2012; Caragiannis *et al.*, 2014].

The classical example of a winner determination problem is a political election, where we are given a number of candidates and a number of votes. Even if the number of votes can be big, the number of candidates is typically small in such elections. Yet, we are facing ever increasing volumes of preference data coming from different sources: Whether a user watches or rates a movie at 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 or rating a product, the sources of these huge numbers of preferences or candidates are not always explicit: 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. With the increasing use of sensors (e.g. through a user's "smart" phone or watch), a user may not even notice that such preference data is generated in the background.

The source of 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). Both sources of huge datasets pose different challenges to the design of algorithms for winner determination. Sequential algorithms and systems for winner determination are not designed 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, as well as 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. In Section 2, we give an introduction to the MapReduce framework. In Section 3, we introduce the necessary background from computational social choice. The main contributions of this paper, given mainly in Sections 4 and 5, are:

- We show that the MapReduce framework can be applied to winner determination in order to cope with huge datasets.
- We present MapReduce algorithms for four concrete voting rules (scoring rules, Copeland, Smith and Schwartz) and investigate the theoretical properties of these algorithms.
- 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.

In Section 6, we shall give concluding remarks, in particular comparing the developed MapReduce algorithms and putting them into context with the results on the limits of parallelizability.

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: The *map-phase*, the *shuffle phase* and the *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 key-value pairs are sent 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; for this example we 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.

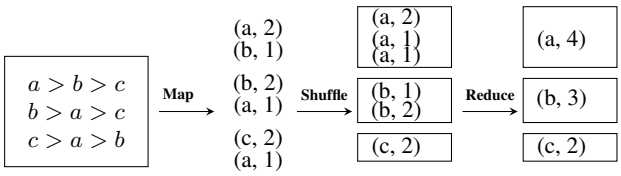

Figure 1: Calculation of scores for the Borda scoring rule.

Analysis of MapReduce algorithms. Various parameters are used to analyze the performance of MapReduce algorithms [Afrati *et al.*, 2013; Leskovec *et al.*, 2014]. By the MapReduce philosophy of splitting the overall computation into small and simple tasks (to be performed in parallel by the reducers), the most 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` statements in our algorithms). We allow input/output actions only of constant size. It is common practice to ignore the input to the map tasks (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.

A parameter 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 aims at measuring the total time needed to complete a MapReduce computation. This corresponds to 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 shall identify the wall clock time with the maximum number of input/output actions (`EMIT`) in any of the computation paths. 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 generalizes to arbitrary scoring rules) yields the following characteristics:

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

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 vote.

We say that 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 in [Brandt *et al.*, 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 *et al.*, 2009]. We refer the reader to [Brandt *et al.*, 2014] and to the handbook chapter of Brandt *et al.* [2016] for an overview on these and other choice sets.

In addition to the aforementioned voting rules we also consider the Single Transferable Vote (STV) rule, which is not based on the dominance relation. We introduce it in Section 5.

4 MapReduce Algorithms

In this section, we propose MapReduce algorithms for "huge" elections. In particular, we present algorithms for determining the Copeland, Smith, and Schwartz sets. Central to these three voting rules is the concept of dominance graph. We thus first present a MapReduce algorithm for computing the (strict or weak) dominance graph.

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 $D_\succ$ to denote the strict dominance graph and $D_\succeq$ for the 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.

The natural first step shared by all MapReduce algorithms presented in this section is an algorithm for computing the (strict or weak) dominance graph. We consider the representation of the dominance graph by an adjacency matrix. Alternative representations of this graph (e.g., by a table of edges) would lead to analogous algorithms. In Algorithm 1, we present a MapReduce algorithm for computing the strict dominance graph. For computing the weak dominance graph, the algorithm is easily adapted by replacing the condition "result > 0" by "result ≥ 0" in the if-statement.

As described earlier, we consider the votes given in the input as preflists. The algorithm utilizes a single round of MapReduce, specified by the map-phase given by MAP and the reduce phase given by REDUCE_SUM, followed by a post-processing step given by COMBINE. In the map-phase, we attribute the value 1 to a key i, j if the voter under consideration prefers candidate i to j; we attribute -1 if j is preferred to i. In the reduce phase, a sum is computed over the values assigned to each reducer – a positive sum indicating that i dominates j and a negative sum indicating that j dominates i. The input values of the reduce phase are fully defined by the EMIT statements of the map phase, so we do not explicitly state the input parameters of each reduce function in our algorithms. Every reduce statement has the same interface, that is REDUCE(key k, list values). In the post-processing step COMBINE, the adjacency matrix D is filled in accordingly.

Algorithm 1 Strict Dominance Graph

```
function MR_DOMINANCE_GRAPH(list preflists)
    MAP(preflists)
    REDUCE_SUM
    COMBINE

function MAP(list preflists)
    for all preflist in preflists do
        for all totalorder in preflist do
            for i=1 to length(totalorder)-1; j=i to length(totalorder) do
                EMIT(key=[totalorder[i], totalorder[j]], value=1)
                EMIT(key=[totalorder[j], totalorder[i]], value=-1)

function REDUCE_SUM(key k, list values)
    return (k, sum(values))

function COMBINE(LIST RESULTS)
    for all ([k1, k2], result) in results do
        if result > 0 then D[k1, k2] = 1 else D[k1, k2] = 0
```

Proposition 2. *Algorithm 1 as well as its variant of computing the weak dominance graph has the following characteristics:* $\mathsf{rr} \leq m$, $\#\mathsf{rounds} = 1$, $\#\mathsf{keys} = m^2$, $\mathsf{wct} \leq n$, *and* $\mathsf{tcc} \leq nm^2$.

While the original input has a size of $\mathcal{O}(nm^2)$, after applying Algorithm 1 we obtain a dominance graph of size $\mathcal{O}(m^2)$. If m^2 is rather small, it might no longer be necessary to use MapReduce algorithms; a conventional sequential algorithm might be able to compute choice sets based on this graph. 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 Copeland Set

Algorithm 2 specifies a MapReduce algorithm that receives the adjacency matrix of the strict dominance graph computed by Algorithm 1 as input and computes the Copeland set. The notation for D given as an adjacency matrix used in this and further algorithms is as follows: By writing "matrix[,] D", we specify that D is a two-dimensional matrix. The notation $D[i,]$ refers to the i-th row of D, while $D[, i]$ refers to the i-th column of D. Although we restrict `EMIT` statements to handle data of constant size, we use `EMIT`(key=i, value=$D[i,]$) to transfer a full row. However, this corresponds to m `EMIT` statements that transfer the row cell by cell.

Algorithm 2 Copeland Set

```
function MR_COPELAND_SET(matrix[,] D)
    MAP_ROWS(D)
    MAP_COLS(D)
    REDUCE_SUM
    FIND_MAX

function MAP_ROWS(matrix[,] D)
    for i=1 to nrows(D) do
        EMIT(key=i, value=D[i,])    # this corresponds to m emits (an entire row)

function MAP_COLS(matrix[,] D)
    for i=1 to ncols(D) do
        EMIT(key=i, value=(−1) · D[, i])    # this corresponds to m emits
```

The main idea of this algorithm is that there is one reducer 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. Note that 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 corresponding row and the (negative) values of the corresponding column. This task is performed by the function REDUCE_SUM specified in Algorithm 1. The sum obtained by each reducer is 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. This algorithm has the following computational properties:

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

Note that this and the following MapReduce algorithms are particularly useful in a situation where m^2 is "large" (e.g., the full dominance graph would be too large to be handled by a single machine – hence non-parallel algorithms would have problems processing it), but m is still manageable (i.e., the set of candidates can still be processed by one of the machines in our cluster or cloud). This is a reasonable assumption to make for most huge datasets, as the set of candidates is rarely larger than in the range of billions.

4.3 Reachability in the Dominance Graph

The first algorithm for computing the Smith set requires reachability queries in the weak dominance graph, i.e., we have to be able to answer whether b is reachable from a. This can be done by repeatedly squaring the adjacency graph of $D_{\succeq}$. Recall that for an adjacency matrix D (assuming that it contains ones in the diagonal), the k-th power of D, D^k, contains a 1 in cell (i, j) if and only if vertex/candidate j can be reached from i in at most k steps. Hence, we can compute the transitive closure of the weak dominance graph by squaring D $\lceil \log_2 m \rceil$ times. Similarly, we can compute the transitive closure of the strict dominance graph – we only have to insert ones in the diagonal before we square it[1].

We use a standard MapReduce algorithm for squaring a matrix, which we refer to as MR_SQUARE_M. Matrix multiplication is a well-studied procedure in the MapReduce literature [Afrati *et al.*, 2013; Leskovec *et al.*, 2014]. The following proposition is implicit in [Afrati *et al.*, 2013]:

Proposition 4. *The MapReduce algorithm* MR_SQUARE_M *for squaring an adjacency matrix has the following characteristics:* $\mathsf{rr} \leq 2m$, $\#\mathsf{rounds} = 1$, $\#\mathsf{keys} = m^2$, $\mathsf{wct} \leq 2m$, *and* $\mathsf{tcc} \leq 2m^3$.

4.4 Computation of the Smith Set

We use the following equivalent characterization of the Smith set by Brandt *et al.* [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 MapReduce algorithm for computing the Smith set would first compute the transitive closure of the weak dominance graph, i.e., compute all distances between candidates. However, it suffices to consider only $D^2_{\succeq}$ and $D^4_{\succeq}$ by applying the following lemma: Brandt *et al.* [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 $t \notin D^2_{\succeq}(v)$. Since $D^2_{\succeq}(v) = D^3_{\succeq}(v)$ implies that $D^3_{\succeq}(v) = D^4_{\succeq}(v)$, we compute only $D^2_{\succeq}$ and $D^4_{\succeq}$.

We refer to Algorithm 3 for a high-level description of a MapReduce algorithm. This MapReduce algorithm consists of four rounds, which we label (MR1) to (MR4).

Algorithm 3 Smith Set

```
function MR_SMITH_SET(matrix[,] D)
    (MR1:) D^2=MR_SQUARE_M(D)
    (MR2:) D^4=MR_SQUARE_M(D^2)
    (MR3:) MAP_ROWS(D^2)
           MAP_ROWS(D^4)
           M = REDUCE_COMPARE
    (MR4:) MAP_COLS(M)
           REDUCE_SMITH_SET

function REDUCE_SMITH_SET(key k, list col)
    for i=1 to length(col) do
        if col[i] ≠ 0 then
            return (k, false)   # candidate k is not contained in the Smith set
    return (k, true)   # candidate k is contained in the Smith set
```

In the first round, we compute D^2, i.e., reachability in at most two steps. We then compute D^4, i.e., reachability in at most four steps. Key to the algorithm is the third round, which is implemented by REDUCE_COMPARE: Reducer i (corresponding to the i-th candidate) receives the i-th row of both D^2 and D^4. It then verifies whether (1) the two rows are equal and (2) there is at least one cell with a 0. If both conditions are met, REDUCE_COMPARE returns the given row; otherwise it does not return anything. We exclude rows consisting solely of ones because then $D^2_{\succeq}(c_i)$ contains all candidates and hence there are no undominated candidates. All rows returned by REDUCE_COMPARE are stored in the matrix M. In the fourth and final MapReduce round we gather all such rows and output a candidate c_j if the j-th column of M consists only of zeros. Note that if $D^2(b) = D^4(b)$ and the size of $D^2(b)$ is smaller than the total number of candidates, then there exist some candidates that are not dominated by the candidates in $D^2(b)$.

Proposition 5. *Algorithm 3 for computing the Smith Set has the following characteristics:* $\mathsf{rr} \leq 2m$, $\#\mathsf{rounds} = 4$, $\#\mathsf{keys} = m^2$, $\mathsf{wct} \leq 7m$, *and* $\mathsf{tcc} \leq 4m^3 + 3m^2$.

Proof idea. The first two rounds carry out the squaring of the weak dominance graph via Algorithm 3. By Proposition 4, $\mathsf{rr} = 2m$, $\mathsf{wct} \leq 2m$, and $\mathsf{tcc} \leq 2m^3$ holds for each of these two rounds. In round MR3, there are m reducers (one for each candidate). Each reducer receives one row from D^2 and one from D^4 and outputs one vector of m values. Hence, we have $\mathsf{rr} = 2$ (with respect to the original input, which was a single matrix, and here we are dealing with two matrices of that size), $\mathsf{wct} \leq 2m$, and $\mathsf{tcc} \leq 2m^2$. Round MR4 yields

[1] Note that if we did not add ones in the diagonal, cell (i, j) in D^k would contain the information whether j is reachable from i in *exactly* k steps.

$\mathsf{rr} = 1$, $\mathsf{wct} \leq m$, and $\mathsf{tcc} \leq m^2$. In total, we get the upper bounds $\mathsf{wct} \leq 7m$ and $\mathsf{tcc} \leq 4m^3 + 3m^2$. □

4.5 Computation of the Schwartz Set

For the Schwartz set we use the following alternative characterization 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 *et al.*, 2009, Lemma 4.5]. In Algorithm 4, we give a MapReduce algorithm for computing the Schwartz set given the strict dominance graph. Unlike the computation of the Smith set, we first have to compute the full transitive closure of the strict dominance graph – making the algorithm require significantly more (namely $\lceil \log_2 m \rceil + 1$) MapReduce rounds.

Algorithm 4 Schwartz Set

```
function MR_SCHWARTZ_SET(matrix[,] D)
    D* = D
    for ⌈log2 m⌉ times do    # can be stopped earlier if fixpoint reached
        D*=MR_SQUARE_M(D*)
    MAP_ROWS(D*)
    MAP_COLS(D*)
    REDUCE_SCHWARTZ_SET

function REDUCE_SCHWARTZ_SET(key k, (list row, list col))
    for i=1 to m do
        if col[i] ≠ 0 ∧ row[i] = 0 then
            return (k, false)    # there is a path from i to k but not vice versa
    return (k, true)
```

In more detail, the algorithm first computes the transitive closure D^* of D in $\lceil \log_2 m \rceil$ rounds. The required number of rounds might be reduced by stopping as soon as a fixpoint is reached (i.e., D^* does not change anymore). Finally, for every candidate a, the outgoing and incoming paths have to be compared in order to check if a belongs to the Schwartz set or not. To this end, the reduce task REDUCE_SCHWARTZ_SET checks for every candidate b if there is a path from a to b, whenever there is a path from b to a.

Proposition 6. *Algorithm 4 for computing the Schwartz Set has these characteristics:* $\mathsf{rr} \leq 2m$, $\#\mathsf{rounds} = \lceil \log_2 m \rceil + 1$, $\#\mathsf{keys} = m^2$, $\mathsf{wct} \leq 2m(\lceil \log_2 m \rceil + 1)$, *and* $\mathsf{tcc} \leq 2m^3(\lceil \log_2 m \rceil + 1)$.

4.6 Computation of the Smith and Schwartz Set from Strongly Connected Components (SCCs)

Our algorithms presented above for computing the Smith and Schwartz set are based on matrix multiplication for computing reachability in the dominance graph. Another approach would be via the strongly connected components (SCCs) of the dominance graph. Recall from Section 2 that the Smith set is the (unique) smallest set of candidates that dominate all outside candidates (or equivalently, the unique minimal undominated SCCs of the weak dominance graph). The Schwartz set is the union of minimal sets that are not dominated by outside candidates (or equivalently, the union of all minimal undominated SCCs of the strict dominance graph).

There are frameworks specifically designed for massively parallel computations on huge graphs, such as Pregel [Malewicz *et al.*, 2010]. Indeed, also for computing strongly connected components in a graph, several Pregel algorithms have been presented [Barnat *et al.*, 2011; Salihoglu and Widom, 2014; Yan *et al.*, 2014]. In this section, we thus assume that we are given the SCCs of the (weak or strict) dominance graph as an input. In Algorithm 5, we give such an algorithm for computing the Smith set based on the SCCs c of the weak dominance graph, as well as the graph D itself. The Schwartz set can be computed by the analogous algorithm applied to the strict dominance graph.

Algorithm 5 Smith and Schwartz Set from SCCs

```
function MR_SCC(matrix[,] D, SCCs c)
    MAP_CELLS(D)
    REDUCE_DEDUPLICATE
    COMBINE_SCC

function MAP_CELLS(matrix[,] D, SCCs c)
    for i=1 to m; j=1 to m do
        if c(i) != c(j) then    # where c(i) returns the SCC of vertex i
            EMIT(key=j, value=false)
```

In more detail, the main goal of the algorithm is to determine whether a given SCC is *not* undominated. In the map step MAP_CELLS, we first determine whether a candidate j is dominated (by a candidate outside of the candidate's SCC), and thus not in an undominated SCC. Consequently, if this is the case, this candidate is emitted. In the reduce step REDUCE_DEDUPLICATE we deduplicate these key-value pairs (as for one candidate j, there could be multiple such (j, false) key-value pairs). Finally, in the post-processing step COMBINE_SCC, since we now know whether a candidate j is dominated, it remains to propagate this information to all other candidates i with $c(i) = c(j)$. All such dominated candidates are not in the Smith (or Schwartz) set, the remaining candidates are in it.

Proposition 7. *Algorithm 5 for computing the Smith or Schwartz set given SCCs has these characteristics:* $\mathsf{rr} = 1$, $\#\mathsf{rounds} = 1$, $\#\mathsf{keys} = m$, $\mathsf{wct} \leq m$, *and* $\mathsf{tcc} \leq m^2$.

5 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 8. 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 *et al.*, 1995]. Given a Boolean circuit with

Problem	Input	Input Size	#keys	rr	#rounds	wct $\leq$	tcc $\leq$
Scoring rules	total orders	$\mathcal{O}(mn)$	m	1	1	n	mn
Dominance graph	partial orders	$\mathcal{O}(nm^2)$	m^2	$\leq m$	1	n	nm^2
Copeland set	dom. graph	$\mathcal{O}(m^2)$	m	2	1	$2m$	$2m^2$
Smith set	dom. graph	$\mathcal{O}(m^2)$	m^2	$\leq 2m$	4	$7m$	$4m^3 + 3m^2$
Schwartz set	dom. graph	$\mathcal{O}(m^2)$	m^2	$\leq 2m$	$\leq \lceil \log_2 m \rceil + 1$	$2m(\lceil \log_2 m \rceil + 1)$	$2m^3(\lceil \log_2 m \rceil + 1)$
Smith / Schwartz	dom. graph, SCCs	$\mathcal{O}(m^2)$	m	1	1	m	m^2
STV	total orders	$\mathcal{O}(mn)$	m	1	$m-1$	$n(m-1)$	$nm(m-1)$

Table 1: Summary of performance characteristics.

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 in such a way that, in the first m rounds, exactly 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 of candidates (m) is the source of P-completeness; the number of voters (n) is not an obstacle to parallelization.

Theorem 9. STV-WINNER *can be solved in* $\mathcal{O}(m + \log(n))$ *space.*

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 9 can also be seen from the perspective of parameterized space complexity [Elberfeld *et al.*, 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 a space complexity of $\mathcal{O}(m \cdot \log(n))$.

To support Theorem 9, 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. Each reducer 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 big m as it requires $m-1$ rounds. However, for small m, this algorithm can be considered feasible – which matches exactly the claims of Theorems 8 and 9. Finally, we state its performance characteristics:

Proposition 10. *For computing STV we obtain the following characteristics:* $\mathsf{rr} = 1$, $\#\mathsf{rounds} = m-1$, $\#\mathsf{keys} = m$, $\mathsf{wct} \leq n(m-1)$, *and* $\mathsf{tcc} \leq nm(m-1)$.

6 Conclusion

This paper presents parallel algorithms for winner determination problems using the popular MapReduce framework. We have analyzed these algorithms with respect to several characteristics. A summary is given in Table 1. Note that some algorithms are based on different inputs than others and thus these characteristics have to be compared with care. The algorithms for computing the Copeland, Smith, and Schwartz sets (Algorithms 2, 3 and 4) are directly comparable and so are the algorithms for scoring rules and STV. For STV, our analysis has shown that the problem is hard to parallelize in general, but allows for efficient parallel computation for elections with few candidates.

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 known to be NP-hard [Bartholdi III *et al.*, 1989; Hemaspaandra *et al.*, 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 and could potentially enable us to deal with huge elections.

Winner determination is a central but not the only algorithmic problem considered in Computational Social Choice. Further possible topics include committee selection, judgement aggregation and problems of fair division. Many computational problems in these domains are known to be computationally hard and thus, as mentioned before, heuristical methods might be necessary to achieve parallelizability.

For other MapReduce algorithms the relationship between the maximum load of reduce tasks and the replication rate has been studied, e.g. [Afrati *et al.*, 2013]. To perform a similar analysis it is necessary to extend our algorithms, such that the maximum load is an input parameter and the algorithms fully exploit the capacity of the reduce tasks. In other words, the algorithms are designed in such a way that the level of parallelization can be tuned using a parameter - the maximum load. Describing such algorithms is subject to future work.

MapReduce is only one of many frameworks proposed for parallel computation. A fourth research direction is to explore the applicability of these different frameworks, in particular Pregel, as well as the flexibility offered by Hadoop 2.

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 (ACCORD) and by the EPSRC programme grant EP/M025268/1.

References

[Afrati *et al.*, 2013] Foto N. Afrati, Anish Das Sarma, Semih Salihoglu, and Jeffrey D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *Proceedings of the VLDB Endowment*, 6(4):277–288, 2013.

[Bachrach *et al.*, 2010] Yoram Bachrach, Nadja Betzler, and Piotr Faliszewski. Probabilistic possible winner determination. In *Proceedings of AAAI-10*. AAAI Press, 2010.

[Barnat *et al.*, 2011] Jiří Barnat, Jakub Chaloupka, and Jaco Van De Pol. Distributed algorithms for SCC decomposition. *Journal of Logic and Computation*, 21(1):23–44, 2011.

[Bartholdi III *et al.*, 1989] John Bartholdi III, Craig A Tovey, and Michael A Trick. Voting schemes for which it can be difficult to tell who won the election. *Social Choice and Welfare*, 6(2):157–165, 1989.

[Betzler *et al.*, 2010] Nadja Betzler, Jiong Guo, and Rolf Niedermeier. Parameterized computational complexity of dodgson and young elections. *Information and Computation*, 208(2):165–177, 2010.

[Brandt *et al.*, 2009] Felix Brandt, Felix Fischer, and Paul Harrenstein. The computational complexity of choice sets. *Mathematical Logic Quarterly*, 55(4):444–459, 2009.

[Brandt *et al.*, 2014] Felix Brandt, Markus Brill, and Paul Harrenstein. Extending tournament solutions. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 580–586. AAAI Press, 2014.

[Brandt *et al.*, 2016] Felix Brandt, Markus Brill, and Paul Harrenstein. Tournament solutions. In Felix Brandt, Vincent Conitzer, Ulle Endriss, Jérôme Lang, and Ariel Procaccia, editors, *Handbook of Computational Social Choice*. Cambridge University Press, 2016.

[Caragiannis *et al.*, 2014] Ioannis Caragiannis, Christos Kaklamanis, Nikos Karanikolas, and Ariel D Procaccia. Socially desirable approximations for Dodgson's voting rule. *ACM Transactions on Algorithms (TALG)*, 10(2):6, 2014.

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

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

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

[Elberfeld *et al.*, 2014] Michael Elberfeld, Christoph Stockhusen, and Till Tantau. On the space and circuit complexity of parameterized problems: Classes and completeness. *Algorithmica*, 71(3):661–701, 2014.

[Greenlaw *et al.*, 1995] Raymond Greenlaw, H James Hoover, and Walter L Ruzzo. *Limits to parallel computation: P-completeness theory*. Oxford University Press, 1995.

[Hemaspaandra *et al.*, 2005] Edith Hemaspaandra, Holger Spakowski, and Jörg Vogel. The complexity of Kemeny elections. *Theoretical Computer Science*, 349(3):382–391, 2005.

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

[Lang *et al.*, 2012] Jérôme Lang, Maria Silvia Pini, Francesca Rossi, Domenico Salvagnin, Kristen Brent Venable, and Toby Walsh. Winner determination in voting trees with incomplete preferences and weighted votes. *Autonomous Agents and Multi-Agent Systems*, 25(1):130–157, 2012.

[Leskovec *et al.*, 2014] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. *Mining of Massive Datasets, 2nd Ed*. Cambridge University Press, 2014.

[Malewicz *et al.*, 2010] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of SIGMOD-10*, pages 135–146. ACM, 2010.

[Salihoglu and Widom, 2014] Semih Salihoglu and Jennifer Widom. Optimizing graph algorithms on pregel-like systems. *Proceedings of VLDB-14*, 7(7):577–588, 2014.

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

[Yan *et al.*, 2014] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of VLDB-14*, 7(14):1821–1832, 2014.