




# The Power of Spreadsheet Computations

Jerzy Tyszkiewicz<sup>(✉)</sup> 

Institute of Informatics, University of Warsaw, Warsaw, Poland  
jty@mimuw.edu.pl

**Abstract.** We investigate the expressive power of spreadsheets. We consider spreadsheets which contain only formulas, and assume that they are small templates, which can be filled to a larger area of the grid to process input data of variable size. Therefore we can compare them to well-known machine models of computation. We consider a number of classes of spreadsheets defined by restrictions on their reference structure. Two of the classes correspond closely to parallel complexity classes: we prove a direct correspondence between the dimensions of the spreadsheet and amount of hardware and time used by a parallel computer to compute the same function. As a tool, we describe spreadsheets which are universal in these classes, i.e. can emulate any other spreadsheet from them. In other cases we provide spreadsheet implementations of a solver for a polynomial-time complete problem, which indicates that the such spreadsheets are unlikely to have efficient parallel evaluation algorithms. Thus we get a picture how the computational power of spreadsheets depends on their dimensions and structure of references.

**Keywords:** Spreadsheets · Expressive power · Lower bounds · Upper bounds · Parallel Random Access Machines · Circuit Value Problem · PTIME · NC

## 1 Introduction

### 1.1 Why Spreadsheets?

Spreadsheets are an extremely popular type of software systems. They have conquered very diverse areas of present day politics, business, research, and, last but not least, our private lives. However, this prevalence is not so evident, because spreadsheets are typically used in the back office and are not presented to the public. They make to the news only when something goes really wrong: for instance a spreadsheet used to justify a widely implemented public policy, as in the case of the extremely influential report [17] concerning a purported causal relationship between high national debt and low economic growth, turns out to contain an error in a formula, affecting the outcome of the calculations [11].

---

**Electronic supplementary material** The online version of this chapter ([https://doi.org/10.1007/978-3-030-48006-6\\_20](https://doi.org/10.1007/978-3-030-48006-6_20)) contains supplementary material, which is available to authorized users.

An *Excel* spreadsheet model was used to manage the investments of JPMorgan Chase & Co. bank, which led to trade losses estimated in billions of dollars [6]. Research is not an exception, and a careful reader of *Science* magazine can read [7, 10], in which a scientific controversy finally turns out to be related to a spreadsheet mistake. Those notable failures indicate the widespread use and critical role of spreadsheets in business and research.

Indeed, spreadsheets are among the most frequently used software tools of any kind. More than 30 years ago after *VisiCalc*, the first spreadsheet and the first killer app in the history of personal computers, still relatively little is known about their computational power.

In recent years there was a significant amount of interest in parallelizing spreadsheet computations, witnessed both by research papers and patent applications [2–4, 12, 14, 19]. In this paper, we analyze the computations expressible in spreadsheets and their relation to parallel complexity classes. Our findings shed light both on parallelization potential of certain structures in spreadsheets, and on the fundamental limitations of this approach.

## 1.2 Measuring the Expressiveness of Spreadsheets

The aim of this paper is to analyze the power of spreadsheets considered as a tool for specifying general-purpose computations. Our analysis is intended to concern the spreadsheet model of computation rather than real-life spreadsheet software. Each spreadsheet is indeed a fully functional program, consisting of many equations (a.k.a. formulas) located in cells, which are computed in data-dependent order, with no side-effects. For mathematical convenience, we assume that the spreadsheet grid is actually infinite and there is no bound on the number of cells with formulas. Next, we assume that the only data type represented in spreadsheets are true, unbounded integers. These two assumptions allow us to apply the methods of computational complexity, which are asymptotic in nature, to the study of spreadsheets. No macros and user-defined functions written in a general programming language are permitted. We also reduce the set of functions permitted in spreadsheets, to keep our analysis manageable. Still, these modifications do not affect the underlying general idea of this model of computation.

Spreadsheets belong to the nonuniform computation models, where for each input size there is a separate computing device. Uniformity can be introduced to such a model by imposing that there is a common, low complexity procedure to create those devices, given the input size. In this respect, spreadsheets come with a natural, built-in tool to do just that: *filling*. It is performed by selecting a rectangular range of cells, clicking a small handle in the lower right corner of it and extending its boundaries either horizontally or vertically, which results in copying the formulas present in the initial range to the new, larger area of the worksheet, with suitable reference adjustments. Filling is the usual way to produce a spreadsheet processing a large amount of data from a few formulas prepared manually, or to extend an already existing one to accommodate a new supply of data.

### 1.3 Technicalities of Spreadsheets

Each cell in the spreadsheet is identified by its column letter and row number, e.g. C2 is located in the third column and second row. A cell may contain a constant value or a formula, which calculates a value of that cell. An example formula  $A\$1+\text{SUM}(\$A\$2:\$A5)$  references a single cell A1 and a (vertical) range A2:A5, consisting of 4 cells in the rectangular spanned by A2 and A5 and its meaning is self-explanatory. The \$ signs indicate how to copy this formula to another cell and do not affect its evaluation. Upon copying, column and row identifiers with \$ in front remain unchanged, while those without are modified to remain in the same relative position to the cell holding the formula as in the original one. If the above formula is copied to another cell two rows down and one column to the right, the copy is  $B\$1+\text{SUM}(\$A\$2:\$A7)$ , i.e., it now references cell B1 and a range A2:A7 consisting of 6 cells.

Finally, the key feature we want to use is *filling*. It is indeed systematic copying of formulas. If, e.g., a range A1:B2 consisting of four cells with formulas is marked and filled down, then the formulas in A3, A5, ... are created by copying (according to the method explained above) the formula from A1; those in B3, B5, ... are copies of the one from B1 etc. In effect, the filled region is covered by  $2 \times 2$  tiles of copies of cells from the original range. Filling can be repeated.

The spreadsheet software automatically chooses an evaluation order of the formulas which follows the references (we do not consider cyclic references).

It is generally quite difficult to describe an algorithm behind a spreadsheet program in plain words. In our case, a small spreadsheet is expanded by filling to an interconnected network of modified copies of itself, resulting in the code to be eventually executed. Its operation involves complex interactions between formulas, their locations which serve as their identifiers, and the mechanism of filling, which produces adjusted references in the newly created cells. Therefore we have decided to provide algorithms in the form of commented spreadsheets in the Electronic Supplementary Material (ESM) of this paper, available from the Publisher. Apart from *Microsoft Excel*, they work also under *LibreOffice*, *Microsoft Excel Online* and *Google sheets*.

### 1.4 Main Results

The structural properties of spreadsheets which we prove to determine their computational properties are defined by restrictions on the pattern of references in formulas, in the sense described above.

**Definition 1.** A spreadsheet  $S$  is *row-organized* iff all its formulas refer to single cells and fragments of rows, only.

Dually,  $S$  is *column-organized* iff all its formulas refer to single cells and fragments of columns, only.

$S$  is *un-organized* iff it is neither row-organized nor column-organized. In each case, references to the inputs of  $S$  are exempt from those limitations.

**Definition 2.** A spreadsheet  $S$  is *row-directed* iff every formula in it refers only to cells and ranges located above itself. Such an  $S$  can be evaluated in any top-down order.

Dually,  $S$  is *column-directed* iff every formula refers only to cells and ranges located to the left of itself. Such an  $S$  can be evaluated in any left-right order.

$S$  is *un-directed* iff it is neither row- nor column-directed.  $S$  is *bi-directed* iff it is both row- and column-directed. Again, references to the input part of  $S$  are exempt from those limitations.

The above properties are preserved by filling.

We describe the computational power of spreadsheets by relating them to Parallel Random Access Machines (PRAM for short), both CRCW priority write and CREW ones.

On several occasions we proceed by implementing instances of the  $P$ -complete *Circuit Value Problem* (abbreviated CVP) in spreadsheets, in order to demonstrate that they are unlikely to have efficient parallel evaluation algorithms.

Our first main result is that any given initial row-organized row-directed spreadsheet can be converted into a program  $\pi$  for an CREW PRAM such that the function computed by that spreadsheet filled to the dimensions of  $c$  columns and  $r$  rows is always the same as that computed by  $\pi$  evaluated on a PRAM with  $c$  processors,  $O(c)$  cells of memory and running for  $O(r \log c)$  time. Thus, if a spreadsheet is row-organized row-directed, its evaluation can be efficiently parallelized: the number of columns contributes only a logarithmic factor to the total computation time. This sets an upper bound on the computational power of row-organized row-directed spreadsheets. An analogous result holds for column-organized column-directed spreadsheets.

In order to get a lower bound, and thus determine the class of functions computable by those spreadsheets, we prove our second main result: there is a row-organized row-directed spreadsheet with 19 formulas which is a universal CRCW PRAM evaluator, i.e., one which given a (suitably encoded) program  $\pi$  together with its input, and filled to the dimensions of  $p$  columns and  $10t$  rows, computes in its last row the description of PRAM after executing  $t$  steps of  $\pi$  on  $p$  processors and with  $p$  cells of shared memory. This demonstrates that spreadsheets can implement a natural and broad class of general-purpose computations.

The same spreadsheet is also a universal row-organized row-directed spreadsheet: any other spreadsheet from this class can be equivalently expressed as a program for a PRAM, which in turn can be executed on that spreadsheet. This above results demonstrate that row-organized row-directed spreadsheets and PRAMs are almost equivalent in computing power, with clear relations between the resources in both models. Indeed, translating a spreadsheet into an equivalent program for PRAM, and then back to spreadsheet, incurs only a logarithmic overhead, a common effect of translations between different parallel computation models.

At the same time PRAMs and spreadsheets are extremely different: the former have only programming primitives and no data analysis ones, while the latter

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	input:	8 input:	7 input:	6 input:	5 input:	4 input:	3 input:	2 input:	1				
2	marker:	8 marker:	1 marker:	1 marker:	1 marker:	1 marker:	1 marker:	1 marker:	1				
3	N:	8 N:	8 N:	8 N:	8 N:	8 N:	8 N:	8 N:	8				
4	i:	0 i:	0 i:	0 i:	0 i:	0 i:	0 i:	0 i:	0				
5	l:	0 l:	0 l:	0 l:	0 l:	0 l:	0 l:	0 l:	0				
6	k:	0 k:	0 k:	0 k:	0 k:	0 k:	0 k:	0 k:	0				
7	addr:	0 addr:	0 addr:	0 addr:	0 addr:	0 addr:	0 addr:	0 addr:	0				
8	val:	0 val:	0 val:	0 val:	0 val:	0 val:	0 val:	0 val:	0				
9	line:	1 line:	1 line:	1 line:	1 line:	1 line:	1 line:	1 line:	1				
10	mem:	0 mem:	0 mem:	0 mem:	0 mem:	0 mem:	0 mem:	0 mem:	0				
11	arg1:	1 arg1:	2 arg1:	3 arg1:	4 arg1:	5 arg1:	6 arg1:	7 arg1:	8				
12	arg2:	0 arg2:	0 arg2:	0 arg2:	0 arg2:	0 arg2:	0 arg2:	0 arg2:	0				
13	arg3:	0 arg3:	0 arg3:	0 arg3:	0 arg3:	0 arg3:	0 arg3:	0 arg3:	0				
14	i:	1 i:	2 i:	3 i:	4 i:	5 i:	6 i:	7 i:	8				
15	l:	0 l:	0 l:	0 l:	0 l:	0 l:	0 l:	0 l:	0				
16	k:	0 k:	0 k:	0 k:	0 k:	0 k:	0 k:	0 k:	0				
17	addr:	0 addr:	0 addr:	0 addr:	0 addr:	0 addr:	0 addr:	0 addr:	0				
18	val:	0 val:	0 val:	0 val:	0 val:	0 val:	0 val:	0 val:	0				
19	line:	2 line:	2 line:	2 line:	2 line:	2 line:	2 line:	2 line:	2				
20	mem:	0 mem:	0 mem:	0 mem:	0 mem:	0 mem:	0 mem:	0 mem:	0				
21	arg1:	0 arg1:	0 arg1:	0 arg1:	0 arg1:	0 arg1:	0 arg1:	0 arg1:	0				
22	arg2:	1 arg2:	2 arg2:	3 arg2:	4 arg2:	5 arg2:	6 arg2:	7 arg2:	8				
23	arg3:	1 arg3:	1 arg3:	1 arg3:	1 arg3:	1 arg3:	1 arg3:	1 arg3:	1				
24	i:	1 i:	2 i:	3 i:	4 i:	5 i:	6 i:	7 i:	8				
25	l:	2 l:	3 l:	3 l:	4 l:	5 l:	6 l:	7 l:	8				
26	k:	0 k:	0 k:	0 k:	0 k:	0 k:	0 k:	0 k:	0				
27	addr:	0 addr:	0 addr:	0 addr:	0 addr:	0 addr:	0 addr:	0 addr:	0				
28	val:	0 val:	0 val:	0 val:	0 val:	0 val:	0 val:	0 val:	0				
29	line:	3 line:	3 line:	3 line:	3 line:	3 line:	3 line:	3 line:	3				
30	mem:	0 mem:	0 mem:	0 mem:	0 mem:	0 mem:	0 mem:	0 mem:	0				
31	arg1:	0 arg1:	0 arg1:	0 arg1:	0 arg1:	0 arg1:	0 arg1:	0 arg1:	0				
32	arg2:	0 arg2:	0 arg2:	0 arg2:	0 arg2:	0 arg2:	0 arg2:	0 arg2:	0				
33	arg3:	1 arg3:	1 arg3:	1 arg3:	1 arg3:	1 arg3:	1 arg3:	1 arg3:	1				
34	i:	1 i:	2 i:	3 i:	4 i:	5 i:	6 i:	7 i:	8				
35	l:	2 l:	3 l:	4 l:	5 l:	6 l:	7 l:	8 l:	9				
36	k:	0 k:	0 k:	0 k:	0 k:	0 k:	0 k:	0 k:	0				
37	addr:	2 addr:	3 addr:	4 addr:	5 addr:	6 addr:	7 addr:	8 addr:	9				
38	val:	1 val:	1 val:	1 val:	1 val:	1 val:	1 val:	1 val:	1				
39	line:	4 line:	4 line:	4 line:	4 line:	4 line:	4 line:	4 line:	4				
40	mem:	0 mem:	1 mem:	1 mem:	1 mem:	1 mem:	1 mem:	1 mem:	1				

**Fig. 1.** PRAM evaluator in a row-organized row-directed spreadsheet S1: structure and mode of operation. Processors are located in columns, and computation time advances downward. A vertical group of 10 cells constitutes a snapshot of a processor at a given time, so that extending computation time by one unit requires filling 10 rows. The filling process is shown in ESM video V1. S1 is provided in ESM.

have only data analysis functions and do not support any form of programming on the level of the spreadsheet itself.

Further, we demonstrate a row-organized but not row-oriented PRAM simulator (ESM spreadsheet S2), significantly more powerful than the one previously described. If it is extended to  $c$  columns and  $r$  rows, computes the description of PRAM after executing  $cr/10p$  steps of  $\pi$  on  $p$  processors and with  $p$  cells of shared memory, where  $p \leq c$  is a part of the input. Thus this PRAM simulator can perform either a parallel or a sequential computation, as instructed in the input, trading off the number of processors and cells of memory for more computation time.

To get the results for other classes of spreadsheets, we implement instances of CVP in them. Each time we do so, we get hypothetical lower bounds on the parallel complexity of evaluating spreadsheets in this class, following from the anticipated but thus far unproven  $NC \subsetneq P$ . The larger instance we implement, the higher the lower bound is.

We summarize our results in Table 1. The highlights are the following:

First, row-oriented but not row-organized spreadsheets have parallel evaluation algorithms with  $c$  processors and of time complexity  $O(r \log cr)$ , similar to those for row-organized row-oriented ones discussed above, except that they need much more memory:  $O(cr)$  instead of  $O(c)$  cells.

**Table 1.** Summary of demonstrated upper and lower bounds for spreadsheet computations, depending on their structure. We disregard small changes depending on whether spreadsheets are row- or column-organized or un-organized, which are discussed in the main text.

	Column directed	Column un-directed
Row directed	Upper bounds $O(r \log cr)$ with $c$ processors and $O(c \log cr)$ with $r$ processors on CREW PRAM No known PRAM simulation $2n$ columns and $4n$ rows implement CVP instance of size $n$	Upper bound $O(r \log cr)$ with $c$ processors on CREW PRAM $c$ columns and $r$ rows simulate CRCW PRAM with $c$ processors and cells of memory for $r/10$ steps 1 column and $n$ rows implement CVP instance of size $n$
Row un-directed	Upper bound $O(c \log cr)$ time on CREW PRAM with $r$ processors $c$ columns and $r$ rows simulate PRAM with $r$ processors and cells of memory for $c/10$ steps $n$ columns and 1 row implement CVP instance of size $n$	Upper bound sequential polynomial time $c$ columns and $r$ rows simulate CRCW PRAM with $p$ processors and cells of memory for $cr/10p$ steps, $p$ is a part of the input $c$ columns and $r$ rows implement CVP instance of size $cr/8$

Second, for spreadsheets which are row-oriented but not column-oriented, and its dual class, one of the dimensions contributes a logarithmic factor to the computation time, while a CVP instance can be encoded in the other dimension, which causes its size to appear as a linear factor in the computation time.

Third, for spreadsheets which are simultaneously row-oriented and column-oriented, one has choice which of the dimensions will contribute a logarithmic factor and which a linear one to the computation time. However, it is unlikely that there is an algorithm polylogarithmic with respect to both dimensions, because if both are large, a large CVP instance can be still encoded.

All the above results taken together give a comprehensive picture of the computing power of spreadsheets without macros. It turns out that this power is strongly influenced by the pattern of references within the spreadsheet, in addition to its size.

## 2 Spreadsheets

As we have already indicated, we assume that the spreadsheet grid is actually infinite and any number of rows and columns can be filled with copies of the initial cells. We never consider spreadsheets of unbounded size: all ranges used have cells as corners, and the size of the spreadsheet is for us the total number of cells which contain formulas or are referenced in formulas. The only data type are true, unbounded integers.

The mechanisms of copying formulas and filling have already been described in Sect. 1.3. Syntactically the present paper is based on *Microsoft Excel* and the reference to syntax and meaning of formulas is the on-line help of *Microsoft Excel* [13]. The ESM software accompanying this paper has been also prepared with *Excel*.

We frequently use names in the ESM spreadsheets. This is a method to assign a name to a frequently used range of cells, and later on use that name in formulas to denote that range. We use it for the sole purpose of making formulas shorter and easier to understand. This method does not increase the computational abilities of spreadsheets.

## 2.1 Functions in Spreadsheets

We use standard arithmetical functions: +, ·, − and /.

The syntax of comparison functions is `value1relvalue2`, where *rel* is any of =, <, >, <=, >=, <>. `value1` and `value2` can be numbers, formulas or cell references to numbers. The result is TRUE if the arguments are in the specified relation, and FALSE otherwise.

Logical functions `AND(value1,value2,...)` and `OR(value1,value2,...)` compute the logical conjunction and disjunction of their arguments, respectively. The flow control functions are the following.

`IF(test,value1,value2)`. If `test` is, refers or evaluates to TRUE, the function returns `value1`, if `test` is, refers or evaluates to FALSE it returns `value2`. In all other cases the result is a #VALUE! error.

`IFERROR(value1,value2)`. This function returns `value2` if `value1` is, refers to or evaluates to any error value, and `value1` otherwise.

`CHOOSE(index-num,value1,value2,...)` is a kind of generalization of IF, because in one formula it allows the choice among up to 29 possible values to be returned. `index-num` specifies which value argument is selected. `index-num` must be a number between 1 and 29, or a formula or reference to a cell containing a number between 1 and 29. If `index-num` is *i*, CHOOSE returns `valuei`.

We use two address functions: `ROW()` and `COLUMN()`, which return the number of row (column, resp.) of the cell in which they are located. In case they are given an argument, a reference to a single cell, they return the row (column, resp.) in which that reference is located.

We also use aggregating functions. We use only their one-dimensional variants: all range arguments must be contiguous fragments of either single rows or single columns.

In `MATCH(lookup-value,lookup-range,match-type)`, `lookup-value` is the value to be found in a range specified by `lookup-range`. `lookup-value` can be a number, a formula or a cell reference to a number. `match-type` in the spreadsheets we create is typically 0 and causes MATCH to find the first value that is exactly equal to `lookup-value` and return its relative position in the `lookup-range`.

If `match-type` is 1, then values in `lookup-range` are assumed to be sorted into an increasing order, and `MATCH` finds the first value that is larger or equal to `lookup-value` and returns its relative position in the `lookup-range`. If `match-type` is  $-1$ , then values in `lookup-range` are assumed to be sorted into a decreasing order, and `MATCH` finds the first value that is smaller or equal to `lookup-value` and returns its relative position in the `lookup-range`.

In all three cases, if no value is found which satisfies the criteria, the result is an error `#N/A!`. In case of `match-type` equal  $\pm 1$  `lookup-value` is larger (smaller, resp.) than all values in the sorted `lookup-range`, the result is the number of the last non-empty cell in `lookup-range`.

In `INDEX(array,num)`, `array` is a range of cells, `num` can be a number, a formula yielding number or a cell reference to a number. The result of the function is a value whose relative position in `array` is given by `num`.

The last function is `SUMIF(criteria-range,criteria,sum-range)`, which computes the sum of all values present in `sum-range`, in the rows/columns, in which `criteria-range` contains a value satisfying `criteria`. The latter argument has the form `relvalue`, where `rel` is any of `=`, `<`, `>`, `<=`, `>=`, `<>` and `value` is a number, formula yielding a number or reference to a single cell. `sum-range` and `criteria-range` must be both horizontal or both vertical. E.g., `SUMIF(A1:A5,<>B1,C1:C5)` sums those values from range `A1:A5` for which the corresponding element in `C1:C5` is not equal to the value in `B1`. This function is treated as a representative of a broad class of spreadsheet functions, which can be treated by methods similar to what we employ below.

## 2.2 Locality

Assume that a small initial spreadsheet  $S$  has been filled (perhaps in several steps) to create a large spreadsheet  $T$ .

If  $k$  is the height (width, resp.) of  $S$ , and a formula in cell  $c$  of  $T$  references a range  $R$ , then each corner of  $R$  is vertically (horizontally, resp.) at most  $k$  rows (columns, resp.) away from either the top row (leftmost column, resp.) due to an absolute reference, or from  $c$  (due to a relative reference).

In particular, in row-organized spreadsheets this very much restricts which rows can be indeed referenced: only those close to the origin and those close to the referencing cell. An analogous property holds for column-organized spreadsheets. We use these *locality properties* several times below.

## 3 PRAM Model

A PRAM machine  $A$  consists of the following components:

- Unbounded number of cells of global read-and-write shared memory, numbered from 1 on, capable of storing one integer.



- Unbounded number of cells of global read-only input memory, analogous to shared memory.
- Unbounded number of processors, numbered from 1 on. Each of them has three private read and write registers  $i, j, k$  for storing integers. Each of them can access the following read-only registers:  $s$  with its own serial number,  $N$  equal to the total number of active processors and  $M$  equal to the number of active cells of shared memory.

The number of private registers can be chosen arbitrarily. We needed some fixed limit, so we have decided to use 3 of them.

- Program  $\pi$ , which is a list of consecutively numbered instructions of the following forms, and where  $x$  is ranging over  $i, j, k$  and  $u, v$  over  $i, j, k, s, N, M, M[i], M[j], M[k], M[s], I[i], I[j], I[k], I[s]$  and integer constants,  $\ell$  ranges over integer constants:

1.  $x := u$
2.  $x := u \circ v$ , where  $\circ$  is among  $\{+, -, *, /\}$
3.  $M[x] := u$
4.  $M[x] := u \circ v$ , where  $\circ$  is among  $\{+, -, *, /\}$
5. **if**  $u < v$  **then goto**  $\ell$

$M[x]$  stands for the shared memory cell whose address is  $x$ , and  $I[x]$  stands for the input memory cell whose address is  $x$ .

The input sequence  $v$  of integers is located in the input memory cells, and  $I[1]$  contains the length of the input sequence, including itself (so that the empty input sequence is passed to the machine as 1 in  $I[1]$ ).

Then a fixed number of its processors (say  $p$ ) is initialized, and a fixed number of shared memory cells (say  $m$ ) is initialized.

During the computation each of the processors follows  $\pi$ , updating its private instruction counter. The state of  $A$  at each time  $t$  of its computation on input  $v$  is defined to be the sequence of  $p$  4-tuples of integers and a sequence of  $m$  integers: the  $n$ -th tuple is the state of the  $n$ -th processor, consisting of: the values of its local variables  $i, j$  and  $k$ , the value of its instruction counter, while the sequence of integers represents the content of the shared memory.

Initially (i.e., at time  $t = 0$ ) the values of local registers are 0, the instruction counter is 1, and the shared memory values are 0.

A single step of computation of  $A$  corresponds to a parallel, simultaneous change of all  $p$  4-tuples and  $m$  integers describing the processors and shared memory of  $A$ .

An attempt to read from a nonexistent input or shared memory cell (i.e., of address higher than  $I[1]$  or than  $p$ , resp.) or to read from cells of numbers smaller than 1 is an error and the result of this operation is unpredictable: it may cause the machine to break and stop operating, or to retrieve some value and continue computation.

An attempt to write to a shared memory cell of zero or negative number is permitted, but has no effect, and similarly if that number is higher than  $p$ . If more than one processor attempts to read from the same shared or input memory cell, all of them succeed and get the same value. If more than one processor attempts

to write to the same shared memory cell, all the requests are executed, and the new value of that memory cell is the one written by the processor with the lowest serial number; the values written by the remaining processors get lost. Reading is performed before writing, so the processors which read from a shared memory cell to which other processors wish to write, get the “old” value.

The way of executing its program by  $A$  is obvious, with the provision that if the value of the instruction counter becomes higher than the number of lines in the program, the processor halts. Thus, given a PRAM  $A$  as above and its input vector  $v$ , the computation of  $M$  on  $v$  is represented by a finite or infinite sequence of states of  $A$ , which may but need not be constant from some moment on. The result of computation of  $A$  after  $n$  steps is the content of the shared memory after completing that step.

Another, substantially weaker model of PRAM is CREW, which results from CRCW by forbidding concurrent writes altogether.

The programming language of our PRAM machines is extremely simple, but, as it is well-known, equivalent in computing power to even very rich ones, so indeed each processor separately has a universal computing power, equivalent to that of a Turing machine. PRAM is a machine which can easily implement referential data structures, such as lists, trees, etc., as well as arrays. Therefore we use them without any further explanation.

## 4 Complexity Theory

In this paper we use the  $P$ -complete problem *Circuit Value Problem* (abbreviated CVP). An instance of CVP is a sequence of  $n$  Boolean substitutions (the reason for starting numbering from 2 is purely technical and explained below):

$$p_2 := \text{conn}_2(\text{inputs}_2); p_3 := \text{conn}_3(\text{inputs}_3); \dots p_n := \text{conn}_n(\text{inputs}_n).$$

The connectives  $\text{conn}_i$  can be binary **and** and **or** and unary **not**. Each of the inputs in  $\text{inputs}_k$  can be either *true*, or *false*, or a variable  $p_i$  with  $i < k$ , indicating that the value of that variable should be used.

The CVP problem is that, given an instance of CVP, to decide if the last variable is *true*. This problem is known to be  $P$ -complete. We encode CVP in various spreadsheets in order to demonstrate that they are unlikely to be efficiently parallelizable. For convenience, when we do so we use 0 in place of *false*, 1 in place of *true*, for variables we use their numbers as names (we have started numbering from 2, so this does not lead to confusing truth values with variables), and we drop all conventional symbols like  $:=$ , parentheses and commas, so that an example instance of CVP

$$p_2 := \mathbf{and}(true, false); p_3 := \mathbf{or}(p_2, false); p_4 := \mathbf{not}(p_3); p_5 := \mathbf{or}(p_4, p_3)$$

is encoded by

$$\mathbf{and} \ 1 \ 0; \ \mathbf{or} \ 2 \ 0; \ \mathbf{not} \ 3; \ \mathbf{or} \ 4, \ 3$$

In this paper we estimate the size of CVP instances, which are computable in the spreadsheets in question. Each time it is easy to translate the size of CVP instances we produce into the potential lower bounds.

## 5 Un-directed Spreadsheets I: Complexity

It is obvious, that if the cells of a small initial spreadsheet  $S$  are filled to create  $c$  columns and  $r$  rows of formulas, then the resulting spreadsheet can be computed in time polynomial in  $cr$ , given the initial  $S$ , the dimensions  $c, r$  and the input data of  $S$ . The following theorem is neither surprising nor difficult to demonstrate. It implies that evaluating spreadsheets is  $P$ -complete.

**Theorem 1.** *There exists an un-directed, un-organized spreadsheet  $S_4$ , such that when it is extended to dimensions of either  $n$  rows and 6 columns or 6 rows and  $n$  columns, it computes the solution to the CVP problem of size  $n$ , given its description as input.*

*Proof.* A fully commented spreadsheet  $S_4$  is provided in ESM. It consists of two functionally separate fragments, which can be independently converted into row-oriented and column-oriented structure.  $\square$

## 6 Directed Spreadsheets I: Simulating Spreadsheet by PRAM

In this section, we are going to formulate and prove a theorem about evaluating spreadsheets by PRAM machines. In our model spreadsheets can be of unbounded size, so we can use asymptotic notation to describe the resources needed by a PRAM to execute a spreadsheet of a given size. The theorem below is formulated for row-organized row-directed spreadsheets. Its dual form for column-organized column-directed spreadsheets holds, too.

**Theorem 2.** *For any row-directed spreadsheet  $S$  with input data, there exists a program  $\pi$  for CREW PRAM, such that if that spreadsheet is filled to make  $c$  columns and  $r$  rows, the values of all its cells can be computed by  $\pi$  run for  $O(r \log cr)$  time on  $c$  processors and  $cr$  cells of memory, given the initial  $S$ ,  $c$ ,  $r$  and the input data of  $S$ .*

*If  $S$  is additionally row-organized, then the values of the cells in the last row can be computed by  $\pi$  run for  $O(r \log c)$  steps on a PRAM with  $c$  processors and  $c$  cells of memory.*

*Proof.* For each column of the spreadsheet we designate one processor, which will be responsible for it. Let the serial number of that processor be equal to the number of the column. The computation of PRAM will be organized into in rounds, where each round corresponds to computing the next row. The codes for evaluating particular formulas are hard-coded into the program  $\pi$ .

For each round we assume that certain auxiliary data structures are available, which enable evaluating aggregating functions efficiently. During each round, first the new values are computed, and then these structures are updated, so that they include the cells in the newly created row, as well. Separately, we must explain how the auxiliary data structures are initialized before the first round.

*Auxiliary Data Structures.* We assume that during each round all the previously created columns and rows are stored in two copies. Each row is stored in several copies:

1. for INDEX: in the original form,
2. for MATCH: as a sorted array, where we sort and store two-element records consisting of the value from the original row and its original address,
3. for SUMIF: for every subset of already existing rows, which potentially may play hold `sum-range` and `criteria-range` in each call of SUMIF, the records formed from the corresponding elements in these two ranges are sorted according to the keys in the `criteria-range`, and then prefix sums are computed from the `sum-range` values.

Each column is stored in similar copies:

1. for INDEX: in the original form,
2. for MATCH: as a balanced binary search tree, in which we store two-element records consisting of the value from the original row (which is the key) and its original address.
3. for SUMIF: for every subset of already existing columns, which potentially may hold `sum-range` and `criteria-range` in a call of SUMIF, the records formed from the corresponding elements in these two ranges are formed. They are stored in a balanced binary search tree, with keys taken from the `criteria-range`, and each node additionally stores the sum of the values from `sum-range` in the subtree rooted at this node.

The key observation is that by locality properties described in Sect. 2.2, there is a bounded number of possible pairs of columns (rows, resp.) that must be indexed using prefix sums and binary search trees.

*Initialization of Auxiliary Data Structures.* The size of the initial spreadsheet with input data is fixed, so this initialization takes constant time and requires constant amount of memory.

*Execution of a Round. Computing Formulas.* Henceforth, PRAM must first evaluate formulas. Due to the row-oriented structure of the spreadsheet, the formulas to be computed refer only to data above themselves, so all of them can be evaluated independently in parallel. For each of the cells, it is done by a single processor, responsible for the column of that cell. The values of all functions except MATCH, INDEX and SUM can be obviously evaluated in constant number of steps. Note that COLUMN function can be evaluated, because each processor knows its serial number, equal to the column number. ROW on the other hand is evaluated by clocking the advancing computation time.

If MATCH looks up a row, a sorted version of that row is in the auxiliary data structures. The processor can find there the suitable value (and its accompanying address to be returned) using binary search in time  $O(\log c)$ .

If MATCH looks up a column, a binary search tree version of that column is in the auxiliary data structures, in which the processor can find the suitable value (and its accompanying address, which should be returned) in time  $O(\log r)$ .

INDEX calls require retrieving a value of a known location from a horizontal or vertical range, so after recomputing the address to be relative to the complete row (column, resp.), the values are retrieved from the auxiliary data structures.

Each SUMIF call requires summing values from a horizontal or vertical range in the auxiliary data structure, as specified by a constraint regarding the values in **criteria-range**. Identifying its boundaries is done by binary search using the key fields, and then the sum can be found using two accesses to the fields with prefix sums.

In total, computing the new values takes  $O(\log c + \log r) = O(\log cr)$  time.

*Execution of a Round. Updating Auxiliary Data Structures.* Updating auxiliary data structures requires sorting several rows of values (including the newly created row) combined with creating their accompanying prefix sums, and inserting new records into the binary search trees holding data about columns. The former can be performed in  $O(\log c)$  time using logarithmic time linear memory sorting employing all  $c$  processors, like the one described in [9, Section 5.2], and the prefix sums can be then computed by the algorithm described in [8, Section 30.1.2].

Then all processors in parallel insert the new values from their columns into the corresponding trees and update sums, in time  $O(\log r)$ .

In total, updating the necessary data structures takes  $O(\log c + \log r) = O(\log cr)$  time.

*Cost of the Algorithm.* First, initialization of auxiliary data structures takes constant time. Then the PRAM computation performs  $r$  rounds, each of them takes  $O(\log cr)$  time, so the total time is  $O(r \log cr)$ . The memory used is constant times  $cr$ .

For the second claim, in a row-organized spreadsheet there is no need to access columns, so we do not need to maintain their auxiliary data structures. Thus in this modified version each round can be completed in  $O(\log c)$  time and the total running time is  $O(r \log c)$ . By locality properties described in Sect. 2.2, we need to store constantly many rows simultaneously, and therefore the total amount of necessary memory is  $O(c)$ .  $\square$

## 7 Directed Spreadsheets II: Simulating PRAM by Spreadsheet

At this point, we have demonstrated that directed spreadsheets can be evaluated by quite efficient parallel algorithms, establishing thereby an upper bound on their expressive power. The question of lower bounds arises naturally.

In order to provide an answer, we are going to demonstrate now that there exists a spreadsheet program, with the following property: any given CRCW PRAM  $A$  can be simulated by a row-organized row-oriented spreadsheet, if suitably encoded in the form of spreadsheet data.

**Theorem 3.** *There exists a row-organized row-directed spreadsheet, which is able to simulate any PRAM  $A$  for which  $p = m$ , so that columns correspond to processors of  $A$  and rows correspond to computation time.*

*Precisely speaking, there exists a single spreadsheet  $S1$  consisting of 19 cells ( $A2$  to  $A20$ ) with formulas, one row for input and a separate input area for representation of a program, such that for every CRCW PRAM  $A$  with program  $\pi$ ,  $p$  processors and  $p$  cells of shared memory, and for every input vector  $v$  for  $A$ , if one*

1. *pastes the encoding of  $\pi$  into the program area of  $S1$*
2. *marks and fills the initial range  $A2:A20$  to the right creating  $p$  columns (corresponding to the processors and shared memory cells of  $A$ )*
3. *selects the rows from 11 to 20 of these  $p$  columns and fills downward so that the bottom row of the new range is  $10t + 10$ ,*
4. *pastes into  $S1$  the input vector  $v$  of  $A$  in the first row,*

*then the cells of the bottom 10 rows compute the state of  $A$  after  $t$  steps of computation on  $v$ .*

*This means, that the spreadsheet created from  $S1$  in steps 1, 2 and 3 performs the first  $t$  steps of the computation of  $A$  on every input, i.e., it simulates  $A$ .*

*Proof.* ESM spreadsheet  $S1$  is the implementation of PRAM in a spreadsheet with explanation of the formulas used for that purpose, and is depicted in Fig. 1. Below we highlight the main elements of the construction of  $S1$ .

Conceptually, the idea is to make a spreadsheet which computes the sequence of configurations of the run of  $A$  on its input. Time is advancing downward and configurations are horizontal blocks of 10 rows each. Each column corresponds to one processor and one cell of shared memory, and formulas located there take care of advancing the computation and handling read and write operations. We may conveniently assume that the processor has been made responsible for operating its associated shared memory cell.

PRAM is a machine with random access. To the contrary, in a spreadsheet cells (which can be thought of as simple processors) can do random read, but are allowed to write only to the memory cell they are associated with. Therefore we have to simulate random writes by other means.

The idea is that any processor willing to write its contents to some shared memory cell, has to announce this in a globally visible location, indicating the address to which it attempts to write and the value to be written. Then all processors use function `MATCH` to search among the announcements for writes to their shared memory cells, and if there is one, fetch the value to be written from the leftmost one using `INDEX`. This conforms to the priority write CRCW conflict resolution policy.  $\square$

Apart from ESM spreadsheet  $S1$ , we also provide its minor variant  $S5$ , permitting structural programming with `while-endwhile` and `if-endif` rather than `goto` jump instructions.

The construction in  $S1$  provides answer for our questions:

1. There is not much room for improvement over Theorem 2. Indeed, it offers  $O(r \log cr)$  and  $O(r \log c)$  algorithm for PRAM with  $c$  processors and  $O(c)$  cells of memory, when the spreadsheet is row-directed and row-organized. On the other hand, every PRAM computation taking time  $t$ , using  $c$  processors and  $c$  cells of memory can be simulated in a row-directed and row-organized spreadsheet with  $c$  columns and about  $10t$  rows.
2. The class of computations expressible in row-directed and row-organized spreadsheets is indeed very rich, and it includes a natural parallel complexity class.

It is worth noting, that according to the results already proven, we have the following.

**Corollary 1.** *S1 is a universal row-oriented row-organized spreadsheet.*

*Proof.* Given a row-oriented row-organized spreadsheet  $S$ , one can derive an CREW PRAM program  $\pi$ , computing the same function as  $S$ . This  $\pi$  can be encoded and provided as (a part of) input to S1, which can execute it.

If the initial  $S$  has  $c$  columns and  $r$  rows, then  $\pi$  should be run on a PRAM with  $O(c)$  processors and memory cells and  $O(r \log c)$  time. Then, in order to simulate it, S1 needs  $O(c)$  columns and  $O(r \log c)$  rows.

Thus the overhead of simulating a spreadsheet by the universal one is logarithmic, typical for other universal devices.  $\square$

The interest in the corollary is that S1 uses very few functions, in particular does not use SUMIF. Therefore this result indicates that the function set with MATCH and INDEX as the only aggregating functions might form a kind of core of the spreadsheet language of formulas, at least for the row-oriented row-organized ones. Thus, in an attempt to create a theoretical model of spreadsheets, this restricted function set appears as a candidate to be the set of basic operations, from which the remaining ones can be defined. It would be very much like the relational algebra and its role in the theoretical formalization of relational databases. Of course, it concerns only row-oriented row-organized spreadsheets created by filling.

## 8 Bi-directed Spreadsheets: Complexity

A bi-directed, column organized spreadsheet extended to dimensions  $r$  and  $c$  can be, according to Theorem 2, evaluated on a PRAM using  $O(cr)$  cells of memory and

1.  $O(r)$  processors in time  $O(c \log r)$ , if treated as column-organized column-directed.
2.  $O(c)$  processors and in time  $O(r \log cr)$ , if treated as row-directed.

One might be tempted to believe that it is possible to combine somehow those two methods together to yield a parallel evaluation algorithm of even better time complexity. However, we prove below that there is no evaluation algorithm of  $O(\log^{O(1)} cr)$  time complexity unless  $P = NC$ .

**Theorem 4.** *There exists a bi-directed, column-organized spreadsheet S3, such that when it is extended to dimensions of  $3n$  rows and  $8n$  columns, it computes the solution to any CVP instance of size  $n$ , given its description as input.*

*Proof.* The main idea is to implement CVP “diagonally”, and a fully commented implementation is provided as ESM spreadsheet S3.  $\square$

Bi-directed spreadsheets are clearly more restrictive than those which are directed in one dimension only. Author’s personal experience from the development of S3 is that the bi-directed structure is quite unnatural, especially in the column-organized version. Otherwise very simple computation of CVP required a significant effort to be programmed. At the same time this structure does not seem to offer any noticeable advantage in terms of complexity of evaluation.

## 9 Un-directed Spreadsheets II: What Can They Compute?

After a successful implementation of a PRAM in a row-directed spreadsheet and demonstrating that a large class of PRAM computations can be expressed in spreadsheets, it seems natural to attempt a similar goal for un-directed ones, too.

We demonstrate below, that one can create an un-directed row-organized spreadsheet which implements PRAM in a much more flexible way than the row-directed row-organized one.

**Theorem 5.** *There exists a row-organized (but not row-directed) spreadsheet S2 consisting of 21 cells (A2 to A22) with formulas, one row for input and a separate input area for representation of a program (including a value of  $p$ ), such that for every CRCW PRAM  $A$  with program  $\pi$  for every input vector  $v$ , if one*

1. pastes the encoding of  $\pi$  into the program area of S2
2. marks and fills the initial range A2:A22 to the right for  $q$  columns,
3. selects rows from 13 to 22 of these  $q$  columns and fills downward so that the bottom row of the new range is  $10t + 12$ ,
4. pastes into S2 the input vector  $v$  of  $A$  in the first row;
5. inserts a number into the input cell  $p$

*then the cells at the intersection of the bottom 10 rows with  $p$  columns of numbers from  $q - p - q \pmod{p}$  to  $q - q \pmod{p}$  compute the state of  $A$  after  $t * (q/p - 1)$  steps of computation on  $v$ .*

Informally, the spreadsheet S2 above is able to simulate any PRAM  $A$ , for which  $p = m$  in such a way, that filled to  $10t$  rows and  $q \geq p$  columns, it can utilize this computation area to encode a PRAM with  $p$  processors and  $p$  cells of shared memory, running for  $tq/p$  time. Moreover, the parameter  $p$  is a part of the input, so only the whole input specifies, how many processors will be used in the computation. In particular, for  $p = 1$ , this results in a fully sequential computation of length  $tq$ .



*Proof.* The commented spreadsheet is provided as ESM spreadsheet S2. It is recommended that the reader first analyzes S1, on which S2 is based.  $\square$

It is instructive to compare spreadsheet S4 mentioned in Sect. 5 with the present S2. It seems at the first glance that the former is a special case of the latter. However, it is not the case. During the whole computation expressed in S4 it is always possible to refer to the values computed in the past, no matter how distant. In S2 the simulated PRAM can only refer to the values computed in the previous step of simulation. This indicates the difficulty of describing the computations of a spreadsheet by a machine model. In a spreadsheet, every cell is immutable, but its value remains accessible forever. In typical machine models of computation, memory locations are mutable and write operations delete their previous contents.

## 10 Summary and Related Research

We have investigated spreadsheets as a class of algorithms, by assuming that they are small templates, which are filled to a larger area of the grid to process data of variable size.

Under this scenario we have identified simple structural properties of spreadsheets, defined in the terms of the pattern of references between cells, which determine the complexity of the expressible computations.

In this paper, we analyze the computations expressible in spreadsheets and their relation to parallel complexity classes. Our findings shed light both on parallelization potential of certain structures in spreadsheets, and on the fundamental limitations of this approach. We have already mentioned research on parallelizing spreadsheet computations [2–4, 12, 14, 19]. There was very little previous research on lower bounds of the computational power of spreadsheets, although already [5] observed that they have universal computing power. The papers [1], [15] and [16] demonstrate simulations of various algorithms and models of computation using spreadsheets, but without any intent to estimate the full power of this computation paradigm. Paper [20] demonstrates how to implement relational algebra queries and several other general algorithms in spreadsheets (which turn out to be column-organized, but not necessarily column-oriented). [18] presents an implementation of a subset of Java in a spreadsheet, but without considering parallelism, which is the core topic of the present paper.

**Acknowledgments.** Research funded by Polish National Science Centre (Narodowe Centrum Nauki). The author wishes to thank Jacek Sroka and Aleksy Schubert for many valuable discussions on the topic.

## References

1. Bernstein, M.: Using spreadsheet languages to understand sequence analysis algorithms. *Comput. Appl. Biosci.* **3**, 217–221 (1987)

2. Biermann, F.: Data-Parallel Spreadsheet Programming, Ph.D. thesis. IT University of Copenhagen, Computer Science (2018)
3. Bock, A.A.: Static partitioning of spreadsheets for parallel execution. In: Alferes, J.J., Johansson, M. (eds.) PADL 2019. LNCS, vol. 11372, pp. 221–237. Springer, Cham (2019). [https://doi.org/10.1007/978-3-030-05998-9\\_14](https://doi.org/10.1007/978-3-030-05998-9_14)
4. Bock, A.A., Biermann, F.: Puncalc: task-based parallelism and speculative reevaluation in spreadsheets. *J. Supercomput.* 1–21 (2019). <https://doi.org/10.1007/s11227-019-02823-8>
5. Casimir, R.J.: Real programmers don't use spreadsheets. *SIGPLAN Not.* **27**(6), 10–16 (1992)
6. Cavanagh, M.: JPMorgan Chase & Co: Report of JPMorgan Chase & Co., Management Task Force Regarding 2012 CIO Losses (2012)
7. Chesler, E.J., et al.: In silico mapping of mouse quantitative trait loci. *Science* **294**(5551), 2423 (2001). In Technical Comments
8. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press, Cambridge (2001)
9. Gibbons, A., Rytter, W.: Efficient Parallel Algorithms. Cambridge University Press, Cambridge (1988)
10. Grupe, A., et al.: In silico mapping of complex disease-related traits in mice. *Science* **292**(5523), 1915–1918 (2001)
11. Herndon, T., Ash, M., Pollin, R.: Does high public debt consistently stifle economic growth? A critique of Reinhart and Rogoff. *Camb. J. Econ.* **38**(2), 257–279 (2014)
12. Kulkarni, S.G., Wierer, J.J., Xu, M.: Calculation of spreadsheet data. US Patent 8,006,175, August 2011
13. Microsoft Corp.: Excel help. <http://office.microsoft.com/en-us/excel-help/>
14. Olkin, T.M.: Threading spreadsheet calculations. US Patent 10289672, May 2019
15. Premachandra, I.M.: Modeling a turing machine on a spreadsheet: a learning tool. *Int. J. Inf. Manag. Sci.* **4**(2), 81–92 (1993)
16. Rautama, E., Sutinen, E., Tarhio, J.: Excel as an algorithm animation environment. In: Proceedings of the 2nd Conference on Integrating Technology into Computer Science Education, ITiCSE 1997, pp. 24–26. ACM, New York (1997)
17. Reinhart, C.M., Rogoff, K.S.: Growth in a time of debt. *Am. Econ. Rev.* **100**(2), 573–578 (2010)
18. Schubert, A., Sroka, J., Tyszkiewicz, J.: Systematic programming in a spreadsheet. In: IS-EUD 2017 6th International Symposium on End-User Development, pp. 10–17. Eindhoven University of Technology (2017)
19. Sroka, J., Leśniewski, A., Kowaluk, M., Stencel, K., Tyszkiewicz, J.: Towards minimal algorithms for big data analytics with spreadsheets. In: Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, p. 1. ACM (2017)
20. Sroka, J., Panasiuk, A., Stencel, K., Tyszkiewicz, J.: Translating relational queries into spreadsheets. *IEEE Trans. Knowl. Data Eng.* **27**(8), 2291–2303 (2015)