Static Partitioning of Spreadsheets for Parallel Execution

Alexander Asp Bock[0000−0003−3792−0807]⋆

IT University of Copenhagen, Copenhagen, Denmark
Rued Langgaards Vej 4
albo@itu.dk

Abstract. Spreadsheets are popular tools for end-user development and complex modelling but can suffer from poor performance. We present an iterative, greedy algorithm for automatically partitioning spreadsheets into load-balanced, acyclic groups of cells that can be scheduled to run on shared-memory multicore processors. A big-step cost semantics for the spreadsheet formula language is used to estimate work and guide partitioning. The algorithm does not require end-users to modify the spreadsheet in any way. We implement three extensions to the algorithm for further accelerating computation; two of which recognise common cell structures known as cell arrays that naturally express a degree of parallelism. To the best of our knowledge, no such automatic algorithm has previously been proposed for partitioning spreadsheets. We report a maximum 24-fold speed-up on 48 cores.

Keywords: Spreadsheets · Partitioning · Parallelism

1 Introduction

Spreadsheets are popular tools for end-user development, modelling and education. Spreadsheet end-users are usually domain experts but are seldom IT professionals. They create and maintain large, complex spreadsheets over several years [14] and this complexity often leads to errors [10] and poor performance [19].

In recent years, multicore processors have become ubiquitous. For spreadsheet end-users to benefit from this powerful hardware, they should have a tool at their disposal to automatically identify and exploit parallelism in their spreadsheets. Spreadsheets lend themselves well to parallelization as they are first-order functional languages.

In this paper, we present an iterative, greedy algorithm for automatically and statically partitioning a spreadsheet into load-balanced, acyclic groups of cells. Partitioning is guided by a cost model based on a big-step semantics to estimate the work of cells and to produce well-balanced partitions. The partition can be seen as a condensation of the cell dependency graph. We implement the algorithm in the research spreadsheet application Funcalc [18] and we believe this is the first thorough investigation of static partitioning of spreadsheets.

⋆ Supported by the Independent Research Fund Denmark

2 Related Work

Spreadsheet research has primarily focused on error detection, handling and mitigation [6] and less on parallelization. In this section, we briefly discuss some of the research relevant to spreadsheet parallelism.

There exist multiple distributed systems for spreadsheet calculation like ActiveSheets [3], Nimrod [2] and HPC Services for Excel [16]. All three systems require manual modification of the spreadsheet which may take a substantial amount of time or require help from experts.

In his 1996 dissertation, Wack [21] investigated parallelization of spreadsheet programs using distributed systems and an associated machine model. He partitioned and scheduled a set of predefined patterns and parallelized them via message-passing in a network of work stations. Our algorithm does not rely on pre-defined patterns or an existing network of work stations but instead targets shared-memory multicore processors.

Biermann et al. [5] rewrote so-called *cell arrays* to higher-order, compiled function calls on arrays completely transparent to end-users. Their approach parallelized the internal evaluation of each rewritten array but evaluated disjoint cell arrays sequentially.

LibreOffice Calc automatically compiled data-parallel expressions into OpenCL kernels that execute on AMD GPU's [20]. They reported a 500-fold speed-up for a particular spreadsheet.

In recent work [4], we presented a task-based parallel spreadsheet interpreter, dubbed Puncalc, that automatically discovers parallelism and finds cyclic references in parallel. The system targets shared-memory multiprocessors and does not require modification of the spreadsheet. The algorithm obtained roughly a 16-fold speed-up on 48 cores on the same set of benchmark spreadsheets used in this paper. Puncalc is a dynamic algorithm that may not distribute work as well as the static approach presented here.

Our static partitioning algorithm is primarily inspired by the work of Sarkar et al. [17] on the first-order functional language SISAL which was intended to supersede Fortran as the primary language for scientific computing. Sarkar worked on an optimising compiler that automatically extracted parallelism by analysing an intermediate graph representation of the program. The program was then partitioned at compile-time and scheduled onto available processors at runtime. SISAL programs were shown to run on par with Fortran programs on a contemporary supercomputer [8].

3 Contributions

We present the following key contributions:

1. A cost model based on a big-step cost semantics for Funcalc's formula language (section 5).
2. An algorithm for statically partitioning spreadsheets and scheduling them on shared-memory multicore processors (section 6).

3. Three extensions to the algorithm for further accelerating execution of the partitioned spreadsheet. Two of the extensions exploit common cell structures known as *cell arrays* that naturally express some degree of parallelism (sections 7.1 to 7.3).

4 Background: Spreadsheet Concepts

We now introduce some basic spreadsheet concepts deemed necessary for reading this paper. Readers already familiar with the subject can skip this section while those interested in learning more are encouraged to read [18].

4.1 Formulas and Cell References

A cell can contain either a constant, such as a number, string or error (e.g. `#NA` or `#DIV/0!`); or a formula expression denoted by a leading equals character (e.g. `=1+2`). Each cell has a unique address denoted by its column and row where columns start at `A` and rows at 1. Formulas can refer to other cells by using their addresses, or they can refer to an area of cells using the addresses of two corner cells separated by a colon. For example, cell `C1` in fig. 1a refers to the cells `A1` and `A2` while cell `C3` refers to the cell area spanned by the cells `A1` and `B2`.

Cell references may be relative or absolute in each dimension. Relative references refer to other cells using offsets, so the referenced cell depends on the position of the referring cell. Absolute references do not change and are prefixed by a dollar sign. For example, the formula `=$A2` in cell `C2` refers absolutely to column `A` but relatively to row 2, so copying it to cell `C3` would change the cell reference to `$A3`. Copying it to cell `D2` would not change the reference since the column is absolute. This reference scheme is called the A1 format. Relative references are more clearly expressed in the R1C1 format where relative references are denoted by square brackets containing an offset and absolute references are denoted by the absence of square brackets and an absolute row or column number. If the referred cell is in the same row or column, the offset can be omitted, but we explicitly use a zero offset for clarity. The same spreadsheet is shown in fig. 1b in R1C1 format.

	A	B	C
1	10	20	=A1+A2
2	30	40	=$A2
3			=A1:B2

(a)

	A	B	C
1	10	20	=R[+0]C[-2]+R[+1]C[-2]
2	30	40	=R[+0]C1
3			=R[+0]C[-3]:R[+1]C[-2]

(b)

Fig. 1: An example spreadsheet in A1 and R1C1 reference formats.

4.2 The Support and Dependency Graphs

Cell references establish a cell *dependency graph*. Its inverse, the *support graph*, captures cell *support* and is analogous to a dataflow graph where nodes are cells and data flows along the edges from dependent cells to supported cells. Cell `C1` in fig. 1a depends on `A1` and `A2` while both `A1` and `A2` support `C1`. Both the dependency and support graphs may be cyclic.

4.3 Recalculation

There are two major types of recalculation. *Full recalculation* unconditionally reevaluates all formula cells. *Minimal recalculation* only reevaluates the transitive closure of cells reachable, via the support graph, from cells modified by the user. In fig. 1a, whenever a user edits the value in `A1`, cells `C1` and `C3` must be updated to reflect the change. The static partitioning algorithm considers all formula cells in the spreadsheet and thus performs a full recalculation.

4.4 Cell Arrays

Also known as copy-equivalent formulas [12] or cp-similar cells [1], cell arrays [9] denote a contiguous rectangular area of formulas that share the same formula expression in R1C1 format and thus the same computational semantics [9]. A cell array consisting of 3 rows and 1 column in column `B` is shown in fig. 2a in A1 format and in fig. 2b in R1C1 format. The latter format clearly shows that the formulas in the cell array share a common expression.

Cell arrays are common in spreadsheets because they describe bulk operations on collections of cells similar to e.g. `map` and `reduce` on arrays in functional programming. These bulk operations can usually be parallelized as we shall see later. For example, the cell array in fig. 2a effectively describes a `map` operation on column `A`. Dou et al. [9] found that 69% (7416 out of 10 754) of spreadsheets containing formulas from the EUSES [11] and Enron [13] corpora also contained cell arrays, and that they contained on average 80 cell arrays each. The benchmark spreadsheets from LibreOffice Calc used in this paper are also mainly comprised of large cell arrays.

	A	B
1	1	`=A1*2`
2	2	`=A2*2`
3	3	`=A3*2`

(a)

	A	B
1	1	`=R[+0]C[-1]*2`
2	2	`=R[+0]C[-1]*2`
3	3	`=R[+0]C[-1]*2`

(b)

	A	B
1	1	`=R[+1]C[+0]*2`
2	2	`=R[+1]C[+0]*2`
3	3	`=R[+1]C[+0]*2`

(c) Transitive cell array.

	A	B
1	1	`=R[+0]C[-1]*2`
2	2	`=R[+0]C[-1]*2`
3	3	`=R[+0]C[-1]*2`

(d) Intransitive cell array.

Fig. 2: Each cell in the cell array of column B in fig. 2a takes the corresponding value in column A and multiplies it by two.

Cell arrays can be classified as either *transitive* (fig. 2c) or *intransitive* (fig. 2d) [5]. If a cell array only contains formulas that do not reference the cell array itself, we say that it is intransitive, otherwise it is transitive. The need for this distinction will become clear later in sections 6 and 7 when we describe the algorithm and two of its extensions.

4.5 Array Formulas

When a user selects a cell area and enters a formula that returns an array, the elements of the array are distributed across the selected area. The cells in the area share the same singular formula expression but each cell refers only to part of the array.

5 Cost Model

Any static partitioning algorithm needs a cost model to produce well-balanced partitions. Specifically, we are concerned with two metrics: the cost of evaluating a cell and the cost of synchronizing groups of cells in the partition when it is run. We discuss these in turn.

5.1 Big-Step Cost Semantics

We have developed a big-step cost semantics for Funcalc's formula language which we only briefly discuss here due to space limitations. We refer interested readers to the full details in our technical report [7]. The general judgement form $\sigma, \alpha \vdash e \Downarrow v, c$ states that given environment σ mapping cells to values and an environment α mapping cells to array formulas, the expression e may evaluate to some value v at cost c. The rule for the `SUM` built-in function is shown below.

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \quad \ldots \quad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n}{\sigma, \alpha \vdash \texttt{SUM}(e_1, \ldots, e_n) \Downarrow \sum_{i=1}^{n} v_i, 1 + \sum_{i=1}^{n} c_i} \text{ (sum)}$$

Rule (sum) states that if all its argument expressions evaluate to values at some cost, the function call may evaluate to the sum of those values. The total cost is the sum of costs of the individual function arguments plus one. There are additional rules for handling errors in function arguments. The semantics currently use unit costs but we intend to use more precise costs in the future such as those obtained from profiling.

5.2 Synchronization Cost

The framework developed by Sarkar et al. [17] targeted both shared-memory and distributed systems so their cost model had to accommodate different types of communication costs. For example in a distributed setting, inter-cluster communication is usually more expensive than intra-cluster communication. Our algorithm targets modern shared-memory multicore architectures where the cost

model must capture synchronization between different threads. For simplicity, we use a constant cost for synchronization between threads based on benchmarking results. While this does not take memory latency and other hardware aspects into account, it is currently sufficient for generating partitions capable of accelerating spreadsheet computation. Moreover, it is difficult to approximate the low-level synchronization costs that are subject to the whims of the operating system over which we have no control.

6 Static Partitioning Algorithm

Sarkar [17] showed that finding the optimal partition is NP-complete in the strong sense and developed an approximate partitioning algorithm which was close to optimal in practice. In this section, we present a similar partitioning algorithm for spreadsheets. After assigning costs to all formula cells, the algorithm can start partitioning. It consists of two steps: iterative merging and scheduling. Afterwards, we introduce a preprocessing step in section 6.3 that speeds up partitioning and a postprocessing step in section 6.4 that applies an optimisation to sequential paths in the resultant partition.

6.1 Problem Formulation

We view a spreadsheet as a graph $G = (V, E)$ consisting of a set of formula cells $V = \{c_0, \ldots, c_n\}$ and a set of support edges $E \subset (V \times V)$. We can follow the edges in the opposite direction to follow cell dependencies. Inspired by Sarkar [17], we wish to partition V into an acyclic partition $P_f = \{\tau_0, \ldots, \tau_m\}$ consisting of disjoint, load-balanced groups τ_i where the cells in a group are a subset of V: $\textsc{Cells}(\tau_i) \subseteq V$, all formula cells are contained in some τ: $\bigcup_{i=0}^{m} \textsc{Cells}(\tau_i) = V$, and P_f minimizes an objective function F: $\arg\min F(P) = P_f$. Note that we do not require that P_f be the optimal partition. The objective function F approximates the balance between parallelism and synchronization cost of a partition, and is introduced in the next section. We can view a partition P as a condensation of the cell graph where subsets of cells have been assigned to some group τ_i. We refer to this condensed graph as the τ-graph. We require that any partition P produced by the algorithm be acyclic to ease the scheduling of the partition, but defer a detailed discussion. We define the following operations on a group τ.

- $\textsc{Cells}(\tau)$ - The set of cells in τ.
- $\textsc{Pred}(\tau)$ - The set of predecessors of τ.
- $\textsc{Succ}(\tau)$ - The set of successors of τ.
- $\textsc{Time}(\tau)$ - The estimated total time to recalculate each cell in $\textsc{Cells}(\tau)$.
- $\textsc{Sync}(\tau)$ - The synchronization cost of τ.

The predecessors and successors are determined by the dependency and support edges of the cells in $\textsc{Cells}(\tau)$.

6.2 Iterative, Greedy Group Merging

We now iteratively and greedily merge pairs of τ's as guided by the objective function F, until we reach the coarsest partition consisting of a single τ containing all cells with no parallelism but no synchronization overhead either. We select the intermediate partition that minimized F as the output of the algorithm. The objective function F is the maximum of the *critical path* term and the *overhead* term [17].

$$Sync(P) = \sum_{\tau \in P} (|\textsc{Pred}(\tau)| + |\textsc{Succ}(\tau)|) \cdot Sync(\tau) \tag{1}$$

$$Time(P) = \sum_{\tau \in P} \textsc{Time}(\tau) \tag{2}$$

$$F(P) = max\left(\frac{CPL(P)}{Time(P) \div N}, 1 + \frac{Sync(P)}{Time(P)}\right) \tag{3}$$

The total synchronization cost of P in equation (1) is the number of predecessors and successors of each τ times its synchronization cost. The total time to execute P in equation (2) is the summation of the time taken to execute each $\tau \in P$, and is constant throughout partitioning since the amount of work in the partition remains constant but its distribution between τ's does not. Finally, the objective function in equation (3) is the maximum of the *critical path* and *overhead* terms. The former term is the critical path length (denoted as CPL in the equation), i.e. the most expensive sequential path in the τ-graph, divided by the ideal parallel execution time of P given N total processors. The overhead term is the synchronization cost of P normalised by the time taken to execute P.

A fine partition with a critical path length close to the ideal execution time would have a critical path term close to one, but is likely to have dominant overhead term since many τ's need to synchronize. Conversely, a coarser partition may have a smaller overhead term as the coarseness of the partition means less groups need to synchronize, but have a dominant critical path term since many τ's might have been merged into the critical path. In this way, the merging step uses F to balance the trade-off between parallelism and synchronization.

When selecting the groups τ_1 and τ_2 to merge, we select τ_1 as the group with the largest overhead in hopes of reducing the partition's overall synchronization cost [17]. We select τ_2 as the group that yields the smallest change in the critical path length of the partition if we were to merge it with τ_1. During iteration, we record $F(P)$ for each partition and return the partition which minimized F as the output of the algorithm. We call this partition P_f.

Acyclic Constraint To keep all partitions acyclic, we impose an *acyclic constraint*[1] on each partition. When two groups τ_1 and τ_2 are selected for a merge,

[1] Originally referred to as the *convexity constraint* in [17] as it relates to convex subgraphs.

we also merge any τ that lies on a path between τ_1 and τ_2, and thus outside the convex subgraph defined by τ_1 and τ_2. Definition 1 defines a convex subgraph in general [17].

Definition 1. *A subgraph H of a directed graph G is convex if for every pair of vertices $a, b \in H$, any path between a and b is fully contained in H.*

For example, if there is a path $\tau_1 \to \tau \to \tau_2$ and we did not merge τ with τ_1 and τ_2, we would introduce a cycle in the merged τ-graph. Intuitively, definition 1 prohibits τ's from spawning and waiting for work (a loop between two groups), and fork-join parallelism where the fork e.g. happens at τ_1 and the join at τ_2. While this may remove some parallelism from the partition, it greatly simplifies scheduling.

6.3 Cell Array Preprocessing

We include a preprocessing step in the algorithm that assigns each cell array to its own τ in the initial partition so we can exploit its internal parallelism.

This has two advantages. First, it decreases the number of groups that need to be considered for merging, lowering the partitioning time. Second, the algorithm initially needs to determine the predecessors and successors of each τ, which is necessary for estimating the synchronization cost of a partition and keeping track of dependencies when merging. Instead of querying each cell in a potentially large cell array in some τ, we can query only its four corner cells to quickly determine its predecessors $\textsc{Pred}(\tau)$ that also contain cell arrays. Due to the complementary nature of the support and dependency edges, this also establishes that τ is a successor of each $\tau_p \in \textsc{Pred}(\tau)$. This also lowers the partitioning time. Any other cases are handled by querying each cell.

The preprocessing step can be said to be optimistic as many real-world spreadsheets are structured in a highly regular manner and contain large cell arrays that refer to other cell arrays, so we expect that the preprocessing will usually succeed. Most of the benchmark spreadsheets fall into this category which the preprocessing step can quickly deduce.

Determining Reachability Consider the scenario in fig. 3a for two single-column cell arrays spanned by the cell areas `B1:B255` and `C1:C255` respectively. The top and bottom cell references of the blue cell array in column `B` can both reach only constants in column `A` and so we cannot conclude anything about the predecessors of the remaining cells in the cell array (a subset of them might be able to reach some other cell array). On the other hand, the top and bottom cells of the red cell array in column `C` can both reach the blue cell array in column `B`. We can thus conclude that all the cells in the range `C2:C254` can also reach the blue cell array by virtue of the identical relative cell references shared by the cells. If the top and bottom cells of column `C`'s cell array could reach different τ's, as in fig. 3b, we cannot conclude anything about the other cells in the cell array and must instead query each individual cell.

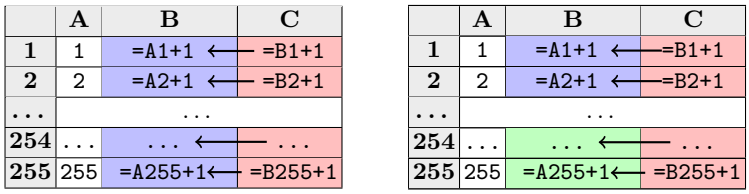

Fig. 3: Preprocessing of different cell arrays.

Array formulas are also analysed and straight forward to handle since their cells share only a single formula expression. Due to space limitations, we omit their analysis here.

We exclude any corners whose formulas are fully transitive, i.e. all cells they refer to are contained in the cell array itself. The rest of the analysis has three primary cases. In the first case, we handle cell references that are absolute in both dimensions (e.g. `$A$1` or `$A$1:$B$2`). If a cell array contains a fully absolute cell reference, then every cell in the cell array can reach it since it refers to the same cell or cell area regardless of the relative position of the referring cell. Any absolute cell areas referenced from the cell array must be fully enclosed in the reachable τ. If they are not, the other part of the cell area may belong to some τ_i which we will only discover by examining each cell in the referenced cell area.

In the second case, we observe that even cell references that are not fully absolute can be considered absolute in the context of a cell array as shown in fig. 4. Since the cell array in column `B` refers to cell `A1` using a row-absolute but column-relative reference, all cells in the cell array will always refer to that cell and it can be viewed as a constant. The same is true for row-relative, column-absolute references and single-row cell arrays.

	A	B
1	`=PI()`	
2		
3	6	`=2*A$1*A3`
4	2	`=2*A$1*A4`
5	...	...

Fig. 4: Spreadsheet calculating the circumference $2\pi r$ of various circle radii. Cell `A1` holds the constant π which the cell array in column `B` refers to. Since the reference is row-absolute and column-relative, all cells in the cell array always refer to `A1`. This scenario occurs in the `building-design` spreadsheet.

The third and final case handles any other *relative* cell references. For each reference in the cell array's formula expression, we consider each unique pair of corners and examine what cells or areas they refer to. This is necessary since

all pairs, even diagonally opposite corners, may refer to the same τ. If for any pair of corners, one of them is transitive, we disregard the pair. For single-cell references, if both corners of a cell array in τ_i can reach cells belonging to the cell array of some τ_j, we add τ_j as a predecessor of τ_i. For cell areas, we require the same conditions but also require that the referenced cell areas are wholly contained in the reachable τ_i as earlier for the second case. Any cells that are not part of a cell array, and thus not handled by this analysis, are put into their own initial τ. We are currently working on a formal proof of the analysis.

6.4 Postprocessing

The algorithm is approximate and is not guaranteed to produce the optimal partition [17] for a given spreadsheet, so it may miss obvious optimisations in the final partition. One such case is a sequential chain of dependencies in the τ-graph whose parts are assigned to different τ's. We could avoid unnecessary synchronization by instead assigning the entire chain to a single τ. Therefore, once the final partition has been found, we traverse the τ-graph to find such chains and ensure that they are assigned to a single τ. Such a scenario occurs in the `building-design` spreadsheet.

6.5 Scheduling Partitions

The merging step of the algorithm leaves us with a final partition $P_f = \{\tau_0, \ldots, \tau_m\}$. Since P_f is acyclic, we can schedule the partition by first topologically sorting the τ-graph by its dependencies then create tasks using the Task Parallel Library (TPL) [15] to run each τ. We iterate through the topologically sorted list and either (1) mark a τ without any dependencies as a *source* and create a task to execute it; (2) create a TPL continuation task that waits for *all* its dependent tasks to finish before starting. We then start all source tasks and wait for all tasks to complete. Each non-source task first checks if all its dependent tasks ran to normal completion. If not, the task immediately stops and propagates any errors to its successors so that the execution can quickly terminate.

7 Extensions

Cells within each τ are evaluated sequentially and the algorithm only parallelizes the execution of the τ-graph, disregarding any additional parallelism inside each τ. In this section, we present three extensions to the algorithm to remedy this: the first extension uses *nested parallelism* within cell arrays; the second extension uses our parallel spreadsheet interpreter [4] to run the cells in each τ in parallel; the third extension uses the rewriting tool from [5] to rewrite cell arrays to calls to compiled higher-order functions that can also be executed in parallel.

7.1 Nested Cell Array Parallelism

This extension relies on the fact that spawning nested TPL tasks within a task will enqueue them in the current threadpool thread's local queue, circumventing the global queue and possibly reducing contention. However, we cannot necessarily spawn a task for each cell in the cell array since its references may be transitive [5].

In fig. 5a, each reference in the formula of the cell array in column `B` refers to a cell in the same row but in column `A`. Since none of the relative cell references are transitive, we can easily spawn a task for each cell in the cell array, and because a τ is only executed when all its inputs are ready, all its dependencies will already have been computed. In fig. 5b, the cell reference refers transitively to a cell five rows below it. Tasks would not properly synchronize if spawned for each cell in the cell array, but we can still parallelize some of the work by subdividing the cell array into subgroups of five so that each subgroup will not have any transitive references to themselves [5]. We then execute each subdivision in parallel in a lockstep fashion. Therefore, we must first perform an analysis of all cell arrays to determine if and how they can be executed in parallel. An analysis is provided in [5]. We do not currently parallelize transitive cell arrays.

	A	B
1	1	`=R[+0]C[-1]/7`
2	2	`=R[+0]C[-1]/7`
3	3	`=R[+0]C[-1]/7`

(a) Intransitive cell array.

	A	B
1	...	`=R[+5]C[+0]/7`
2	...	`=R[+5]C[+0]/7`
3	...	`=R[+5]C[+0]/7`

(b) Transitive cell array.

Fig. 5: An intransitive cell array that can be executed in parallel and a transitive cell array that can be executed in a lockstep fashion.

7.2 Puncalc: A Parallel Interpreter for Spreadsheets

Unlike nested cell array parallelism, using our parallel spreadsheet interpreter does not require an additional analysis of cell arrays since the algorithm already ensures proper synchronization [4]. The interpreter follows the support graph in parallel in search of cells to compute, but this would mean that cells belonging to successor τ's might be evaluated prematurely. To avoid this, we disallow the interpreter from following support edges.

7.3 Rewriting Cell Arrays To Higher-Order Function Calls

Biermann et al. [5] analysed each cell array and rewrote eligible ones to an array formula consisting of a call to a higher-order, compiled function based on patterns exhibited by the cell array's formulae. Higher-order, compiled functions

are a feature of Funcalc [18]. Users can define functions in cells which are then compiled to .NET bytecode. These are referred to as *sheet-defined functions*. Based on the cell array analysis, an expression might be rewritten to a `map` or `prefix` operation or not rewritten at all. This has led to good speed-ups, even with no parallelization and even for spreadsheets that contain little computation such as some from the EUSES corpus [11]. The spreadsheet is rewritten after being loaded from disk, so no change to the static partitioning algorithm is necessary since we already handle array formulas.

8 Results

8.1 Experimental Setup

We used the spreadsheets from the LibreOffice Calc benchmark suite[2], adapted to run in Funcalc, to evaluate our algorithm. We did not use any of the EUSES [11] or Enron [13] corpora since they contain very little computation in general and we do not expect significant speed-ups for these spreadsheets. We partitioned all spreadsheets for each core configuration since the partitioning algorithm is dependent on the number of available cores (see equation (3) in section 6.2).

Our test machine was an Intel Xeon E5-2680 v3 with 24 physical 2.5 GHz cores and hyperthreading (48 logical cores total), running 64-bit Windows 10 and .NET 4.7.1. We initially performed three warm-up runs and ran each benchmark for two iterations. For each iteration, we ran the benchmark ten times, for a total of 20 runs, and computed the average execution time[3]. We report the average of those two averages in table 2. We disabled the TPL's heuristics for thread creation and destruction so that a thread was created per processor at start-up.

Spreadsheet	Cell Arrays	% of Formulas in Cell Arrays	Rewritten Cell Arrays
`building-design`	6	99.93%	6/0
`energy-markets`	76	99.99%	76/0
`grossprofit`	9	99.94%	9/0
`ground-water`	12	100%	12/0
`stock-history`	22	99.97%	20/0
`stocks-price`	8	99.99%	8/0

Table 1: The number of cell arrays in the LibreOffice Calc spreadsheets; the percentage of the spreadsheet's formulas in cell arrays; and the number of rewritten intransitive and transitive cell arrays. No transitive cell arrays are rewritten since none of the spreadsheets contain any.

[2] Available at https://gerrit.libreoffice.org/gitweb?p=benchmark.git;a=tree

[3] Raw data available at https://github.com/popular-parallel-programming/p3-results/tree/master/static-partitioning

Spreadsheet	Sequential	x2	x4	x8	x16	x32	x48	Speed-up
Base Implementation								
building-design	32.12	**30.72**	30.92	31.26	31.05	30.85	31.79	1.01x
energy-markets	168.16	157.08	95.75	66.51	**52.95**	75.45	139.41	1.21x
grossprofit	102.19	102.33	53.86	33.25	**32.59**	32.73	34.66	2.95x
ground-water	81.26	72.42	36.13	24.49	17.65	21.02	**17.30**	4.70x
stock-history	64.90	61.90	35.54	19.12	**17.20**	18.69	17.64	3.68x
stocks-price	102.74	**158.94**	171.10	169.56	174.53	168.10	172.34	0.60x
Nested Cell Array Parallelism Extension (section 7.1)								
building-design	32.12	26.23	13.32	7.33	3.98	2.16	**1.62**	19.84x
energy-markets	168.16	156.68	95.84	66.67	**53.22**	89.35	200.28	0.84x
grossprofit	102.19	102.72	53.31	32.46	21.06	**17.16**	19.95	5.12x
ground-water	81.26	69.97	35.59	19.29	10.41	5.32	**3.71**	21.89x
stock-history	64.90	58.84	29.94	17.73	10.47	7.00	**6.17**	10.52x
stocks-price	102.74	130.46	166.70	164.37	**74.44**	145.48	166.87	0.62x
Puncalc Extension (section 7.2)								
building-design	32.12	31.97	16.12	8.86	4.82	2.68	**1.91**	16.81x
energy-markets	168.16	199.63	146.66	128.06	158.50	**90.95**	202.04	0.83x
grossprofit	102.19	106.70	55.22	33.48	21.57	**17.55**	20.25	5.05x
ground-water	81.26	80.47	41.22	22.81	12.17	6.26	**4.31**	18.87x
stock-history	64.90	59.05	29.97	17.81	10.92	7.12	**7.03**	9.24x
stocks-price	102.74	148.56	174.75	168.01	**65.26**	143.08	168.25	0.61x
Cell Rewriting Extension (section 7.3)								
building-design	32.12	45.35	22.79	12.49	6.58	3.35	**2.39**	13.44x
energy-markets	168.16	206.16	150.10	99.84	**91.04**	335.60	400.26	0.42x
grossprofit	102.19	109.75	58.79	36.74	**26.56**	31.98	63.51	1.61x
ground-water	81.26	111.90	57.74	32.07	16.71	8.34	**5.79**	14.03x
stock-history	64.90	51.81	26.94	14.31	7.37	3.91	**2.72**	23.88x
stocks-price	102.74	149.98	91.09	66.45	**62.60**	205.99	239.11	0.43x

Table 2: Absolute running times in seconds for each configuration of cores for the base implementation and its three extensions. Speed-up is for parallel execution on 48 cores relative to normal sequential execution of Funcalc. Bold numbers denote the fastest execution for each spreadsheet. The standard deviation is within ± 0.08 for all results, except for the base implementation running `stocks-price` on 16 cores with a standard deviation of ± 0.18.

8.2 Discussion

Partitioning currently takes on the order of a few minutes where the dominating factor is applying the big-step cost rules for each cell. We plan to rectify this in the future by caching and reusing computed costs. It is also possible to save the partition alongside the spreadsheet data so that it can be loaded quickly next time without having to partition again. There are five key observations to be made from tables 1 and 2.

Observation 1 The benchmark spreadsheets contain large cell arrays that contain almost all formula cells.

Table 1 shows that all our benchmark spreadsheets are dominated by large cell arrays which contain almost all formula cells. This aligns with the observations made by Dou et al. [9] that cell arrays are common structures in spreadsheets. This has two implications. First, the preprocessing step successfully analyses many of these cell arrays. Second, the many large, intransitive cell arrays means that there is a lot of computation we can exploit with the three proposed extensions from section 7.

Observation 2 The performance of the base implementation shows that it is necessary to exploit the internal parallelism of cell arrays.

In table 2, we get varying results for the base implementation but do get some speed-up, especially for the `grossprofit`, `ground-water` and `stock-history` spreadsheets. However, it is evident that we must also exploit the additional parallelism exposed by cell arrays when comparing results with the three extensions.

Observation 3 The nested cell array extension produces the best overall speed-ups on 48 cores.

Out of the three extensions, the nested cell array parallelism (section 7.1) gives the overall best speed-ups on 48 cores with a maximum speed-up of 21.89 for the `ground-water` spreadsheet.

Observation 4 The `energy-markets` and `stocks-price` spreadsheets have less predictable speed-ups and peak performance consistently occurs at 16 or 32 cores. Adding more cores slows down recalculation.

Observation 4 is true for the base implementation and its three extensions with the exception of `stocks-price` for the base implementation. It is especially perplexing for `energy-markets` since it contains ample parallelism which is captured by partitioning. Likewise, `stocks-price` also contains some degree of parallelism, albeit less. The slowdown may stem from TPL scheduling or hardware. Our test machine has 2 separate chips of 12 physical cores each which may result in off-chip communication for a large amount of threads.

However, we get approximately between 1.3–3.0x speed-up for 16 and 32 cores for both these spreadsheets which may be consistent with the above hardware observation since using more threads may increase off-chip communication. The upside is that most modern hardware can run between 8 and 16 concurrent threads with hyperthreading for which we get positive overall speed-ups.

Observation 5 The cell rewriting extension achieves different speed-ups compared to the other extensions for some spreadsheets.

Table 1 shows that all intransitive cell arrays are rewritten except for the `stock-history` spreadsheet where 2 cell arrays are not rewritten. No transitive cell arrays exist in any of the spreadsheets. The results are quite different from the other two extensions. The `energy-markets` and `stocks-price` spreadsheets have even worse performance on 48 cores but their peak performance at 16 and 32 cores is comparable to the peak performances of the other two extensions. For the `ground-water` spreadsheet, we observe 14.03x speed-up as opposed to 21.89x and 18.87x for extension 1 and 2 respectively. On the other hand, we see that the best speed-up out of all the results is 23.88 for 48 cores for the `stock-history` spreadsheet which has around a 10-fold speed-up for the other two extensions. It is unclear why we observe these results but one explanation might be that more efficient sheet-defined functions are generated for the `stock-history` spreadsheet.

9 Conclusion

We have presented a novel static partitioning algorithm for spreadsheets that can automatically identify sufficient parallelism and achieve good speed-ups on a set of benchmark spreadsheets. The partitioning algorithm is based on a big-step cost semantics for the formula language of Funcalc. We have proposed three extensions to the algorithm that further accelerate computation.

While cell arrays are common structures in spreadsheets, they may not universally be so. We do not benchmark on spreadsheets that contain few or no cell arrays where it should be expected that the partitioning time and speed-up will be affected. In future work, we intend to find large spreadsheets with these characteristics and run the partitioning algorithm on them. Likewise, we should find or construct spreadsheets with large, transitive cell arrays as well.

It may not always be best to use the full capabilities of the hardware for all spreadsheets as showcased by the `energy-markets` and `stocks-price` spreadsheets. It would be interesting to see if we can use the cost model to capture this information and to control the amount of parallelism if we suspect that execution may suffer if we use too many threads. It may also suffice to re-enable TPL's heuristics for thread creation, or instead opt for manual thread-based scheduling to see if the internal scheduling algorithm of TPL is the cause of the slowdowns.

Lastly, we are working on an abstract interpreter for assigning the costs of our big-step cost semantics from section 5. The interpreter should provide better cost estimates for branching constructs such as `=IF(1, "yes", "no")` where we do not always know which branch will ultimately be taken, and will also include a closure analysis for more precise costs of function applications.

Acknowledgements

The author would like to thank Peter Sestoft and Florian Biermann for valuable insight and discussions during the development of this work, as well as Peter Sestoft and Holger Stadel Borum for proofreading.

Bibliography

[1] Abraham, R., Erwig, M.: Inferring Templates from Spreadsheets. ICSE '06.
[2] Abramson, D., Sosic, R., Giddy, J., Hall, B.: Nimrod: a tool for performing parametrised simulations using distributed workstations. HPDC '95.
[3] Abramson, D., Roe, P., Kotler, L., Mather, D.: Activesheets: Supercomputing with spreadsheets. HPC '01.
[4] Biermann, F., Bock, A.A.: Puncalc: Task-based Parallelism and Speculative Reevaluation in Spreadsheets. HLPP '18.
[5] Biermann, F., Dou, W., Sestoft, P.: Rewriting High-Level Spreadsheet Structures into Higher-Order Functional Programs. PADL '18.
[6] Bock, A.A.: A Literature Review of Spreadsheet Technology. Technical report '16. ISBN 978-87-7949-369-8.
[7] Bock, A.A., Bøgholm, T., Sestoft, P., Thomsen, B., Thomsen, L.L.: Concrete and Abstract Cost Semantics for Spreadsheets. Technical report '18. ISBN 978-87-7949-369-8.
[8] Cann, D.: Retire Fortran? A debate rekindled. Commun. ACM '92.
[9] Dou, W., Cheung, S.C., Wei, J.: Is Spreadsheet Ambiguity Harmful? Detecting and Repairing Spreadsheet Smells due to Ambiguous Computation. ICSE '14.
[10] EuSpRiG Horror Stories, http://eusprig.org/horror-stories.htm
[11] Fisher, M., Rothermel, G.: The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanisms. SIGSOFT SEN '05.
[12] Hermans, F., Dig, D.: BumbleBee: A Refactoring Environment for Spreadsheet Formulas. SIGSOFT FSE '14.
[13] Hermans, F., Murphy-Hill, E.: Enron's Spreadsheets and Related Emails: A Dataset and Analysis. ICSE '15.
[14] Hermans, F., Pinzger, M., van Deursen, A.: Supporting Professional Spreadsheet Users by Generating Leveled Dataflow Diagrams. ICSE '11.
[15] Leijen, D., Schulte, W., Burckhardt, S.: The Design of a Task Parallel Library. SIGPLAN Not. '09.
[16] Microsoft: HPC Services For Excel.
[17] Sarkar, V.: Partitioning and Scheduling Parallel Programs for Multiprocessors. Research Monographs In Parallel and Distributed Computing, MIT Press '89.
[18] Sestoft, P.: Spreadsheet Implementation Technology. MIT Press '14.
[19] Swidan, A., Hermans, F., Koesoemowidjojo, R.: Improving the Performance of a Large Scale Spreadsheet: A Case Study. SANER '16.
[20] Trudeau, J.: Collaboration and Open Source at AMD: LibreOffice, https://developer.amd.com/collaboration-and-open-source-at-amd-libreoffice/, accessed on 31.07.2015
[21] Wack, A.P.: Partitioning Dependency Graphs for Concurrent Execution: A Parallel Spreadsheet on a Realistically Modeled Message Passing Environment. Ph.D. thesis, Newark, DE, USA '96.