Abstract

Spreadsheets are popular tools for end-user development and complex modelling but can suffer from poor performance. While end-users are usually domain experts they are seldom IT professionals that can leverage today’s abundant multicore architectures to offset such 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 logical cores.

Keywords

Spreadsheets - Partitioning - Parallelism
Abstract. Spreadsheets are popular tools for end-user development and complex modelling but can suffer from poor performance. While end-users are usually domain experts they are seldom IT professionals that can leverage today’s abundant multicore architectures to offset such 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 logical cores.

Keywords: Spreadsheets · Partitioning · Parallelism

1 Introduction

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

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 both declarative and 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 big-step cost semantics to estimate the work of cells and to produce well-balanced partitions. We implement the algorithm in the research spreadsheet application Funcalc [18] written in C#, and we believe this is the first thorough investigation of static partitioning of spreadsheets.

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 (Sect. 5).
2. An algorithm for statically partitioning spreadsheets and scheduling them on shared-memory multicore processors (Sect. 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 (Sects. 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 cells A1 and A2 while cell C3 refers to the cell area spanned by 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, but it would remain unchanged if copied it to D2 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. The same spreadsheet is shown in Fig. 1b in R1C1 format.

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 dependencies 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 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 3 rows by 1 column cell array 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 10754) 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.

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

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 Sects. 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 concrete 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 \( \sigma \) mapping cells to values and an environment \( \alpha \) mapping cells to array formulas, the expression \( e \) may evaluate to some value \( v \) at cost \( c \). The rule for the \texttt{SUM} built-in function is shown below. The cost is given as integers that effectively describe the number of operations needed to evaluate an expression but we intend to use more precise costs in the future e.g. obtained from profiling.

\[
\begin{align*}
\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 & \quad \ldots \quad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n \\
\sigma, \alpha \vdash \texttt{SUM}(e_1, \ldots, e_n) \Downarrow \sum_{i=1}^{n} v_i, 1 + \sum_{i=1}^{n} c_i
\end{align*}
\]

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.

The big-step cost semantics has other uses. It may serve as a reliable, formal reference for implementations as it does for other functional programming languages; or used to guide other types of partitioning or parallelisation strategies such as off-loading work to a GPU [20]. We could also envision a refactoring tool that can identify and report bottlenecks or costly operations in spreadsheets.
5.2 Synchronization Cost

Sarkar’s framework [17] targeted both shared-memory and distributed systems so their cost model had to accommodate different types of communication costs. Our algorithm targets shared-memory multicore architectures where the cost model must capture synchronization between 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. In the future, we intend to develop a more precise model based on the hardware and operating system.

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. We first assign costs to all formula cells, then partition the spreadsheet by iteratively merging groups of cells and scheduling the computation of those groups. Afterwards, we introduce a preprocessing step in Sect. 6.3 that speeds up partitioning and a postprocessing step in Sect. 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 $\tau_i$ where the cells in a group are a subset of $V$: $\text{CELLS}(\tau_i) \subseteq V$, all formula cells are contained in some $\tau$: $\bigcup_{i=0}^{m} \text{CELLS}(\tau_i) = V$, and $P_f$ minimizes an objective function $F$: $\arg \min F(P) = P_f$. Note that we do not require $P_f$ to be optimal. The objective function $F$ approximates the trade-off between parallelism and synchronization in 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 $\tau_i$ and refer to this condensed graph as the $\tau$-graph. Any partition $P$ produced by the algorithm is required to be acyclic to ease scheduling but we defer a detailed discussion. We define the following operations on a group $\tau$.

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

The predecessors and successors are determined by the dependency and support edges of the cells in $\text{CELLS}(\tau)$. 
6.2 Iterative, Greedy Group Merging

Starting from some initial partition, we now iteratively and greedily merge pairs of \( \tau \)'s as guided by the objective function \( F \), until we reach the coarsest partition consisting of a single \( \tau \) 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] as given by Eq. (3).

\[
Sync(P) = \sum_{\tau \in P} (|\text{Pred}(\tau)| + |\text{Succ}(\tau)|) \cdot \text{Sync}(\tau) \tag{1}
\]

\[
Time(P) = \sum_{\tau \in P} \text{Time}(\tau) \tag{2}
\]

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

The total synchronization cost of \( P \) in Eq. (1) is the number of predecessors and successors of each \( \tau \) times its synchronization cost. The total time to execute \( P \) in Eq. (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 \( \tau \)'s is not. Finally, the objective function in Eq. (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 \( \tau \)-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 \( \tau \)'s need to synchronize. Conversely, a coarser partition may have a small overhead term as the coarseness of the partition means less groups need to synchronize, but a dominant critical path term since many \( \tau \)'s might have been merged into the critical path. In this way, the merging step uses \( F \) to balance the degree of parallelism versus the cost of synchronization.

When selecting two groups \( \tau_1 \) and \( \tau_2 \) to merge, we select \( \tau_1 \) as the group with the largest synchronization cost in hopes of reducing the partition’s overall synchronization cost [17]. We select \( \tau_2 \) as the group that yields the smallest change in the critical path length if we were to merge it with \( \tau_1 \). During iteration, we record \( F(P) \) for each partition and return the partition \( P_f \) which minimized \( F \) as the output of the algorithm.

**Acyclic Constraint.** To keep all partitions acyclic, we impose an acyclic constraint\(^1\) on each partition [17]. When two groups \( \tau_1 \) and \( \tau_2 \) are selected for a merge, we also merge any \( \tau \) that lies on a path between \( \tau_1 \) and \( \tau_2 \), and thus outside the convex subgraph defined by \( \tau_1 \) and \( \tau_2 \).

\(^1\) Originally referred to as the convexity constraint in [17] as it relates to convex subgraphs.
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 $\tau_1 \rightarrow \tau \rightarrow \tau_2$ and we did not merge $\tau$ as well, we would introduce a cycle in the $\tau$-graph. Intuitively, the acyclic constraint prohibits $\tau$’s from spawning and waiting for work (a loop between two groups), and fork-join parallelism where the fork e.g. happens at $\tau_1$ and the join at $\tau_2$. While this may remove some parallelism from the partition, it greatly simplifies scheduling.

6.3 Cell Array Preprocessing

In a preprocessing step, we assign each cell array to its own $\tau$ in the initial partition so we can later exploit any 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 $\tau$, which is necessary for computing 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 $\tau$, we can in most cases query only its four corner cells to quickly find predecessors in $\text{Pred}(\tau)$ that also contain cell arrays. Due to the complementary nature of the support and dependency graphs, this also establishes that $\tau$ is a successor of each such $\tau_p \in \text{Pred}(\tau)$. In other cases, we conservatively query each cell.

The preprocessing step can be said to be optimistic as many real-world spreadsheets contain large cell arrays that refer to other cell arrays, so we expect that the preprocessing will usually succeed. Most of our benchmark spreadsheets fall into this category.

Determining Reachability. Consider the two single-column cell arrays spanned by cell areas B1:B255 and C1:C255 respectively in Fig. 3a. 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 dependencies of the remaining cells in the cell array since a subset of them might be able to reach some other cell array. The top and bottom cells of the red cell array in column C can both reach the blue cell array in column B and we 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 $\tau$’s, as in Fig. 3b, we cannot conclude anything about the other cells in the cell array and instead query each cell.

We can similarly analyse array formulas which is straightforward since their cells share a single formula expression. Due to space limitations, we omit their analysis here.

We exclude any corners whose cell references are transitive, i.e. all cells they refer to belong to 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). Every cell in the cell array depends on such a reference regardless of the relative position of the referring cell. Absolute cell areas referenced from the cell array must be fully enclosed in the reachable $\tau$. If they are not, the other part of the cell area may belong to some $\tau_i$ 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.

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 $\tau$. For single-cell references, if both corners of a cell array in $\tau_i$ can reach cells belonging to the cell array of some $\tau_j$, we add $\tau_j$ as a predecessor of $\tau_i$. For cell areas, we require the same conditions but also require that the referenced cell areas are wholly contained in the reachable $\tau_i$ as 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 $\tau$. We are currently working on a formal version of the analysis.

![Fig. 3. Preprocessing of different cell arrays.](image)

![Fig. 4. Spreadsheet calculating the circumference $2\pi r$ of various circle radii. Cell A1 holds the constant $\pi$ 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.](image)
6.4 Postprocessing

The algorithm is approximate and not guaranteed to produce the optimal partition [17] and may miss obvious optimisations, such as a sequential chain of dependencies in the \( \tau \)-graph whose parts are assigned to different \( \tau \)'s. We could avoid unnecessary synchronization by instead assigning the entire chain to a single \( \tau \). Therefore, once the final partition has been found, we traverse the \( \tau \)-graph to find such chains and ensure that they are assigned to a single \( \tau \).

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 \( \tau \)-graph by its dependencies then create tasks using the Task Parallel Library (TPL) [15] to run each \( \tau \). We iterate through the topologically sorted list and either (1) mark a \( \tau \) without 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 every source task and wait for all tasks to complete. Each non-source task first checks if all its dependent tasks ran to normal completion. If not, it immediately stops and propagates any errors to its successors so that execution can quickly terminate.

7 Extensions

Cells within each \( \tau \) are evaluated sequentially and the algorithm only parallelizes the execution of the \( \tau \)-graph, disregarding any additional parallelism inside each \( \tau \). 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] in each \( \tau \); 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. 2d on page 4, 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 \( \tau \) is only executed when all its inputs are ready, its dependencies will already have been computed. In Fig. 2c, each cell reference refers transitively to a cell five rows below it. Blindly spawning tasks for each
cell would not properly synchronize, but we can still parallelize some of the work by subdividing the cell array into subgroups of five which 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, but do not currently parallelize transitive cell arrays.

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 \( \tau \)'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 cell arrays 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. The higher-order, compiled functions are called sheet-defined functions and are a feature of Funcalc [18]. Users can define functions in cells which are then compiled to Common Intermediate Language (CIL) bytecode. 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

To evaluate our algorithm, we adapted the spreadsheets from the LibreOffice Calc benchmark suite\(^2\) to Funcalc. We partitioned all spreadsheets for each core configuration since the partitioning algorithm is dependent on the number of available cores (see Eq. (3) in Sect. 6.2).

Our test machine was an Intel Xeon E5-2680 v3 with two separate hardware chips with 12 2.5 GHz cores each 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. In 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.

### 8.2 Discussion

Partitioning currently takes on the order of a few minutes where the dominating factor is applying the big-step cost rules to each cell. We could rectify this by caching and reusing the computed costs if cells are not modified between partitioning. This may inflate memory usage as the spreadsheets contain between 108,332 and 812,693 cells whose costs would need to be cached. 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. The partitioning itself is very fast primarily due to the cell array analysis discussed in Sect. 6.3 and the presence of large, dominating cell arrays as we discuss in a moment. 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, intransitive 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 which has two implications. First, the preprocessing step successfully analyses most of the cell arrays. Second, the many large cell arrays means that there is a lot of parallel computation we can exploit with the three proposed extensions from Sect. 7.

**Table 1.** From left to right: The number of cell arrays in the LibreOffice Calc spreadsheets; the percentage of formulas contained in cell arrays; the average size of cell arrays; and the number of rewritten intransitive and transitive cell arrays. No transitive cell arrays are rewritten because none of the spreadsheets contain any.

<table>
<thead>
<tr>
<th>Spreadsheet</th>
<th>Cell arrays</th>
<th>% of Formulas</th>
<th>Average size</th>
<th>Rewritten cell arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>building-design</td>
<td>6</td>
<td>99.93%</td>
<td>18 042</td>
<td>6/0</td>
</tr>
<tr>
<td>energy-markets</td>
<td>76</td>
<td>99.99%</td>
<td>7032</td>
<td>76/0</td>
</tr>
<tr>
<td>grossprofit</td>
<td>9</td>
<td>99.94%</td>
<td>15 000</td>
<td>9/0</td>
</tr>
<tr>
<td>ground-water</td>
<td>12</td>
<td>100%</td>
<td>10 533</td>
<td>12/0</td>
</tr>
<tr>
<td>stock-history</td>
<td>22</td>
<td>99.97%</td>
<td>10 292</td>
<td>20/0</td>
</tr>
<tr>
<td>stocks-price</td>
<td>8</td>
<td>99.99%</td>
<td>101 578</td>
<td>8/0</td>
</tr>
</tbody>
</table>

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.

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

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 these results with those of 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 extension 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`, `grossprofit` and `stocks-price` spreadsheets have less predictable speed-ups and performance consistently peaks at 16 or 32 cores. Adding more cores seems to slow down recalculation.

Observation 4 applies to the base implementation and its three extensions with the exception of `stocks-price` for the base implementation where the best speed-up is achieved at 2 cores, although it is still slower than sequential execution. It is especially perplexing for `energy-markets` that contains ample parallelism since it consists of multiple independent cell arrays which is captured by partitioning. Likewise, `stocks-price` and `grossprofit` also contain some degree of parallelism.

The slowdown may stem from TPL scheduling and hardware. Our test machine has two separate chips of 12 physical cores each which may result in increasing amounts of off-chip communication. We still get approximately 1.3–3.0x speed-up for 16 and 32 cores for these spreadsheets which may be consistent with the above hardware observation since using more threads may increase off-chip communication.

One would also be inclined to suspect our simplified communication model since the amount of synchronization needed to execute a partition may outweigh the amount of parallelism in the spreadsheets in some cases.

In [4], we measured different structural properties of the spreadsheets but found no correlation with performance. Upon further investigation, the garbage collector was spending excessive amounts of time on generation zero and one collections which meant that less time was spent doing useful computation. When inspecting the managed heap, memory usage stemmed partly from allocation of many small, ephemeral objects related to calling specific intrinsic Funcalc functions which were mostly used by the spreadsheets that exhibit poor performance. We also noticed memory usage related to TPL internals and task creation which may increase with the number of cores. The latter is likely caused by the fine granularity work distribution from parallelisation of cell arrays. We could chunk cell arrays based on the number of processors to create less tasks with a more

---

4 We used the Windows Performance Monitor to monitor performance characteristics and WinDbg to inspect the managed heap.
sensible work distribution. One could also switch to manual thread management to circumvent TPL and task allocations. Finally, some object references might be retained when they should be garbage collected instead. We are currently working on identifying and rectifying these issues.

**Observation 5.** The cell rewriting extension achieves different speed-ups compared to the other extensions for some spreadsheets. The nested cell array and Puncalc extensions achieve similar speed-ups.

Table 1 shows that all intransitive cell arrays are rewritten except for two in the stock-history spreadsheet and that 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. The best speed-up out of all the results is 23.88 for 48 cores for the stock-history spreadsheet. We offer two explanations for these differences. First, more efficient and parallelizable sheet-defined functions may be generated for the stock-history spreadsheet. Second, Table 1 shows the average size of cell arrays is much larger in stock-history so more cells are rewritten that invoke generated bytecode instead of using the slower interpreter.

The two other extensions achieve similar speed-ups since their method of parallelization is similar. The nested cell array extension beats the Puncalc extension which is likely because it directly spawns tasks for each cell in the cell array while the Puncalc extension uses additional synchronization and a global shared work queue, as it is built to evaluate any kind of topology, not just cell arrays.

9 Conclusion

We have presented a static partitioning algorithm for spreadsheets that automatically identifies sufficient parallelism and achieves good speed-ups on a set of benchmark spreadsheets. A big-step cost semantics for the formula language of Funcalc was used to estimate the cost of cells. Finally, we extended the partitioning algorithm in three different ways to 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 the partitioning time and speed-up will likely be affected. In future work, we intend to benchmark large spreadsheets with these characteristics and ones with large, transitive cell arrays.

We have not compared any of our results to Excel or LibreOffice Calc and we do not believe this is particularly useful. Both Excel and LibreOffice Calc are intended for real-world use while Funcalc is primarily intended for research and has not been publicly distributed.
It would be interesting to capture hardware characteristics in the cost model 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 opt for manual thread management. Lastly, we acknowledge that our synchronization model may be too simplistic and it would be more meaningful to develop a model that takes hardware into account, e.g. based on benchmarks that compare on-chip versus off-chip synchronization. Another hardware aspect to consider is that memory bandwidth and cache hierarchies often limit scalability in NUMA systems which could also be incorporated into our synchronization cost model.

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.

References

16. Microsoft: HPC Services For Excel
17. Sarkar, V.: Partitioning and Scheduling Parallel Programs for Multiproces-
sors. Research Monographs In Parallel and Distributed Computing, MIT Press, 
Cambridge (1989)
large scale spreadsheet: a case study. In: SANER (2016)
developer.amd.com/collaboration-and-open-source-at-amd-libreoffice/
spreadsheet on a realistically modeled message passing environment. Ph.D. thesis, 
Newark, DE, USA (1996)
## Author Queries

### Chapter 14

<table>
<thead>
<tr>
<th>Query Refs.</th>
<th>Details Required</th>
<th>Author’s response</th>
</tr>
</thead>
<tbody>
<tr>
<td>AQ1</td>
<td>Per Springer style, both city and country names must be present in the affiliations. Accordingly, we have inserted the city name “Copenhagen” in affiliation. Please check and confirm if the inserted city name is correct. If not, please provide us with the correct city name.</td>
<td></td>
</tr>
</tbody>
</table>