Parallel Spreadsheet Evaluation and Dynamic Cycle Detection

Journal:	*Concurrency and Computation: Practice and Experience*
Manuscript ID	CPE-20-0803
Editor Selection:	Prof. David Walker
Wiley - Manuscript type:	Research Article
Date Submitted by the Author:	17-Jun-2020
Complete List of Authors:	Bock, Alexander; Computer Science
Keywords:	spreadsheet, parallelism, cycle detection, recalculation, declarative, Funcalc

SCHOLARONE™
Manuscripts

Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

Parallel Spreadsheet Evaluation and Dynamic Cycle Detection

Alexander Asp Bock[1]*

[1]Computer Science Department, IT University of Copenhagen, Denmark

Correspondence
*Alexander Asp Bock, Rued Langgaards Vej, 2300, Copenhagen, Denmark. Email: albo@itu.dk

Present Address
Rued Langgaards Vej, 2300, Copenhagen, Denmark

Abstract

It was estimated there would be 72 million users using spreadsheets monthly in 2017 some of which build complex financial, scientific and mathematical models. Most of these end-users are not trained IT professionals but domain experts. In the age of multicore computing and ever-increasing amounts of data, how can end-users access this powerful, parallel hardware to accelerate spreadsheet computation?

Some existing solutions are usually not fully automatic and require a level of interaction from end-users to facilitate parallel execution. Ideally, an end-user tool would transparently exploit the underlying hardware and automatically discover available parallelism in the spreadsheet without any required interaction.

This paper presents an algorithm for automatic parallel evaluation of the cells in a spreadsheet and dynamic, parallel cycle detection. It is implemented in the Funcalc research spreadsheet application which supports higher-order functions in the spreadsheet paradigm. Altogether, this promises a powerful and expressive platform for end-user development. Our results on a 48 logical core machine show a maximum 10–20x speed-up on a set of benchmark spreadsheets and a maximum 15–32x speed-up on a set of synthetic spreadsheets with predefined topologies.

KEYWORDS:
spreadsheet; parallelism; cycle detection; recalculation; declarative; Funcalc

1 | INTRODUCTION

Spreadsheets are valuable organisational tools used today in e.g. finance, engineering and science for complex modelling and computation. They are commonly declarative, first-order functional languages but are also unusual in their graphical representation of data and reactive dataflow model of computation which in part has led to their widespread success. Their declarative aspect enables end-users to focus on specifying *what* needs to be computed without having to worry about *how* it gets computed. Their functional aspect frees end-users from having to worry about stateful effects. Scaffidi[1] estimated in 2017 that there would be 72 million professional spreadsheet end-users in that same year, making spreadsheets one of the most popular forms of functional programming.

With the advent of multicore computing as well as growing datasets, it is becoming more and more important to develop software that can exploit the increasingly commonplace parallel hardware to speed up computation. However, spreadsheet end-users are seldom trained IT professionals or computer scientists but rather possess domain-specific expertise, so how can they leverage this parallel hardware? Existing solutions usually require some level of user interaction to facilitate parallel execution, such as explicitly stating what part of the spreadsheet should execute in parallel, but an ideal tool would automatically and transparently take advantage of the underlying hardware without requiring input from the user. The prospect of such a tool aligns well with the declarative and functional nature of spreadsheets: end-users can disregard the intricacies of the implementation and focus on problem solving while enjoying increased performance; functional languages and immutability of spreadsheet cells alleviate parallel implementations as cells are never directly modified.

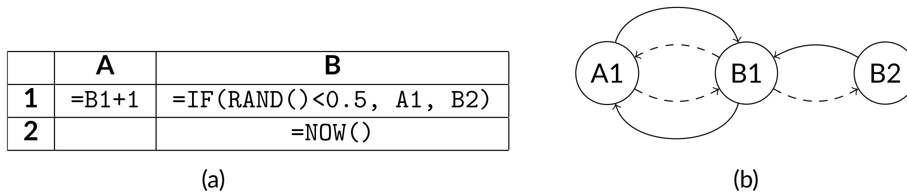


FIGURE 1 (a) A small spreadsheet with a static cycle **(b)** Its corresponding dependency and support edges.

In this paper, we present an algorithm with these desirable features that specifically targets shared-memory multicore systems to recalculate the cells of a spreadsheet in parallel. The algorithm is implemented in Funcalc [2,3], a spreadsheet application for research that introduces higher-order, user-defined functions to the spreadsheet paradigm. We also present a method for detecting cyclic dependencies between cells in parallel, inspired by a distributed cycle detection algorithm [4]. Detecting cycles dynamically during cell evaluation is necessary since some cycles only manifest at this time and computation cannot usually proceed meaningfully. This usually indicates an error which should be reported to the user. Furthermore, detecting cycles avoids computation becoming stuck when in a multi-threaded context.

For Peer Review

Contributions. This paper makes the following contributions:

- A largely topology-agnostic algorithm for recomputing the cells of a spreadsheet in parallel (section 5.1).
- A lock-based method for detecting cycles in parallel where threads gather reachability information to discover cycles (section 5.3).

The rest of the paper is structured as follows. In section 2, we introduce some basic spreadsheet concepts deemed necessary to follow the rest of the paper. We discuss related work in section 3 and section 4 describes the Funcalc spreadsheet application, the concept of sheet-defined functions (SDFs), sequential recalculation and cycle detection. Using sequential recalculation as a foundation, section 5 gives the details of parallel recalculation and parallel cycle detection. Our results are presented in section 6. We then suggest multiple possible extensions and improvements to the algorithm in section 7 and lastly summarise and conclude the paper in section 8.

2 | CORE SPREADSHEET CONCEPTS

In this section, we provide a brief introduction to some of the core spreadsheet concepts and terminology that we deem necessary for understanding this paper. Readers already familiar with the subject can skip this section while those interested in learning more are encouraged to read Sestoft's book on the subject [3].

2.1 | Formulas and Cell References

A spreadsheet consists of a grid of cells and a cell can contain either a constant, such as a number, a string, or an error value (e.g. `#NA` for "not available" or `#DIV/0!` for division by zero); 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 position with columns starting at A and rows at 1. Formulas can refer to other cells using cell addresses (e.g. `=A1+1`) or refer to an area of cells using the addresses of two opposing corner cells that span the cell area, separated by a colon e.g. `=A1:B2`.

2.2 | The Support and Dependency Graphs

The cell references in formulas establish a dependency graph between cells. The inverse graph is called the support graph and captures cell support. A cell's supported cells are also called its support set. It is analogous to a dataflow graph [5] where cells constitute nodes and data flows along the edges from dependencies to supported cells. For the remainder of this paper, we use dashed lines (- - ›) to denote dependency graph edges and solid lines (⟶) to denote support graph edges. In fig. 1, cell B1 conditionally depends on A1 and B2 and both A1 and B2 support B1. Both the dependency and support graphs may be cyclic.

2.3 | Types of Recalculation

A cornerstone of spreadsheets is automatic recalculation: if a cell is modified, all cells that transitively depend on its value are automatically updated to reflect the change, providing users with visual feedback on the effect of a modification. There are two main types of recalculation [3].

A *full* recalculation reevaluates all formula cells in the spreadsheet. A *minimal* or *partial* recalculation evaluates a subset of cells as its names imply. Such a recalculation is initiated when the user modifies a cell and only the supported cells reachable from it need to be recalculated. In actuality, some cells may call functions that are *volatile* and must be recalculated regardless of whether they are directly modified or in the transitive closure of a modified cell. For example, the volatile `RAND` function used in fig. 1 returns a number in the interval $[0, 1[$. If `RAND` was not volatile, it would produce a single random value which would become stale after the first recalculation and we would lose the desired non-determinism of the function, unless we explicitly evaluate the cell again. The same is true for the `NOW` function in B2 which returns the number of days and fractional days since an epoch. Together, we call the modified and volatile cells the *recalculation roots* of a recalculation. Our algorithm focuses primarily on parallel minimal recalculation but can easily be adapted to perform a full recalculation (see section 7.4).

2.3.1 | Static and Dynamic Cycles

Cycles can occur if two cells directly or indirectly refer to one another. A cycle usually indicates an error since it obstructs computation as there is no way to meaningfully proceed without resolving it. Sometimes cycles may be intentional. For example, Excel allows for iterative, user-controlled recalculation of cyclic spreadsheets to model converging computations.

We distinguish between two types of cycles: *static* and *dynamic* cycles. Static cycles describe *potential* cycles that can occur through cell references explicitly listed in a formula's expression. Consider the expression in cell B1 of fig. 1 where there is a static cycle between B1 and A1. Dynamic cycles describe *actual* cycles that occur during cell evaluation such as the condition in B1 being true and evaluating A1 to discover the cycle. A static cycle *may* thus lead to a dynamic cycle. Most spreadsheet implementations, including Funcalc, allow dynamic cycles and our parallel algorithm must detect them to avoid threads becoming stuck. Alternatively, one could flag all static cycles as errors to preclude dynamic cycles but this would require a sequential check to find static cycles before each recalculation, defeating the purpose of running recalculation in parallel in the first place.

3 | RELATED WORK

Research on spreadsheets has primarily focused on detecting and handling errors [6] due to high error rates in spreadsheets, and less on parallelisation. While some systems exist for accelerating recalculation of spreadsheets, few are fully automatic and require interaction from the user.

ActiveSheets [7] uses special plan files to describe inputs and work to the system which are then dispatched to Nimrod [8], a research tool for distributed computation. The system focuses on parallel execution of user-defined Visual Basic for Applications (VBA) functions. These custom functions send the necessary data to the backend for evaluation. ActiveSheets automates aggregating and importing results back into the spreadsheet which is normally required to be done by a user of Nimrod. ActiveSheets is capable of both intra- and inter-cell parallelism.

HPC Services for Excel [9] off-loads the evaluation of user-defined functions (UDFs) or entire workbooks to a Windows high-performance computing (HPC) cluster using a service-oriented architecture (SOA). In the UDF case, the user specifies an auxiliary file containing each UDF and its dependencies to be run on the cluster. As for workbooks, a framework is available to let users define how independent calculations in the workbook can be partitioned and individual results merged.

In his 1996 dissertation, Wack [10] investigated parallelisation of spreadsheet programs using distributed systems and an associated machine model. He used the functional language Scheme as the spreadsheet language making higher-order functions available to users. He partitioned and scheduled a set of predefined patterns and parallelised them via message-passing in a network of work stations. Wack accounted for cycles by disallowing them in the predefined patterns.

Biermann et al. [11] rewrote so-called *cell arrays* to calls to SDFs. Cell arrays are common, contiguous rectangular areas of formulas that share the same formula expression and thus the same computational semantics [12], and may express some degree of data-parallelism. Cell arrays were rewritten to higher-order function calls on arrays such as map or prefix, completely transparent to end-users. Their approach parallelised the internal evaluation of each rewritten cell array but evaluated disjoint rewritten cell arrays sequentially.

LibreOffice Calc automatically compiled data-parallel expressions in cell arrays into OpenCL kernels that execute on AMD GPUs [13]. They reported a 500-fold speed-up for a single spreadsheet. Presentations from the LibreOffice conferences also discuss threaded execution of cell arrays [14].

In earlier work [15], we presented a task-based parallel spreadsheet interpreter that automatically discovers parallelism, finds cyclic references in parallel and also targets shared-memory multiprocessors without requiring modification of the spreadsheetor user interaction. Tasks were spawned

For Peer Review

	A	B
1	=DEFINE("triarea", B6, B2, B3, B4)	
2	"a="	2
3	"b="	3
4	"c="	4
5	"s="	=(B2+B3+B4)/2
6	"result="	=SQRT(B5*(B5-B2)*(B5-B3)*(B5-B4))

FIGURE 2 An SDF for computing the area of a triangle. The function takes three parameters (cells B2, B3 and B4 in green), has one intermediate cell (B5 in light blue) and a single output cell (cell B6 in blue).

using the Task Parallel Library (TPL)[16] and the algorithm obtained roughly a maximum 16x speed-up on the same set of benchmark spreadsheets used in this paper.

Also in earlier work[17], we developed a static partitioning algorithm for globally partitioning spreadsheet cells into load-balanced groups using a cost model based on a big-step cost semantics[18,19]. The groups were then run on multiple processors using the TPL[16]. The paper presented three extensions, two of which incorporate work from the task-based spreadsheet interpreter[15] and Biermann's cell array rewriting[11]. While the algorithm yielded overall positive speed-ups, partitioning time was largely dominated by applying the cost semantic rules to estimate the evaluation time of each cell resulting in partitioning taking up towards one hour to finish. We suggested improvements to reduce the partitioning time.

4 | FUNCALC

Funcalc[2,3] is a spreadsheet application for research prototyping written in C#. It features so-called sheet-defined functions (SDFs) that are higher-order functions defined by users in special function sheets. We first briefly describe how SDFs are defined and used in section 4.1, then describe how sequential recalculation and cycle detection work in Funcalc in section 4.2.

4.1 | Sheet-Defined Functions

A hallmark of Funcalc is its support for higher-order compiled functions called SDFs. Figure 2 shows a simple SDF for computing the area of a triangle. The `DEFINE` function in cell A1 takes the name of the function, followed by a single output cell and zero or more input cells. Input cells (B2, B3, and B4) have a green background, cells containing intermediate computations (B5) have a light blue background, and the output cell (B6) has a blue background. SDFs are automatically compiled to Common Intermediate Language (CIL) bytecode when they are initially created or subsequently modified, supporting the edit-run cycle of ordinary data sheets. SDFs offer the usual benefits of functions such as modularity and reuse. Additionally, they are defined in the formula language already familiar to end-users. Although the implementation of SDFs is not our work[2], we believe they offer a powerful framework for end-user development alongside the parallel spreadsheet algorithm we present in this paper.

4.2 | Sequential Recalculation and Cycle Detection

Sequential recalculation[3] uses a simple abstract syntax tree interpreter that recursively interprets cell dependencies. Sequential *minimal* recalculation evaluates cells in a breadth-first manner using a queue Q as shown in fig. 3. The recalculation roots are first enqueued in Q then each cell is popped from the queue, evaluated and its supported cells enqueued. This continues until either Q is empty or a cycle is discovered. Each cell has an internal state that can be either *dirty* to denote that the cell has not yet been computed; *enqueued* if the cell is in Q; *computing* if the cell is currently being computed; or *uptodate* if the cell has been evaluated. A state transition diagram is shown in fig. 4 as a reference for the following discussion.

The pseudo-code for sequential, minimal recalculation is shown in algorithm 1 (line 31). The recalculation roots are initially marked *dirty* (line 33) using the MarkDirty function (line 1) which recursively marks a cell and its supported cells *dirty*. This corresponds to the dashed state transition in fig. 4 as all cells are usually initially *uptodate* prior to a recalculation. The roots are then enqueued in Q (line 34) and their state is changed to *enqueued* (line 35). This ensures that a cell is only enqueued once as only cells with the *dirty* state can be enqueued. The main loop (line 37) continually dequeues a cell from Q and evaluates it as long as Q is not empty or a cycle has not been discovered, corresponding to the overview in fig. 3.

Function Eval on line 9 evaluates a cell and recursively evaluates any cell dependencies in the cell's expression. The specifics of evaluation depend on a cell's state. If the state is *computing* (line 11), we have encountered a cell we have already tried to evaluate through the dependency graph and discovered a cycle. Thus cycles are discovered through recursive evaluation of cell dependencies whose state is consequently *computing*. Function NotifyCycleAndStop halts recalculation and ReportCycle (line 42) notifies the user of the error via the graphical user interface.

For Peer Review

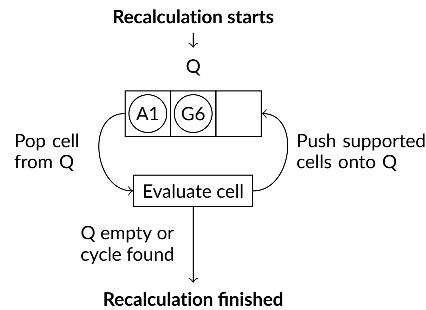


FIGURE 3 Using a queue to compute cells in a breadth-first manner in sequential minimal recalculation.

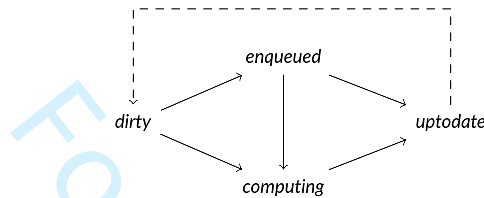


FIGURE 4 The possible transitions of a cell's state.

If the state is either *dirty* or *enqueued* (line 13), we set the state to *computing*, evaluate the cell's expression to a value using EvalExpr, cache the value and set its state to *uptodate*. Function EvalExpr evaluates the cell's expression and any cell references recursively. The state transition from *dirty* to *computing* corresponds to evaluating a *dirty* cell through a cell dependency in the dependency graph; the state transition from *enqueued* to *computing* corresponds to evaluating a cell that was dequeued from Q along the support graph, hence the two paths from the *dirty* state to the *computing* state in fig. 4. Afterwards, the cell's supported cells are enqueued in Q (line 17) using the EnqueueSupport function (line 23). Notice that a supported cell is only enqueued if its state is *dirty* so cells are only enqueued once. Finally, the cell's state may be *uptodate* (line 19) in which case we read the cell's value from the cache via the Cache function.

In a full recalculation, all cells are marked *dirty* and iteratively evaluated without using a queue and the support graph. We omit the code for a full recalculation here for the sake of brevity.

Algorithm 1 Funcalc's algorithm for sequential minimal recalculation

```
 1: function MarkDirty(cell)
 2:   if State(cell) ≠ dirty then
 3:     State(cell) ← dirty
 4:     for u in SupportSet(cell) do
 5:       MarkDirty(u)
 6:     end for
 7:   end if
 8: end function

 9: function Eval(cell, q)
10:   switch State(cell)
11:     case computing :
12:       NotifyCycleAndStop(); break
13:     case dirty or enqueued :
14:       State(cell) ← computing
15:       Cache(cell) ← EvalExpr(cell)
16:       State(cell) ← uptodate
17:       EnqueueSupport(cell, q)
18:       break
19:     case uptodate :
20:       break
21:   return Cache(cell)
22: end function

23: function EnqueueSupport(cell, q)
24:   for u in SupportSet(cell) do
25:     if State(u) = dirty then
26:       State(u) ← enqueued
27:       Enqueue(q, u)
28:     end if
29:   end for
30: end function

31: function MinimalRecalc(roots)
32:   for root in roots do
33:     MarkDirty(root)
34:     Enqueue(Q, root)
35:     State(root) ← enqueued
36:   end for
37:   while ¬(IsEmpty(Q) ∨ CycleFound()) do
38:     cell ← Dequeue(Q)
39:     Eval(cell, Q)
40:   end while
41:   if CycleFound() then
42:     ReportCycle()
43:   end if
44: end function
```

For Peer Review

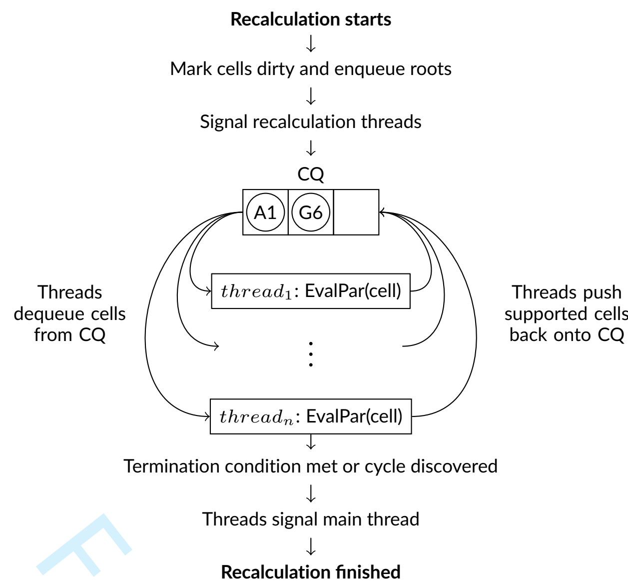


FIGURE 5 Overview of parallel, minimal recalculation.

For Peer Review

5 | PARALLEL FUNCALC

We are now ready to describe the parallel implementation of minimal recalculation in Funcalc which builds on the sequential variant from the previous section. Section 5.1 gives a high-level overview of parallel recalculation, while section 5.2 gives a similar overview from the perspective of the recalculation threads that cooperatively compute cells in parallel. We explain why the termination condition for sequential recalculation, using the size of the queue alone, is no longer sufficient in a parallel context. Lastly, we describe the concept of cell ownership in section 5.3.1 and finish with a description of the parallel cycle detection algorithm in section 5.3.

5.1 | Parallel Recalculation

Parallel minimal recalculation proceeds in a breadth-first fashion like its sequential counterpart but does so in parallel using multiple *recalculation threads* which are controlled by the main thread. The recalculation threads are created at application start-up to match the system's number of logical processors $\mathtt{n}$ and assigned a unique identifier $\mathtt{i} = 1, \ldots, \mathtt{n}$. The overall recalculation process is shown in fig. 5. To start a recalculation, the main thread first marks and enqueues the recalculation roots in a concurrent, evaluation queue CQ, specifically C#'s `ConcurrentQueue` class, that is shared by all threads. All threads initially wait for a signal from the main thread to start recalculation. Once signalled, each thread attempts to dequeue a cell from CQ, evaluates it and pushes its supported cells back onto CQ in parallel. Each recalculation thread has its own instance of the recursive interpreter modified to work in parallel as we will see later in section 5.3.3. The threads continue evaluating cells until the termination condition is met or a cyclic dependency is discovered. We defer a detailed discussion of the parallel termination condition until section 5.2.

The function MinimalRecalcPar in algorithm 2 (line 10) is executed by the main thread. Pseudo-code functions that have been modified for parallel execution are conveniently suffixed with "Par" for "parallel" to distinguish them from their sequential counterparts. A concurrent counter, inspired by Java 8's `LongAdder` class [20], is first initialised to zero (line 11) and passed as an argument to function MarkDirtyPar (line 13). The function marks all cells reachable from the recalculation roots *dirty*, but additionally counts the number of marked cells by atomically incrementing the counter (line 4). This cell count is exactly the number of cells that need to be recalculated to complete the minimal recalculation and serves as our new termination condition. Reading state is now thread-safe via the StatePar function (pseudo-code omitted).

The main thread then signals the recalculation threads using a barrier [21] (line 15), a synchronization construct that waits for $\mathtt{n}$ participant threads to signal it before letting all threads past the barrier. The barrier is initialised to $\mathtt{n} + 1$ participants corresponding to all recalculation threads and the main thread. Thus all $\mathtt{n}$ recalculation threads remain blocked at the barrier until the main thread signals it. The barrier we use automatically resets after all participants have signalled it and can be readily reused. The main thread then immediately signals the barrier again and waits for all recalculation threads to finish. As each recalculation thread finishes evaluation, it signals the barrier. Once all threads have signalled the barrier, they are let past it and immediately signal it again and await a signal from the main thread to start a new recalculation. The main thread checks if

a cyclic dependency was discovered during recalculation (line 17) which is then reported to the user via the graphical user interface via function ReportCycle.

Algorithm 2 Main functions for parallel, minimal recalculation.

```
1: function MarkDirtyPar(cell, counter)
2:   if StatePar(cell) ≠ dirty then
3:     StatePar(cell) ← dirty
4:     Increment(counter)
5:     for u in SupportSet(cell) do
6:       MarkDirtyPar(u, counter)
7:     end for
8:   end if
9: end function

10: function MinimalRecalcPar(roots)
11:   counter ← NewCounter(0)
12:   for root in roots do
13:     MarkDirtyPar(root, counter)
14:   end for
15:   Signal(barrier) // Signal recalculation threads to start
16:   Signal(barrier) // Wait for recalculation threads to finish
17:   if CycleFoundPar() then
18:     ReportCycle()
19:   end if
20: end function
```

5.2 | Recalculation Threads

Algorithm 3 shows the pseudo-code for ThreadRecalculate that is executed by each recalculation thread. The main loop is executed as long as the Running function returns true which it does until the application must close down. As explained in the previous section, each thread waits at the barrier (line 3) and once past it, they enter the main recalculation loop (line 4). The size of the queue is not a sufficient termination condition by itself anymore since cells may be evaluated in parallel while the queue is empty. Instead, we disregard the size of the queue and use the remaining number of unevaluated cells stored in the concurrent counter as the sole termination condition by calling the Get function to atomically query the counter. Function CycleFoundPar checks if a thread-safe, shared flag has been set by a thread that has discovered a cycle. The inner recalculation loop continues for as long as there are cells left to evaluate and a cycle has not been found by one of the threads.

In the body of the loop, each recalculation thread continuously polls CQ for a cell to compute and attempts to evaluate it using function EvalPar (lines 5 to 8). The function ensures that only one thread evaluates the cell while other threads must wait for its value to become available, since threads may try to evaluate the cell simultaneously. Once a cell has been evaluated, EvalPar pushes any supported cells onto the queue and decrements the concurrent counter. Cycle detection is also handled in EvalPar but we defer its implementation till section 5.3.3 until we have given a high-level description of the cycle detection algorithm in section 5.3.

Algorithm 3 The function executed by each recalculation thread.

```
1: function ThreadRecalculate()
2:   while Running() do
3:     Signal(barrier) // Wait for main thread to signal barrier
4:     while Get(counter) > 0 ∧ ¬CycleFoundPar() do
5:       cell ← TryDequeue(CQ)
6:       if cell ≠ null then
7:         EvalPar(cell)
8:       end if
9:     end while
10:    Signal(barrier) // Signal main thread that the thread is done
11:  end while
12: end function
```

5.3 | Parallel Cycle Detection

A key concept underpinning the implementation of the parallel algorithm is *cell ownership*, discussed in section 5.3.1, that allows a recalculation thread to identify which thread is recalculating its dependency. Afterwards, we discuss the idea behind the *reachability matrix* used to record global information on reachability among threads and show how it enables us to detect cycles in section 5.3.2. Finally section 5.3.3 gives the full details of the parallel cycle detection implemented in function EvalPar.

For Peer Review

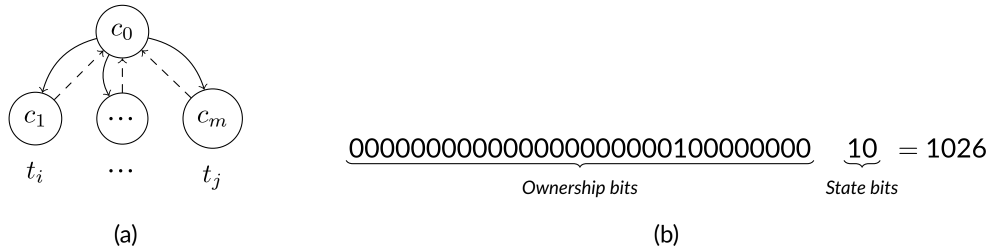


FIGURE 6 (a) A set of recalculation threads evaluating cells $c_1, \ldots, c_m$ that all depend on cell c_0 **(b)** Encoding thread t_9's ownership of a cell in the 30 most significant bits (MSBs) of the cell's 32-bit integer state. The two least significant bits (LSBs) denote the *computing* state (2).

5.3.1 | Cell Ownership

Cell ownership was originally introduced in earlier work [15] for two purposes. First, as a way for threads to claim exclusive ownership of a cell. As threads inevitably race to evaluate cells, e.g. if a cell is in both support sets of two dependencies being evaluated by different threads, we must ensure only one thread can claim exclusive ownership of the cell in a thread-safe manner. Any other threads must wait for the owner to finish computing the cell before retrieving its value. Second, to allow threads to identify the owner thread of a cell dependency whose state is *computing* which is used for cycle detection.

Consider the scenario in fig. 6a where multiple threads are each evaluating cells c_1 to c_m who all depend on a single, unclaimed cell c_0. When a thread examines an unclaimed, unevaluated cell whose state is either *dirty* or *enqueued*, it attempts to atomically set a new state that is a bitwise encoding of the *computing* state and its unique thread identifier. Other recalculation threads can read the encoded state and decode both the cell's owner and actual state. The new encoded state is set using a compare-and-swap (CAS) [21]. In this paper, we adopt the convention that a call Cas(M, A, B) atomically stores A at memory address M if B and the contents already stored at M are equal, then returns true; otherwise the contents stored at M remains unmodified and the call returns false. This ensures that only one CAS will succeed and claim exclusive ownership of the cell while the CAS of other threads will fail.

The four cell states can be encoded in two bits (*dirty*, *enqueued*, *computing* and *uptodate* are assigned numbers zero to three respectively). Ownership is encoded in the remaining bits by flipping the $(i + 2)^{\text{th}}$ bit of a thread t_i with identifier i. The identifier corresponds to the thread identifiers $i = 1, \ldots, n$ assigned at application start-up where n is the system's number of logical processors. An example is shown in fig. 6b for thread t_9 using a 32-bit integer where the state bits are encoded in the two least significant bits (LSBs). Given that threads only own *computing* cells, the ownership bits are only relevant for the *computing* state and are zero for the three other states. Functions for encoding ownership and state and decoding state are shown in algorithm 4. The encoding function EncodeOwner left-shifts the thread's identifier to its appropriate position and uses a bitwise `OR` to add the *computing* state bits. To decode the state bits with function DecodeState, we mask away everything but the two LSBs.

Algorithm 4 Functions for encoding ownership and state and decoding state.

```
1: function EncodeOwner(owner)
2:     return 1 << (2 + owner − 1) | computing
3: end function
4: function DecodeState(encoding)
5:     return encoding & 2
6: end function
```

If we now created a thread-safe version of the Eval function from section 4.2 based on cell ownership, we would soon encounter a fundamental problem with detecting cycles in parallel: an arbitrary number of threads can become stuck waiting for each other in a cycle as shown in fig. 7 for five threads $t_1, \ldots, t_5$ who have each claimed ownership of five different cells that depend on each other in a cycle. A resolution strategy must be in place to break the cycle and avoid the five threads waiting for each other indefinitely. For example, our task-based spreadsheet interpreter [15] used *speculative reevaluation* to detect and resolve cycles in parallel. Threads with lower identifiers were given precedence over threads with higher identifiers. High-precedence threads were allowed to claim a cell already owned by a low-precedence thread. Since thread identifiers are unique and form a total order, some thread in any given cycle will have the highest precedence, claim all cells in the cycle and eventually discover the cyclic dependency.

Our parallel cycle detection algorithm is inspired by a distributed algorithm for cycle detection in large directed graphs [4] where each vertex in the system forwards a set of vertices to their neighbours in iterations using message passing, and each vertex builds a local set of received vertices. Vertices initially forward themselves and subsequently forward an increasing set of vertices passed to them in subsequent iterations. Information propagates further after each consecutive iteration, and in the presence of a cycle, some vertex will eventually receive a set that includes itself and discover the cycle.

For Peer Review

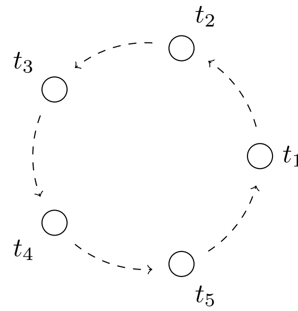
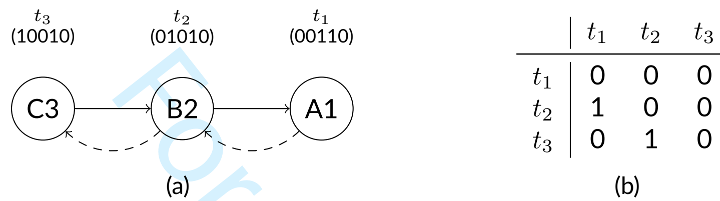


FIGURE 7 Five threads stuck indefinitely in a cell dependency cycle unless the cycle can be detected and resolved.



	t_1	t_2	t_3
t_1	0	0	0
t_2	1	0	0
t_3	0	1	0

FIGURE 8 Example of three recalculation threads updating the reachability matrix R.

For Peer Review

In Funcalc, cycles are discovered through the dependency graph by recursive evaluation of cell references as we discussed in section 4.2 and this is also the case for the parallel implementation. Whereas cycle detection happens across a number of iterations in the distributed algorithm [4], our approach must be dynamic and information on reachability must only be propagated between threads that are computing cells (the state is *computing*) and clear this information when a cell becomes *uptodate*.

5.3.2 | Reachability Matrix

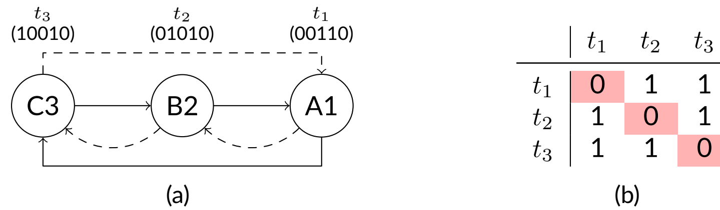
In this section, we establish some intuition for how we detect cycles in parallel before showing the actual implementation. We use a *reachability matrix* R, given by definition 1, to record which threads can be reached by other threads instead of passing around sets of threads.

Definition 1. Let R be a binary reachability matrix indexed by i and j. If $R[i, j] = 1$, we say that thread t_i can be reached by thread t_j and that t_i is currently computing a cell whose state is *computing*. Otherwise, $R[i, j] = 0$ and t_j cannot reach t_i.

In the last part of the definition, we do not require t_i to not be computing a cell since t_j may not yet have discovered t_i. From definition 1, the 1's in a row i in R correspond to which threads can reach thread t_i. To see how R is used, consider the illustrative example in fig. 8a where thread t_1 is evaluating cell A1 which depends on B2 owned by t_2 which in turn depends on C3 owned by t_3. The encoded state is shown in parentheses under each thread. Recall that the two LSBs encode the cell state which is $\mathit{computing} = 2$. When t_1 wants to evaluate B2, it examines the ownership bits of its cell state and discovers it is currently owned by t_2. Thread t_1 therefore sets $R[2, 1] = 1$ to record it can reach t_2; likewise, t_2 can record that it can directly reach t_3 while t_3 itself has no dependencies. The updated matrix R is shown in fig. 8b.

Once a thread is waiting for a dependency owned by another thread, it attempts to discover other threads that can be reached through that dependency by examining R. For example, t_1 can check if it can reach t_3 through t_2 by examining if $R[3, 2] = 1$, which is true, and since $R[2, 1] = 1$, t_1 can set $R[3, 1] = 1$ to record that it can transitively reach t_3 through t_2. Hence, we call such queries *transitive queries* and different thread identifiers are queried in round robin for as long as the owner of the cell dependency is computing, or a cycle is discovered. The only exception is that we do not query the owner of the cell dependency since it is a prerequisite for performing transitive queries to be able to reach it, e.g. t_1 already knows that it can reach t_2 from fig. 8b so querying it again would be redundant. The round robin scheme implies that recalculation threads can query if they can reach themselves.

When t_3 eventually finishes evaluation of cell C3, it must set the cell state to *uptodate*, clear its the ownership bits and reset its row in R. The cell state change signals t_2 that C3 has been evaluated while the update to R signals that t_3 cannot be reached by any other threads (until it starts computing another cell). Setting state and clearing a row in R must happen atomically to ensure correctness as we discuss in section 7. Once t_2 is done as well, it performs the same two actions to signal t_1 that B2's value is *uptodate*. Finally, the value of A1 can be computed by t_1.



	t_1	t_2	t_3
t_1	0	1	1
t_2	1	0	1
t_3	1	1	0

(b)

FIGURE 9 (a) A similar scenario as in fig. 8a except cell C3 now depends on A1 creating a cycle **(b)** A possible, intermediate outcome of R if cell C3 depended on A1.

How does this help us detect cycles? Suppose cell C3 depended on cell A1 as in fig. 9a and each thread establishes all information except enough to discover the cycle. E.g. thread t_3 establishes that it can reach t_1 and subsequently that it can reach t_2 transitively through t_1. According to definition 1, a thread t_i attempting to set $R[i, i] = 1$ in R's diagonal can directly or transitively reach itself via a cyclic reference. These diagonal elements are highlighted with a red background in fig. 9b and the state of R at this point in time is one possible intermediate outcome. Now either t_1, t_2 or t_3 can query if they can transitively reach themselves and discover the cycle. Suppose t_2 performs the transitive query. Since $R[3, 2] = R[1, 3] = R[2, 1] = 1$, t_2 can reach itself and discovers the cycle. This particular sequence of entries in R corresponds to the following path in the dependency graph $t_2 \dashrightarrow t_3 \dashrightarrow t_1 \dashrightarrow t_2$.

For Peer Review

5.3.3 | Cycle Detection Implementation

In practice, R is represented by an array of integers such that each row of the matrix is represented by a separate integer and each column c is represented by the cth bit of each integer. This gives us a light-weight data structure with a one-off allocation cost. Setting entries in R is a simple matter of bit manipulation and a row is cleared by assigning zero to the corresponding integer. Like cell ownership, we encode bits starting from the LSB so e.g. position $R[1, 2]$ represents the second LSB of the first integer in the array. Since R is shared between all recalculation threads, each row is protected by its own lock stored in a separate array. Using locks enables us to set both a cell's state to *uptodate* and clear a row in R in the same critical section, which is necessary for correctness (see section 7.1). Recall that we are using CAS to control access to cell state to claim cells, while we are using locks to control access to R. While we used the subscript in t_i as a thread's identifier in the text, we use a function Id to access the identifier in the following algorithms.

Algorithm 5 lists the functions for updating R as described in section 5.3.2. OwnerToBits converts a thread identifier to its bit representation for encoding into R. Function UpdateOwner is used to initially record the owner of the cell dependency reachable by a thread. It acquires the lock for the row in R of the owner of the calling thread's cell dependency. We check if the cell dependency has become *uptodate* since we acquired the lock in which case we do nothing and release the lock; we must *not* update R since it would incorrectly record we are waiting on a cell which is already *uptodate*. Otherwise, we use OwnerToBits to flip the bit for the current thread and record that it can reach the owner. Then the lock is released. Function UpdateR is called to perform transitive queries. We first attempt to update transitive reachability via a call to TrySetR then check if the update (or an update by another thread) caused the current thread to be able to reach itself by fetching its reachability and calling CanReach (pseudo-code omitted) to check if the thread's bit is set. If so, we return true to indicate that we have found a cyclic reference, otherwise we return false.

Function TrySetR finds the next thread identifier (tid) to transitively query by calling NextQueryId (pseudo-code omitted) which returns identifiers for all threads, including itself, in round robin for each consecutive call, except for its argument. This is the owner of the calling thread's cell dependency which we already know is reachable by the current thread as a prerequisite to performing transitive queries. We then acquire the lock and retrieve the row for the thread with tid (lines 20 to 21) and call CanReach to see if the current thread can reach the thread that we are transitively querying (line 22). If not, we release the lock and return, otherwise we check if the state of the cell has not become *uptodate* before we initially acquired the lock. If the cell is not *uptodate*, we can still reach its owner and encode this information in R using OwnerToBits. Finally, we release the lock. Function ClearR clears reachability in a row by first acquiring the lock for the thread's row, sets the cell's state to *uptodate*, assigns zero to the thread's row in R, then releases the lock. We can safely set the cell's state in the critical section since we have exclusive ownership of the cell when ClearR is called and no other thread will write to it.

At last we present the pseudo-code for the missing piece of parallel minimal recalculation: EvalPar and its remaining auxiliary functions in algorithm 6. EvalPar reads and decodes the cell's state, and similar to the sequential Eval, it takes actions according to the current state.

Algorithm 5 Functions for updating reachability.

```
 1: function OwnerToBits(owner_id)
 2:   return 1 << (owner_id − 1)
 3: end function

 4: function UpdateOwner(thread_id, owner_id, cell)
 5:   lock locks[owner_id]
 6:   if StatePar(cell) ≠ uptodate then
 7:     R[owner_id] ← R[owner_id] | OwnerToBits(thread_id)
 8:   end if
 9:   unlock locks[owner_id]
10: end function

11: function UpdateR(thread_id, owner_id, cell)
12:   TrySetR(thread_id, owner_id, cell)
13:   lock locks[thread_id]
14:   cycle ← CanReach(R[thread_id],thread_id)
15:   unlock locks[thread_id]
16:   return cycle
17: end function

18: function TrySetR(thread_id, owner_id, cell)
19:   tid ← NextQueryId(owner_id)
20:   lock locks[tid]
21:   r ← R[tid]
22:   if CanReach(r, owner_id) then
23:     if StatePar(cell) ≠ uptodate then
24:       R[tid] ← r | OwnerToBits(thread_id)
25:     end if
26:   end if
27:   unlock locks[tid]
28: end function

29: function ClearR(thread_id, cell)
30:   lock locks[thread_id]
31:   StatePar(cell) ← uptodate
32:   R[thread_id] ← 0
33:   unlock locks[thread_id]
34: end function
```

If the state is *computing*, we check if we are following a dependency since we only wish to record reachability and detect cycles along the dependency graph. The thread-local variable isDependency is set to true whenever a cell reference or cell area is recursively evaluated and set to false when it returns. Recording reachability through the support graph would quickly lead to false positives since it would effectively make the graph undirected. If we are not following a dependency, we call CachedPar (line 29) to wait for the cell to become *uptodate*; otherwise, we decode the owner from the ownership bits of the cell state using OwnerFromBits (line 48) which right-shifts away the state bits and use the logarithmic function to base 2 to convert the flipped bit's position to a thread identifier. We must add one to convert the zero-based result of the logarithmic function to a one-based thread identifier. If we own the cell (lines 7 to 8), we call NotifyCycleAndStopPar (pseudo-code omitted) which atomically sets a shared flag so that CycleFoundPar returns true and clear reachability for the current thread. Recall that CycleFoundPar is part of the termination condition that is checked in the main recalculation loop by every recalculation thread in algorithm 3 in section 5.2. Setting the cell's state to *uptodate* before finishing evaluation allows any pending threads to finish computing with a stale value and then immediately find that a cycle has been discovered when they next call CycleFoundPar. If we do not own the dependency, we record its owner in R via UpdateOwner, then enter a loop where we continuously update R via transitive queries until the cell becomes *uptodate* or a cycle is discovered (lines 11 to 16).

If the cell state is *dirty* or *enqueued*, we try to evaluate the cell via EvalExprPar (line 21). Function EvalExprPar rereads the current state of the cell, and if it is still either *dirty* or *enqueued*, attempts to claim the cell using a CAS with the new encoded state of the current thread. If we successfully claim the cell, EvalExprPar evaluates the cell via EvalExpr, decrements the global, atomic cell counter, clears reachability information for the current thread, enqueues any supported cells and returns true (lines 22 to 24). Notice that due to the exclusivity of cell ownership only one thread performs these actions and we can safely call the sequential EvalExpr to evaluate the cell. If the CAS fails, we return false and go to the *uptodate* case and wait for the cell to finish being evaluated.

Function EnqueueSupportPar (line 41) also uses a CAS to change the state of each supported cell from *dirty* to *enqueued*, so a cell is only ever enqueued once even if two or more threads attempt to enqueue a cell simultaneously. The CachedPar function (pseudo-code omitted) used on line 29 in the *uptodate* case continuously spins until the cell's state becomes *uptodate* before returning its value.

6 | RESULTS

We benchmarked our algorithm on six spreadsheets from the LibreOffice Calc spreadsheet program [22] and six synthetic spreadsheets with different topologies. These are described below and some of their properties are listed in table 1. The **Roots** column denotes the total number of recalculation roots and the **Support** column denotes the total number of edges in the support graph. Built-in functions listed in the table are implemented in C#. 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 3 warm-up runs and computed the average of 20 runs for each benchmark. The average runtimes are listed for all spreadsheets in seconds in table 2.

For Peer Review

Algorithm 6 Functions for evaluating a formula in parallel.

```
 1: function EvalPar(cell)
 2:   encoded_state ← StatePar(cell)
 3:   thread_id ← Id(GetCurrentThread())
 4:   switch DecodeState(encoded_state)
 5:     case computing :
 6:       if isDependency then
 7:         owner ← OwnerFromBits(encoded_state)
 8:         if thread_id = owner then
 9:           NotifyCycleAndStopPar(cell) // We own the cell
10:         else
11:           UpdateOwner(thread_id, owner, cell)
12:           while StatePar(cell) ≠ uptodate do
13:             if UpdateR(thread_id, owner, cell) then
14:               NotifyCycleAndStopPar(cell)
15:             end if
16:           end while
17:         end if
18:       end if
19:       break
20:     case dirty or enqueued :
21:       if EvalExprPar(cell, thread_id) then
22:         Decrement(counter)
23:         ClearR(thread_id, cell)
24:         EnqueueSupportPar(cell, CQ)
25:       end if
26:       break
27:     case uptodate :
28:       break
29:   return CachedPar(cell)
30: end function

31: function EvalExprPar(cell, id)
32:   state ← StatePar(cell)
33:   if DecodeState(state)<computing and
34:       Cas(StatePar(cell), EncodeOwner(id), state) then
35:     CachedPar(cell) ← EvalExpr(cell)
36:     return true
37:   else
38:     return false
39:   end if
40: end function

41: function EnqueueSupportPar(cell, cq)
42:   for u in SupportSet(cell) do
43:     if Cas(State(cell), enqueued, dirty) then
44:       Enqueue(cq, u)
45:     end if
46:   end for
47: end function

48: function OwnerFromBits(encoding)
49:   return log2(encoding >> 2) + 1
50: end function
```

6.1 | LibreOffice Spreadsheets

The spreadsheets from LibreOffice Calc were chosen to represent a set of realistically structured spreadsheets. Functions that were not supported by Funcalc were implemented as SDFs (original spreadsheets available at https://gerrit.libreoffice.org/gitweb?p=benchmark.git;a=tree). To start a recalculation to evaluate all cells in the spreadsheet, we find all recalculation roots whose collective transitive closure in the support graph covers all cells. This resembles a full recalculation but performs a partial recalculation instead and enables us to better examine the scalability of the algorithm and how well it can dynamically find local parallelism.

The speed-ups relative to sequential recalculation are shown in fig. 10 and we obtain overall positive speed-ups. The `building-design`, `ground-water` and `stock-history` spreadsheets exhibit better scalability (11.58x-14.64x speed-ups) than the `energy-market`, `grossprofit` and `stocks-price` spreadsheets (7.14x-9.51x speed-ups). Hyperthreading benefits all spreadsheets as performance continues to grow, albeit more slowly in some cases, beyond 24 cores on our 48 logical core machine. Locking in the cycle detection algorithm causes threads to be descheduled by the operating system which may give hyperthreading time to execute another thread on the same core. Neither the algorithm's use of CAS or locks appear to slow down performance for an increased number of threads. Another reason for the decreasing speed-up of the `energy-market`, `grossprofit` and `stocks-price` spreadsheets may be their use of SDFs. Funcalc's current compiler currently could be improved to include more type information in order to generate more efficient code. We discuss this more in section 7.5.

6.2 | Synthetic Spreadsheets

To test how the algorithm adapts to different topologies, we generated six synthetic spreadsheets with specific structures. Examples of the structure of their support graph structure are shown in fig. 11. A single cell serves as a recalculation root that ensures that every other cell in the spreadsheet is computed. This is the striped cell in fig. 11. The evaluation time for each cell in the synthetic spreadsheets was tailored to be approximately equal to the average evaluation time for cells in the LibreOffice Calc spreadsheets, and the number of cells was set to match a sequential running time close to the slowest runtime of the LibreOffice spreadsheets. To gain some control over the evaluation time of a cell, we defined a built-in function `LOOP` taking a single argument n which runs a for-loop for $i = 0, \ldots, n$ iterations and calculates the total sum of the loop variable i i.e. $\sum_{i=0}^{n} i$.

The speed-up results in fig. 12 suggest that the algorithm is largely topology-agnostic with satisfactory overall speed-ups between 21.04x and 32.42x. The `fork`, `fork-join` and `map` spreadsheets contain the most independent computations and also achieve the best speed-ups. The `binary-join` and `binary-tree` spreadsheets, while having less formulas than the previous three, are more connected which is reflected in the lower speed-up. Finally, we achieve the least speed-up on the highly-connected `prefix` spreadsheet (see the **Support** column in table 1). This is

For Peer Review

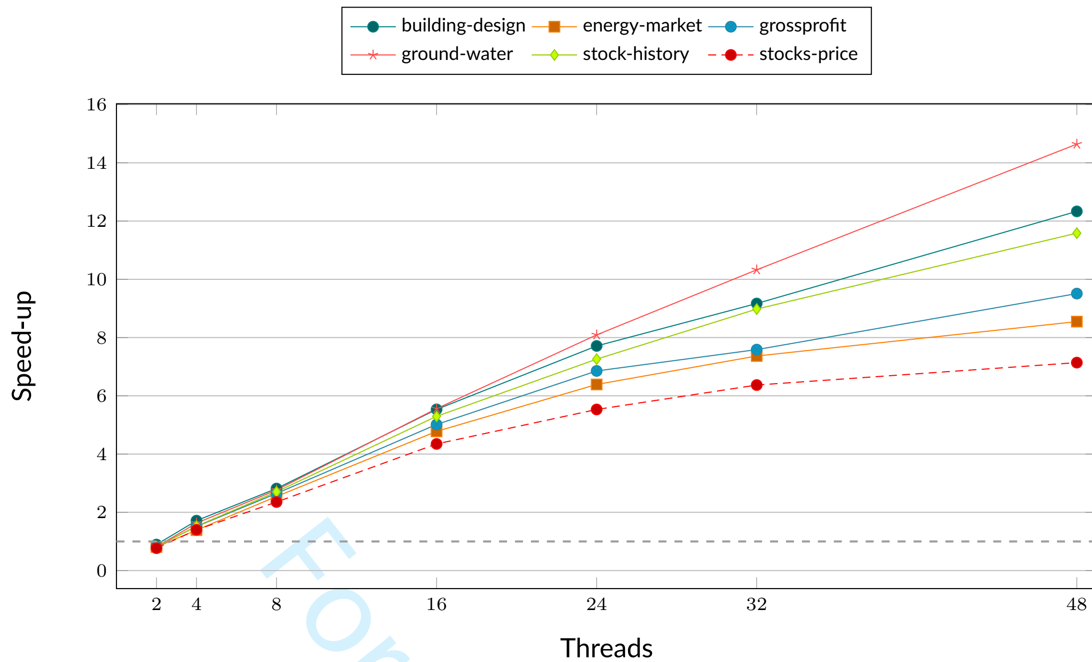


FIGURE 10 Speed-ups for the six LibreOffice Calc benchmark spreadsheets compared to sequential recalculation. The grey, dashed line is the sequential baseline.

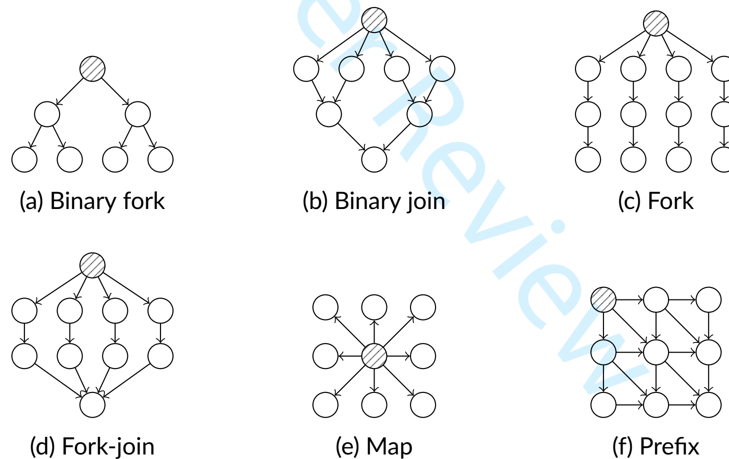


FIGURE 11 Representative examples of the underlying support graphs of the synthetic spreadsheets. Each striped node denotes the recalculation root which is used to start a minimal recalculation that ensures all cells are evaluated.

to be expected as the high connectivity increases contention between threads and they will inevitably wait on other threads more often. We still achieve a maximum 15x speed-up on 48 logical cores.

7 | DISCUSSION

We chose to implement parallel cycle detection using locks but they are not without downsides. In general, they can deadlock, do not tend to scale well, threads can starve if they never get to acquire the lock, and if a thread crashes while holding a lock the system can stall indefinitely. Fortunately, deadlock is not an issue for our algorithm due to the absence of nested lock acquisition. Another concern might be that we are using both CAS and locking in the parallel algorithm. However, CAS is only ever used to claim cells. Once claimed, the owner thread has exclusive access

For Peer Review

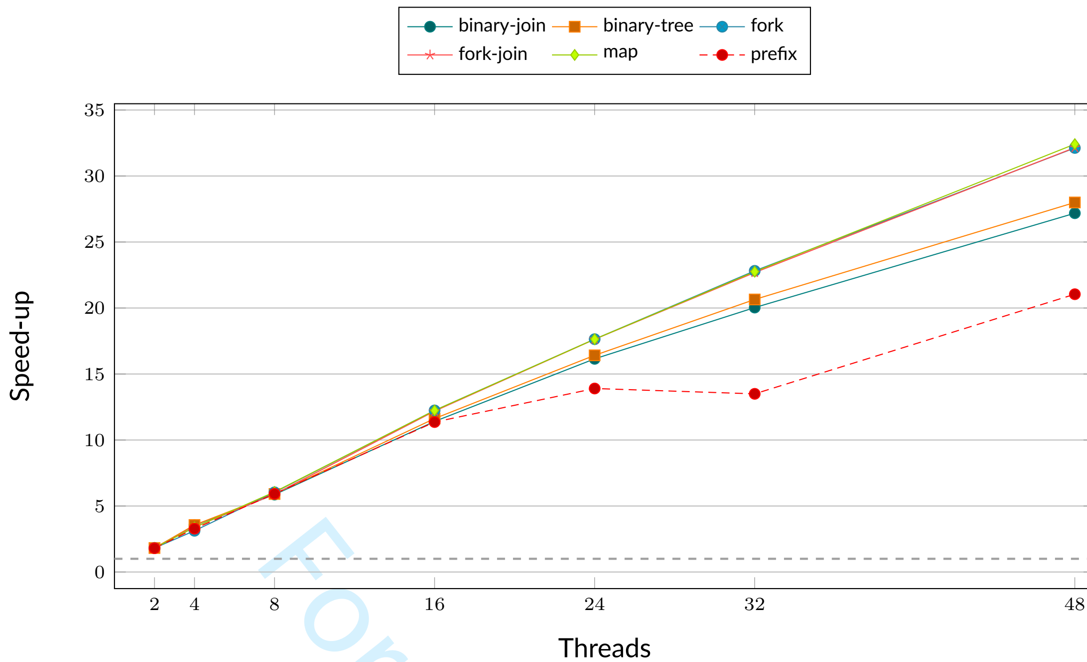


FIGURE 12 Speed-ups for the six synthetic spreadsheets compared to sequential recalculation. The grey, dashed line is the sequential baseline.

to the cell and its state and all other threads simply read its state and wait for it to become *uptodate*. When the owner writes to the cell state in the critical section (see function ClearR in algorithm 5), it has sole ownership of the cell and other the update will become visible to other threads when the lock is released.

In the following subsections, we discuss some alternative implementations of the algorithm, how we can adapt it to perform a full recalculation and finally we state some threats to validity.

7.1 | Lock-Free Implementation of Cycle Detection

We believe a lock-free implementation of cycle detection is possible but would require more sophisticated hardware instructions than CAS. A crucial property of the cycle detection algorithm is that it must not be possible for a recalculation thread t_i to finish evaluating a cell, clear its reachability and then continue recalculating while some other thread t_j believes that it can still reach t_i. Precisely our choice of locks allows us to perform both actions in the same critical section and avoid creating a window where some thread sees one update and reacts before seeing the second. CAS alone cannot perform both actions atomically and a naive implementation quickly breaks down. To see this, suppose we used two CAS for these operations and consider the series of actions given in the timeline in fig. 13 where thread t_2 is computing a cell that depends on a cell owned by t_1. Every action is done using CAS. Initially, t_2 observes that its dependency is in the *computing* state. Thread t_1 finishes computing the cell and sets its state to *uptodate* then clears its entry in R using two separate CAS operations. Having already observed that the dependency is *computing*, t_2 subsequently must update R but does so under the assumption that the cell state is still *computing*, incorrectly recording that it can reach t_1. Later, t_1 might erroneously discover itself. Using locks, we avoid this kind of interleaving since we ensure either t_1 acquires the lock, sets the cell state to *uptodate* and clear its row in R; when t_2 acquires the same lock, it sees that the cell has become *uptodate* and continues evaluation. Otherwise, t_2 acquires the lock, observes that the cell state is still *computing* and sets the entry in R before releasing the lock to allow t_1 to acquire it and finish evaluation.

One alternative to CAS is the load-link (LL) and store-conditional (SC) instruction pair. The LL instruction loads a memory address and the SC instruction stores a value at the same memory address. As opposed to CAS, the SC instruction also checks if the memory location was modified since the last LL instruction regardless of which value is stored there. If, in the example timeline of fig. 13, an LL instruction was used to load the cell state and entry in R, either of the subsequent SC instructions would fail since t_1 modified both between the load and the store. Unfortunately, these instructions are not as widely available as the CAS instruction although Brown et al.[23] provide the load-link extended (LLX) and store-conditional extended (SCX) synchronization primitives that can be implemented in software using CAS. Yet another solution would be instructions like double-length compare-and-swap (DCAS) or multi-word compare-and-swap (MCAS) where we can atomically set both the state and a row in R simultaneously, but such instructions are not widely available either.

For Peer Review

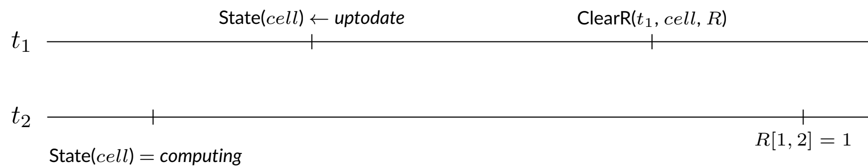


FIGURE 13 A naive lock-free implementation of the parallel cycle detection algorithm can lead to incorrect states.

7.2 | Thread-local Queues

When there is only a single cell in a cell's support set, it would be wasteful to enqueue it in the global queue as the thread could just directly evaluate it instead and save concurrent pulls from the global queue in search of more work. In our previous work [15] on a task-based spreadsheet interpreter, a variant was developed that used thread-local queues for evaluating such sequential chains of supported cells. The algorithm presented in this paper could be similarly extended where the single supported cell is enqueued in a recalculation thread's local queue. However, observe in table 1 that most of our real-world benchmark spreadsheet do not have a large span, and it is doubtful whether most realistic spreadsheets would benefit from such an extension; it did not prove fruitful in the results for the task-based interpreter but further testing should be conducted to verify the value of this optimisation.

Alternatively, we could enqueue a single cell in the thread-local queue regardless of the total number of supported cells to minimise overall contention on the global queue. If there is only a single supported cell, this implementation would act identically to the variant of the task-based implementation [15] described above. One could even consider enqueueing more supported cells in the thread-local queue to further minimise global contention but still retain some the desirable load-balancing properties that are normally provided by using a global queue.

7.3 | Representation of the Reachability Matrix

By representing the reachability matrix R as an array of integers, we limit scalability to the number of bits in the largest integral type supported by the CAS intrinsic of the system, save for two bits for the cell state. On our 64-bit Intel Xeon machine, we can only scale to a maximum of 62 threads. However, since our algorithm chiefly targets commodity hardware the current scalability should suffice for today's systems. Hopefully as hardware supports an increasing number of threads, the size of integral types and CAS instructions will evolve alongside it.

Despite limiting scalability, R has a one-off allocation cost and very little overhead. Still, one might consider other data structures to circumvent this limitation. A simple extension would be to employ arrays of integers that would be allocated once at application start-up. Setting an entry equates to setting an array index and clearing a row in R to zeroing out an array. Instead of encoding the bit position corresponding to a thread identifier, we encode the actual numerical value of the thread identifier in the ownership bits.

7.4 | Adapting the Algorithm To Full Recalculation

In section 2.3, we mentioned that our algorithm for parallel minimal recalculation could easily be adapted to perform a full recalculation. In fact, we have already done so when benchmarking the LibreOffice Calc spreadsheets: we find the cells that make up the roots of the support graph, disregarding cells containing constants but still consider cells that depend on those constants. All other cells in the spreadsheet are contained in the transitive closure of these cells so that when recalculated, we ensure that all cells in the spreadsheet are evaluated.

7.5 | Threats to Validity

The `LOOP` function used in the synthetic spreadsheets was implemented as a built-in function in C# as opposed to an SDF. Being a research prototype, Funcalc's SDF compiler still lacks some features found in modern compilers. For example, its type system could be improved to generate more efficient code. When benchmarking our task-based interpreter [15], a recursive SDF for computing Fibonacci numbers was used in the synthetic spreadsheets to control cell evaluation time, but suffered from excessive boxing/unboxing of argument and return values between recursive calls due to a lack of sufficient type information in the compiler. A bachelor project [24] sought to improve the compiler's type system and gained speed-ups for most benchmarks, especially for recursive functions. In order to showcase the algorithms capability of achieving satisfactory speed-ups on the synthetic spreadsheets without the results being overshadowed by other factors such as compiled code of SDFs, we chose to implement `LOOP` as a built-in function. As can be seen from table 1, the LibreOffice spreadsheets also use quite a few SDFs, although none are recursive, and will

For Peer Review

in part also suffer from boxing and unboxing of values. This is also supported by the fact that the `ground-water` spreadsheet uses no SDFs and achieves the highest speed-up for the LibreOffice spreadsheets.

We also made several performance improvements to the Funcalc spreadsheet platform not related to our parallel algorithm and which were not present in the task-based version of Funcalc [15]. Therefore, comparing the algorithm presented in this paper to the task-based algorithm would not be a fair comparison.

The LibreOffice Calc spreadsheets all contain a large amount of data-parallel computation. Indeed, they were used to show how LibreOffice Calc could rewrite such expressions to OpenCL kernels that were executed on AMD GPUs [13]. It would be insightful to run our algorithm on additional real-world spreadsheets with more diverse structures. Unfortunately, it is usually difficult to find larger publicly available spreadsheets with sufficient computation for such purposes since private and public firms tend to keep their spreadsheets to themselves to protect their business.

8 | CONCLUSION AND FUTURE WORK

In this paper, we have shown a dynamic, parallel algorithm that automatically and transparently recalculates the cells in a spreadsheet without requiring interaction from end-users. The algorithm uses CAS for claiming ownership of cells and a lock-based method for cycle detection where recalculation threads gather reachability information to discover cycles. Our results show good speed-ups on a set of benchmark and synthetic spreadsheets. Our parallel recalculation algorithm together with the additional expressive power of higher-order SDFs make the Funcalc spreadsheet application a powerful framework for end-user development.

There are multiple interesting directions for future work. First, we have suggested some variations of the algorithm in section 7 which should be explored in future work. One such variation uses thread-local queues to minimise contention on the global queue. Second, future work should strive to obtain more diverse large-scale, realistic spreadsheets for benchmarking to test the algorithm's adaptive capabilities in relation to different topologies and layout of cells. Third, the type system of the internal SDF compiler could be improved to generate better code [24].

ACKNOWLEDGEMENTS

The author would like to thank Peter Sestoft for his guidance during the writing of this paper. This work has been supported by the Independent Research Fund Denmark (grant number DFF-FTP-4005-00141), Popular Parallel Programming (P3) 2015-2019.

References

1. Counts and earnings of end-user developers. https://www.linkedin.com/pulse/counts-earnings-end-user-developers-chris-scaffidi?published=t Published September 21, 2017. Accessed October 19, 2017.

2. Corecalc and Funcalc Spreadsheet Technology in C#. http://www.itu.dk/people/sestoft/funcalc/ Updated August 3, 2017. Accessed June 1, 2016.

3. Sestoft P. *Spreadsheet Implementation Technology*. The MIT Press; 2014.

4. Rocha R, D Thatte B. Distributed cycle detection in large-scale sparse graphs. In: *SBPO*. Pernambuco, Brazil; 2015:11. **doi: 10.13140/RG.2.1.1233.8640**.

5. **Johnston WM, Hanna JRP, Millar RJ**. Advances in Dataflow Programming Languages. *ACM Comput Surv*. 2004; 36(1): 1–34.

6. Bock AA. *A Literature Review of Spreadsheet Technology*. Copenhagen: IT University of Copenhagen; 2016:33. Available at: https://forskningsdatabasen.dk/en/catalog/2350168960.

7. Abramson D, Roe P, Kotler L, Mather D. ActiveSheets: Super-computing with spreadsheets. In: *HPC*. 2001:5. Available at https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.25.9599.

8. Abramson D, Sosic R, Giddy J, Hall B. Nimrod: a tool for performing parametrised simulations using distributed workstations. In: HPDC. :IEEE; 1995;10. **doi: 10.1109/HPDC.1995.518701**

For Peer Review

9. HPC Services For Excel. https://technet.microsoft.com/en-us/library/ff877820(v=ws.10).aspx Published February 7, 2015. Updated June, 2011. Accessed June 30, 2016.

10. Wack AP. Partitioning Dependency Graphs for Concurrent Execution: A Parallel Spreadsheet on a Realistically Modeled Message Passing Environment. 1996.

11. Biermann F, Dou W, Sestoft P. Rewriting High-Level Spreadsheet Structures into Higher-Order Functional Programs. In: *PADL, LNCS(10702)*. :Springer International Publishing; 2018:16. **doi: 10.1007/978-3-319-73305-0_2**

12. Dou W, Cheung SC, Wei J. Is Spreadsheet Ambiguity Harmful? Detecting and Repairing Spreadsheet Smells due to Ambiguous Computation. In: *ICSE, ACM*. ; 2014:11. **doi: 10.1145/2568225.2568316**

13. Collaboration and Open Source at AMD: LibreOffice. https://developer.amd.com/collaboration-and-open-source-at-amd-libreoffice/ Published July 15, 2015. Accessed July 31, 2015.

14. Meeks M. Calc: The challenges of scalable arithmetic. (Presented at FOSDOM). 2018.

15. **Bock AA, Biermann F**. Puncalc: task-based parallelism and speculative reevaluation in spreadsheets. *J Supercomput*. 2019. **doi: 10.1007/s11227-019-02823-8**

16. **Leijen D, Schulte W, Burckhardt S**. The Design of a Task Parallel Library. *SIGPLAN Not.*. 2009; 44(10): 227–242. **doi: 10.1145/1639949.1640106**

17. Bock AA. Static Partitioning of Spreadsheets for Parallel Execution. *PADL*. 2019: 221–237. **doi: 10.1007/978-3-030-05998-9_14**

18. Bock AA, Bøgholm T, Sestoft P, Thomsen B, Thomsen LL. *Concrete and Abstract Cost Semantics for Spreadsheets*. Copenhagen: IT University of Copenhagen; 2018:56. Available at: https://forskningsdatabasen.dk/en/catalog/2446847276.

19. **Bock AA, Bøgholm T, Sestoft P, Thomsen B, Thomsen LL**. On the semantics for spreadsheets with sheet-defined functions. *J Comput Lang*. 2020; 57. **doi: https://doi.org/10.1016/j.cola.2020.100960**

20. Class LongAdder. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAdder.html Accessed February 21, 2017.

21. Herlihy M, Shavit N. *The Art of Multiprocessor Programming*. Morgan Kaufmann; 2011.

22. The Document Foundation. *LibreOffice Calc*. 2010.

23. Brown T, Ellen F, Ruppert E. Pragmatic Primitives for Non-blocking Data Structures. In: *PODC, ACM*. New York: ACM; 2013:10. **doi: 10.1145/2484239.2484273**

24. Skovrup HS, Røygaard JC. Optimization of sheet-defined functions in spreadsheet compiler. 2019.

For Peer Review

For Peer Review

TABLE 1 Different properties of the LibreOffice Calc and synthetic spreadsheets used for benchmarking.

Spreadsheet	Formulas	Roots	Span	Support	Builtin Functions	Sheet-Defined Functions
LibreOffice Calc Spreadsheets						
building-design	108,332	18,378	4	488,351,887	SUM	PRODUCT2, PRODUCT3, PRODUCT4
energy-market	534,507	35,198	3	287,818,610	AVERAGE	POWER, PEARSON, SLOPE, SUMMAP
grossprofit	135,073	15,301	3	112,612,549	AND, COUNTIF, IF	COUNTIFS, PRODUCT, SUMIFS, SUMMAP
ground-water	126,404	31,601	1	1,099,366,302	AVERAGE, MIN, MAX	-
stock-history	226,503	23,402	3	317,049	AVERAGE, SUM	COUNT, SUMMAP, SUMPRODUCT
stocks-price	812,693	10,876	3	233,376,389	AVERAGE, ROWS, COLUMNS	COVAR, DIFFPOWER2, MULTDIFF, SUMMAP, VAR
Synthetic Spreadsheets						
binary-join	262,146	1	18	393,215	-	LOOP
binary-tree	266,145	1	17	262,143	-	LOOP
fork	300,001	1	1001	300,301	-	LOOP
fork-join	300,002	1	1001	300,600	-	LOOP
map	300,001	1	1	300,001	-	LOOP
prefix	300,000	1	1100	745,009	-	LOOP

TABLE 2 Absolute running time in seconds for sequential and parallel recalculation, rounded to two decimal points. A column for 24 threads was added as it matches the maximum number of physical processors available on our test machine. The final column shows speed-up for 48 threads. Standard deviation was below 0.1 in most cases and only between 0.1 and 2.81 otherwise.

Spreadsheet	1	2	4	8	16	24	32	48	Max Speed-up
LibreOffice Calc Spreadsheets									
building-design	12.91	14.46	7.53	4.58	2.33	1.67	1.41	1.05	12.33
energy-market	62.28	76.97	44.85	24.47	13.03	9.75	8.45	7.29	8.55
grossprofit	65.10	78.92	43.01	24.58	12.98	9.50	8.58	6.85	9.51
ground-water	27.17	33.35	16.68	9.83	4.89	3.36	2.63	1.86	14.64
stock-history	22.68	27.80	14.88	8.38	4.28	3.12	2.53	1.96	11.58
stocks-price	31.87	41.45	22.77	13.54	7.33	5.76	5.00	4.46	7.14
Synthetic Spreadsheets									
binary-join	47.14	26.08	13.31	8.04	4.13	2.92	2.35	1.73	27.18
binary-tree	47.06	25.91	13.22	7.96	4.05	2.87	2.28	1.68	28.00
fork	53.64	29.56	17.18	8.89	4.38	3.04	2.35	1.67	32.13
fork-join	53.70	29.52	15.75	9.11	4.41	3.05	2.37	1.67	32.15
map	53.86	29.49	15.92	8.88	4.40	3.05	2.37	1.66	32.42
prefix	53.43	29.62	16.30	9.02	4.70	3.84	3.96	2.54	21.04

How to cite this article: Alexander Asp Bock (2019), A Parallel Spreadsheet Interpreter With Cycle Detection, *X*, *2019;XX:Y–Z*.

For Peer Review