The Journal of Supercomputing (2020) 76:4977–4997
https://doi.org/10.1007/s11227-019-02823-8

Puncalc: task-based parallelism and speculative reevaluation in spreadsheets

Alexander Asp Bock[1] · **Florian Biermann**[1]

Published online: 23 March 2019

Abstract
Spreadsheets are commonly declarative, first-order functional programs and are used as organizational tools, for end-user development and for educational purposes. Spreadsheet end users are usually domain experts who use spreadsheets as their main computational model, but are seldom trained IT professionals who can leverage today's abundant multicore processors for spreadsheet computation. In this paper, we present an algorithm for automatic, parallel evaluation of spreadsheets targeting shared-memory multicore architectures, which lets end users transparently make use of their multicore processors. We evaluate our algorithm on a set of synthetic and real-world spreadsheets and obtain up to 16 times speedup on 48 cores.

Keywords Spreadsheets · Parallelism · Tasks · Speculative · Declarative programming · End-user programming

1 Introduction

Spreadsheets are abundant in many application areas, such as science and finance, where they are used as organizational tools [1], for end-user development [7, 8] and for educational purposes [2, 12].

Spreadsheet end users are usually domain experts and use spreadsheets as their main computational model, but are seldom trained IT professionals. They create and maintain

Alexander Asp Bock and Florian Biermann have contributed equally to this article.

Alexander Asp Bock: Supported by the Independent Research Fund Denmark (Grant No. DFF-FTP-4005-00141). Florian Biermann: Supported by the Sino-Danish Center for Education and Research (SDC).

✉ Alexander Asp Bock
albo@itu.dk

Florian Biermann
fbie@itu.dk

[1] IT University of Copenhagen, Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark

large, complex spreadsheets over several years [14], and their complexity often leads to errors [11] and poor performance. For instance, Swidan et al [23] report on a case study of refactoring a spreadsheet that would originally take 10 h to recompute.

In recent years, multicore processors have become ubiquitous in commodity hardware. For end users to leverage the performance of multicore systems, it has until now been necessary to hire experts to reengineer spreadsheets [23].

How can end users hope to directly leverage the parallel programming power at their disposal to accelerate the computation of slow spreadsheets without requiring help from such expensive experts?

Spreadsheets are declarative, first-order purely functional programs [9]. Declarative languages enable end users to focus on specifying *what* needs to be computed without having to worry about *how* it gets computed. Being purely functional, all values are immutable which guarantees data-race freedom and allows for implicit parallelization of computations without putting the burden of traditional, shared mutable multicore programming on the end user. These aspects make spreadsheets a prime candidate for automatic parallelization.

In this paper, we present an algorithm for automatic, parallel evaluation of spreadsheets targeting shared-memory multicore architectures. We have implemented our algorithm in the experimental spreadsheet engine Funcalc [21] which introduces efficient, sheet-defined, higher-order functions to the spreadsheet paradigm. The combination of these sheet-defined functions (SDFs) and our parallel recalculation algorithm can contribute to change the general perception of spreadsheets as not being "real" programming languages [9, 10, 20, 27] and enable end-user programmers to use spreadsheets for heavyweight computations with a more reusable, modular, safer and scalable programming platform. Our key contributions are:

- A parallel, task-based, topology-agnostic algorithm for minimal recalculation of spreadsheets, implemented in Funcalc.
- An accompanying extension of the algorithm for dynamic parallel cycle detection called speculative reevaluation.
- A thread-local evaluation optimization that exploits a specific spreadsheet topology on the fly.
- Evaluation of a set of benchmarks for different types and sizes of spreadsheets with different characteristics and topologies.

To our knowledge, no such algorithm for parallel evaluation of spreadsheets with dynamic cycle detection has previously been proposed. Our benchmarks show that we achieve between 1.4 and 6.5 times speedup on 16 cores and nearly a 16-fold speedup on 48 cores.

	A	B
1	42	=A1*23
2		
3	=-A1	=A1+B1

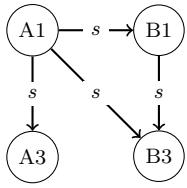

Fig. 1 A spreadsheet containing formulas (left) and its corresponding support graph (right). We label supporting edges with *s*. Editing A1 will recalculate cells A1, B1, A3 and B3; changing B1 will recalculate B1 and B3; and editing A3 will only recalculate A3

2 Background: spreadsheet concepts

In this section, we will introduce some of the basic concepts at the core of the spreadsheet paradigm that are necessary for understanding our algorithm. Readers already familiar with the subject can skip this section, while those interested in learning more are encouraged to read [21, Chapter 1].

2.1 Formulas and cell references

A cell in a spreadsheet contains either a constant, such as a number, a string or an error (e.g. `#NA` or `#DIV/0!`), or a formula expression, indicated by a leading equals character, e.g. `=1+2`. Each cell is denoted by its column and row, where columns start at `A` and rows at 1.

A formula can refer to other cells by naming their column and row address; this establishes dependencies between cells. For example, B3 refers to the cell in the second column and third row. The formula `=A1+B1` refers to two cells, and its value depends on the contents of those cells. Formulas may also refer to a rectangular cell area using the `:` operator by referring to two of its opposing corners, or call functions. For example, computing the sum of all values in the first ten rows of column A may be expressed as `=SUM(A1:A10)`.

2.2 The support and dependency graphs

While the *dependency graph* of a spreadsheet captures cell dependencies, its inverse, the *support graph*, captures cell support. The support graph is analogous to a dataflow graph [15], where nodes are cells and data flows along the edges from precedent cells to dependent cells. In Fig. 1, cell B3 contains the formula `=A1+B1`, which means that B3 depends on A1 and B1, and A1 and B1 support B3.

The dependency and support graphs may be cyclic. In Fig. 2, cells A1 and B1 conditionally refer to one another; this is a *static* cycle. It may or may not cause a *dynamic* cycle during recalculation (see Sect. 2.3) depending on whether `RAND()` evaluates to a value less than 0.5 or not. Cyclic references are usually disallowed,

	A	B
1	=B1+1	=IF(RAND()<0.5, A1, B2)
2		=NOW()

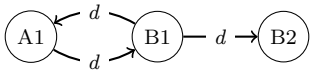

Fig. 2 A static cyclic reference in a spreadsheet (left) and its corresponding cyclic dependency graph (right). We label dependency edges with d

because recalculation cannot proceed meaningfully. When a cyclic reference is found, the user is commonly alerted through the GUI and recalculation aborts.

Some functions like `ROW` and `INDIRECT`, which are found in both Excel and LibreOffice Calc, can dynamically refer to other cells by interpreting argument strings as cell references. They may cause dynamic cycles even when their dependencies are not statically given. For example, to evaluate the formula `=INDIRECT("B"&A1)`, the string `"B"` and the contents of cell A1 are concatenated with the string concatenation operator `&`. The resulting string is interpreted as a cell reference which will refer to some cell in column B, but exactly which cell will depend on the contents of cell A1. In Sect. 4.7, we show how these dynamic indexing functions are handled by our parallel algorithm.

2.3 Recalculation

There are two 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 and all volatile cells. We call the cells that start a minimal recalculation the *recalculation roots*. In Fig. 1, if a user changes the formula in B1 from `=A1*23` to `=A1+A3`, then B1 is a recalculation root and both B1 and B3 must be updated to reflect the change.

As noted above, some cells are marked as volatile because they contain calls to volatile functions such as the `NOW` and `RAND` functions in Fig. 2, and are automatically reevaluated regardless of whether the user modified the cell or not. The function `NOW` returns the current time and would otherwise contain a stale value if it was not recomputed. Similarly, if a cell calling `RAND` was not recomputed on each recalculation, it would only produce a single random number and the call to `IF` in cell B1 would be stuck taking one branch until the cell was evaluated again.

2.4 Consistency requirements

The purpose of recalculation is to bring the spreadsheet to a *consistent state* [21, sec. 1.8.3]. Let ϕ be a mapping from cells to formula expressions and σ a mapping from cells to their values. The former models the underlying computations and dependencies between cells in a spreadsheet, whereas the latter models the result of a specific recalculation. Non-formula cells (i.e. constants) are not in the domain of ϕ. An evaluation has the form $\sigma \vdash e \Downarrow v$ and says that expression e may evaluate to a value v, given the mapping σ. We can define the following consistency requirement:

$$\mathrm{dom}(\sigma) = \mathrm{dom}(\phi) \tag{1}$$

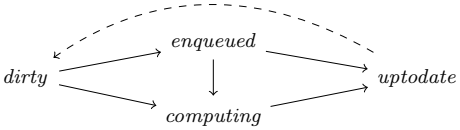

Fig. 3 The possible state transitions of a cell. The dashed connection denotes marking cells dirty in the transitive closure of a recalculation root at the beginning of a particular recalculation

$$\forall c \in \mathrm{dom}(\phi) \;\cdot\; \sigma \vdash \phi(c) \Downarrow \sigma(c) \tag{2}$$

Requirement (1) states that the domains of ϕ and σ must be the same. This implies that a recalculation does not evaluate constants as they involve no work. Requirement (2) states that for every cell c in the domain of ϕ, and thus also in the domain of σ by way of (1), the evaluation of its formula $\phi(c)$ must agree with the value of $\sigma(c)$. The consistency requirements do not specify how recalculation must otherwise proceed, sequentially, in parallel, or in which order cells must be evaluated.

A cyclic dependency can be resolved by assigning the cell c that contains the cyclic reference a `#CYCLE!` error which is then propagated across cells such that the spreadsheet eventually assumes a consistent state.

3 Funcalc: sequential implementation

Funcalc is an experimental spreadsheet engine [21] implemented in C# that adds sheet-defined, higher-order functions to the spreadsheet paradigm. In this section, we solely focus on Funcalc's existing *sequential* implementation of minimal recalculation.

In Funcalc, cells are assigned a state, which is either *dirty*, *enqueued*, *computing* or *uptodate*. During a single recalculation, state changes only monotonically—e.g. a cell cannot go back from *uptodate* to *computing*, according to the state transitions in Fig. 3. During recalculation, the cell state indicates whether a cell should be evaluated anew or not. Moreover, cells cache their values. A cell that is *uptodate* has cached its most recent value.

Minimal recalculation is a breadth-first traversal of the support graph. Funcalc uses a global work queue to maintain all cells that have been encountered during traversal and that have not yet been computed. The minimal recalculation algorithm MinimalRecalc (Algorithm 1) (1) marks the transitive closure of the recalculation roots dirty by calling MarkDirty; (2) adds the recalculation roots to the global work queue; and (3) dequeues cells from the head of the queue and evaluates them by calling Eval until the queue is empty. Table 1 lists a subset of the functions used in pseudo-code.

During its evaluation, a cell's state is *computing*. After evaluation, the algorithm enqueues all the cell's directly supported cells, updates the cell's cache and sets its state to *uptodate*. It enqueues supported cells via EnqueueSupported if their state is *dirty* and changes their state to *enqueued*.

Table 1 Overview of the functions used in pseudo-code

Name	Signature	Description
Cache	*cell* → *value*	Get the most recent, cached value of a cell
Cycle?	() → *bool*	True if a cycle is present, false otherwise
EvalExpr	*cell* → *value*	Evaluate the cell's formula to a value
NotifyCycle!	() → ()	After the call, calling Cycle? returns true
State	*cell* → *state*	Get or set a cell's state
Supported	*cell* → [*cell*]	Get a list of the cell's supported cells

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 all u in Supported(cell) do
 5:       MarkDirty(u)

 6: function Eval(cell)
 7:   switch State(cell)
 8:     case computing:
 9:       NotifyCycle!()
10:     case enqueued or dirty:
11:       State(cell) ← computing
12:       Cache(cell) ← EvalExpr(cell)
13:       State(cell) ← uptodate
14:       EnqueueSupported(cell)
15:   return Cache(cell)

16: function EnqueueSupported(cell)
17:   for all u in Supported(cell) do
18:     if State(u) = dirty then
19:       State(u) ← enqueued
20:       Enqueue(Q, u)

21: function MinimalRecalc(roots)
22:   for all r in roots do
23:     MarkDirty(r)
24:     Enqueue(Q, r)
25:     State(r) ← enqueued
26:   while ¬(Empty?(Q) ∨ Cycle?) do
27:     cell ← Dequeue(Q)
28:     Eval(cell)
```

If a cell is being evaluated and one of its dependencies d is not yet *uptodate*, EvalExpr will recursively evaluate the dependency by calling Eval(d). If the dependency's state is *computing*, recalculation has detected a cyclic reference (line 9 in Algorithm 1, function Eval).

4 Puncalc: parallel implementation

Puncalc, short for *parallel Funcalc*, is a parallel variant of Funcalc based on the . NET Task Parallel Library (TPL) [16]. We use TPL tasks and its work-stealing queue implementation which we do not describe in further detail. In the following sections, we introduce thread-safety requirements on cell state; our approach to parallel, minimal recalculation and cycle detection; and how it complies with the consistency requirement from Sect. 2.4. We then extend the algorithm with a thread-local optimization technique that exploits a specific spreadsheet topology on the fly.

4.1 Thread safety

Funcalc is conceptually a strict, purely functional language. However, the implementation uses mutable state to make the language efficient, which we must make thread safe: the global recalculation queue must be thread safe; cells *cache* the result of their evaluation, and all threads should agree on the cached result due to the consistency requirement from Sect. 2.4; cells also have a *state* that should be consistent among all threads; and each cell should only be evaluated once. We relax the latter requirement in Sect. 4.3 in order to detect cycles in parallel.

We handle the cells' underlying mutable state using the following scheme. If multiple threads try to evaluate the same cell, one thread takes *ownership* of the cell and sets the cell's state to *computing*. We call this thread the cell's owner. The thread will then evaluate the cell's formula, write the result to the cell's value cache and finally set the cell's state to *uptodate*. The remaining threads block until the cell's state is set to *uptodate* and then read the value from the cell's value cache.

We can use intrinsic locks on cells to implement this scheme but as it turns out, locking on cells comes at the cost of performance and correctness. Threads that wait for locks are *de-scheduled*, but often times, the cell's value will be available soon, since the average computation time per cell is usually rather low (see Sect. 5). This makes de- and rescheduling a waste of time. In terms of correctness, it is legal to create cyclic references in spreadsheets (see Sect. 2.2), but such cyclic references can lead to deadlocks. Suppose thread t_1 locks on cell c. When another thread t_2 examines c, it sees that c is locked by thread t_1 and blocks. If threads t_1 and t_2 both evaluate cells that are part of a cycle, they will deadlock as no thread is able to make progress.

Instead of using locking, we implement our scheme using *compare and swap* (Cas) [13, sec. 5.8] on a cell's state and value cache. This allows us, among other things, to detect cyclic dependencies dynamically as described in Sect. 4.3. We write F_{TS} to denote that function F is now thread safe. Furthermore, we switch to a thread-safe, scalable and concurrent queue *CQ* (we use .NET's thread-safe, concurrent queue). Note that all reads from and writes to $\textsc{State}_{\mathrm{TS}}$ are lock-free; reads from $\textsc{Cache}_{\mathrm{TS}}$ are lock-free if the cell is *uptodate*.

The overall idea is to let threads compete for setting a cell's state to *computing* using Cas, as detailed in Algorithm 2. The thread that wins the race proceeds as described above, while the other threads enter a busy-wait loop. Table 2 shows definitions for functions that we use in addition to those presented earlier in Table 1.

Table 2 Overview of the functions particular to parallel recalculation

Name	Signature	Description
Cas	$r \times c \times v \to bool$	Atomically set r to v if its value is c
Dec	$counter \to ()$	Atomically decrease the counter by one
EncodeOwner	$id \times state \to int$	Encode an identifier into a cell state's upper bits
Get	$counter \to int$	Atomically get the counter's current value
Id	$thread \to id$	Get the identifier for a thread
Inc	$counter \to ()$	Atomically increase the counter by one
OwnerBits	$int \to id$	Read only ownership bits from encoded state
Spawn	$fun \to ()$	Spawn a task to execute a function in parallel
StateBits	$int \to state$	Read only state bits from encoded state

Algorithm 2 Thread-safe evaluation function

```
1:  function EvalPar(cell)
2:    s ← State_TS(cell)
3:    switch s
4:      case computing:
5:        while State_TS(cell) ≠ uptodate do nothing
6:      case dirty or enqueued:
7:        if Cas(state_TS(cell), s, computing) then
8:          Cache_TS(cell) ← EvalExpr(cell)
9:          State_TS(cell) ← uptodate
10:         EnqueueSupported_TS(cell)
11:   return Cache_TS(cell)
```

Only the thread whose Cas succeeded is allowed to enqueue (line 10, Algorithm 2). The cells in the support set may however also be part of some other cell's support set, so there still is a possibility for races. Note that $\text{EnqueueSupported}_{TS}$ makes sure that each cell gets enqueued at most once (see Algorithm 1).

4.2 Parallel minimal recalculation

We must address two main problems when implementing a parallel recalculation algorithm: first, choosing adequate termination criteria for recalculation, and second, detecting cycles correctly, which we discuss in Sect. 4.3.

Algorithm 3 shows the main loop of *parallel* recalculation that handles termination. The emptiness of the global work queue alone is no longer a sufficient termination criterion. The queue may be empty, while there are still cells being evaluated by other threads, which may in turn enqueue more cells. Therefore, we use a concurrent and scalable atomic *counter* class, inspired by the Java 8 `LongAdder` implementation [19] to keep track of the number of cells currently being evaluated.

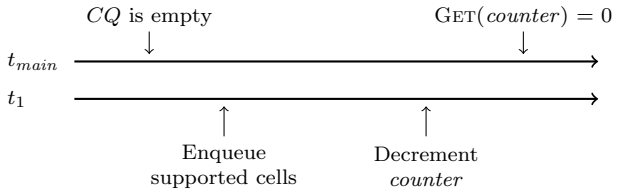

Fig. 4 A timeline (from left to right), showing the interleaving of actions that could cause premature termination of the algorithm if the termination condition was to be reversed. Thread t_{main} is the main thread, and thread t_1 is a worker thread

Algorithm 3 Main algorithm for parallel minimal recalculation

```
1:  function MinimalRecalcPar(roots)
2:    counter ← AtomicCounter(0)
3:    for all r in roots do
4:      MarkDirty_TS(r)
5:      Enqueue_TS(CQ, r)
6:      State_TS(r) ← enqueued
7:    while (Get(counter) > 0 ∨ ¬Empty?(CQ)) ∧ ¬Cycle?() do
8:      cell ← Dequeue_TS(CQ)
9:      if cell ≠ null then
10:       Inc(counter)
11:       Spawn(fun() ⇒ {
12:         EvalPar(cell)
13:         Dec(counter)
14:       })
```

Parallel minimal recalculation begins similarly to its sequential counterpart by marking all cells in the transitive closure of the recalculation roots *dirty*, enqueuing the roots and changing their state to *enqueued*.

If the main thread successfully dequeues a cell from the queue, it increments *counter* (line 10, Algorithm 3) and spawns a new task to compute it. The task is sent to a thread pool where it evaluates the cell and subsequently decrements *counter*. In Sect. 4.1, we made sure that only one task gets to set the computed value of the cell.

The termination condition of the while loop in line 7, Algorithm 3 states that it should keep running as long as (1) there is at least one cell being evaluated or (2) the queue is not empty, and (3) no cycles have been detected.

It is crucial that the checks for termination (1) and (2) are ordered as they are. Imagine we were to swap (1) and (2) as in the timeline in Fig. 4, and initially, the queue was empty and GET(counter) = 1. The main thread t_{main} would evaluate the while loop condition and see that *CQ* would be empty. Before t_{main} would continue, t_1 would finish evaluating a cell and enqueue a non-empty set of supported cells such that *CQ* would be non-empty. Thread t_1 would then decrement the counter so that GET(counter) = 0. Now t_{main} would incorrectly believe that there were no cells currently being evaluated and would exit the loop prematurely. This subtle race does not occur when we order the checks as in Algorithm 3.

4.3 Cyclic dependency detection

To detect a cyclic dependency during sequential recalculation, it is sufficient to inspect a cell's state before evaluating it and to check whether its state is *computing*. Detecting cycles in parallel is less straightforward. If any thread sees a cell that is *computing*, it has not necessarily found a cyclic dependency as *another* thread may currently be computing the cell. In this section, we discuss the challenges of parallel cycle detection and then discuss our solution.

We could circumvent the problem by sequentially checking for cycles before initiating a parallel recalculation, but this would defeat the purpose of recalculating in parallel in the first place. A sequential static cycle check would be too conservative and lead to false positives (see Fig. 2).

As mentioned in Sect. 4.1, simply locking on a cell, either while evaluating it or while waiting for another thread to evaluate it, is not a feasible solution either. Alternatively, a thread that discovers a *computing* cell could immediately report a cyclic dependency, but that would be overly pessimistic.

What we need is a tie-breaker that allows at least one thread to proceed so that it can discover the cycle. During parallel recalculation, a cyclic dependency occurs only if a thread t_i encounters a cell that is *computing* and whose *owner* is t_i itself. If a cell is *computing* but owned by another thread, t_i waits until the cell becomes *uptodate* and then reads the cell's cached value.

How do we decide which thread is allowed to proceed? We use thread IDs, which impose an arbitrary, numerical order on threads, to determine *thread precedence*. A thread t_i has precedence over t_j if $\textsc{Id}(t_i) < \textsc{Id}(t_j)$. If t_i and t_j wait for a cell that the respective other thread owns due to a cyclic reference, then, at some point, t_i may proceed and discover the cycle.

4.4 Encoding ownership in cell state

We want to manipulate state and ownership of a cell using a single atomic operation to avoid adding logic for handling partial states. Internally, cell state is represented by some bits of an integer. There are four cell states, so it suffices to use two bits to encode these.

We can encode the ID of the current thread in the remaining, unused bits along with the *computing* cell state to claim ownership of the cell, allowing us to manipulate both using a single Cas operation. Function OwnerBits(s) only returns the ownership bits of s, and StateBits(s) returns the state bits. For all other cell states, the ownership bits are all zero.

4.5 Parallel recalculation with speculative reevaluation

This section details the implementation of a dynamic resolution of cyclic dependencies that we call *speculative reevaluation*. We only want to report cycles that actually exist and occur *dynamically* during recalculation.

Algorithm 4 shows the pseudo-code for the EvalParSpec function that we now invoke instead of EvalPar in line 12, Algorithm 3. Functions EvalParSpec and TryEvalExpr directly encode the scheme described in Sect. 4.3. If the cell's state is *computing*, we check whether the current thread t_{cur} is the owner of the cell. If yes, t_{cur} has detected a cyclic dependency and we must abort recalculation. To allow any waiting threads to finish up, t_{cur} sets the cell's state to *uptodate* and notifies the main thread to stop spawning new tasks via NotifyCycle!$_{\text{TS}}$ which then exits the main loop. Other threads will simply terminate when they are done evaluating their current cell.

Algorithm 4 Parallel speculative recalculation with dynamic cycle detection

```
1:  function TryEvalExpr(cell, v_old, s)
2:    s' ← EncodeOwner(Id(t_cur), computing)
3:    if Cas(State_TS(cell), s, s') then
4:      v ← EvalExpr(cell)
5:      if Cas(Cache(cell), v_old, v) then
6:        State_TS(cell) ← uptodate
7:        EnqueueSupported_TS(cell)
8:  function EvalParSpec(cell)
9:    s ← State_TS(cell)
10:   v_old ← Cache_TS(cell)
11:   switch s
12:     case computing:
13:       if Id(t_cur) = OwnerBits(s) then
14:         State_TS(cell) ← uptodate
15:         NotifyCycle!_TS()
16:       else if Id(t_cur) < OwnerBits(s) then
17:         while Id(t_cur) < OwnerBits(s) ∧ StateBits(s) = computing do
18:           TryEvalExpr(cell, v_old, s)
19:           s ← State_TS(cell)
20:       else
21:         while State_TS(cell) ≠ uptodate do nothing
22:     case dirty or enqueued:
23:       TryEvalExpr(cell, v_old, s)
24:   return Cache_TS(cell)
```

If t_{cur} is not the owner, it checks whether it has precedence over the current owner and, if so, attempts to evaluate the cell by calling TryEvalExpr. If t_{cur} is neither the owner of the cell nor has precedence over the current owner, it spins until it can retrieve the cell's cached value. If the cell is either *dirty* or *enqueued*, the thread attempts to evaluate it directly, also using TryEvalExpr. If the cell is *uptodate*, the function just returns the cell's cached result.

Whenever a thread attempts speculative reevaluation, it claims ownership of the cell. This reduces the number of redundant speculative evaluations and

	A	B
1	=B1	=NOW()
2	=B1	

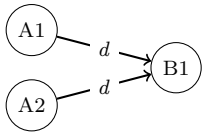

Fig. 5 Two cells depend on the same cell containing a call to NOW() If both cell A1 and A2 are evaluated in parallel and both recursively attempt to evaluate B1, their respective threads must agree upon which value B1 has evaluated to

is important for cycle detection. If a thread t_i has precedence over thread t_j and claims ownership of cell c, and another thread t_k has precedence over t_j but not t_i, such that $\text{ID}(t_i) < \text{ID}(t_k) < \text{ID}(t_j)$, t_k is not allowed to speculatively evaluate c. If t_k happens to claim ownership of c before t_i, then t_i has to try and reclaim ownership from t_k again to detect cycles correctly.

To see the need for this, imagine cell c had a cyclic dependency on cell x owned by t_i, but t_k successfully took ownership of c, while t_i failed to take ownership and would spin. As soon as t_k arrived at cell x, it would detect that it does not have precedence over t_i and recalculation would become stuck. If thread t with $\text{ID}(t) = n$ does not return from evaluating a cell due to a cyclic reference, then at worst only $n - 1$ threads with lower IDs can evaluate the same cell speculatively before one of them detects the cycle, so every cycle will eventually be discovered.

4.5.1 Ensuring consistency

It is possible that two or more threads attempt to evaluate the same cell, as illustrated in Fig. 5. In Sect. 4.1, we discussed that all threads should agree on the cached value of each cell, so we must ensure that only one of the evaluating threads gets to set the cached value; the other threads must discard the result of their own evaluation and continue using the now updated cached value. Function TryEvalExpr ensures that only one thread gets to update the cell's cache by using Cas.

Our algorithm for parallel minimal recalculation retains the consistency requirement stated in Sect. 2.4 up to cyclic dependencies, similar to sequential Funcalc. Using Cas in line 5 in Algorithm 4 makes sure that all threads will agree on the value of each cell in σ (see Sect. 2.4). If the spreadsheet contains a cyclic dependency, we cannot guarantee consistency. In this case, we let all threads continue using possibly stale values and notify the main thread that a cyclic reference has been found. While this approach may seem simplistic, it elegantly terminates the recalculation process.

4.5.2 Delayed speculative evaluation

In practice, we do not want to allow a thread with precedence to immediately speculatively evaluate a cell as in line 18, Algorithm 4. Instead, the thread first spins for a short amount of time while continuously checking the cell state. If the cell becomes *uptodate* during spinning, the spinning thread does not attempt to evaluate the cell

	A	B	C
1	=1+2	=SIN(B2)	
2	=A1+B1	=INDIRECT("A"&C2)	2

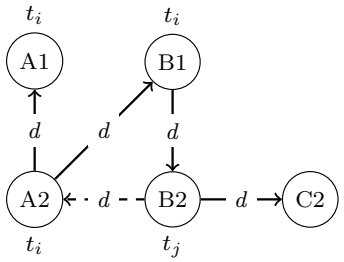

Fig. 6 Example of how a cycle introduced by the use of `INDIRECT` is handled by the parallel algorithm. We have purposefully used a dashed line for the dependency of cell B2 on cell A2 because the dependency is not explicitly given in the formula's expression. Cell thread ownership is shown at each cell

speculatively; otherwise, it proceeds with the evaluation of the cell. This heuristic makes sure that we do not needlessly start evaluating when the result will be available early.

4.6 Thread-local evaluation

If a cell only has a single outgoing support edge, i.e. only a single cell in the spreadsheet refers to it, Algorithm 3 will still spawn a new task for the single supported cell, even though there is no parallelism that we can exploit. Instead the current thread could evaluate the cell locally, circumventing the global queue and avoid spawning a new task.

We can implement an optimization for such sequential chains by detecting when a cell supports only a single cell. If so, evaluate the supported cell locally on the current thread which continues to evaluate cells locally, until it reaches a cell that supports either zero or more than one cell, or is already *uptodate*.

We must use TryEvalExpr for thread-local evaluation as well, since the cells in the sequential chain may still have multiple dependencies, and another thread may still attempt to evaluate the same cell simultaneously.

4.7 Dynamic indexing functions

As mentioned in Sect. 2.2, some functions can dynamically refer to other cells by interpreting strings as cell references. In this section, we show how Puncalc is able to handle `INDIRECT` and similar dynamic indexing functions.

When a thread evaluates a cell containing a call to `INDIRECT`, it evaluates the formula's expression and its dependencies as usual. How should one model the support edges of such a cell? Since `INDIRECT` can refer to *any* cell in the spreadsheet, we cannot statically create support edges for an `INDIRECT` cell, which means that such cells may never get enqueued in the evaluation queue. This may result in an inconsistent spreadsheet after recalculation. Alternatively, we could modify the dependency graph during evaluation but this complicates the algorithm and puts further thread-safety requirements on Puncalc. To avoid any inconsistencies or further static analysis, we choose to make `INDIRECT` volatile [21, Section 5.5] so that it is

Table 3 Spreadsheet statistics and benchmark results

Sheet	Cells	Roots	Support	Span	Seq.(s)	×16	×16*	×48	×48*
Building-design	108,332	18,378	488,351,887	4	32.12	5.57	5.61	12.90	12.64
Energy-markets	534,507	35,198	287,818,610	3	168.16	2.17	2.17	1.53	1.54
Grossprofit	135,073	15,301	112,612,549	3	102.19	4.41	4.40	2.54	2.55
Ground-water	126,404	31,601	1,099,366,302	1	81.26	5.47	5.56	15.59	15.94
Stock-history	226,503	23,402	317,049	3	64.90	6.51	6.48	12.53	12.22
Stocks-price	812,693	10,876	233,376,389	3	102.74	2.57	2.58	0.84	0.62
Binary-join	262,146	1	393,215	18	138.63	4.12	2.75	2.34	1.19
Binary-tree	266,145	1	262,143	17	141.14	4.20	4.30	2.31	2.32
Fork	300,001	1	300,301	1001	160.14	4.45	4.04	2.42	2.34
Fork-join	300,002	1	300,600	1001	158.92	4.28	3.34	2.39	1.95
Map	300,001	1	300,001	1	160.82	3.77	3.74	2.24	2.24
Prefix	300,000	1	745,009	1100	161.32	1.37	1.02	0.56	0.35

Columns labelled ×*n* show relative speedup for *n* cores. Columns labelled ×*n** show speedup for *n* cores with thread-local evaluation enabled, as described in Sect. 4.6

reevaluated on each recalculation. While perhaps not fully optimal, this is a simple and elegant solution and since threads evaluate their dependencies as usual, the parallel algorithm will also work in the presence of such functions. To convince readers of this, we provide a small example in Fig. 6 where cell B2 is implicitly volatile.

Suppose that thread t_i initially claims ownership of A1. After it is done evaluating the cell, it evaluates cells A2 and B1. Thread t_i does not follow a support edge to B2 since there is currently no such edge. Since cell B2 calls `INDIRECT`, it is volatile and thread t_j claims ownership of it. Thread t_j evaluates cell B2's dependency on C2, evaluates the call to `INDIRECT` and subsequently attempts to evaluate cell A2 which is owned by thread t_i. The situation at this point in time is depicted in Fig. 6.

One of two things can happen now, depending on the thread IDs of t_i and t_j. If $\text{ID}(t_i) < \text{ID}(t_j)$, t_i will have precedence over t_j and claim cell B2. It will evaluate the cell, evaluate its dependency on A2 and discover it is already the owner of A2. On the other hand, if $\text{ID}(t_i) > \text{ID}(t_j)$, t_j will have precedence over t_i and claim cell A2. It will evaluate its dependencies through B1 to B2 and discover itself. Thus, the implicit cycle introduced by `INDIRECT` is detected in both cases.

5 Results and validation

5.1 Benchmark spreadsheets

We use the following spreadsheet suites to benchmark Puncalc:

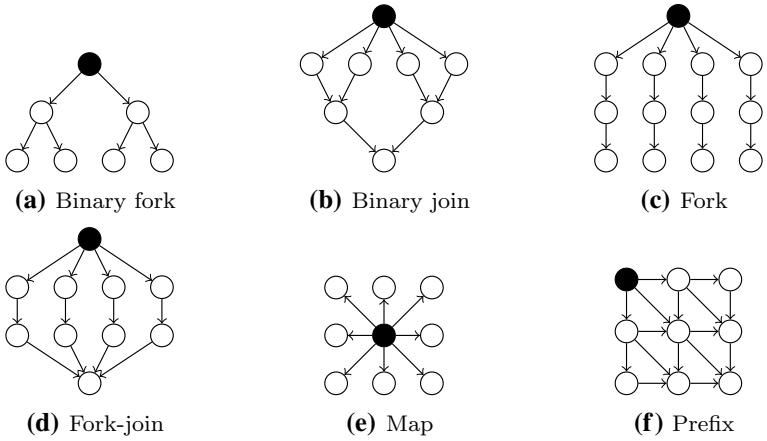

Fig. 7 Illustrations of the underlying support graph structures of the synthetic spreadsheets for benchmarking. Black nodes mark recalculation roots. We only use one recalculation root per sheet to simulate editing a single cell

Real-World Spreadsheets LibreOffice Calc [24] provides a set of large benchmark spreadsheets.[1] Benchmarking on large supposedly "real-world" spreadsheets is meant to give us insight into how well Puncalc copes with realistically structured spreadsheets. To be able to run the spreadsheets in Puncalc, we have removed all convenience macros and implemented unsupported functions as SDFs. Furthermore, we detect all formula cells that have no formula dependencies and use them as recalculation roots (the "Roots" column in Table 3) to simulate minimal recalculation. As a result, they are initially enqueued in the global work queue, and the main thread can then dequeue cells from the queue with little interference from enqueueing threads. However, this may have a positive effect on performance and is unrealistic since users usually only edit one cell at a time.

Artificial spreadsheets To explore Puncalc's behaviour in a controlled and systematic fashion, we use six programmatically generated spreadsheet topologies, as shown in Fig. 7. Each cell calls a recursive SDF implementation of the Fibonacci function `FIB` without tail-call optimization which allows us to control the amount of work per cell. We pass a parameter to `FIB` that corresponds to roughly 0.7 ms evaluation time per call, which is the maximum, average work per cell from all LibreOffice spreadsheets.

5.2 Experimental setup

Our test machine is an Intel Xeon E5-2680 v3 with 48 logical 2.5 GHz cores and 32 GB of memory, running 64-bit Windows 10, version 1607, and .NET 4.7.1. We

[1] Available at https://gerrit.libreoffice.org/gitweb?p=benchmark.git.

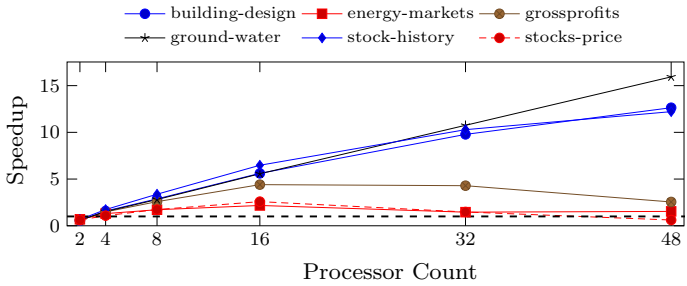

Fig. 8 Average benchmark results over 50 runs per spreadsheet from the LibreOffice Calc spreadsheet suite *with* thread-local evaluation enabled. Values are speedup factors over sequential performance on the same machine; higher is better. The grey dashed line indicates 1-core performance. The standard deviation is ≤ 0.21 for all benchmarks

initially performed three warm-up runs and ran each benchmark for five iterations.[2] For each iteration, we ran the benchmark ten times and computed the average execution time. We report the average of those five averages and their standard deviation in Table 3. Sequential (1-core) running times are measured without volatile reads and writes, or any other thread-safe primitives or data structures to ensure a fair comparison.

We limit the number of TPL threads to match the number of available, logical cores for each run. Additionally, we disable TPL's heuristics for thread creation and destruction so that all threads are created at start-up. We have chosen a spin time of 1ms as this reflects the maximum, average evaluation time for formulas in the LibreOffice benchmark suite.

5.3 Validation

We have validated that our algorithm for *parallel minimal recalculation* in Puncalc produces the same result as sequential minimal recalculation in Funcalc for all sheets from the real-world benchmark suite. Hence, we believe that parallel recalculation respects the consistency requirements (Sect. 2.4).

5.4 Performance evaluation

There are three main observations to be made from the performance benchmarks:

Observation 1 Figure 8 shows that our approach scales for the majority of tested spreadsheets up to 16 cores, where we gain most speedup on average.

[2] Raw data available at https://github.com/popular-parallel-programming/puncalc-benchmarks/tree/xeon.

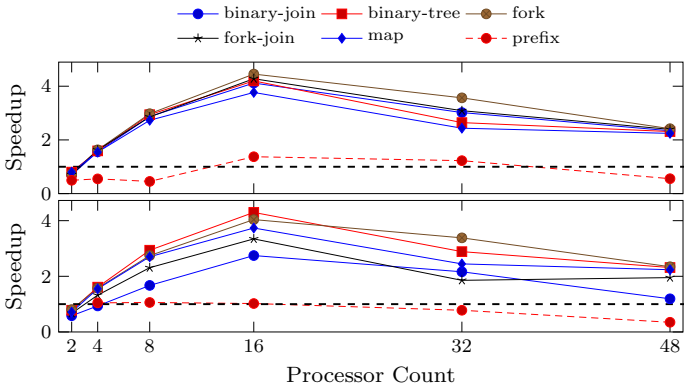

Fig. 9 Average benchmark results over 50 runs per spreadsheet from the synthetic spreadsheet suite. Top: *without* thread-local evaluation. Bottom: *with* thread-local evaluation. Values are speedup factors over sequential performance on the same machine; higher is better. The grey dashed line indicates 1-core performance. The standard deviation is ≤ 0.1 for all benchmarks

The relative speedup decreases for all spreadsheets for more than 16 cores, except for *building-design*, *ground-water* and *stock-history* from the LibreOffice benchmarks. It is unclear what causes these three spreadsheets to continue to improve, but there are likely multiple factors in play.

The performance decline after 16 cores may simply be caused by increased contention and more speculative evaluations. Another explanation relates to our Intel Xeon, which consists of two chips with twelve cores each. Up to 16 "logical" cores (i.e. including hyper-threading), communication does not happen across chips. Therefore, we do not have to pay an excessive synchronization cost when threads wait for *computing* cells whose owners are scheduled off-chip. The structure of the three aforementioned sheets might correct for such expensive communication.

Observation 2 Thread-local evaluation does not improve performance compared to eagerly spawning a task for each cell as shown in Fig. 9 and often leads to worse performance than eagerly spawning tasks.

This may be due to two factors. First, thread-local evaluation is a *depth-first* traversal, while eagerly spawning tasks are akin to *breadth-first* traversal. Therefore, thread-local evaluation makes recursive evaluation of dependencies more likely, which is slower than using the global work queue. For heavily sequential spreadsheets such as prefix (Fig. 7f), thread-local evaluation can alleviate the overhead of parallelization, which may be favourable for a robust implementation. However, recursive evaluation can lead to stack overflow errors. Second, the TPL [16] uses work-stealing: idle threads steal work in the form of tasks from other threads. If we spawn less tasks and hence have more idle threads, they will attempt to steal work more often. Frequent work-stealing is more costly if it happens across chips.

Observation 3 Neither the number of cells, roots, support edges or span (i.e. the longest sequential path) of a spreadsheet are good indicators for parallel performance.

There is no apparent correlation between these statistics and the performance results in Table 3. This is much to our surprise, and a deeper structural analysis may be required in order to discover the causes behind the observed results.

6 Related work

Little research deals with parallel recalculation of spreadsheets. The general focus has instead been on detection and handling of errors [6].

There exist multiple distributed systems for spreadsheet computations, such as ActiveSheets [4], Nimrod [3] and HPC Services for Excel [18]. All three systems require reengineering of the spreadsheet, which may take a substantial amount of time and require expert engineers [23]. In contrast, Puncalc runs on a shared-memory multiprocessor and automatically exploits the machine's available processors without needing to change the spreadsheet itself.

Wack [26] bridges the gap between distributed systems and automatic parallelization. His dissertation describes an improved spreadsheet model that statically partitions and schedules a set of predefined patterns and parallelizes them via message-passing in a network of work stations. Apart from using a different machine model, his work simply disallows cyclic dependencies [26, sec. 2.8.3], which corresponds to static cycle detection.

Biermann et al. [5] parallelize spreadsheets by statically rewriting so-called cell arrays to calls to higher-order functions on arrays, exploiting their inherent parallelism. Their approach does not parallelize the evaluation of disjoint cell arrays and requires certain predefined structures to be present.

Both works require static analysis of the spreadsheet prior to recalculation, whereas Puncalc detects both parallelism and cyclic dependencies dynamically.

6.1 Commercial and open-source applications

Excel is probably the most well-known commercial spreadsheet application. Being closed source, little information is available about its recalculation engine although it has an option that allows users to enable multi-threaded recalculation. Sestoft [21] gives some additional information based on speculation and experimentation.

SpreadsheetGear [22] is a collection of commercial plug-ins for Excel, one of which is a calculation engine that allows for multi-threaded recalculation through .NET's TPL. Further details are not available.

In collaboration with the LibreOffice open-source project, AMD has implemented GPU parallelization for LibreOffice Calc by automatically compiling formulas involving cell ranges, such as `=SUM(A1:A100)`, into OpenCL kernels [25].

They report between 30 and 500 times speedups [17], but do not take additional improvements to their internal data representation into account.

None of the applications above report results for systematic performance benchmarks or give a detailed description of the underlying algorithms.

7 Conclusion

In this paper, we have presented Puncalc, a spreadsheet engine that targets shared-memory multiprocessors and automatically extracts parallelism from spreadsheet computations, obtaining overall satisfactory speedups without adding any engineering overhead. To our knowledge, this is the first algorithm for parallel spreadsheet recalculation with dynamic cycle detection that has been described in the literature.

We have given a number of possible explanations for the performance results in Sect. 5, but further investigation is needed. Furthermore, we are lacking a direct comparison of the performance of Puncalc to that of other frameworks for spreadsheet parallelization, such as those mentioned in Sect. 6.

We believe that our work, combined with the work on sheet-defined functions [20, 21], is a first step towards a powerful framework for end-user development and hope to pave the way for a paradigm shift where spreadsheets are viewed as a serious computational tool for a broad range of problems by both researchers and IT professionals.

Acknowledgements Thanks to Peter Sestoft for useful technical discussions and to Claus Brabrand, Peter Sestoft and Kenneth Ry Ulrik for useful comments on an earlier draft.

Compliance with ethical standards

Conflict of interest Author Alexander Asp Bock has been employed for three months as a research intern at Microsoft Research Cambridge (MSRC) in 2017. The work conducted there is protected under a non-disclosure agreement.

References

1. Abraham R, Erwig M (2006) Type inference for spreadsheets. In: Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming. ACM, pp 73–84. https://doi.org/10.1145/1140335.1140346
2. Abraham R, Erwig M (2007) UCheck: a spreadsheet type checker for end users. J Vis Lang Comput 18(1):71–95. https://doi.org/10.1016/j.jvlc.2006.06.001
3. Abramson D, Sosic R, Giddy J, Hall B (1995) Nimrod: a tool for performing parametrised simulations using distributed workstations. In: Proceedings of the Fourth IEEE International

Symposium on High Performance Distributed Computing, pp 112–121. https://doi.org/10.1109/HPDC.1995.518701
4. Abramson D, Roe P, Kotler L, Mather D (2001) Activesheets: super-computing with spreadsheets. In: 2001 High Performance Computing Symposium (HPC '01), Advanced Simulation Technologies Conference, Citeseer, pp 22–26
5. Biermann F, Dou W, Sestoft P (2018) Rewriting high-level spreadsheet structures into higher-order functional programs. In: Calimeri F, Hamlen K, Leone N (eds) Practical aspects of declarative languages, vol 10702. Lecture notes in computer science. Springer, Berlin, pp 20–35. https://doi.org/10.1007/978-3-319-73305-0_2
6. Bock AA (2016) A literature review of spreadsheet technology. Technical report 199, IT University of Copenhagen. http://forskningsdatabasen.dk/en/catalog/2350168960
7. Burnett M (2009) What is end-user software engineering and why does it matter?. Springer, Berlin, Heidelberg, pp 15–28. https://doi.org/10.1007/978-3-642-00427-8_2
8. Burnett M, Cook C, Rothermel G (2004) End-user software engineering. Commun ACM 47(9):53–58. https://doi.org/10.1145/1015864.1015889
9. Casimir RJ (1992) Real programmers don't use spreadsheets. SIGPLAN Not 27(6):10–16. https://doi.org/10.1145/130981.130982
10. Erwig M, Burnett M (2002) Adding apples and oranges. In: Practical Aspects of Declarative Languages. Springer, pp 173–191
11. EuSpRiG. EuSpRiG horror stories. http://eusprig.org/horror-stories.htm. Accessed 14 June 2016
12. Harvey B, Wright M (1999) Simply scheme: introducing computer science. MIT Press, Cambridge
13. Herlihy M, Shavit N (2008) The art of multiprocessor programming. Elsevier/Morgan Kaufmann. http://www.worldcat.org/isbn/9780123705914
14. Hermans F, Pinzger M, van Deursen A (2011) Supporting professional spreadsheet users by generating leveled dataflow diagrams. In: Proceedings of the 33rd International Conference on Software Engineering. ACM, ICSE '11, pp 451–460. https://doi.org/10.1145/1985793.1985855
15. Johnston WM, Hanna JRP, Millar RJ (2004) Advances in dataflow programming languages. ACM Comput Surv 36(1):1–34. https://doi.org/10.1145/1013208.1013209
16. Leijen D, Schulte W, Burckhardt S (2009) The design of a task parallel library. SIGPLAN Not 44(10):227–242. https://doi.org/10.1145/1639949.1640106
17. Meeks M (2014) LibreOffice calc: spreadsheets on the GPU, iWOCL. http://www.iwocl.org/iwocl-2014/abstracts/libreoffice-spreadsheets-on-the-gpu/. Accessed 13 Mar 2018
18. Microsoft (2015) HPC services for excel. https://technet.microsoft.com/en-us/library/ff877820(v=ws.10).aspx. Accessed 30 June 2016
19. Oracle (2014) Class LongAdder. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAdder.html. Accessed 21 Feb 2017
20. Peyton-Jones S, Blackwell A, Burnett M (2003) A user-centred approach to functions in excel. In: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming. ACM, ICFP '03, pp 165–176. https://doi.org/10.1145/944705.944721
21. Sestoft P (2014) Spreadsheet implementation technology. The MIT Press, Cambridge
22. SpreadsheetGear LLC (2012) Easily take advantage of multi-core CPUs. https://www.spreadsheetgear.com/support/help/spreadsheetgear.net.7.0/Key_Concepts_Easily_Take_Advantage_of_Multi-Core_CPUs.html. Accessed 18 Jan 2018
23. Swidan A, Hermans F, Koesoemowidjojo R (2016) Improving the performance of a large scale spreadsheet: a case study. In: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). IEEE, pp 673–677. https://doi.org/10.1109/saner.2016.100, http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2016-003.pdf
24. The Document Foundation. LibreOffice Calc. https://www.libreoffice.org/discover/calc/. Accessed 09 May 2016
25. Trudeau J (2015) Collaboration and open source at AMD: LibreOffice. https://developer.amd.com/collaboration-and-open-source-at-amd-libreoffice/. Accessed 2015 July 31
26. Wack AP (1996) Partitioning dependency graphs for concurrent execution: a parallel spreadsheet on a realistically modeled message passing environment. PhD thesis, University of Delaware, Newark. http://portal.acm.org/citation.cfm?id=269551
27. Yoder AG, Cohn DL (1994) Real spreadsheets for real programmers. In: Proceedings of the 1994 International Conference on Computer Languages, 1994, pp 20–30. https://doi.org/10.1109/ICCL.1994.288396