
Puncalc: Task-Based Parallelism and Speculative Reevaluation in Spreadsheets

Florian Biermann · Alexander Asp Bock

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, that 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, 13].

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 large, complex spreadsheets over several years [15] and their complexity often leads to errors [11] and poor performance. For

Both authors have contributed equally to this article.

Florian Biermann: Supported by the Sino-Danish Center for Education and Research.

Florian Biermann and Alexander Asp Bock
IT University of Copenhagen
Rued Langgaards Vej 7
2300 Copenhagen S, Denmark
E-mail: {fbie, albo}@itu.dk

instance, Swidan et al [24] report on a case study of refactoring a spreadsheet that would originally take 10 hours 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 re-engineer spreadsheets [24].

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 [23] which introduces efficient, sheet-defined, higher-order functions to the spreadsheet paradigm. The combination of sheet-defined functions and our parallel recalculation algorithm can contribute to change the general perception of spreadsheets as not being “real” programming languages [9, 10, 22, 27] and enable end-user programmers to use spreadsheets for heavyweight computations with a more reusable, modular, performant, 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.

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 [23, Chapter 1].

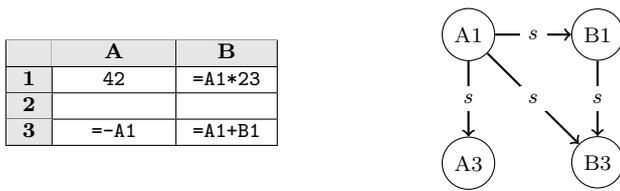


Fig. 1: A spreadsheet containing formulas (left) and its corresponding support graph (right). We label supporting edges with s . Editing A1 will recalculate cells A1, B1, A3 and B3; changing B1 will recalculate B1 and B3; and editing A3 will only recalculate A3.

2.1 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 [16], 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 sec. 2.3) depending on whether RAND() evaluates to a value less than 0.5 or not. Cyclic references are usually disallowed, because recalculation cannot proceed meaningfully. When a cyclic reference is found, the user is commonly alerted through the GUI and recalculation aborts.

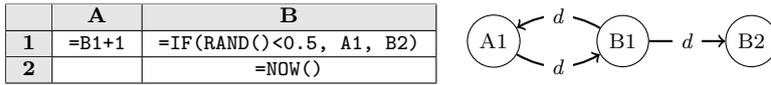


Fig. 2: A static cyclic reference in a spreadsheet (left) and its corresponding cyclic dependency graph (right). We label dependency edges with d .

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. We call the cells that start a minimal recalculation the *recalculation roots*. In fig. 1, whenever 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.

2.4 Consistency Requirements

The purpose of recalculation is to bring the spreadsheet to a *consistent state* [23, 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:

$$\text{dom}(\sigma) = \text{dom}(\phi) \quad (1)$$

$$\forall c \in \text{dom}(\phi) . \sigma \vdash \phi(c) \Downarrow \sigma(c) \quad (2)$$

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 [23] implemented in C# that adds sheet-defined, higher-order functions to the spreadsheet paradigm. In

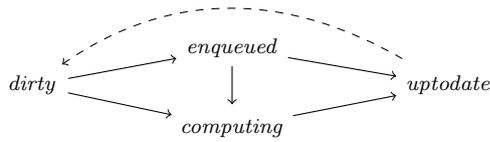


Fig. 3: The possible state transitions of a cell. The dashed connection denotes marking cells dirty in the transitive closure of a recalculation root at the beginning of a particular recalculation.

Name	Signature	Description
CACHE	$cell \rightarrow value$	Get the most recent, cached value of a cell.
CYCLE?	$() \rightarrow bool$	True if a cycle is present, false otherwise.
EVALEXPR	$cell \rightarrow value$	Evaluate the cell's formula to a value.
NOTIFYCYCLE!	$() \rightarrow ()$	After the call, calling CYCLE? returns true.
STATE	$cell \rightarrow state$	Get or set a cell's state.
SUPPORTED	$cell \rightarrow [cell]$	Get a list of the cell's supported cells.

Table 1: Overview of the functions used in pseudo-code.

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*.

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).

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)

```

4 Funcalc: Parallel Implementation

Funcalc, short for *parallel Funcalc*, is a parallel variant of Funcalc based on the .NET Task Parallel Library (TPL) [17]. 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 sec. 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 sec. 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 sec. 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 sec. 5). This makes de- and re-scheduling a waste of time. In terms of correctness, it is legal to create cyclic references in spreadsheets (see sec. 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) [14, 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 sec. 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 $STATE_{TS}$ are lock-free; reads from $CACHE_{TS}$ 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.

Algorithm 2 Thread-safe evaluation function.

```

1: function EVALPAR(cell)
2:    $s \leftarrow STATE_{TS}(cell)$ 
3:   switch  $s$ 
4:     case computing :
5:       while  $STATE_{TS}(cell) \neq uptodate$  do nothing
6:     case dirty or enqueued :
7:       if CAS( $STATE_{TS}(cell)$ ,  $s$ , computing) then
8:          $CACHE_{TS}(cell) \leftarrow EVALEXPR(cell)$ 
9:          $STATE_{TS}(cell) \leftarrow uptodate$ 
10:        ENQUEUE_SUPPORTEDTS(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 ENQUEUE_SUPPORTED_{TS} makes sure that each cell gets enqueued at most once (see algorithm 1).

Name	Signature	Description
CAS	$r \times c \times v \rightarrow bool$	Atomically set r to v if its value is c .
DEC	$counter \rightarrow ()$	Atomically decrease the counter by one.
ENCODEOWNER	$id \times state \rightarrow int$	Encode an identifier into a cell state's upper bits.
GET	$counter \rightarrow int$	Atomically get the counter's current value.
ID	$thread \rightarrow id$	Get the identifier for a thread.
INC	$counter \rightarrow ()$	Atomically increase the counter by one.
OWNERBITS	$int \rightarrow id$	Read only ownership bits from encoded state.
SPAWN	$fun \rightarrow ()$	Spawn a task to execute a function in parallel.
STATEBITS	$int \rightarrow state$	Read only state bits from encoded state.

Table 2: Overview of the functions particular to parallel recalculation.

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 sec. 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 [21] to keep track of the number of cells currently being evaluated.

Algorithm 3 Main algorithm for parallel minimal recalculation.

```

1: function MINIMALRECALCPAR(roots)
2:   counter  $\leftarrow$  ATOMICCOUNTER(0)
3:   for all r in roots do
4:     MARKDIRTYTS(r)
5:     ENQUEUETS(CQ, r)
6:     STATETS(r)  $\leftarrow$  enqueued
7:   while (GET(counter) > 0  $\vee$   $\neg$ EMPTY?(CQ)  $\wedge$   $\neg$ CYCLE?()) do
8:     cell  $\leftarrow$  DEQUEUETS(CQ)
9:     if cell  $\neq$  null then
10:      INC(counter)
11:      SPAWN(fun()  $\Rightarrow$  {
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*, enqueueing 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

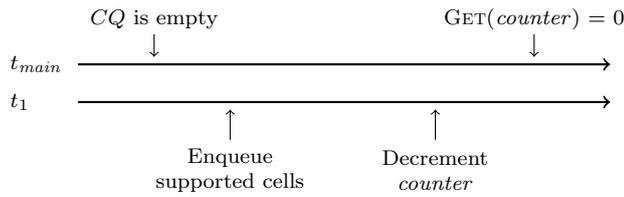


Fig. 4: A timeline (from left to right), showing the interleaving of actions that could cause premature termination of the algorithm if the termination condition were to be reversed. Thread t_{main} is the main thread and thread t_1 is a worker thread.

counter. In sec. 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 were empty and $\text{GET}(\text{counter}) = 1$. 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 $\text{GET}(\text{counter}) = 0$. 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 sec. 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, that impose an arbitrary, numerical order on threads, to determine *thread precedence*. A thread t_i has precedence over t_j if $\text{ID}(t_i) < \text{ID}(t_j)$. 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 $\text{OWNERBITS}(s)$ only returns the ownership bits of s , $\text{STATEBITS}(s)$ 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 sec. 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!_{TS} which then exits the main loop. Other threads will simply terminate when they are done evaluating their current 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.

Algorithm 4 Parallel speculative recalculation with dynamic cycle detection.

```

1: function TRYEVALEXPR(cell, vold, s)
2:   s' ← ENCODEOWNER(ID(tcur), computing)
3:   if CAS(STATETS(cell), s, s') then
4:     v ← EVALEXPR(cell)
5:     if CAS(CACHE(cell), vold, v) then
6:       STATETS(cell) ← uptodate
7:       ENQUEUESUPPORTEDTS(cell)
8: function EVALPARSPEC(cell)
9:   s ← STATETS(cell)
10:  vold ← CACHETS(cell)
11:  switch s
12:    case computing :
13:      if ID(tcur) = OWNERBITS(s) then
14:        STATETS(cell) ← uptodate
15:        NOTIFYCYCLE!TS()
16:      else if ID(tcur) < OWNERBITS(s) then
17:        while ID(tcur) < OWNERBITS(s) ∧ STATEBITS(s) = computing do
18:          TRYEVALEXPR(cell, vold, s)
19:          s ← STATETS(cell)
20:        else
21:          while STATETS(cell) ≠ uptodate do nothing
22:        case dirty or enqueued :
23:          TRYEVALEXPR(cell, vold, s)
24:  return CACHETS(cell)

```

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



Fig. 5: Two cells depend on the same cell containing 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.

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 sec. 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 sec. 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 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.

5 Results and Validation

5.1 Benchmark Spreadsheets

We use the following spreadsheet suites to benchmark Puncalc:

Real-World Spreadsheets LibreOffice Calc [12] provides a set of large benchmark spreadsheets¹. 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 sheet-defined functions. 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 behavior in a controlled and systematic fashion, we use six programmatically generated spreadsheet topologies, as shown in fig. 6. 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 0.7ms 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 32GB of memory, running 64 bit Windows 10, Version 1607, and .NET 4.7.1. We initially performed three warm-up runs and ran each benchmark for five iterations². 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.

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

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

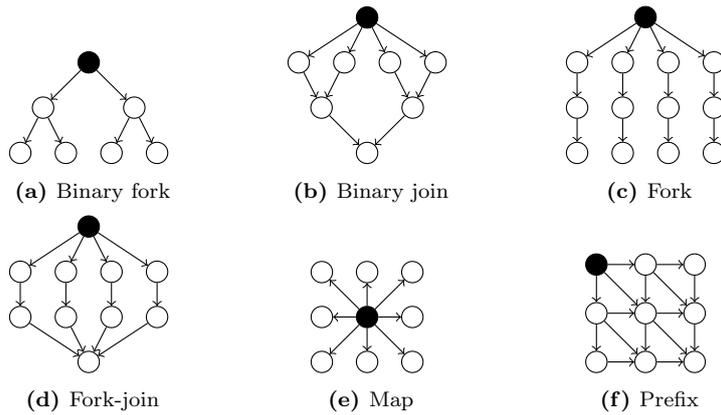


Fig. 6: Illustrations of the underlying support graph structures of synthetic spreadsheets for benchmarking. Black nodes mark recalculation roots. We only use one recalculation root per sheet to simulate editing a single cell.

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 (sec. 2.4).

5.4 Performance Evaluation

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

Observation 1 Fig. 7 shows that our approach scales for the majority of tested spreadsheets up to 16 cores, where we gain most speedup on average.

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.

Sheet	Cells	Roots	Support	Span	Seq. (s)	$\times 16$	$\times 16^*$	$\times 48$	$\times 48^*$
building-design	108,332	18,378	488,351,887	3	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

Table 3: Spreadsheet statistics and benchmark results. Columns labeled $\times n$ show relative speedup for n cores. Columns labeled $\times n^*$ show speedup for n cores with thread-local evaluation enabled, as described in sec. 4.6.

Observation 2 Thread-local evaluation does not improve performance compared to eagerly spawning a task for each cell as shown in fig. 8, 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 is 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. 6f), thread-local evaluation can alleviate the overhead of parallelization, which may be favorable for a robust implementation. However, recursive evaluation can lead to stack overflow errors. Second, the TPL [17] 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 [20]. All three systems require reengineering of the spreadsheet, which may take a substantial amount of time and require expert engineers [24]. In contrast, Puncalc runs

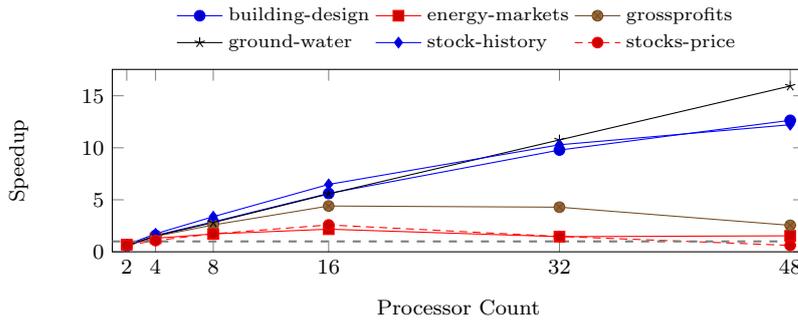


Fig. 7: Average benchmark results over 50 runs per spreadsheet from the LibreOffice Calc spreadsheet suite *with* thread-local evaluation enabled. Values are speedup factors over sequential performance on the same machine; higher is better. The gray dashed line indicates 1-core performance. The standard deviation is ≤ 0.21 for all benchmarks.

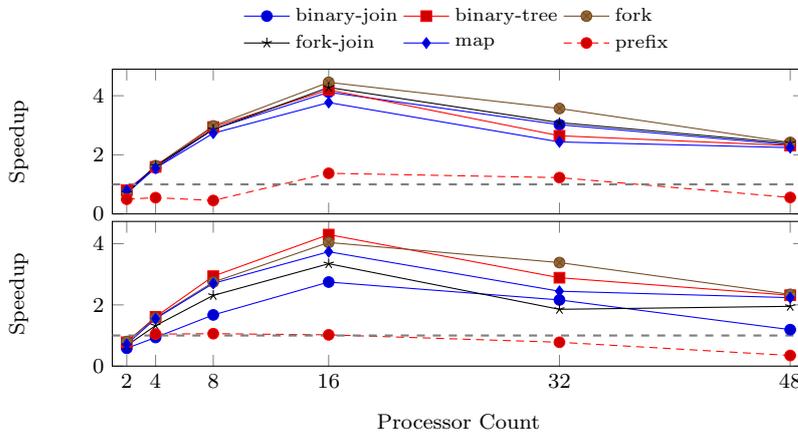


Fig. 8: Average benchmark results over 50 runs per spreadsheet from the synthetic spreadsheet suite. Top: *without* thread-local evaluation. Bottom: *with* thread-local evaluation. Values are speedup factors over sequential performance on the same machine; higher is better. The gray dashed line indicates 1-core performance. The standard deviation is ≤ 0.1 for all benchmarks.

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 [23] gives some additional information based on speculation and experimentation.

SpreadsheetGear [18] is a collection of commercial plugins 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 30-500 times speedups [19], 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 literature.

We have given a number of possible explanations for the performance results in sec. 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 sec. 6.

We believe that our work, combined with the work on sheet-defined functions [22, 23], 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.

References

1. Abraham R, Erwig M: Type Inference for Spreadsheets. PPDP '06.
2. Abraham R, Erwig M: Ucheck: A Spreadsheet Type Checker for End Users. In: Journal of Visual Languages & Computing 18(1) '07.
3. Abramson D, Sosic R, Giddy J, Hall B: Nimrod: A Tool for Performing Parametrised Simulations Using Distributed Workstations. HPDC, '95.
4. Abramson D, Roe P, Kotler L, Mather D: Activesheets: Super-Computing With Spreadsheets. HPC '01.
5. Biermann F, Dou W, Sestoft P: Rewriting High-Level Spreadsheet Structures into Higher-Order Functional Programs. PADL '18.
6. Bock AA: A Literature Review of Spreadsheet Technology. Technical Report, 2016. URL <http://www.itu.dk/people/albo/publications/>.
7. Burnett M: What Is End-User Software Engineering and Why Does It Matter? IS-EUD '09.
8. Burnett M, Cook C, Rothermel G: End-User Software Engineering. Commun. ACM 47(9) '04.
9. Casimir RJ: Real Programmers Don't Use Spreadsheets. SIGPLAN Not. 27(6) '92.
10. Erwig M, Burnett M: Adding Apples and Oranges. PADL '02.
11. EuSpRiG: Eusprig Horror Stories. URL <http://eusprig.org/horror-stories.htm>, accessed on 2016-06-14.
12. The Document Foundation: LibreOffice Calc. URL <https://www.libreoffice.org/discover/calc/>, accessed on 2016-05-09.
13. Harvey B, Wright M: Simply Scheme: Introducing Computer Science. 1999. MIT Press.
14. Herlihy M, Shavit N: The Art of Multiprocessor Programming. 2008. Elsevier/Morgan Kaufmann.
15. Hermans F, Pinzger M, van Deursen A: Supporting Professional Spreadsheet Users by Generating Leveled Dataflow Diagrams. ICSE '11.
16. Johnston WM, Hanna JRP, Millar RJ: Advances in Dataflow Programming languages. ACM Comput. Surv. 36(1) '04.
17. Leijen D, Schulte W, Burckhardt S: The Design of a Task Parallel Library. OOPSLA '09.
18. SpreadsheetGear: Easily Take Advantage of Multi-Core CPUs. URL https://www.spreadsheetgear.com/support/help/spreadsheetgear.net.7.0/Key_Concepts_Easily_Take_Advantage_of_Multi-Core_CPUs.html, accessed on 2018-01-18.
19. Meeks M: LibreOffice Calc: Spreadsheets on the GPU. 2014. URL <http://www.iwoc1.org/iwoc1-2014/abstracts/libreoffice-spreadsheets-on-the-gpu/>.
20. Microsoft: HPC Services for Excel. URL [https://technet.microsoft.com/en-us/library/ff877820\(v=ws.10\).aspx](https://technet.microsoft.com/en-us/library/ff877820(v=ws.10).aspx), accessed on 2018-03-28.
21. Oracle: Class LongAdder. URL <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAdder.html>, accessed on 2017-02-21.
22. Peyton-Jones S, Blackwell A, Burnett M: A User-Centred Approach to Functions in Excel. ICFP '03.
23. Sestoft P: Spreadsheet Implementation Technology. 2014. The MIT Press.
24. Swidan A, Hermans F, Koesoemowidjojo R: Improving the Performance of a Large Scale Spreadsheet: A Case Study. SANER '16.
25. Trudeau J: Collaboration and Open Source at AMD: LibreOffice. URL <https://developer.amd.com/collaboration-and-open-source-at-amd-libreoffice/>, accessed on 2015-07-31.
26. Wack AP: Partitioning Dependency Graphs for Concurrent Execution: A Parallel Spreadsheet on a Realistically Modeled Message Passing Environment. PhD thesis, 1996, Newark, DE, USA, URL <http://portal.acm.org/citation.cfm?id=269551>.
27. Yoder AG, Cohn DL: Real Spreadsheets for Real Programmers. ICCL '94.