Parallel Spreadsheet Programming for the Masses

PLDI 2018 Student Research Competition

Alexander Asp Bock (albo@itu.dk)
Department of Computer Science, IT University of Copenhagen, Denmark

Abstract Spreadsheets are abundant in many areas such as science and finance, and are likely the world's most widely used form of functional programming. Equally as ubiquitous are multi-core processors and for spreadsheet end-users to leverage them for heavy-weight computation, it has until now been necessary to hire experts. We report on static and dynamic approaches for enabling automatic parallelisation of spreadsheets, obtaining nearly a 16-fold speed-up and requiring no additional effort from end-users.

1 Introduction

Spreadsheets are abundant in many areas such as science and finance, and are likely the world's most widely used form of functional programming [12]. Equally as ubiquitous are multi-core processors, and for spreadsheet end-users to leverage them, it has until now been necessary to hire experts to re-engineer spreadsheets. Recalculation of spreadsheets can be slow: for instance Swidan et al. [15] report on a case study of refactoring a spreadsheet that would regularly take 10 hours to recompute. Our aim is to let end-users transparently and automatically use their multi-core machines to accelerate recalculation, and the immutability of spreadsheet cells make them a prime candidate.

```
let rec fact n = if n < 2 then 1
                 else n * fact(n - 1)
```

	A	B
1		=DEFINE("FACT", B3, B2)
2	"n="	0
3	"out="	=IF(B2<2, 1, B2*FACT(B2-1))

Figure 1. *Computing the factorial using recursion in F# and Funcalc. SDFs are defined using the* `DEFINE` *function where cell B3 is the output cell and B2 is the input cell.*

Funcalc [13] is a research spreadsheet engine featuring *sheet-defined* functions (SDFs), that allows end-users to define higher-order functions in a paradigm which they are already familiar with. Fig. 1 shows the definition of a recursive factorial function in F# and in Funcalc. SDFs are compiled to .NET bytecode and automatically recompiled when a user edits cells in its definition. In addition to earlier work [13, 14], we have highlighted the expressive power of SDFs by translating 16 SISAL programs of varying complexity to Funcalc.
We can approach the problem of automatic parallelism in spreadsheets in many ways: statically or dynamically with local or global exploitation of parallelism, and the design space is yet to be fully explored. We investigate both a static approach which partitions cells globally, and a dynamic approach that finds available parallelism. The static approach constitutes the author's own work, while the remaining work is collaborative.

PLDI 2018 Student Research Competition, June 18 – 22, Philadelphia, PA, USA
ACM Student Member Number: 5743209
Advisor: Professor Peter Sestoft
Category: Graduate

2 Related Work

Static partitioning is inspired by Sarkar et al.'s work [11] on an optimising compiler for the purely functional language SISAL. The compiler automatically partitions the program to exploit opportunities for parallel execution, which helped showcase that functional super-computing was viable [5]. Partitioning happens during an intermediate compilation step where programs are transformed into a graph format akin to the dataflow graph that exists between cells in a spreadsheet. Wack [16] used spreadsheets as a model for computer networks, but partitioned only predefined topologies. Others have built distributed solutions [1, 2, 9], but these require manual effort by end-users which we wish to avoid. Local parallelism has been exploited by Biermann et al. [4] where high-level spreadsheet structures are rewritten to higher-order SDF calls. To our knowledge, no previous work has attempted to statically partition general spreadsheets for parallel execution. Little scientific literature has dealt with dynamic approaches to automatic parallelism in spreadsheets.

3 Contributions

3.1 Static Partitioning

We statically partition cells globally into acyclic, independent groups, that can be run in parallel using a simple scheduling algorithm. Our algorithm approximates and balances the partition's parallelism and synchronisation costs which guides the partitioning. Starting with a fine-grained partition, the algorithm iteratively merges groups until it reaches the coarsest partition with a single group without any synchronisation overhead, but no parallelism either. Groups are merged so that the critical path length, the largest sequential chain of work in the partition, and the amount of synchronisation between groups, are minimised. The best, intermediate partition found during merging is selected as

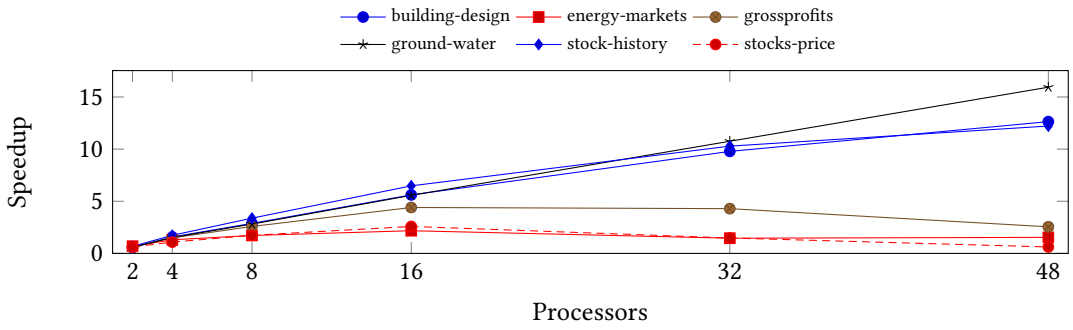

Figure 2. *Average benchmark results for our dynamic, parallel spreadsheet evaluator over 50 runs per spreadsheet from the LibreOffice Calc [7] spreadsheet suite. Values are speed-up factors over sequential performance; higher is better.*

the output of the algorithm. We exploit common spreadsheet structures known as *cell arrays* [18] that naturally express parallelism by grouping them together, lowering the running time of the algorithm as there are fewer groups to merge.
Static partitioning requires a good cost model and we are currently developing big-step semantic cost rules for Funcalc's formula language to get accurate cost estimates. We also plan to use these semantics to obtain cost estimates for running a recalculation, estimate work for off-loading computation to GPGPUs and build an abstract interpreter for Funcalc's formula language [19].

3.2 Dynamic Spreadsheet Evaluator

Whereas the static approach attempts to globally load-balance work, our dynamic approach [3] corresponds to running the finest partition. A main thread continuously polls a global work queue of cells waiting to be computed and dispatches thread pool tasks for each using the Task Parallel Library [8]. Cycles are usually allowed in spreadsheets, so our algorithm is capable of detecting cycles in parallel using a method denoted as *speculative reevaluation*. In short, threads claim ownership of cells using their thread ID. To break cyclic dependencies, we allow threads with lower IDs to reevaluate cells, owned by threads with higher IDs, multiple times.

4 Results

While the static partitioning algorithm is actively being developed, our dynamic, parallel spreadsheet interpreter obtains between 1.4- and 6.5-fold speed-ups on 16 cores and roughly 16-fold speed-up on 48 cores on a set of benchmark spreadsheets taken from LibreOffice Calc [7], without any development effort required from end-users (see fig. 2). We expect that the load-balancing of static partitioning will yield even better speed-ups.
We hope that automatic parallelisation coupled with the expressive power of SDFs can transform spreadsheets into a powerful programming tool for end-user development and encourage a positive change in the general perception of spreadsheets [6, 10, 17].

References

[1] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: a tool for performing parametrised simulations using distributed workstations. HPDC '95.
[2] D. Abramson, P. Roe, L. Kotler, and D. Mather. Activesheets: Supercomputing with spreadsheets. HPC '01, Citeseer.
[3] A. A. Bock and F. Biermann. Puncalc: Task-based Parallelism and Speculative Reevaluation in Spreadsheets. In submission.
[4] F. Biermann, W. Dou, and P. Sestoft. Rewriting High-Level Spreadsheet Structures into Higher-Order Functional Programs. PADL '18.
[5] D. Cann. Retire Fortran? A debate rekindled. CACM, '92.
[6] R. J. Casimir. Real Programmers Don't Use Spreadsheets. SIGPLAN Not. '92. ISSN 0362-1340.
[7] The Document Foundation. Libreoffice Calc. URL https://www.libreoffice.org/discover/calc/.
[8] D. Leijen, W. Schulte, and S. Burckhardt. The Design of A Task Parallel Library. SIGPLAN Not. '09. ISSN 0362-1340.
[9] Microsoft. HPC Services for Excel. URL https://technet.microsoft.com/en-us/library/ff877820(v=ws.10).aspx.
[10] S. Peyton-Jones, A. Blackwell, and M. Burnett. A User-centred Approach to Functions in Excel. ICFP '03, ACM. ISBN 1-58113-756-7.
[11] V. Sarkar. Partitioning and Scheduling Parallel Programs for Multiprocessors. Research Monographs In Parallel and Distributed Computing. MIT Press, '89. ISBN 0262691302.
[12] C. Scaffidi. Counts and earnings of end-user developers. URL https://www.linkedin.com/pulse/counts-earnings-end-user-developers-chris-scaffidi?published=t.
[13] P. Sestoft. Spreadsheet Implementation Technology. MIT Press, '14. ISBN 9780262526647.
[14] P. Sestoft and J. Zeilund. Sheet-defined functions: implementation and initial evaluation.
[15] A. Swidan, F. Hermans, and R. Koesoemowidjojo. Improving the Performance of a Large Scale Spreadsheet: A Case Study. SANER '16, IEEE. ISBN 978-1-5090-1855-0.
[16] A. P. Wack. Partitioning Dependency Graphs for Concurrent Execution: A Parallel Spreadsheet on a Realistically Modeled Message Passing Environment. PhD thesis, Newark, DE, USA, '96.
[17] A. G. Yoder and D. L. Cohn. Real spreadsheets for real programmers. ICCL '94.
[18] R. Mittermeir and M. Clermont. Finding high-level structures in spreadsheet programs. WCRE '02. ISBN 0-7695-1799-4.
[19] D. A. Schmidt. Trace-Based Abstract Interpretation of Operational Semantics. LISP and Symbolic Computation '98. ISSN 1573-0557.