IT University of Copenhagen

Declarative Parallel Programming in Spreadsheet End-User Development

A Literature Review

Florian Biermann

IT University Technical Report Series TR-2016-192
ISSN 1600–6100 February 2016

ISSN 1600–6100

ISBN 978-87-7949-357-5

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 Copenhagen S
Denmark

Telephone:	+45 72 18 50 00
Telefax:	+45 72 18 50 01
Web	`www.itu.dk`

Declarative Parallel Programming in Spreadsheet End-User Development
A Literature Review

Florian Biermann
fbie@itu.dk

Abstract

Spreadsheets are first-order functional languages and are widely used in research and industry as a tool to conveniently perform all kinds of computations. Because cells on a spreadsheet are immutable, there are possibilities for implicit parallelization of spreadsheet computations. In this literature study, we provide an overview of the publications on spreadsheet end-user programming and declarative array programming to inform further research on parallel programming in spreadsheets. Our results show that there is a clear overlap between spreadsheet programming and array programming and we can directly apply results from functional array programming to a spreadsheet model of computations.

1 Introduction

Domain experts from widely different disciplines within industry and science use spreadsheets to conveniently model complex computations [46]. Research on spreadsheet technology has recently gained increased interest, which is probably due to a renewed popularity of functional programming principles. Spreadsheets do not possess mutable state, which makes reasoning about their correctness rather easy. Moreover, this particular trait makes it possible to investigate transparent and implicit parallelization of spreadsheet calculations. The widespread use of spreadsheets together with the apparent opportunities for spreadsheet parallelization provide synergy for research on "popular parallel programming".

As an example of recent development in parallel spreadsheet technology, AMD has implemented OpenCL kernel generation in LibreOffice Calc. Their modification generates kernels from operations on cell areas, which highly parallel GPGPU hardware then executes [54].

In this literature study, we want to provide the reader with an overview of the publications on spreadsheet end-user programming and declarative array programming to inform further research on parallel programming in spreadsheets with a focus on functional programming and higher-order functions.

Declarative array programming allows the language run-time to re-write expressions to perform implicit parallelization of computations. The user does not need to be aware of the parallelism inherent to a problem and as long as they choose to use the highest abstraction to model their computation [5], a sophisticated compiler can infer such parallelism automatically. We focus on approaches that do not require special hardware, i.e. shared-memory multiprocessors.

1.1 Approach

Systematic literature reviews have gained a lot of traction in the software engineering research community and Kitchenham and Charters [31] have developed practical guidelines for conducting such reviews. The methodology is strict and makes studies reproducible.

However, systematic literature reviews usually focus on generating meta-level insight on the cumulative research progress by analyzing quantitative or qualitative studies. As already outlined, we instead want to map out the research landscape of two disjoint research areas. Luckily, there exists a methodology that is based on systematic literature reviews which allows for a less strict analysis of publications, namely systematic mapping studies [42].

Systematic mapping studies can identify areas that lack formal research and provide an overview of the current state of the art. The trade-off between a systematic literature review and a systematic mapping study is that of depth against breadth [31, 42]. Therefore, systematic mapping study is our methodology of choice.

However, for our purpose, even systematic mapping studies are too strict, as we do not want to chart out, say, all existing research on spreadsheet technology. Instead, we want to identify possibly non-obvious synergies between array and spreadsheet programming. Therefore, we chose the middle way where we do not synthesize insight from the accumulated publications but still focus on a somewhat deeper understanding of their contributions

We present a detailed protocol of the study in Sections A and B.

1.2 Research Questions

We formulate our research questions as follows:

1. What is the state of the art in declarative parallel array programming languages?
2. What are the most scrutinized topics in array programming?
3. Has there been research on declarative parallel array programming in a spreadsheet model of computation?
4. Are there any obvious or non-obvious ways to combine declarative parallel array and spreadsheet end-user programming?

1.3 Threats to Validity

Naturally, even with a well defined methodology, the validity of our research can be questioned. There are two problems which we discuss here briefly.

First, systematic mapping studies gain validity by redundancy: a group of researchers performs the same classification tasks in order to minimize bias [31]. Due to restrictions in personnel, the amount of redundancy is severely limited in our study.

Secondly, we must acknowledge that this study has blind spots in terms of scientific literature but also in terms of patents and industry development. We have omitted focusing on patents as the bare amount of patent applications on spreadsheets is overwhelming. The former however is due to limitations in the search results we have obtained and due to possibly too restrictive exclusion of publications due to the aforementioned limited redundancy.

Sestoft [44] provides a comprehensive but quite dated list of patents on spreadsheet technology.

2 Declarative Parallel Programming

We have identified four key topics within parallel array programming: nested data parallelism, array fusion, data structures and loop parallelization. While these clearly are interconnected and we therefore cannot just see them all by themselves, we will use them to structure the review of the research body on parallel array programming.

There are a few key languages that appear in the literature. The two oldest are Fortran, which is an imperative high-performance language, and APL [26], which is a high-level declarative and functional language. Furthermore, much research has focused on Single-assignment-C (SAC), which is a C-like functional language, and Data Parallel Haskell (DPH) and REPA, both of which are library and compiler extensions for Haskell. Another important language is NESL [6], which introduced the idea of statically flattening nested arrays.

The opportunity for parallelism in these languages stems from bulk-operations over arrays. Where imperative languages like Fortran use iterative loops, functional languages depend on higher-order functions like `map` or `fold` to express data-parallel computations. Ching [15] argued that it is easier for programmers to express parallelism in such a declarative high-level way than by explicitly scheduling work to different processors and implemented such an approach in the APL370 compiler. The only requirement to make use of such implicitly parallel constructs is that the programmer writes idiomatic code which the language runtime can parallelize efficiently [5]. Not only that parallelism is much easier to extract, but also sequential programs written in such a high-level style will often perform faster [5].

2.1 Nested Data Parallelism

Nested data parallelism describes the nesting of parallel operations over a nested array. A standard example is matrix-vector multiplication. We can represent a matrix as an array of vectors and then implement the operation conveniently via higher-order functions [27], here illustrated in F#:

```
type SparseVec = (int * float) []
type SparseMat = SparseVec []

let vecMul (sv : SparseVec) v =
    sum (map (fun (i, e) -> (get v i) * e) sv)
```

```
let matMul (sm : SparseMat) v =
    sum (map (fun sv -> vecMul sv v) sm)
```

The type SparseVec is a list of index-value pairs, which models all non-zero entries of a vector. We can then represent a sparse matrix as an array of sparse vectors, as in SparseMatrix. To compute the dot product of a sparse and a dense vector, we sum up the point-wise multiplication of the two vectors' scalars, as expressed in the vecMul function. The dot-product of a sparse matrix and a dense vector, matMul, is then the sum of the dot-products of each of the sparse vectors with the dense vector.

Such a declarative implementation of matrix multiplication lends itself naturally to an implicitly parallel execution. In this example, parallelism stems from the point-wise multiplication in vecMul as well as from the map in matMul: the latter is a possibly parallel operation that performs possibly parallel operations. Since the single vector columns of a sparse matrix might vary in length, simply assigning the same amount of sub-vectors to each processor might result in bad work balancing. Early work on the parallel language Actus [41] recognizes the difficulty of nested data parallelism and as a temporary solution restricts the level of parallelism to one dimension which the programmer has to choose ahead of time.

A naive way of implementing such nested parallelism would be to simply start new parallel threads from within each parallel thread. There would be quite some overhead to this solution. Research on nested data parallelism has focused on eliminating this overhead in two different ways.

2.1.1 Flattening

Blelloch et al. [8] introduced the idea of statically flattening nested arrays to perform an optimal parallelization of operations over arrays. The key to doing so is the right choice of array representation and there exists a representation which enables the compiler to flatten nested arrays in constant time. Flattening requires the generation of lifted versions of all functions, called lifting. Lifting the addition operator + of type $int * int \to int$, for example, returns a new operator $+_L$ of type $[int] * [int] \to [int]$ that takes arrays of scalars as parameters instead. The function has been "lifted" to the next rank. NESL implements this flattening scheme and Blelloch and Greiner [7] showed, based on NESL's built-in cost semantics, that NESL actually can be implemented without any additional overhead due to flattening.

Other researchers have since taken up the idea of flattening nested parallelism again [34] and implemented it in Haskell. In contrast to NESL, Haskell is a fully-fledged functional programming language with types and higher-order functions that is compiled instead of interpreted.

As it turns out, flattening nested data parallelism is much harder to achieve in such a higher-order language. The presence of higher-order functions makes lifting of functions more complicated [34]. Higher-order flattening introduces many intermediate arrays that take time and space. Therefore, Keller et al. [30] developed a technique to avoid flattening in such cases.

The more recently developed high-level language NOVA builds on the principles developed in NESL and DPH, using a lot of the state-of-the-art techniques and targets both, shared-memory multiprocessors as well as general purpose GPUs [16].

Flattening nested data parallelism is tightly coupled to array fusion, which we will discuss in detail in Section 2.2.

2.1.2 Dynamic Scheduling

Flattening nested data parallelism statically targets homogeneous computations on possibly irregular arrays. Possible imbalance of work distribution is a result of irregular nested data structures, such as sparse vectors, where adjacent zero-valued entries are not stored in memory. This demands a strong type system, such that all elements of an array are of the same type. Otherwise, a two-dimensional array might mostly contain numbers but spuriously also more complex data types such as arrays. This means that not the entire parallelism is visible to the compiler, so static flattening might not be feasible to balance the workload.

Dynamic scheduling leaves the distribution of work to run-time of the program. SAC, for instance, uses work-stealing queues [12, 20] to dynamically schedule work. Such scheduling schemes can produce some overhead at run-time that we otherwise could alleviate by using compile-time transformations. To minimize overhead for different kinds of computations, Fluet et al. [18] argue for a mix of schedulers and to let the run-time choose which scheduler to use for distributing work.

A notable scheduling heuristic is lazy tree splitting [4]. In this scheme, every thread gets assigned some part of the workload. Threads communicate with other threads via a technique inspired by and as efficient as work-stealing queues. Representing arrays via balanced binary trees makes splitting arrays a constant time operation. When a worker thread iterates over an array, it checks at every n-th iteration step whether any of the remaining threads are idle. If so, it splits its remaining array in half, dispatches the latter half to the idle threads and continues to the next iteration step. This heuristic shows low overhead in experimental settings [4].

2.2 Fusion

Fusion refers to avoiding intermediate representations of arrays for consecutive bulk operations. For instance, we can fuse two succeeding applications of `map` as follows, where `.` is the function composition operator:

```
map g (map f xs)   ⟹   map (f . g) xs
```

This optimization is very valuable in all kinds of programs, sequential and parallel alike, for example in sequential Fortran 90 programs [24].

Fusion is a static technique performed at compile time. Chakravarty and Keller [9] express the fusion transformations as straight-forward equational rewrite rules. They also observe that flattening nested arrays makes fusion a much simpler task which emphasizes the connection between flattening and fusion. Some researchers have been focusing on making such optimizations visible to the programmer via types [33]. This enables programmers to reason about the performance of their declarative code.

More aggressive fusion is possible if the compiler performs a more complex analysis. Henriksen and Oancea [23] use a data-flow based graph-reduction to analyze functional programs with second-order functions on arrays for fusion possibilities. Their analysis can detect code structures that inhibit fusion, subsequently re-writes the program in such a way that fusion becomes possible and avoids duplicate computations

A special variant of fusion is destructive update analysis. Since data structures are immutable in purely functional languages, writing to a single index of an array produces a new array. The update operation copies all elements from the original array with exception of the index that it updated. If the updated array is not referenced again throughout the program, we can perform the update in-place, or destructively, without the overhead of copying all data. Sastry and Clinger [43] developed an analysis for destructive array updates using live-variable data-flow analysis.

2.3 Data Structures

Choosing the right data structure is a key element to high-performance array programming. Lowney [35] developed carrier arrays as an extension to APL that are able to express irregular nested parallelism, in contrast to regular nested parallelism where all sub-arrays must be of the same length. Carrier arrays decide automatically to which rank a function must be lifted to, to be applied to all elements of the array.

Performance of single operations on arrays are also of concern. For instance, functional arrays should be able to perform constant-time lookup and preferably also constant-time update [40]. It is well-known that data structures embody a trade-off between the asymptotic complexity of the different operations, like random access, update, appending etc. Nevertheless, Stucki et al. [52] developed a general-purpose array data structure, the *relaxed radix-bound* (RRB) array. They proved the bounds for all operations and for practical sizes of an RRB array to be constant or amortized constant. RRB arrays are of clear value to parallel array programming, especially in conjunction with dynamic scheduling schemes.

It is a challenge to implement such high-performance arrays in a purely functional language. Arvind et al. [2] developed I-Structures which conceptually are write-once arrays. Each subscript can be written exactly once during the entire program. Reads and writes to indices can be re-ordered according to re-write rules at the cost of sacrificing referential transparency.

Type-based run-time specialization of functions and optimization of data structures is a widely applicable technique in functional programming [21]. REPA [29] uses types in order to represent the (irregular) shapes of arrays and to specialize higher-order functions. REPA arrays consist of unboxed values and are lazy. Forcing a single element of an array forces the evaluation of the entire array, where the single array elements can evaluate in parallel. Lazy arrays can avoid creating intermediate arrays and therefore do not require fusion [29].

2.4 Loop Parallelization

Parallelism in imperative languages is expressed via some kind of `do-loop` construct that roughly translates to “for each element of a list or for each integer in some range, perform the given body”. The problem with this style of parallelism is that imperative languages allow for non-trivial data dependencies and side-effects. As a consequence, the analysis that a compiler needs to perform in order to safely parallelize a `do-loop` is much more complicated than when using higher-order functions.

One way to safely parallelize imperative loops is to enforce a read-write order that can be computed at compile time [53]. When the read-write order of elements is known, a reading thread synchronizes with the writing thread to avoid lost updates. A similar approach uses a data-flow analysis of accesses to array-subscripts on an inter-procedural level [36]. Instead of enforcing an ordering of read and write accesses, Knobe and Sarkar [32] simply fall back to single-assignment arrays, thereby eliminating a whole class of data dependencies in loops. The most dominant technique for data dependency analysis is data-flow analysis of loops and array indices [32, 36].

In bounded, iterative loops, the bounds must be checked after every iteration. To alleviate this costly and repetitive computation, Henriksen and Oancea [22] lift the bounds check out of the loop at compile time and specialize the body for the bounds of the loop.

These techniques for imperative loop-parallelization are relevant for the implementation of high-level operations over arrays.

2.5 Other Topics

This section summarizes research that does not fit into the four major categories.

2.5.1 Domain-Specific Languages

Some of the research on array programming focused on domain-specific languages (DSLs). Grelck et al. [19] implemented explicit coordination of parallel computations in SAC via an embedded DSL in a declarative fashion. This, however, removes some of the declarative nature of program code. Another DSL on top of SAC is StagedSAC which adds compile-time shape inference of nested arrays to SAC [55]. The compiler adds all requirements that it could not prove statically to be satisfiable as run-time checks to the program.

2.5.2 Retran

Retran is a declarative, Fortran-like, purely functional language [47]. Similar to the APL extension by Lowney [35], Retran automatically applies lower-rank functions to all elements of a higher-rank array. Therefore, there are no higher-order functions in Retran. Retran uses anti-currying to lift functions to the required rank [47].

2.5.3 Remap and Data Layout

Declarative functional programming languages tend to free the programmer from thinking explicitly about data layout. For performance reasons, it can be necessary to expose data layout to the programmer. Also, high-level languages often provide functions to transpose two-dimensional arrays or to more generally change the layout of an array. The most general operator of this kind is `remap`, sometimes also referred to as `scatter` on the assignment's right-hand side and `gather` on the assignment's left-hand side.

Walinsky and Banerjee [56] implemented implicit remapping at compile time for functional programming languages. They provide inference rules for remapping. By using such a remapping, they effectively avoid intermediate representations of arrays. Implicit remapping also aids improving vectorization of higher-order functions, as shown by Sinkarovs and Scholz [49].

The ZPL programming language is based around regular-shaped arrays and a concept called regions which roughly equal named index-sets [11]. Furthermore, it makes all communication visible to the programmer through syntax and types, featuring a "what you see is what you get" performance model [10]. Data remapping is a crucial operation in ZPL. Deitz et al. [17] show ZPL's `scatter-gather` operator semantics. The operator can modify data layout of arbitrarily ranked arrays and exhibits high performance.

3 Spreadsheet Technology

Spreadsheets are visual programming environments. A collection of spreadsheets is a workbook. A spreadsheet contains a rectangular grid of cells. Each cell contains either a constant or a formula that computes a value and possibly references other cells.

One major topic in research on spreadsheet programming is the lack of abstraction: spreadsheets bundle data and computations in a single representation [25]. Moreover, spreadsheets encourage copying of formulas across cells to replicate computations [3, 39]. This lack of abstraction makes spreadsheets less powerful than general purpose programming languages [38].

Another main topic is general programming paradigms in a spreadsheet model of computation. Researchers have augmented spreadsheets with object orientation [3] and more declarative programming approaches [48, 51] which we will look at in greater detail in Section 3.2. Again, even though we cannot strictly separate abstraction and programming paradigms, this categorization is convenient for the discussion of how researchers propose to handle the complexity of spreadsheet models.

There are many more topics in spreadsheet research, i.e. testing of spreadsheets, visualization or analysis of "real-life" spreadsheet corpora, to only name a few. We focus however on research on end-user programming in a spreadsheet model of computations and will therefore not consider these other topics here.

3.1 Abstraction

It is convenient to define two groups of spreadsheet abstraction. That is (1) manual abstraction, where the user has the means to build their own abstractions so to hide implementation details and (2) automatic abstraction, where the user constructs spreadsheets in a familiar way and the system later analyzes them to infer the model and to (subsequently) separate it from data.

3.1.1 Manual Abstraction

Many researchers observed that spreadsheets lack the most basic abstraction of general-purpose programming languages: named functions [28]. Named functions make it possible to hide implementation detail that is not important for the overall computations of a specific model. Therefore, Jones et al. [28] proposed to allow end-users to define their own abstractions in terms of spreadsheet computations. Each newly introduced function is essentially a spreadsheet prototype that has designated input cells and a designated output cell. Each time the user calls such a sheet-defined function, a new spreadsheet instance of this function is instantiated to perform the computation.

Sestoft [45] extended upon this idea by allowing sheet-defined, recursive and run-time compiled functions. This approach is more general and alleviates the need for instantiating explicit spreadsheets. This approach is implemented in the experimental spreadsheet engine Funcalc which is described in great detail in [46].

3.1.2 Automatic Abstraction

Researchers have developed systems that analyze spreadsheets to infer their model and to (subsequently) separate this model from data. This allows users to build spreadsheets in a familiar manner. Isakowitz et al. [25] developed a system that automatically performs such a separation and manages spreadsheet logic for modular re-use. They observe that the majority of spreadsheet errors they encounter are not simple off-by-one reference errors and typos but severe errors in the model. They relate them to classic programming errors where the programmer has not chosen an adequate level of abstraction.

The visual layout of spreadsheets is often implicit documentation of the logic. Mittermeir and Clermont [39] however, observe that this visual layout often leads to misconceptions if another user takes over the spreadsheet. Therefore, they developed a set of logical and semantic equivalence classes for cells. These equivalence classes help visualizing repetitions in spreadsheet grids, which are the high-level structures a user needs to understand in order to be able to maintain the spreadsheet.

Types are also useful abstractions over spreadsheets. The literature includes different type inference systems that make the user aware of formulas where the expected type differs from the actual type [1, 14]. Researchers have proposed different solutions to handle types in spreadsheets and a common problem is efficient typing of cell areas [1, 13]. We classify types as a kind of automatic abstraction because the types are inferred rather than annotated.

3.2 Programming Paradigms

Researchers have also proposed to apply different programming paradigms to the spreadsheet domain with a focus on raising the abstraction level.

3.2.1 Object Orientation

Functional Model Development (FMD) [3] is a domain-specific language for Excel and exposes objects to spreadsheet users. Objects are an accumulation of data with some functions defined on these objects. Users can use a special syntax to declare variables that model input parameters for user-defined functions. Functions are defined inline (as in on the same spreadsheet) by prototype formulas where the cell that would yield the result actually evaluates to the new defined function. As spreadsheets encourage copying of formulas over the same row or column, FMD introduces a high-level `map` operator that applies the same user-defined function across a column or row.

To apply stronger separation of implementation and instantiation, Mendes [37] developed ClassSheet. The logic of a computation is defined in a model-spreadsheet, while each common spreadsheet that performs the computations is an instance of this model. This approach resembles a manual version of the abstraction model by Isakowitz et al. [25].

3.2.2 Constraint Programming

Stadelmann [51] proposed to let spreadsheet users express their models by providing the system with constraints to solve. This approach considerably reduces the amount of code required to model complicated logic [51].

	A	B	C
1	23	`= F(A1)`	`= G(B1)`
2	42	`= F(A2)`	`= G(B2)`
3	96	`= F(A3)`	`= G(B3)`

	A	B	C
1	23	`= F(A1)`	`= J(B1, B2)`
2	42	`= F(A2)`	`= J(B2, B3)`
3	96	`= F(A3)`	`= J(B3, B1)`

Figure 1: Examples of non-fusable stencil-like structures on interpreted spreadsheets. *Left:* columns *B* and *C* cannot be fused because every cell needs to evaluate to the result of its formula. *Right:* columns *B* and *C* cannot be fused because the intermediate values of column *B* are referenced more than once in column *C*.

However, due to the requirement of being able to name a cell multiple times in a constraint, it is infeasible to let constraints directly replace cell formulas. Instead, the system provides a second window that contains constraints. This side-steps the spreadsheet model slightly.

3.2.3 By Example

Programming by example allows users to explain how to transform data by performing a few transformations manually, from which the system can infer general transformation rules. This is useful for bulk-processing similar items. Singh and Gulwani [48] extended Excel with a DSL that allows users to provide such example transformations such that Excel then automatically transforms the remaining items. They combine a probabilistic approach of parsing with joint learning of transformation rules. They require however that the data type that a user wants to transform already exists as a predefined model.

4 Discussion

Much research on spreadsheet end-user programming focuses on bulk-data transformations. Also, researchers aim to avoid repetition in spreadsheets to avoid error sources. Semantic spreadsheet analysis reveals repetitive patterns of homogeneous computations that are similar to explicit mappings of functions. These observations fit nicely to declarative, higher-order array programming and we will discuss them in greater detail in the following sections.

4.1 High-Level Structures and Functional Programming

Detecting high-level structures in spreadsheets relates directly to identifying repeating formulas [39], much like stencils in scientific computing. Such repetition lowers the level of abstraction and also hides potential for parallelism.

A system that detects repeating formulas would also be able to parallelize these repetitions, as they essentially are an unwrapped bulk-application via a higher-order primitive like `map`. The experimental spreadsheet platform Funcalc already infers and maintains explicit knowledge about copied formulas by means of the support-graph [46].

It is unclear if an even better understanding of such stencil-like structures in spreadsheets leads to possibilities for loop fusion. For instance, if all values in column A are first transformed by a repeating formula in column B and then subsequently by another column C, one could in principle avoid the middle step and fuse the computations in columns B and C so that one (implicit) `map` would suffice. However, due to spreadsheet semantics, each cell must evaluate to the value its formula computes [46]. This forbids fusion, as the intermediate values would vanish and could not be displayed to the user. Fusion would however be allowed in compiled user-defined functions. Furthermore, if the formulas in column C references the cells in column B more than once each, fusion becomes outright impossible. In the former case, the system could suggest the user to manually inline the formulas for performance reasons. In the latter case, this is not possible. Figure 1 illustrates cases where fusion is not possible.

4.2 Representing Arrays

Even though not visible to the spreadsheet user, arrays need to be implemented efficiently under the hood to allow for high-performance computations. Even though not available currently in spreadsheet systems like Excel, index update or arbitrary remap operators are widespread in array programming and therefore must also be present in a spreadsheet language for array programming.

Our literature study has shown that there has been much research on array implementations [29, 40, 52] that we can nearly directly apply to a spreadsheet environment. Most array designs focus on one-dimensional arrays, but in a spreadsheet model, arrays are two-dimensional and regular. Generalizing from one dimension to two dimensions or more seems not problematic as long as these arrays remain regular.

Lazy arrays, as implemented in REPA [29], are interesting for avoiding intermediate structures in single cells. They do not extend beyond that because fusion of intermediate cells is not possible, as discussed in Section 4.1.

Destructive array updates [43] are also only interesting in the context of single cells. The analysis is much simpler, for the same reason of why fusion across cells is not possible and because there are no let-bindings within the formula of a single cell.

4.3 Flattening and Scheduling

Nested parallelism is a huge topic in array programming and has gained a lot of traction [8, 16, 34] because it allows programmers to arbitrarily combine parallel operations without needing to worry about whether there is any overhead introduced by the nested parallelism.

Static flattening, as in NESL [8] or in DPH [34], applies well to regular shaped arrays that have a type which is statically inferable. Cell areas on spreadsheets are regular arrays. In a simple spreadsheet model, the type is less important, because there only exist scalar values.

In an experimental platform like Funcalc, a single cell can also contain an entire array [28, 46]. This makes the type of arrays more interesting again, because any array might now contain more arrays and therefore generalize to irregular three- or even higher dimensional arrays. Arrays might even contain a mixture of scalar and array values. These cases severely complicate static scheduling as work now cannot be distributes across threads evenly any more.

There are two techniques that can help here. One is to simply resort to a dynamic scheduling approach, like lazy tree splitting [4], instead of trying to solve the scheduling problem statically. The low overhead suggests that this is a feasible solution.

The second is a hybrid solution of dynamic and static scheduling, where the choice of the scheduler is made by static type analysis of cells [1, 13, 14]. If an array has a statically determinable type, we can safely perform static flattening of the nested parallel computation. Otherwise, the system resorts to dynamic scheduling and accepts the small overhead. Additionally, the system could warn the user about possible performance losses.

5 Conclusion

In this study, we have summarized the literature on declarative array programming and end-user spreadsheet programming using systematic literature mapping as a methodology. The reviews show that there is a clear overlap between spreadsheet programming and array programming and we can directly apply results from functional array programming to a spreadsheet model of computations.

Current research focuses still on classic high-performance languages like Fortran, but modern languages are now also now vehicles for research on parallel array programming and many have been designed with certain goals in mind, e.g. ZPL [50]. Moreover, new languages like NOVA [16] emerge to combine the collective research effort. The majority of these modern languages are purely functional, which not reflects the idea that the absence of shared mutable state makes programming and reasoning about parallel programs much easier for the end-programmer, but also allows more compiler optimizations.

We have discussed and categorized research topics on array programming into three major fields: nested data parallelism, loop fusion and efficient implementation of arrays.

There seem not to exist many publications that report on combining array programming and spreadsheet programming. Notable exemptions are experiments by AMD and LibreOffice [54] as already noted in Section 1, and the domain specific language FMD [3]. This suggests that there is much room for further research on array programming in a spreadsheet model of computations.

Our analysis shows that there are broad possibilities to combine spreadsheet programming with array programming technology to implement implicit parallelization of operations over explicit and implicit arrays. Spreadsheet computations are of a repetitive nature [3, 39] and therefore lend themselves naturally to array programming optimizations.

Acknowledgements

Peter Sestoft originally suggested systematic literature review as a method and helped reviewing the literature on spreadsheets. He also gave valuable feedback on drafts of this paper. Paolo Tell explained and discussed with us systematic literature review methodologies and related literature. David Christiansen helped with a lot of know-how on Emacs-Lisp during the development of SLIRM (see Section B.2).

Thanks!

References

[1] Robin Abraham and Martin Erwig. Type Inference for Spreadsheets. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '06, pages 73–84, New York, NY, USA, 2006. ACM. ISBN 1-59593-388-3. doi: 10.1145/1140335.1140346. URL `http://doi.acm.org/10.1145/1140335.1140346`.

[2] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data Structures for Parallel Computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, oct 1989. ISSN 0164-0925. doi: 10.1145/69558.69562. URL `http://doi.acm.org/10.1145/69558.69562`.

[3] Lee Benfield. FMD: Functional Development in Excel. In *Proceedings of the 2009 Video Workshop on Commercial Users of Functional Programming: Functional Programming As a Means, Not an End*, CUFP '09, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-943-5. doi: 10.1145/1668113.1668121. URL `http://doi.acm.org/10.1145/1668113.1668121`.

[4] Lars Bergstrom, Mike Rainey, John Reppy, Adam Shaw, and Matthew Fluet. Lazy Tree Splitting. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 93–104, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863558. URL `http://doi.acm.org/10.1145/1863543.1863558`.

[5] Robert Bernecky and Sven-Bodo Scholz. Abstract Expressionism for Parallel Performance. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2015, pages 54–59, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3584-3. doi: 10.1145/2774959.2774962. URL `http://doi.acm.org/10.1145/2774959.2774962`.

[6] Guy E. Blelloch. NESL: A Nested Data-Parallel Language (Version 2.6). Technical report, Pittsburgh, PA, USA, 1993. URL `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/papers/CMU-CS-95-170.html`.

[7] Guy E. Blelloch and John Greiner. A Provable Time and Space Efficient Implementation of NESL. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, ICFP '96, pages 213–225, New York, NY, USA, 1996. ACM. ISBN 0-89791-770-7. doi: 10.1145/232627.232650. URL `http://doi.acm.org/10.1145/232627.232650`.

[8] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. Implementation of a Portable Nested Data-parallel Language. *SIGPLAN Not.*, 28(7):102–111, July 1993. ISSN 0362-1340. doi: 10.1145/155332.155343. URL `http://dx.doi.org/10.1145/155332.155343`.

[9] Manuel M. T. Chakravarty and Gabriele Keller. Functional Array Fusion. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 205–216, New York, NY, USA, 2001. ACM. ISBN 1-58113-415-0. doi: 10.1145/507635.507661. URL `http://doi.acm.org/10.1145/507635.507661`.

[10] B. Chamberlain, S. Choi, E. Lewis, C. Lin, L. Snyder, and W. Weathersby. ZPL's WYSIWYG Performance Model. In *Proceedings of the High-Level Parallel Programming Models and Supportive Environments*, HIPS '98, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8412-7. URL `http://portal.acm.org/citation.cfm?id=822265`.

[11] Bradford L. Chamberlain, E. Christopher Lewis, Calvin Lin, and Lawrence Snyder. Regions: An Abstraction for Expressing Array Computation. In *Proceedings of the Conference on APL '99 : On Track to the 21st Century: On Track to the 21st Century*, APL '99, pages 41–49, New York, NY, USA, 1999. ACM. ISBN 1-58113-126-7. doi: 10.1145/312627.312713. URL `http://dx.doi.org/10.1145/312627.312713`.

[12] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28, New York, NY, USA, 2005. ACM. ISBN 1-58113-986-1. doi: 10.1145/1073970.1073974. URL `http://dx.doi.org/10.1145/1073970.1073974`.

[13] Tie Cheng and Xavier Rival. An Abstract Domain to Infer Types over Zones in Spreadsheets. In Antoine Miné and David Schmidt, editors, *Static Analysis*, volume 7460 of *Lecture Notes in Computer Science*, pages 94–110. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-33125-1_9. URL `http://www.di.ens.fr/~chengtie/pubs/paper_48.pdf`.

[14] Tie Cheng and Xavier Rival. Static Analysis of Spreadsheet Applications for Type-Unsafe Operations Detection. In Jan Vitek, editor, *Programming Languages and Systems*, volume 9032 of *Lecture Notes in Computer Science*, pages 26–52. Springer Berlin Heidelberg, 2015. doi: 10.1007/978-3-662-46669-8_2. URL `http://www.di.ens.fr/~chengtie/pubs/esop.pdf`.

[15] Wai-Mee Ching. Automatic Parallelization of APL-style Programs. In *Conference Proceedings on APL 90: For the Future*, APL '90, pages 76–80, New York, NY, USA, 1990. ACM. ISBN 0-89791-371-X. doi: 10.1145/97808.97826. URL `http://doi.acm.org/10.1145/97808.97826`.

[16] Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. NOVA: A Functional Language for Data Parallelism. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 8:8–8:13, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2937-8. doi: 10.1145/2627373.2627375. URL `http://doi.acm.org/10.1145/2627373.2627375`.

[17] Steven J. Deitz, Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. The Design and Implementation of a Parallel Array Operator for the Arbitrary Remapping of Data. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 155–166, New York, NY, USA, 2003. ACM. ISBN 1-58113-588-2. doi: 10.1145/781498.781526. URL `http://doi.acm.org/10.1145/781498.781526`.

[18] Matthew Fluet, Mike Rainey, and John Reppy. A Scheduling Framework for General-purpose Parallel Languages. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 241–252, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7. doi: 10.1145/1411204.1411239. URL `http://doi.acm.org/10.1145/1411204.1411239`.

[19] C. Grelck, S.-B. Scholz, and A. Shafarenko. Coordinating Data Parallel SAC Programs with S-Net. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, March 2007. doi: 10.1109/IPDPS.2007.370408.

[20] Clemens Grelck and Sven-Bodo Scholz. SAC: Off-the-shelf Support for Data-parallelism on Multicores. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 25–33, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-690-5. doi: 10.1145/1248648.1248654. URL `http://doi.acm.org/10.1145/1248648.1248654`.

[21] Cordelia V. Hall. Using Hindley-Milner Type Inference to Optimise List Representation. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, LFP '94, pages 162–172, New York, NY, USA, 1994. ACM. ISBN 0-89791-643-3. doi: 10.1145/182409.156781. URL `http://doi.acm.org/10.1145/182409.156781`.

[22] Troels Henriksen and Cosmin E. Oancea. Bounds Checking: An Instance of Hybrid Analysis. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 88:88–88:94, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2937-8. doi: 10.1145/2627373.2627388. URL `http://doi.acm.org/10.1145/2627373.2627388`.

[23] Troels Henriksen and Cosmin Eugen Oancea. A T2 Graph-reduction Approach to Fusion. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 47–58, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2381-9. doi: 10.1145/2502323.2502328. URL `http://doi.acm.org/10.1145/2502323.2502328`.

[24] Gwan-Hwan Hwang, Jenq Kuen Lee, and Dz-Ching Ju. An Array Operation Synthesis Scheme to Optimize Fortran 90 Programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 112–122, New York, NY, USA, 1995. ACM. ISBN 0-89791-700-6. doi: 10.1145/209936.209949. URL `http://doi.acm.org/10.1145/209936.209949`.

[25] Tomás Isakowitz, Shimon Schocken, and Henry C. Lucas, Jr. Toward a Logical/Physical Theory of Spreadsheet Modeling. *ACM Trans. Inf. Syst.*, 13(1):1–37, jan 1995. ISSN 1046-8188. doi: 10.1145/195705.195708. URL `http://doi.acm.org/10.1145/195705.195708`.

[26] Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, 1962. ISBN 0471430145. URL `http://www.worldcat.org/isbn/0471430145`.

[27] Simon P. Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell, 2008. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.141.1748`.

[28] Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. A User-centred Approach to Functions in Excel. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 165–176, New York, NY, USA, 2003. ACM. ISBN 1-58113-756-7. doi: 10.1145/944705.944721. URL `http://doi.acm.org/10.1145/944705.944721`.

[29] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, Shape-polymorphic, Parallel Arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 261–272, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863582. URL `http://doi.acm.org/10.1145/1863543.1863582`.

[30] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Ben Lippmeier, and Simon Peyton Jones. Vectorisation Avoidance. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 37–48, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1574-6. doi: 10.1145/2364506.2364512. URL `http://doi.acm.org/10.1145/2364506.2364512`.

[31] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. In *Technical report, Ver. 2.3 EBSE Technical Report. EBSE*. 2007. URL `http://www.cse.chalmers.se/~feldt/advice/kitchenham_2007_systematic_reviews_report_updated.pdf`.

[32] Kathleen Knobe and Vivek Sarkar. Array SSA Form and Its Use in Parallelization. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 107–120, New York, NY, USA, 1998. ACM. ISBN 0-89791-979-3. doi: 10.1145/268946.268956. URL `http://doi.acm.org/10.1145/268946.268956`.

[33] Ben Lippmeier, Manuel Chakravarty, Gabriele Keller, and Simon Peyton Jones. Guiding Parallel Array Fusion with Indexed Types. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 25–36, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1574-6. doi: 10.1145/2364506.2364511. URL `http://doi.acm.org/10.1145/2364506.2364511`.

[34] Ben Lippmeier, Manuel M.T. Chakravarty, Gabriele Keller, Roman Leshchinskiy, and Simon Peyton Jones. Work Efficient Higher-order Vectorisation. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 259–270, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1054-3. doi: 10.1145/2364527.2364564. URL `http://doi.acm.org/10.1145/2364527.2364564`.

[35] P. Geoffrey Lowney. Carrier Arrays: An Idiom-preserving Extension to APL. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, pages 1–13, New York, NY, USA, 1981. ACM. ISBN 0-89791-029-X. doi: 10.1145/567532.567533. URL `http://doi.acm.org/10.1145/567532.567533`.

[36] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array-data Flow Analysis and Its Use in Array Privatization. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 2–15, New York, NY, USA, 1993. ACM. ISBN 0-89791-560-7. doi: 10.1145/158511.158515. URL `http://doi.acm.org/10.1145/158511.158515`.

[37] J. Mendes. Classsheet-driven spreadsheet environments. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 235–236, Sept 2011. doi: 10.1109/VLHCC.2011.6070409.

[38] Gary Miller. The Spreadsheet Paradigm: A Basis for Powerful and Accessible Programming. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH Companion 2015, pages 33–35, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3722-9. doi: 10.1145/2814189.2814201. URL `http://doi.acm.org/10.1145/2814189.2814201`.

[39] R. Mittermeir and M. Clermont. Finding high-level structures in spreadsheet programs. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 221–232, 2002. doi: 10.1109/WCRE.2002.1173080.

[40] J.T. O'Donnell. MPP implementation of abstract data parallel architectures for declarative programming languages. In *Frontiers of Massively Parallel Computation, 1988. Proceedings., 2nd Symposium on the Frontiers of*, pages 629–636, Oct 1988. doi: 10.1109/FMPC.1988.47507.

[41] R. H. Perrott. A Language for Array and Vector Processors. *ACM Trans. Program. Lang. Syst.*, 1(2): 177–195, oct 1979. ISSN 0164-0925. doi: 10.1145/357073.357075. URL http://doi.acm.org/10.1145/357073.357075.

[42] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic mapping studies in software engineering. In *12th International Conference on Evaluation and Assessment in Software Engineering*, volume 17. sn, 2008. URL http://www.rbsv.eu/courses/rmtw/mtrl/SM.pdf.

[43] A. V. S. Sastry and William Clinger. Parallel Destructive Updating in Strict Functional Languages. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, LFP '94, pages 263–272, New York, NY, USA, 1994. ACM. ISBN 0-89791-643-3. doi: 10.1145/182409.182486. URL http://doi.acm.org/10.1145/182409.182486.

[44] Peter Sestoft. A Spreadsheet Core Implementation in C#. Technical report, IT University of Copenhagen, September 2006. URL http://www.itu.dk/people/sestoft/funcalc/ITU-TR-2006-91.pdf.

[45] Peter Sestoft. Implementing Function Spreadsheets. In *Proceedings of the 4th International Workshop on End-user Software Engineering*, WEUSE '08, pages 91–94, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-034-0. doi: 10.1145/1370847.1370867. URL http://doi.acm.org/10.1145/1370847.1370867.

[46] Peter Sestoft. *Spreadsheet Implementation Technology: Basics and Extensions*. The MIT Press, September 2014. ISBN 0262526646. URL http://ieeexplore.ieee.org/xpl/bkabstractplus.jsp?bkn=6940404.

[47] A.V. Shafarenko. RETRAN: a recurrent paradigm for massively parallel array computing. In *Massively Parallel Computing Systems, 1994., Proceedings of the First International Conference on*, pages 478–487, May 1994. doi: 10.1109/MPCS.1994.367042.

[48] Rishabh Singh and Sumit Gulwani. Transforming Spreadsheet Data Types Using Examples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 343–356, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837668. URL http://doi.acm.org/10.1145/2837614.2837668.

[49] Artjoms Sinkarovs and Sven-Bodo Scholz. Semantics-preserving Data Layout Transformations for Improved Vectorisation. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 59–70, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2381-9. doi: 10.1145/2502323.2502332. URL http://doi.acm.org/10.1145/2502323.2502332.

[50] Lawrence Snyder. The Design and Development of ZPL. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-7. doi: 10.1145/1238844.1238852. URL http://dx.doi.org/10.1145/1238844.1238852.

[51] Marc Stadelmann. A spreadsheet based on constraints. In *Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology*, UIST '93, pages 217–224, New York, NY, USA, 1993. ACM. ISBN 0-89791-628-X. doi: 10.1145/168642.168664. URL http://doi.acm.org/10.1145/168642.168664.

[52] Nicolas Stucki, Tiark Rompf, Vlad Ureche, and Phil Bagwell. RRB Vector: A Practical General Purpose Immutable Sequence. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 342–354, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3669-7. doi: 10.1145/2784731.2784739. URL http://doi.acm.org/10.1145/2784731.2784739.

[53] Peiyi Tang, Pen-Chung Yew, and Chuan-Qi Zhu. Compiler Techniques for Data Synchronization in Nested Parallel Loops. In *Proceedings of the 4th International Conference on Supercomputing*, ICS '90, pages 177–186, New York, NY, USA, 1990. ACM. ISBN 0-89791-369-8. doi: 10.1145/77726.255155. URL http://doi.acm.org/10.1145/77726.255155.

[54] Jim Trudeau. Collaboration and Open Source at AMD: LibreOffice, July 2015. URL http://developer.amd.com/community/blog/2015/07/15/collaboration-and-open-source-at-amd-libreoffice/. Accessed on 31.07.2015.

[55] Vlad Ureche, Tiark Rompf, Arvind Sujeeth, Hassan Chafi, and Martin Odersky. StagedSAC: A Case Study in Performance-oriented DSL Development. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*, PEPM '12, pages 73–82, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1118-2. doi: 10.1145/2103746.2103762. URL `http://doi.acm.org/10.1145/2103746.2103762`.

[56] Clifford Walinsky and Deb Banerjee. A Functional Programming Language Compiler for Massively Parallel Computers. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 131–138, New York, NY, USA, 1990. ACM. ISBN 0-89791-368-X. doi: 10.1145/91556.91610. URL `http://doi.acm.org/10.1145/91556.91610`.

A Protocol

A.1 Background

The goal of this systematic mapping study is to coalesce two apparently disjoint research areas, namely declarative parallel array programming techniques and end-user development in a spreadsheet model of computation. We base our methodology on Kitchenham and Charters [31] and Petersen et al. [42], see also Section 1.1.

This study is meant to motivate and inform further research on declarative parallel programming in spreadsheets for end-users. We want to gain an overview over well known techniques for transforming expressions that describe seemingly sequential operations into efficient parallel code. Furthermore, we want to gain insight into how this can be combined with spreadsheet end-user development techniques and whether there has been made any scientific effort into this direction already.

We purposely do not focus on data-flow parallelism in spreadsheets. This is a separate research project orthogonal to declarative parallel array programming in spreadsheets.

To answer the research questions raised in Section 1.2, we run three different search queries:

Declarative, parallel array programming languages What are promising techniques for automatic parallelization of declarative, functional languages?

Spreadsheet end-user development Which paradigms have been developed, what do users use spreadsheets for?

Parallelism in spreadsheets To what degree have parallel spreadsheet engines been investigated? What are challenges and possibilities? Where does our agenda fit in?

A.2 Study Selection Criteria

We perform two disjoint literature studies, one for declarative parallel array programming languages and one for spreadsheet end-user development.

In the following, we give a list of criteria for inclusion or exclusion of studies. Naturally, some of the studies can fulfill criteria of both lists. We choose therefore to perform a majority vote on the number of fulfilled criteria when making a decision of inclusion or exclusion. The criteria lists can be seen as disjunctions of the single criteria.

A.2.1 Declarative Parallel Array Programming

We include a publication if it:

- Mentions implementations of prominent array languages.
- Focuses on automatic parallelization of array expressions.
- Mentions caching and false sharing.
- Talks about compiler optimizations.
- Focuses on implementation of declarative, functional parallel programming techniques.
- Mentions homogeneous systems and shared memory.
- Mentions loop fusion and nested loops.
- Talks generally about program transformation.

We exclude a publication if it:

- Focuses on the application of parallel programming, e.g. within machine learning.
- Develops techniques for distributed memory or mentions message passing.
- Develops techniques for focuses on formal verification.
- Targets GPU, GPGPU, FPGA and hardware accelerated techniques on heterogeneous systems.
- Includes I/O.
- Works towards automatic parallelization of imperative languages.
- Is a review paper without any novel contribution.
- Mentions storage, disk etc.
- Focuses on transactional memory.

A.2.2 Spreadsheet End-User Development and Parallel Spreadsheets

We include a publication if it:

- Mentions functional programming or functional language.
- Describes the implementation of a spreadsheet engine.
- Focuses on gaining or providing spreadsheet understanding.
- Mentions types or type-inference.

We exclude a publication if it:

- Focuses on the application of spreadsheets and specific spreadsheet models, such as simulations or in teaching.
- Describes systems inspired by spreadsheets.
- Mentions CSCW and knowledge work or performs ethnographic studies.
- Focuses on data structures in spreadsheets.
- Mentions mashups, mobile apps or web development.
- Focuses on external tools and architectures for spreadsheet users.
- Develops techniques for transforming spreadsheets.
- Surveys "real-life" spreadsheets.

A.3 Quality Assurance

We use two-fold quality assurance. First, we define a list of authors and publication topics that must be part of the literature list obtained by our search query. This is a relaxed version of the quality assurance suggested by Kitchenham and Charters [31].

A.3.1 Declarative Array Programming

We require the following authors and publications in the literature list:

- Guy Blelloch on NESL
- Simon Peyton-Jones on Data-Parallel Haskell
- REPA

A.3.2 Spreadsheet End-User Development and Parallel Spreadsheets

We require the following authors and publications in the literature list:

- Peter Sestoft on Funcalc and user-defined functions
- Margaret Burnett and Simon Peyton-Jones on user-defined functions in spreadsheets

Furthermore, Peter Sestoft, an expert on research on spreadsheet end-user programming, double-checked the literature list for relevance.

B Process

B.1 Literature Search

We have use IEEEExplore and ACM Digital Library as sources for our literature search. The number of results from these sources varies drastically, which is probably due to IEEEExplore interpreting search queries very strictly. We avoid Google Scholar and CiteSeerX, as these meta engines return way over a thousand publications for each query, which is infeasible for our scope.

The uniqueness of a publication in the following means that a publication is listed only once. Sometimes, publications have multiple entries in a digital library, for instance one for a conference's proceedings and one for SIGPLAN Notes.

We have an initial list of 681 publications to consider, which we construct as described in the following:

B.1.1 Declarative Parallel Array Programming

To generate a literature list for declarative parallel array programming languages, we use the following query:

```
(functional AND array AND programming AND parallel) AND (data-parallel OR
    ``data parallel'' OR multi-core OR multicore  OR ``multi core'')
```

This results in 250 publications of which 194 are unique from the ACM Digital Library. IEEExplore, however, only returns eight publications, all of which are unique. This adds 202 publications to consider.

B.1.2 Spreadsheet End-User Development

The search query we use for finding publications on spreadsheet end-user development is:

```
spreadsheets AND (end-user-development OR "end user development" OR
    "end-user development")
```

This results in 435 publications of which 386 are unique from the ACM Digital Library and in 104 publications, all of which are unique, from IEEExplore. This adds 424 publications to consider.

B.1.3 Parallel Spreadsheets

We use the following query to generate a list of publications focusing on anything parallel in spreadsheets:

```
spreadsheets AND parallel
```

ACM returns 21 publications of which 17 are unique and IEEExplore returns 38 publications. This adds 55 publications to consider.

B.2 Literature Selection

Using the criteria defined in Section A.2, we include studies based on their titles and their abstracts. If the title is not informative enough, we accept the study and will later screen it again based on the abstract.

To perform the selection of literature move conveniently, we have developed an Emacs[1]-based tool, called the Systematic Literature Review Mode (SLIRM). SLIRM automatically downloads abstracts and full-text files on demand for BibTeX-formatted entries that have been exported from a digital library, such as the ACM Digital Library. SLIRM is open-source and freely available[2].

[1]https://www.gnu.org/software/emacs/
[2]Download SLIRM from https://github.com/fbie/slirm