# A Parallel Functional Programming

## Introduction

At the beginning of the millenium, microprocessor designs lay on a number of exponential curves, with exponentially increasing transistor counts, exponentially rising clock frequencies, exponentially improving performance—and exponentially rising power consumption. Intel's CTO Patrick Gelsinger remarked "By mid-decade, that Pentium PC may need the power of a nuclear reactor... Soon after 2010, PC chips could feel like the bubbly hot surface of the sun itself." Of course, this never happened—while transistor counts have continued to follow Moore's law, clock frequencies have stagnated, with the result that power consumption per chip has fallen slightly from its peak. Instead of delivering higher performance through faster clocks and microarchitectural improvements, processor manufacturers are delivering it through *multiple processor cores*. We can expect to see Moore's law translate into an exponentially growing *number* of cores, for at least the next decade—already Intel and AMD are delivering chips with 10–16 x86 cores, and other manufacturers are supplying chips with even more. Plans for exaflop super-computers (performing $10^{18}$ floating-point operations per second) call for over a million cores working together—and for further dramatic improvements in power consumption per core, since otherwise an exaflop computer will consume an entirely unrealistic gigawatt of power.

In constrast to previous hardware performance improvements, which benefited virtually all programs, multi- and many-core chips can only be exploited by *parallel programs*. Thus, parallel programming has acquired dramatically increased significance; having once been the purview only of a few specialists, it is now a fundamental skill which is essential even to take full advantage of the hardware in our laptops. Unfortunately, parallel programming has traditionally been regarded as difficult—beyond the capabilities of most software developers. Finding *easy* ways to construct parallel software has thus become one of our most urgent research tasks.

Many of the difficulties of parallel programming arise from mutable shared data, the race conditions that result when it is not properly protected, and the concurrency bugs and bottlenecks that result when it is. One way to avoid these difficulties is to use *functional programming*, in which most data-structures are not mutable at all. Parallel functional programming is attracting increasing attention from industry as a result. To quote Intel's CTO, Justin Rattner, "Functional programming looks to be one of the foundations for parallel programming going forward with higher levels of abstraction and more automation of parallelism". Functional programming ideas underlie, among others, Google's Map-Reduce, Intel's Concurrent Collections, Ericsson's Erlang and Scala's Akka concurrency framework, which delivers scalable parallel performance for Twitter's back-end processing.

Parallel functional programming is an active research area (see the next section), but nevertheless much more research is needed to make it into a widely applicable technique. We believe it behooves functional programming researchers to *focus on parallelism* in the coming years, and our goal in this proposal is to retarget our own research in precisely this direction.

Our goal is *not* to develop programming methods for the kind of small-scale parallel systems already found in all but the smallest computers. We are specifically aiming some years ahead, at machines with at least tens, more likely hundreds of cores. We expect such systems to be *heterogenous*, with a mixture of general purpose cores and specialised, data-parallel cores, and a mixture of shared and distributed memories. Already, virtually every laptop or desktop contains both multiple CPU cores, and a GPU. Today's super-computers are similar: for example, Los Alamos Lab's petaflop RoadRunner combines 12,000 x86 cores with 100,000 simple cell processors for data parallelism. We aim to develop functional programming methods to make use of such diverse architectures: a good example application would be a Barnes-Hut $N$-body simulation, running on a cluster of multicore computing nodes, and using a GPU at each node to accelerate the computation.

To a large extent we intend to focus on *deterministic parallelism*, as in this example—the result of a gravitational simulation should, of course, depend deterministically on its inputs. Indeed, part of the strength of parallel functional programming is that it avoids the unwanted non-determinism that so easily arises in other paradigms. However, we also plan to consider some parallel applications that are inherently *non*-deterministic—for example, a database in the cloud which must react to requests from many concurrent sources, where the results will obviously depend on the order in which requests are served.

The larger the number of processors, the greater the risk that at least one of them fails during a long computation—and indeed, as voltages are further reduced to save power, then individual processors are likely to become *less*, not more, reliable. Therefore, a convincing approach to highly parallel computation must provide a degree of *fault tolerance*, so that computations can recover from individual errors without restarting from scratch.

**Parallel Functional Programming** Research in parallel functional programming has a long history. In the 1980s, here at Chalmers, Johnsson and Augustsson developed the $\nu$G-machine with encouraging results [1]. They compiled Lazy ML for a shared memory multiprocessor, and benchmarked against the sequential LML compiler, at that time the best compiler for a lazy functional language, and observed speed-ups of up to 3.3x on four processors. They used *sparks* to start parallel computations, which were ignored, and thus cheap, unless a processor was actually available. The style of programming that this encourages—in which the programmer indicates *possible* tasks for parallelisation—is today typically called *semi-explicit*, forming an attractive middle ground between fully implicit parallelism (which is not promising in general [12]) and explicit threading (which overburdens the programmer).

In the late 1980s and early 1990s, Blelloch developed NESL, a *nested data parallel* programming language, and demonstrated its utility, both for expressing complex algorithms sweetly [4] and for implementing them efficiently on the parallel machines of the day, such as the Connection Machine. NESL's performance was competitive with machine-specific code for regular dense data, and often superior for irregular data. Nested data parallelism allows the easy expression of less regular computations, and so encourages a style of parallel programming in which the user expresses algorithms (and in particular recursive divide

and conquer algorithms) in a functional programming style that is familiar from sequential programming. NESL in turn influenced languages like Cilk, now an important part of Intel's push towards multicore programming. Yet NESL itself has not developed further in the last 15 years; it was well-suited to wide-vector parallel machines, but today's multicores are very different. It was a minimal programming language, lacking many features of modern functional languages (such as user-defined types and higher-order functions).

At Chalmers, we have introduced a Parallel Functional Programming course for Masters students [22]. The development of the course has entailed a thorough study of the current state of the art, and also much interaction with leading researchers in the community. The first part of the course concerns deterministic parallel programming in Haskell. Here, there is a clear sequence of research results that bring useful tools for practical parallel programming to ordinary Haskell programmers. The `par` combinator allows the user to indicate sparks, that is to indicate potential parallelism; `pseq` allows the user to control order of evaluation, which is necessary in the presence of lazy evaluation. Parallel strategies build upon this idea, allowing an elegant separation of algorithmic code from code indicating how parallelisation should happen [24]. Overall, the semi-explicit approach to parallelising Haskell programs has proven very effective [16]. The semantics of the program remains completely deterministic, and the programmer is not required to identify threads, communication, or synchronisation. She simply annotates subcomputations that might be evaluated in parallel, leaving the choice of whether to actually do so to the runtime system. These *sparks* are created and scheduled dynamically, and their grain size varies widely. A recent update of the strategies idea improves the API, broadens its applicability and fixes a space leak in the older approach [17].

The sequence has continued with the introduction of the `Par` monad, in which the user creates data flow graphs to indicate the shape of the computation, and the runtime system may choose to evaluate parts of the graph in parallel [18]. This seems to find a sweet spot in which the programmer can concentrate on algorithm design and get just enough control over the resulting parallel computation.

NESL has strongly influenced developments in parallel programming in Haskell as well. It is the inspiration for work on Data Parallel Haskell (DPH) [5], which provides a parallel array data type. Also based on the work on DPH, the Repa array library provides flat data parallelism [14].

**Domain Specific Languages** Embedded Domain Specific Languages (DSLs) and libraries are a recurring theme of the above review. This topic has been an important focus in our research group. With Ericsson funding, we have developed Feldspar, an embedded language for DSP algorithm design, generating C code [2]. In doing this, we have built a generic DSL toolkit called `Syntactic`, which can represent a wide range of typed languages, including Feldspar [3]. As a result it is now very much easier for us to experiment with new (purely functional) DSLs. In a further Ericsson funded project, Feldspar itself is currently being evaluated in the development of a parallel implementation of part of the uplink of the LTE telecoms standard on a Tilera multicore machine.

Obsidian, developed in our group is an embedded domain specific language that generates CUDA kernels from functional descriptions [23]. A symbolic array construction allows us to guarantee that intermediate arrays are fused away. Claessen has proposed a new form of symbolic arrays called Push arrays that remove some restrictions imposed by the original arrays. This extension to Obsidian has demonstrated on a sequence of sorting kernels, with good results [9]. We have only just begun to explore the possibilities of these more sophisticated symbolic array representations, and we view this as a particularly promising line of research in data parallel programming.

**Concurrency in Haskell** Although our inclination is to remain with deterministic parallel programming, we recognise that we will need concurrency when dealing with reactive systems, and with distribution in heterogeneous systems. In keeping with its strategy of allowing users to mix and match many approaches to parallelism and concurrency, Haskell provides explicitly threaded concurrent programming [20]. An interesting recent development is Cloud Haskell, which provides Erlang style message passing communication as a shallow embedded DSL [11]. We are also interested in exploring Erlang style concurrency in Haskell, but we would like to avoid non-determinism to a greater degree than is proposed in Cloud Haskell.

**Why more needs to be done** The above summary of the state of the art indicates the promise of purely functional programming for parallelism and concurrency. There are many promising first steps here, particularly in the Haskell community. But there is fragmentation too, and much remains to be done! The existing libraries, discussed above, provide the means to express and control parallelism, but often at too low a level of abstraction. We surely need to provide users with better abstractions, built upon the existing work.

The long (and continuing) gestation of Data Parallel Haskell points to the difficulty of taking Blelloch's flattening transformation (which converts nested into flat data parallelism automatically) and making it work in the setting of a complex, modern functional programming language. Peyton Jones' analogy in a talk at the DAMP'12 workshop was apt: he said that it was as though Blelloch had left a $10,000 note on the ground, just waiting to be picked up, only for those who came after to find that it was firmly stuck to the ground! We are interested in answering the question: "Is nested data parallelism really necessary, or can we get away with flat data parallelism and sophisticated array representations?".

The International Workshop on Declarative Aspects and Applications of Multicore Programming, associated with the top conference POPL in 2012, had a total of only 10 submissions, in a field that we think is extremely important, both academically and industrially. Of course there are other places to publish work in this area, and there is much going on in mainstream programming for parallelism, but still, we feel that the pressing nature of the problem of how to program future massively parallel architectures is being ignored by too many researchers. Our plan is to focus the work of our group on exactly this problem.

**Project description**

Our overall goal is to develop methods and libraries to enable Haskell programmers to solve computationally intensive parallel programming problems easily and efficiently on heterogenous computing systems, namely clusters of computing nodes, each with a multi-core CPU, disk, and GPGPU. Typical problems include $N$-body gravitational simulations, first-order logic theorem proving, data mining, and large scale key-value storage. We aim to exploit Haskell's power of abstraction to capture reusable ideas in easy-to-deploy libraries and domain specific languages.

**Cache Sensitive Functional Programming.** The purpose of parallelism is performance; therefore it is important to ensure that *single-processor* performance is good. On today's architectures, making effective use of the *cache* is critical in high-performance code. While data can be retrieved from the L1 cache in only a few processor cycles, retrieving data from external RAM can take closer to 100 cycles. This has led to a discipline of "cache sensitive programming", in which data-structures and access patterns are designed to minimize the number of cache misses. Techniques used include factoring data-structures into "hot" and "cold" parts, so that the frequently-referenced ("hot") components are gathered together into small structures likely to fit into one cache line; prefetching data so it is available in the cache by the time it is needed; cache-aware algorithms that use intermediate data carefully designed to fit into the cache at a particular level of the hierarchy; cache-oblivious algorithms that aim to access memory in an effective way *regardless* of the size of the cache—thus performing well at all levels of the memory hierarchy. Divide-and-conquer algorithms are an important technique for improving cache behaviour: for example, a matrix transpose implemented as a pair of nested loops will encounter a cache miss for each matrix element, while a divide-and-conquer cache-oblivious version can reduce this by a factor proportional to the size of a cache line, by transposing submatrices at a time. Blelloch has studied ways to develop divide and conquer algorithms with provably good cache performance [6].

It is clear that some of the most popular data-structures in functional programs, such as linked lists, do not play particularly well with the cache. Moreover, common higher-order functions such as `map`, `filter` and `foldr`, do not traverse data in a cache-efficient manner. Our goal in this part of the project is to investigate more cache-efficient alternatives. We intend to implement a variety of cache-sensitive algorithms from the literature in Haskell, measure their performance, and then extract reusable ideas into a library of cache-efficient data-structures and higher-order functions, thus making efficient Haskell programs easier to develop in the future.

**Programming a cluster.** Many parallel applications today—for example, no-SQL databases such as Riak—run on a cluster of commodity processing nodes connected by a fast network. Today's supercomputers are built in a similar manner (albeit from higher-performance components). A cluster is also a good model of future highly parallel computers, because as the

number of cores rises it will no longer be practical to support the shared memory abstraction between more than a small set of cores. Thus, we plan to develop methods to distribute a parallel functional computation across a cluster of distributed memory machines.

Erlang, with its share-no-memory model, is already well suited to this kind of problem; Haskell's recently developed "Cloud Haskell" extension borrows heavily from Erlang to address the same problems [11]. However, these offer only low-level mechanisms such as the ability to monitor a process running on another node; orchestrating the nodes to work together to solve a common problem is still left to the Erlang or Cloud Haskell programmer.

Google's Map-Reduce framework is designed to run on hardware of this sort, and in a sense offers a higher-level approach to the problem; the programmer instantiates the framework with mapping functions and reducing functions, and the framework takes care of orchestrating the execution, monitoring for faults, and so on. As a starting point, we plan to investigate Map-Reduce implementations in Cloud Haskell, and later to exploit Haskell's power of abstraction to define other generic patterns of computation.

**Algebra of parallel programs.** Algebraic laws offer a powerful mechanism to develop and optimise functional programs; in particular the elegant Bird/Meertens approach to algorithm synthesis is based on starting from simple, obviously correct algorithms, and applying a succession of algebraic transformations (each of which can be tested with QuickCheck) to obtain the final, efficient result. Associative operators play an important rôle in the Bird/Meertens method, especially in their "theory of lists". Associative operators are also at the heart of parallel algorithms such as reduce and scan. While Bird/Meertens focussed on *sequential* algorithms, we plan to develop a similar approach to *parallel* algorithms.

We are already making interesting discoveries in the area of parallel *parsing*, using an associative operator which combines partial parses of substrings in an associative way. While this operator is prohibitively expensive in the worst case, we have discovered that—for inputs and grammars in the right form—this worst case never arises in practice. Thus, by expressing the grammar appropriately—among other things, using a built-in repetition operator rather than representing repetition via recursion—it seems that we can develop a highly efficient parallel parsing method. Moreover, this method is naturally *incremental*, because a local change to the input need only result in reparsing the local context and then recombining its parse result with the rest of the input via our associative operator. The method can thus naturally be applied in editors which parse code in order to highlight syntax, indicate parse errors, and so on.

**Property-driven development.** Parallel functional programs need testing as much as any other kind of program; our approach is based on *property-based testing* using our testing tool QuickCheck [7, 10]. QuickCheck defines a domain-specific property language, embedded in Haskell or Erlang, whose properties are tested in (randomly) generated test cases. When a property fails, the failing case is reduced to a minimal failing example via "shrinking", which usually results in a failure that is easy to debug.

While property-based testing can be applied effectively to either new or existing code, we are particularly interested in *property driven development* (by analogy with *test driven development*), in which properties and code are developed *together*, and each reveals weaknesses in the other. We intend to develop a PDD methodology for parallel functional programs. One important question is *what properties* should hold of non-deterministic, parallel systems? We have enjoyed considerable success using *serializability* as a property-to-test [8]; last year this property enabled us to track down long-standing race conditions in the database delivered with Erlang, fixing bugs which caused crashes in production every week or two at large industrial users [13].

Yet it is not clear that the best way to test for correct parallel systems is by provoking actual race conditions in the real system itself—real races may be rare and hard to provoke, while their effects may be much easier to model. For example, last year we helped to find eventual consistency bugs in Riak, a well-known no-SQL key-value database implemented in Erlang. The bugs are manifested by a particular (unlikely) sequence of node failures and restarts, but testing for them by taking nodes up and down during each test would be unlikely to reveal them, because such tests are so slow that only a few can be run. Instead, we extracted the purely functional logic from the database, and tested it in a simulated setting in which the node failures could be directly and easily modelled. This enabled us to run a large number of generated tests in a short time, and led to rapid discovery of the faults (a paper is in preparation).

In this case, we used an ad-hoc approach to solve a particular testing problem for a parallel functional system. In this subproject, we plan to develop systematic methods for developing such systems, together with property-based tests in simulated settings to ensure correct behaviour in the presence of non-determinism and node failures. We expect our *inductive testing* approach (the subject of Claessen's individual proposal), which draws on proof-by-induction to formulate appropriate properties to test, to be very useful here. Likewise, parallel programs that depend on algebraic properties can be developed in parallel with tests of those properties.

**DSLs for parallel Haskell programming** Although this project concerns programming in Haskell, it can still build upon our previous strong research in code-generating embedded DSLs. Building on the Haskell Foreign Function Interface, Persson (a doctoral student in our group) has developed a system where generated code from the Feldspar embedded language can be seamlessly compiled, linked and introduced into a running Haskell program, including the Haskell interpreter. The prototype uses Template Haskell to synthesize a harness automating (re-)compilation, linking and marshaling of arguments and results. Marshalling is enabled by the Feldspar size-propagation. This removes the need to rely on external tools and frameworks for testing the functionality of code generated from embedded DSLs, making tools like Haskell QuickCheck available for this purpose. However, it also opens interesting new possiblities. Code-generating embedded DSLs now become an immediately usable tool for *Haskell* programmers. We plan to generalise and extend the *plugin* tool to work with other embedded languages. We will explore the effects of storing a

richer set of decorations on the AST. This approach will permit users of parallel Haskell to use embedded DSLs to gain the finer control over memory use or parallelism. It is here we forge the link between our DSL expertise and our new ambitions to influence how parallel Haskell programming is done. Using DSLs in this way will provide a possible approach to heterogeneity – different DSLs for different platforms, used together in Haskell. It will also enable us to explore combinators for fault tolerance.

**Securing parallel systems.** Security and confidentiality are of increasing importance, yet particularly awkward to address in parallel systems—the standard information-flow languages Jif [19] and FlowCaml [21] ignore concurrency altogether. We propose a library that mitigates and eliminates termination and timing channels arising in concurrent systems, while allowing timing and termination of loops and recursion to depend on secret values. Because the significance of these covert channels depends on concurrency, *we fight fire with fire by leveraging concurrency to mitigate these channels*: we place potentially nonterminating actions, or actions whose timing may depend on secret values, in separate threads. Although we do not address leakage of information produced by the underlying hardware (e.g., exploiting cache timing behavior), our proposed solution can be combined with hardware-level mechanisms as needed to provide comprehensive defenses against such vulnerabilities. More specifically, we plan to extend the Haskell LIO security library [**?** ] (designed for a sequential language) as follows.

► *Termination covert channel*: we propose to decouple the execution of public events from computations that manipulate secret data. Using the primitives `forkLIO` and `waitLIO`, computation depending on secret data proceeds in a new thread. If a thread needs to observe the termination behavior of a newly spawned thread, it firstly enters a state where public events are no longer allowed, thus avoiding leaks due to termination.

► *Internal timing channel*: we close this covert channel using the same approach as termination leaks: we decouple the execution of public events from computations that manipulate secret data. As a result, the number of instructions executed before producing public events does not depend on secrets. A possible race to a shared public resource cannot be affected by secret data, which eliminates internal timing leaks.

► *External timing channel*: Our contribution to mitigate external timing channels in concurrent systems is to bring the mitigation techniques from the OS community into programming languages. Zhang et al. [25] describe a black-box mitigation technique in which the source of observable events is wrapped by a timing mitigator that delays output events so that they reveal only a bounded amount of information. Leveraging Haskell monad transformers [15], we propose to modularly extend LIO, or any other library performing side-effects in Haskell, to provide a suitable form of Zhang et al.'s mitigator.

**Applications driving the research** An important part of our research will be the implementation of a wide variety of parallel algorithms, in the search for programming idioms, and for the design of an abstraction layer above the exisiting libraries like the Par

monad. We will study algorithms from "Big Data", which is an important topic in the Chalmers Algorithms group, from finance (where we have a strong link to the Hiperfit Centre at Copenhagen University) and from Digital Signal Processing (via the group at our sister department, Signals and Systems). We have already had considerable success in parallelizing our first order theorem prover Equinox, achieving a speed-up of 2.5 on a 4-core machine. (One of the two alternating phases of the prover is very hard to parallelize, but cheap, while the other is expensive but easily parallelizable.) Parallelizing automated reasoning tools will provide us with demanding case studies.

**Group Description, Grants and Collaboration**

The Functional Programming group at Chalmers is one of the strongest in the field. It consists of four professors, one visiting professor, five assistant professors (forskarassistenter), one post-doc, and ten doctoral students. We are the originators of the award-winning QuickCheck testing tool, the foundation of a recently-completed top-rated EU FP7 project, ProTest. We focus strongly on domain-specific languages embedded in Haskell, including the influential Lava hardware design language. We hold a five-year strategic grant of 25 million SEK from the Strategic Research Foundation, in Resource Aware Functional Programming, targetting signal processing and automotive applications via domain specific languages that generate low-level target code. As well as Ericsson, we collaborate actively with Quviq (a start-up that arose from our group marketing a version of QuickCheck), Stanford, and with Microsoft Research in Cambridge.

**Significance**

Pure functional programming provides the best hope of solving the multicore programming challenge. Haskell is thus the obvious starting point for our work, as it has a rich set of libraries, a type system that tracks uses of mutation, very good sequential performance and existing support for a variety of approaches to concurrent and parallel programming. In this project, we will develop the necessary methods and tools to allow Haskell programmers to develop parallel programs that provide decent speed-ups on present and future heterogeneous machines *with ease*. This is, without doubt, a highly ambitious goal - broader in scope than our previous VR frame project. Frightened by the fact that a relatively small number of researchers in functional programming are working on enabling practical parallel programming, we have decided to focus the efforts of our group on this key challenge. Should we succeed, we will have a palpable effect on how programming is done in a wide variety of industries and application areas, from finance to big data. We relish the prospect of helping to solve a key question in computer science by applying our talent for finding simple elegant solutions to pressing practical problems.

# Bibliography


[1] L. Augustsson and T. Johnsson. Parallel graph reduction with the (v,g)-machine. In *FPCA '89: Proc. on Functional programming languages and computer architecture*, pages 202–213. ACM Press, 1989.

[2] E. Axelsson, K. Claessen, et al. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 169–178. IEEE, 2010.

[3] E. Axelsson and M. Sheeran. Feldspar: Application and implementation. In *Central European Functional Programming School, Revised Selected Lectures*, volume 7214 of *LNCS*. Springer, 2012.

[4] G. Blelloch. Programming Parallel Algorithms. *Communications of the ACM*, 39(3), 1996.

[5] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton-Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A status report. In *Proc. Int. Workshop on Declarative Aspects of Multicore Prog. (DAMP)*. ACM Press, 2007.

[6] R. A. Chowdhury, V. Ramachandran, G. E. Blelloch, P. B. Gibbons, S. Chen, and M. Kozuch. Provably Good Multicore Cache Performance for Divide-and-Conquer Algorithms. In *SIAM/ACM Symposium on Discrete Algorithms (SODA)*, 2008.

[7] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN notices*, 35(9):268–279, 2000.

[8] K. Claessen, M. Palka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. Wiger. Finding race conditions in Erlang with QuickCheck and PULSE. *ACM Sigplan Notices*, 44(9):149–160, 2009.

[9] K. Claessen, M. Sheeran, and B. Svensson. Expressive array constructs in an embedded GPU kernel programming language. In *DAMP*, pages 21–30. ACM, 2012.

[10] J. Derrick, N. Walkinshaw, T. Arts, C. Benac Earle, F. Cesarini, L. Fredlund, V. Gulias, J. Hughes, and S. Thompson. Property-based testing-the protest project. *Formal Methods for Components and Objects*, pages 250–271, 2010.

[11] J. Epstein, A. Black, and S. Peyton-Jones. Towards Haskell in the cloud. In *Haskell Symposium*. ACM, 2011.

[12] T. Harris and S. Singh. Feedback Directed Implicit Parallelism. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional Programming (ICFP)*. ACM Press, 2007.

[13] J. Hughes and H. Bolinder. Testing a database for race conditions with quickcheck. In *Erlang Workshop*. ACM SIGPLAN, 2011.

[14] G. Keller, M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton-Jones, and B. Lippmeier. Regular, Shape-polymorphic, Parallel Arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional Programming (ICFP)*. ACM Press, 2010.

[15] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *In Proceedings of the 22nd ACM Symposium on Principles of Programming Languages. ACMPress*, 1995.

[16] H.-W. Loidl, F. Rubio, et al. Comparing parallel functional languages: Programming and performance. *Higher Order Symbol. Comput.*, 16(3):203–251, 2003.

[17] S. Marlow, P. Maier, H.-W. Loidl, M. K. Aswad, and P. W. Trinder. Seq no more: Better Strategies for Parallel Haskell. In *Proceedings of the Haskell Symposium*. ACM Press, 2010.

[18] S. Marlow, R. Newton, and S. Peyton-Jones. A monad for deterministic parallelism. In *Proceedings of the Haskell Symposium*. ACM SIGPLAN, 2011.

[19] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, Jan. 1999.

[20] S. Peyton-Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. Int. Conf. on Principles of Programming Languages (POPL)*. ACM Press, 1996.

[21] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, Jan. 2002.

[22] M. Sheeran and J. Hughes. Parallel Functional Programming, an MSc and doctoral course given by the Functional Programming Group at Chalmers, 2012.

[23] J. Svensson, K. Claessen, and M. Sheeran. GPGPU kernel implementation and refinement using Obsidian. *Procedia Computer Science*, 1(1):2065–2074, 2010.

[24] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton-Jones. Algorithm + strategy = parallelism. *J. Funct. Program.*, 8:23–60, 1998.

[25] D. Zhang, A. Askarov, and A. C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proc. of the 18th ACM CCS*. ACM, 2011.