Functional High Performance Financial IT

The HIPERFIT* Research Center in Copenhagen – Project Paper –

Jost Berthold[1], Andrzej Filinski[1], Fritz Henglein[1], Ken Friis Larsen[1], Mogens Steffensen[2], and Brian Vinter[3]

[1] Department of Computer Science (DIKU), University of Copenhagen, Denmark.
[berthold,kflarsen,andrzej,henglein]@diku.dk
[2] Department of Mathematical Sciences (IMF), University of Copenhagen, Denmark.
mogens@math.ku.dk
[3] eScience Center, Niels Bohr Institute (NBI), University of Copenhagen, Denmark.
brian.vinter@nbi.dk

Abstract. The world of finance faces the computational performance challenge of massively expanding data volumes, extreme response time requirements, and compute-intensive complex (risk) analyses. Simultaneously, new international regulatory rules require considerably more transparency and external auditability of financial institutions, including their software systems. To top it off, increased product variety and customisation necessitates shorter software development cycles and higher development productivity. In this paper, we report about HIPERFIT, a recently etablished strategic research center at the University of Copenhagen that attacks this triple challenge of increased performance, transparency and productivity in the financial sector by a novel integration of financial mathematics, domain-specific language technology, parallel functional programming, and emerging massively parallel hardware.
HIPERFIT seeks to contribute to effective high-performance modelling by domain specialists, and to functional programming on highly parallel computer architectures in particular, by pursuing a research trajectory informed by the application domain of finance, but without limiting its research scope, generality, or applicablity, to finance. Research in HIPERFIT draws on and aims at producing new research in its different scientific fields, and it fosters synergies between them to deliver showcases of modern language technology and advanced functional methods with the potential for disruptive impact on an area of increasing societal importance.

1 Introduction

Today, the financial sector is confronted with fundamental computational challenges: Data volumes to be handled are growing at an exponential rate; stochastic

* HIPERFIT (http://hiperfit.dk) is a strategic research center funded by the Danish Council for Strategic Research (DSF) under grant no. 10-092299, founded in cooperation with the following partners from the financial industry: Danske Bank, Jyske Bank, LexiFi, Nordea, Nykredit Bank, and SimCorp.

Ricardo Peña-Marí and Rex Page (Eds.), Trends in Functional Programming (TFP'11) - 12th Int. Symp., Madrid, Spain, May 16-18, 2011, Revised Selected Papers, LNCS (to appear).

simulations consume in principle limitless numbers of compute cycles; quantitative and auditable risk management is becoming mandatory; real-time requirements hit speed-of-light limitations. At the same time, it becomes more and more common to negotiate non-standardised financial contracts, so-called over-the-counter (OTC) contracts. These are complex to model, manage and analyse, and yet product development cycles have become shorter than imagined even five years ago. This requires complex computational models, specifications and systems that are guaranteed to be correct, transparent, rapidly developed, and scalable on today's and tomorrow's hardware. What makes this a fundamentally new and interesting *scientific* challenge is that the problems need to be solved *simultaneously*, and thus trade-offs between the underlying financial mathematics, problem modelling, programming language technology, high-performance systems, and practical applicability must be explicitly accounted for.

To address these problems, we have recently established the *Research Center for Functional High-Performance Computing for Financial Information Technology* (HIPERFIT) at the University of Copenhagen, which brings together key researchers in the required scientific fields – programming languages, parallel systems, and mathematical finance – with the relevant industrial partners. Our fundamental hypothesis is that the above-mentioned simultaneous challenges of high transparency, high computational performance and high productivity can be solved more easily by an integrated approach using declarative domain-specific and high-level functional programming languages rather than by an incremental approach building on top of historically evolved software architectures and code bases that have originally been developed for sequential computer architectures. The approach taken by HIPERFIT is to eliminate low-level imperative programming by exploiting natural parallelism in declaratively expressed solutions and mapping it directly to emerging massively parallel commodity hardware.

1.1 Overview

In the present paper we first describe the research paradigm, strategy and organisation of HIPERFIT. We then explain the integrated approach taken, and the particular research themes we will work on (Section 3). Section 4 focuses on the functional programming aspects: We summarise the state of the art in language support for financial applications (Section 4.1) and give an overview of parallel functional programming paradigms and trends (Sections 4.2 and 4.3). In Section 5, we outline the two first project activities within HIPERFIT related to functional programming. Section 6 concludes.

2 Motivation and Background

In the year 2008, we saw one of the most severe worldwide financial crises ever. Induced by defaults in the American real-estate market (sub-prime loans), some investment banks collapsed and a large numbers of others were affected – taking down many other industries and ultimately leading to a general economic crisis

of global scale [19]. The crisis in 2008 demonstrates how complex dependencies are built up in the financial industry and that experts can vastly misjudge the impact of a local crash on other sectors.

2.1 Need for More Accurate Modelling in Mathematical Finance

To help avoid a repeat of the 2008 crash, financial institutions have initiated internal activities at a massive scale. Huge sums are invested in computational methods to improve modelling financial phenomena with all concerned parties. While the banks already have extensive modelling and pricing activities, the new problems establish a modelling and simulation paradigm vastly different from the existing system. Existing systems are based on macroscopic models and only model individual contracts in parameterised representations. The new requirement will be a detailed system of microeconomic models of the individual businesses and the combination of these into a global economic barometer that identifies the value and risk in a given bank.

2.2 Need for More Financial System and Software Transparency

The financial crisis that hit the world economy in 2008 has also triggered several new legislative initiatives that seek to govern the financial sector more carefully. The Basel-II agreement, its successor Basel-III under preparation (as CRD II-IV), and recently proposed SEC rules for computational models of securities [40], impose new capital adequacy and transparency requirements on the financial sector. These new rules have impact on banks' IT systems at all levels, ranging from high-level modelling of financial instruments to auditable internal risk models and their reliable implementation.

2.3 Need for More Computational Performance

Quantitative analyses in the financial industry have always called on great computing power. Such analyses have usually been devised by so-called "quants", having a background in mathematical finance, financial engineering, mathematics and physics. Their expertise is in the fields of option pricing, calibration, simulation, stochastic differential equations, partial differential equations, and statistics. Only recently have we seen increased focus on the efficiency and transparency of numerical and computational methods used in the analyses, which increasingly use Monte-Carlo and other simulation techniques [21]. Reasons for this trend lie both inside the industry, through an ever-growing competition for achieving more and more marginal benefits, and outside, by imposing new auditing and solvency procedures from international regulation (c.f. Section 2.2).

Recently, domain experts have started using the potentially tremendous parallel computing power of modern General-Purpose Graphics Processing Units (GPGPUs), encoding their algorithms in highly platform-dependent low-level languages. Low-level code written by a domain expert may perform well in the

short term, but is bound to lead to over-specialised, unmaintainable systems that do not satisfy auditing and transparency requirements. In consequence, there is an increasing demand for high-level programming language and high-performance systems expertise, complementing the requisite principal financial expertise.

3 The HIPERFIT Center

Funded by the Danish Council for Strategic Research, HIPERFIT started its work in January 2011. The center comprises four main research areas involving three departments of the University of Copenhagen, five partners from the Danish financial industry, and a French functional-programming based finance IT company. The center has been made possible by a grant by the Danish Strategic Research Council under its Programme for Strategic Growth Technologies. The grant provides funding for 1 permanent faculty, 3 post-doctoral and 6 PhD scholarship positions, totalling 33 person years spread over the different scientific disciplines. The first HIPERFIT appointments will be in place by the end of 2011.

3.1 Research Goals, Organisation, and Methodology

Research in HIPERFIT aims at solving problems of today's computing in finance in a holistic, integrated approach. HIPERFIT therefore joins researchers with state-of-the-science expertise in four *research areas* relevant for high-performance financial applications: Theory and practice of mathematical finance (MF), domain-specific languages (DSL), functional programming (FP), and high-performance systems (HPS).

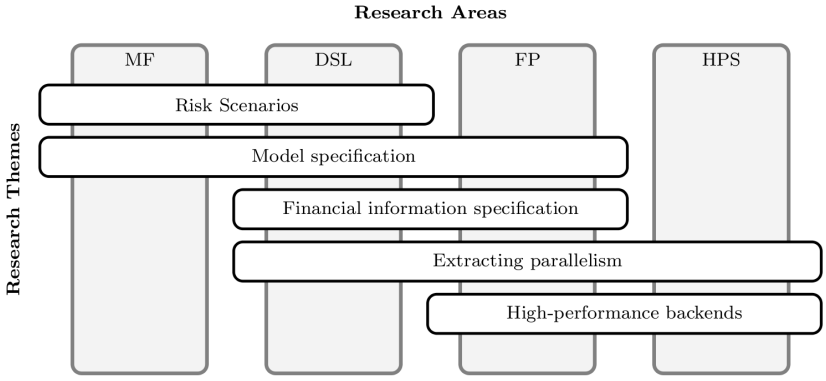

Fig. 1. Relationship between research areas and research themes

A major goal of HIPERFIT is to present alternatives to the above-mentioned low-level code with platform-dependent optimisations so as to facilitate a more

enduring development process of efficient maintainable systems. Essential ingredients to achieving this are declarative languages and advanced compilation of domain-specific abstractions. We believe that side-stepping imperative programming bears the elements of a *disruptive technology* with drastic productivity and performance improvement potential.

The work in HIPERFIT is organised in general *research themes*, which cut across research areas and are driven by *cases*. Figure 1 depicts the research areas and their relationship to our initial research themes. *Cases* are concrete projects for exploration and development, either motivated by practical needs of industry partners (problem-driven), or by the intent to evaluate novel technologies and gather know-how for later use (technology driven). Cases may or may not contain information protected by industry partners. They usually have focused objectives adequate for Master's thesis projects, and they realise useful and timely short-term goals. The overarching research themes, on the other hand, are more open-ended to foster exploratory thinking that is not entrenched in and tied to incremental evolution of current practice. Research theme work is carried out primarily by faculty, postdocs and Ph.D. students researchers.

3.2 Research Themes in HIPERFIT

Initial discussions with our industry partners have led to identifying several cross-cutting research themes for the start of HIPERFIT, depicted in Figure 1. Each research theme will be supported by cases, part of which are provided by the industry partners.

Risk Scenarios We try to describe the transition from observables (like current prices and historical data) to scenario generation and from scenario generation to reporting and management. Adequate risk scenarios have immediate relevance for management decisions, including deriving capital requirements to ensure stability in unlikely and extreme situations.

Model Specification Financial models in practical use today vary from so-called "model-free" evaluation (prices given completely in terms of other prices) to sophisticated stochastic processes (such as advanced multi-dimensional jump-diffusions). We want to systematically explore and compare benefits and costs of models for different applications (solvency, accounting, or management), parallelisation and optimisation of numeric methods, and the impact of imprecisions that might result from the latter.

Domain-Specific Languages (DSLs) for Finance. Declarative DSLs to describe a range of financial products have already come into widespread use in the financial sector. We aim to complement these languages with similarly expressive DSLs for other financial information, and especially for financial models. Our goal is a complete DSL framework with broad application coverage, suitable

both for internal reporting and statistics, external auditing, and computation in large risk scenarios. We will describe the DSL approach in the financial domain, and our goals, in more detail in Section 4.1.

Extracting Parallelism from Declarative Specifications The core goal of this research theme is to analyse and transform large-scale financial computations to expose their inherent computational parallelism. Departing from work on existing applications and domain-specific abstractions, we plan to derive a tailor-made language for large-scale numeric computations which suits the needs of mathematical finance, while efficiently executable on modern parallel hardware. Thanks to their high-level nature, parallel functional languages appear to be an excellent platform for this. We expect vector and matrix operations and accumulating reductions to be the major source of parallelism at this stage, but aim to identify more domain-specific parallelisation schemes. The DSL development for financial models will lead to additional or modified requirements. Typical operations for *valuation (pricing)* of stochastic financial models need to be translated into the parallel operations provided. The functional approach we take gives us a good position to formally assess correctness and precision of the obtained results, and – to some extent – to statically estimate the translated programs' performance. Sections 4.2 and 4.3 expand on previous and related work in the area of parallel functional programming and parallel hardware support.

High-Performance Backends for Novel Hardware Embracing novel parallel hardware like GPGPUs is an integral part of HIPERFIT. Models and language framework will be designed with execution on next-generation processors in mind from the start, mapping the parallelism that is expressed by the functional programming activities onto a number of parallel computer architectures. In this research theme, activities will start by optimising existing algorithms and implementations, and profit from synergies with other scientific computing activities on parallel hardware. We expect to follow a byte-code based approach and just-in-time compilation, and ultimately intend to deliver a full high-performance backend tailored for financial and scientific applications.

4 Functional Programming and HIPERFIT

4.1 Domain-Specific Languages for Financial Applications

Pervasive Trend to Domain-Specific Languages. Domain-specific languages (DSLs) capture knowledge of application experts in tailor-made constructs and thereby offer great programing comfort. DSLs are so widespread and successful in practice that it is easy to overlook them: Logical data modelling and declarative querying, with high-level support for physical storage layout (particular index data structures) and automatic query optimisation, as embodied in Relational Database Systems (RDBMSs); functional dependencies

between atomic, vector- and matrix-based data, with automatic incremental recomputation, as embodied in spreadsheets; structural specification of strings, with automatic generation of provably efficient streaming processors, as embodied in regular expression ("lexing") and context-free grammar ("parsing") tools.

Programming language research has only recently discovered DSLs as a research area and capitalised on the notion, though [32]. Simultaneously with the rise of the term in research, one could observe DSL technology invading profitable commercial domains. For example, the Cryptol language [26] enables constructing reliable cryptographic software and hardware implementations with ease and high assurance. Recently, we also see some proposals for "DSLs" for parallel programming [42], or specifically for next-generation parallel hardware, GPGPUs or FPGAs. However, whether to really label these "DSL" is a debatable subject: A particular target platform definitely does not constitute an *application* domain, and the particular field hardly exposes characteristics which would justify DSL development (special notation, automation, data structures [32]). We are not aware of many scientific projects combining a proper DSL approach with novel parallel hardware. Notable exception are a relatively new project Diderot [45] (a "parallel DSL" for image analysis), and the Feldspar project [17] which targets GPGPUs for high-performance signal processing using a DSL approach.

DSLs in Finance Financial applications have been identified as a promising DSL area relatively early. Researchers have successfully modeled and analysed financial instruments [35], commercial contracts [2], and risk management [6] using DSL technology. The French company LexiFi, one of the industrial partners in HIPERFIT, has matured the research on financial DSLs [35] into the language MLFi [27], which is embedded into OCaML as a combinator library for describing contracts and valuation (called a "domain-specific embedded language", DSEL).

The hallmark feature of such contract languages is that they allow more complex instruments/contracts/risk models to be built up by composing simpler, often reusable, components that can be shared amongst different instruments. Also, the same domain-specific descriptions enable different interpretations. For instance, a description of a financial instrument in MLFi can be used both for pricing the instrument and for backoffice automation; that is, managing when options and obligations described in the instrument are to be exercised and when payments are to be made or received.

Project Goal: DSL Framework for Finance The general goal of DSLs is to support fast implementation, extensibility, reuse across financial institutions, maintainability and low total cost of ownership (TCO) for the domain expert as a user. We want to create a framework for financial information applications which covers various applications: reporting to auditors and public authorities, data communication with clearing houses, internal reporting and statistics, computations for the purpose of internal risk management, and flexible integration for standard routines such as accounting and confirmation processing.

DSLs for financial instruments are commonly used in many companies today, but often mix contract and valuation aspects. A crucial goal of HIPERFIT is to design similarly expressive languages to describe the stochastic models and computational valuation methods, and to achieve clear separation and interfaces towards a universal valuation engine. We will investigate existing DSL approaches in the different areas and experiment with combining them to identify the lines of separation and useful language features.

4.2 Parallel Functional Programming in HIPERFIT

Why parallel functional programming matters. [4]

Functional programs are easy to read and understand, program construction and code reuse are simplified (glue), and programs are transformed, optimised and formally reasoned about with relative ease. More specific to parallel computations, the absence of side effects makes data dependencies and inherent parallelism manifest, (purely) functional parallel programs have deterministic semantics irrespective of the evaluation order, and reduction semantics is inherently parallel. Last but not least, higher-order functions can nicely describe common parallelisation patterns as skeletons [15,38], without the reader getting lost in technical details or particularities of the concrete algorithm. In all, irrespective of the concrete programming model, the high level of abstraction provided by functional languages makes them suitable languages to conceptually describe parallelism, in an executable specification.

Models, paradigms and classification A number of programming models for parallel functional programming have been developed. They can be categorised along different aspects of programming and implementation. A good criterion for classifying parallel programming models is the *degree of explicitness*: how much parallelism needs to be controlled and specified by the programmer. Skillicorn and Talia [41] subdivide explicitness along several aspects: decomposition, mapping, communication, and synchronisation, as increasing degrees of explicitness for parallel subcomputations.

The main credo in functional languages being high abstraction, it is not surprising that most approaches to parallelism try to limit the programmer's control of parallelism. Parallelism should ideally be non-invasive, i.e. not require large changes to a program's source code. In the extreme, inherent parallelism exploited stems from the reduction semantics, for example in *parallel Haskell* (pH [1]): lazy graph reduction is changed to eager evaluation for performance. However, experience has shown that such completely implicit approaches are of limited use. The predominant category is a mid-level of "controlled parallelism" [23], where programmers specify parallelism, while details are left to the language implementation. In Figure 2, we provide a categorisation of parallel

[4] In reverence to Backus [7], Hughes [24] and Hammond/Michaelson [23, Introduction].

functional languages that expands this semi-explicit mid-level further into subcategories. As another aspect, the vertical axis in the figure shows to what extent units of computation in parallel programs are *explicitly interacting*.

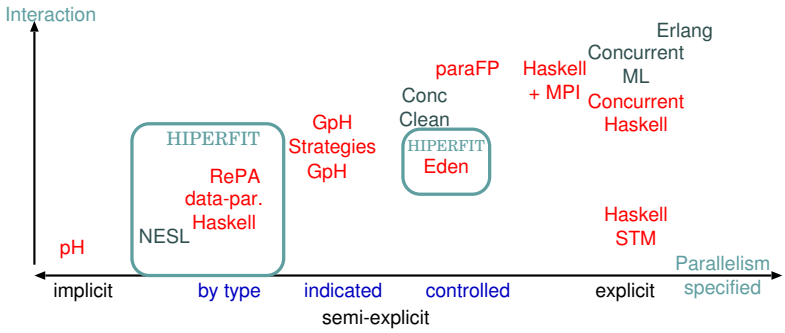

Fig. 2. Parallel Haskells and other functional languages classified

One classical approach is to parallelise operations over special bulk data types – data parallel languages. Examples are NESL [12], Data-parallel Haskell [14], and its newer variant RePA [25]. Language extensions targeting GPGPUs [29,13] also fall in this category of type-driven parallelism.

Slightly more powerful, and more involved, is to *indicate inherent parallelism* in a functional program by annotations or special evaluation combinators, to inform compiler and runtime system about whether an independent computation *should* be done in parallel. This is the model of Glasgow parallel Haskell (GpH) [44]. Evaluation strategies built on GpH [43] (recently overhauled [31]) provide slightly more control, enabling the programmer to force evaluation of subexpressions to a certain degree (in parallel or sequentially). This facilitates opportunistic parallel evaluation. It does not guarantee parallelism, however. In contrast, parallelism annotations used in Concurrent Clean [37] have mandatory operational semantics, providing *controlled* parallelism. The programmer explicitly specifies parallel scheduling; programs using controlled parallelism are indeed parallel and expose their parallel behaviour. Skeleton-based parallelisation [15] could be included in this category since, commonly, the programmer has to explicitly choose the algorithmic pattern implemented by a certain skeleton, and to follow it. However, we prefer to categorise them as implicit (likewise Skillikorn and Talia [41]), since a skeleton's parallel implementation is entirely hidden in libraries. Other examples of controlled functional parallelism are Hudak's para-functional programming approach and successors [33], and the language Eden [28]. Often we find the concept of *processes* and *channels* between them to define process networks. The language Eden [28] is the major representative of this approach in the Haskell world. Eden retains a mostly [10] functional interface, with a notion of processes specified by their input-output mapping, and implicitly connected via channels which may transfer data as *streams*. It has

been demonstrated [9,8,5] that Eden provides good support for skeleton-based programming, both for the skeleton user and as an implementation language.

Languages like Concurrent Clean and Eden are still (mostly) implicit about the communication details and synchronisation. Going even further, we find functional languages with *explicit message-passing and concurrency*. Examples using message passing are Concurrent ML [39], Haskell-MPI, and notably Erlang [3]. Concurrent Haskell [36] and Haskell transactional memory (STM) use shared memory, where threads communicate via shared mutable variables. A side remark on our categorisation: While interaction and explicitness of parallelism are mostly correlated, Haskell STM is the notable "outlier". There are no STM constructs for interaction between concurrent threads.

In general, *concurrency* is a programming model which allows to separate independent (usually effectful) computations into multiple (sometimes interacting) execution threads. Historically, this aims at supporting responsive distributed and interactive systems, which is also useful in the absence of actual parallel execution. Concurrency constructs are often also used to achieve genuine *parallelism*, speeding up a computation by executing its computational steps simultaneously ("in parallel") on computers with multiple processing units – and guaranteeing to do so. In contrast to this, concurrency can be understood as *sequential* computation, but with internal nondeterministic choices for selecting the next step. This is done by splitting the computation in a set of (sequential) *threads* or (sequential) *processes* (threads with shared memory). Assuming the implementation executes threads in parallel, concurrency can be a good implementation tool for parallel algorithms. Experience has shown that the large degree of control offered by concurrency abstractions and explicit message passing can prove useful for advanced parallel functional programming [10]. Functional languages also allow for more deterministic models to implement parallelism.

Project Goal: Tailored Parallel Functional Language Within HIPERFIT, we aim to develop a functional language that can be productively used to express computations in mathematical finance, and which exposes inherent parallelism in these computations. Driven by the application domain of financial modelling, we will identify common computation patterns and their potential for parallelisation. Potentially parallel computations should be easy to extract and transform into explicitly parallel operations on a variety of modern parallel platforms.

In Figure 2, we have sketched the functional programming languages we expect to be most relevant for HIPERFIT. Apart from functional languages we also expect to draw on the heritage of classical bulk-data programming languages such as APL, SETL and SQL. Principally, data parallelism [12] appears to be a good match for the HIPERFIT application domain: it enables concise and long-term maintainable specifications of a wide variety of inherently parallelisable computations, without committing to any particular implementation strategy or execution environment. It facilitates correctness proofs and performance estimates, and, under eager evaluation, it has a useful compositional parallel cost model. Pure data parallelism, on the other hand, is less suitable for loosely-

coupled systems. We therefore expect to also use more explicit and coarse-grained programming models (like e.g. Eden), however avoiding the burden of explicit message-passing and using implementation skeletons where possible.

At a later stage, we expect the DSL development for financial models to yield additional or modified requirements. Useful abstractions and patterns of parallelism will be identified from working on concrete projects. Ultimately, our language should support specific typical operations tailored to the application domain, risk analysis and valuation in a financial context, but without hard-wiring the application domain into it.

4.3 Support for Multicore and Novel Parallel Hardware

In the previous section, we have motivated our functional approach by a number of historic achievements of relevance, based on more than 20 years of research in parallel functional programming. Yet, it is interesting to see how much the availability of advanced GPGPU hardware in practice changes the scientific landscape. GPGPUs are made for SIMD-style parallel computations with minor memory requirements. Parallel software has often been built as a match to existing well-performing and well-understood hardware. Functional approaches claim to capture parallelism at a more abstract level, but recent publications about GPGPU programming in functional languages focus exactly on these simple embarrasingly parallel problems, where quick success can be expected.

Especially for accelerating financial simulations, the approach of modern GPGPUs appears promising; we already know that Monte Carlo methods can get massive speedups, due to their simple structure. This holds not only for finance, but also for various scientific applications using Monte Carlo simulations, for instance particle physics and computational geo-science. Today, we find several language bindings to GPGPU accelerators in the Haskell research community. They realise easy data parallelism on specially designated parallel vectors (Nikola [29]) or arrays (Accelerate [13]). These research prototypes deliver important insight for future GPGPU language design and pragmatics, but we still have a way to go towards making this research software work in practice for the average programmer or domain expert. And as mentioned, we observe an antithetic trend in scientific computation: scientists of various disciplines choose to operate at the lowest abstraction level API, Cuda C code.

Before GPGPUs became the prodigy of parallelism, a first wave of interest for parallelism was induced by *multicore CPUs*. Having several cores is a mere normality today, yet major functional languages have only recently optimised their multicore support. The high level of the languages, and implementation traditions, makes it sometimes very hard to optimise locality, but promising results have been obtained [5,11,30], and even entire new projects for multicore were set up, for instance Manticore [20]. With the movement towards OpenCL [34], both multicore processors, GPUs, and future heterogenous manycore architectures can be captured in a single computational idiom. OpenCL is supported by major manufacturers of novel hardware, and HIPERFIT will likely contribute to advancing its development and use as an intermediate target language.

Another recent effort is the initiative to make parallel Haskell apt for widespread commercial use, initiated by the Well-Typed consultancy and sponsors [4]. One of the first activities was to revive Haskell-MPI (from 2001) – which seems to be a major industrial demand, while some researchers consider message passing "harmful" [22]. Aiming at a higher hardware abstraction level, the latest efforts of that project are into performance analysis tool support [16].
Various activities on parallel Haskell are going on and in diverse directions. We believe that there is still important work to do, however. Our intention with HIPERFIT in this direction is to advertise and test various existing approaches through prototype implementations. We will closely follow and adopt the latest research in parallel functional programming, and at the same time continue work on our own high-performance backend, providing as general a platform for processing bulk data as we can realise.

5 Project Start and First Activities

Integrating Valuation and Contract Specification. The major use case of existing contract specification languages is *valuation (pricing)*: determining the value of a financial contract at any point in time, based on a stochastic model of the future. Existing contract languages have usually been developed together with a valuation semantics from the start. Based on a probabilistic model of unknown variables (for instance, modelling changes in interest rate for zero-coupon bonds), a range of possible outcomes and their probabilities is computed. A simple stochastic method for valuation is Monte Carlo simulation, which is inherently parallel by nature. More advanced methods might lead to a large number of possible outcomes and are thus computationally intensive; again massive parallelisation can hopefully lead to faster results.

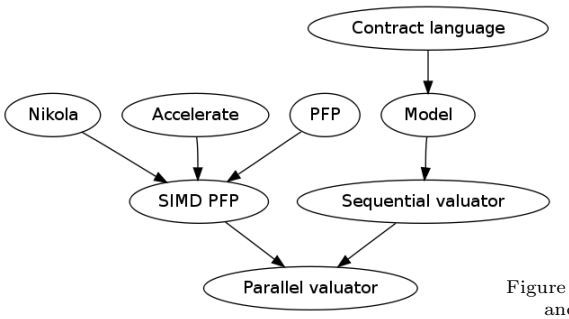

Figure by Michael Flænø Werk and Joakim Ahnfelt-Rønne

Fig. 3. Integration Overview for Contracts, PFP, and Data Parallelism

As one strand of HIPERFIT activities, we are evaluating existing GPGPU support in Haskell, namely the Nikola [29] DSEL and the accelerated Haskell array library [13], to offload vector computations to a GPGPU. Figure 3 gives an

overview of the evaluated technologies. A recently concluded Master's thesis in HIPERFIT prototypes a Haskell system that combines existing technologies and applies them to accelerated stochastic contract valuation. Another strand is a domain-specific approach to Probabilistic Functional Programming (PFP) [18]. This DSEL separates the method of evaluation from the stochastic model and is thus helpful in structuring the implementation of our intended parallel valuation engine. Ultimately, we aim at producing a fully modular valuation engine, where instruments (contracts) and models (stochastic processes) are specified independently.

Port of Data.Array.Accelerate to OpenCL. In view of our general goal to use and produce open standards and open source software in HIPERFIT, we would like to pave the way towards using the standardized OpenCL [34] rather than the proprietary Cuda for GPGPU computations. We are therefore porting Data.Array.Accelerate [13] to OpenCL. The technology for this accelerated array library is well understood; we expect to mainly solve technical and engineering hurdles here. As a by-product, a new library of OpenCL bindings will be created. At a later time, we might also be able to maturate the Nikola [29] research to better usability by non-experts, and port it to OpenCL as well.

As discussed earlier (see Section 4.3), the GPU platform and programming model appears to be tailored, if not rigidly limited, to data parallelism. Control structures are very limited, memory accesses are entirely explicit, recursion is not possible, branching constructs execute both alternatives. On the other hand, precisely these properties could provide the magic wand for cost analysis and thereby performance prediction of parallelised valuation code. In view of this long-term goal, it is a strategic decision to generate know-how about GPU bindings, involving embedded compilation, in the context of HIPERFIT.

Other activities. Work has also started in other research areas of HIPERFIT. To give a general idea of what our case-based working methodology looks like in practice, we mention a few other activities. One interesting area is to parallelise random number generation in a reproducible manner, for use in Monte Carlo simulations. A HIPERFIT project is investigating existing research to extract best practice on using GPGPUs for this problem. In a second strand of activities, we aim to extract patterns and common usage from existing in-house bank software, by inspecting and parallelising kernel routines of an in-house C++ library. In another project, we want to take the perspective of an informed economist on the topic of instrument valuation, by creating a survey and classification of financial instruments and models. Parallel implementations of selected valuation models will follow, which can be structured to reflect the generalities that have been identified. The implementation work also serves to evaluate other declarative parallel languages (to be determined) and to identify recurring patterns and potentially useful features for later DSL development.

6 Conclusions

We have presented motivation, goals and methods of the HIPERFIT research center, a joint activity of researchers in mathematical finance, programming languages, parallel computing, and computer systems in collaboration with Finance IT professionals. In order to meet new and increasing computational needs of a complex global industry of major impact, HIPERFIT aims at integrated solutions that transcend a single researcher's field of expertise, and explicitly fosters interdisciplinarity and practical relevance through its paradigm of case-driven research themes.

We want to develop advanced new methods in mathematical finance and work towards a framework of domain-specific languages to express financial instruments, models and valuation methods. Parallelisation techniques using a functional approach should both lead to efficient parallel execution on novel hardware, and leave the code accessible for proofs of semantic properties and, to some extent, performance predictions.

The goals of HIPERFIT which relate to programming languages appear to carry the highest risk of achieving practical impact, but arguably also promise the best long-term investment. Past research on parallelism concepts has often come to success and innovation by focusing on particular application domains. Immediate practical use and challenging problems derived from practice are a good touchstone for research. Especially because of the unique combination of advanced programming language technology and parallelism envisioned in HIPERFIT, we consider it an exciting opportunity to perform and promote research in DSLs and parallel functional programming, and hope to make it one of its major showcases.

The HIPERFIT Website: `http://www.hiperfit.dk`.

References

1. Aditya, S., Arvind, Augustsson, L., Maessen, J.W., Nikhil, R.S.: Semantics of pH: A parallel dialect of Haskell. In: Hudak, P. (ed.) Proceedings of the Haskell Workshop. pp. 35–49. La Jolla, USA (1995)
2. Andersen, J., Elsborg, E., Henglein, F., Simonsen, J.G., Stefansen, C.: Compositional specification of commercial contracts. International Journal on Software Tools for Technology Transfer (STTT) 8(6), 485–516 (2006)
3. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in ERLANG. Prentice Hall, Hertfordshire, 2nd edn. (1996)
4. Astapov, D.: Parallel Haskell project underway. Blog post (Oct 2010), `http://www.well-typed.com/blog/48`
5. Aswad, M., Trinder, P., Al Zain, A.D., Michaelson, G., Berthold, J.: Low Pain vs. No Pain Multicore Haskells. In: Horváth, Z., Zsók, V., Achten, P., Koopman, P. (eds.) Trends in Functional Programming (TFP)'09. pp. 49–64. Intellect, Exeter (2010)
6. Augustsson, L., Mansell, H., Sittampalam, G.: Paradise: A two-stage DSL embedded in Haskell. In: ICFP'08, Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming. pp. 225–228. ACM, New York (2008)

7. Backus, J.: Can programming be liberated from the von neumann style. Communications of the ACM 21(8), 613–641 (1978)
8. Berthold, J., Dieterle, M., Loogen, R.: Implementing Parallel Google Map-Reduce in Eden. In: Sips, H., Epema, D., Lin, H. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 990–1002. Springer, Heidelberg (2009)
9. Berthold, J., Dieterle, M., Loogen, R., Priebe, S.: Hierarchical Master-Worker Skeletons. In: Hudak, P., Warren, D. (eds.) PADL'08. LNCS, vol. 4902, pp. 133–152. Springer, Heidelberg (2008)
10. Berthold, J., Loogen, R.: Parallel Coordination Made Explicit in a Functional Setting. In: Horváth, Z., Zsók, V. (eds.) IFL'06. LNCS, vol. 4449, pp. 73–90. Springer, Heidelberg (2007)
11. Berthold, J., Marlow, S., Hammond, K., Al Zain, A.: Comparing and Optimising Parallel Haskell Implementations for Multicore Machines. In: Tomoya Enokido et al. (ed.) 3rd Int. Workshop on Advanced Distributed and Parallel Network Applications (ADPNA'09). IEEE (2009), (previously presented at IFL'08)
12. Blelloch, G.: Programming parallel algorithms. CACM 39(3), 85–97 (1996)
13. Chakravarty, M.M., Keller, G., Lee, S., McDonell, T.L., Grover, V.: Accelerating Haskell array codes with multicore GPUs. In: DAMP'11, Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming. pp. 3–14. ACM, New York (2011)
14. Chakravarty, M., Leshchinskiy, R., Jones, S.P., Keller, G., Marlow, S.: Data parallel Haskell: A status report. In: DAMP'07, Workshop on Declarative Aspects of Multicore Programming. pp. 10–18. ACM, New York (2007)
15. Cole, M.I.: Algorithmic Skeletons: Structured Management of Parallel Computation. Research Monographs in Parallel and Distributed Computing, MIT Press, Cambridge(MA) (1989)
16. Coutts, D.: Spark visualisation in threadscope. Contribution to the Haskell Implementors' Workshop 2011 (Tokyo) (September 2011)
17. Dévai, G., Tejfel, M., Gera, Z., Páli, G., Nagy, G., Horváth, Z., Axelssony, E., Sheeran, M., Vajda, A., Lyckegård, B., Persson, A.: Efficient code generation from the high-level domain-specific language Feldspar for DSPs. In: ODES-8, 8th Workshop on Optimizations for DSP and Embedded Systems. Toronto (April 2010)
18. Erwig, M., Kollmansberger, S.: Probabilistic functional programming in Haskell. J. of Functional Programming 16(1), 21–34 (2006)
19. FCIC: The financial crisis inquiry report. Tech. rep., Financial Crisis Inquiry Report Commission (Jan 2011), available online: `http://www.fcic.gov/report`
20. Fluet, M., Rainey, M., Reppy, J., Shaw, A., Xiao, Y.: Manticore: A Heterogeneous Parallel Language. In: Glew, N., Blelloch, G.E. (eds.) DAMP'07, Workshop on Declarative Aspects of Multicore Programming. pp. 37–44. Nice, France (2007)
21. Glasserman, P.: Monte Carlo methods in financial engineering, Applications of Mathematics (New York), vol. 53. Springer, New York (2004)
22. Gorlatch, S.: Send-receive considered harmful: Myths and realities of message passing. ACM TOPLAS 26(1), 47–56 (2004)
23. Hammond, K., Michaelson, G. (eds.): Research Directions in Parallel Functional Programming. Springer, London (2000)
24. Hughes, J.: Why functional programming matters. The Computer Journal 32(2), 98–107 (1989)
25. Keller, G., Chakravarty, M.M., Leshchinskiy, R., Peyton Jones, S., Lippmeier, B.: Regular, shape-polymorphic, parallel arrays in Haskell. In: ICFP'10, Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. pp. 261–272. ACM, New York (2010)

26. Lewis, J.: Cryptol: specification, implementation and verification of high-grade cryptographic applications. In: FMSE '07, Proceedings of the ACM workshop on Formal methods in security engineering. pp. 41–41. ACM, New York (2007)
27. LexiFi: Contract description language (MLFi). Web page and white paper, http://www.lexifi.com/technology/contract-description-language
28. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel Functional Programming in Eden. Journal of Functional Programming 15(3), 431–475 (2005)
29. Mainland, G., Morrisett, G.: Nikola: embedding compiled GPU functions in Haskell. In: Haskell'10, Proceedings of the third ACM SIGPLAN Symposium on Haskell. pp. 67–78. ACM, New York (2010)
30. Marlow, S., Jones, S.P., Singh, S.: Runtime Support for Multicore Haskell. In: ICFP'09, Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. pp. 65–78. New York (2009)
31. Marlow, S., Maier, P., Loidl, H.W., Aswad, M.K., Trinder, P.: Seq no more: Better strategies for parallel Haskell. In: Haskell'10: Proceedings of the third ACM SIGPLAN Symposium on Haskell. pp. 91–102. ACM, New York (2010)
32. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys 37(4), 316–344 (2005)
33. Mirani, R., Hudak, P.: First-class monadic schedules. ACM TOPLAS 26(4), 609–651 (2004)
34. Munshi, A.: The OpenCL Specification. Khronos OpenCL Working Group (2010), http://www.khronos.org/opencl/
35. Peyton Jones, S., Eber, J.M., Seward, J.: Composing contracts: an adventure in financial engineering (functional pearl). In: ICFP '00, Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming. pp. 280–292. ACM, New York (2000), (Later extended to a book chapter)
36. Peyton Jones, S., Gordon, A., Finne, S.: Concurrent Haskell. In: Proceedings of POPL '96. pp. 295–308. ACM, New York (1996)
37. Plasmeijer, M., van Eekelen, M.: Functional Programming and Parallel Graph Rewriting. Addison-Wesley, Reading (MA) (1993)
38. Rabhi, F.A., Gorlatch, S. (eds.): Patterns and Skeletons for Parallel and Distributed Computing. Springer, London (2003)
39. Reppy, J.H.: Concurrent Programming in ML. Cambridge Univ. Press (1999)
40. Securities and Exchange Commission: Proposed rule: Asset backed securities (2010), http://www.sec.gov/rules/proposed/2010/33-9117.pdf
41. Skillicorn, D.B., Talia, D.: Models and languages for parallel computation. ACM Computing Surveys 30(2), 123–169 (1998)
42. Sobral, J.L., Monteiro, M.P.: A domain-specific language for parallel and Grid computing. In: DSAL'08, Proceedings of the 2008 AOSD Workshop on Domain-Specific Aspect Languages. pp. 2:1–2:4. ACM, New York (2008)
43. Trinder, P., Hammond, K., Loidl, H.W., Peyton Jones, S.: Algorithm + Strategy = Parallelism. J. of Functional Programming 8(1), 23–60 (1998)
44. Trinder, P., Hammond, K., Mattson Jr., J., Partridge, A., Peyton Jones, S.: GUM: a Portable Parallel Implementation of Haskell. In: PLDI'96. pp. 78–88. ACM, New York (1996)
45. Diderot project. Website (2010), http://diderot-language.cs.uchicago.edu/