Generative Code Specialisation for High-Performance Monte-Carlo Simulations

Don Stewart[!]
Hugh Chaffey-Millar[2]
Gabriele Keller[1]
Manuel M. T. Chakravarty[1]
Christopher Barner-Kowollik[2]

[1]Programming Languages and Systems
School of Computer Science & Engineering
University of New South Wales
Email: {dons,keller,chak}@cse.unsw.edu.au

[2]Centre for Advanced Macromolecular Design
School of Chemical Sciences and Engineering
University of New South Wales
Email: h.chaffey-millar@student.unsw.edu.au
c.barner-kowollik@unsw.edu.au

UNSW-CSE-TR-0710
April 2007

Abstract

We address the tension between software generality and performance in the domain of scientific and financial simulations based on Monte-Carlo methods. To this end, we present a novel software architecture, centred around the concept of a *specialising simulator generator*, that combines and extends methods from generative programming, partial evaluation, runtime code generation, and dynamic code loading. The core tenet is that, given a fixed simulator configuration, a generator in a functional language can produce low-level code that is more highly optimised than a manually implemented generic simulator. We also introduce a skeleton, or template, capturing a wide range of Monte-Carlo methods and use it to explain how to design specialising simulator generators and how to generate parallelised simulators for multi-core and distributed-memory multiprocessors.

We evaluated the practical benefits and limitations of our approach by applying it to a highly relevant problem in computational chemistry. More precisely, we used a Markov-chain Monte-Carlo method for the study of advanced forms of polymerisation kinetics. The resulting implementation executes faster than all competing software products, while at the same time also being more general. The generative architecture allows us to cover a wider range of chemical reactions and to target a wider range of high-performance architectures (such as PC clusters and SMP multiprocessors).

We show that it is possible to outperform low-level languages with functional programming in domains with very stringent performance requirements if the domain also demands generality.

Generative Code Specialisation for High-Performance Monte-Carlo Simulations [*]

Don Stewart
Gabriele Keller   Manuel M. T. Chakravarty
Programming Languages and Systems
School of Computer Science and Engineering
University of New South Wales
{dons,keller,chak}@cse.unsw.edu.au

Hugh Chaffey-Millar
Christopher Barner-Kowollik
Centre for Advanced Macromolecular Design
School of Chemical Sciences and Engineering
University of New South Wales
h.chaffey-millar@student.unsw.edu.au
c.barner-kowollik@unsw.edu.au

Abstract

We address the tension between software generality and performance in the domain of scientific and financial simulations based on Monte-Carlo methods. To this end, we present a novel software architecture, centred around the concept of a *specialising simulator generator*, that combines and extends methods from generative programming, partial evaluation, runtime code generation, and dynamic code loading. The core tenet is that, given a fixed simulator configuration, a generator in a functional language can produce low-level code that is more highly optimised than a manually implemented generic simulator. We also introduce a skeleton, or template, capturing a wide range of Monte-Carlo methods and use it to explain how to design specialising simulator generators and how to generate parallelised simulators for multi-core and distributed-memory multiprocessors.

We evaluated the practical benefits and limitations of our approach by applying it to a highly relevant problem in computational chemistry. More precisely, we used a Markov-chain Monte-Carlo method for the study of advanced forms of polymerisation kinetics. The resulting implementation executes faster than all competing software products, while at the same time also being more general. The generative architecture allows us to cover a wider range of chemical reactions and to target a wider range of high-performance architectures (such as PC clusters and SMP multiprocessors).

We show that it is possible to outperform low-level languages with functional programming in domains with very stringent performance requirements if the domain also demands generality.

Keywords Generative programming; code specialisation; runtime code generation; high-performance computing; monte carlo methods.

[*] This work was funded by the UNSW FRGP *High Performance Parallel Computing for Complex Polymer Architecture Design.*

1. Introduction

The tension between software generality and performance is especially strong in computationally intensive software, such as scientific and financial simulations. Software designers will usually aim to produce applications with a wide range of functionality, which in the case of simulations means that they are highly parameterisable and, ideally, target different types of high-performance hardware. Scientific software is often used for new research tasks, and financial software is often employed for new products and new markets. In both cases, there is a high likelihood that the boundary of previous uses will be stretched. However, generality often comes with a performance penalty, as computations become more interpretative, require more runtime checks, and use less efficient data structures.

As an example, consider a computational chemistry simulation, using a Monte Carlo method, that in its innermost loop repeatedly selects a random chemical reaction from a set of possible reactions. The probability of the selection is determined by the concentration of available reactants and the relative probabilities of each of the reactions. If we aim for generality, the code will have the capability to handle a wide range of reactions. These reactions and their probabilities will be stored in a data structure that the code will have to repeatedly traverse when making a selection and when updating the level of concentration of the various reactants. This is an interpretative process whose structure is not unlike that of an interpreter applying the rules of a term rewriting system repeatedly to the redexes of a term. The more general the rules handled by the interpreter, the more interpretative overhead we get, and the less rewrites will be executed per second.

On the other hand, to maximise performance, we need to eliminate all interpretative overhead. In the extreme we have a program that hard codes a single term rewriting system; i.e., we compile the term rewriting system instead of interpreting it. We can transfer that idea to the chemistry simulation by specialising the simulator so that it only applies a fixed set of reactions. Such specialisation can have a dramatic impact on simulators, where a relatively small part of the code is executed very often and even the minor inefficiency of a few additional cpu cycles can add significantly to the overall running time of the application. As giving up on generality is not an option, and we cannot expect the user to manually specialise and optimise the simulation program for the exact reactions and input parameters of a particular simulation, we instead apply the compilation model to high-performance simulations.

In programming languages the move between interpreter and compiler is well known from the work on partial evaluation [11]. More generally, research on generative programming [5] and selfoptimising libraries [19] introduced approaches to code specialisation in a range of application areas, including numerically intensive applications [7, 20]. Much of this work is concerned with providing general libraries that are specialised at compile time. In this paper, we transfer these ideas from libraries to applications and combine them with runtime code generation and dynamic code loading.

More precisely, we introduce a novel software architecture for simulators based on Monte-Carlo methods. This architecture, based on generative code specialisation, uses functional programming to overcome the tension between generality and performance in a way that is transparent to the end user. Specifically, we implement a *specialising simulator generator* in a functional language and use it to generate highly optimised C code specialised for specific simulator configurations. Our software architecture uses a standard, optimising C compiler together with Haskell's *foreign function interface* and a variant of our previous work [15] on dynamically loaded plugins to compile and load the generated simulator into the main application for execution and rendering of the simulation results.

We capture the essence of a wide range of Monte-Carlo methods in the form of a skeleton; i.e., a template expressed by a type class and a parametrised function. We discuss the design of specialising simulator generators and the parallelisation of the generated simulators for Monte-Carlo methods captured by the skeleton. Moreover, we demonstrate the practical relevance of the skeleton and our software architecture by concrete applications from finance and computational chemistry. In particular, we implemented a computational chemistry application that simulates polymerisation kinetics by way of a Markov-chain Monte-Carlo method. The resulting implementation executes faster than all competing software products, while at the same time also being more general. The generative architecture allows us to cover a wider range of chemical reactions and to target a wider range of high-performance architectures (such as PC clusters and SMP multiprocessors).

In summary, our main contributions are the following:

- A software architecture using generative code specialisation for computationally intensive simulations based on Monte-Carlo methods (Section 2).
- A skeleton and a generative specialisation framework for Monto-Carlo methods and its instantiation to a Markov-chain Monte-Carlo simulator for polymerisation kinetics (Section 3).
- Foreign language plugins for Haskell (Section 4).
- A parallelisation strategy for Markov-chain Monte-Carlo methods (Section 5).
- A detailed performance evaluation of our Monte-Carlo simulator for polymerisation kinetics, which shows that the application of methods from functional programming can lead to code that is significantly more efficient than what can be achieved with traditional methods and imperative languages (Section 6).

As already mentioned, we build on a host of previous work from generative programming, partial evaluation, runtime code generation, and dynamic code loading. We discuss this related work as well as other work on polymerisation kinetics in Section 7.

2. A generative code specialisation architecture

Simulations based on Monte-Carlo methods are very popular in the study of complex systems with a large number of coupled degrees of freedom, this includes applications ranging from computational physics (e.g., high energy particle physics) to financial mathematics (e.g., option pricing). The underlying principle is the *law of large numbers;* that is, we can estimate the probability of an event with increasing accuracy as we repeat a stochastic experiment over and over. This principle can be applied to any system that we can model in terms of *probability density functions (PDFs).* This includes the numerical approximation of many purely mathematical constructs with no apparent stochasticity or randomness, such as approximating the value of π or the numerical integration of complex functions [17].

Monte-Carlo methods use probability density functions to drive the sampling during a simulation. This can be the repeated evaluation of a function at random points, for example to integrate it numerically, or it can be a sequence of system state changes, each of which occurs with a certain probability. As an example of the latter, consider studying reaction kinetics in computational chemistry, where the system is characterised by the likelihood of each simulated chemical reaction in a given system configuration.

In any case, to exploit the law of large numbers, all Monte-Carlo simulations have to repeat one or more stochastic experiments a large number of times as well as tallying the many results, and possibly continuously evolving some system state and compute variance reduction information. In fact, with increasing complexity of the system and with increasing need for precision, more and more stochastic experiments need to be performed. This highly repetitive nature of Monte-Carlo simulations is one of the two key points underlying the software architecture that we are about to discuss.

The second key point is that, in many application areas, Monte-Carlo simulations should come with a large configuration space. For example, in reaction kinetics, we would like to handle a wide variety of chemical reactions and, in financial modelling, we would like to model complex of financial products. In fact, to explore new chemical processes and new financial products, we definitely need to (a) repeatedly alter configurations and experimentally explore a design space; and (b) have short turn-arounds in an interactive system. We certainly cannot afford to modify the source code of our simulator—in fact, many users in application areas using Monte-Carlo simulation won't have the expertise to even recompile the simulator. On the other hand, when the user found a point in the design space that they want to simulate in more detail, a simulation may run for hours or even days.

In summary, the two crucial properties of Monte-Carlo simulations guiding the following discussion are thus:

- *Property 1:* The simulation repeats one or more stochastic experiments and associated book keeping a large number of times.
- *Property 2:* It simulates complex systems with a large number of degrees of freedom and a rich configuration space.

2.1 The classical approach: a simulator in C, C++, or Fortran

As discussed, Monte-Carlo methods are typically used in complex systems with a large number of degrees of freedom, where they require a many repetitions of a stochastic experiment to achieve numeric accuracy. For example, sophisticated simulations in the domain of polymerisation kinetics can take days to execute on a uniprocessor. Hence, manually optimised simulator code in a low-level languages like C, C++, and Fortran is the state of the art, and the use of functional languages in many domains where Monte-Carlo simulations are applied is out of the question, unless the same level of performance can be achieved. Even a 50% increase in running time is not acceptable when a program runs for days.

While Property 1 encourages the use of a low-level language, the amount of optimisations that can be performed in a low-level language is limited by Property 2. A simulator in a low-level language must be sufficiently generic to handle a large configuration space in which it has to evaluate functions with a large number of inputs. In other words, the code in the repeatedly executed in-

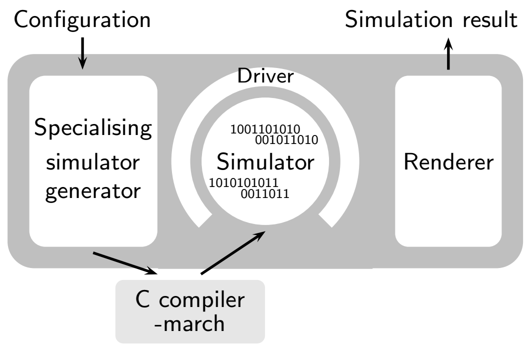

Figure 1. Architecture for generative simulators

ner loop will be complex and possibly traverse sophisticated data structures. However, given the number of repetitions, each cpu cycle counts significantly towards the final running time. Additional instructions required to implement a more general solution lead to notable inefficiencies compared to specialised implementations.

To illustrate this situation, consider reaction kinetics again. Each reaction occurs with a probability that depends on the relative concentration of the various reactants. If it occurs, it will consume one or more reactants and release new reactants into the solution. A Monte-Carlo simulator will have to keep track of these concentrations and the associated reaction probabilities. It has to select reactions according to the implied probability distribution function. The reactions have to be modelled in a data structure and the more variations we allow, the more interpretative the process in the inner loop will be. In other words, the larger the configuration space, the slower the simulator.

This situation calls for a generative approach. The simulator has a large configuration space and its inner loop will be executed many times for a single configuration, making it worthwhile to specialise the inner loop for one configuration, effectively giving us a custom simulator for one problem, before executing it. This specialisation has to be transparent to the user and, as we discussed previously, has to occur in an interactive environment. Hence, we propose the use of online generative code specialisation: that is, in dependence on user input, the application specialises its inner core to produce highly optimised code and dynamically loads and links that code into the running application. As we will see in Section 3.4, depending on the type of Monte-Carlo method used, we may have to generate one or more specialised simulators per executed user-level simulation.

2.2 From Haskell to C to a C generator

Writing a new Monte-Carlo simulator using this architecture is a three-step process:

1. Implement a *prototype simulator* in a functional language like Haskell as an executable specification and to explore alternative designs.
2. Manually specialise for one or more concrete parameter settings and translate into C to explore possible low-level optimisations. This involves selecting appropriate imperative data structures. The C program, which we call the *specialised simulator*, is validated against the functional prototype. (Instead of C, languages like C++ or Fortran may of course also be used.)
3. Rewrite the prototype simulator as a *simulator generator* in the functional language, such that when fed with the same parameter settings it produces the specialised generator we manually implemented in the previous step.

The generator is actually a three-tier architecture. At the bottom, we have a set of generic combinators producing C constructs, such as type declarations, initialisers, control structures, and so forth. These are reusable between different simulators. On top of the generic combinators, we implement domain-specific combinators. For example, for polymerisation kinetics, we have combinators creating the representation of a reaction or applying a reaction. We implement the simulator generator with these domain-specific combinators.

The code generated by the simulator generator can be validated against the functional prototype and the C implementation. The later is often more convenient for debugging purposes, as it is easier to compare the internal states of the two implementations.

The concrete specialisation strategy is, in fact, domain-specific and can require a rather complicated separation of specialisation time versus runtime data. In Section 3, we will come back to this point and describe two general specialisation strategies as well as detail the strategy that we used for the polymerisation kinetics. For the moment, it suffices to say that in the simulator for polymerisation kinetics, we specialise on the possible reactions and their probabilities in dependence on the concentration of the reactants. This forms the basis for the computation of the probability distribution functions in this application.

Figure 1 illustrates the overall architecture with the simulator generator at the very left. The generated C code is passed to a *driver* that we discuss next.

2.3 Runtime compilation, loading, and simulator execution

The driver in the middle of Figure 1 is responsible for turning the C source of the specialised simulator into executable code, running that code, and passing the results to the third component of the architecture, the *renderer.*

Unlike specialising generators for specific problems, such as FFTW [7], we generate machine-independent C code, rather than native code. Our experience is that, while implementing the specialising simulator generator, we are still exploring a significant design space for the core data structures and algorithm. To do this, we need a light weight approach to code generation, that makes it as simple as possible to perform structural changes to the generated code. Our three-tier architecture for the simulator generators simplifies this process.

To compile the C code, we use a standard C compiler, such as GNU's `gcc` or Intel's `icc`. We require high-performance, so we rely on state-of-the-art optimising C compilers that may vary with machine architectures. In fact, access to low-level and architecture-specific optimisations is another important reason for generating C code, rather than attempting to directly generating native code. For example, it would be extremely difficult to beat `icc` with the instruction scheduling for floating-point computations on Intel processors. Given the long running times of typical Monte-Carlo simulations, we can even amortise time consuming compilation with costly optimisations enabled. To generate good code, the driver passes simulator-specific and hardware-specific flags to the C compiler. Another option to improve the compiled code further, which we have only lightly explored, would be to use tools like Acovea [12], a simulated annealer, to breed a best fit for the compiler flags.

In addition to the simplification of data and control structures explicitly performed by the specialising simulator generator, the C compiler can exploit the fact that many of the variables of the generic simulator are now embedded as constants. This leads to additional constant folding, loop unrolling, and loop vectorisation.

At this point it would be possible to run the generated simulator as a separate process. However, we found it advantageous to dynamically load and link the compiled simulator into the main appli-

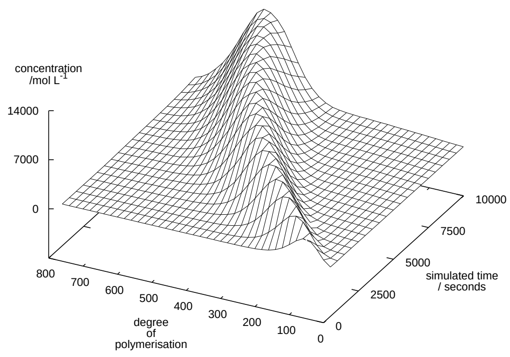

Figure 2. Molecular weight distribution during polymerisation

cation. This is especially valuable during the exploration and design phase of the simulator user. The loaded code can be more tightly integrated with the renderer, which may for example graphically animate a running simulation. Moreover, depending on the specific Monte-Carlo method, the generated simulator will still have runtime arguments and be executed repeatedly by the driver on a range of argument values. We will return to this point in Section 4, where we discuss the foreign language interface that connects the compiled C code of the simulator with the main application.

2.4 Processing results

The results of a simulation can, of course, be used in a variety of ways. In particular, they can be visualised online or offline. In an exploratory context, a user may run a few approximate short running simulations that are visualised online, until they find a system configuration that they want to explore in more detail. At this point, they will start a long running simulation whose data set will be dumped to disk. An example visualisation from our simulator for polymerisation kinetics is in Figure 2. An important property in polymerisation is the concentration of molecules with particular chain lengths of a synthesised polymer. The given graph plots this concentration in dependence with the simulated time elapsed since the start of the chemical process, where the chain length is indirectly given by the molecular weight of the polymer molecules.

2.5 Summary

In summary, Figure 1 outlines an architecture for implementing high-performance Monte-Carlo simulators as lightweight, interactive applications. Instead of achieving high-performance by implementing in a low-level language, we achieve even higher performance by generating specialised low-level code in a functional language. In the next section, we will see that this approach applies to a wide range of Monte-Carlo methods and that functional languages are a good match to the task of implementing generative code specialisers.

3. Generative Monte-Carlo Methods

After covering specialising simulator generators in the previous section, we are now going to discuss the generative specialisation of Monte-Carlo methods at three concrete examples. We will see that for specialisation to be worthwhile, simulations need to (a) perform sufficiently complicated operations during stochastic experiments and (b) have a number of input parameters that are fixed for at least part of one simulation. We call such methods *generative Monte-Carlo methods*. Given that Monte-Carlo methods are typically used for complex systems with a large number of degrees of freedom, practically useful Monte-Carlo methods usually meet these two criteria. To clarify the general structure of Monte-Carlo methods and to highlight the points of opportunity for specialisation, we will discuss a general Monte-Carlo simulation skeleton after the concrete examples.

3.1 Computing π

The probably simplest Monte-Carlo method is that to compute an approximation of π. We know that the area of a circle is $A = \pi r^2$. Hence, $\pi = A/r^2$; i.e., $\pi/4$ is the *probability* that a point picked at random out of a square of side length $2r$ is within the circle enclosed by that square. As explained in the previous section, the fundamental idea underlying Monte-Carlo methods is to *estimate the probability of an event with increasing accuracy by repeating a stochastic experiment over and over.* Here the stochastic experiment is to pick a point in the square at random, and we use that experiment to approximate the probability that picked points lie inside the circle. By multiplying that approximated probability with 4, we approximate π.

The following Haskell function `piMC` implements this idea. It's first argument is a stream of random `Doubles` from the interval $[-1, 1]$ and the second one is the number of random points to be picked.

```
piMC :: [Double] -> Int -> Double
piMC rs n =
  let within = [ point
               | point <- take n (points rs)
               , inCircle point]
      hits   = fromIntegral (length circlePoints)
  in
  hits / fromIntegral n * 4
  where
    points (x:y:rs) = (x, y) : points rs
    inCircle (x, y) = sqrt (x*x + y*y) <= 1.0
```

Although, the computation of π illustrates the Monte-Carlo principle very well, it is not a particularly efficient method of computing π. It is also not really amenable to code specialisation. After all, the only variables in the code are the number of stochastic experiments to perform and the random values. For specialisation to be of use, a simulation must be parametrised by a significant number of parameters, which may be numeric or have a complex structure. We call methods, such as that computing π, *simple Monte-Carlo methods*. Simple methods perform little work per stochastic experiment and have few, or no, configuration parameters.

3.2 Modelling financial products

Monte-Carlo methods are routinely used in finance to solve problems, such as the arbitrage-free pricing of derivatives [10]. Such problems can often be expressed in terms of partial differential equations, but when the degrees of freedom (i.e., the dimensionality) is large, non-stochastic methods often converge slowly and require large amounts of memory. If the desired value (i.e., the valuation of a derivative) can be expressed as the probability of a stochastic experiment, a standard alternative is the use of a Monte-Carlo method.

The basic idea is the same as in the previous computation of π, but there are two big differences:

1. Where the only arithmetic in the computation of π is the Pythagoras test $\sqrt{x^2 + y^2} \leq 1$, *a single stochastic experiment* in valuating complicated derivatives involves already very complex computations. Moreover, these derivatives usually in-

```
data Config = Config {  -- simulation configuration:
                samples      :: Int,    -- samples to take over lifetime of stock
                initialPrice :: Double, -- initial stock price
                drift        :: Double, -- percentage drift of GBM
                volatility   :: Double, -- percentage volatility of GBM
                strike       :: Double  -- option strike price
              }

optionMC :: [Double] -> Int -> Double -> Config -> Double
optionMC rs n tT (Config samples stock0 dr vo strike) =
  let pricePaths = [ map (stock tT) path | path <- take n (paths rs)]
      results    = [ 0 `max` ((sum prices / fromIntegral samples) - strike)
                   | prices <- pricePaths]
  in
  exp (-dr * tT) * sum results / fromIntegral n
  where
    -- sets of random variables of variance tT and mean 0
    paths rs = let (path, rs') = splitAt samples rs
               in
               map (sqrt tT *) path : paths rs'
    -- calculate stock price as geometric Brownian motion (GBM)
    stock tT wt = stock0 * exp ((dr - vo * vo / 2) * tT + vo * wt)
```

Figure 3. Monte-Carlo estimate of the price of an Asian option

volve a significant number of dependent assets, each of which has their own variables. So, instead of scalars (or in the case of π, pairs), we have to manipulate entire matrices of parameters for a single experiment.

2. Where a single random point in the approximation of π already gives us the outcome of one stochastic experiment, complicated derivatives usually require us to evolve a set of assets over a time interval in discrete steps; e.g., for an Asian option to sample its price once a week or month.

Let us consider a simple and well-understood form of option, namely an Asian call option. The code for a Monte-carlo estimate of the pricing of an Asian option is in Figure 3. The code uses the local function `stock` to calculate future stock prices on the basis of their initial price and a standard model based on geometric Brownian motion (GDM) that is parametrised by *drift* and *volatility*. The last parameter of `stock`, namely `wt`, is a random variable drawn from a normal distribution with mean 0 and variance `tT`, which is the expiration time of the option. To value an Asian option, we need to sample it a number of times during its lifetime. The number of samples, called `samples` in the code, determines how many random numbers we need to obtain per stochastic experiment. We use the local function `paths` to chop the input stream of random numbers `rs` into sublists of just enough random numbers for each stochastic experiment.

Based on these functions, `pricePaths` calculates the stock prices of the Asian option at all sample points (using one sublist per stochastic experiment). Out of these prices, `results` computes the obtained wins per stochastic experiment—we perform `n` such experiments. The results of all experiments are combined to the final estimate in the body of the `let` expression.

Although this is a simple derivative, we obviously already perform much more work per stochastic experiment than in the approximation of π; in fact, the amount of work depends on the configuration parameter `samples`, which fixes the path length per experiment. The function `optionMC` has a number of configuration parameters, and except `samples`, these parameters are determined by the option that we are modelling. If `initialPrice`, `drift`, `volatility`, and `strike` are known, we can obviously remove some of the arithmetic operations by performing constant folding. Moreover, if `samples` is statically known, we can unroll the recursion (or in a C program, the loop) that traverses the path of each experiment. Given a C version of the option simulator with a hardcoded configuration, any optimising C compiler will perform the constant folding. However, it is already less likely that it will unroll the loop traversing each path.

More complex derivatives consist out of a number of simpler derivatives that are often highly correlated. As a result, even computing properly distributed random variables already involves complex matrix operations. Moreover, the random variables in one path may form a more complex Markov chain, which requires more sophisticated computations. All in all, the code quickly gets sufficiently complex that the limited optimisations of a C compiler will not be able to adequately specialise the code for a given configuration. In contrast, a specialising simulator generator can still generate a highly specialised form of the simulator, including specialised data structures.

Despite the differences in complexity, the actual Monte-Carlo method used in `optionMC` is essentially the same as the one we used to approximate π. We call it a *parallel Monte-Carlo method*, as the individual stochastic experiments—i.e., the computations leading to one element of the `results` list—are independent. We only need to combine the results of the individual stochastic experiments at the end to compute the final result.

3.3 Modelling polymerisation kinetics

Our largest example, a simulator for polymerisation kinetics, is much to big to present the Haskell code in a paper. Instead, we will simply describe the application and our implementation strategy in more detail here. The details from a Chemist's perspective can be found in a companion paper [2].

The kinetics simulator models two classes of molecules: simple molecules and polymers. Simple molecules are uniquely characterised by their name, whereas polymers also have a chain length, and in the case of star polymers, actually a set of chain lengths associated with each molecule. A reaction can consume up to two molecules, and produces up to two new molecules. Reactions that involve polymers are usually parametrised over the chain length.

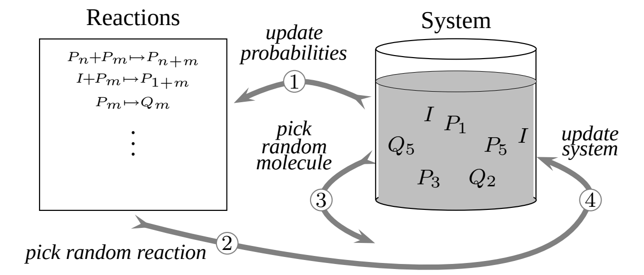

Figure 4. Structure of the Monte-Carlo simulator for polymerisation kinetics

For example, a reaction might consume a polymer P with a chain length n and another of the same type with chain length m, and produce a new polymer molecule of the same type with chain length $n + m$, which we denote here as $P_n + P_m \mapsto P_{n+m}$. The simulation keeps track of the actual number of each type of molecule in the system. A single simulation step consists of the following substeps, as displayed in Figure 4:

1. Determine the probability of each reaction. The probability of each possible reaction is the product of the relative probability and the current concentration of the reactants involved.
2. Randomly pick a reaction according to their probability.

   For example, in Figure 4, the result of this random choice might be the reaction $P_n + P_m \mapsto P_{n+m}$. Note that the reaction is specific w.r.t. the type of molecules involved, but is parametrised by their chain length.
3. Randomly pick the molecules involved in the reaction—-this step is trivial for simple molecules. For polymers, it may have to be taken into account that polymers with different chain lengths react with different probability.

   For the reaction $P_n + P_m \mapsto P_{n+m}$ we have to pick two random chain length n and m. The probability depends on the number of molecules of each chain length currently in the system, adjusted by a factor in case the chain length influences the reactivity of a molecule.
4. Update the system state, that is, the concentration of molecules and time. We have to delete the molecules consumed by the reaction from the system, and the product of the reaction, and increment the system clock.

   For our example reaction, this means that we need to remove one molecule of each P_n and P_m from the system and add P_{n+m}. When adjusting the system time, we need to take the size of the system, i.e., overall number of molecules, into account, as well as the probability of the reaction. Since we have, on average, ten times more reactions in a system with ten million molecules than with one million, the increment of the time stamp in the larger system would, for the same reaction, only be one tenth of that of the smaller system.

For each of these substeps, it is essential that the representation of the system state supports the following operations very efficiently:

- The mapping of a reaction to the reactants involved.
- The mapping of a molecule type to the number of molecules of that type in the system.
- Updating the molecule count.

In the Haskell prototype of the simulator, both molecules and reactions are modelled using parametrised user-defined data types. However, our specialising simulator generator for polymerisation kinetics encodes the set of possible reactions and molecules into the specialised simulator. Hence, in the emitted C code, reactions and molecules are simply represented by scalar values, or an array of scalars in the case of star polymers. The lookup tables can then be implemented as simple arrays.

Moreover, the Haskell prototypes makes extensive use of pattern matching to process the data structures. In contrast, the specialised simulator generator turns this into simple C switch statements over the scalar values representing the various molecules and reactions. Note that this isn't easy to achieve in a generic C simulator that reads a molecules and reactions configuration at runtime, as C switch statements can only use constants as case labels. We will come back to this point when discussing the benchmark results in Section 6. However, the specialising simulator generator is still written in Haskell and it still uses all the conveniences of parametrised user-defined data structures and pattern matching to represent configurations and to traverse them to generate the C code. Hence, we combine not only generality with high-performance, but we also maintain programmer convenience.

The more molecules we have in the system, the more accurate the result of the simulation will be, and the more reactions we have to simulate for a fixed amount of system time. Another way to improve the quality of the result is to run the same simulation several times and calculate the average. However, it is important to keep in mind that running the same simulation ten times will not, in general, lead to a result of the same quality as a single simulation of a system ten times the size, since the concentration of some of the reactants is so low, that, for small systems there would be less than one molecule available and the fact that the simulation is discrete would distort the result.

Generally, we call Monte-Carlo methods that evolve a system state sequentially as a chain of stochastic experiments *sequential methods*.[1] As just discussed, we can sometimes split one long chain into a few parallel, but shorter chains.

3.4 A Monte-Carlo skeleton

Monte-Carlo methods come in a large number of flavours and variants, some of which are not widely publicised outside the domain in which they are used. This makes it difficult to generalise over all Monte-Carlo methods. However, we believe that the function `QC` from Figure 5 captures many, maybe most, Monte-Carlo methods commonly used. The function is parametrised by a type of systems configurations that is a member of the type class `Config` of *Monte-Carlo simulator configurations.* Before we look at the type class in more detail, it is useful to consider the general structure of the skeleton. It nests three levels of standard recursive traversals:

1. *Configuration space:* The outermost recursion is a simple map over the list of configurations `cfgs`. It is realised by the list comprehension forming the body of the function `mc` and applies the local function `simulate` to each configuration.
2. *Number of trials:* The middle recursion is again a map; this one is over the number of random generator `seeds` requested by the current configuration `cfg`. It is realised by the list comprehension in `simulate`, and the number of *seeds* is determined by `chains cfg` in `pickSeeds`.
3. *Chain length:* The innermost recursion is realised by an `unfoldr` in the body of `simulate`'s comprehension.[2]

[1] Our definition of sequential methods does not exactly coincide with the term "sequential Monte-Carlo" as found elsewhere in the literature.

[2] Unfolding is part of Haskell's standard list library: `unfoldr :: (b -> Maybe (a, b)) -> b -> [a]`.

```
class Config cfg where
  type Result cfg                 -- result of individual experiments
  type Final  cfg                 -- combined result of simulation at multiple configurations

  chains     :: cfg -> Int        -- number of chains

  experiment :: (cfg, [Double])   -- one stochastic experiment
             -> Maybe (Result cfg, (cfg, [Double]))

  average    :: cfg -> [[Result cfg]] -> [Result cfg]    -- combine chain results
  merge      :: [(cfg, [Result cfg])] -> Final cfg       -- merge multiple sumulations

mc :: Config cfg                      -- type of Monte-Carlo configurations
   => [cfg]                           -- set of configurations to simulate
   -> [Seed]                          -- stream of random generator seeds
   -> Final cfg
mc cfgs seedPool =
  merge [(cfg, simulate cfg seeds) | (cfg, seeds) <- pickSeeds cfgs seedPool]
  where
    -- pair each config with as many seeds as there are chains in that config
    pickSeeds []         _     = []
    pickSeeds (cfg:cfgs) seeds =
      let (chainSeeds, seeds') = splitAt (chains cfg) seeds
      in
      (cfg, chainSeeds) : pickSeeds cfgs seeds'

    -- run the simulator on one configuration
    simulate cfg seeds =
      average cfg [unfoldr experiment (cfg, seedToStream seed) | seed <- seeds]
```

Figure 5. Generic Monte-Carlo skeleton

Depending on the concrete Monte-Carlo method, only one or two of the recursions may be non-trivial. For example,the approximation of π and the simple option pricing did not make use of the configuration space. For polymerisation kinetics, the use of a larger configuration space to run multiple simulations, and thereby reduce the system size, is an option, as discussed at the end of Section 3.3. Moreover, in the case of approximating π, the chain length was 1. In contrast, the chain length in our production runs with the polymerisation kinetics is in the order of 10^{10}, but we only execute one trial. In applications of financial mathematics it is common to have multiple trails, each of which has a chain length greater than 1.

Configurations as represented in the type class `Config`, determine a type `Result` for the individual stochastic experiments and another type `Final` for the final result derived from the different simulations for all given configurations. Both types depend on the configuration type `cfg`; i.e., they are so-called associated types [3]. Individual stochastic experiments are performed by `experiment`, which gets a configuration and a stream of random numbers. In addition to a result, it produces a possibly altered configuration and the reminder of the stream of random numbers. The `unfoldr` in `simulate` chains experiments until one returns `Nothing`. Finally, `average` and `merge` combine the results of all trials in a simulation and all simulations in a set of configurations into a compound results.

3.5 What to specialise

In the case of the computational chemistry simulator, we have three input parameters: (1) the types of molecules which occur in the system, (2) a description of all possible reactions and their relative probability, and (3) the concentration of each reactant.

Knowing the type of reactions and molecules involved makes it possible to generate efficient representations of the reactions and the system state. On the other hand, the concentration of the molecules changes constantly, so specialising for the initial state would not actually allow for any additional optimisations. Therefore, the system is only specialised for the type of reactions and molecules. The initial concentration of the reactants is read in separately at runtime.

Splitting the set of input parameters into these two classes is an important step when implementing an application following this architecture. For each parameter, we have to check if (1) knowing the value of the parameter at compile time enables any optimisations (2) the optimisations are worth the additional overhead of generating specialised code. In the chemistry application, this decision is fairly straight forward: even for simple systems, the simulation runs for several minutes, so even minor optimisations pay off.

Considering the Monte-Carlo skeleton from the previous subsection, we see that some simulators run multiple simulations on a set of configurations. Depending on the variation between these configurations, it may be worthwhile to generate multiple specialised simulators.

4. Foreign language interface for plugins

Besides the specialising simulator generator, the other component of our architecture from Figure 1 that warrants some attention is the driver. The amount of interaction between the main application and the generated simulator varies among domains, but loading the executable code of the specialised simulator into the main application usually simplifies matters. In particular, it simplifies a tight integration between the running simulator and a graphical frontend displaying the evolving simulation. While exploring a design space with many very imprecise, but comparatively short-running simulations, users often want to see the continuous progress of the simulation.

4.1 Dynamic linking

In the polymerisation kinetics application, we use a small, custom dynamic linker to dynamically load and link the compiled C code back into the main Haskell program. It is based on the core of the dynamic linker described in [15] and provides a simple binding to the GHC Haskell runtime, implementing a barebones linking system for C objects, using the four functions:

```
pluginInit    :: IO ()
pluginLoad    :: CString -> IO Bool
pluginResolve :: IO Bool
pluginSym     :: CString -> IO (Ptr a)
```

The entire linker and compilation manager is itself just over 100 lines of Haskell code.

When passed an object handle by the compilation manager, a simulator object is loaded and resolved, and the linker returns a C function pointer to the C simulator, wrapped as a Haskell value. We achieve this by the following function, which use the conversion function `withCString` from the Haskell foreign function interface (FFI).

```
loadAndGetSym :: String -> String -> IO (Ptr a)
loadAndGetSym objFile sym =
  withCString objFile $ \objFileC ->
  withCString sym     $ \symC -> do
    pluginInit
    pluginLoad objFileC
    pluginResolve
    pluginSym symC
```

We might call this with

```
loadAndGetSym "simulator.o" "doSimulate"
```

.

4.2 Foreign evaluation

Once the C pointer associated with the given symbol is retrieved by the linker, control can then pass from Haskell to C, by evaluating the function pointed to by the obtained C pointer. However, we cannot directly coerce the C function pointer to a normal Haskell function value, as normal Haskell functions are represented differently. We can however, throw the function pointer to the C runtime, which can then execute the function. We do this by calling a C wrapper `apply` function, with the function pointer as an argument. The C function `apply` itself is imported into Haskell via the Haskell FFI:

```
foreign import ccall unsafe "apply.h apply"
   apply :: Ptr () -> IO ()
```

where `apply` itself is a function to evaluate its argument:

```
void apply(void (*f)(void)) {
    f();
}
```

Now, we can evaluate a C function by passing it to `apply`. Using the FFI, C functions can also call into Haskell and so realise a two-way connection between the simulator and the renderer.

5. Parallelisation

Processor technology has, at least for the moment, finally hit the limit of its ability to speed up sequential programs by increasing clock rates and similar. Instead, processors increasingly rely on explicit, software-controlled parallelism to achieve speed up over successive generations of architectures—examples, are multi-core and many-core architectures as well as modern GPUs. Hence, any method to solve computationally intensive problems needs to address parallelisation if it is to be of continuing relevance.

Simple Monte-Carlo methods, such as the approximation of π, are almost trivial to parallelise. The large number of independent stochastic experiments can be easily distributed over multiple processors, as long as we ensure that the statistic properties of our source of randomness are robust with respect to parallel execution. The only communication between the parallel threads is at the end of the simulation when the local results need to be combined into a global result. This can be achieved using standard parallel reduction operations (e.g., parallel folds) and will usually not contribute to the overall runtime in any significant way.

5.1 Parallel Monte-Carlo methods

Parallel Monte-Carlo methods, such as the option pricing discussed in Section 3.2, belong to the class of embarrassingly parallel problems. They require only very little communication and synchronisation, and so eliminate one big problem for efficient parallel execution. What remains is adequate load balancing; i.e., for complex options, some stochastic experiments may need more processor time than others. So, in general, given P processing elements (PEs) and N independent stochastic experiments, assigning N/P experiments to each PE may not utilise the parallel machine optimally. However, given the stochastic nature of the algorithm, we expect to see a uniform distribution of experiments of different complexity over all PEs, which leads to good load balancing if N is much bigger than P.

Unfortunately, the situation is often more complicated in practice. Although, approximation by Mont-Carlo methods, in their typical application areas, usually converges faster than deterministic numeric approximations, the standard convergence is for complex problems still not sufficiently fast. In this case, *variance reduction* methods are usually used to improve the convergence behaviour of the algorithm. Some of these methods, such as the additional use of an *antithetic path* and the use of a *control variate*, do not interfere with a parallel implementation. However, others techniques, such as *sequential importance sampling*, prevent us from generating the random variables of the different stochastic experiments independently. In fact, they may be become sequentially dependent.

5.2 Sequential Monte-Carlo methods

Sequential Monte-Carlo methods are, as their name suggests, not straight forward to parallelise, especially if the best way to achieve convergence fast is to keep tracking one path for a large number of steps, as is the case in our polymerisation kinetics simulator.

At the end of Section 3.3, we discussed that for polymerisation kinetics, we can improve the quality of the simulation result in two ways: by increasing the system size, or by repeatedly simulating the same system and averaging over the result. The first approach is not suitable for parallelisation, due to the inherent sequential nature of Markov-chains. However, the second approach, which is trivially parallel, is also problematic for our particular application: as it turned out, the systems the Chemists were interested in had an unfortunate property. In order to get feasible numbers of molecules with the lowest concentration in the system, the system size had to be so big that the simulation took hours or more to run on one processor, and the result of a single simulation already was of sufficient quality. Fortunately, there is a solution to this problem: the kinetics simulation (like all non trivial simulations) makes simplifying assumptions about the laws that determine its behaviour. One of the simplifications in our context is that we abstract over the position of the molecule in the system. If two molecules are far apart, they would, in reality, be less likely to react. Now, we can make use of this and split the system into several subsystems, run the simulation of each subsystems independently in parallel on separate PEs. We only have to make sure that we *mix*—i.e., gather, average, and re-distribute—with sufficient frequency to model the Brownian

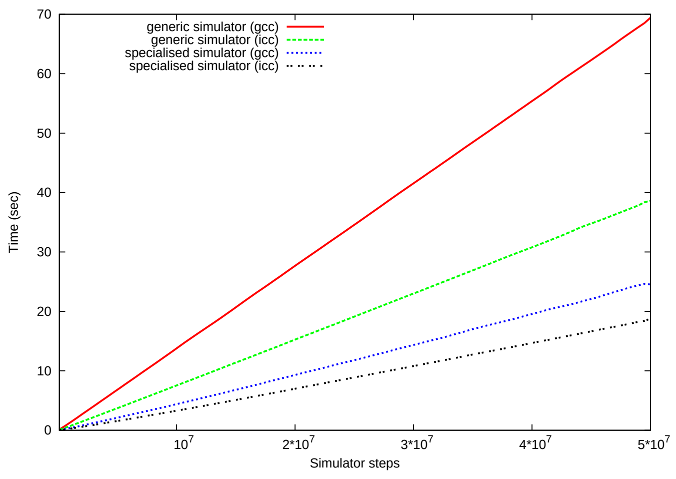

Figure 6. Comparative running times for generic and specialised polymerisation kinetics

motion of the molecules. In this way, we can parallelise the application without compromising the quality of the result. Of course, the speed up will be slightly less than for a trivially parallel Monte-Carlo simulation, as the mixing triggers communication. However, as the benchmarks in the following section show, the parallelisation is still very good.

Although, our use of regularity averaging over a set of parallel executing sequential Monte-Carlo simulations was motivated by the physical intuition of spatial separation and Brownian motion in a liquid, the same approach to parallelisation is more generally applicable to sequential Monte-Carlo. Regular averaging over a set of parallel sequential simulations will improve the accuracy of the intermediate results and so in many cases accelerate convergence.

6. Results

The initial objective of our computational chemistry project was simply to implement a fast, parallel simulator. It needed to be sufficiently generic to support the type of advanced systems, namely star polymers, that the available commercial simulators could not satisfactorily run. The commercial simulators did not directly support reactions involving star polymers, and even though it was possible to work around this restriction, the simulations just took too long. We started by implementing a generic prototype in Haskell. It quickly became clear that a pure Haskell implementation would not be sufficiently efficient, as there would always be a certain overhead. So we implemented a second simulator, in C this time. The Haskell version relied heavily on algebraic data types, higher order functions, and pattern matching. In C, the generic simulator was much more painful to implement, so we opted for a stripped down, less generic and therefore fairly optimised version of the Haskell implementation. Even in C, though, it was clear that the costs for the limited generality were high, which inspired us to explore the described generative approach.

6.1 Performance of sequential simulator

For the benchmarks discussed in this subsection, we simulated a simple styrene RAFT (reversible addition fragmentation transfer) polymerisation, with only linear (single chain) polymers.

Figure 6 compares the running times of the (not fully) generic C simulator with the specialised code generated and loaded by the Haskell framework. As we can see, the specialised code is close to a factor of three faster than the already partially optimised C version.

The reason for the significant performance difference becomes clear when we compare the assembly code generated by the C compiler for the following program fragment, which is part of the actual reaction application code. First, let us have a look at the C code, for one particular reaction type in the generic simulator:

```
int  doReact (..) {
  ...
  /* Disproportionation */
  if (isPolyMol (*mol1) &&
      isPolyMol (*mol2) &&
      ms2               &&
      isPolySpec (*ms1) &&
      isPolySpec (*ms2))
  {
      *noRes = 2;
      initMol (*rmol1, ms1, mol1->len);
      initMol (*rmol2, ms2 ,mol2->len);
      return (1);
   }
  ...
}
```

As we can see, the generic polymer simulator must interpret a data structure representing the various molecule types, handling them according to general rules. This means a wide range of reactions can be simulated. However, there is a performance cost. This performance cost is starkly illustrated by comparing the assembly generated for `doReact()` between the generic simulator and the reaction-specialised version.

In particular, a number of boolean conditions must be satisfied, to perform the reaction. This involves indirections into the generic molecule data structure. When the reaction finally takes place, we must again indirectly update fields of the result structure.

Unsurprisingly, the C compiler[3] produces less than ideal assembly for the generic C code, with less than ideal register use, branches, and too much memory traffic.

```
doReact:
;;;       if (isPolyMol (*mol1) &&
          movl       (%esi), %edx
          movl       (%edx), %edx
          testl      %edx, %edx
          jne        ..B1.19
..B1.5:
;;;       isPolyMol  (*mol2) &&
          movl       (%ebx), %edx
          movl       (%edx), %edx
          testl      %edx, %edx
          jne        ..B1.10
..B1.6:
          testl      %eax, %eax
          je         ..B1.10
..B1.7:
;;;       ms2                 &&
;;;       isPolySpec (*ms1) &&
          movl       (%ecx), %edx
          testl      %edx, %edx
          jne        ..B1.10
..B1.8:
;;;       isPolySpec (*ms2)) {
          movl       (%eax), %edx
          testl      %edx, %edx
          jne        ..B1.10

;;;         /* Disproportionation */
..B1.9:

;;;         *noRes = 2;
          movl       36(%esp), %edx
```

[3] Intel C compiler, which outperformed GCC in our tests.

```
        movl      $2, (%edx)

;;;       initMol (*rmol1, ms1, mol1->len);
        movl      28(%esp), %edx
        movl      %ecx, (%edx)
        movl      4(%esi), %ecx

;;;       initMol (*rmol2, ms2 ,mol2->len);
        movl      32(%esp), %esi
        movl      %ecx, 4(%edx)
        movl      %eax, (%esi)
        movl      4(%ebx), %eax
        movl      32(%esp), %ebx
        movl      %eax, 4(%ebx)

;;;       return (1);
        movl      $1, %eax
        popl      %ebx
        popl      %esi
        ret
```

The situation is vastly different once we specialise doReact() body to the reaction at hand. Rather than employ generic structures and logic for interpreting the reaction rules, we are able to encode the specific, statically-known reaction cases and results as C enumerations. As a result we get an efficient implementation of the doReact () logic as a switch. The various results are statically known, and may also be encoded in simple atomic types, leading to much improved register utilisation. Here we see the online-generated code for the same reaction as above, specialised for a particular system:

```
switch (reactionIndex) {
   ...
   case dispro:
     no_of_res = 2;
     spec1_ind = D;
     spec2_ind = D;
     rl1 = mol1_len;
     rl2 = mol2_len;
     break;
   ...
```

This code fragment is compiled to almost optimal assembly, a simple vectored jump for the switch, and 4 instructions to perform the reaction:

```
        movl      ..1..TPKT.11_0.0.10(,%ebp,4), %edx
        jmp       *%edx

..1.11_0.TAG.5.0.10:
..B7.26:
        movl      $2, %edx
        movl      $4, %ebp
        movl      $4, %ecx
        jmp       ..B7.38
```

Comparing the two assembly code fragments, it becomes obvious why the generated code easily outperforms the hand written generic C simulator as shown in Figure 6.

The Monte-Carlo method we used for our implementation has been applied before, for example in [6]. The specialised version of our generator outperforms the implementation described in [6] by a factor of 2.6 on an AMD Athlon 64 3200+. That is, the performance in [6] is comparable to that of our generic simulator.

6.2 Performance of parallel implementation

Let us now have a look at the performance of the parallel code. Parallelism is implemented using standard MPI [8] library calls. As MPI libraries are available for virtually all multiprocessor machines, the code can run on most architectures. Figure 7 and 8 show the absolute running times and speedups respectively of the specialised, parallel simulator code, both on a cluster and a shared memory machine. The cluster is a commodity Linux cluster, containing a mixture of 2.6Ghz and 3.2Ghz Pentium 4S processors. The shared memory machine contains eight AMD Athlon 64 3200+, 2.2Ghz processors. Even though they run on a lesser frequency, they are faster than the cluster nodes, as they have a bigger first and second level cache, and higher memory bandwidth. We ran both simulations with 10^9 and 10^{10} on both architectures. Due to the better hardware architecture, the shared memory machine outperformed the cluster on both benchmarks (Figure 7). Both architectures show excellent speedup for the 10^{10} benchmark. The 10^9 system is too small for the communication and synchronisation overhead to pay off (Figure 8 & Figure 9).

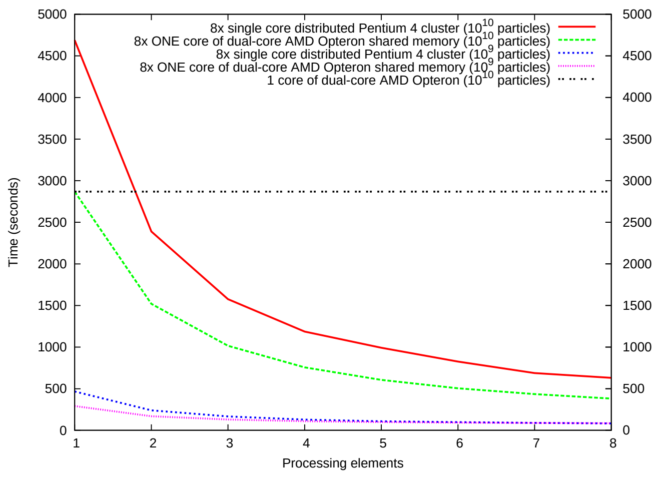

Figure 7. Run time of generated simulator on cluster and shared memory systems

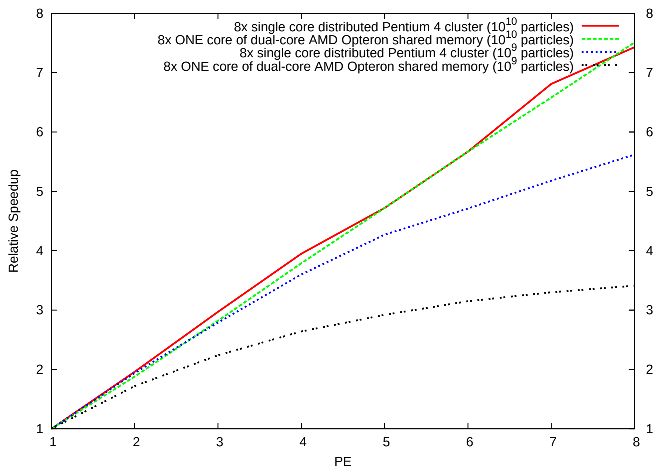

Figure 8. Speedup on shared memory and cluster systems

6.3 Comparison with PDE Solver

Finally, we compared the running time of of various specialised polymer simulators against the leading commercial package PREDICI (version 6.36.1, produced by Computing in Technology (CiT) GmbH) [23, 22], running the same reaction. PREDICI is not a Monte-Carlo simulator—it calculates the distributions by solving

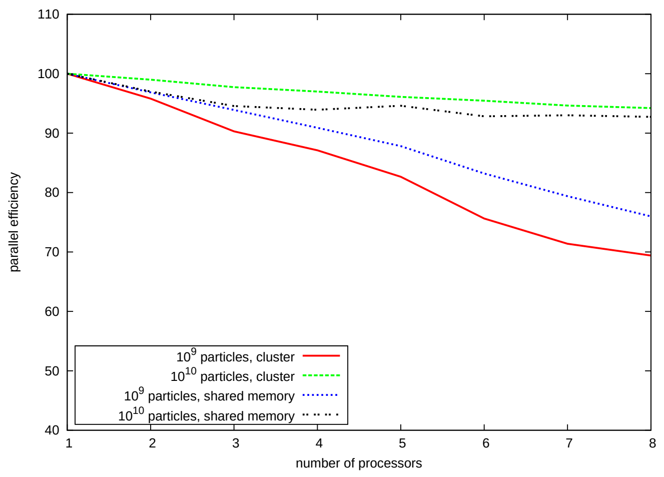

Figure 9. Comparative efficiencies of shared memory and distributed simulations

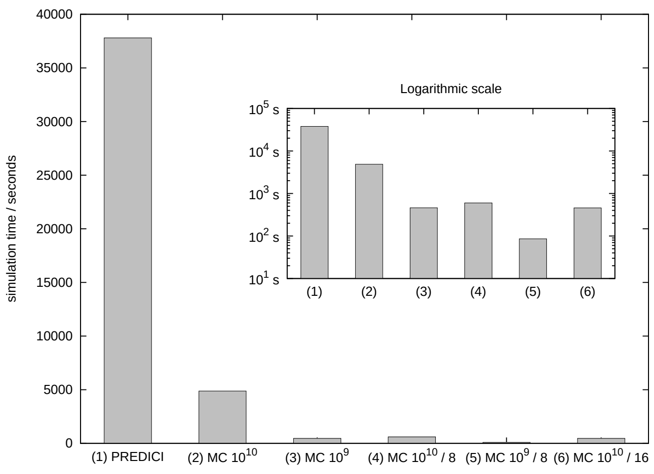

Figure 10. Comparative running times for commercial PDE versus our parallelising MC simulator

the partial differential equations (PDE) that describe the system and the reactions. Therefore, the result of a PREDICI run can be regarded as the correct distribution (as modelled by the system), not an approximation. For 10^{10} particles, the simulation results were identical to those of PREDICI. For 10^9 particles, the results diverged when we used more than eight processors. We used a more complicated system star RAFT system for the benchmarks in this subsection. For the simple styrene RAFT that we used for the previous set of benchmarks, PREDICI's performance is comparable to the Monte-Carlo simulator in [6]; i.e., our specialised simulators still outperform it by more than a factor of two. On even simpler systems, PREDICI is more efficient then our Monte-Carlo system. This is not surprising, as PDE solvers usually perform better than Monte-Carlo simulators for simple systems. Monte-Carlo simulators show their strength when the simulated systems get more complex and the degrees of freedom increase.

We compared PREDICI against the specialised simulator, compiled with the Intel C compiler. Moreover, we ran our parallelised simulator on a variety of cluster configurations, using the MPI implementation MPICH [14], with varying particle counts. The columns of Figure 10 list the results for the following benchmarks:

1. PREDICI, on a 3.4Ghz single core Windows XP machine
2. Specialised polymer MC simulator, on a 3.2Ghz Linux Pentium 4, 10^{10} particles
3. Specialised polymer MC simulator, on a 3.2Ghz linux Pentium 4, 10^9 particles
4. Specialised polymer MC simulator, 8 node cluster, 10^{10} particles,
5. Specialised polymer MC simulator, 16 node cluster, 10^{10} particles,
6. Specialised polymer MC simulator, 8 node cluster, 10^9 particles.

For a fair comparison with PREDICI, we need to consider the runs with the 10^{10} particles. Figure 10 shows the results on a linear as well as a logarithmic scale. Our specialised Monte-Carlo simulator clearly outperforms PREDICI already with one cpu. Moreover, our Monte-Carlo simulator scales well with increasing node count and so is able to exploit current clusters and the emerging many-core architectures. Parallelisation of PDE solvers is not an easy task. This in combination with the fact that the advantage of Monte-Carlo methods increases with the complexity of the simulated molecules and reactions makes Monte-Carlo methods a very attractive option in numerically intensive applications that can be modelled by stochastic experiments. On top of this, our novel generative approach to Monte-Carlo simulators improves the running time significantly while simultaneously supporting a wider range of chemical systems—e.g., our system is the only one that can directly simulate star polymers.

7. Related Work

7.1 Partial evaluation

Partial evaluators for C, such as C-Mix [1], has similar objectives as the work presented here, namely to achieve high-performing, yet easily maintainable code. It is also similar in that it relies on the C compiler to apply standard optimisations which have been enabled by the specialisation. However, there are also a number of significant differences: In contrast to our approach it is neither possible nor necessary in C-Mix to provide domain specific specialisation information to the tool. This can be an advantage, since the implementer of the generic code need not be concerned about possible specialisations. However, as the computational chemistry example demonstrates, it also has serious drawbacks: the most effective optimisations in the specialised code were a consequence of the customised data structures and jump tables used, both of which could not have been automatically deducted from the generic simulator. Furthermore, the use of C-Mix is not transparent to the user. In our approach, the user does not need to have the source code of the generator, and (as long as there is a C compiler available on the system) needs not be aware at all that the specialisation is happening behind the scenes because the C code is compiled and dynamically linked back into the executable of the simulator. Also, implementing the fully generic simulator in C would have been a major effort compared to the Haskell implementation—as mentioned before, we heavily relied on higher-order functions, pattern matching and algebraic data types, all language features not or not particularly well supported in C.

Other partial evaluators, such as Tempo [4], focus on different application areas, such as systems programming.

Using C++ templates as a substrate for partial evaluation, as in [18], allows the addition of domain specific information, and it would be interesting to investigate if it is possible to get similar results as we have with our approach. However, it would definitely be necessary to push the limits of C++ template programming, a technique which can be fairly tricky and error prone. As with C-Mix, however, the user of the simulator would have to have access to the source code, and instantiate, compile and link the program.

7.2 Generative programming

Generative programming also targets a similar problem, and, for example [21], employ a meta-language to describe the generating component, whereas we use an existing general purpose language, Haskell, which because of its support of higher-order functions and user defined operators, is a good substrate for the simple EDSL we use.

FFTW [7] shares with our approach, the idea to use a functional language to generate highly optimised low-level code. In fact, FFTW has clearly been an inspiration in that matter. This is, however, where the similarity ends. FFTW provides a library, whereas we presented an application architecture. FFTW also used dynamic code optimisation, whereas our approach is purely static. Furthermore, the type of specialised algorithms are rather different.

7.3 Polymerisation kinetics

The Monte-Carlo strategy we use for the simulation of the polymerisation models is similar to the one method used by [16], [6], [13], and [9]. Although [6] use multiple machines to run independent simulations at the same time, to increase the accuracy of the result, no one has implemented a parallel version of a single simulation. In Subsection 6.2, we compared the performance of [6] with our system.

8. Conclusion

We presented a software architecture based on specialising simulation generators for numerically intensive simulations employing Monte-Carlo methods. We discussed how various Monte-Carlo methods can be mapped to this design, including a detailed description of a polymerisation kinetics simulation. We benchmarked the polymerisation kinetics simulator in detail and found that it is much fast than all competing software products. At the same time it handles a wider range of chemical processes. In other words, we use functional programming to reconcile high performance and generality in simulators based on Monte-Carlo simulation. Moreover, we have successfully parallelised the kinetics simulation, which required a new technique as the simulation is based on a Markov-chain. This makes out approach future proof in the light of the current trend in hardware to add extra parallelism instead of increasing single-threaded performance.

References

[1] H. M. Arne J. Glenstrup and J. P. Secher. C-Mix – specialization of C programs. In *Partial Evaluation: Practice and Theory*, 1999.

[2] H. Chaffey-Millar, D. B. Stewart, M. Chakravarty, G. Keller, and C. Barner-Kowollik. A parallelised high performance monte carlo simulation approach for complex polimerization kinetics. *Submitted to Macromolecular Theory and Simulations*, 2007.

[3] M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. In *ICFP '05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 241–253, New York, NY, USA, 2005. ACM Press.

[4] C. Consel, L. Hornof, J. L. Lawall, R. Marlet, G. Muller, J. Noy, S. Thibault, and E.-N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys*, vol 30(no 3), September 1998.

[5] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[6] M. Drache, G. Schmidt-Naake, M. Buback, and P. Vana. Modeling RAFT polymerization kinetics via Monte Carlo methods: cumyl dithiobenzoate mediated methyl acrylate polymerization. *Polymer*, 2004.

[7] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on ”Program Generation, Optimization, and Platform Adaptation”.

[8] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference*, volume 2—The MPI-2 Extensions. The MIT Press, second edition, 1998.

[9] J. He, H. Zhang, and Y. Yang. Monte carlo simulation of chain length distribution in radical polymerization with transfer reaction. *Macromolecular Theory and Simulation*, 4:811–819, 1995.

[10] P. Jäckel. *Monte Carlo methods in finance*. John Wiley and Sons, 2002.

[11] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, Hemel Hempstead, Hertfordshire, UK, 1993.

[12] S. R. Ladd. Acovea (analysis of compiler options via evolutionary algorithm). `http://www.coyotegulch.com/products/acovea/`, 2007.

[13] J. Lu, H. Zhang, and Y. Yang. Monte carlo simulation of kinetics and chain-length distribution in radical polymerization. *Macromolecular Theory and Simulation*, 2:747–760, 1993.

[14] MPICH. Argonne national laboratory. `http://www-unix.mcs.anl.gov/mpi/mpich/`, 2007.

[15] A. Pang, D. Stewart, S. Seefried, and M. M. T. Chakravarty. Plugging Haskell in. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 10–21. ACM Press, 2004.

[16] S. W. Prescott. Chain-length dependence in living/controlled free-radical polymerizations: Physical manifestation and monte carlo simulation of reversible transfer agents. *Macromolecules*, 36:9608–9621, 2003.

[17] C. Robert and G. Casella. *Monte Carlo Statistical Methods*. Springer Verlag, second edition, 2004.

[18] T. L. Veldhuizen. C++ templates as partial evaluation. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 13–18, 1999.

[19] T. L. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*, 1998.

[20] T. L. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM Press, 1998.

[21] E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[22] M. Wulkow. The simulation of molecular weight distributions in polyreaction kinetics by discrete galerkin methods. *Macromolecular Theory Simulation*, 5:393–415, 1996.

[23] M. Wulkow. Predici. `http://www.cit-wulkow.de/tbapred.htm`, 2007.