Implementing deterministic declarative concurrency using sieves

Implementing deterministic declarative concurrency using sieves

Sam Lindley
University of Edinburgh

ABSTRACT

The predominant thread-based approach to concurrent programming is bug-prone, difficult to reason about, and does not scale well to large numbers of processors. Sieves provide a simple way of adding deterministic declarative concurrency to imperative programming languages. Sieve programs have a straightforward semantics, are not significantly more difficult to reason about than sequential imperative programs, and should scale to large numbers of processors as well as different processor architectures.

1. INTRODUCTION

Multicore processors are becoming mainstream. In the first quarter after its release, Intel shipped over six million multi-core Core Duo 2 processors [4]. In order to take advantage of multicore hardware, software developers must adapt the way they write code, ideally without having to throw away all of their existing imperative code.

Codeplay's *sieve* system [6] gives a way of extending existing imperative programming languages with a form of deterministic declarative concurrency, without requiring developers to rewrite all of their existing codebase. Codeplay has extended their vectorising C++ compiler to support sieves. This paper gives a brief overview of sieves through an operational semantics for a core language of sieves CoreSieve and presents some preliminary benchmarks for Codeplay's implementation of sieves in C++.

Sieve programs support *declarative concurrency* in that they allow the programmer to specify *what* code should be executed concurrently as opposed to *how* code should be executed concurrently. Declarative concurrency has the advantage that it allows code to be automatically tailored to a particular target architecture, at compile time, at run time or both. For instance, Codeplay's implementation of sieves in C++ compiles the same source code into a choice of binaries, which can take advantage of one CPU, multiple CPUs, or even a PPU (physics processing unit).

Sieve programs are *deterministic* in that they have the same observable behaviour no matter what architecture they are executed on. Determinism ensures that sieve programs can be robustly debugged on one architecture (e.g. a single core machine) and deployed on another (e.g. a multi-core machine with a PPU).

The key component of sieve programs is the *sieve block*. A sieve block has the same semantics as any other block of sequential code, except that any side-effects inside the block are delayed until the end of the sieve block. Delaying side-effects allows the compiler to use a straightforward dependency analysis to automatically partition sieve blocks into fragments that can be safely executed in parallel. Execution of a sieve block produces a queue of side-effects. Once the sieve block has finished executing, the queue of side-effects is executed sequentially on a single processor.

2. OPERATIONAL SEMANTICS

In order to provide a formal basis for sieve programs, we present the syntax and dynamic semantics of a core imperative language CoreSieve extended with sieves.

$$\begin{array}{ll} \text{(expressions)} & e, e' ::= i \mid x \mid f(e) \mid e + e' \mid e - e' \mid e \times e' \\ \text{(programs)} & P, Q ::= e \mid \mathsf{int}\ x = e \mid x = e \mid x := e \\ & \quad \mid \mathsf{fun}\ f(x) = P\ \mathsf{in}\ Q \\ & \quad \mid \mathsf{sievefun}\ f(x) = P\ \mathsf{in}\ Q \\ & \quad \mid \mathsf{print}\ e \mid P; Q \mid \mathsf{sieve}\ \{P\} \end{array}$$

Comparisons, iteration and recursion are omitted from the language, but are straightforward to add. Integer constants are ranged over by i and variables are ranged over by f, x. For simplicity we assume that all variables have distinct names, and well-formed programs do not have free variables. Expressions are integers, variables, function applications and the usual arithmetic operators excluding division. Division is omitted in order to avoid having to handle the possibility of division by zero.

Programs are constructed from expressions, assignments, print commands, function definitions, sequential composition and sieve blocks. There are three assignment operators:

- $\mathsf{int}\ x = e$ declares a new variable x and assigns the value of the expression e to x.
- $x = e$ performs an *immediate assignment* of the value of the expression e to the *immediate variable* x.
- $x := e$ performs a *delayed assignment* of the value of the expression e to the *delayed variable* x.

The function definitions fun $f(x) = P$ and sievefun $g(y) = Q$ define respectively standard functions and *sieve functions*. The command print e prints the value of the expression e. The sequential composition $P; Q$ runs P followed by Q. The sieve block sieve $\{P\}$ introduces a new scope $\{P\}$ such that side-effects are "sieved out" of P. Running sieve $\{P\}$ first performs the *pure* (i.e. side-effect free) part of P, and then performs the side-effects.

We now elaborate on the semantics. A variable is *immediate* in a given scope if it is declared in that scope. A variable is *delayed* in a given scope if it is declared in an outer scope. *Immediate assignment* has the usual assignment semantics: $x = e$ assigns the value of e to x straight away. Immediate assignment is only valid for immediate variables. *Delayed assignment* does not perform the assignment straight away; instead, the assignment is stored in a queue which is processed when the enclosing sieve block is exited. Like delayed assignment, printing inside a sieve block is delayed until the sieve block is exited. Printing and delayed assignment share the same side-effect queue. Furthermore, if a call to a standard function appears inside a sieve block then the call will be delayed by placing it on the side-effect queue. A call to a sieve function inside a sieve block is made immediately.

The scope in which a program is run is called the *top-level*. Running a top-level program produces three entities: a return value; a new *environment*, which is a mapping from variable names to values; and a collection of side-effects, which is a list of print commands. When a sieve block exits to the top-level, the side-effect queue is run — performing a series of assignment, function calls and printing operations. When a nested sieve block exits to an enclosing sieve block, any writes to variables which are immediate in the enclosing sieve block are extracted from the queue and performed. The remainder of the queue is appended to the queue of the enclosing sieve block queue.

Values are ranged over by v. A value is either an integer i, the *unit value* $()$, a standard function Fun (f, P), or a sieve function Sieve (f, P). Environments are ranged over by ρ, σ. We write $\emptyset$ for the empty environment, ρ, σ for the concatenation of environments ρ and σ, and $\rho, (x, v)$ for the *extension* of the environment ρ with the binding $x = v$. We distinguish between *internal* (ι) and *external* (ξ) effects (ε). Internal effects only affect the internal state of a program, whereas external effects are externally observable. Delayed assignment is an internal effect. Printing is an external effect. The semantics refers to lists of effects $[\varepsilon_1, \ldots, \varepsilon_k]$ ranged over by E and lists of external effects $[\xi_1, \ldots, \xi_k]$ ranged over by X. We write :: for the infix cons operation on lists and $+\!\!+$ for the infix concatenation operation on lists. Arithmetic operators are ranged over by $\odot$. Given a CoreSieve arithmetic operator $\odot$ we write $\ulcorner\odot\urcorner$ for the corresponding operator in the metalanguage.

Top-level evaluation. If we run a top-level program P with initial environment σ, then we obtain a value v, a new environment σ', and a list of external side-effects X. We write: $\sigma \vdash P \rightsquigarrow (v; \sigma'; X)$. The top-level evaluation relation appears in Figure 1. Reading an integer constant returns the value of the integer, leaving the enviroment unchanged and causing no side-effects. Similarly, reading a variable returns the value of the variable, leaving the enviroment unchanged and causing no side-effects. At the top-level, standard functions can be applied, but sieve functions cannot. The rule for arithmetic operators evaluates the left argument followed by the right argument, updating the environment accordingly, and concatenating the resulting side-effect queues, before performing the actual arithmetic operation. Declarations and assignments always return $()$. A variable declaration creates a new variable and assigns a value to it, whereas an immediate assignment updates the value of an existing variable. Delayed assignments are not allowed at the top-level. Printing appends a print effect to the list of external side-effects. Standard and sieve function definitions add a corresponding value to the environment. The composition $P; Q$ of programs P and Q discards the return value of P giving the return value of Q when run in the environment created by running P. The side-effect queues are concatenated. The last rule is the most interesting. It describes how to evaluate a sieve block. First, the program P is evaluated using the sieve block evaluation relation, then the resulting side-effects are evaluated. The semantics of a sieve block are only defined if the evaluation of E removes all internal effects. The definition of $\cdot \vdash \cdot \rightsquigarrow \cdot$ depends on two auxiliary evaluation relations: one for evaluation inside sieve blocks, and the other for evaluation of side-effects (on exiting a top-level sieve block).

Evaluation inside sieve blocks. Given a delayed environment ρ, an immediate environment σ and a program P, we obtain a return value v, a new immediate environment σ', and a list of side-effects E. We write: $\rho; \sigma \vdash P \longrightarrow (v; \sigma'; E)$. The sieve block evaluation relation appears in Figure 2. The sieve block evaluation relation is much like the top-level evaluation relation. The main difference is the separation of the environment into a delayed environment and an immediate environment. Accordingly, there are rules for reading and writing delayed variables. Reading a delayed variable is just like reading an immediate variable, except that the variable comes from the delayed environment instead of the immediate environment. Writing to a delayed variable (with a delayed assignment), places the assignment in the side-effect list. Functions can only be defined at the top-level. Inside a sieve block, a call to a standard function is delayed, whereas a call to a sieve function is made immediately. The rule for nested sieve blocks looks rather similar to the rule for top-level sieve blocks. For a nested sieve block, the immediate environment σ of the outer sieve block becomes part of the delayed environment of the inner sieve block. When the inner sieve block exits, only the writes to σ are performed, leaving the rest of the effect list intact.

Evaluation of side-effects. Given an environment σ and a list of side-effects E, we obtain a new environment σ' and a new list of side-effects E'. We write: $\sigma \vdash E \Longrightarrow (\sigma'; E')$. The side-effect evaluation relation appears in Figure 3. Side-effects are evaluated in the order in which they appear in the input list. For an assignment $x = v$, if x is in the domain of the environment then the environment is updated and the assignment is performed; otherwise the side-effect is transferred to the new side-effect list. Similarly, a delayed function delay (f, v) is called if f is in the domain of the environment, and transferred to the new side-effect list otherwise. Note that delayed functions must return $()$. Print commands are transferred to the new side-effect list. When evaluating side-effects at the top-level the new side-effect list will con-

$$\frac{}{\sigma \vdash i \rightsquigarrow (i; \sigma; [])} \qquad \frac{}{\sigma, (x, v) \vdash x \rightsquigarrow (v; \sigma, (x, v); [])}$$

$$\frac{\sigma \vdash f \rightsquigarrow (\mathsf{Fun}\ (x, P); \sigma; []) \quad \sigma \vdash e \rightsquigarrow (v; \sigma'; X) \quad \sigma', (x, v) \vdash P \rightsquigarrow (v'; \sigma''; X')}{\sigma \vdash f(e) \rightsquigarrow (v'; \sigma''; X +\!\!+ X')} \qquad \frac{\sigma \vdash e \rightsquigarrow (v; \sigma'; X) \quad \sigma' \vdash e' \rightsquigarrow (v'; \sigma''; X')}{\sigma \vdash e \odot e' \rightsquigarrow (v \ulcorner\odot\urcorner v'; \sigma''; X +\!\!+ X')}$$

$$\frac{\sigma \vdash e \rightsquigarrow (v; \sigma'; X)}{\sigma \vdash \mathsf{int}\ x = e \rightsquigarrow ((); \sigma', (x, v); X)} \qquad \frac{\sigma \vdash e \rightsquigarrow (v'; \sigma'; X)}{\sigma, (x, v) \vdash x = e \rightsquigarrow ((); \sigma', (x, v'); X)} \qquad \frac{\sigma \vdash e \rightsquigarrow (v; \sigma'; X)}{\sigma \vdash \mathsf{print}\ e \rightsquigarrow ((); \sigma'; X +\!\!+ [\mathsf{print}\ v])}$$

$$\frac{\sigma, (f, \mathsf{Sieve}\ (\rho, x, P)) \vdash Q \rightsquigarrow (v; \sigma'; X)}{\sigma \vdash \mathsf{sievefun}\ f(x) = P\ \mathsf{in}\ Q \rightsquigarrow (v; \sigma'; X)} \qquad \frac{\sigma, (f, \mathsf{Fun}\ (\rho, x, P)) \vdash Q \rightsquigarrow (v; \sigma'; X)}{\sigma \vdash \mathsf{fun}\ f(x) = P\ \mathsf{in}\ Q \rightsquigarrow (v; \sigma'; X)}$$

$$\frac{\sigma \vdash P \rightsquigarrow (v; \sigma'; X) \quad \sigma' \vdash Q \rightsquigarrow (v'; \sigma''; X')}{\sigma \vdash P; Q \rightsquigarrow (v'; \sigma''; X +\!\!+ X')} \qquad \frac{\rho; \emptyset \vdash P \longrightarrow (v; \sigma; E) \quad \rho \vdash E \Longrightarrow (\rho'; X)}{\rho \vdash \mathsf{sieve}\ \{P\} \rightsquigarrow ((); \rho'; X)}$$

Figure 1: Top-level evaluation relation

$$\frac{}{\rho; \sigma \vdash i \longrightarrow (i; \sigma; [])} \qquad \frac{}{\rho; \sigma, (x, v) \vdash x \longrightarrow (v; \sigma, (x, v); [])} \qquad \frac{}{\rho, (x, v); \sigma \vdash x \longrightarrow (v; \sigma; [])}$$

$$\frac{\rho; \sigma \vdash f \longrightarrow (\mathsf{Sieve}\ (x, P); \sigma; []) \quad \rho; \sigma \vdash e \longrightarrow (v; \sigma'; E) \quad \rho; \sigma', (x, v) \vdash P \longrightarrow (v'; \sigma''; E')}{\rho; \sigma \vdash f(e) \longrightarrow (v'; \sigma''; E +\!\!+ E')} \qquad \frac{\rho; \sigma \vdash f \longrightarrow (\mathsf{Fun}\ (x, P); \sigma; []) \quad \rho; \sigma \vdash e \longrightarrow (v; \sigma'; E)}{\rho; \sigma \vdash f(e) \longrightarrow ((); \sigma'; E +\!\!+ [\mathsf{delay}\ (f, v)])} \qquad \frac{\rho; \sigma \vdash e \longrightarrow (v; \sigma'; E) \quad \rho; \sigma' \vdash e' \longrightarrow (v'; \sigma''; E')}{\rho; \sigma \vdash e \odot e' \longrightarrow (v \ulcorner\odot\urcorner v'; \sigma''; E +\!\!+ E')}$$

$$\frac{\rho; \sigma \vdash e \longrightarrow (v; \sigma'; E)}{\rho; \sigma \vdash \mathsf{int}\ x = e \longrightarrow ((); \sigma', (x, v); E)} \qquad \frac{\rho; \sigma \vdash e \longrightarrow (v'; \sigma'; E)}{\rho; \sigma, (x, v) \vdash x = e \longrightarrow ((); \sigma', (x, v'); E)}$$

$$\frac{\rho; \sigma \vdash e \longrightarrow (v'; \sigma'; E)}{\rho, (x, v); \sigma \vdash x := e \longrightarrow ((); \sigma'; E +\!\!+ [x = v'])} \qquad \frac{\rho; \sigma \vdash e \longrightarrow (v; \sigma'; E)}{\rho; \sigma \vdash \mathsf{print}\ e \longrightarrow ((); \sigma'; E +\!\!+ [\mathsf{print}\ v])}$$

$$\frac{\rho; \sigma \vdash P \longrightarrow (v; \sigma'; E) \quad \rho; \sigma' \vdash Q \longrightarrow (v'; \sigma''; E')}{\rho; \sigma \vdash P; Q \longrightarrow (v'; \sigma''; E +\!\!+ E')} \qquad \frac{\rho, \sigma; \emptyset \vdash P \longrightarrow (v; \sigma''; E) \quad \sigma \vdash E \Longrightarrow (\sigma'; E')}{\rho; \sigma \vdash \mathsf{sieve}\ \{P\} \longrightarrow ((); \sigma'; E')}$$

Figure 2: Sieve block evaluation relation

tain only external effects. This constraint is expressed in the top-level sieve block rule of Figure 1 and enforced by the side-conditions on x and f in the side-effect relation. The side-conditions guarantee that an internal side-effect can only appear in the output if the side-effect refers to a variable that is not bound by the environment. At the top-level this could only happen if the variable was free, and hence the program was ill-formed.

3. IMPLEMENTATION IN C++

Codeplay has implemented sieves as an extension to C++. Currently three backends have been implemented for: a single-processor machine, a multi-processor machine and a single-processor machine with an Ageia PhysX PPU. The same source code can be compiled for each different architecture.

In single-threaded mode the compiler essentially implements the operational semantics of Section 2 (with many optimisations and a number of extensions that deal with special features of C++ and make the system more useful).

In multi-threaded mode the compiler takes advantage of the delaying of side-effects in order to partition sieve blocks into fragments of code that can be executed in parallel. The different fragments are dispatched using the threading system provided by the host operating system.

In PPU mode the compiler performs the same kind of partitioning of sieve blocks. However, unlike the multi-core x86 architecture in which each processor has an equal status and memory is shared, the PPU has a radically different architecture. An Ageia PhysX chip [5] has many cores (the exact figure is not public), a high internal memory bandwith (nearly two terabits per second), but it has limited bandwidth to main memory via a PCI bus interface. Special code has to be compiled in order to load fragments onto the PPU.

In the future it would be interesting to formalise the sieve block partitioning algorithm in the context of CoreSieve, and prove that the parallel output code is observationally equivalent to the semantics described in Section 2.

Iterators and accumulators. In order to make sieve blocks more concurrent, the number of dependencies between variables needs to be reduced. Sometimes a dependency can be broken by observing that a variable is only ever accessed in

$$\frac{}{\sigma, (x, v) \vdash [x = v'] \Longrightarrow (\sigma, (x, v'); [])} \qquad \frac{}{\sigma \vdash [x = v'] \Longrightarrow (\sigma, (x, v'); [x = v'])}\ x \notin dom(\sigma)$$

$$\frac{\sigma, (x, v) \vdash P \rightsquigarrow ((); \sigma'; E)}{\sigma, (f, \mathsf{Fun}\ (x, P)) \vdash [\mathsf{delay}\ (f, v)] \Longrightarrow (\sigma'; E)} \qquad \frac{}{\sigma \vdash [\mathsf{delay}\ (f, v)] \Longrightarrow (\sigma; [\mathsf{delay}\ (f, v)])}\ f \notin dom(\sigma)$$

$$\frac{}{\sigma \vdash [\mathsf{print}\ v] \Longrightarrow (\sigma; [\mathsf{print}\ v])} \qquad \frac{\sigma \vdash [\varepsilon] \Longrightarrow (\sigma'; E') \quad \sigma' \vdash E \Longrightarrow (\sigma''; E'')}{\sigma \vdash \varepsilon :: E \Longrightarrow (\sigma''; E' +\!\!+ E'')}$$

Figure 3: Side-effect evaluation relation

a certain way.

An important case is a loop iterator that is only ever incremented at the end of the loop. In this case we should be able to execute different iterations of the loop in parallel. A naïve dependency analysis on such a loop will fail, as each loop iteration depends on the previous value of the iterator. However, because we know that the iterator is only ever incremented in one place, we can statically determine the value of the iterator on subsequent iterations, and so break the dependency.

Another case is an accumulator variable that is: initialised to 0, write-only inside a sieve block and whose only admissable operation inside the sieve block is addition. Here we can safely break the dependencies by initialising the variable to 0 in each fragment, and adding all the results together at the end of the sieve block. This construction can be generalised to any monoid.

Sieve C++ provides special support for user-defined iterators and accumulators using C++ classes.

4. PERFORMANCE

Some preliminary performance figures for Sieve C++ programs appear in Table 1. All benchmarks were performed using a PC equipped with: Intel PentiumD 2.8 Ghz dual-core CPU, 2GB RAM, 200GB HDD, Intel Express Graphics adaptor and an Ageia PhysX PCI card. The Mandelbrot benchmark draws a Mandelbrot set and zooms in ten times, following a fixed path. The FFT benchmark a Fast Fourier Transform for an input size of 4194304. The CRC benchmark performs a 32-bit cyclic redundancy check. First it calculates the parity bits, then it checks that the message+parity passes and finally it checks that with errors introduced the CRC correctly detects them. The matrix multiplication benchmark uses the standard algorithm to multiply two 500×500 matrices. The raytracing benchmark ray traces a three-dimensional Julia set.

Each benchmark has three versions: no sieves, single-threaded sieves and multi-threaded sieves. In the latter two cases the source code is identical, but the binary is different. Unfortunately, we are unable to publish performance figures for sieve programs that use the Ageia PhysX card yet. Each number is the average of running the benchmark five times. The timings were quite consistent in that they never varied by more than 1% for the same benchmark.

Single-threaded sieves cost a small penalty for the Mandelbrot, CRC, matrix multiplication and raytracing benchmarks. This is unsurprising, as sieves require a queue of side-effects to be maintained. For these benchmarks, multi-threaded sieves lead to roughly a two times speedup over single-threaded sieves. The FFT results are more interesting. The single-threaded sieve version is 15% slower than the non-sieve version, and even the multi-threaded sieve version is 10% slower than the non-sieve version. Further investigation revealed that the sieve version of the FFT algorithm is performing a large number of delayed writes, and this is dominating the running time. These writes arise from the algorithm copying an array containing all of the output data. The entire array has to be copied to the side-effects queue and then the side-effects queue is executed, effectively copying the array again. In the non-sieve version, the array only needs to be copied once. To make matters worse, the current version of the compiler takes advantage of vector instructions to optimise array copies, but the corresponding optimisation is not yet implemented for delayed writes. We believe that it should be possible to eliminate the delayed writes by using an array accumulator.

5. RELATED SYSTEMS

Many existing systems have similarities with the sieve system. We briefly mention three of them: STM [2], Cilk [1] and BSPlib [3].

STM (Software Transactional Memory) is a form of optimistic concurrency for programming languages. An STM transaction is similar to a sieve block in that inside an STM transaction side-effects are not visible to the outside world, and they only become visible when the transaction commits. The main difference is that unlike a sieve block, an STM transaction can read the values it has written to non-local memory. This can allow for more concurrency, but leads to a non-deterministic system (though each transaction is internally deterministic).

Cilk has similar goals to Sieve C++, providing a small declarative concurrency extension to C. Cilk uses futures to allow blocks of code to execute in parallel, and includes an explicit synchronisation mechanism. As with STM, Cilk is non-deterministic.

BSPlib (Bulk Synchronous Processing) is a declarative concurrency library with implementations in C and Fortran. BSP divides a computation into a series of *supersteps*. A superstep is similar to a sieve block, in that it consists of a parallel computation phase that allows each process direct access only to local memory, followed by a synchronisation phase that transfers data between processes. Supersteps differ from sieve blocks in several ways.

- In a superstep each process must run the same code, whereas sieve blocks can be decomposed into arbitrary fragments by the compiler.
- Unlike the execution of side-effects at the end of a sieve

Table 1: Sieve C++ running times (in seconds)

	no sieves	single-threaded sieves	multi-threaded sieves
Mandelbrot	51.15	59.53	29.29
FFT	2.25	2.58	2.48
CRC	17.28	21.11	9.55
matrix multiplication	0.60	0.63	0.33
raytracing	6.20	6.53	3.48

block, superstep synchronisation is non-deterministic. Global reads and writes are not guaranteed to happen in any particular order.

- Side-effects and message passing are allowed inside a superstep computation, but not inside a sieve computation.

6. CONCLUSION

Sieves are a promising approach to the problem of multi-core programming. They have a straightforward semantics, support debugging on a single core machine, and preliminary testing indicates that they can be made to scale to different architectures. Initial performance figures suggest that sieves can be implemented efficiently. In future more realistic benchmarks should be performed using different target architectures. It would be particularly interesting to see how well sieves scale to more than two x86 cores, and how much source programs need to be tweaked in order to make the best use of different target architectures.

Acknowledgements. Thanks to Andrew Cook, Colin Riley and Verena Achenbach, for providing the preliminary performance figures, and to Ezra Cooper and Jeremy Yallop for helpful feedback.

7. REFERENCES

[1] The Cilk project. `http://supertech.csail.mit.edu/cilk/index.html`.

[2] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP*, pages 48–60, New York, NY, USA, 2005. ACM Press.

[3] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, 1998.

[4] Intel earnings release: Q3, 2006. `http://www.intel.com/intel/finance/earnings/IntelQ32006EarningsRelease.pdf`.

[5] Advanced gaming physics defining the new reality in PC hardware, Mar. 2006. White paper. `http://www.ageia.com/pdf/wp_advanced_gaming_physics.pdf`.

[6] A. Richards. The Codeplay Sieve C++ parallel programming system, 2006. `http://www.codeplay.com/downloads_public/sievepaper-2columns-normal.pdf`.