Delayed Side-effects Ease Multi-core Programming

Anton Lokhmotov[1]*, Alan Mycroft[1], and Andrew Richards[2]

[1] Computer Laboratory, University of Cambridge
15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK
{anton.lokhmotov,alan.mycroft}@cl.cam.ac.uk
[2] Codeplay Software Ltd, 45 York Place, Edinburgh, EH1 3HP, UK
andrew@codeplay.com

Abstract. Computer systems are increasingly parallel and heterogeneous, while programs are still largely written in sequential languages. The obvious suggestion that the compiler should automatically distribute a sequential program across the system usually fails in practice because of the complexity of dependence analysis in the presence of aliasing.
We introduce the sieve language construct which facilitates dependence analysis by using the programmer's knowledge about data dependences and makes code more amenable to automatic parallelisation.
The behaviour of sieve programs is deterministic, hence predictable and repeatable. Commercial implementations by Codeplay shows that sieve programs can be efficiently mapped onto a range of systems. This suggests that the sieve construct can be used for building reliable, portable and efficient software for multi-core systems.

1 Introduction

The evolution of high-performance single-core processors via increasing architectural complexity and clock frequency has apparently come to an end, as multi-core processors are becoming mainstream in the market. For example, Intel expects [1] that by the end of 2007 most processors it ships will be multi-core.

Homogeneous, shared memory multi-core processors, however, are but a part of the multi-core advent. Another growing trend is to supplement a general-purpose "host" processor with a special-purpose co-processor, which is typically located on a separate plug-in board and connected to large on-board memory. Graphics accelerators have been available since the 1990s and are increasingly used as co-processors for general-purpose computation. AGEIA's PhysX processor [2] is an accelerator for the highly specialized simulation of physical environment. Yet another example is ClearSpeed's SIMD array processor [3] targeted at intensive double-precision floating-point computations. These accelerators containing tens to hundreds of cores can be dubbed deca- and hecto-core to distinguish them from the currently offered dual- and quad-core general-purpose processors.

* This author gratefully acknowledges the financial support by the TNK-BP Cambridge Kapitza Scholarship Scheme and Overseas Research Students Awards Scheme.

Computer systems composed of multi-core processors can be fast and efficient in theory but are hard to program in practice. The programmer is confronted with low-level parallel programming, architectural differences between system components, and managing data movement across non-uniform memory spaces—sometimes all at once. Writing parallel programs is hard (as people tend to think sequentially) but testing is even harder (as non-deterministic execution can manifest itself in evasive software errors).

Ideally, the programmer wants to write a high-level program in a familiar sequential language and leave the compiler to manage the complexity of the target hardware. However, modern mainstream programming languages, particularly object-oriented ones, derive from the C/C++ model in which objects are manipulated *by reference*, *e.g.* by using pointers. While such languages allow for efficient implementation on traditional single-core computers, *aliasing* complicates dependence analysis necessary for sophisticated program transformations including parallelisation.

In this paper we consider an original approach to automatic parallelisation employed in Codeplay's Sieve C++ system [4]. In C99, the programmer can declare a pointer with a `restrict` qualifier to specify that the data pointed to by the pointer cannot be pointed to by any other pointer. In Sieve C++, the programmer makes a stronger statement about code enclosed inside a special block: no memory location defined outside of the block is written to and then read from within the block (we will also say that *the block generates no true dependences* on such memory locations). This guarantee makes code more amenable to auto-parallelisation. Also, the block-based structure maps well to a natural programming style for emerging heterogeneous hierarchical systems.

We describe the basic *sieve* concept in Section 2 and emphasise its importance in Section 3. Section 4 provides experimental evidence. We briefly mention a few recent approaches to parallel programming that are similar to the sieve system in Section 5 and conclude in Section 6.

2 Sieve concept

We describe the basic sieve concept as an extension to a C-like language. The extension is both syntactic and semantic. Essentially, the programmer encloses a code fragment inside a *sieve block*—by placing it inside a new lexical scope prefixed with the `sieve` keyword. As a semantic consequence, all side-effects on data defined outside the sieve block, are *delayed* until the end of the block.

The name "sieve" has been proposed by Codeplay [4]. We can draw an analogy with a French press, or cafetière. A code fragment inside a sieve block (*cf.* a cylindrical jug with a plunger) is a mix of operations that either have delayed side-effects (*cf.* coffee grounds) or not (*cf.* boiling water). By depressing the plunger, equipped with a sieve or mesh, we collect the side-effects (grounds) at the bottom of the block (jug), leaving operations without side-effects (*cf.* drinkable coffee) that can be freely re-ordered (*cf.* thoroughly enjoyed).

2.1 Preliminaries

We say that a variable is *bound* in a given lexical scope if it is defined within this scope; a variable is *free* in a given lexical scope if it occurs in this scope but is defined outside of this scope.

2.2 Syntax

The programmer opens a new lexical scope and prefixes it with the `sieve` keyword denoting a sieve block:

```
sieve { int b; ... } // sieve block
```

We will call code enclosed in a sieve block as *sieve code*.

2.3 Semantics

Lindley presented the formal semantics of a core imperative language extended with sieves in [5]. We illustrate the sieve semantics by drawing an analogy with calling a function having *call-by-value-delay-result* parameter passing mechanism, which we detail below.

In C-like languages, entrance to a new lexical scope can be seen as equivalent to calling a function (whose body is the scope), arranging that the parameters to this function are the free variables of the scope and passing these by reference. For example, in the following

```
int main() {
   int a; ...
   { ... = a ... a = ... } ...
}
```

the enclosed code fragment accessing the variable `a` can be abstracted as a function call

```
void f(int *ap) { ... = *ap ... *ap = ... }
int main() {
   int a; ...
   f(&a); ...
}
```

By passing `a` by reference we ensure that all modifications to `a` are immediately visible in the program. Note that reads and writes to `a` are treated equally by replacing occurrences of `a` with `*ap`.

The `sieve` keyword changes the semantics of a lexical scope to mean that all modifications to free variables are *delayed* until the end of the scope, whereas all modifications to bound variables remain *immediate*. In accordance with this semantics, we will also refer to free and bound variables as, respectively, delayed and immediate.

Using the function call analogy, we say that

```
int main() {
   int a; ...
   sieve { ... = a ... a = ... } ...
}
```

is equivalent to

```
void f(int *ap) {
   const int ar = *ap;
   int aw = *ap;
   { // sieve block entry
      ... = ar ... aw = ...
   } // sieve block exit
   *ap = aw;
}
int main() {
   int a; ...
   f(&a); ...
}
```

Note the different treatment of reads and writes. On entry to the function the parameter (passed by reference) is copied into local variables `ar` and `aw`. All reads of the parameter are replaced with reads of `ar`, and all writes to the parameter are replaced with writes to `aw`. On exit from the function, `aw` is copied out to the parameter.

We coin the term *call-by-value-delay-result* (CBVDR) for this, as the translation is similar to traditional *call-by-value-result* (used, for example, for `in-out` parameters in Ada), where `ar` and `aw` are coalesced into a single variable.

2.4 Understanding the change in semantics

The theory of data dependence [6] helps in understanding how the sieve block semantics departs from that of a standard lexical scope. In the presence of data dependences, the behaviour of sieve code is affected as follows:

1. If a write to a free variable is followed by a read of it (*true dependence*), delaying the write *violates* the dependence.
2. If a read of a free variable is followed by a write to it (*anti-dependence*), delaying the write *preserves* the dependence.
3. If a write to a free variable is followed by another write to it (*output dependence*), delaying the writes *preserves* the dependence *if the order of writes is preserved*.

Since true dependences are violated, it is up to the programmer to ensure that sieve code generates no true dependences on delayed data (this gives the desired equivalence with the conventional semantics). This is hardly restrictive, however, as instead of writing into and subsequently reading from a delayed variable, the programmer can write into and read from a temporary immediate variable, updating the delayed variable on exit from the block.

Anti-dependences present no problem.

Preserving output dependences can be achieved by replacing every write to a delayed variable with a push of *address-value* pair onto a FIFO queue, and applying all queued writes in order on exit from the sieve block. We will refer to such a queue as *side-effect queue*.

Note that the programmer's implicit claim that sieve code generates no true dependences on delayed data, can be easily verified at run-time (and used for debugging) by additionally recording executed reads in the queue and checking that no read from a memory address is followed by a write to the same address.

2.5 Illustrative example

Consider the following example:

```
int main() {
   int a = 0;
   sieve {
      int b = 0;
      a = a + 1; b = b + 1; print(a, b); // prints 0,1
      a = a + 1; b = b + 1; print(a, b); // prints 0,2
   }
   print(a); // prints 1
}
```

The first two `print` statements behave as expected as writes to the free variable `a` are delayed until the end of the sieve block, but the result of the third may come as a surprise. This result, however, is easy to explain using the CBVDR analogy, since the sieve block is equivalent to

```
void f(int *ap) {
   const int ar = *ap;
   int aw = *ap;
   {
      int b = 0;
      aw = ar + 1; b = b + 1; print(ar, b); // prints 0,1
      aw = ar + 1; b = b + 1; print(ar, b); // prints 0,2
   }
   *ap = aw; // *ap = 1, since aw == 1
}
int main() {
   int a = 0;
   f(&a);    // passing ap, where *ap == 0
   print(a); // prints 1
}
```

The immediate variable `ar` is never modified (by construction), hence both assignments to `aw` inside the sieve block write `1`. After the sieve block, the immediate variable `aw` is copied into the delayed variable `a`.

This behaviour seems counter-intuitive because the sieve code violates the requirement of the previous section by generating a true dependence on the delayed variable `a` (the compiler can reasonably warn the programmer).

If the programmer wants to use the updated value of `a`, he needs to write

```
int main() {
   int a = 0;
   sieve {
      int b = 0, c = a;
      c = c + 1; b = b + 1; print(c, b); // prints 1,1
      c = c + 1; b = b + 1; print(c, b); // prints 2,2
      a = c;
   }
   print(a); // prints 2
}
```

As this code generates no true dependences on delayed variables, the conventional semantics is preserved.

2.6 Function calls

A common imperative programming style is to use functions for their side-effects. When calling a function inside a sieve block, the programmer can specify whether to execute the function call immediately and have its side-effects delayed (this is natural for functions returning a result) or delay the call itself until the end of the block (this can be useful for I/O).

3 Importance of the sieve construct

3.1 Delayed side-effects facilitate dependence analysis

Effectively exploiting parallel hardware (whether executing synchronously or asynchronously, in shared or distributed memory environment) often requires the compiler to re-order computations from the order specified by the programmer. However, indirect reads and writes, which are endemic in languages like C/C++, are difficult to re-order, as *alias analysis* is undecidable in theory, and even state-of-the-art implementations often give insufficient information for programs written in mainstream programming languages.

Consider a typical multi-channnel audio processing example

```
for (int i = 0; i < NCHANNELS; i++) process_channel(i);
```

Often, the programmer "knows" that each channel is independent of the others and hence hopes that the code will be parallelised. In practice, this hope is usually misplaced, as somewhere in `process_channel()` there will be indirect memory accesses causing the compiler to preserve the specified (sequential) execution order.

The kernel of the problem is that the programmer writes clear and concise sequential code but has no language-oriented mechanism to express the deep knowledge that sequenced commands can actually be re-ordered.

The sieve construct provides the programmer a way to conclude a treaty with the compiler:

> "I solemnly swear that sieve code generates no true dependences on delayed data. Please preserve false dependences on delayed data by maintaining the side-effect queue."

As a result of this treaty, the compiler assumes that sieve code can generate dependences only on immediate data. This reduces the complexity of dependence analysis and thereby makes the code fragment more amenable to parallelisation.

3.2 Programming hierarchical heterogeneous systems

In modern heterogeneous computer systems, each processor can have its own memory space mitigating a potential bottleneck when several processors require access to shared memory. For example, a general-purpose processor connected to a (reasonably large) memory can be supplemented with a co-processor (for specialised compute-intensive tasks, such as graphics, physics or mathematics) having its own local memory.

When programming such systems, it is often desirable to transfer code and data for the off-loaded computation from host's main memory to co-processor's local memory, perform the computation with the co-processor accessing only its local memory, and then transfer the results back to main memory.

The sieve construct provides a high-level abstraction of this programming model. Assume that code outside of any sieve blocks is to be executed on the host processor. Think of a sieve block as containing code to be executed on the co-processor and immediate data to be (statically) allocated in the co-processor's local memory. Think of delayed data (either statically or dynamically allocated) as residing in main memory.

Conceptually (recall CBVDR), delayed data is passed to a sieve block by reference, read on entry to the block, and written to on exit. The actual implementation can be system and program specific.

Suppose the co-processor can only access data in local memory, *i.e.* requires DMA transfers to access data in main memory. The compiler replaces main memory accesses in sieve code with calls to a run-time system.

The run-time system maintains the side-effect queue for writes to main memory; furthermore, it can optimise reads from main memory by prefetching and servicing them from local memory.

Run-time operation can be guided by a system description (specifying, for example, latency and bandwidth of DMA requests) and pragma annotations. The annotations give a benevolent *hint* to the compiler at what might be the most efficient implementation of a particular sieve block (perhaps, suggested by profiling).

For example, for a system composed of a multi-core general-purpose processor and a co-processor, the programmer can hint the compiler that it is better to parallelise a particular sieve block across multiple cores than off-loading it to the co-processor. As another example, if sieve code reads an array allocated in main memory, the programmer can hint whether array accesses are *dense* (hence it is worth prefetching the array using a contiguous DMA transfer) or *sparse* (array

elements can be read on demand). The programmer can also specify *when* the side-effect queue is to be committed to main memory. After dispatching code and data to the co-processor, the host processor can continue execution until off-loaded computation results are needed, apply the queued side-effects, and resume execution.

3.3 Auto-parallelising sieve blocks

The sieve construct relieves the compiler from complex inter-procedural dependence analysis on delayed data. The compiler, however, still needs to analyse dependences on immediate data, and again the programmer can assist in this.

The programmer is discouraged from accessing immediate storage via *immediate* pointers[3], as this can hinder dependence analysis and defeat the very purpose of the sieve construct.

Scalar variables are a frequent source of data dependences [6]. Two important classes of scalar variables that give rise to loop-carried dependences are iterators (used to control loops) and accumulators (used as the target of reduction operations). The auto-parallelising compiler needs to know the exact behaviour of such variables, *e.g.* that a loop iterator is modified only within the loop header or that a reduction operation is associative. By defining and using special classes for accumulator and iterator variables, the programmer can pass his knowledge about such variables to the compiler.

4 Experimental evaluation

Sieve C++ is an extension to C++ by Codeplay [4], which supports the sieve construct and several refinements, including the support of iterator and accumulator classes. As of May 2007, Sieve C++ backends exist for: homogeneous multi-core x86 systems, x86 supplemented with an AGEIA PhysX board [2], and the IBM/Sony/Toshiba Cell processor.

The Codeplay Sieve system consists of a Sieve C++ compiler and a run-time system. The compiler partitions code inside a sieve block into fragments which can be executed in parallel. The run-time system is invoked on entry to a sieve block with the independent fragments, which the system distributes among multiple cores. In particular, a parallel loop can be strip-mined and speculatively executed by parallel threads. The threads build their own side-effect queues and return them to the run-time system which then commits the side-effects in order.

In Fig. 1 we present results[4] obtained on a Dell PowerEdge SC1430 system, with two 1.6GHz quad-core Intel Xeon E5310 processors and 2GB RAM, running under Windows XP. The execution time is normalised with respect to the original C++ code.

[3] Immediate and global pointers are incompatible, as they may refer to distinct memory spaces.

[4] We thank Colin Riley and Alastair Donaldson for providing performance figures.

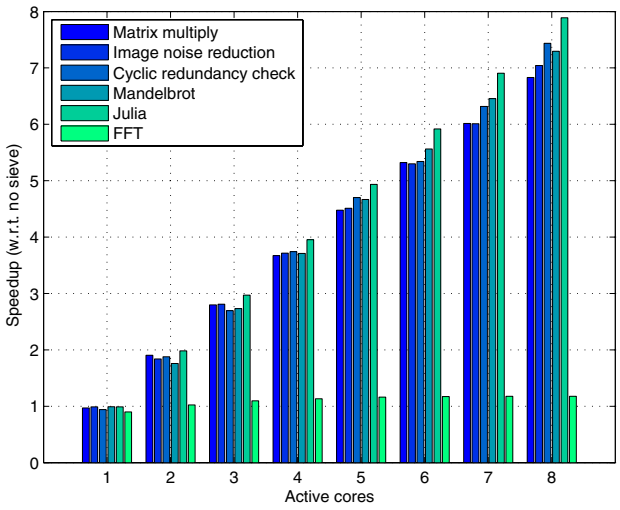

Fig. 1. Experimental results on Dell PowerEdge SC1430.

The matrix multiplication is performed for square 750×750 matrices. The noise reduction program applies a median filter to a 512×512 image, using a 20×20 neighbourhood per pixel. The cyclic redundancy check is performed on a random 1M ($1\text{M} = 2^{20}$) word message. The Julia program ray traces a 1024×1024 3D slice of a 4D quaternion Julia set. The Mandelbrot program calculates a 4500×4500 fragment of the Mandelbrot set. The FFT program performs a 16M-point Fast Fourier Transform.

The Sieve C++ programs suffer up to a 10% overhead on a single core[5], but show a performance improvement on multiple cores. The noise reduction program has nearly linear speed up. The FFT program, however, shows little improvement. We attribute this to cache-line locking because of the program's irregular memory access pattern.

5 Related work

Recent approaches to shared memory parallel programming include Software Transactional Memory (STM) [7] and Intel Threading Building Blocks (TBB) [8].

In the STM [7] approach, the programmer places a fragment of code inside an *atomic block*, which behaves similar to a database transaction: transaction

[5] Conceivably, the cost of maintaining the side-effect queue can be offset for some programs by (stable) sorting the queue by *address* and thus improving spatial locality of writes; besides, writes to the same address (in practice, these should be suspicious: remember Section 2.5) can be optimised by writing only the last queued value.

side-effects are not visible until the transaction commits. Unlike code in a sieve block, code in an atomic block can immediately read new values of modified free variables. Unlike code in an atomic block, code in a sieve block always "commits" its side-effects without retrying.

Intel TBB [8] is a C++ runtime library that simplifies multithreaded application development. Unlike Sieve C++, the TBB is a template library and works with existing compilers. Using TBB, however, implies parallel programming, not sequential programming and auto-parallelisation by the compiler.

PeakStream [9] and RapidMind [10] offer high-level software development platforms for programming HPC algorithms to run on GPU hardware. As with the sieve system, the same source code can be compiled to a range of systems, but again the programmer explicitly manages parallelism and data movement.

6 Conclusion

This paper has introduced the sieve concept—a novel language construct which facilitates dependence analysis by using the programmer's knowledge about dependences in his code and makes code more amenable to automatic parallelisation. Essentially, the sieve construct plays the rôle of a treaty-point between what is easy for the programmer to guarantee and what the compiler is capable of refactoring.

Observable behaviour of sieve programs is deterministic, hence predictable and repeatable. Codeplay's Sieve C++ implementation has demonstrated that sieve programs can be efficiently mapped onto a range of systems. All this suggests that the sieve construct can be used for building reliable, portable and efficient software for multi-core systems.

Since the sieve construct is a high-level abstraction, its performance is implementation dependent. Future work will concentrate on advanced implementation and optimisation techniques for performance and scalability of sieve programs.

References

1. White paper: Intel is leading the way in designing energy-efficient platforms (2006)
2. AGEIA Technologies: The PhysX processor. `http://www.ageia.com/`
3. ClearSpeed Technology: The CSX processor. `http://www.clearspeed.com/`
4. Codeplay: Portable high-performance compilers. `http://www.codeplay.com/`
5. Lindley, S.: Implementing deterministic declarative concurrency using sieves. In: Proceedings of the ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP). (2007)
6. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann. (2002)
7. Harris, T., Fraser, K.: Language support for lightweight transactions. In: Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA), ACM Press (2003) 388–402
8. Intel: Threading building blocks. `http://www.intel.com/software/products/tbb/`

9. PeakStream: SW development platform. http://www.peakstreaminc.com/
10. RapidMind: SW development platform. http://www.rapidmind.net/