The Codeplay Sieve C++ Parallel Programming System

Andrew Richards, Chief Technology Architect

The Problem

If you want to make a processor that is capable of processing lots of data (like a video processor for a mobile phone, or a physics processor for a games console) but also uses a small amount of electrical power (so you don't need loud fans or large batteries) then you would make a processor with lots of separate processing units in it. The processing units would do separate parts of the work concurrently. Such a processor is called a *multi-core processor.*

Multi-core processors provide high levels of processing power for low levels of electrical power consumption. However, existing methods of programming for such devices are time-consuming, difficult and unreliable.

Small errors in timing can lead to unpredictable results which means manufacturers of devices with multi-core processors in them tend to be extremely cautious about software testing and reliability. The time taken to write reliable software for multi-core processors leads to high cost and a long time-to-market.

The Ideal Solution

What developers ideally want to be able to do is take a single-core C++ program, pass it through an auto-parallelizing compiler and automatically get out a multi-core program. The resulting multi-core program would be as portable, reliable and easy to debug as the original single-core program. The advantage of this ideal is that the compiler can split up the program among as many cores as are suitable for the project. It would therefore be possible to try out different combinations of cores and clock speeds to get the best performance/power consumption/cost ratios.

Why the Ideal Solution is Impossible

Single-core C++ software contains large numbers of *dependencies*. Dependencies are situations where one part of the software *must* be executed after another part of the software. Therefore, the 2 sections cannot be executed at the same time. It might be possible for the programmer to remove those dependencies, either because they are *false dependencies*[1] or because the algorithm used has dependencies in it (so the programmer would use a different algorithm if they were writing the program for a multi-core processor).

It is therefore impossible for the compiler to automatically change the order of execution of the program to make several parts of the program to be executed at the same time on different processors. Programmer intervention is required. The Codeplay Sieve C++ solution is aimed at reducing to a minimum the extent of such programmer intervention.

The Codeplay Sieve C++ Solution

The programmer marks a section of their program with the “sieve” marker. The resulting section of the program is called a “sieve block”. Inside sieve blocks, it is simple and safe for the compiler to perform automatic parallelization.

Inside sieve blocks, side effects are delayed. This means that program code inside sieve blocks can be automatically split into 3 parts: reading data from memory, pure computation and writing data back to memory. Because of this split, the program computation can be parallelized and the memory operations can be separated out parallelized and performed by a DMA system.

The Sieve Concept

The sieve concept is very simple, but has a significant impact on the ability of programmers to write software for parallel systems.

1. *A sieve is defined as a block of code contained within a sieve {} marker and any functions that are marked with sieve.*

[1] false dependencies are dependencies that exist because of the way the program is written, they do not have to be there

2. *Inside a sieve, all side-effects are delayed until the end of the sieve.*
3. *Side effects are defined as modifications of data that are declared outside the sieve.*

These 3 rules have a huge impact on the ability of a compiler to auto-parallelize.

The sieve concept is called "*sieve*" because it sieves out the side effects from your software and then lets you apply them later.

A Simple Example

Here is a trivial example:

```
void simple_loop (float *a,
                  float *b,
                  float f,
                  int n) {
      for (int i=0; i<n; i++) {
            a [i] = b [i] - f;
      }
}
```

This simple C++ loop cannot be safely automatically parallelized by a compiler. There are too many unexplained dependencies in the function, because the 2 pointers 'a' and 'b' could be pointing to memory spaces that overlap.

If we use the sieve concept to parallelize the loop within this function, it looks like this:

```
void simple_loop (float *a, float
    *b, float f, int n) {
    sieve {
       for (int i=0; i<n; i++) {
          a [i] = b [i] - f;
          }
    }    // the assignments to the 'a'
            array are delayed until here
}
```

A Sieve C++ compiler can safely automatically parallelize the loop above because of the well-defined meaning of the 'sieve' construct. The assignments to 'a' are being delayed until the end of the loop, so it is safe to transform this loop so that the new array 'a' is calculated in parallel on separate processors and the result stored back to the array 'a' at the end of the sieve block.

Auto-Parallelization of Sieve Blocks

C++ programs contain large numbers of dependencies. Dependencies are situations where one section of the program must be executed after other sections of the program. Parallelization requires changing the order of execution. Delaying the side effects removes a huge number of dependencies, which allows the compiler to safely alter the order of execution without breaking the reliable execution of the program. This means that the compiler can automatically split up the program and distribute it amongst multiple processors to be executed at the same time (*i.e.* re-ordering).

Inside a sieve block, dependencies can only exist on named local variables. Global variables or pointers to external data can never have dependencies inside a sieve block. This means that any dependencies that do still exist inside a sieve block can be identified by the compiler and output in a simple message that the programmer can easily understand. The compiler will print a message saying that there is a dependency on variable '*x*' at line *n* and that the programmer might want to find a way to remove the dependency to increase parallelism. Removing the last few dependencies is essential to achieving parallel execution of the program. So by providing clear, understandable information to the programmer about where the compiler cannot auto-parallelize, the programmer is able to modify the program to be in a form that the compiler *can* auto-parallelize.

Multiple Memory Spaces with Sieve Blocks

Separating data outside the sieve from data inside the sieve also allows multiple memory spaces to be used. Multiple memory spaces can improve performance of multi-core software by having a different memory space for each processor. Having just one memory space would create a serious memory bottleneck which would stop the multiple processors operating at full speed. By having a separate memory space for each processor, each processor can load and store data from its local memory very quickly. By also providing slower, shared memory spaces, processors can work on shared data. Special Direct Memory Access units (DMA) can be created to quickly transfer data between the different memory spaces. DMA has the advantage over random memory access that it can stream data quickly from large, cheap DRAM.

The data inside a sieve block can be stored on a processor's local memory. The data in main memory that is being read in and written out can be stored in a queue that could be stored in a processor's local memory, main memory or a combination of the two. The actual reading and storing can be performed by a DMA unit.

Using the System in Practice

The process a programmer would use to develop software using the sieve system is:

1. Take an existing non-parallel piece of C++ software
2. Identify a section of the software that is suitable for parallelization.
3. Mark the section with the sieve marker.
4. Mark any functions called by the section with sieve function specifiers.
5. Compile the code and fix any errors reported due to mis-matched sieve levels.
6. Run and test on a single-core processor.
7. Make any adaptations required to keep the program working using sieve semantics.
8. Run the compiler in a special mode that advises the user of situations where dependencies are blocking parallelism.
9. Use the split/merge system to fix any dependencies reported (see below).
10. Compile, run and test on a single-core processor.
11. Once working on a single core processor, compile, run, test and performance analyse on a multi-core processor.
12. Go back to step 2 if there are any more sections suitable for parallelization.

Extending the Sieve Concept to Parallel Algorithms Using Split/Merge

In situations where an algorithm has to change to achieve parallelism, programmers need a way to specify this. A simple example is going through an array of numbers and adding up the numbers to produce a total. On a single-core processor, the programmer would write this:

```
int sum (int *array, int size) {
  int total = 0;
  int index;
  for(index=0; index<size; index++) {
    total += array [index];
  }
  return total;
}
```

This function cannot be auto-parallelized using just a sieve block, because each iteration of the loop is dependent on the previous iteration. The “total” variable would have to be placed inside the sieve block so that writing to it would not be counted as a “side-effect” and delayed until the end.

The sieve system allows the programmer to create special library classes that can be applied to a local variable to solve this problem:

```
int sum (int *array, int size) {
  int result;
  sieve {
    IntSum total (0) splits result;
    IntIterator index;
    for(index=0;index<size;index++) {
      total += array [index];
    }
  }
  return result;
}
```

The “IntSum” class allows parallel accumulators. Within the class definition are methods that allow the sum operation to be split across multiple processors. The “total” variable is split into several “total” variables, each going to a different processor. At the end of the sieve block, all the “total” variables are supplied to a merge function defined inside the “IntSum” class that adds together all the totals and stores the result in the “result” variable. Because the split/merge algorithm defined here only allows accumulate operations to be performed on “total”, the class only defines the “+=” and “-=” operators. It is not legal to read the value of “total” inside the loop. This is the simplest example of an accumulator class. Classes to accumulate trees, lists and other complex data structures can also be defined by the programmer.

The “IntIterator” class allows the “index” variable to be split up and sent to different processors. If the parallel system run-time decides to schedule 10 iterations of the loop on one processor and 10 iterations on another processor, then the value “0” will be assigned to “index” for the first processor and the value “10” for the 2nd. Also, the 1st processor will stop working once the iterator reaches 10. This is the simplest example of a splittable iterator. More complex iterators for lists and trees can also be defined by the programmer.

Both iterators and accumulators would normally be defined by the programmer as templates in header files, so the algorithms could be applied to different data types.

Situations for which the Sieve System is Likely to be Useful

Software development for parallel processors. Suitable parallel processors include:

- Multi-core special-purpose processors.
- Special-purpose coprocessors
- Dual-core or quad-core PC processors
- Multiple processor server systems.
- Distributed computing environments in which a single application is to be distributed across multiple computers across a network (e.g. a “grid”).
- Situations where a processor can be customized to the application. It will be possible to write a single Sieve C++ program and try it out on different combinations of processors, memory sizes and clock speeds to compare power vs performance.

The sieve system is capable of providing portable, scalable software development of complex software on parallel processors.

Sieve Functions

The programmer might want to produce functions that can be called from within sieve blocks. It is not safe to call normal C++ functions immediately inside a sieve block, because the side-effects would have unpredictable effects. For this reason, normal C++ function calls are delayed until the end of a sieve block. This allows functions like “printf” to be called inside sieve blocks and the arguments to the function will be calculated in parallel, but the output will be printed out (in order) at the end.

To call a function immediately inside a sieve block, it must be a sieve function. A sieve function is declared like this:

```
int function (int parameter) sieve;
```

Sieve functions have all their side-effects delayed, like sieve blocks. Because of this, sieve functions can safely be called in parallel. This means that if each iteration of a loop inside a sieve block calls a function, the iterations can be executed in parallel. Even if the sieve function is supplied as a library function without any source code, the compiler can safely parallelize calls to it.

Sieve functions can modify main memory. However, these modifications will be delayed until after the sieve block that called the function.

Immediate Pointers

Sometimes, it is necessary to create complex data structures inside sieve blocks, or to pass variables inside sieve blocks to functions as reference parameters. For this reason, the system needs to have “immediate pointers”. Immediate pointers point to data inside the sieve block. Writing to data pointed to by immediate pointers has immediate effect. Therefore, immediate pointers introduce complex dependencies. So the programmer should use them with care. However, they do allow complex structures to be produced inside sieve blocks and stored in fast, local memory.

Immediate pointers are defined as: “int *0 p;”

Implementing the Sieve System

The sieve system consists of an extension to a C++ compiler, a multi-core linker and a runtime to schedule the processes.

The extended Sieve C++ compiler compiles the program outside sieve blocks normally. Code inside sieve blocks is separated and compiled according to the sieve rules. Side-effects are extracted and converted into delayed side-effects.

To auto-parallelize, the compiler first determines the dependencies in the sieve block. Dependency analysis is well covered in compiler literature. The difference with sieve blocks, however, is that delayed side-effects can be calculated in any order as long as they end up being applied in the correct order. So, there cannot be any read-after-write dependencies to data defined outside the sieve block. This greatly simplifies dependency analysis. In effect, dependency analysis only needs to performed on variables inside sieve blocks and data pointed to be immediate pointers.

The compiler then proceeds to find points in the sieve block where there are no dependencies. If it finds such a point, it is called a "split point".

Often, it will not be possible to find split points without making use of the split/merge operations. Variables defined with split/merge classes (either iterators or accumulators) can have their dependencies removed by calling the merge operation before the split point and the split operation after the split point. Therefore, the compiler can produce more split points by inserting calls to the split and merge operations on the variables to which they apply.

If no split points can be found, then auto-parallelization is not safe.

The compiler then takes the sieve block and splits it up into "fragments" (*figure 1)* of code that start and end at split points (or the start or end of the sieve block). These fragments can be executed in parallel. Fragments have one entry and one or more exit points.

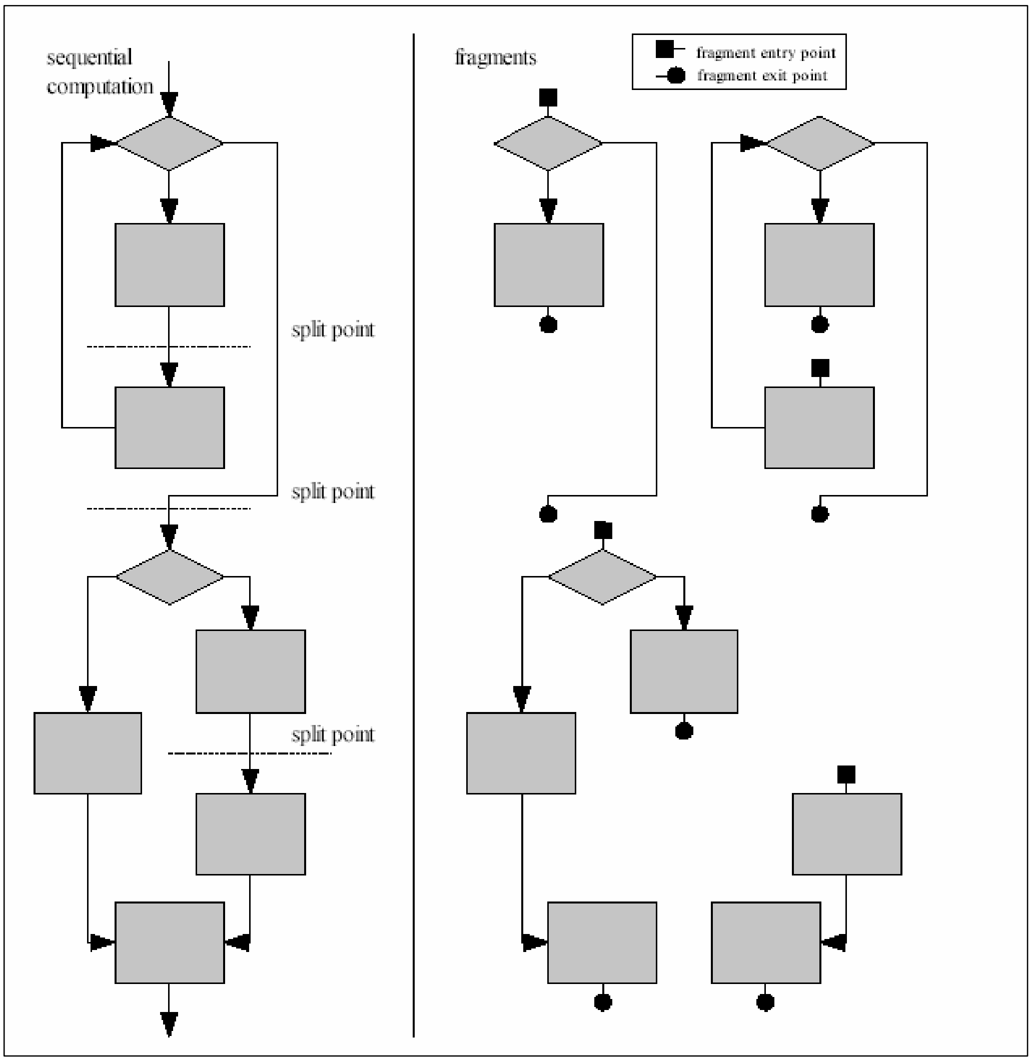

Figure 1

The maximum level of parallelism possible may not be the most efficient implementation, because introducing parallelism involves a fixed cost of calling split/merge operations, sending data and code to a processor, and collecting the results afterwards. So, the compiler will normally take the fragments and re-join them into larger fragments of a size that is known to be optimal for the given architecture.

The sieve system run-time will be invoked on entry to a sieve block. It will be provided with the fragments that were output by the compiler. It will then send those fragments off to the relevant processors. If there are split iterators inside the fragments, then the run-time must choose how many iterations to supply to each processor, giving a start and end iteration count for each.

Because fragments may have more than one exit point, they must return to the run-time which exit point they reached. The run-time can then determine which fragment to execute next.

If the run-time executes a fragment with more than one exit point, it cannot know in advance which exit point the fragment will reach. Therefore, it will have to make a choice: either execute the fragments that correspond to each exit point, or wait until the fragment has finished and then execute which ever fragment corresponds to the exit point it did reach. If the run-time chooses the 1st option, it is executing “speculatively”. When executing speculatively, it must store the side-effects produced by the speculatively executed fragments in separate queues. When the original fragment returns an exit point, the run-time will finish executing the fragment corresponding to the relevant exit point and apply the side effects returned. The other fragments will be killed and their side-effects queues destroyed.

This speculative execution is inefficient, but increases parallelism dramatically, so improving performance.

Implementing the Sieve Run-Time in Hardware

In a typical hardware implementation, there will be a main CPU which has direct access to a main

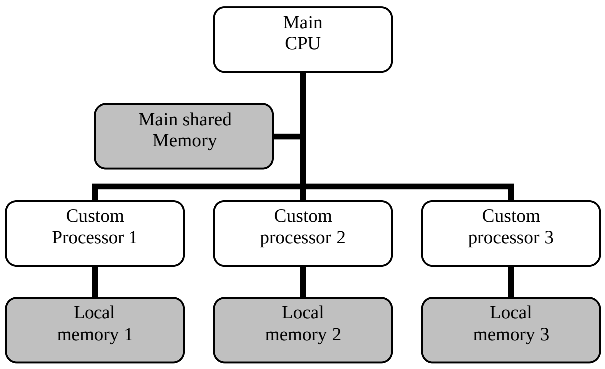

Figure 2

shared memory (*Figure 2*). All C++ code outside sieve blocks will execute on this processor and use the main shared memory. Code inside sieve blocks will be parallelized onto the custom processors, use the local memory for variables inside the sieve blocks, and use main memory for all heap data and variables declared outside the sieve blocks. A single sieve block can be parallelized onto multiple custom processors with multiple local memories. All memory reads from the custom processors to the main shared memory will happen immediately, either using a load instruction or using a DMA access. All memory writes to the main shared memory occur through a run-time library or DMA unit that can queue up the memory writes and apply them on exit from the sieve block.

The sieve system operates best if the custom processors can issue memory load instructions from main memory that cannot cause any exceptions or other side-effects. This makes it safe to speculatively issue load instructions, which improves parallelism. Because the main memory is expected to have a high latency, the ability to speculatively issue load instructions early, reduces the impact of the load latency.

If a processor designer wanted to implement the sieve system efficiently in hardware, they could implement the queuing system as a hardware FIFO (*Figure 3*). Any time a sieve block writes to main memory, the value and address are added to the FIFO. On exit from the sieve block, a DMA unit applies all the memory stores that have been queued in the FIFO to the main shared memory.

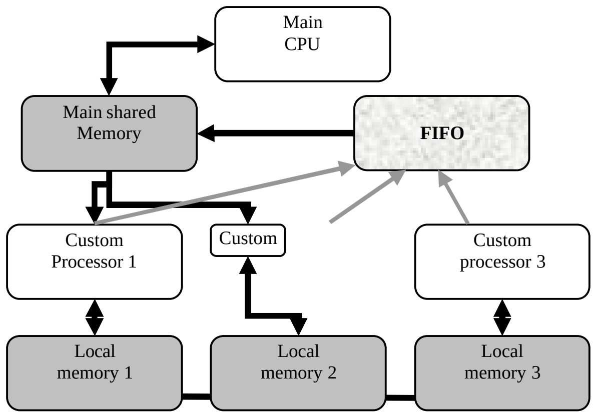

Figure 3

It is not necessary to implement hardware to maintain memory and cache consistency. Because the main memory cannot change while inside a sieve block, the system does not require hardware for maintaining consistency.

Because the sieve C++ compiler outputs small code fragments, it may be possible to implement the code fragments in hardware, as VHDL custom processors, for example. In the simplest case, the custom processors would need hardware to load data from main memory, process the data, and output back to the FIFO.

Impact of the Sieve System on Performance

The sieve system has a benefit and a cost associated with it. The benefit is parallelism. The cost is the overheads of managing the sieve system. The benefits must outweigh the costs for the system to be useful. So the sieve system is suited to high levels of parallelism, or to architectures that map well to the sieve system.

On entry to a sieve block, parallel code sections have to be distributed to processors. Within sieve blocks, stores to memory outside the sieve block must be queued. On sieve block exit, the queued side-effects must be applied before processing on the main CPU can continue. If split/merge classes are used, then the split/merge operations must be called every time a sieve block is parallelized to multiple processors.

One way to reduce the cost of queuing is not to queue. By changing the meaning of sieve blocks to mean that memory writes occur at any time within the sieve block (*i.e.* out of order) then the cost is removed. However, this removes the determinism of the system and hence makes the system harder to test. We therefore don't consider this option further within this paper.

Memory architectures map well to the sieve system. Queued memory stores work well with typical DRAM architectures, and speculative loads hide DRAM load latency. It may even be possible to implement a paged-memory system that allows the processors to continue execution even while queued writes are being applied.

Debugging Sieve C++ Programs

It is possible to create a debugger for Sieve C++ programs. The debugger will be very similar to existing C++ debuggers. It will execute the program in-order, in a single thread, so it will not be necessary for programmers to keep switching between threads to find bugs. The debugger will not interfere with the results of the program. The only change will be that the debugger will show 1 or 2 values for delayed variables – the value that the variable had at the start of the sieve block and (optionally) the value it will have at the end of the sieve block (*i.e.* the value that it has been assigned within the sieve block).

Advantages of the Sieve System over OpenMP

OpenMP is an existing system for parallelizing C++ programs. The sieve system is similar, but has some strong advantages:

1. OpenMP requires the programmer to provide detailed, correct information about the C++ program supplied. The OpenMP parallelizer needs this information to be correct otherwise the parallelize will produce a program with undefined behaviour. This can cause unreliability
2. The OpenMP parallelizer is not able to check that the information supplied by the programmer is correct, whereas the Sieve C++ parallelizer can check dependency information and provide clear errors and advice about how to make the program more parallel.
3. It is possible to debug Sieve C++ programs in a single-threaded environment, but duplicate and fix bugs that exist in multi-threaded execution.
4. The Sieve C++ system can handle multiple non-uniform memory spaces. The sieve blocks separate out data into memory spaces. Variables inside sieve blocks go in memory spaces close to the processors, while variables and heap data go in the largest, shared memory space. It is even possible to handle multiple, hierarchical memory spaces.
5. The Sieve C++ system can operate with DMA as well as load/store memory architectures.
6. The Sieve C++ system can easily use speculative execution, which increases the range of programs that can be parallelized.

Contact: Andrew Richards,
andrew@codeplay.com

Codeplay Software Limited
105 Hanover Street
EDINBURGH EH2 1DJ

Tel +44(0)131 466 0503

codeplay™ WWW.CODEPLAY.COM
PORTABLE HIGH PERFORMANCE COMPILERS