MIT-CSAIL-TR-2011-015 March 22, 2011

Intel Concurrent Collections for Haskell

Ryan Newton, Chih-Ping Chen, and Simon Marlow

Intel Concurrent Collections for Haskell

Ryan Newton Chih-Ping Chen Simon Marlow

Intel Microsoft Research

Abstract. Intel Concurrent Collections (CnC) is a parallel programming model in which a network of *steps* (functions) communicate through message-passing as well as a limited form of shared memory. This paper describes a new implementation of CnC for Haskell. Compared to existing parallel programming models for Haskell, CnC occupies a useful point in the design space: pure and deterministic like Evaluation Strategies, but more explicit about granularity and the structure of the parallel computation, which affords the programmer greater control over parallel performance. We present results on 4, 8, and 32-core machines demonstrating parallel speedups over 20× on non-trivial benchmarks.

1 Introduction

Graph-based parallel programming models, including data-flow and process networks, offer an attractive means of achieving robust parallel performance. For example, programming systems based on synchronous dataflow (SDF) have been able to show fully automatic parallelization and excellent scaling on a wide variety of parallel architectures, without program modification (8), and without specializing the programming model to particular hardware. Graph-based programming can provide both more accessible and more portable parallel programming, and covers an application domain that is large and rapidly growing, including most data-intensive processing from signal processing to financial analytics.

Past research has explored the relationship between functional programming and synchronous dataflow (11) and the semantics of streams and signals (15; 19). But in practice, functional programmers have had few practical tools for parallelism available to them. That is gradually changing. Recent versions of the Glasgow Haskell Compiler (GHC) provide excellent access to pure parallelism (and, increasingly, to nested data-parallelism). Producer-consumer parallelism is also possible in GHC, but typically requires leaving purity behind for IO threads communicating with FIFOs—a low-level, nondeterministic form of programming that does not support the high-level transformations that can make graph-based programming efficient.

The Haskell edition of Intel Concurrent Collections (CnC) (10) is an attempt to bring an efficient multicore implementation of a graph-based programming model to Haskell, while exposing an interface that remains deterministic and pure. The primary contributions of this paper are the following:

– We bring a new parallel programming paradigm to Haskell, which occupies a useful point in the design space: pure and deterministic like Evaluation

Strategies (18), but more explicit about granularity and the structure of the parallel computation.
- Haskell CnC allows multiple scheduling policies to be provided, giving the programmer greater control over parallel execution. New schedulers can be readily developed, and we describe schedulers that we have already built.
- Compared to implementations of the CnC model for other languages, the Haskell implementation can *guarantee* determinism. The Haskell implementation is shorter, clearer, and easier to modify and extend.
- We report results on parallel machines ranging from 4 to 32 cores, and demonstrate speedups in the range of 7× to 22× on non-trivial benchmarks with the current implementation. We also identify some areas for improvement.

This work has stress-tested GHC's parallel capabilities, exposed one runtime bug, and highlighted the effects of improvements that will be available in the 7.0 GHC release. The lessons learned from this project and described in this paper are potentially useful to other projects aiming to achieve significant parallel performance on GHC.

2 The CnC Model

In the Intel CnC model, a network of *steps* (functions) communicate through message-passing as well as a limited form of shared memory. Steps are pure functions with one argument (a message), but may use the contents of that message to *get* data from, and *put* data into, shared tables called *item collections*. As a simple example, consider a wavefront computation in which we compute each position `(i,j)` of a matrix from positions `(i-1,j-1)`, `(i-1,j)`, `(i,j-1)` (also known as a stencil computation). In Haskell CnC we would define a step that computes each (i, j) position of an item collection named `matrix`, as follows:

```
-- Trivial example: add neighbors' values to compute ours
    wavestep (i,j) =
     do nw ← get matrix (i-1,j-1)
        w  ← get matrix (i-1,j)
        n  ← get matrix (i,j-1)
        put matrix (i,j) (nw + w + n)
```

Item collections are single-assignment key-value stores. That is, for each key, only one *put* operation is allowed, and more than one is a dynamic error[1]. If a step attempts to `get` an item that is not yet available, that step blocks until a producer makes the item available. This can be seen as an extension of synchronous dataflow—a step can only complete its execution when all of its inputs (in `wavestep`, the three `gets`) are available. But rather than reading from a fixed set of FIFOs, the step reads a data-dependent (but deterministic) set of

[1] This makes item collections a table of *IVars*(2) similar to Haskell's MVars

items. One way to think of it is that CnC dynamically constructs dataflow networks, where item collections contain an unbounded number of communication channels and coin new ones on demand. This makes CnC a desirable programming model—while still deterministic, unlike data-flow and stream processing the communication relationships can be data-dependent and dynamic.

Our proof of determinism(1) does not directly rely on the determinism of synchronous dataflow. It is instead based on the monotonicity of item collections acquiring new entries. The contents of an item collection—when available—will always be deterministic, and therefore the output of an individual step is also determined. Likewise the result of evaluating a network of CnC steps (the result being the final contents of item collections) is deterministic.

In CnC parlance the messages that cause steps to execute are called *tags* (a term that derives from the fact that these messages typically serve as keys for looking up data within item collections). A step can, in addition to putting and getting items, output new tags and thereby trigger other steps. These `put` and `get` commands give steps an imperative look and feel. The abstraction, however, remains pure; one can think of a step as executing only when its inputs are ready, reading immutable data from item collections, and producing a list of new items and tags as output, i.e. functional updates. Indeed, we provide both pure and imperative *implementations* of Haskell CnC (exposing the same interface) and our pure implementations represent steps in exactly this way.

Finally, in addition to item collections, CnC uses *tag collections* as an intermediary between steps. Each step is controlled by one tag collection. A tag collection receives new tags and broadcasts them to an arbitrary number steps. It is said to *prescribe* those steps. In fact even steps are seen as collections; `wavestep` above would be a step collection while each invocation `wavestep(i,j)` corresponds to a step *instance*. The name "Concurrent Collections" (CnC) is due to the symmetry between step, item, and tag collections.

Historically, CnC has primarily been concerned with static graphs of collections in which all step, item, and tag collections and their dependencies are declared statically (sometimes with metadata enabling index analysis). The result is called a CnC graph or CnC specification, and it enables a greater degree of offline analysis. Usually a CnC tool generates the code for constructing graphs, and therefore it can implement graph transformations (such as fusing steps based on profiling information). Haskell CnC, while compatible with this mode of operation also allows the dynamic construction of CnC graphs via the API described in the next Section (2.1), and, further, allows the use of steps/items outside of an explicit graph (Section 2.2).

2.1 Haskell CnC API

The Haskell CnC API is structured around two monads: `StepCode` and `GraphCode`. Computations in the `GraphCode` monad construct CnC graphs. The monad's methods include `newTagCol` and `newItemCol` for creating tag and item collections, respectively. A call to `prescribe` has the effect of adding to the graph both a step and an edge connecting it to exactly one tag collection[2]. The step itself is a function of one argument (its tag).

```
newTagCol  :: GraphCode (TagCol a)
newItemCol :: GraphCode (ItemCol a b)
type Step tag = tag → StepCode ()
prescribe  :: TagCol tag → Step tag → GraphCode ()
```

Each step in a CnC graph is realized by a computation in the `StepCode` monad. A step interacts with its neighbors in the graph by getting and putting items and by emitting new tags.

```
get  :: Ord a => ItemCol a b → a → StepCode b
put  :: Ord a => ItemCol a b → a → b → StepCode ()
-- Emit a new tag into the graph; putt is short for put-tag:
putt :: TagCol tag → tag → StepCode ()
```

One way to think of a tag collection `TagCol tag` is as a *set* of steps with type `Step tag`. Steps are added to the set by calling `prescribe`. The steps in a `TagCol` are executed by `putt`, which applies each step in the set to the supplied tag.

In the Haskell edition of CnC there are no explicitly established edges between item/tag collections and steps. Instead, item collections are first class values. The Haskell CnC idiom is to define steps within the lexical scope of the collection bindings, or to define steps at top-level and pass needed collections as arguments (a ReaderT monad transformer would do as well).

The above functions allow us to create graphs and steps. Only one thing is missing before we have a basic but useful interface: inserting input data and retrieving results from outside CnC. In CnC, we refer to the program outside the CnC graph as the *environment*. The environment can `put` an initial set of tags and items into the graph, run the graph until no more steps can execute, and finally retrieve output values in item collections.

A third monad could be used to represent environment computations that interact with CnC graphs. But to keep things simple we instead provide a way to lift `StepCode` computations into the `GraphCode` monad and we use it to both construct and execute graphs. We currently enforce a split-phase structure in which the environment executes one *initialize* and one *finalize* step for input and output respectively.

```
initialize :: StepCode a → GraphCode a
finalize   :: StepCode a → GraphCode a
```

Once a `GraphCode` computation is assembled, evaluating the graph to yield a final value is accomplished by `runGraph`. The final result is the value returned by the finalize step.

[2] We support direct visualization and user manipulation of CnC graphs in other CnC implementations, but one advantage of *programmatically* constructing graphs is that the normal tools of abstraction can be used for building and reusing graph topologies.

```
runGraph :: GraphCode a → a
```

2.2 Discussion: Alternate Dynamic API

If the user imports the `Intel.CncDynamic` rather than `Intel.Cnc`, they can access a more basic interface that may be more convenient for some tasks, but which moves away from the explicit graph specifications in the CnC methodology. The idea behind this interface is that the `StepCode` and `GraphCode` monads become a single monad. Item collections may be constructed dynamically by steps, and computations may be spawned as (asynchronous) step instances. This can be used to implement a variety of different control constructs, for example, parallel for loops.

```
data CnC a
instance Monad CnC
runCnC     :: CnC a → a
newItemCol :: CnC (ItemCol k v)
get        :: Ord k ⇒ ItemCol k v → k → CnC v
put        :: ord k ⇒ ItemCol k v → k v → CnC ()
forkStep   :: CnC a → CnC ()
```

Like its graph-based counterpart, the above API provides deterministic parallel execution. It may even be more efficient in some situations, but it also precludes scheduling strategies that involve scheduling many of the same step.

3 Implementation

Haskell CnC provides several different modules that implement the same monadic interface. These include both pure implementations and imperative ones; the latter internally use the IO monad, but hide it through `unsafePerformIO`. Pure implementations provide a clear reference implementation that illustrates the CnC semantics—steps produce lists of new items, which can be lazily integrated into item collections in any order. The imperative implementations, on the other hand, currently provide better parallel performance on most benchmarks.

The key choice in implementing CnC for GHC is how much to rely on existing mechanisms in GHC for scheduling and synchronization. For example, the most simple implementation of CnC is nearly trivial—`forkIO` executes steps and `MVars` provide synchronization on missing items.

3.1 Runtime Schedulers

Haskell CnC version 0.1.4 provides schedulers based, respectively, on *IO-threads*, a *global task queue*, or *sparks* and which use either *MVars* or *continuations* for synchronizing access to items. These choices, in turn, determine how *termination* is handled. We go to the trouble of describing multiple schedulers because each of them is best for at least one of the benchmarks we will see in Section 4—switching between schedulers is part of optimizing parallel performance. Also, because Haskell is an outlier among languages, particularly in implementing lazy evaluation, it should not be assumed that conventional wisdom always applies. (For example, global task queues beat work-stealing in most of our benchmarks.)

3.2 Blocking, MVar-based schedulers

Haskell's MVars are essentially one-element FIFOs that support reading without popping. An item can be represented by an MVar which is initially empty and filled exactly once. IO-threads work together with MVars, blocking when an MVar is read. Most Haskell CnC schedulers use MVars.

IO-Thread scheduler GHC has very lightweight user threads; for a long time it won the "language shootout" Threadring benchmark. In the *IO-thread scheduler* we map each CnC step onto its own thread (e.g. one `forkIO` per step). The result is a simple and predictable scheduler. But, alas, it suffers on programs with finer grained steps. Haskell threads, while lightweight, are still preemptable (and require per-thread stack space). Thus they are overkill for CnC steps, which need not be preempted. It is possible to get a clear idea for the implementation strategy from the types chosen for the two CnC monads, and the representations for tags and item collections.

```
– Step and graph code directly use the IO monad (safely):
newtype StepCode  t = StepCode (IO t)
newtype GraphCode t = GraphCode (IO t)

– Tag collections store executed tags (for optional memoization)
– and a list of steps that are controlled by the tag collection.
newtype TagCol a = TagCol (IORef (Set a), IORef [Step a])

– Mutable maps with support for synchronization:
newtype ItemCol a b = ItemCol (IORef (Map a (MVar b)))
```

Global task pool schedulers These *work sharing* implementations use a global stack or queue of steps, with all worker threads feeding from that pool. The number of worker threads is *roughly* equal to the number of processors. The reason the correspondence is not exact is that blocking on an MVar will stop the associated thread. At start-up, both task-pool-based schedulers fork `numCapabilities` threads, but before blocking on a get, a worker thread must fork a *replacement*.

When a blocked thread wakes up, *over-subscription* will occur (more workers than processors). Our task-pool schedulers minimize, but do not prevent, over-subscription. The strategy is to mark threads that block on an MVar operation as *mortal*—when they wake up they will complete the step they were executing but then terminate rather than continuing the scheduler loop.

Here we will describe two schedulers, designated *workQ_busy* and *workQ_lazy*. Where they differ is in their treatment of termination. When the global task pool runs dry, each worker has a choice to make. Either spin/sleep or terminate; *workQ_busy* takes the former approach, and *workQ_lazy* the latter.

Under *workQ_lazy*, whenever a worker observes the task pool in an empty state, it terminates and sends its ID number back to the main scheduler thread through a `Control.Concurrennt.Chan`. (The scheduler thread is the one that called `runGraph`.) By itself, this strategy creates a different problem—a serial bottleneck will cause workers to shutdown prematurely even if there is another parallel phase coming (i.e., a currently running step will refill the task pool). To compensate, it adopts the following method: upon enqueueing work in the task-pool, if the pool was previously empty, then reissue any worker-threads that are dead.

Both schedulers augment the representation of `StepCode` to include extra state that tracks the task pool, worker ID, and a pointer to the container for mortal threads:

```
newtype StepCode a = StepCode  (StateT (HiddenState) IO a)
```

3.3 Non-blocking, continuation or retry-based schedulers

When a step performs a `get` for unavailable data, rather than blocking on an MVar (which uses the underlying thread-scheduling mechanism to sleep and wake the thread), another option is to *abort* the step and try again later. Aborted steps are registered in a *wait-list* on the item itself. Whenever that item becomes available, steps on the wait-list can be requeued for execution.

There's a design choice to be made as to exactly which function to place on the wait-list. Either (1) the step could be *replayed* from the beginning, or (2) its continuation could be captured at the point of the failed `get` and the computation resumed from that point onward. Typically steps acquire their input data before doing any real work, so the former strategy is not as bad as it sounds, and other CnC implementations use this approach where continuation-capture is not possible. However, this scheduler, which we call simply the *ContT scheduler* uses a continuation monad transformer to provide a limited form of continuation-passing-style (CPS)—specifically, the ability to capture continuations at the point of each `get`. The advantage of monads, in this case, is that they allow library code to CPS-transform a part of the user's program, without modifying the compiler. Finally, the ContT scheduler can use either a global queue to manage tasks or work stealing (as in Cilk (7)). We refer to the resulting variants as *ContT_Q* and *ContT_WS*.

Spark-based Scheduling There's one more scheduler that will appear in Section 4, which is based on a package we built called *HCilk*. GHC's `par` mechanism for pure parallelism implement *futures* which subsume the strictly nested (fork-join) parallelism of Cilk (7). In the GHC runtime, the work stealing dequeues are called *spark pools*. But there's a limitation, sparks cannot safely be used for IO computations because sparks may be dropped at any time. None of the scheduler designs we've discussed thus far use GHC's native work-stealing mechanism. In order to leverage it we built HCilk, which implements fork-join parallelism by *spark*ing IO computations but also storing the sparks in the parent computation, which must *sync* on them before it may return. HCilk can be used to build a CnC implementation by simply forking on all tag or item `puts`, and syncing before a step completes.

3.4 Hash-Tables vs. `Data.Map` and Friends

All (non-Haskell) CnC implementations use some form of hash table to represent item collections. In Haskell, we have the choice of using either mutable data structures (`Data.HashTable`) or mutable pointers to immutable data structures (`Data.Map`). But because Haskell/Hackage do not presently contain a concurrent hash-table implementation, as of this writing we can only (easily) use hash tables concurrently via coarse-grained locking on each table. In our limited tests, we found that even when locking overhead was omitted, `Data.HashTable`-based item collections underperformed `Data.Map` implementations and we settled on Map-based implementations for the time being. For Haskell CnC we have built an implementation of a Map datatype based on indexed type families that allow for certain optimizations in data representation. The details are described in Appendix A.

3.5 Problems and solutions for the GHC runtime system

Parallel schedulers strive to balance load and reduce scheduling overhead; likewise, programmers try to increase locality and manage granularity, but in a complex runtime system (RTS), all of these efforts can be undermined by unforeseen RTS interactions. This section describes an improvement that was necessary for the GHC runtime to support effective parallel scaling in Haskell CnC (which it did not as of the 6.12 release).

The key issue in this case was the handling of "BLACKHOLE" objects used to synchronize when multiple threads try to evaluate the same thunk. We will return to this issue to quantify the impacts of the new BLACKHOLE architecture in Section 4, but first we discuss the structure of the problem and its solution.

Ultimately, we believe targeting new, high-level parallel abstractions to GHC is a mutually beneficial proposition, as evidenced by Haskell CnC's (1) highlighting performance problems, (2) validating the new BLACKHOLE architecture, and (3) uncovering a GHC parallel runtime deadlock bug in the process!

Blocking and lazy evaluation in GHC Lazy evaluation presents an interesting challenge for multicore execution. The heap contains nodes that represent suspended computations (thunks), and in a shared heap system such as GHC it is possible that multiple processors may try to evaluate the same thunk simultaneously, leaving the runtime system to manage the contention somehow.

Fortunately, all suspended computations are pure[3], so a given thunk will always evaluate to the same value. Hence we can allow multiple processors to evaluate a thunk without any ill effects, although if the computation is expensive we may wish to curtail unnecessary duplication of work. It comes down to finding the right balance between synchronisation and work duplication: preventing all work duplication entails excessive synchronisation costs (9), but reducing the synchronization overhead may lead to too much work duplication.

The GHC RTS takes a relaxed approach: by default duplication is not prevented, but at regular intervals (a context switch) the runtime takes a lock on each of the thunks under evaluation by the current thread. The lock is applied by replacing the thunk with a BLACKHOLE object; any other thread attempting to evaluate the thunk will then discover the BLACKHOLE and block until evaluation is complete. This technique means that we avoid expensive synchronisation in the common case, while preventing arbitrary amounts of duplicate work.

It is important for the blocking mechanism to be efficient: ideally we would like to have no latency between a thunk completing evaluation and the threads that are blocked on it becoming runnable again. This entails being able to find the blocked threads corresponding to a particular BLACKHOLE. Unfortunately, due to the possibility of race conditions when replacing thunks with BLACKHOLEs, it was not possible in GHC to attach a list of blocked threads to a BLACKHOLE, so we kept all the blocked threads on a separate global linked list. The list was checked periodically for threads to wake up, but the linear list meant that the cost of checking was $O(n)$, which for large n became a bottleneck.

This issue turned out to be important for the CnC implementation. An item collection is essentially a shared mutable data structure, implemented as a mutable reference in which an immutable value (the mapping from keys to values) is stored. In many cases, the contents of the reference is either unevaluated (a thunk) or partially evaluated, and since the reference is shared by many threads, there is a good chance that one of the threads will lock a thunk and block all the others, leading to at best sequentialisation, and at worst a drastic slowdown due to the linear queue of blocked threads. We observed this effect with some of the CnC benchmarks: often the benchmark would run well, but sometimes it would run a factor of 2 or more slower.

Noticing that the blocking scheme was becoming a bottleneck in certain scenarios, the GHC developers embarked on a redesign of this part of the runtime in GHC. We defer a detailed description of the new scheme for future work, but the key ideas can be summarized as:

– A BLACKHOLE refers to the *owning thread.*

[3] except for applications of `unsafePerformIO`, which present interesting problems. A full discussion is out of scope for this paper, however.

- Blocking is based around message passing; when blocking on a BLACKHOLE, a message is sent to the owning thread.
- The owner of a BLACKHOLE is responsible for keeping track of threads blocked on each BLACKHOLE, and for waking up threads when the BLACKHOLE is evaluated.

Together with some careful handling of race conditions, this scheme leads to a much more efficient and scalable handling of blocked threads. A blocked thread will be woken up promptly as soon as the thunk it was blocked on is evaluated. Since the owner of a BLACKHOLE can be identified, the scheduler can give extra runtime to the owner so as to clear the blockage quickly.

Following the implementation of the new scheme, we noticed significant improvements in many of the Concurrent Collections benchmarks (Section 4).

4 Evaluation

We evaluate Haskell CnC on three platforms, termed: *desktop*, *small server*, and *large server*, the first two representing common platforms today, and the latter representing the coming generation of hardware (or today's very high-end). Most of the benchmarks below are direct ports of their counterparts included in the distribution package for the C++ version of CnC and these provide a natural point of comparison (see Table 1, Figure 1).

- **Black-Scholes** – a differential equation used in finance that describes how, under a certain set of assumptions, the value of an option changes as the price of the underlying asset changes. This benchmark achieved a maximum speedup of 18.4× (Figure 3).
- **Cholesky Decomposition** – an algorithm in linear algebra that factors a symmetric, positive definite matrix A into $LL*$ where L is a lower triangular matrix and $L*$ its conjugate transpose. Cholesky was the largest of the benchmarks we ported (see Table 1).
- **N-body problem** – quadratic algorithm to compute accelerations on N bodies in 3D space. Maximum speedup achieved was 22.1× over single threaded execution.
- **Mandelbrot** – compute each pixel of a Mandelbrot fractal in parallel. Max speedup was 11.8× (Figure 2).
- **Primes** – naive primality test in parallel. Max speedup was 25.5×.
- **Queens** – the N-queens problem (place N queens on a NxN chessboard so none threatens another).

Experimental setup: Our *desktop* platform consisted of a single-socket 3.33 GHz quad-core Core i7, Nehalem architecture; the *small server* was a dual-socket 2.27 GHz 8-core Nehalem, whereas the *large server* was a 32-core platform consisting of four 2.27 GHz 8-core processors, Westmere architecture. Hyperthreading (SMT) was enabled on the desktop and disabled on both servers. Speedup

	blackscholes	cholesky	mandel	primes	nbody	queens
C++ LOC	293	390	147	84	81	
Haskell LOC	90	158	51	29	46	35
Serial Slowdown	0.78	2.47	12.45	2.75	77.3	

Table 1. A table comparing non-comment, non-blank lines of code in C++/CnC benchmarks and Haskell CnC. A 62% reduction in size is achieved on average, in large part by reducing type-definition and boilerplate code. However, serial speed is also lower; dramatically so for nbody.

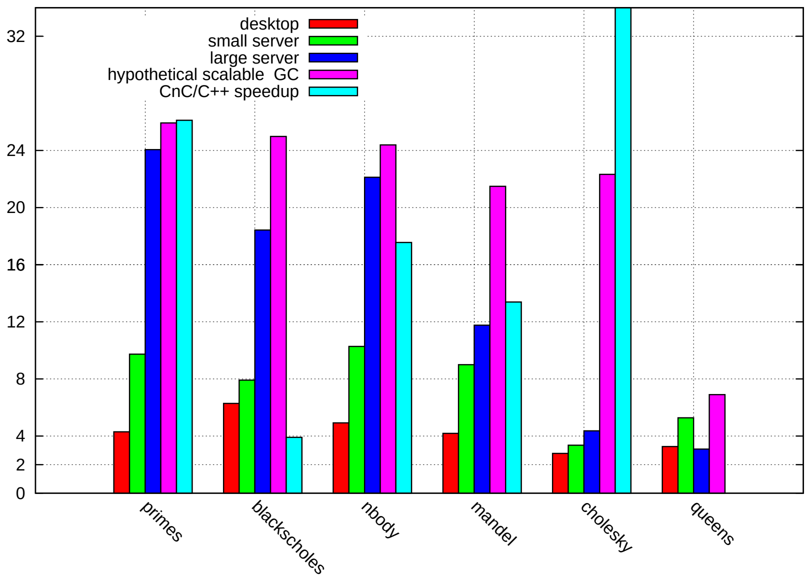

Fig. 1. Best parallel speedup with any number of threads or Haskell CnC scheduler. All Haskell results used the GHC development compiler. C++ results are from the large server platform using identical input sizes; speedups are relative to C++ serial times. (Superlinear speedup on Cholesky was to 34.1×.)

results across our benchmark suite are summarized in Figure 1. All results reported are from the 6.13.20100607 development version of GHC. In contrast, GHC 6.12 produced highly chaotic results and typically parallel slowdown rather than speedup. We verified that the improvement is due to the new BLACKHOLE architecture (Section 3.5) by rewinding to exactly before and after that patch to the compiler. Nevertheless, the programmer must be on the lookout for unpredictable speedups. For example, one culprit is loops that don't allocate, as we found when we implemented a simple multiply-add benchmark. (A thread must allocate to be preempted by the GHC scheduler.)

Examining Figure 1, the reader will notice speedups exceeding 4× on the four-core desktop platform. In our earlier experiments, enabling hyperthreading on larger (8, 32 core) platforms did not provide much benefit and in fact increased unpredictability. On the desktop platform, however, hyperthreading was useful, and as Figure 1 shows, the Black-Scholes benchmark achieved a 6.28× speedup on four cores. Further, all but one benchmark (Cholesky) achieved their best speeds at six or eight (rather than four) threads.

Another striking thing about the results was that superlinear speedups on the small server platform were quite common. Using both processors on a dual-socket system increases the available memory bandwidth and processor cache, which can have a non-linear effect on some applications. Our benchmarks tended produce a lot of memory traffic, a subject to which we will return when we discuss garbage collection.

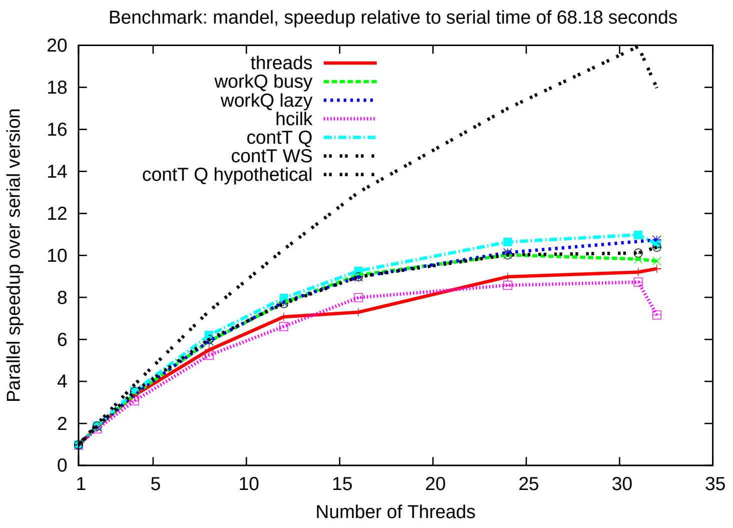

Fig. 2. Mandelbrot benchmark's parallel scaling on large server platform. Three runs per data-point, median time shown. Includes plot for simulated scalable garbage collector (contT_Q hypothetical). The graph also demonstrates the well-known "last core parallel slowdown" when running GHC on Linux.

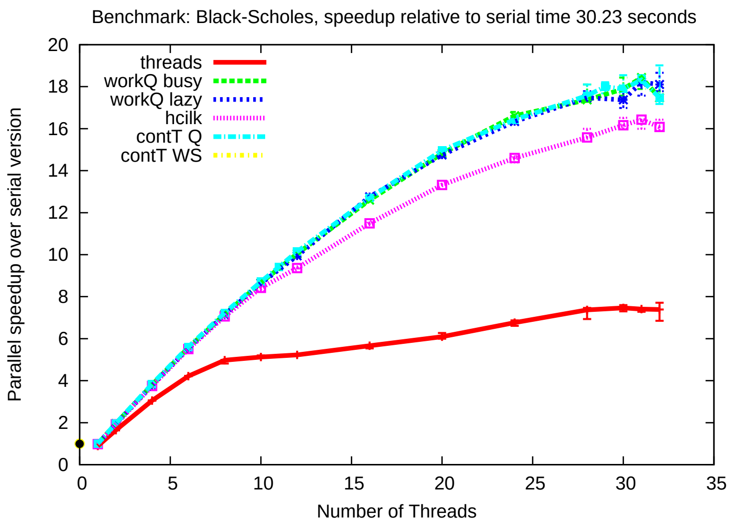

Fig. 3. Black-Scholes benchmark, large server. This benchmark demonstrates how with a larger number of tasks the IO-thread-based implementation lags behind.

The effects of garbage collection

To determine the bottlenecks limiting our test suite's scaling on the large server, we used the performance analysis tools built-in to GHC itself, which include heap residency profiles, multi-threaded system event traces, and simply recording garbage-collection statistics. In addition to the fundamental concerns of scheduler overhead, task granularity, synchronization cost, and locality, we found that the architecture of the garbage collector (GC) is a primary barrier to scaling parallelism to larger numbers of cores with GHC. We can determine this by examining the amount of time spent in garbage collection as a function of the number of threads. Typically it descends from under 10% on one thread to over 40% on 32 threads (sometimes much over). The amount of time spent in garbage collection is sufficient to explain the relative lack of scaling in the Cholesky and Mandelbrot benchmarks. For visualization purposes we included bars in Figure 1 that correspond to a simulated scalable garbage collector, which is assumed to consume a constant amount of time as the number of cores scales. You can see a line representing the same simulated scalable collector on Figure 2.

At the time of writing, GHC is using a stop-the-world parallel garbage collector (12) in which each processor has a separate nursery (allocation area). The garbage collector optimizes locality by having processors trace local roots during parallel GC and by not load-balancing the GC of young-generation collections (13), and this strategy has resulted in reasonable scaling on small numbers of processors.

heap MB	N-body: time / collects	Black-Scholes: time / collects
10	8.9s / 6705	1.63s / 716
100	6.3s / 1080	1.63s / 574
250	6.23s / 421	2.1s / 387
500	6.2s / 211	2.96s / 386
1,000	6.2s / 103	5.29s / 386
10,000	9.45s / 14	93.5s / 386
100,000	44s / 5	

Table 2. (32 threads) The effects of suggested heap size (`-H`) on parallel performance and number of collections—different right answers for different benchmarks. All benchmarks slow down at massive heap sizes (we tested up to 110 gigabytes), but N-body increases in performance up to 500M, whereas Black-Scholes, for example, peaks much earlier and is suboptimal at heap size 500M.

However, the stop-the-world aspect of the garbage collector remains the most significant bottleneck to scaling. The nurseries have to be kept small (around the L2 cache size) in order to benefit from the cache, but small nurseries entail a high rate of young-generation collections. With each collection requiring an all-core synchronization, this quickly becomes a bottleneck, especially in programs that allocate a lot. Hence, the most effective way to improve scaling in the context of the current architecture is to tune the program to reduce its rate of allocation; we found this to be critical in some cases.

This locality trade-off means that the common technique of increasing the heap size to decrease the overall GC overhead often doesn't work. As shown in Table 2, the best selection for one benchmark can do badly on another. (All our results from other figures are reported *without* any per-benchmark tuning parameters, heap-size or otherwise.) A new garbage collector is currently under development that allows individual processors to collect their own local heaps without synchronizing with the other processors, and this should give a significant boost to scaling.

It is difficult to predict the interaction of the heap size, scheduler, and a CnC program. For example, the Mandelbrot benchmark on the large server: using a 10G heap did not significantly lessen the performance (with either one thread or up to 32) for all schedulers *except* the IO-thread based one, where the larger heap size had a catastrophic effect. (It became $2\times$ slower in the serial case, $15.8\times$ slower in the 32-thread case.) Because of the compounded complexity of memory effects and parallelism we recommend auto-tuning approaches. Reinforcing this view is the issue of scheduler choice: every scheduler presented here performs the best in at least one benchmark.[4]

In this study, we experimented systematically with some runtime parameters and informally with others. The GHC runtime can optionally make use of the

[4] Fortunately, because GHC's concurrency abstractions are composable, separately compiled modules can use different Haskell CnC schedulers.

OS's affinity APIs to pin threads to particular cores (the `-qa` flag), and we found that this consistently helped performance. The `-qb` flag disables load balancing within parallel garbage collections (aiding some parallel programs). But across our benchmark suite, enabling the flag results in a geometric mean slow-down of 23% (measured by the best wall-clock time achieved for each benchmark under any number of threads).

Detailed Benchmark Discussion

Granularity of computation is a universal problem in parallel scheduling (though it has been overshadowed by memory concerns in these tests). It is easy to write programs whose step granularity is too small, or allocation rate is too high, to get any parallel speedup whatsoever (or worse, dramatic and unpredictable slow-down), in spite of a large amount of parallelism being exposed. Requiring some support for user control of granularity is typical of systems that rely on dynamic scheduling (e.g. TBB (6), or Haskell CnC) as opposed to static scheduling (e.g. StreamIT (8)).

The Black-Scholes and Cholesky benchmarks, ported from C++ already had a *blocked structure*, wherein steps, rather than operating on individual elements, process batches of elements of a configurable size, thus addressing the granularity problem but requiring tuning of block size. Black-Scholes performed well after its initial port, but Cholesky did not. It turns out that Cholesky performance in Haskell is not affected greatly by block size, rather, on the large server it reaches peak performance at four or eight threads where it hits a memory wall. Cholesky manipulates a large matrix, and in this case suffers greatly from inefficient in-memory data representations.

Many of these benchmarks have fairly high allocation rates. Mandelbrot, primes, Black-Scholes, Cholesky and N-body all produce output of the same size as their input—by allocating arrays or individual items within the steps. N-body, when first ported, achieved no speedup. It allocated two gigabytes over a 2.5 second execution. N-body's inner loop for each body sums over the other bodies in the system to compute an acceleration. This loop was written in idiomatic Haskell (over lists), and while the lists were being deforested, the tuples were not being unboxed/eliminated. After manually inlining the loops, deforesting, and unpacking the tuple loop-state, N-body's allocation decreased by a mere 25% and yet it started achieving excellent speedups. This is indicative of the kinds of sensitivities in parallel performance given the state of the tools at this moment.

5 Discussion and Related Work

CnC is situated in a landscape of graph-based computational models. These include data-flow, stream processing, task graphs, actors, agents, and process networks. These models appear in disparate parts of computer science ranging from databases to graphics to cluster computing, leading to many divergent terminologies. For our purposes, a graph-based computational model is one in which message-passing along defined channels (edges) is the only form of communication between separate computations (nodes). We summarize some of the basic design choices in Table 3.

Set of nodes	• dynamic, • static
Edge data rates	• dynamic, • static
Nodes	• processes, • stateless tasks, • stateful tasks
Node/edge synchronization:	• read all inbound edges at once • read edges in deterministic order • read edges in nondeterministic order (event-driven / asynchronous)

Table 3. Major choices to build-your-own graph-based model.

First, the nodes of a computation graph may be either continuously running processes or discrete tasks that execute for a finite duration after receiving data on incoming edges. Discrete tasks may or may not maintain state between invocations. The set of tasks may be known statically, as in synchronous data-flow (11) and "task scheduling" (16), or change dynamically (as in most streaming databases (3)).

Stream processing systems typically have ordered edges and statically known graph topologies. Generally, they allow both stateful and stateless tasks (the latter providing data parallelism). They may be based on synchronous data-flow (e.g. StreamIt (8)) and have known edge data-rates, or dynamic rates (e.g. WaveScope (14)).

A key choice is how nodes with multiple inbound edges combine data. They can, for example, read a constant number of messages from each inbound edge during every node execution (SDF). Alternatively, edges can be read in an input-dependent but deterministic order (e.g. Kahn networks (4)). Or, finally, edges can be processed by a node in a nondeterministic order, as when handling real time events or interrupts (e.g. WaveScope (14) and most streaming databases).

In contrast with these systems, CnC has unordered communication channels (carrying tags) and stateless tasks (steps). (Unordered channels with state*ful* tasks would be a recipe for nondeterminism.) CnC also has item collections. In a purely message-passing framework, item collections can be modeled as stateful tasks that are connected to step nodes via *ordered* edges that carry *put*, *get*, and *get-response* messages. That is, a producer task sends a `put` message to the item collection and a consumer task first sends a `get` message and then blocks on receipt of a value on the response edge. The edges must be ordered to match up `get` requests and `responses`. This also requires a formulation of steps that allows them, once initiated, to synchronously read data from other incoming edges. Thus CnC can be modeled as a hybrid system with two kinds of edges and two kinds of nodes.

In addition to the above mentioned systems, there are many less obviously related precedents as well, including graphical workflow toolkits (such as Labview and Matlab Simulink) and Linda/Tuple-spaces. Also, the functional programming literature includes several projects that explore the connection between functional programming, stream processing, and synchronous data-flow (15; 5), but not all projects have achieved (or aimed for) effective parallel performance.

6 Conclusion and Future Work

We overcame a number of obstacles to achieve a platform for effective parallel programming in Haskell, delivering excellent performance on the multicore desktop and server platforms that comprise the majority of today's computing landscape. Moreover, while Haskell CnC delivers reasonable performance on larger-scale shared memory machines, it has also concretely identified the areas for future improvements that will enable further scaling.

Future work will refine the data structures used by Haskell CnC, incorporating, perhaps, low-level implementations of concurrent data structures (hash tables and work stealing dequeues). Second, an overhaul of the GHC garbage collector to enable independent per-thread collections is already underway and Haskell CnC is helping to test the new architecture.

Bibliography

[1] Zoran Budimlic, Michael Burke, Vincent Cave, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto1, Vivek Sarkar, Frank Schlimbach, Sagnak Tasrlar. The Concurrent Collections Programming Model. In Press.

[2] Arvind, Rishiyur Nikhil, and Keshav Pingali. I-structures: Data structures for parallel computing. In Joseph Fasel and Robert Keller, editors, *Graph Reduction*, volume 279 of *Lecture Notes in Computer Science*, pages 336–369. Springer Berlin / Heidelberg, 1987.

[3] D. Carney, U. Cetintemel, M. Cherniak, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams—a new class of data management applications. In *VLDB*, 2002.

[4] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-synchronous kahn networks: a relaxed model of synchrony for real-time systems. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 180–193, New York, NY, USA, 2006. ACM.

[5] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a Higher-order Synchronous Data-flow Language. In *ACM Fourth International Conference on Embedded Software (EMSOFT'04)*, Pisa, Italy, September 2004.

[6] Intel Corporation. Intel(R) Threading Building Blocks reference manual. Document Number 315415-002US, 2009.

[7] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of PLDI'98, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[8] M. I. Gordon et al. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, New York, NY, USA, 2006. ACM.

[9] Tim Harris, Simon Marlow, and Simon Peyton Jones. Haskell on a shared-memory multiprocessor. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 49–61. ACM Press, September 2005.

[10] Intel Corporation. Intel Concurrent Collections Website. `http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/`.

[11] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, 1987.

[12] Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap.

In *ISMM '08: Proceedings of the 7th international symposium on Memory management*. ACM, June 2008.

[13] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore haskell. In *ICFP '09: Proceeding of the 14th ACM SIGPLAN international conference on Functional programming*, August 2009.

[14] Ryan R. Newton, Lewis D. Girod, Michael B. Craig, Samuel R. Madden, and John Gregory Morrisett. Design and evaluation of a compiler for embedded stream programs. In *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 131–140, New York, NY, USA, 2008. ACM.

[15] John Peterson, Valery Trifonov, and Andrei Serjantov. Parallel functional reactive programming. In *PADL '00: Proceedings of the Second International Workshop on Practical Aspects of Declarative Languages*, pages 16–31, London, UK, 2000. Springer-Verlag.

[16] Oliver Sinnen. *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2007.

[17] Martin Sulzmann, Edmund S.L. Lam, and Simon Marlow. Comparing the performance of concurrent linked-list implementations in haskell. *SIGPLAN Not.*, 44(5):11–20, 2009.

[18] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *J. Funct. Program.*, 8(1):23–60, 1998.

[19] Tarmo Uustalu and Varmo Vene. Comonadic notions of computation. *Electron. Notes Theor. Comput. Sci.*, 203(5):263–284, 2008.

A Appendix: Generic Map Implementation Details

There are many improvements to be made to a basic `Data.Map` implementation. In particular, the Haskell CnC distribution includes its own implementation of "*generic maps*" (`GMap`) using indexed type families. GMaps can take on different physical representations based on their key types (and potentially value types as well). All key types must provide an instance of the class `GMapKey`, a simplified version of which follows:

```
– A simplified class GMapKey
class GMapKey t where
   data GMap t :: * → *
   empty :: GMap t v
   lookup :: t → GMap t v → Maybe v
```

We can then define instances for each different key type we are interested in. For example, `Data.IntMap` is more efficient than `Data.Map` when keys are integers. Also, pair-keys can be deconstructed and represented using *nested* maps. Likewise, `Eithers`, `Bools`, and unit key types also have specialized implementations.

The problem with this approach is that GMaps are not a drop-in replacement for `Data.Map`. The user would have to be aware that item collections are implemented as GMaps and define their own `GMapKey` instances. They are not derivable, and have no "fallthrough" for index types that satisfy `Ord` but do not have explicit `GMapKey` instances defined. Such a fallthrough would constitute an *overlapping instance* with the specialized versions. The language extension `OverlappingInstances` permits overlaps for regular type classes, but not for indexed type families.

Fortunately there is a work-around for this type-checking limitation, suggested by Oleg Kiselyov on the Haskell-cafe mailing list. The idea is to use an auxiliary type class to first *classify* a given key type, and then dispatch on it (without overlaps) in the indexed type family. The "categories" are represented by `newtypes`, and might include things like `PairType` or `EitherType`, but here we consider only two categories: those types that can be packed into a single word, and those that cannot.

```
– We will classify types into the following categories
newtype PackedRepr t = PR t deriving (Eq,Ord,Show)
newtype BoringRepr t = BR t deriving (Eq,Ord,Show)
```

Next, we assume a class `FitInWord` that captures types that can be packed into a machine word. (Template Haskell would be useful for generating all tuples of scalars that share this property, but this is not yet implemented.)

```
class FitInWord v where
 toWord   :: v → Word
 fromWord :: Word → v

– Example: Two Int16's fit in a word:
fI x = fromIntegral x
instance FitInWord (Int16,Int16) where
 toWord (a,b) = shiftL (fI a) 16 + (fI b)
 fromWord n = (fI$ shiftR n 16,
               fI$ n .&. 0xFFFF)
```

Next, we introduce the class `ChooseRepr`, which permits overlapping instances and does the "classification". We generate an instance for every instance in `FitInWord` that selects the packed representation.

```
-- Auxiliary class to choose the appropriate category:
class ChooseRepr a b | a → b where
   choose_repr  :: a → b
   choosen_repr :: b → a

-- Choose a specialized representation:
instance ChooseRepr (Int16,Int16)
                    (PackedRepr (Int16,Int16)) where
   choose_repr = PR
   choosen_repr (PR p) = p

-- Fall through to the default representation:
instance (c ~ BoringRepr a) => ChooseRepr a c where
   choose_repr = BR
   choosen_repr (BR p) = p
```

It is then possible to create non-overlapping instances of `GMapKey` that use `IntMaps` where applicable and `Maps` otherwise.

```
import qualified Data.IntMap as IM
import qualified Data.Map as Map

-- For PackedRepr we pack the key into a word:
instance FitInWord t => GMapKey (PackedRepr t) where
 data GMap (PackedRepr t) v = GMapInt (IM.IntMap v)
 empty = GMapInt IM.empty
 lookup (PR k) (GMapInt m) = IM.lookup (fI$ toWord k) m

-- For BoringRepr we use Data.Map:
instance Ord t => GMapKey (BoringRepr t) where
 data GMap (BoringRepr t) v = GMapBR (Map.Map t v)
 empty = GMapBR Map.empty
 lookup (BR k) (GMapBR m) = Map.lookup k m
```

Finally, there is one last basic data structure trade-off to make. Haskell CnC's use of Maps requires storing them in a mutable variable and performing atomic updates to that variable. In our implementation we currently include a toggle to select between `TVars`, `MVars`, and `IORefs` for all "hot" mutable variables in the CnC implementation. We reach the same conclusion as previous authors (17), and select `TVars` by default.