Intel Concurrent Collections for Haskell

Ryan Newton
Intel
ryan.r.newton@intel.com

Chih-Ping Chen
Intel
chih-ping.chen@intel.com

Simon Marlow
Microsoft Research
simonmar@microsoft.com

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 Strategies, but more explicit about granularity and the structure of the computation, which affords the programmer greater control over parallel performance. We present results on 4, 32, and 48-core machines demonstrating parallel speedups ranging between 7X and 22X on non-trivial benchmarks.

Keywords Parallel Programming, Runtime Systems, Graph-based models

1. Introduction

Graph-based parallel programming models, including data-flow and stream processing, offer an attractive means of achieving robust parallel performance. At their best, 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. StreamIT (6), for example, effectively targets mainstream shared memory multicores, Tilera many-cores, IBM Cell, GPUs, and workstation clusters.

If we are to avoid a world where dominant parallel programming models are based on particular hardware platforms (e.g. CUDA (15)), graph-based programming based on strong theoretical foundations is one good solution, enabling *both* more accessible and more portable parallel programming. And, while domain specific, the application domain is surprisingly large and rapidly growing—including most data-intensive processing, such as graphics, digital signal processing, and financial analytics.

How does the Haskell programmer fit in? Past research has explored the relationship between functional programming and synchronous dataflow (10) and the semantics of streams and signals (16; 20). But in practice, while Haskell programmers have excellent access to pure parallelism (and, increasingly, to nested data-parallelism) to accomplish producer-consumer parallelism, they typically leave purity behind and venture into the domain of IO threads, MVars, and Chans. This is a low-level, nondeterministic form of programming that does not support the high-level transformations that can make graph-based programming so efficient.

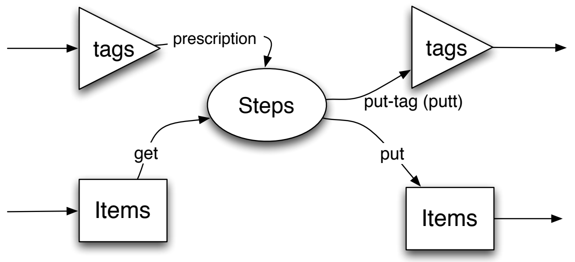

Figure 1. The visual notation for (static) CnC collection graphs. By convention, triangles denote tag collections, ovals step collections, and rectangles item collections. Our "GraphBuilder" tool for Visual Studio allows the construction of graph specifications via a GUI tool.

The Haskell edition of Intel Concurrent Collections (CnC) 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 Strategies (19), but more explicit about granularity and the structure of the 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 ten different 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 demonstrate respectable speedups with the current implementation, and identify areas for improvement.

This paper describes Haskell CnC version 0.1.4 (soon to be released on Hackage; 0.1.3 is available as of this writing). This work has stress-tested GHC parallel capabilities, exposed one runtime bug, and highlighted the effects of improvements that will be available in the 6.14 release. We report results on parallel machines ranging from 4 to 48 cores. The resulting lessons are potentially useful to other projects aiming to achieve significant parallel performance on GHC. Our tests achieved maximum parallel speedups in the range of 7X to 22X on non-trivial benchmarks.

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 stateless and execute only when they receive a message, but may use that message to *get* and *put* data into shared tables. These are immutable tables of key-value pairs called *item collections*. Item collections are essentially constructed both lazily **and** collaboratively. If a step attempts to `get` an item that is not yet available, it blocks until a producer makes it available.

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). Thus a *CnC graph* consists of *step*, *tag*, and *item collections* in producer/consumer relationships. An individual step, when executed, can issue `puts` and `gets` to item collections as well as `puts` to create new tag messages (invoking downstream steps).

CnC Methodology

One of the reasons CnC has been well received within Intel and with existing customers is that it espouses a clear *methodology*. A CnC program consists of (1) a *specification*, typically written in a simple graph-description language[1], and (2) code in a traditional ("host") language that provides an implementation for steps. The Intel Concurrent Collections programming model is described more fully in (8). For this paper, we will omit discussion of the graph specification language (described elsewhere) and consider only standalone Haskell CnC programs that specify their own graphs (albeit without the metadata present in a separate specification file).

The CnC methodology allows for a separation of concerns between the *domain expert* who understands the structure of the computation and writes the spec, and the *tuning expert* who knows how to achieve efficient parallel execution by choosing and tuning schedulers and so on. The domain expert is expected to first identify the computation steps in their application and, second, to identify control and data dependencies—which become tag and item collections, respectively.

3. Haskell CnC

Intel's primary implementation of CnC is *Intel Concurrent Collections for C++*™. Customers use CnC as a C++ library (plus optional mini-language for graph specifications). The model itself is independent of the host-language that supplies step functions; indeed, our academic collaborators have created implementations of CnC for Java and .NET. But why, then, yet another CnC for Haskell? There are four reasons:

- **Determinism and purity** – Our proof of determinism for CnC (in press) requires that steps be pure functions. (For example, if steps have hidden internal state, they can expose nondeterministic execution order in the results.) An unsatisfying aspect of all other CnC implementations is that it is impossible to *enforce* step purity, especially when CnC is only a library. Haskell provides the ability to enforce the model's own rules and make its determinism guarantee solid. Conversely, CnC provides to Haskell programs the ability to execute graph computations inside pure functions.
- **Rapid prototyping** – Haskell has an excellent combination of high-level programming, reasonably good performance, and multi-paradigm support for parallelism. We have been able to experiment with new language features (such as adding parallel for loops within step code, Section 3.1.1) with extremely small amounts of code.
- **Reference implementation** – The simplest of our Haskell CnC runtime schedulers are very short indeed. For anyone who understands Haskell, it provides an easy to understand reference model for CnC. Further, among our implementations are *pure schedulers* that not only expose safe interfaces, but also are internally effect-free. Such a pure implementation provides a clearer illustration of the CnC semantics.
- **Haskell is an exciting place for parallelism**
  Therefore we wish to both contribute to that space and benefit from innovations there. In Haskell (and CnC), concepts like *idempotent work stealing* (13) become applicable—an impossibility in Intel Concurrent Collections for C++, which is based on Intel Thread Building Blocks (TBB).

  Further, while most CnC implementations (including IO-based Haskell implementations) use some form of mutable state to represent item collections, the *pure* implementation of Haskell CnC, on the other hand, performs only functional updates to a data structure representing all item collections in a graph. These updates are associative and commutative (set union), which opens up a number of possibilities for more loosely coupled parallel (and even distributed) implementations—interesting avenues for future work available only in the pure implementation.

3.1 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 the 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.

The following code snippet uses `put` and `get` to define a simple "plus one" step which reads the number indexed by an input tag and outputs its successor.

```
incrStep inpI outI tag =
  do x ← get inpI tag
     put outI tag (x+1)
```

[1] Graphs can also be edited visually and appear as in Figure 1.

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

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

The above functions allow us to create graphs and steps. We almost have a basic but useful interface; only one thing is missing—inserting input data and retrieving results. 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, execute the graph to *quiescence* (no more steps can execute), and finally retrieve outputs with `gets`.

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. One such function would be sufficient, but to allow a maximally large range of scheduling strategies, we currently enforce a split-phase structure in which the user executes one *initialize* and one *finalize* action for input and output respectively. (Continuous interaction between environment and graph is a topic not treated in this paper.)

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

A particular runtime scheduler is permitted to disallow `gets` in the initialize step and to quiesce the graph before the finalize step, thus forcing these entry-points to be used for the purposes indicated by their names. Once a `GraphCode` computation is assembled, evaluating the graph to yield a final value is done, naturally, with `runGraph`:

```
runGraph :: GraphCode a → a
```

Note that, like `runState` in the standard library, `runGraph` is a safe operation. The CnC API may appear to emphasize effects (putting and getting) but this interface is merely a convenience. Again, steps could just as well be implemented as pure functions that return a list of new items and tags (or a “Block” value to indicate that execution blocked on an unavailable item). Indeed, in some of our implementations steps are implemented in just this way, yet even then we find the monadic interface more convenient.

Now with the pieces in place let’s look at an extremely simple but complete program. The following expression computes the value 4. It uses string tags.

```
runGraph $
 do tags ← newTagCol
    i1 ← newItemCol
    i2 ← newItemCol
    prescribe tags (incrStep i1 i2)
    initialize$ do put tags ‘‘key’’
                   put i1 ‘‘key’’ 3
    finalize$ get i2 ‘‘key’’
```

Finally, we look at a larger example program which will also serve as one of our benchmarks in Section 5. The full (executable) text of the *Mandelbrot* benchmark is shown in Figure 2. This program has an identical structure to the simple example above—a *SIMD* computation with a single step collection.

3.1.1 Going further

Before moving on to discuss the implementation, here we will mention some of the additional features and functions in Haskell CnC.

```
import Intel.Cnc
import Data.Complex; import Control.Monad

type CDbl = Complex Double
fI = fromIntegral

– The serial kernel:
mandel :: Int → CDbl → Int
mandel max_depth c = loop 0 0 0
  where
   fn = magnitude
   loop i z count
     | i == max_depth = count
     | fn(z) >= 2.0   = count
     | otherwise      = loop (i+1) (z*z + c) (count+1)

mandelGrph :: Int → Int → Int → GraphCode Int
mandelGrph max_row max_col max_depth =
    do position :: TagCol  (Int,Int)      ← newTagCol
       dat      :: ItemCol (Int,Int) CDbl ← newItemCol
       pixel    :: ItemCol (Int,Int) Int  ← newItemCol

       – A step in the CnC graph:
       let mandelStep tag =
             do cplx ← get dat tag
                put pixel tag (mandel max_depth cplx)
       prescribe position mandelStep

       initialize $
        forM_ [0..max_row] $ \i →
         forM_ [0..max_col] $ \j →
          let z = (r_scale * fI j + r_origin) :+
                  (c_scale * fI i + c_origin) in
          do put dat (i,j) z
             putt position (i,j)

       finalize $
        – For simplicity, below we compute a meaningless metric
        – of the image. Instead, write the pixels to disk here:
        foldM (\acc i →
          foldM (\acc j →
                 do p ← get pixel (i, j)
                    if p == max_depth
                     then return (acc + (i*max_col + j))
                     else return acc)
                acc [0..max_col]
              ) 0 [0..max_row]
   where
    r_origin = -2.0               :: Double
    r_scale  = 4.0 / (fI max_row) :: Double
    c_origin = -2.0               :: Double
    c_scale  = 4.0 / (fI max_col) :: Double

main = let check = runGraph $ mandelGrph 10 10 10 in
       putStrLn (‘‘Mandel check ’’ ++ show check)
```

Figure 2. A complete Haskell CnC program runnable with release 0.1.4, built with `ghc -fglasgow-exts --make mandel.hs`. For the inputs 10, 10, 10 above, it should produce 593.

- `itemsToList` – it is often useful to collect all items within a collection (without knowing which keys are present). For example, one can filter a set (say, for prime numbers) and learn which passed without querying all keys in the domain.

  ```
  itemsToList :: Ord tag => ItemCol tag b
              -> StepCode [(tag,b)]
  ```

  `itemsToList` does what its name implies, but its problem is that it requires that the CnC graph be quiesced before it can be sure no additional items are outstanding. Thus, it is a runtime error to use `itemsToList` anywhere but in a finalize step, and in any scheduler that does not support quiescence.

- `cncFor` – A common abstraction in other parallel runtime systems, including Cilk, TBB, and OpenMP, is the notion of a parallel loop in which contiguous sub-ranges of the iteration space can be assigned to processors. As of version 0.1.4, Haskell CnC includes the `cncFor` operator for use within step code.

  The default `cncFor` implementation (used in most schedulers) is shown below. It dynamically extends the graph with a new step that takes a sub-range as input, then partitions the input range into even sized pieces based on a heuristic (four times the number of processors), and creates instances of the new step for each of the sub-ranges.

  One important use of `cncFor` is for spawning work in a more structured way, i.e. `cncFor 1 1000 (putt tags)` rather than `forM_ [1..1000]`

  ```
  cncFor :: Int -> Int -> (Int -> StepCode ())
         -> StepCode ()
  cncFor start end body =
   do tags <- graphInStep newTagCol
      graphInStep$ prescribe tags$ \(x,y) ->
         -- Here we execute a sub-range serially:
         forM_ [x..y] body
      -- Use the same splitting heuristic as the TBB auto-partitioner:
      let ranges = splitRange (4 * numCapabilities)
                              (start,end)
      forM_ ranges (putt tags)

  -- This allows dynamic graph extension:
  graphInStep :: GraphCode a -> StepCode a
  ```

  The short `cncFor` definition above is a testament to the easy extensibility of Haskell CnC compared to other incarnations of CnC. [3]

4. Implementation

Haskell CnC provides two *families* of implementations, both exposing the same monadic interface: `Intel.Cnc` and `Intel.CncPure`. For the remainder of this paper we will focus primarily on the IO-based `Intel.Cnc`, because it currently provides the best 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 parallel execution and synchronizing data access (items) vs. rebuilding them from scratch. The most simple (IO-based) implementation of CnC is nearly trivial—`forkIO` executes steps and `MVars` provide synchronization on missing items.

Laziness and the strictness of `put`

There is one fundamental impedance mismatch between a lazy language and the implementation techniques that have been designed for graph-based programs. Work on scheduling (6; 9) assumes that the computation represented by each graph node happens (to completion) when that node is scheduled. In a Haskell implementation it is easy to write CnC steps that do no real work at all, instead using `put` or `putt` to produce thunks that are not executed until later—when the parallel graph execution is finished!

To reduce the likelihood of this mistake we have made *put* and *putt* strict in their arguments. We have not currently opted for `deepseq` because we wish the user to retain some control. The *strategies* (19) approach may be useful for controlling the strictness of steps.

4.1 Runtime Schedulers

In Haskell CnC version 0.1.4, runtime schedulers are numbered rather than named. The idea being that there are many of them and they proceed in a sequence of implementations from less to more complex. Importing `Intel.Cnc3` or `Intel.Cnc10` employs scheduler 3 or 10 respectively. In this paper we discuss schedulers 3, 4, 7, 8, 9, and 10. These fall into three major families, based, respectively, on *IO-threads*, a *global task queue*, or *sparks*.

4.1.1 IO Threads (Scheduler 3)

GHC has very lightweight user threads; for a long time it won the "language shootout" Threadring benchmark. In scheduler 3 we map each CnC step onto its own IO thread (e.g. one `forkIO` per step). Therefore scheduler 3 is simple and predictable. 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.

```
-- 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 memoization)
-- and a list of steps that are controlled by the tag collection.
newtype TagCol   a =
        TagCol (IORef (Set.Set a), IORef [Step a])

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

Above are the core type definitions used in scheduler 3. The issue of memoization has not yet been discussed. Like quiescence, memoization is an optional property of the model; CnC can store the tags that have already been executed to suppress repeated execution. (This requires no storage for results, because the results of a step are already disseminated into the tag and item collections of the graph.) But schedulers that allow disabling memoization must also be somewhat looser with respect to allowing multiple puts. That is, multiple puts of the same key-value pair (i.e. overwriting the value with an equal value) are allowed.

4.1.2 Global task pool (Schedulers 4,7,10)

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.

[3] The one caveat for this simple implementation is that the programmer gets no guarantee as to deadlock-equivalence with a serial loop. A serial loop with inter-iteration put-get dependencies could be deadlock free, but its parallel counterpart could deadlock. To solve this problem, scheduler specific implementations of `cncFor` are necessary that have the ability to execute at the granularity of iteration ranges, but *block* at the granularity of a single iteration.

Schedulers 4 and 7 both use MVars for data synchronization. Steps use *blocking* operations (e.g. `readMVar`) to get items. How then to maintain the pool of worker threads? At start-up, all task-pool-based schedulers fork `numCapabilities` threads, but when a thread blocks on a `get`, it goes out of service for an arbitrarily long time. Therefore 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). Both schedulers 4 and 7 adopt a strategy of minimizing, but not preventing, 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.

Where schedulers 4 and 7 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. Scheduler 4 takes the former approach, scheduler 7 the latter.

Scheduler 4 does not support quiescence. Rather than complete execution *before* beginning the `finalize` action, it allows all worker threads to continue spinning (in a `Control.Concurrent.yield` loop) until the finalize action has completed. Once the final action has succeeded in all its `gets`, the scheduler can then "kill" the workers by setting a flag.

Scheduler 7 uses a different strategy. Whenever a worker observes the task pool in an empty state, it terminates and sends its ID number back to the scheduler thread through a `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, scheduler 7 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.

Schedulers 4 and 7 retain the same types for tag and item collections as scheduler 3. However, they add a state transform monad to store extra state about the graph execution. Below is the definition for `StepCode` (and `GraphCode` is just the same in this case).

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

The implicit state "`HiddenState`" stores five things:

- the task pool used in this graph execution
- the number of workers for this graph
- the "make worker" function to spawn new threads (given ID as input)
- the set of "mortal threads"
- the worker ID of the current thread

Scheduler 10: do-it-yourself data synchronization

Scheduler 10 departs from schedulers 4 and 7 by eschewing MVars for synchronization, instead *aborting* a step when a `get` fails. Aborted steps are registered in a *wait-list* on the missing item. 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. Other MVar-free Haskell CnC schedulers implement the replay approach, but scheduler 10 selects the continuation approach.

Scheduler 10 is implemented with a continuation monad transformer (`ContT`), providing a limited form of continuation-passing-style (CPS) and 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. (Lacking this ability, the C++ implementation of CnC uses only the replay approach.)

```
-- StepCode in scheduler 10 adds a second monad transformer:
newtype StepCode a = StepCode
       (ContT ContResult (StateT (HiddenState) IO) a)

-- GraphCode needn't capture continuations:
newtype GraphCode a =
        GraphCode (StateT (HiddenState) IO a)

-- Item collections now store wait-lists rather than MVars:
newtype ItemCol a b = ItemCol
       (IORef (Map a ((Maybe b), WaitingSteps b)))
```

Finally, note that work sharing with a global queue *should not*, in theory, scale as effectively as work-stealing. We will let the results speak for themselves, however. With Haskell CnC, we aim to do a broad empirical comparison of schedulers and to that end to include a wide range of scheduling strategies.

4.1.3 Spark-based Scheduling

GHC already includes an efficient implementation of work-stealing for pure computations. Each machine thread maintains a queue of *sparks* (7)—thunks which can be stolen and executed in parallel. The programmer can add to the spark queue using *par*. Why not simply create a spark for every step? In the `Intel.CncPure` implementation, this can indeed be done. But the IO variants described in this section have a problem: steps are implemented using IO. One is then tempted to try the following:

```
-- Incorrect! Don't do this in Haskell!
runStep ioaction =
   par (unsafeDupablePerformIO ioaction)
```

There's a fundamental problem with this. Spark pools in GHC are *lossy*. When a spark pool overflows, sparks are dropped. (Worse yet, in future releases of GHC spark pools may not even serve as roots for garbage collection.) They represent *optional* parallelism and are suitable only for thunks which will be executed on the forward path of the program with or without parallel steals.

Yet it is still possible to use spark pools to schedule CnC computations implemented with IO. In this subsection we describe an approach that accomplishes this using *Cilk-style* nested parallelism.

Scheduler 8: Cilk-Style

In Cilk (5) (an extension of C++), the unit of parallelism is the procedure. Procedures spawn potentially parallel sub-computations by annotating sub-procedure calls with `cilk_spawn`. Before the end of the caller's lexical scope a `cilk_sync` must occur—a barrier, blocking on all spawned sub-procedures. The result is a form of strictly nested parallelism wherein child computations must complete before the parent call can return.

We take a similar approach here, replacing procedures by CnC steps. To overcome the problem of dropped sparks, we allow a step to spark downstream steps but also tuck away those thunks in the `StepCode` monad's state. Thus, irrespective of whether sparked thunks are stolen, at the end of a step's execution it flushes its buffer of child computations by forcing each thunk (serially).

```
newtype StepCode a =
        StepCode (S.StateT (HiddenState8) IO a)

– The hidden state stores two things:
– (1) "Self": the current action, if needed for requeueing.
– (2) A list of child tasks/thunks that were spawned in parallel.
newtype HiddenState8 = HiddenState8 (StepCode (), [()])

– In this version we don't use MVars because gets don't block:
newtype ItemCol a b = ItemCol
        (IORef (Map.Map a ((Maybe b), WaitingSteps)))
```

Using this infrastructure, the scheduler begins a mostly depth-first traversal of a CnC graph by simply executing the initialize step. When the initialize step uses `putt` to produce tags, downstream steps are exposed for parallel execution via work stealing. These downstream steps become the children of the initialize step, and the initialize step their parent. If no stealing occurs, the graph is traversed in a depth-first order.

As in scheduler 10, a `get` operation on an unavailable item terminates the current step and registers it in a wait-list before "returning" to the parent. Thus that step becomes a *leaf* in the tree-shaped traversal of the graph. When a step performs a `get`, on the other hand, it simply spawns awoken steps as children. Unlike scheduler 10, scheduler 8 uses the *replay* method, restarting steps from the beginning when they are rescheduled after blocking on input data. Moreover, scheduler 8 must use **exceptions** to escape the current step at the point of a failed get. (An incomplete "scheduler 9" combines Cilk-style nested parallelism but with the `ContT` approach of Scheduler 10.)

Scheduler 8 has a advantage on CnC executions with (dynamic) step invocations in the shape of a tree. (Intuitively, thieves can steal entire subtrees rather than single steps as in schedulers 4,7, and 10.) However, there's a significant disadvantage as well, which affects the class of programs supported. The other schedulers described can all support *cyclic graphs*. Scheduler 10, in general, cannot—it will stack overflow. This limitation is ameliorated by an optional `tail_putt` variant of `putt` which can be used to inform the library that a particular `putt` is the last action within the enclosing step.

Discussion: Sparks vs. IO Threads

Note that Haskell applications can simultaneously contain both imperative parallelism (IO threads) and pure parallelism (par annotations). In the current GHC runtime, spark-based parallelism is *optional* and extra machine threads are only used for sparks if they are not already executing threads. Therefore, if large numbers of IO threads are used in the larger application context surrounding a CnC application, these will prevent the spark-based scheduler from achieving any parallelism. In the future it may make sense to for spark-based CnC schedulers to approach resource sharing more aggressively, for example, by explicitly creating worker threads that call `GHC.Conc.runSparks`.

4.2 Hash-Tables vs. Various Maps

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.

Yet 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, `Either`s, `Bool`s, 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 `newtype`s, 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
```

Finally, it is now 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
```

Pick Your Atomic Variable

Finally, another basic data structure trade-off is in what mutable pointer type to use to accomplish atomic updates. 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 (18), and select `TVars` by default.

4.3 Problems and solutions in GHC parallel performance

In short, Haskell CnC performs terribly with version 6.12 of GHC and quite well with the current development version (as of 06/10/2010). The key issue 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 5, but first we discuss the structure of the problem and its solution.

Ultimately, we believe targeting new, high-level parallel abtractions (like CnC) to GHC is a mutually beneficial prospect, as evidenced by Haskell CnC (1) highlighting these 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[4], 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 (7), 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*.
- 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.

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

	blackscholes	cholesky	mandel	primes	threadring
C++	293	390	147	84	63
Haskell	90	158	51	29	28

Table 1. A table comparing non-comment, non-blank lines of code in C++/CnC benchmarks and Haskell CnC. A 64% reduction in size is achieved on average, in large part by reducing type-definition and boilerplate code.

Following the implementation of the new scheme, we noticed significant improvements in many of the Concurrent Collections benchmarks (Section 5).

5. Evaluation

In this section we look at parallel speedups achieved on seven benchmarks. We measure the impacts of the implementation choices described in Section 4. Most of the benchmarks below are direct ports of their counterparts included the distribution package for the C++ version of CnC (see Table 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 18.4X (Figure 7).
- **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). Cholesky was sped up by a factor of 3.14 on a four-core machine (Figure 4).
- **N-body problem** – quadratic algorithm to compute accelerations on N bodies in 3D space. Maximum speedup achieved was 22.1X over single threaded execution (Figure 6).
- **Mandelbrot** – compute each pixel of a Mandelbrot fractal in parallel. Max speedup was 11.8X (Figure 5).
- **Primes** – naive primality test in parallel. Max speedup was 25.5X.
- **"Embarassingly parallel"** – simplistic microbenchmark that performs arithmetic in a tight loop on each of N tasks for exactly N cores. We also discuss a variant with alternating parallel and serial stages (Figure 10).
- **"sched_tree"** – a benchmark that, like the traditional "parfib" benchmark, computes near-empty tasks in a tree topology.

Experimental setup: We evaluate on three platforms: (1) *Desktop*, a 3.33 GHz quad-core Core i7, Nehalem architecture; (2) *Intel Server*, a 32-core platform consisting of four 2.27 GHz 8-core processors, Westmere architecture; and finally, (3) *AMD Server*, a 48-core system containing eight 6-core Istanbul processors running at 2.4 GHz.

GHC 6.12 Results: Our first measurements on 6.12 (Desktop) were all over the map—no speedup on N-body, only meager speedups on mandel (2.3X) and primes (1.5X, see Figure 3). Further, hyperthreading caused a total collapse in performance. Even embarassingly parallel was showing spurious serializations where one of N tasks would run in serial before branching out to other threads.

GHC Development Version Results: Switching to the development version of GHC immediately brought mandel and primes (two benchmarks with relatively large numbers of steps) up to 2.82X and 3.64X speedup, respectively. This improvement is due to the new BLACKHOLE architecture (Section 4.3), which we verified by rewinding to exactly before and exactly after that patch to the compiler.

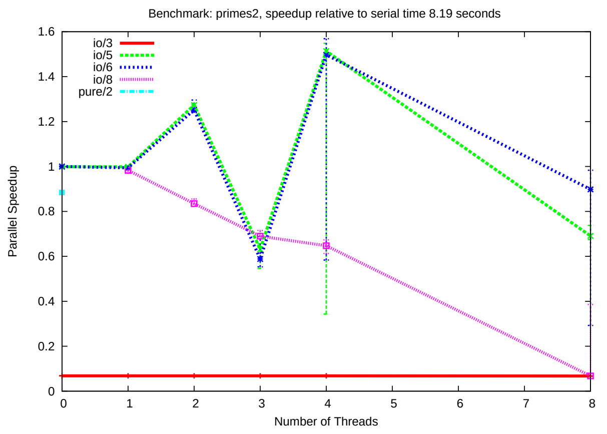

Figure 3. Undesirable parallel speedup results under GHC 6.12.1, Desktop configuration. Schedulers are denoted "io/4" for IO-based scheduler number 4. Five runs per data-point, error bars representing *min* and *max* execution time. The data-points at "8-threads" represent hyper-threading. Likewise, zero on the X-axis refers to building the application without `-threaded`.

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. The effects of suggested heap size (`-H`) on parallel performance (32 threads) 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.

This change greatly improved our parallel performance overall, but there still remain quirks and unpredictable outcomes, as illustrated by Figure 9. One source of especially quirky behavior are loops that run for a long stretch without allocating. These threads cannot yield until they allocate, and other threads may end up waiting on them to initiate a global garbage collection.

GMaps: Again, tracking mandel and primes, the speedups improved to 3.19X and 3.69X, respectively, after adding GMaps. Both mandel and primes can benefit from `Data.IntMap`-based item collections.

TVars: Switching from IORefs to TVars for hot variables (with atomic operations) improves the aforementioned speedup to 3.31X and 3.76X, respectively.

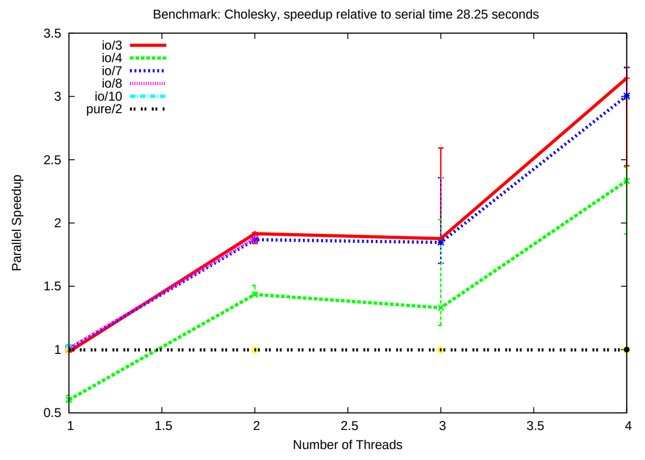

Figure 4. Cholesky benchmark, Desktop configuration.

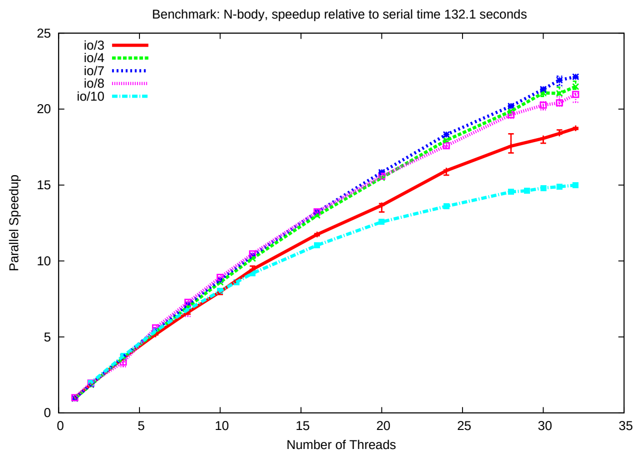

Figure 6. N-body benchmark, Intel Server.

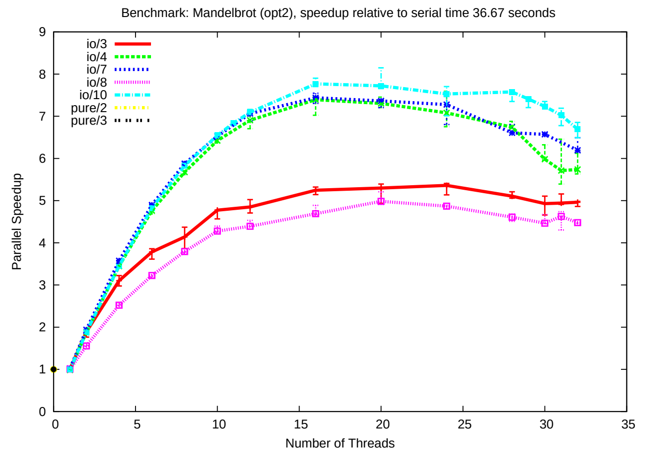

Figure 5. Mandelbrot benchmark, Intel Server.

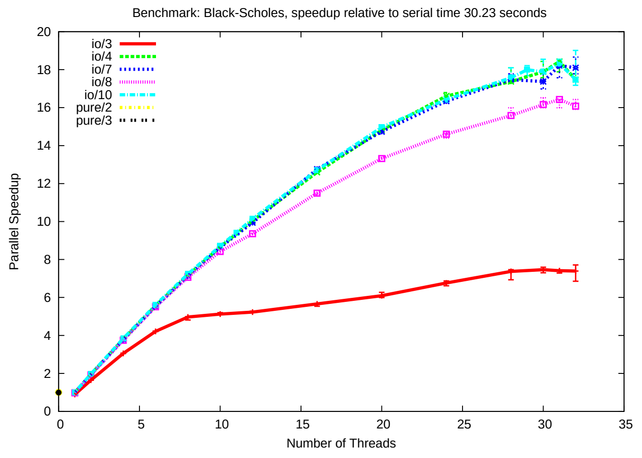

Figure 7. Black-Scholes benchmark, Intel Server.

The effects of garbage collection

The architecture of the garbage collector (GC) is the primary barrier to scaling parallelism to larger numbers of cores. At the time of writing, GHC is using a stop-the-world parallel garbage collector (11) 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 (12), and this strategy has resulted in reasonable scaling on small numbers of processors.

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.)

In the future, the GHC developers hope to modify the garbage collector to allow 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 Intel Server: using a 10G heap did not significantly lessen the

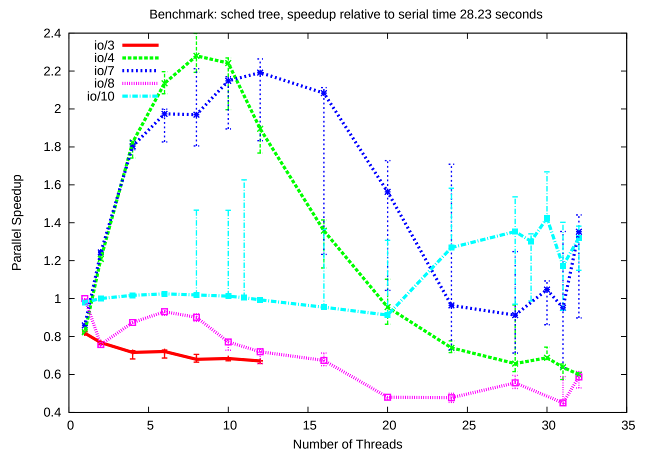

Figure 8. Sched_tree benchmark, Intel Server. We would not expect global task-pool based schedulers to do well. But scheduler 8 should be doing better.

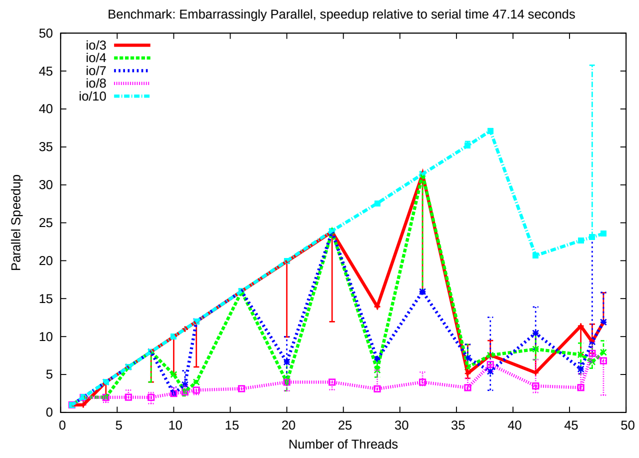

Figure 9. Detailed speedup results: embarrassingly parallel benchmark, all schedulers (AMD Server). "Zig-zags" are indicative of remaining problems, especially with allocation-free loops.

performance (with one thread or 32) for all schedulers *except* number 3, where the larger heap size had a catastrophic effect. (It became 2X slower in the serial case, 15.8X slower in the 32-thread case.)

Detailed Benchmark Discussion

As usual, benchmark performance is largely a measure of effort spent improving and tuning a given implementation. We try to report accurately the degree of implementation effort invested in these benchmarks. Granularity of computation is a universal problem in parallel scheduling. Black-Scholes and Cholesky, 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 solving the granularity problem but requiring manual tuning of block size. Neither of these benchmarks was modified substantially after the initial port. Black-Scholes performed well from the start, but Cholesky is much more unpredictable. It achieves speedups on the Desktop configuration, but not at all on either server configuration with the same matrix and tile size. At larger matrix sizes, Cholesky could show an extremely modest (less than 4X) speedup on the server configurations.

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 slowdown), 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 (4)) as opposed to static scheduling (e.g. StreamIT (6)).

But, in addition to granularity, allocation and garbage collection bring a second complication to bear. 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 (except modestly under `Intel.CncPure`). 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.

Finally, we include a simple microbenchmark, "sched_tree", similar to the "parfib" benchmark used elsewhere in the literature. Sched_tree generates a tree of step invocations. A system like Cilk can achieve linear speedup on such a benchmark, but we had no such success (Figure 8), even with scheduler 8, which *should* have worked performing only a constant number of steals. In fact, scheduler 8, while being the conceptually most promising and (in principle) scalable scheduler, was a flop. This could perhaps be due to the use of exceptions to escape step executions, or to some other bottleneck we've yet to identify.

One reason a range of implementations is useful here is that our intuitions have been so far off. As another example, scheduler 10 was supposed to be substantially more efficient than other task-pool based schedulers that use MVars. And while it did well some of the time, other times it lagged behind, as in N-body (Figure 6).

Notes on parameters and tuning

We experimented systematically with some runtime parameters and informally with others. The GHC runtime can optionally make use of the 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).

6. 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

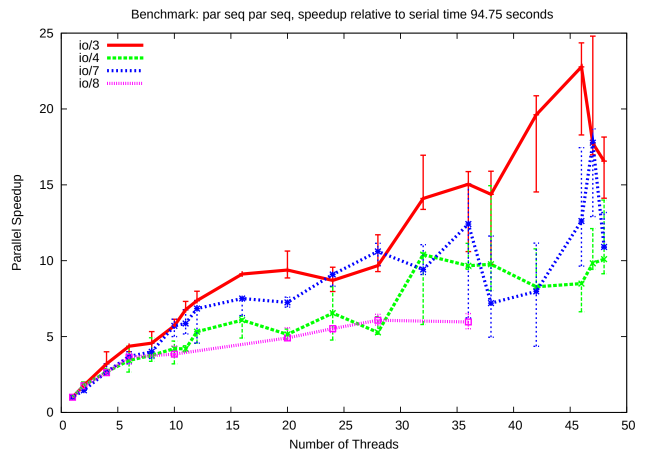

Figure 10. "Par_seq" variant of embarrassingly parallel benchmark, AMD Server. This benchmark alternates serial and embarassingly parallel phases of computation. It is the one benchmark where scheduler 3 does relatively well on parallel speedup. It shows the importance of task *topology* in determining scheduler performance.

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.

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.

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 (10) and "task scheduling" (17), or change dynamically (as in most streaming databases (1)).

Stream processing systems typically have ordered edges and statically known graph topologies. Generally, they allow both stateful and stateless tasks (the later providing data parallelism). They may be based on synchronous data-flow (e.g. StreamIt (6)) 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 (2)). Or, finally, edges can be processed by a node in a nondeterministic order, as when handling real time events or interrupts (e.g. WaveScope 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 (3; 16), but not all projects have achieved (or aimed for) effective parallel performance.

7. Conclusion and Future Work

With a modest amount of tweaking we have been able to get significant parallel performance out of Haskell CnC benchmark applications. CnC makes it easy to write parallel programs, but performance predictability remains a key issue. Fortunately, there are already several changes on the horizon which will help the situation.

Our implementation has benefited from a series of improvements described in this paper. There remain a number of low-hanging fruit, however, in terms of increasing the performance of Haskell CnC. In the future we (or others—the project is open source) will look at using other data structures such as `Data.Judy` arrays as candidate item collections. Also, we are far from the last word on scheduling. If spark-based schedulers cannot be made to perform as desired, it may be necessary to create our own work stealing deque, or find a way to reuse the one present inside GHC's RTS.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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 program-*

ming languages and operating systems, pages 151–162, New York, NY, USA, 2006. ACM.

[7] 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.

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

[9] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. *SIGPLAN Not.*, 43(6):114–124, 2008.

[10] 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.

[11] 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.

[12] 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.

[13] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 45–54, New York, NY, USA, 2009. ACM.

[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] NVIDIA. CUDA reference manual. Version 2.3, 2009.

[16] 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.

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

[18] 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.

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

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