A Comparison of Lock-based and Lock-free Taskpool Implementations in Haskell

Michael Lesniak
University of Kassel
Research Group Programming Languages / Methodologies
Wilhelmshöher Allee 73
Kassel, Germany

Abstract
Today, synchronization of shared data structures in multithreaded software is mostly implemented using locks, which leads to difficult to understand and error-prone programs. Software Transactional Memory allows lock-free concurrent programming by handling the synchronization of shared variables implicitly. We show how to implement different instances of the widely-used taskpool pattern (global and private taskpools with and without task stealing) using both lock-based and lock-free synchronization mechanisms in the functional programming language Haskell. We examine their performance using two synthetic algorithms and LU decomposition and report our observations about parallel performance and the complexity of the implementation. Our results show that lock-free taskpools are not only on par with lock-based implementations concerning parallel performance but are also easier to comprehend and develop.

Keywords: Concurrent Programming, Taskpools, Haskell

1. Introduction
Today, one of the hardest problem in parallel programming is to keep data that is shared by multiple threads consistent. The classic (manual) approach is to restrict access to the data by wrapping the modification right around a lock: Before a thread is allowed to modify data, it has to obtain the lock. Until the lock is released no other thread is allowed to modify the shared data (depending on the particular scenario, even reading can be unsafe). This technique has well-known drawbacks that makes the development of correct and performant parallel programs difficult, even for experts. A modern approach in concurrent software development named Software Transactional Memory[1], coming historically from database research, uses the idea of transactions to allow concurrent access to shared data structures without explicit locking and unlocking; instead, functions access shared data structures in transactions and the runtime systems takes care of consistency issues, e.g. by restarting transactions in case of conflicts.

In this paper we analyze the advantages and disadvantages of the lock-based and lock-free synchronization models with respect to performance and ease of development by implementing variants of the well-known taskpool pattern[2] in Haskell.

---

Email address: mlesniak@uni-kassel.de (Michael Lesniak)
A taskpool[3] is a design pattern to distribute independent tasks to threads: it stores processable tasks in a shared data structure, such that threads can take tasks out of and add new tasks to the pool. Taskpools are good candidates to compare the parallel performance of the two synchronization models: First, the number of tasks and thus accesses to a taskpool is often large. The underlying parallel runtime system and especially the building blocks of the particular synchronization mechanism are therefore constantly under load. Second, the application of taskpools in real-world applications is enormous (for example, Java’s Executor Framework[4] is based on this pattern). Results of a comparison have therefore practical relevance both for the implementation of similar parallel programs and parallel runtime systems.

Haskell[5] is a pure and non-strict functional programming language with support for different concurrency approaches (see [6, 7, 8]), especially supporting both lock-based and lock-free concurrent programming. It is far from obvious how the taskpool pattern, typically implemented in imperative languages, maps onto the respective (parallel) Haskell counterparts and which implementation delivers the best performance.

Our contributions in this paper are:

- The implementation of three taskpool variants in Haskell. We examine global taskpools and private taskpools with and without task stealing.
- A comparison of the parallel performance of these taskpools with focus on the different synchronization models. For benchmarking we used three example problems: two synthetic algorithms with varying task structures and LU decomposition.
- A list of observations comparing the development effort of both approaches.

To briefly state our results, the development of parallel programs using STM is easier and more comprehensible when compared with standard lock-based approaches and the performance of lock-free taskpool variants is comparable to manual locking.

The rest of this paper is structured as follows: In section 2, we explain the used Taskpool variants. In Section 3 we describe the various implementations. In Section 4 we explain our benchmark design and discuss the results of the benchmark outcome. Section 5 compares related work and Section 6 concludes and gives an outlook to possible future work. Due to the lack of space we refer to [7, 9] for an introduction to concurrent programming using MVars and to [8, 10] for STM, respectively.

2. Taskpool variants

While selecting and implementing the taskpools we were guided by three requirements: First, the taskpool implementation should not imply a large overhead, leaving most of the processing time for the actual calculations. Second, the taskpool’s interface should be consistent and simple to understand; different variants and their implementations should be interchangeable to ease experimentation. Third, the functioning of the taskpool should be task independent. Besides simple task structures it should support complex task structures, where subtasks are spawned, dependencies between tasks exists and advanced termination detection is needed. We study three well-known variants of taskpools: With a global taskpool all threads access a single homogeneous data structure to take tasks and add new subtasks. A global pool is sufficiently performant when accesses to the pool are infrequent. If threads access the pool often, they can delay each other due to the needed synchronization. A private taskpool uses a data structure that consists of a public pool as well as private thread-local storage for each thread. On each access to the public pool a set of tasks is transferred into the private storage. Successive takes are served by this storage until it becomes empty. A private taskpool prevents frequent synchronized accesses to the public pool. If tasks are large, taking a set of tasks of the public pool can leave idle threads workless. A shared taskpool resembles a private taskpool but has an extended operation to get new tasks: If the public pool is empty, an idle thread accesses the private storage of a still working thread and steals some of its tasks. This approach prevents idle threads but enforces additional synchronization on thread-local storage.
3. Taskpool Implementations

In this section we describe a general taskpool interface for Haskell and lock-based and lock-free implementations for global, private and shared taskpools with a focus on synchronization.

3.1. The Abstract Taskpool Interface

Our common interface to all taskpools consists of two parts: a multiparameter typeclass with functional dependencies defines the necessary functions of a taskpool and a monad defines the environment in which these functions are executed. An application can thus be programmed against the typeclass and use the monad while the different taskpool implementations can be changed transparently. The typeclass is defined by

```haskell
class Taskpool pool task | pool -> task where
    put :: task -> TPMonad pool ()
    get :: TPMonad pool (Maybe task)
    wait :: TPMonad pool ()
```

where `put` adds a task to the taskpool and can both be called by the main thread for initialization and by a working thread. `get` returns `Just t` if a task `t` is available and `Nothing` if the calculation is finished. If the pool is empty and other tasks are still working, `get` blocks until either all threads are finished or a new task becomes available. The function `wait`, only used by the main-thread that created and initially filled the taskpool, blocks until all tasks have been processed. It is used to wait both for the the finish of the final calculation and for fulfilling of task dependencies between iterations (see below).

The monad `TPMonad` is used to hide pool-internal data from the interface. It encapsulates a state containing a thread-specific index (Section 3.2 explains its necessity) and the used pool and is defined by

```haskell
type TPMonad pool a = StateT (Int,pool) IO a
```

The following example shows the taskpool independent code for the synthetic algorithm (described in Section 4.2), clarifies the use of the described operations and demonstrates the simplicity of their usage:

```haskell
synthetic :: Taskpool pool Int => Int -> Int -> pool -> IO ()
synthetic constFactor initTask pool = do
taskpool pool $ do
    for KN numCapabilities (thread (task constFactor))
    put initTask
    wait

task :: Taskpool pool Int => Int -> Int -> TPMonad pool ()
task constFactor t = do
    if t > 0 then do -- Use seq to force evaluation of simulated computation.
        calcPi (1*constFactor) 'seq' put (t-2)
        calcPi (5*constFactor) 'seq' put (t-1)
        calcPi (10*constFactor) 'seq' return ()
        else calcPi (10*constfactor) 'seq' return ()
```

In the example, `numCapabilities` worker threads are forked; `numCapabilities` is set on the command line using `-N<threads>` and sets the number of operating system threads used by the parallel runtime. The functions `taskpool`, `forkN` and `thread` are internally defined helper functions: `taskpool` encapsulates the Taskpool monad functions and initializes the taskpool state, `forkN` forks a number of threads whereby each thread receives a running index and `thread` passes received tasks to the given function until the pool is empty. After the worker threads are forked, the initial task `initTask` is put in the pool, and the main-thread waits until all tasks have been processed.

Our approach supports dependencies between parallel phases (called iterations) as for example needed in the parallelization of the LU decomposition: an iteration consists of initial tasks added to the pool by the main-thread and waiting for these tasks to be processed. In the following example the main-thread waits until all tasks for a particular `i` and all `j` have been processed before adding tasks for the next iteration:
3.2. Lock-based Taskpool variants

We describe the implementation of the global taskpool in detail since it is the basis for the other variants, and explain private and shared pools by focusing on the differences and additions.

3.2.1. Global Pool.
An efficient implementation of global pools needs to prevent busy-waiting and implement general termination detection. After a description of the pool’s type we will describe the general control flow of the taskpool operations and explain in detail how these problems are solved using the information stored in the type.

The global pool is defined by a type

```hs
data GPool a = GPool { -- put get wait
    gChan :: Chan a -- W R
    , gWork :: MVar (Set ThreadId) -- RW RW RW
    , gState :: IORef GState -- R W
    , gWait :: IORef [MVar ()] -- RW RW
    , gFinish :: MVar (MVar ()) -- W R
} }
```

The comments after each field mark the different access modes for each of the taskpool functions, R stands for reading access, W for writing and RW for both. We used different MVars to synchronize parts of the global state instead of one single lock that protects the whole state. This reduces the time each thread needs to hold locks.

The channel gChan stores tasks of an arbitrary type a and is the only part of the pool responsible for task storage; the other parts are used for synchronization and termination detection. A channel is used solely for its FIFO-like interface and its simple usage. In retrospect, Data.Sequence would have been a viable alternative that allows additional access patterns (e.g. LIFO) with similar complexity.

The pool data structure is concurrently accessed by multiple threads and access is synchronized by the lock gWork. Besides being the global lock, we use gWork to store the set of ThreadIds of working threads. We have chosen to specify thread identifiers instead of counting working threads because it allows a more thorough overview of the taskpool’s functionality; since the number of threads is small and the set-operations insert and delete are O(log n), usage did not induce a measurable performance penalty.

The taskpool can be in one of two states, Put or Wait. Initially, before wait in the main-thread is called, it is in Put-state: initial tasks can be inserted into the pool by the main-thread, idle worker threads need to wait if they can not get tasks. After the pool changes into Wait-state, all initial tasks for the current iteration have been inserted. In Wait-state, when the pool is empty and all threads are idle, the next iteration can start.

In general, there are two possibilities to block a thread that is waiting on an event: First, the thread can create an empty MVar and block until it is filled (blocked waiting). Second, the thread can repeatedly acquire a lock, check if the event occurred, release the lock, wait some time and retry again (busy waiting). By using busy waiting, the event source does not need to be aware of all listeners, but the manual lock-check-unlock cycle creates unnecessary locking operations. Since performance was important, we used the first approach, which is implemented as follows: the thread that holds the global lock and wants to wait creates a new empty MVar w. Depending on the event it wants to be informed of, it either adds w to gWait to be informed on the arrival of new tasks or sets gFinish to be informed on the start of a new iteration (see get-operation below). Afterwards it releases the global lock and tries to read w. The thread is later unblocked when a value is written to its MVar.

The put-operation works by writing a new task into the channel and unblocking a waiting thread (in gWait), if one exists (Figure 1, put).
When the wait-operation is called the pool’s state is changed to Wait. An empty MVar is collected for each finished and idle thread using gFinish. Afterwards, the pool’s state is changed to Put to recreate the initial state for a possible next iteration, and threads are unblocked (Figure 1, wait).

The get-operation is called by each thread to receive a new task or to be informed that the current iteration has been finished. Since termination detection is implemented in get, it is the most complex of the three operations (Figure 1, get): After the pool structure is locked, the availability of unprocessed tasks is checked. If tasks are available, one is read and returned. If the pool is empty but the pool’s state still in Put-state or other tasks are still working and could add tasks, the thread starts waiting. If on the other hand all threads have finished their work, the remaining one is the last non-waiting thread; after creating a MVar for gFinish, it unblocks one of the other waiting threads until all are unblocked.

3.2.2. Private Pool.

A private taskpool is similar to a global pool but has in addition to the pool data structure a thread-local (non-synchronized) storage for tasks. The implementation was complicated because all details of the local storage should be hidden from the forked function. The type PPool is extended by an additional array pPrivate as follows:

```haskell
data PPool a = PPool {
  -- ... same as GPool
  , pPrivate :: IOArray Int (ArrayList a)
  , pSize :: Int
}
```

ArrayList is a custom data structure in the IO monad based on IOArrays which supports additional functions, e.g. getting blocks of elements. We generate a thread-specific index in forkN and modify the initial state for each forked thread. The get function can therefore access the thread-local ArrayList by reading the value at its specific index of pPrivate. This approach has the advantages of being performant and easy to implement.

3.2.3. Shared Pool.

A shared pool uses the same approach as the private pool but allows threads to access the local storage of still working threads to steal tasks; other implementations divide the thread-local pool in a private and shared pool (cf. [11]) but since we were mainly interested in the synchronization behavior we implemented sharing as follows: The thread-local ArrayList needs to be extended to be concurrency-safe; the SPool-type has the additional attributes:

```haskell
data SPool a = SPool {
  -- ... same as PPool
  -- not needed anymore: pPrivate :: IOArray Int (ArrayList a)
  , sPrivate :: IOArray Int (ArrayListLock a)
  , sSize :: Int
}
```
The ArrayListLock is the lock-based equivalent of a non-synchronized ArrayList, where each of the access functions uses a lock to prevent concurrent access. When the private pool is empty, a thread tries to steal half of all tasks of a randomly chosen working thread.

3.3. Lock-free Taskpool Variants

In the following sections we describe our implementation of the lock-free taskpool variants, where we used STM instead of manual locking. After an explanation of the global lock-free taskpool we briefly describe the differences of the other variants to the lock-based approaches.

3.3.1. Global Pool.

The lock-free global taskpool uses the same ideas and techniques as the lock-based one, but we need fewer fields:

```haskell
data STMGPool a = STMGPool { -- put, get, wait
  stmChan :: TChan a -- W, R, R
, stmState :: TVar STMState -- R, W
, stmFinished :: TChan (TMVar ()) -- W, R
, stmWorking :: TVar (Set ThreadId) -- RW, R
} data STMState = SPut | SWait deriving Eq
```

The fields follow the same naming scheme and serve the same purpose as in the lock-based example, the main differences are:

- the use of STM-supported types (TChan, TVar, TMVar), substituting Chan and MVar.
- no explicit locking. We leave the details of keeping the pool structure consistent to the runtime system.
- no explicit blocked waiting. Instead of having an explicit list of waiting threads, a thread calls `retry` if it wants to be informed of events and automatically blocks until then.

These advantages directly map onto the implementation complexity of the taskpool-specific operations: the put-operation does not need to inform waiting threads about new tasks and the get-operation does not need to keep track of manual notifications for waiting; all operations ignore the various lock- and unlock occurrences.

3.3.2. Private and Shared Pools.

From a synchronization point of view the implementation of private and shared pools is identical. Both pools use a structure `ArrayListSTM` to store thread-local data; instead of an `IOArray`, a `TArray` is used internally. Since the synchronization is done by the STM implementation, we did not need to develop both a locking and non-locking version for the shared and private pools, respectively.

4. Benchmarks

We tested the different taskpool implementations against each problem on a 2.2 GHz 8-core AMD Opteron 875 running Linux-kernel 2.6.18 with GHC 6.12.1. We ran each benchmark, consisting of a particular example problem, a taskpool variant and its synchronization model, five times from one to eight cores and used the mean value for speedup calculation. The basis for the speedup calculations were sequential versions of each problem. The local storage had space for 256 tasks for all taskpool variants. We chose our instance sizes such that the absolute runtime on eight cores was around thirty seconds.

We chose example problems for using the taskpools based on the following considerations: First, the problems should be processable independently of problem-dependent data structures: A problem where each task must store its result in a set of synchronized linked lists would be unfavorable, for instance since the additional synchronizations are non-trivial and allow many taskpool-independent optimizations[12]. Second, the tasks should have varying and possibly unpredictable computation time, such that the approach of using taskpools is reasonable. Third, the tasks generated by the problem should not use much (dynamic) memory: since we test the implementations with a large number of tasks, it would stress the garbage collector and thus complicate the interpretation of the results even more. Since the observations and their interpretation are similar, we explain the π-calculation-benchmark in detail in the next section and focus mainly on the differences for the other ones.
4.1. Calculating Digits of \( \pi \)

Our most basic and somewhat artificial example task refers to the calculation of \( \pi \) to an arbitrary number of digits. Benchmark problems consist of a list of numbers that each specify the calculation of \( \pi \) to the given precision. This problem has the following properties: First, tasks do not spawn subtasks, hence if the pool is empty, all given tasks have been processed. Second, since only arithmetical operations are used, the calculation is memory efficient. Third, since no subtasks are spawned we can manually assign the tasks to threads beforehand, leaving out the taskpool. Therefore we can benchmark the maximal speedup and deduce the overhead of the taskpools for this particular case.

We benchmarked the \( \pi \)-calculation using two scenarios: first, using 131272 tasks with a random task size of 100±10 (\( \text{pi-small} \), ca. 0.001s/task), second with 8192 tasks with a random size of 1000±100 (\( \text{pi-large} \), ca. 0.12s/task). Since the tasks in \( \text{pi-small} \) are so short-lived, the pool is accessed frequently to obtain new tasks. This effect is weakened in \( \text{pi-large} \). To compare the taskpool’s overhead, we split the tasks manually to threads beforehand in an additional benchmark run. Table 1 shows the speedup for \( \text{pi-small} \) and \( \text{pi-large} \); values larger than 1.0 for one core in the lock-based variants are measurement errors. In the following we describe our observations and give possible explanations:

a) For short-lived tasks the speedup is better when a lock-based variant is used: Since accesses occur extremely often, the impact of the overhead of the STM implementation is significant.

b) The lock-based shared pool is better than manual distribution for small tasks: Since tasks are randomly generated, threads become idle when they have processed their individual chunks. This is prevented when threads are allowed to steal tasks.

c) The global lock-free pool performs better than the lock-based one: The speedup for both variants shows that the lock-free taskpools scales better with many threads. We think that STM’s optimistic transactional approach[8] and earlier restart of waiting threads (namely, when a transactional variable is changed) are the reasons for this observation.

d) For large tasks all lock-free variants perform better than their lock-based counterparts: the pool is not accessed as often since the computation time per task is larger, hence the impact of the lock-free overhead is not as severe. At the same time, the STM variants still scale better (see c)), thus delivering the better speedup.

4.2. A Synthetic algorithm

The next example problem is a synthetic algorithm[13] that involves spawning of subtasks, where each task \( A(i) \) typically generates two new subtasks \( A(i-1) \) and \( A(i-2) \): In the base case \( i \leq 0 \) one final calculation (\( \{10/f\} \)) is done and no subtasks are spawned. In the normal case \( i > 0 \) increasing sized calculations are interleaved with spawning of two smaller-sized subtasks (\( \{1/f\}A(i-2)\{5/f\}A(i-1)\{10/f\} \)). The values in curly braces describe simulated compu-
tational intensive tasks (we chose the calculation of $\pi$ as indicated above). By varying $f$, the amount of computation per task can be modified, by varying the initial $i$ the number of tasks and the degree of irregularity can be modified.

We benchmarked the synthetic algorithm using two problem instances: first, with very small tasks (default size of 1) and a depth of 24 (196418 tasks) (\textit{syn-small}), second with larger tasks (default size 100) and a depth of 15 (2584 tasks). The results are shown in Table 2. In addition to the mostly similar results of a) to d) we make the following observations:

e) The performance of the private and shared lock-free variants for small tasks is low. We think that the impact of the overhead for (synchronized) thread-local storage is even higher: since the tasks are smaller than in pi-small, the storage is accessed more often due to the shorter computation time.

f) The private variants are slower than the respective global pools: Since tasks are stored thread-local and task stealing is not enabled, large tasks remain unreachable for idle threads.

4.3. LU Decomposition

Solving a system of linear equations described by $Ax = b$ with $A \in \mathbb{R}^{n\times n}$ and $x, b \in \mathbb{R}^{n}$ is common in many scientific applications. A well-known approach is the Gaussian Elimination Algorithm and its specializations. If the system has to be solved for multiple $b$, a more efficient approach than solving the system repeatedly uses the LU-decomposition of $A$.

There are efficient approaches to parallelize LU decomposition. Due to the available space we omit a thorough description and solely describe the task structure. We refer to [3] for an overview and an extended description of our approach. There are two distinct differences in comparison with the previous example problems: First, both the number of tasks and the computation time per task becomes smaller with progressing decomposition. Second, the calculation works in data dependent parallel phases that we call \textit{iterations}. To calculate values for the current iteration, all values from the previous one are needed; hence, additional data dependencies occur and all taskpool implementations need to support iterations and waiting on their processing. We explicitly want to clarify that our chosen parallelization does not achieve optimal parallel performance; for better performance one would switch to a sequential algorithm when tasks become too small. Nevertheless, we chose this particular example to examine task structures of this type.

For benchmarking the LU problem we measured the decomposition time for a randomly generated $5000 \times 5000$ matrix. Table 3 shows the speedup. We made the following observations:

g) The drop in speedup for eight cores for the lock-based variant is a well-known problem with the parallel runtime system of the used GHC version[14]; although this bug has been fixed in later GHC versions, we were not able to use them at the time of benchmarking due to technical reasons. Currently we can not explain why the drop does not happen with the lock-free variants and instead they even gain performance.

h) All implementations have low speedup. The sequential implementation was a straightforward nested loop implementation of solely math operations that perform naturally fast. We believe that the overhead of the taskpools in
conjunction with the difficult task structure leads to these results and that the better performance of the lock-based implementations is due to the better handling of short-lived tasks (see a)) and utilization of thread-local storage.

### 4.4. Summary

The benchmarks have shown that for typical scenarios of task pools lock-free programming is performance-wise a viable alternative to the traditional lock-based approach. The gap between lock-based and lock-free implementations for corner cases can possibly be closed with the implementation of native thread-local storage (either through a library or compiler extension) or performance improvements for \texttt{TArrays}.

### 5. Related work

We chose to examine classical taskpools to compare the performance of lock-based and lock-free synchronization models because there exists an enormous amount of parallel algorithms that use the taskpool pattern (see [15, 3] for an overview) and our work eases the transformation of such algorithms into Haskell. Since we want to reduce the development overhead and the difficulty of interpreting the benchmark results we made the design choices outlined in Section 3. An extensive discussion of the large taskpool design space can be found in [16].

Due to the high level of abstraction that he Haskell language provides, many more abstract models for parallel programming exist: The de-facto standard Glasgow Haskell compiler supports two other parallelization models that are feasible for simple task structures (when tasks do not spawn subtasks, involve additional operations in the \texttt{IO} monad or need access to shared data structures). By using \texttt{data parallel Haskell}, vectorized operations can be executed very fast; approaches to port this model to GPUs exist[17][18]. By using \texttt{semi-implicit} parallelism[6] with the \texttt{par}-operator, it is possible to leave all details of parallelization besides annotating potential sources to the compiler. An analysis of different problems solely for semi-implicit parallelism can be found in [19]; the examined problems from the nofib-benchmark suite have similar irregular tasks, i.e. calculation of mandelbrot sets, matrix multiplication and raytracing. Intel’s Concurrent Collection Library show a new message-based high-level approach on implementing parallel algorithms while providing more control over the parallel execution and granularity[20]. The parallel programming language Eden defines additional constructs on top of a Haskell-like language to control the parallel evaluation of expressions[21]. It especially makes the definition of skeletons[2] easier and allows the general and performant definition of e.g. taskpool-based algorithms[22].

An analysis similar to ours for concurrent linked-list implementations, but using different implementations (STM, \texttt{MVars} and \texttt{IORefs}), can be found in [12]. The authors came to similar results concerning the scalability of STM, even though the different variants perform quite differently, hence a future comparison of these variants in the context of taskpools would be interesting. The idea of comparing different taskpool variants and irregular tasks with synthetic problems has been examined for Java in [13] and OpenMP in [11] and came to similar results concerning the differences between global, private and shared taskpools.

### 6. Conclusion and Future Work

We have shown how to implement different variants of the taskpool pattern (global and private pools with and without task stealing) in Haskell using both manual lock-based and automatic lock-free approaches. The lock-free taskpools were implemented using Software Transactional Memory (STM). We benchmarked each taskpool using two synthetic problems and the LU decomposition of a matrix. We made the following observations:

<table>
<thead>
<tr>
<th>Variant</th>
<th>Lock-based</th>
<th>Lock-Free</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>global</td>
<td>0.64</td>
<td>1.14</td>
</tr>
<tr>
<td>private</td>
<td>0.64</td>
<td>1.14</td>
</tr>
<tr>
<td>shared</td>
<td>0.63</td>
<td>1.18</td>
</tr>
</tbody>
</table>

Table 3: Speedup for LU-decomposition
The implementation of parallel programs using STM is easier and less error-prone than their lock-based counterparts, since the developer does not have to keep synchronization issues in mind.

The performance of lock-based and lock-free taskpools is comparable. Our benchmark results suggest that STM-based implementations currently scale better with a large number of cores.

The reasons for worse performance were not in the concept of STM per se, but lay either in implementation details, for example, inefficient arrays or very unusual scenarios, i.e. with task durations around $10^{-3}$ seconds.

Finding performance bottlenecks is difficult in the lock-based and even more difficult in the lock-free synchronization models, since there exist no advanced profilers or other development tools for either implementation.

Our results suggest that the implementation of STM in GHC is mature enough to use it to develop parallel programs that are easier to comprehend and perform comparable to their lock-based variants.

We see more topics for future research on lock-based and lock-free taskpools in Haskell: first, it remains interesting to examine if the lock-free variants continue to scale better with more cores, e.g. on Intel’s ManyCore architecture[23], which recently made GHC on their platform available. Second, by implementing more performant alternatives for thread-local storage for STM, the advanced taskpool variants should deliver better parallel performance. Third, since taskpool are so widely used and our results are equal to those of imperative languages, developing and analyzing advanced taskpool variants in Haskell is a viable alternative to implementations in imperative languages. Fourth, exploring the advantages and disadvantages of the different taskpools by implementing additional real world examples, e.g. from [15], looks promising.

5. S. Marlow, Haskell 2010 language report.
11. A. Wirz, M. Siß, C. Leopold, A comparison of task pool variants in openmp and a proposal for a solution to the busy waiting problem, Int. Workshop on OpenMP.
14. GHC Bug #3553: parallel gc suffers badly if one thread is descheduled. URL http://hackage.haskell.org/trac/ghc/ticket/3553