Applying Dataflow and Transactions to Lee Routing

Chris Seaton, Daniel Goodman, Mikel Luján, and Ian Watson

University of Manchester
{seatonc,goodmand,mikel.lujan,watson}@cs.man.ac.uk

Abstract. Programming multicore shared-memory systems is a challenging combination of exposing parallelism in your program and communicating between the resulting parallel paths of execution. The burden of communication can introduce complexity that is hard to separate from the pure expression of the algorithm and can negate the performance that is gained from parallelism. We are extending the Scala language with dataflow for creating parallelism and transactions for the controlled mutation of shared state. We take an early look at applying this work to Lee's algorithm for routing circuit boards and consider the potential benefits of programming with this system with regard to the elegance of expression and the resulting performance. We show how our approach reduces the number of lines of code and synchronisation operations needed, at the same time as improving real-world performance.

1 Introduction

A great many strategies have been proposed to make writing parallel programs that run on multicore shared-memory systems easier and less error prone, at the same time as achieving a good return on the invested number of processors. This endeavour becomes more pressing as multicore systems become the majority on servers, desktops and mobile devices. The number of cores looks likely to continue to increase, requiring more parallelism in our programs in order to exploit this power.

We have looked at work to combine two existing constructs that have shown considerable potential on their own – dataflow [11] and transactional memory [4]. We have used an established benchmark, Lee's algorithm for routing printed circuit boards, to make an early assessment of their utility for creating efficient, simply written and correct parallel programs. This paper shows how using the DFLib and MUTS [3] implementations of dataflow and transactional memory makes the parallel implementation of our program simpler, at the same time as achieving a real world performance increase compared to coarse locks on typical desktop hardware, even when all overhead is included.

Presented at the Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG), January 2012, Paris

Sections 2 and 3 briefly describe the concepts of dataflow and transactional memory, and the implementations that we employed in this work. Section 4 describes Lee's algorithm and why we and others are using it to evaluate transactional programs. Section 5 explains how we implemented Lee sequentially, using coarse locks and then using transactions and dataflow. Section 6 considers what difference this approach makes to the programr and the performance achieved. Section 7 considers future directions and Section 8 concludes.

2 Dataflow

Dataflow decomposes a program into a directed acyclic graph with nodes that perform computation and edges that take the output of one computation and provide it as input to one or more other nodes. A node is runnable if all of the nodes preceding it have finished their computation, and so all of the inputs are available. In a model where the computations do not have any side effects, such as in functional languages, the graph completely expresses dependencies between nodes, and if more than one node is runnable at the same time then they may be run in parallel. The name 'dataflow' emphasises that it is the data flowing between nodes that causes them to be scheduled to run, rather than the a program counter reaching procedures and causing them to run, as in the von Neumann model.

As well as helping to expose parallelism by dividing a program into groups of computations without interdependencies, dataflow can also help us by handling the synchronisation between running threads. Threads will only run when all of their inputs are already available, so there is no code needed to achieve the common barrier or join operations as in Java threads. However, dataflow does not help us to address the problem of a shared mutable state. As we shall show using our example problem, Lee's algorithm as described in Section 4, shared state can be a key part of the algorithm.

For this work we used the Scala dataflow library, DFLib, being developed as part of the Teraflux project at the University of Manchester. A dataflow node is created by wrapping a Scala function in a `DFThread` object that has methods to either explicitly set an input, or to link an input to the result of another `DFThread`. This allows a dataflow graph to be built up and implicitly run by DFLib, which will schedule runnable functions using enough OS threads to occupy all hardware threads.

3 Transactions

Transactional memory allows a series of memory reads and writes to be executed as if they were a single indivisible, or atomic, operation. This removes the need for explicit mutually exclusive locks around data structures. Instead, a data structure is modified within an atomic block and so appears to be a single

operation that can be applied without excluding others. There are many different algorithms that implement this basic transactional behaviour [4], but this programming interface is common to most of them.

Locks can be a blunt tool that disallows concurrent access to memory based on the assumption that such accesses will conflict. For some applications, this may be a correct assumption, but given a particular problem we may be confident that conflict is rare. Most implementations of transactional memory allow transactions that do not have conflicting memory accesses to proceed in parallel, dealing with the less common case of conflicting memory access by restarting one or both of the transactions involved. This is in contrast to a standard locking approach which would disallow parallel memory accesses on the same data structure, even when they do not conflict.

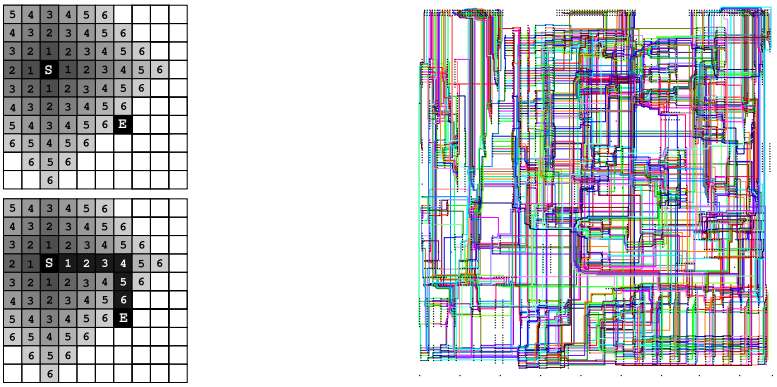
For the work presented here we used the Manchester University Transactions for Scala (MUTS) library. Taking the Java Deuce transactional memory library [6] as a starting point, MUTS extends and modifies this work to implement a selection of different techniques for implementing software transactional memory for the Scala programming language.

MUTS uses a Java agent to visit all Java classes as they are loaded into the JVM. This allows it to create a transactional version of all methods that instrument read and write operations. MUTS can then pass these reads and writes to one of several implementations of software transactional memory algorithms included in the library. A typical algorithm stores reads and writes in a log, deferring writing to shared memory from the log until the transaction is complete and values in the read log are verified to not have been written to by another thread in the elapsed time.

4 Lee's Algorithm

Lee's algorithm [7] solves the problem of finding independent routes between a set of pairs of points on a discrete grid. The applications originally proposed included drawing diagrams, wiring and optimal route finding, but the algorithm is now best known as a method for routing paths on a printed circuit board. There are later algorithms for route finding with less computational complexity, but Lee produces a shortest solution for any given route and board state, as opposed to using heuristics to arrive at a good-enough solution in less time. Figure 1b shows the output of one application of Lee's algorithm – routing paths on a printed circuit board – as generated by our implementation.

A simple overview of the Lee algorithm is that from the start point of the route it looks at all adjacent points and marks them as having cost one. From each point P so marked, it marks each adjacent point P_{adj} as having cost $cost(P_{adj}) = cost(P) + 1$, as long they were not already marked with a lower cost. If an adjacent point is already part of another route then using that point would cost more, by an arbitrary constant factor, as a bridge of one route over another needs to be built. This expansion, as it is called, continues until the end point is a member of the set of adjacent points. Typically, this expansion forms



(a) Basic expand and trace phases [1]

(b) Output from our output implementation

Fig. 1: An illustration of the basic expand and trace phases of Lee's algorithm and output from our implementation

a circle around the start point, enveloping obstacles such as existing routes, and finishing at the end point.

A trace is then run from the end point to the start point, always moving to a point of lower cost until the start point is reached. This produces a route which can be marked on the grid. These two key steps are illustrated in Figure 1a.

Unlike classic problems such as finding Fibonacci numbers or merge sort, Lee does not belong to the class informally known as 'embarrassingly parallel'. Such problems can be broken down into smaller problems that are entirely independent, and so can be executed in parallel. There are several ways to decompose Lee's algorithm into a set of subproblems, but it cannot be guaranteed a priori that any two subproblems will not want to lay a route in the same cell of the shared grid, even when the route's start and end points are known.

The key to the problem is that any set of subproblems still need to share access to a single resource – the grid. It is not simple for each thread to have an independent copy of the grid, as two threads could need use the same point for multiple routes and would then have to synchronise between themselves. This would add logic to the program that is unrelated to the algorithm that we are implementing. It is also not simple for each thread to have one part of the larger grid, as you can not guarantee which parts a route will use before the expansion has been calculated and such a scheme would vastly increase the complexity of the program. However, given all routes on a circuit board it is unlikely that any two being routed at a particular time will conflict. The parallelism is there – it is just that it is hard to determine statically and is more apparent as the program is running.

Work has already been done to use Lee to evaluate the runtime characteristics of a transactional program [12], and to evaluate the performance of implementa-

tions of transactional memory for Java [1]. Our work builds on this by evaluating the combined use of transactions and dataflow, with Scala, MUTS and DFLib.

5 Implementation

Scala [9] is a hybrid functional and imperative programming language that runs on the Java Virtual Machine. The functional aspect allows a clean expression of a problem in a way that is open to parallelisation with minimum shared or mutable state. The imperative aspect allows us to have controlled mutation of the shared state when we require it for efficient execution.

We wrote a set of programs in Scala to solve Lee. One sequential, `seq`; one using coarse locks, `coaselock`; one using the MUTS library, `muts` and one using both MUTS and DFLib, `dataflow`. They share common input and output formats and many internal data structures. The output of the parallel implementations is non-deterministic due to the interleaving of separate threads of execution, so we check the the total routing cost of the solution to compare the equivalence of different implementations.

5.1 Sequential

We first created a sequential implementation of Lee using a purely functional approach, combined with a central mutable data structure to represent the board. Our implementation of Lee is minimal and we excluded some refinements that Lee described such as weighting against turns in paths. While these refinements are sensible for actual routing applications, they do not have any effect on the parallel characteristics of the program and can be considered constant factors to performance in both space and time. We also allow only one level of bridging, where one route can cross an existing route, as this is usually sufficient for our test boards.

The sequential program, `seq`, represents a clear and succinct expression of the algorithm. We are adding parallelism because we want the program to run faster, not because the algorithm requires it, so ideally we want this optimisation to require minimum new code and minimum modifications to existing code. The perfect expression of parallelism would be completely orthogonal to the sequential expression of the algorithm.

5.2 Coarse Lock

Our first parallel implementation, `coarselock`, creates a thread for each hardware thread in the system. Each thread works in a loop. First it allocates a route from the input and obtains a private copy of the shared board data structure. Then it expands and traces the route, using this private copy. The route is then validated, to ensure that since the private copy was made the board has not been changed by another thread to make the proposed solution invalid and then commits the route to the shared board data structure.

There are three resources being shared here – the board data structure, the list of available routes and the list of solutions. Additionally, the master thread needs to know when all the solutions are in the list, which we achieve by waiting for all threads to finish with the `join()` method.

We make access to these shared resources mutually exclusive using Scala's implementation of the M-structure [2], `SyncVar[T]`. For example, an instance of `SyncVar[MutableBoardState]` holds the shared mutable board state data structure. When a thread wants to lock the data structure so it can validate its route and commit it to the board without interruption by another thread, it calls the `take` method to atomically remove the data structure and leave the `SyncVar` empty. Any other thread trying to read or write from the same data structure will block within `take` until the former thread is done with the data structure and calls the `put` method to return it for other threads to use.

```
val boardStateForFreeze = boardStateVar.take()          // Lock
val privateBoardState = boardStateForFreeze.freeze
boardStateVar.put(boardStateForFreeze)                  // Unlock

val expansion = expandRoute(board, route, privateBoardState)
val solution = traceRoute(board, route, expansion)

val boardStateForLay = boardStateVar.take()             // Lock
val verified = verifyRoute(route, solution, boardStateForLay)
if (verified)
    layRoute(route, solution, boardStateForLay)
else
    scheduleForRetry(route)
boardStateVar.put(boardStateForLay)                     // Unlock
```

In this implementation we had a single lock for the entire board. Another option would be to use multiple locks to control different parts of the board. We haven't created such an implementation, but we do refer to this strategy in the analysis section.

5.3 MUTS

We used the MUTS library to create a parallel implementation by modifying `coarselock`. Where `coarselock` acquires a resource with `take` to the exclusion of all other threads before `put`ting it back when it is done, we can instead perform that action inside a transaction that will only allow other threads to read the same values as long as they don't write to them, and will automatically retry if such a conflict is found.

```
// Atomically copy the shared data structure
val privateBoardState = atomic { boardState.freeze }

val expansion = expandRoute(board, route, privateBoardState)
val solution = traceRoute(board, route, expansion)

// Atomically write to the shared data structure
atomic {
```

```
    if (verifyRoute(route, solution, boardState))
        layRoute(route, solution, boardState)
    else
        scheduleForRetry(route)
}
```

This code looks very similar to the use of a `pthread_mutex_t`, a Java `synchronized` block or a `SyncVar` as used in `coarselock`. The idea that `atomic` could be implemented as a global single lock is one model for thinking about transactional memory [4]. However, in practice the `atomic` blocks will allow multiple threads to read at the same time, and to write at the same time as long as they do not try to write routes using the same point. If there is a conflict, they will be retried, just as `coarselock` does explicitly.

5.4 Dataflow

We then extended our `muts` implementation to use the Scala dataflow library, DFLib. Where the `muts` implementation created parallelism by spawning Java threads, the dataflow constructs provided by DFLib allow us to express this creation of parallelism in a different way. Each route is a `DFThread` that will have its inputs ready at the start of the program's execution and so will all be runnable. DFLib will schedule them for us so that only a sensible number are running at any time.

A final `DFThread`, the 'collector thread' will be then created that has each route's `DFThread` as one of its arguments. This will therefore be run when the solutions are complete. This is a convenience construct provided by DFLib, as it is expected to be a common pattern, and replaces the synchronisation needed to create the list of solutions and the `join` operation to wait for all threads to finish that we used in `coarselock`. Figure 2 shows the resulting dataflow graph, and illustrates how the scheduler executes a small subset of routes at time.

```
// Accepts solutions as arguments and build a list from them
val solutionCollector = DFManager.
    createCollectorThread[Solution](routes.length)

for (route <- routes) {
    // Create a thread to solve a route
    val routeSolver = DFManager.createThread(solveRoute _)

    routeSolver.arg1 = board
    routeSolver.arg2 = route
    routeSolver.arg3 = boardState

    // It will send the solutions to this function
    routeSolver.arg4 = solutionCollector.token1
}
```

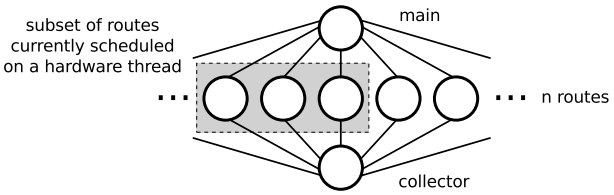


Fig. 2: The dataflow graph created by `dataflow`

6 Analysis

6.1 Results

The programs were compiled with Scala 2.9.1 and run on the JVM 1.6.0_27 on an Intel Core i7 processor comprising four physical cores, each with two-way hyper-threading for eight hardware threads. The OS was openSUSE 11.2 with Linux 2.6.31.14.

Table 1 shows the running time of the core part of each program, measured using `System.nanoTime()`. This excludes the startup time for the JVM and in the case of `muts` and `dataflow`, time to rewrite classes for transactional access. Each program was run ten times with mean average and standard deviation taken, rounded to three decimal places. programs were constrained to use 1, 2, 4, 6 or all 8 of the available hardware threads by replacing calls to `availableProcessors()` with a constant value. These results indicate the relative performance than can be achieved within a longer running system. All parallel implementations show a decrease in performance when using all available hardware threads. This is in line with other researchers' findings [8] and is probably caused by higher contention on resources, limits of hyper-threading, and time sharing with the kernel and other system processes. We show these results as they are what would be achieved with a simple implementation that would by default try to use all available hardware threads.

Impl.	Hardware Threads									
	1		2		4		6		8	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD
`seq`	29.355	0.310	29.396	0.212	29.535	0.303	29.376	0.220	29.330	0.215
`coarselock`	30.374	0.321	17.485	0.396	14.986	1.266	13.748	0.776	21.961	1.550
`muts`	31.422	0.421	16.648	1.157	13.357	0.493	11.528	0.425	14.869	0.683
`dataflow`	32.093	0.282	16.994	1.149	13.630	1.105	11.805	0.460	14.570	0.401

Table 1: Mean algorithm running time (seconds)

Figure 3 shows the resulting speedup of the algorithms, compared to the sequential implementation. That is the time for the sequential implementation divided by time for each other implementation, for a given number of hardware threads. These results indicate the return on investment for the number of hardware threads applied to the problem.

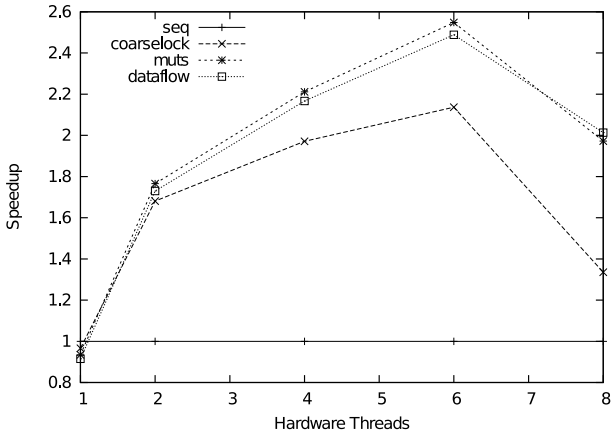


Fig. 3: Algorithm speedup compared to sequential

Table 2 shows the running time of the entire program shown next to the algorithm time, when running on 8 hardware threads. This includes JVM startup (common to all implementations), and for `muts` and `dataflow` the time to rewrite classes for transactional access. These results are indicative of the total real-world cost of implementing Lee's algorithm for a medium sized board in each of the implementations, including all overhead of the libraries that we have employed, and the rewriting cost involved in MUTS. Rewriting could be done ahead of time, and we show these numbers here to give an indication of this cost, whether it is made at runtime or not. The overhead becomes a less significant proportion of whole program time given larger problem sizes of a greater number of problems processed in batch.

Implementation	Algorithm	Whole program	Overhead
`seq`	29.330	29.697	0.367
`coarselock`	21.961	22.634	0.673
`muts`	14.774	17.136	2.267
`dataflow`	14.215	18.877	4.307

Table 2: Whole program running time on 8 hardware threads (seconds)

Table 3 shows some informal metrics of the program code required to implement the different parallel algorithms. 'Parallel operations' refers to the number of operations at the source code level related to nothing but the parallel architecture around the pure algorithm. This includes creating or joining a thread, reading or writing a synchronisation variable and atomic blocks. Each parallel operations detracts from the pure algorithm and is a potential source of error.

All of the parallel implementations are correct and usable, but we had two goals that we can analyse them against. First, the only reason for creating parallelism was that we wanted the program to run faster than our sequential version. We therefore

Implementation	Lines of code	Parallel operations
`seq`	251	0
`coarselock`	330 (+79)	11
`muts`	328 (+77)	6
`dataflow`	300 (+49)	5

Table 3: Code metrics

analysed their performance against the sequential implementation, given a multicore system. Secondly, we said that ideally we didn't want to distract from the elegant expression of the algorithm that the sequential implementation gives us. We wanted to introduce minimum new code and to modify minimum existing code. Each addition or modification further ties the expression of the algorithm and the parallelism together and makes modification or debugging to the algorithm itself harder. We therefore analyse the changes needed to create the different parallel implementations.

6.2 Coarse Lock

Implementing Lee's algorithm using coarse locks would probably be the default approach by most industrial programrs. Shared data structures has been identified and locks placed around them. This has created a more verbose program with 11 parallel operations.

The key problem with `coarselock` is that all threads need to read and write the board to make any progress, and given that we are making that access mutually exclusive, we are only allowing one thread to make progress at a time. There is still a degree of work that can be done in parallel – after creating a private copy of the board, the expansion can proceed in parallel – but when we add more threads trying to complete the work faster, we just end up with a bigger queue for the board lock.

As described in Section 5, it would be possible to develop a finer grained system of locks. However, `coarselock` is already the most complicated of the programs that we have written, as shown in Table 3, and that is with just one lock. Multiple locks would require logic to work out which locks to acquire, as well as a protocol for the order in which to acquire the locks in order to avoid classic deadlock. Even if it tested well, how would we gain confidence that our locking protocol worked every time?

6.3 MUTS

The `muts` implementation looks similar to `coarselock`, with the same thread creation and the same points of synchronisation on the same data structures. However, as we are using transactional memory, the semantics of the code is very different, and will not block another thread unless there is a conflict in the memory locations that they want to read and write. The `muts` implementation achieves better performance from essentially the same code as in `coarselock` because the MUTS `atomic` block will allow more than one thread to run inside it, as long as the are not conflicting on the memory that they use, which as we already described, is unlikely.

Software transactional memory introduces a significant overhead to programs, in that within a transaction all reads and writes have to be instrumented and logged. This

will entail allocating and populating a data structure in proportion to the number of reads and writes the transaction performs. For example, to create a private copy of our test boards involves creating a log with a not-insignificant 600^2 read entries. In MUTS, there is also the overhead at runtime of rewriting all class files to include transactional variants of methods used within transactions. In this evaluation we are looking at the resulting performance of the program, so we included all of these overheads in our whole program timing measurements. This transformation can alternatively be made ahead of time for a known set of class files. Even with all of the overhead, `muts` still runs significantly faster than `seq` and `coarselock` on 8 hardware threads.

6.4 Dataflow

The dataflow implementation uses the same code as `muts` to synchronise access to the shared board data structure, but by structuring the program as dataflow we simplify the parallel parts of the algorithm. As we create one `DFThread` for each route, we have removed the synchronisation needed for sharing the available remaining routes between worker threads. By creating a final `DFThread` to collect the results we have also removed the synchronisation needed there. DFLib can manage the scheduling and dependencies of both of these two problems for us. This reduces the number of parallel operations to a lower number than that of `muts` alone. We would argue that where less parallel constructs are required, development is easier and there is less possibility for error, as has been found empirically by other researchers [10].

7 Further Work

7.1 DFLib

Our implementation of Lee's algorithm uses dataflow to express only a couple of simple dependencies, and although this is a legitimate use of DFLib that does reduce the volume of code needed for synchronisation and a work-queue, it is likely that a more advanced dataflow decomposition of Lee's algorithm, or another problem entirely, will reveal much greater gains in elegant expression and runtime performance available using DFLib.

7.2 MUTS

MUTS will currently make any read or write operation within a transactional block part of the transaction, regardless of whether or not the object is shared or mutable. MUTS could be improved with static analysis to reduce the size of the read and write sets. Other transactional memory implementations such as that of Haskell achieve this with explicitly typed transactional objects [5], but if this was the case with MUTS we could not have used our existing sequential board data structure without modification, and modifying it to use a transactional type such as Haskell's `TVar` would have been more work unrelated to the actual algorithm.

7.3 Other Algorithms

It is likely that other algorithms with similar properties are also well suited to dataflow with transactions. It would be interesting to investigate exactly what these properties are and which algorithms they apply to, so that when they are observed a dataflow-transactional approach can be recommended.

8 Conclusions

Our evaluation shows that dataflow combined with transactional memory is a succinct and efficient method for a parallel implementation of Lee's algorithm and is worth further development and investigation.

When applied to Lee's algorithm, dataflow and transactions allow a parallel implementation that is closest to the original sequential implementation. This makes any modifications needed to the algorithm simpler, as one has to consider less parallel code, and it reduces the chance of error as there are less instances of their use that could be incorrect.

These methods expose more parallelism than simple coarse locks and even with runtime transactional overhead the core of the implementation always runs faster than coarse locks, and with 8 hardware threads will run faster even when time consuming rewriting is included in timings.

We believe that transactions and dataflow in Scala using MUTS and DFLib can be used in other research development and real world applications to express parallel programs with minimal modifications and extra code, while achieving good comparative performance and speedup.

9 Acknowledgements

The authors would like to thank the European Communitys Seventh Framework program (FP7/2007-2013) for funding this work under grant agreement no 249013 (TERAFLUX-project). Chris Seaton is an EPSRC funded student. Mikel Luján is supported by a Royal Society University Research Fellowship.

References

1. Ansari, M., Kotselidis, C., Jarvis, K., Luján, M., Kirkham, C., Watson, I.: Lee-tm: A non-trivial benchmark for transactional memory. In: Proceedings of the 7th International Conference on Algorithms and Architectures for Parallel Processin (2008)
2. Barth, P., Nikhil, R., Arvind: M-structures: Extending a parallel, non-strict, functional language with state (1991)
3. Goodman, D., Khan, B., Khan, S., Kirkham, C., Luján, M., Watson, I.: Muts: Native scala constructs for software transactional memory. In: Proceedings of Scala Days (2011)
4. Harris, T., Larus, J., Rajwar, R.: Transactional Memory. Morgan & Claypool, second edn. (2010)
5. Harris, T., Marlow, S., Peyton Jones, S., Herlihy, M.: Composable memory transactions. In: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming (2005)
6. Korland, G., Shavit, N., Felber, P.: Noninvasive concurrency with java stm. In: Programmability Issues for Multi-Core Computer (2010)
7. Lee, C.Y.: An algorithm for path connections and its applications. IRE Transactions on Electronic Computers (1961)
8. Marlow, S., Peyton Jones, S., Singh, S.: Runtime support for multicore haskell. In: International Conference on Functional Programming (2009)

9. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. artima, second edn. (2010)
10. Pankratius, V., Adl-Tabatabai, A.R.: A study of transactional memory vs. locks in practice. In: Symposium on Parallel Algorithms and Architectures (2011)
11. Watson, I., Woods, V., Watson, P., Banach, R., Greenberg, M., Sargeant, J.: Flagship: a parallel architecture for declarative programming. In: Proceedings of the 15th Annual International Symposium on Computer architecture (1988)
12. Watson, I., Kirkham, C., Luján, M.: A study of a transactional parallel routing algorithm. In: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (2007)