Bringing Software Transactional Memory to Julia

David Orion Girardo

Massachussets Institute of Technology, Cambridge, MA

Abstract

While robust abstractions exist for parallel SIMD algorithms, writing concurrent programs is notoriously difficult, and teasing MIMD performance from these programs even more so. As a result, many obviously parallel problems are left unoptomized due to the programming complexity. In particular, managing mutex locks and race conditions is not only complex, but unscalable, in the sense that two working lock-based implementations cannot be composed in a safe way. Software transactional memory (STM) is an alternative abstraction for speculative concurrency and parallelism that resolves these scalability issues. However, STM has historically proved tricky to implement performantly in common scientific languages. Julia's unique combination of expressive macros and efficient user types provides a promising foundation for effective STM. This paper explores the design space for STM in Julia, and proposes an implementation.

Contents

1 Introduction

From the programmer's perspective, code is organized into atomic transaction blocks, such that the system is never left in an unfinished transaction state. Concurrency/parallelism is achieved by scheduling threads eagerly (but not necessarily greedily - the scheduler is left unspecified to allow optimizations), rather than waiting for locks to release. Instead of locks, if a concurrent process becomes invalid because of a change in global state, it will retry with the new values. This kind of speculative parallelism means processes never need to wait for input, so in practice often outperforms hand-made lock-based implementations. Most importantly, it is composable: any thread-safe STM algorithm will still be thread-safe when combined with another STM algorithm. This is not true of other lock-based implementations, and allows great scalability from the programmer's perspective, allowing algorithms that would be too hard to parallelize to at least gain a modest speedup with very little effort. Programmers also need not think about complicated locking dependencies, and the system will never deadlock (however, some care must be taken by the scheduler to avoid live-lock, where a long-running task is repeatedly trumped by many short running tasks). In practice, concurrent programs structured around Software Transactional Memory reduce bugs and reduce development time compared with lock-based implementations, but tuning performance can be tricky [12]

2 Background

STM was once very popular in the C/C++ community, drawing research from industry heavyweights such as Intel Corperation [13] [11] and Sun Microsystems [7]. Interest in production has declined due to the general difficulty of implementing and scheduling STM in a strongly stateful environment without garbage collection, as C family encourages [4]. Many of these projects (such as the Intel Transactional C++ Compiler) were largely abandoned by commercial enterprises, and relegated to the fringes of academia, though more recent developments have seen a unified push for transactional support in C++ [1]. Since however, it has become widely popular in the functional programming communities, such as the Haskell [5] and Clojure [15] ecosystems, as pragmatic tools for concurrency management because of the amiability to garbage collection, fast green threads [14], and a stateless style. [9] Because of Julia's garbage collection facilities and expressive internal expression transforms [2], it provides a powerful foundation for STM, rivaling that of functional-first languages.

3 Technical Approach

3.1 User Interface

Transactional abstractions are exposed to the programmer via the @stmatomic and @stmvar macros. Users wrap atomic expressions in the @stmatomic macro,

and it is converted to a form where Transactional guarentees (that the shared state is only modified if the entire transaction succeeds) are enforced. Expressions within the @stmatomic block may refer to global variables as if they were the only user, provided that the variables were forward declared with an @stmvar declaration. Variables reads and assignments for such variables within the @stmatomic block are automatically converted into the implementation-depended *readstm* and *writestm* expressions. An example Julia Software Transaction looks as follows:

Currently there are two main implementations, which expose a different @stmatomic macro, utilizing the Task or Shared Array backend respectively. Future work remains to unify these implementations into a single transactional type, which dispatches on an implementation parameter.

3.2 Task Implementation

The basic reference implementation utilizes Julia's **Task** library coming with the standard Julia distribution. Julia **Task**s provide lightweight concurrency facilities via "green threads", based on context switching of coroutines. As such, they do not provide true parallelism natively, but are still useful for enforcing transactional concurrency without excessive overhead. They also prove the simplest to implement, since much of the capability is built into the **Task** infrastructure already.

There are three main features necessary to build Software Transactional Memory on Julia's **Task**s: Variable versioning, variable commits, and retries.

Versioning is implemented in two parts. The global version is provided by a TVar wrapper type encapsulating a shared resource together with its public version number.

```
type TVar{T}
        v::T
        version::Int
        varID::Int
end
```

@stmvar declarations like

```
@stmvar x::Int = 3
```

are converted into initialization of a TVar object

```
TVar(3,0,hash("x")
```

as well as adding the (hash of the) var's name to the global list of names to be converted inside an @stmatomic block. Locally, each @stmatomic block creates its own STMTable, which is just a hashmap from global variable names to version numbers. *ReadTVar* then, in addition to returning the value from the TVar, implicitly reads the global version number into the local version table. *WriteTVar* resynchronizes the atomic thread by locking the TVar and comparing its global version number to the local version number. If the global

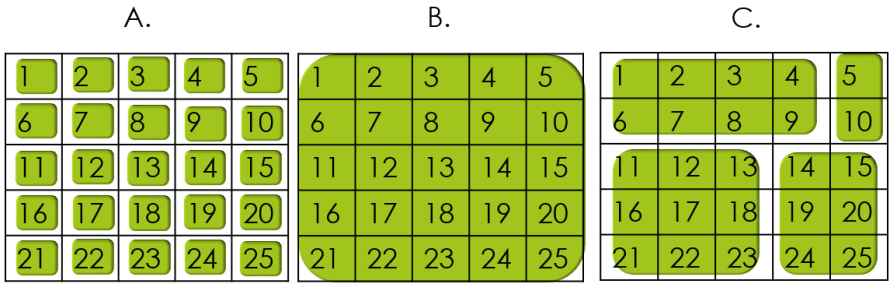

Figure 1: Example chunkings of a 5×5 **SharedArray**,
with (A.) *discrete* chunking, (B.) *indiscrete* chunking, and (C.) *hybrid* chunking

version number is higher than the local, this means that another transaction has executed before this one has finished. To be conservative, we assume that such cases invalidate the transaction, so the new TVar is read in via *ReadTVar* and the computation is restarted. If the version numbers are equal, the local value is written and the global version is incremented, releasing the lock. It should be noted that even though we use an explicit lock here, because of the restricted nature of our usage (only updating the TVar version and value with no extra computation), there is no chance of deadlock, since the computation holding the lock always has all resources available to complete the computation (the new TVar value and version number), and never needs to wait on another computation.

3.3 Shared Array Implementation

The Shared Array implementation is built on top of Julia's Shared Array facilities, and the Task implementation detailed above. The key value here is that Shared Array allow true shared-memory parallelism, rather than simply coroutine-based concurrency abstractions. Since all the values in a Shared Array are logically independent, there are multiple choices for transactional boundaries, depending on which parts of the array should be considerd distinct transactions. Different choices for transaction boundaries can be seen in Figure 1. Each choice groups some indicies together, so that they are tracked using the same versioning. Thas has obvious performance implications, as recording more versions has overhead, but finer grained transaction boundaries prevent unecessary transaction abortion, when a computation only touches a small number of indices (for example, updating a single user in a massive medical database). The choice of the optimal boundary choice, given some partially known access pattern is known as the "Chunking Problem".

The simplest case can be called the "indiscrete" chunking, where all indices are versioned together. This is logically equivalent to putting the entire array inside a TVar. The other extreme is "discrete" chunking, where each index gets

its own version. To accommodate both styles, the **TVar** type is modified to include an index table (a partitioning on the index set). This choice has the benefit of automatically generalizing to more complex irregular chunkings like Figure 1 C, as well as removing memory overhead by allocating only one global TVar record, rather than one for each chunk. Otherwise, the version tracking mirrors that of the Transaction implementation, with *ReadTVar* reading in versions for a specified index range, and *WriteTVar* checking that the version for those indices was not changed before committing. The user should be mindful to read as many indices in at once as possible, to minimize the number of calls to *ReadTVar*. The beauty of the Shared Array implementation is that parallelism is handled implicitly by the indexing mechanics dispatched by the Shared Array type, so that it can be otherwise utilized the same as a standard array.

Currently, the choice of chunking is built into the @stmvar macro, rather than being exposed to the user. This is just a simplifying choice made for the reference implementation, and could be just as easily replaced by a macro with more parameters to decide chunking. Ideally, static analysis would decide a good trade-off without extra parameters, hiding the complexity from the user - optimal chunking remains a research problem.

4 Comparisons

Benchmarks were utilized from the STAMP [10] (*bayes* and *k-means*) and RMS-TM [8] (ScalParC) benchmark suites, comparing both Julia implementations against niave non-parallel algorithms, and Intel's C++ compiler.

Some of the most interesting benchmarks (for example, a fully functional Quake server) were highly nontrivial to implement. As such, due to time constraints, we only consider this subset of benchmarks, but feel that they cover the range of transaction profiles well enough to provide a good comparison.

Note that benchmarking was performed on an AMD FX-8320E, which, while presenting 8 logical cores to the Operating System, actually has 8 integer processing units but only 4 floating point processing units. While interesting to consider the behavior of the floating point benchmark past 4 cores, this is a reflection of the quirks of the FX family's threading capabilities, not of the Julia Software Transactional Memory implementation. Unlike the other two, the ScalParC benchmark as in Figure 4 can be seen to scale past 4 cores, indicating that its bottleneck is on integer performance.

4.1 Discussion

It is worth noting that a message passing implementation is possible, where standard threads are used rather than Julia's **Task**s. However, we do not attempt it here, because the necessary copy semantics would negate the performance benefit of shared memory, and so would be more clearly and efficiently written as an explicit message passing algorithm, through MPI for example. If persistent data structures are substantiated in Julia, it may be worth revisiting a message

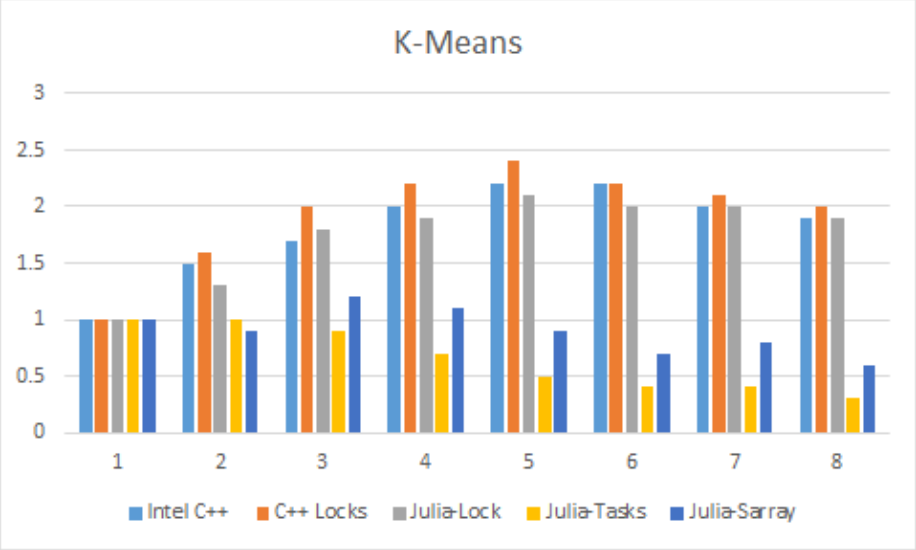

Figure 2: Speedup/Cores chart for K-Means benchmark

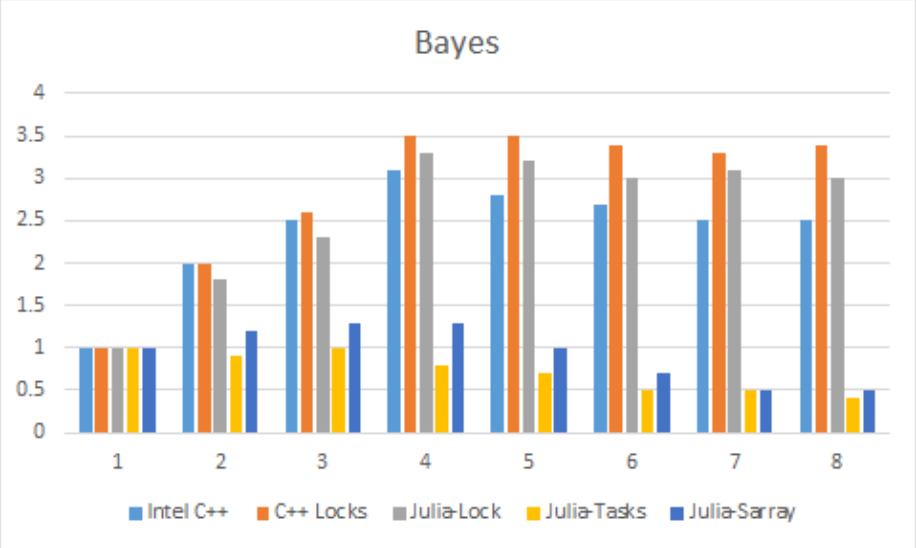

Figure 3: Speedup/Cores chart for Bayes benchmark

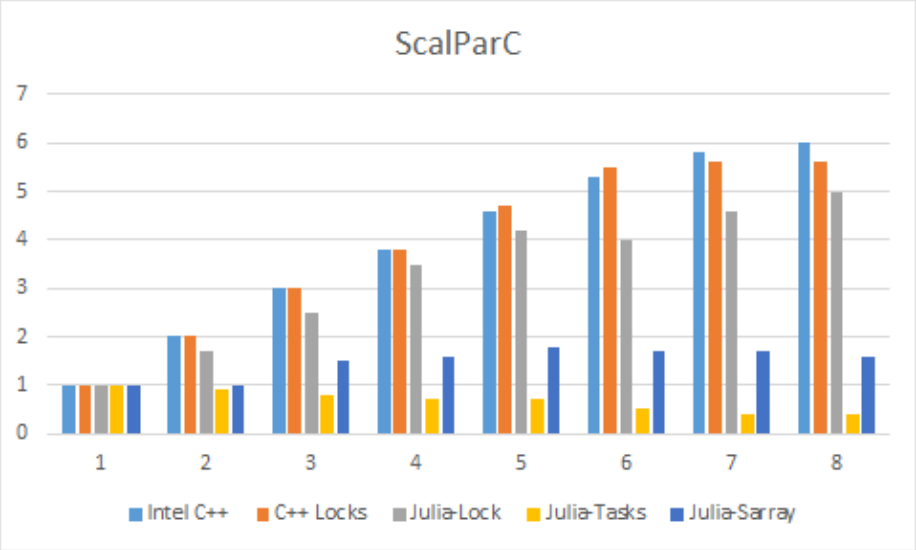

Figure 4: Speedup/Cores chart for ScalParC benchmark

passing model that only sends lazy transaction descriptions, rather than copying the entire data.

It is clear that there remains work to be done for The consistency of Julia's lock based implementations with the C++ version is remarkable, but is slightly overshadowed by the constant factors not depicted in a a dimensionless speedup chart. The dramatic performance difference between the lock-based and Transaction based algorithms is that the lock-based algorithm is hand-rolled so as not to introduce additional overhead for greater thread numbers. The low performance of **Task** implementation is expected, since it is not parallel at all, though there is certainly room to reduce the overhead for greater threads (potentially by ignoring them completely). We suspect that much of the bottleneck for Shared Array Transactions is in the Shared Array framework itself, as Julia is known for having patchy support for true shared memory - preferring an explicit distributed model instead.

5 Conclusion

This work sets the foundation for Software Transactional Memory in Julia. However, mature Software Transactional Memory implementations, such as the efforts of the Intel C++ compiler are highly tuned, and commonly long community efforts.

Julia was not benchmarked against the Haskell or Clojure Software Transactional Memory implementations, even though they are the industry standard for practical usage, because the comparison would not be sensible. The key fea-

ture of those two systems is to keep all data in *P*ersistent Data-structures [6], allowing for relatively fast data updates while preserving all previous versions in case of transaction rollback. This is possible for pointer-linked data-structures by only copying and updating the critical path to a modified leaf value, causing the rest of the structure to point back to the unmodified original. Mirroring this approach for Julia to allow a fair benchmark would require a total rewrite of the algorithm, and would do more to compare the languages' capacity for persistent data-structures, since functional languages have a Garbage Collector highly tuned for such purposes. While the algorithm conversion may be automated (see Future work on @shared macros), native performance will not be matched without heavy modifications to the Julia Garbage Collector.

Because of the top-level macro-based approach taken here, the Julia implementations do not currently support *nested* transactions, where an arbitrary transaction may occur (possibly dynamically) while executing another transaction, as the second transaction will not be properly converted by the macro. This is a limitation of the existing front-end, not a fundamental limitation to the implementation. The solution would be, rather than relying only on Julia's macro facility, to fully internalize the Software Transactional Memory specification as a data-type in the **Task** hierarchy, and utilize Julia's multiple dispatch facilities for automatic conversion to Transactions, similarly to the "Monad" implementation favored by Haskell. This would provide a bind/join semantics, allowing a transaction returning a transaction to be "flattened" into a single transaction. [9]

6 Analysis and Future Work

In this section are laid out a number of future directions to expanding this work into a robust Software Transactional Memory ecosystem for Julia, rivaling those mature implementations.

6.1 Reduce overhead with "Chunked" shared arrays

Currently, there are many choices for how to enforce transaction boundaries (one for every partition of an $n \times m$ matrix) in the shared array implementation. While all are correct, each has a performance impact: memory overhead is linearly proportional to the number of chunks, but retry rates from block collisions threaten a potentially super-linear slowdown, and even live-locks, for poorly chosen transaction boundaries. The safest choice then for programmers utilizing this implementation is to use the indiscreet chunking, allowing for minimal overhead, and manual chunking by splitting the matrix if necessary. However, the optimal choice of chunking is completely determined by memory access patterns within the matrix, so if these details are known statically at compile time (even approximately known), one might expect an automatic optimization choice made programmatically. These sorts of optimizations, those which make critical performance choices for the programmer, are rightfully re-

garded with caution when peak performance is necessary. In this case, skeptics need not worry, for the choice is no choice at all: for a given memory access pattern, there is a uniquely optimal chunking choice. Often, this access pattern is statically known, implicit in the structure of the program. However, if the access pattern is dictated by external data received by the program during execution, no such analysis is possible. In this case, the human programmer has the advantage of more intimate knowledge of the kind of input that will be received. Again, this is not a fundamental limitation, as programmers must learn the input pattern from experience, so to may an optimization engine learn to pick the right parameters through automated benchmarking.

6.2 @distributed and @shared transforms for data-types

While shared and distributed arrays are powerful tools, they work best on regularly arranged data, and do not support, for example, nested data parallelism [3]. This becomes particularly important for Software Transactional Memory, where optimal performance comes from depth-first partitioning along transaction boundaries, rather than the traditional breadth-first partitioning of data-parallel applications. Fortunately, there is nothing particular about Julia's shared and distributed arrays that hinges on the array structure - the shape simply happens to be baked into the implementation. The major hindrance then, is that implementing new shared or distributed versions for each data-type is time consuming, and pollutes the library environment. Shared and distributed *arrays* are chosen because they are the most general purpose data-structure, a good value-to-effort payoff. Fortunately, the machinery behind shared and distributed arrays is a direct transform of the original structure, and so can be generalized into a dedicated @shared and @distributed macro infrastructure. Usage takes the form

```
@distributed type SharedActionTable
    RowID::Array{Int,1}
    User::Array{User,1}
    TimeStamp::Array{DateTime,1}
    Action::Array{String,1}
end
```

Creating a distributed datastructure with heterogeneous fields. The **ClusterManager** used at runtime then controls how to effectively distribute the data.

References

[1] Hans Boehm Calin Cascaval Steve Clamage Robert Geva Justin Gottschlich Richard Henderson Victor Luchangco Virendra Marathe Maged Michael Mark Moir Ravi Narayanaswamy Clark Nelson Yang Ni Daniel Nussbaum Torvald Riegel Tatiana Shpeisman Raul Silvera Xinmin Tian Douglas Walls Adam Welc Michael Wong Peng Wu Ali-Reza Adl-Tabatabai, Kit Barton. Draft specification of transactional language constructs for c++. Technical report.

[2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *CoRR*, abs/1411.1607, 2014.

[3] Guy E. Blelloch. *Vector Models for Data-parallel Computing*. MIT Press, Cambridge, MA, USA, 1990.

[4] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Commun. ACM*, 51(11):40–46, November 2008.

[5] Anthony Discolo, Tim Harris, Simon Marlow, Simon Peyton Jones, and Satnam Singh. Lock free data structures using stms in Haskell. In *FLOPS 2006: Eighth International Symposium on Functional and Logic Programming*, April 2006.

[6] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, February 1989.

[7] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[8] Gokcen Kestor, Vasileios Karakostas, Osman S. Unsal, Adrian Cristal, Ibrahim Hur, and Mateo Valero. Rms-tm: a comprehensive benchmark suite for transactional memory systems. *SIGSOFT Softw. Eng. Notes*, 36(5):335–346, September 2011.

[9] Simon Marlow. Parallel and concurrent programming in Haskell. In V. Zsók, Z. Horváth, and R. Plasmeijer, editors, *CEFP 2011*, volume 7241 of *LNCS*, pages 339–401. 2012.

[10] ChÃŋ Cao Minh, Jaewoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing.

[11] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and implementation of transactional constructs for c/c++. *SIGPLAN Not.*, 43(10):195–212, October 2008.

[12] Victor Pankratius and Ali-Reza Adl-Tabatabai. Software engineering with transactional memory versus locks in practice. *Theor. Comp. Sys.*, 55(3):555–590, October 2014.

[13] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 187–197, New York, NY, USA, 2006. ACM.

[14] Andreas Richard Voellmy, Junchang Wang, Paul Hudak, and Kazuhiko Yamamoto. Mio: A high-performance multicore io manager for ghc. *SIGPLAN Not.*, 48(12):129–140, September 2013.

[15] R. Mark Volkmann. Software transactional memory. 2009.