

# Parallel Genome Assembly with Software Transactional Memory

Ketil Malde

Institute of Marine Research

**Abstract.** Parallel programs are key to exploiting the performance of modern computers, but traditional facilities for synchronizing threads of execution are notoriously difficult to use correctly, especially for problems with a non-trivial inherent structure.

Software transactional memory is a different approach to managing the complexity of interacting threads. By eliminating locking, many of the complexities of concurrency is eliminated, and the resulting programs are composable, and can easily be refactored.

Here, we investigate STM in the context of genome assembly, and demonstrate that a program using STM is able to successfully parallelize the genome scaffolding process with a near linear speedup.

## 1 Introduction

As multi-core processors are becoming commonplace, parallel programs are crucial for performance sensitive computation. Many problems can easily be partitioned into subproblems that can be solved independently (so-called “embarrassingly parallel” problems) but other problems are inherently more complicated, and are best solved by multiple interacting threads. In this case, care must be taken to keep separate threads of execution from interacting in ways that cause the program to behave incorrectly.

Traditionally, the shared data in parallel programs is protected by synchronization primitives (locks) that prevent simultaneous access to data structures. However, it is still quite difficult to write correct programs using these primitives, and incorrect or careless usage cause well-known problems like deadlocks and race conditions [6]. In addition, independent program parts that use locking primitives are in general not composable, and for instance, refactoring a previously correct program can introduce synchronization problems [5].

Software transactional memory [12], or STM, represents a different approach. Here, state that is shared between threads is accessed in *transactions*, and the state is stored in *transactional variables*. If multiple threads run transactions that attempt to modify the same state, one transaction succeeds, and the others are rolled back, and will be rescheduled by the run-time system.

Since there is no explicit locking, deadlocks are eliminated, and transactions are either committed completely or not at all, so intermediate (and possibly inconsistent state) is never exposed. In addition, STM transactions are composable

[5]. The disadvantage is a potentially higher overhead, both because transactions need to log access to transactional variables, and because transactions sometimes need to be restarted from scratch, which duplicates work.

Here, we investigate how STM can be applied to the problem of *genome scaffolding*, the process where the components of a partially assembled genome sequence are ordered and oriented to provide a more coherent (but often discontinuous) whole. A scaffolder program is implemented in Haskell using STM, and achieves a near linear speedup with the number of processors.

## 1.1 Software Transactional Memory in Haskell

There exists implementations of software transactional memory for many programming languages (e.g., [4, 10]). Some of the problems faced by implementers is that the encapsulation of transactions is not *enforced*, and exceptions, I/O operations and global, mutable state can break the transaction abstraction. Harris *et al.* [5] discusses this in more detail.

One distinguishing feature that sets Haskell apart from the majority of programming languages, is that it is *pure*: the result of a function depends only on its parameters. The evaluating of a function may not depend on state, I/O, or have other effects.

Many effectful computations can be simulated in pure code (e.g. state can be passed between functions as an explicit parameter), but this can quickly become a syntactical burden. Haskell uses a structure called a *monad* that allows the programmer to create an environment where specific effects are available. This can also include non-pure effects, and, unsurprisingly, I/O operations are only available in the appropriate monad.<sup>1</sup>

The type system distinguishes effectful computations from pure computations, and enforces that pure computations never can access impure operations. I/O operations are guaranteed to only be executed in the context of the IO monad, for instance. A monad is a *parametric type*, so for some type `a`, the type `IO a` designates an I/O action which when executed produces a value of type `a`. For instance, `getChar` has type `IO Char`, as it is an I/O action that can produce a character. Apart from the ability to be executed by the run-time system, such a value is not special, and values in the IO monad can be assigned to variables, and manipulated with functions. Using combining functions, larger programs can be built that interact with their environment in complex ways.

In Haskell, STM is implemented as a monad, and transactions are confined to this environment. Similar to the IO example, a type `STM a` designates a transaction that, when executed return a value of some type `a`. In the STM monad, mutable data structures are available as explicitly declared transactional variables, or TVars. Using the same mechanism as other monads, simple transactions can be composed into more complex ones. Transactions can be executed in the

---

<sup>1</sup> While most monads can be – and usually are – implemented as simple libraries, the IO monad is special, and executed by the run-time system.

IO monad, using the `atomically` function, which converts a value of type `STM a` to a value of type `IO a`.

It is important to note that TVars are *only* accessible from the STM monad. This makes them unavailable to non-transactional computations, and the static type system rigidly enforces this encapsulation. Similarly, transactions have no means to modify *other* state, in particular, they are prevented from performing I/O operations or modifying global variables. This avoids the problems that have been critical for earlier STM implementations.

## 1.2 Genome assembly and scaffolding

The sequencing process usually produces a large set of short fragments (or *reads*) from random positions in the genome. Given such a set of reads, the genome assembly problem is to reconstruct the originating genome sequence. The traditional approach is the method called *overlap-layout-consensus* [3, 9], or OLC:

1. Identify *overlaps* by aligning each sequences against all others
2. Determine the *layout* – order and orientation – of the reads that is best supported by the alignments
3. Merge sequences according to layout to produce a single contiguous *consensus* sequence

The first step is trivially parallelizable (each read is independent of the others, and can be independently aligned), but the second step is more complicated. Usually, the problem is modeled as a graph where each read is a node, and there exists an edge between nodes if the corresponding reads are determined to overlap. The assembly is then equivalent to identifying a Hamiltonian path in the graph, which is an NP-complete problem.<sup>2</sup>

The layout phase processes the overlap graph to produce a linear progression of the reads, and although distant parts of the graph can be processed independently, care must here be taken if two operations attempt to modify the same nodes simultaneously. The implementation details of assemblers are not often published, but observation of some common OLC assemblers indicates that they commonly perform alignments in parallel, but later run the layout phase using a single thread of execution.<sup>3</sup> This supports the view that constructing a correct locking scheme for doing graph updates in parallel is difficult, and it would probably be inefficient, since it would incur locking overhead also for the non-colliding updates - likely to be the vast majority of them.

Genome scaffolding is closely related to assembly. Here, the assumption is that a genome has been sequenced and assembled into a set of contigs. Instead

---

<sup>2</sup> A popular alternative to OLC is the *de Bruijn* assembly [11]. This is less resource-intensive, as it avoids the all-against alignment phase, and it is equivalent to identifying an Eulerian path. But it is also easier to parallelize in practice, which may also be a factor that contributes to its popularity.

<sup>3</sup> E.g. Newbler only parallelizes computing alignments and generating output. [1]

of overlaps, there exists information about the orientation and order of the contigs. This is typically a set of paired reads, where the members of any pair is separated by some known distance. As for assembly, it is not straightforward to implement a parallel scaffolding algorithm correctly using locking, and commonly used programs like SSPACE [2] are single-threaded.

Scaffolding simplifies the process in two ways: first, it reduces the amount of data that needs to be considered (E.g. for the sea louse assembly, the initial assembly involves up to one billion reads). Second, mapping reads to contigs make it practical to use standard alignment tools and file formats. For these reasons, the rest of this paper will focus on scaffolding.

## 2 Algorithm and Implementation

The implemented scaffolder constructs scaffolds from aligned, paired reads, read from a standard BAM[13] file. The algorithm consists of four steps:

1. calculate alignment statistics
2. extract relevant alignments
3. build scaffolds
4. output the result

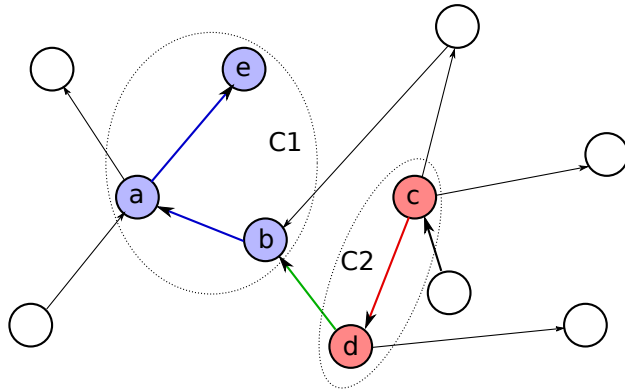
First, some statistics on the input alignments are calculated. By examining read pairs that map to the same contig, we obtain approximations for the distance between paired reads (called the *insert length*), and its variance. Also, the number of contigs is extracted from the BAM file.

The second step iterates over the contigs, and extracts all alignments relevant for scaffolding. In other words, the read must map near the end of the contig, it must be oriented correctly, and its mate must be mapped to a different contig. These alignments are stored in an associative data structure.

The third step uses two arrays, one which maps each contig to its scaffold, and one which for each scaffold stores the scaffold layout, i.e., the set of ordered and oriented contigs. Initially, each contig is in its own singleton scaffold.

Although a practical scaffolding program is likely to involve heuristics to resolve ambiguous cases, we will here limit ourselves to link together any pair of contigs that has a mutual best match. The program iterates over all contigs, at each end identifying the contig that has most read pairs pointing to it. If the relationship is reciprocal (i.e. the identified contig similarly has most of its outgoing links pointing back), the scaffolds are merged. For instance, in the example graph in 2, the heaviest outgoing edge from **b** is the one going to **d**, and similarly, it is the heaviest incoming edge to **d**. This causes these two contigs to be identified as adjacent, and cause their scaffolds to be merged.

Merging two scaffolds involves updating the one scaffold's entry in the scaffold array to contain the new scaffold, and deleting the other scaffold's entry (see Figure 2). Then, the elements in the contig array corresponding to contigs in the scaffold that was deleted are updated to point to the new scaffold.



**Fig. 1.** An example overlap graph. Two scaffolds are identified, C1 containing nodes a, b, and e, and C2 containing nodes c and d. Adding the edge from d to b will merge these into a single scaffold.

To parallelize, we simply split the iteration over the contig array so that each thread iterates an equally sized segment of the array. Note that even if threads work on separate segments, they will affect contigs outside their segment, as they are joined in scaffolds.

Statistically, merging operations will usually be independent if the arrays are large compared to the number of concurrent operations. This also depends on the criteria for merging being local. For instance, merging could examine several candidates, and it could consider more reciprocal links, in effect making the decision depend on a larger subset of the graph. This would increase the chance of colliding operations. In any case, collisions will occur occasionally, and a parallel implementation must take them into account.

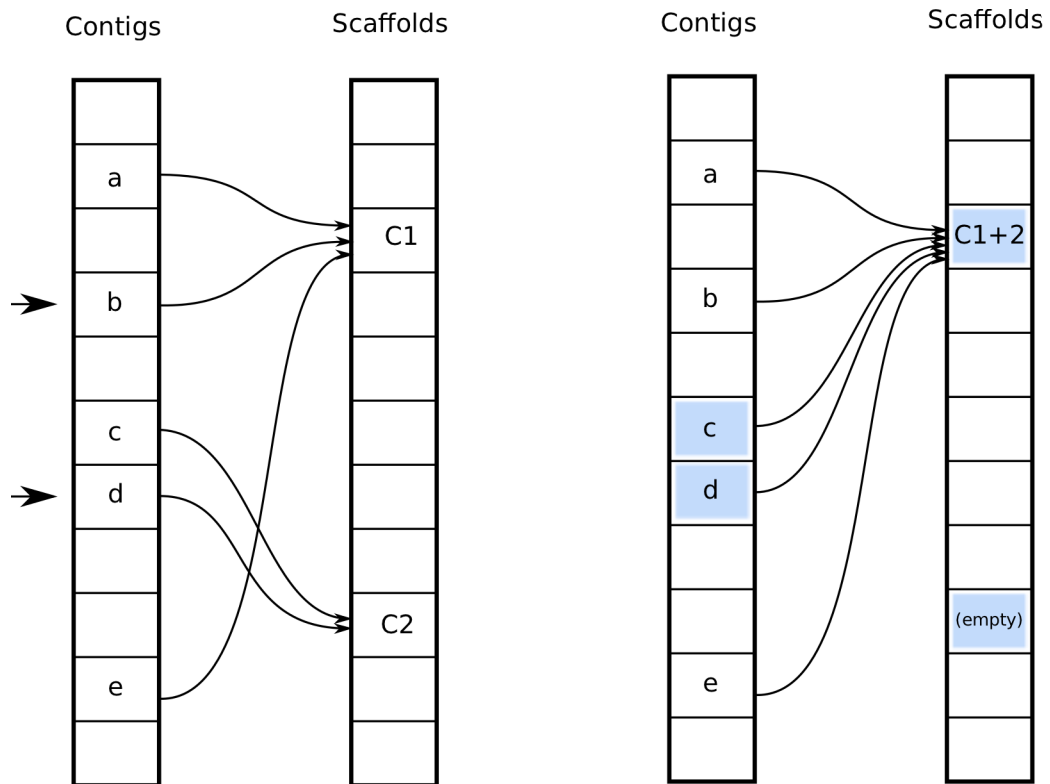
STM here makes this process easy, and in fact, the code implementing this algorithm using mutable arrays in the IO monad and using transactions in the STM monad is *exactly the same!* Only the top-level function is different, as the STM version must spawn multiple threads that process an array segment each.

### 3 Results

In order to test the implementation, a set of contigs resulting from the assembly of sea louse (*Lepeophtheirus salmonis*) sequences were used. The assembly was constructed using the Newbler program (Roche), which assembled approx. 50 million 454 reads [8] into approximately 191 000 contigs.

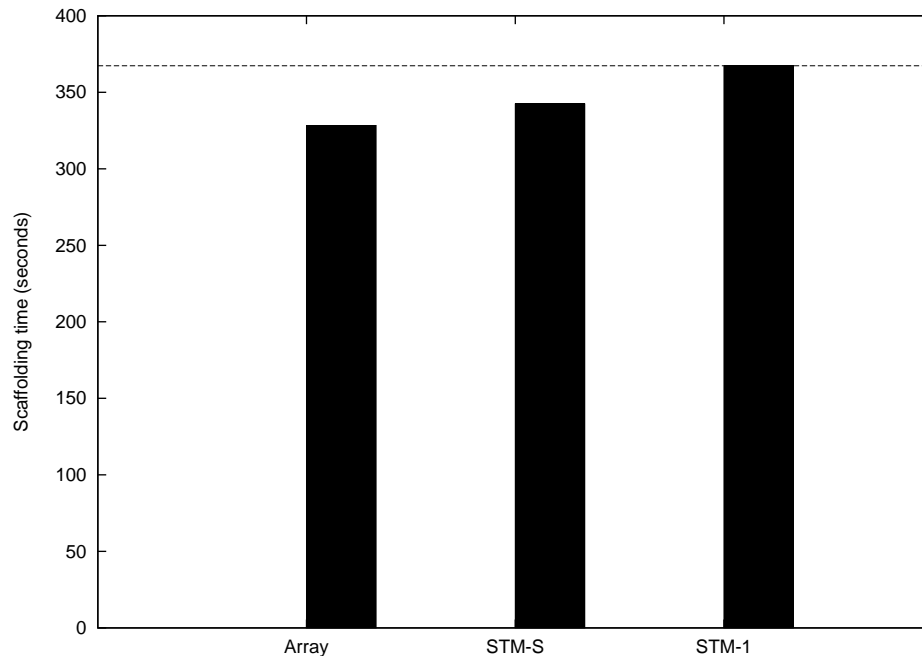
As our pairing data, we use a set of 72 200 652 Illumina reads, where each pair consists of two 100bp reads, spaced about 150bp apart. The reads were aligned using BWA [7], resulting in 68 569 814 alignments (95% of the reads), and with 10 187 580 alignments with the mate mapped to a different contig.

The program was compiled with GHC 7.0.2, using the `-O2` option. It was executed on a computer with eight Intel Xeon E7340 processors, using options `+RTS`



**Fig. 2.** A schematic presentation of the arrays used in the scaffolding algorithm. Contigs a, b, and e are initially (left) in scaffold C1, and contigs c and d are in scaffold C2. When the algorithm decides that contigs b and d (indicated by arrows) should be adjacent, the scaffolds are merged, causing several cells to be updated (shaded, right).

-A100M. The parallel STM version was additionally compiled with `-threaded`, and run with `-qg`.

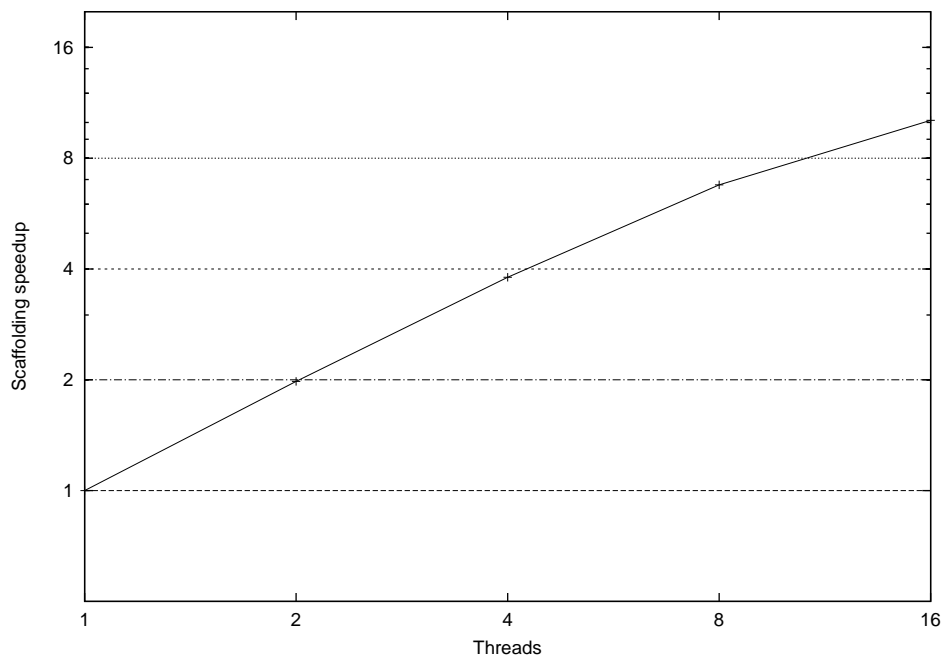


**Fig. 3.** Speed spent in the scaffolding stage. “Array” is the implementation using mutable arrays, “STM-S” is the STM implementation running on the single-threaded runtime, and “STM-1” is the STM implementation using a single thread with the threaded run-time.

Figure 3 shows the running time for the scaffolding stage. We see that there is some overhead associated, both with using arrays of transactional variables (`TArray`) over regular mutable arrays (`IOArray`), and with running on the multi-threaded GHC run-time over the single-threaded one.

The STM implementation scales well. From Figure 3, we see that as we increase the number of parallel threads, the speedup is close to the optimum, up to eight threads, matching the number of CPUs. The CPUs use hyperthreading, and each processor core appear to the OS as two processing units. Thus, the STM implementation is still achieving a substantial speedup with 16 threads, even though it means running two threads per physical core.

The resulting scaffolds were checked against scaffolds produced by `SSPACE`, and were found to differ slightly, but for the most part, they identified the same layout of contigs.



**Fig. 4.** Speedup of the STM implementation with increasing number of threads.

## 4 Conclusions and discussion

Software transactional memory is most attractive when the program can be structured as set of mostly-independent operations, and where each operation only involves a small set of variables. If the operations are completely independent, the problem most likely can be trivially partitioned, and if the number of variables involved in each operation is large, performance will deteriorate as the transaction log increases in size.

The overlap-layout-consensus approach to the sequence assembly problem fits well with these criteria, and is well suited to an STM approach. In the implementation presented, we observe a small overhead for using software transactional memory compared to regular arrays, and an additional overhead for using a multi-threaded implementation compared to a single threaded one, but the STM implementation scales well with the number of threads, and with only two threads, it is substantially faster.

In this particular benchmark, extracting the alignment information consumed almost twice the amount of time as the scaffolding. We also do not include the time to run BWA to compute alignments, this is by far the most time consuming operation, typically taking many hours. Since the non-scaffolding operations are so costly, one might argue that parallelizing the scaffolding process is less important, especially as the alignment and file parsing tend to be easy to run in parallel using conventional methods. This does not mean that the scaffold building is unimportant, however, and as the number of processors in a system continue to grow, Amdahl's law will at some point make any non-parallel stage a bottleneck. The current program has not been heavily optimized. It is likely that optimization will result in faster operations overall, diminishing the net gains from a parallel implementation relative to the overhead incurred by STM. But conversely, the analysis used here is naïve, only taking into account the single best-supported candidate for each contig. A more advanced analysis is likely to require more computation, thus increasing the gain. In general, the quality of the result is more important than the CPU time spent, and by making the analysis parallel, it becomes more practical to dedicate more computing resources to this stage, opening up for qualitative improvements as well as quantitative ones.

It is of course possible to implement a parallel scaffolder program using traditional locking schemes, and although this level of detail is rarely discussed in the literature, it would be surprising if no assembler or scaffolding software already implemented it. The main advantage of STM over traditional locking schemes is that STM removes much of the burden of correctness from the programmer, and here, the fact that the exact same code works both for regular, mutable arrays and for transactional arrays is remarkable. Together with the composability that STM offers, this makes it vastly easier for the programmer to refactor and refine the algorithm and implementation.

## References

1. 454 Life Sciences Corp., Branford, CT 06405: 454 Sequencing System Software Manual, v 2.5p1, part C (August 2010)
2. Boetzer, M., Henkel, C.V., Jansen, H.J., Butler, D., Pirovano, W.: Scaffolding pre-assembled contigs using SSPACE. *Bioinformatics* 27, 578–579 (2011), <http://bioinformatics.oxfordjournals.org/content/early/2010/12/12/bioinformatics.btq683.short>
3. Bonfield, J.K., Smith, K.F., Staden, R.: A new DNA sequence assembly program. *Nucleic Acids Research* 23, 4992–4999 (1995), <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC307504/pdf/nar00024-0066.pdf>
4. Brevnov, E., Dolgov, Y., Kuznetsov, B., Yershov, D., Shakin, V., Chen, D.Y., Menon, V., Srinivas, S.: Practical experiences with java software transactional memory. In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. pp. 287–288. PPOPP '08, ACM, New York, NY, USA (2008), <http://doi.acm.org/10.1145/1345206.1345259>
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*. pp. 48–60. PPOPP '05, ACM, New York, NY, USA (2005), <http://doi.acm.org/10.1145/1065944.1065952>
6. Lee, E.A.: The problem with threads. Tech. Rep. UCB/EECS-2006-1, EECS Department, University of California, Berkeley (Jan 2006), <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html>, the published version of this paper is in *IEEE Computer* 39(5):33-42, May 2006.
7. Li, H., Durbin, R.: Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics* 25, 1754–1760 (2009), <http://bioinformatics.oxfordjournals.org/content/25/14/1754.full>
8. Margulies, M., Egholm, M., Altman, W.E., Attiya, S., Bader, J.S., et al.: Genome sequencing in microfabricated high-density picolitre reactors. *Nature* 437, 376–380 (2005), <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1464427/>
9. Myers, E.W., Sutton, G.G., Delcher, A.L., Dew, I.M., Fasulo, D.P., Flanigan, M.J., et al.: A whole-genome assembly of drosophila. *Science* 287(5461), 2196–2204 (2000), <http://www.sciencemag.org/content/287/5461/2196.abstract>
10. Ni, Y., Welc, A., Adl-Tabatabai, A.R., Bach, M., Berkowits, S., Cownie, J., Geva, R., Kozhukow, S., Narayanaswamy, R., Olivier, J., Preis, S., Saha, B., Tal, A., Tian, X.: Design and implementation of transactional constructs for c/c++. In: *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. pp. 195–212. OOPSLA '08, ACM, New York, NY, USA (2008), <http://doi.acm.org/10.1145/1449764.1449780>
11. Pevzner, P.A., Tang, H., Waterman, M.S.: An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences* 98(17), 9748–9753 (2001), <http://www.pnas.org/content/98/17/9748.abstract>
12. Shavit, N., Touitou, D.: Software transactional memory. In: *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. pp. 204–213. PODC '95, ACM, New York, NY, USA (1995), <http://doi.acm.org/10.1145/224964.224987>
13. The SAM Format Specification Working Group: The SAM Format Specification <http://samtools.sourceforge.net/SAM1.pdf>