Rust and C++ performance on the Algorithmic Lovasz Local Lemma

ANTHONY PEREZ, Stanford University, USA

Additional Key Words and Phrases: Programming Languages

ACM Reference Format:
Anthony Perez. 2017. Rust and C++ performance on the Algorithmic Lovasz Local Lemma. 1, 1 (December 2017), 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 SUMMARY

The performance and ease of use of a programming language are two major considerations for developers when choosing the appropriate language for a project. This work compares C++ and Rust through the lens of performance and usability in both serial and concurrent contexts, and uses the algorithmic Lovasz Local Lemma as a test-bed. Performance is evaluated quantitatively through run time statistics collected over different instances of K-SAT while usability is evaluated qualitatively. It was found that although both languages make trade-offs with respect to usability, Rust outperforms C++ in both serial and parallel contexts. From this the author concludes that, regardless of the performance of the optimal implementation in each language, non-optimal but more practical implementations of similar development effort seem to result in higher performance code in Rust, likely due to Rust's ownership and safety features.

2 BACKGROUND

Performance and ease of use are two key design considerations when designing any programming language. As previously mentioned, this work attempts to compare the Rust and C++ programming languages along these two axes. Any performance comparison between two languages requires a testing frame work to empirically verify its claims. In this work, we choose to implement the algorithmic Lovasz Local Lemma (LLL) to solve constrained instances of the K-SAT problem. The algorithmic LLL lends itself to performance comparisons as the algorithm is simple but non-trivial and has both serial and parallel implementations that are provably correct. In order to provide a context for the performance statistics, definitions and details of the problem instances (K-SAT) and the solving algorithm (algorithmic LLL) are provided below.

2.1 K-SAT

The Boolean Satisfiability problem, or the SAT problem, has applications in EDA, automatic test generation, logic synthesis, and AI [2]. K-SAT is a constrained instance of the Boolean Satisfiability

Author's address: Anthony Perez, Stanford University, Stanford, CA, 23185, USA, aperez8@stanford.edu.

XXXX-XXXX/2017/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

problem where the boolean formula is constrained to be the conjunction of disjunctions over K variables. The following is an example of a satisfiable 3-SAT instance and its satisfying assignment.

$$\phi = (x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z)$$
$$\text{assignment} = \{x : TRUE, y : FALSE, z : TRUE\}$$

In this work we solve constrained instances of the K-SAT problem in which each variable appears in at most $\frac{2^k}{ke}$ clauses (disjunctions). This is done so that the Lovasz Local Lemma can be applied.

2.2 Lovasz Local Lemma

The algorithmic Lovasz Local Lemma (algorithmic LLL) [1] solves the following problem. Suppose there is a set of bad events $\mathbb{A} = \{A_1, ..., A_n\}$ that you wish to avoid. Let each event depend on a set of variables and denote this set as $vble(A_i) = \{v_{i,1}, ..., v_{i,c_i}\}$. This means that A_i is a function of only $vble(A_i)$. Let V denote the set of all these variables: $V = \cup_i vble(A_i)$. Let $d = \max_{A_i} |\{A_j | vble(A_i) \cap vble(A_j) \neq \emptyset\}|$. In other words, d is the maximum number of other events that a single event depends on. Choose a distribution $\mathbb{D}$ over V, then if for all A_i, $Pr_{\mathbb{D}}[A_i] \leq \frac{1}{e(d+1)}$ Algorithm 1 will find an assignment to V such that all bad events are avoided in an expected linear number of resampling steps.

ALGORITHM 1: Algorithmic LLL

```
Function Lovasz_Local_Lemma
    V ← sample from 𝔻 ;
    while ∃A ∈ 𝔸 s.t. A is in a bad state do
        vble(A) ← newly sampled values from 𝔻 ;
    end
    Return V ;
end
```

We can map the abstractions in the algorithm description to K-SAT instances in the following manner:

(1) Let A_i be the event that the ith clause is not satisfied.
(2) Let $vble(A_i)$ be the variables in the ith clause.
(3) Let $\mathbb{D}$ be the Bernoulli(0.5) distribution (variables are true with probability 0.5).

We see that if variables are constrained to be in at most $\frac{2^k}{ke} - 1$ clauses then $Pr[A_i] = \frac{1}{2^k} \leq \frac{1}{e(d+1)} = \frac{k}{2^k}$. Note that the algorithmic Lovasz Local Lemma presented here is a simplified version of the algorithm.

3 APPROACH

3.1 Goals

The goal of this work is to provide a relative ranking of C++ and Rust in terms of both performance and usability. The algorithm is guaranteed to find satisfying assignments if implemented correctly and given K-SAT instances that fit the constrains of the problem, therefore correctness is not evaluated. Performance is evaluated by timing the run-time of the algorithm using timers built in to each language. Usability is evaluated by a qualitative comparison of the pain points of working in each language. The full scope of this goal cannot be realized with a single experiment in a single test environment. Therefore the author considers this paper a stepping stone toward future research on the topic.

3.2 Experimental Setup

We seek to only vary the language in each trial of the experiment while keeping all other factors constant. There are three parameters that control the size of the K-SAT problem (assuming all variables are assigned randomly). These parameters are:

- the number of variables per clause (K),
- the number of clauses,
- and the number of variables.

The value of K and the number of clauses were varied to provide a comparison across several sizes of the problem. The values of the number of clauses ranges from values that result in near-instant solutions to the maximum value feasible on the available hardware. The value of K varies from the minimum possible value under the constraints of the algorithimc LLL to a value for which solutions are typically 2-3x faster than the minimum value. The number of variables is always set to the minimum possible amount, as this is the most difficult setting. Five random instances were generated for each settings of the parameters to the K-SAT problem to control randomness due to the problem instance. Additionally, each problem instance was run 5 times to control randomness due to the processor. Note that the problem instances used for bench-marking are the same across both languages and across serial and parallel versions of the algorithm in order to provide a fair comparison.

3.3 Why LLL

The algorithmic LLL is a very convenient algorithm for comparing the performance of two languages. The algorithm exists as both serial and parallel versions, which allows comparisons to be made in both contexts. Additionally, the problem the algorithm solves is interesting and difficult enough that the algorithmic LLL is not a "toy" algorithm. Finally, the algorithm itself is fairly straight forward, which makes analyzing potential performance gaps in the language easier.

3.4 Algorithm Design

3.4.1 Input/Output. The input and output (i.e. the signature) of the algorithm was the same across languages and serial vs. parallel versions. As input, the algorithm is given a K-SAT object which contains the following properties:

- "num_vars": An integer representing the number of variables.
- "k": An integer representing the number of variable in each clause.
- "num_clauses": An integer representing the number of clauses.
- "clauses": A vector containing Clause objects. Each clause object contains only a vector of Variables. Each Variable is a struct containing an integer "id" representing the identity of the variable and an integer "affinity" representing whether the literal in the clauses is negated or not.

As output, the algorithm is expected to return a vector of booleans which corresponds to a satisfying assignment for the problem instance. The ith element in the vector should correspond to the truth value of the ith variable in the satisfying assignment.

3.4.2 Implementation. Both serial and parallel implementations of the algorithm maintain a current assignment to the variables and update it in order to find the satisfying assignment. Both implementations share two procedures:

- **satisfied** which checks if a clause is satisfied. **satisfied** runs in $O(n)$.
- **resample** which samples a new assignment to a set of variables and overrides the current assignment. **resample** runs in $O(n)$.

Refer to Algorithm 2 for the implementation of the serial algorithm and Algorithm 3 at the end of the document for the implementation of the parallel algorithm.

ALGORITHM 2: Serial Algorithmic LLL

```
Function serial KSAT ksat_inst
    assignment ← a random boolean vector with size equal to the number of variables ;
    while A clause was found to be unsatisfied do
        forall Clause c in ksat_inst do
            if not satisfied(c, assignment) then
                resample(c, assignment) ;
            end
        end
    end
    Return assignment ;
end
```

3.4.3 Details. The algorithms above were implemented by the author without any existing code base. In Rust, the crossbeam package was used to implement parallelism. In C++, only std::thread was used to implement parallelism.
There is a serial algorithm that runs in expected $O(n2^k)$ time (rather than expected $O(kn^2)$ time) but this algorithm was not explored for two reasons. The first is that the purpose of this work is to compare languages not implement the most efficient algorithmic LLL. The second is that this algorithm ran slower empirically.

4 RESULTS

4.1 Performance

4.1.1 Fairness. As was previously mentioned, the problem instances are the same across all versions of the algorithm. Likewise, timing is always computed as the time from which the function is called to when the function returns. Approximately equal time was spent across both languages and the code structure is similar where appropriate.

4.1.2 Analysis. Figure 3 shows that run-time increases with the number of clauses in the K-SAT instance while Figure 2 shows that the run-time decreases as the value of K increases. These are both expected results. Although increasing K increases the size of the problem instance, the expected fraction of unsatisfied clauses is $\frac{1}{2^k}$, and thus the run-time of the algorithm is expected to decrease as K increases. Figure 1 provides run-time statistics for all choices of problem and algorithm parameters. As expected, the parallel algorithmic LLL outperforms the serial algorithm in Rust. Perhaps surprisingly, the parallel C++ algorithm was slower than the serial C++ algorithm. We also see that in both serial and parallel contexts, Rust outperforms C++. CPU sampling reveals that 60% of the time used by the parallel C++ algorithm was spent running and joining threads (.6 is the ratio of samples in std::thread and thread.join to samples in the function as a whole). This means that the code pertaining to threading ran at least 3 times slower in C++ than in Rust in the experiments in this work.

Its surprising that Rust outperforms C++ given that, within the realm of fast systems languages, Rust optimizes for safety while C++ optimizes for speed. The performance difference could be attributed to the following:

(1) Rust's ability to enforce safety leads to code that compilers can better optimize.

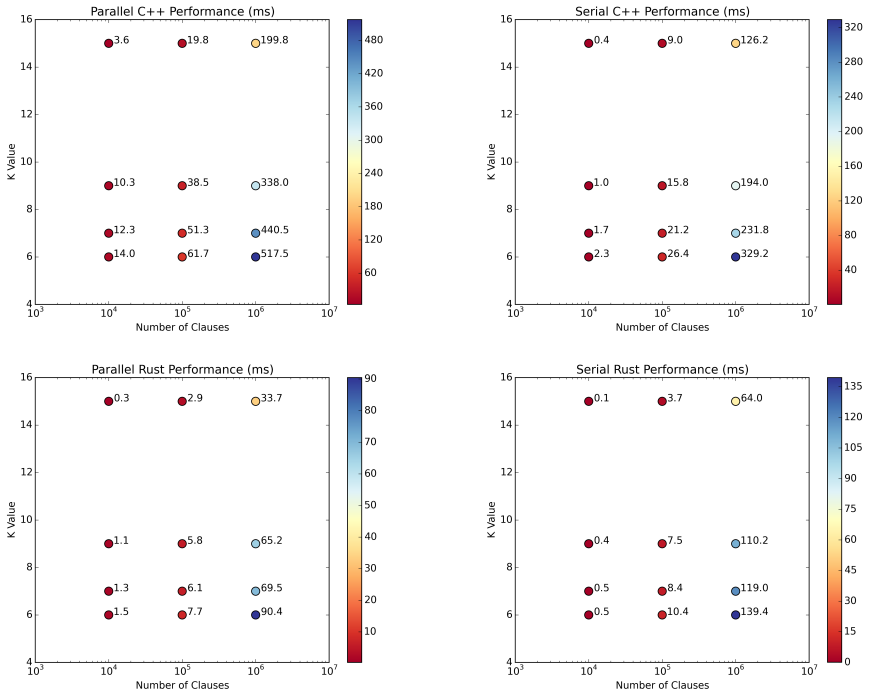

Fig. 1. Scatter plot illustrating performance as K-SAT parameters vary. Numbers adjacent to points indicate the run-time in milliseconds averaged over 5 problem instances (different assignment of variables to clauses) with 5 trials each.

(2) Rust's safety features lead developers to create code with better memory sharing between threads.

These factors will be discussed in the next section, which describes the usability differences between the two languages.

4.2 Language Design / Usability

One factor that needs to be considered is that neither the Rust nor C++ implementations of these algorithms were optimal. It may be the case that the optimal C++ implementation outperforms the optimal Rust implementation or vice-versa. However, the reality is that optimal performance is difficult to achieve in practice and this leads to an argument for why Rust's language design may have lead to higher performing code.

4.2.1 Compiler. Most of the debugging time spent in Rust involved dealing with the compiler. The compiler is fairly descriptive in informing the developer of ownership rule violations or other issues specific to Rust. For errors that might occur in all languages, the author found that Rust compiler error messages were more helpful than C++ compiler error messages, but this is offset by C++'s IDE support. The trade off of dealing with compiler warnings was that there were almost no logic errors, and no (non-compiler) errors due to misunderstanding the argument types of functions, ownership of variables, or passing by value versus reference.

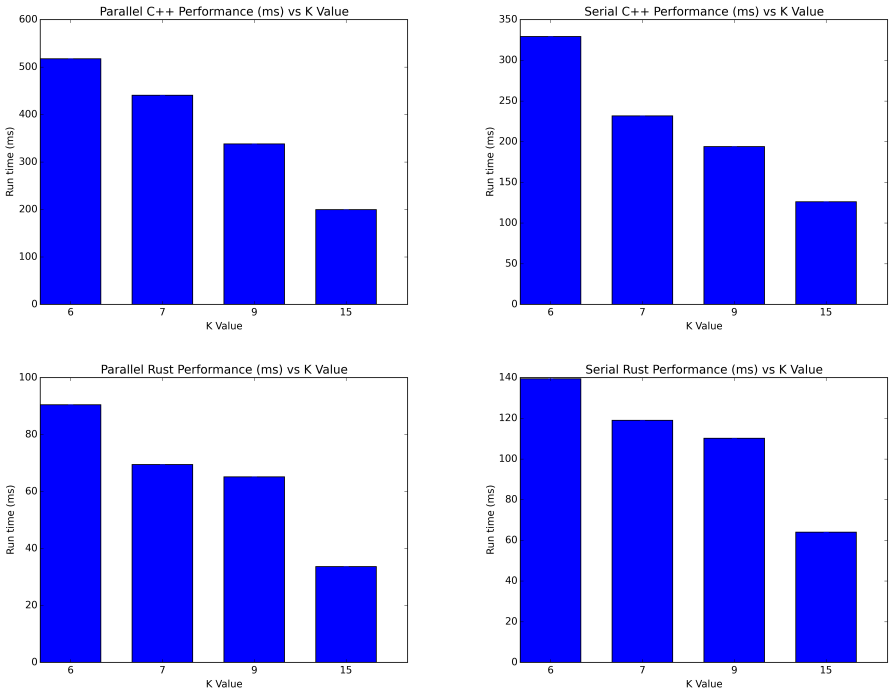

Fig. 2. The number of clauses was fixed to 1,000,000 for all plots in this figure.

4.2.2 Code Organization. Dealing with errors of importing function across files was much easier in Rust than in C++. Dealing with header files in C++ can be tedious, specifically regarding issues due to importing the same header twice, dealing with naming collisions between functions, and interpreting non-intuitive compiler errors. Importing modules was simpler in Rust, and compiler error messages were easily interpretable.

4.2.3 IDE. Visual Studio was used to develop and profile the C++ code, while vim was used to develop the Rust code. The author is not aware of a standard powerful IDE for Rust, while Visual Studio seems to be fairly widespread. Visual studio far outclasses vim. Visual studio caught almost all would-be compiler errors before compilation, especially syntax errors. Thanks to Visual Studio, the vast majority of bugs were due issues with the code file structure and issues with logic rather than syntax and language issues. The Rust development cycle was slowed down by having to write out larger sections of code, then run the compiler, and then fix the compiler-found issues in the code.

4.2.4 Threading. C++'s std::thread API was much simpler to use than Rust's crossbeam API, but std::thread had unexpected behavior. std::thread's constructor will copy all arguments by value, including reference arguments, and during implementation this lead to a silent bug (that was eventually fixed) in which a large object was being copied by value. The bug did not result in any warnings, but slowed performance significantly, and thus took a significant amount of time to debug. In Rust, code that compiled generally worked, but the crossbeam package and Rust's ownership rules made the process of getting the code to compile take much longer than in C++. The assignment vector had to be wrapped in an Arc pointer in Rust due to its ownership rules,

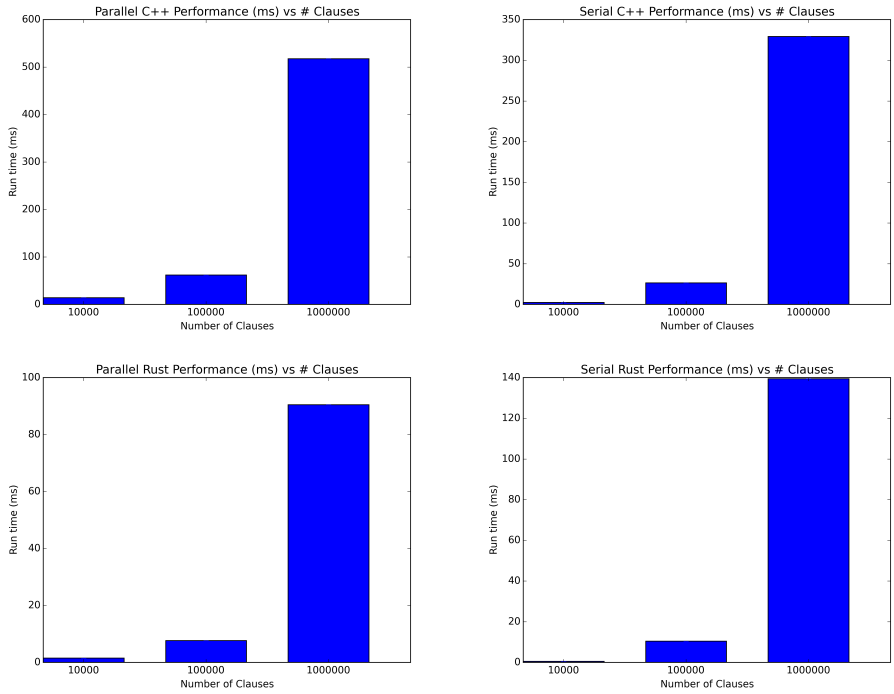

Fig. 3. The value of K was set to 6 for all plots in this figure.

although this was inefficient in this instance. Likewise, dealing with closure and their scoping rules was more time-consuming than std::thread's function pointer based constructor.

ALGORITHM 3: Parallel Algorithmic LLL. Note that the **get_independent** computation is not required to wait for all threads of computation in the **get_unsatisfied** computation to finish. In the actual implementation, these two functions are interwoven so that the unsatisfied clauses are processed as soon as a thread makes them available.

```
Function get_unsatisfied KSAT ksat_inst, Vec<bool> assignment
    chunks ← split the clauses in ksat_inst into chunks ;
    forall chunk in chunks, in parallel do
        unsatisfied_clauses ← an empty vector ;
        forall Clause c in chunk do
            if not satisfied(c, assignment) then
                Add c to unsatisfied_clauses ;
            end
        end
    end
    Return the concatenation of all unsatisfied_clauses vectors ;
end
Function get_independent Vec<Clause> unsatisfied_clauses
    independent_set ← an empty vector ;
    forall Clause c in unsatisfied_clauses do
        if c does not contain any variable already in independent_set then
            Add c's variables to independent_set ;
        end
    end
    Return independent_set ;
end
Function parallel KSAT ksat_inst
    assignment ← a random boolean vector with size equal to the number of variables ;
    while loop forever do
        unsatisfied_clauses ← get_unsatisfied(ksat_inst, assignment) ;
        if unsatisfied_clauses is empty then
            Return assignment ;
        end
        independent_set ← get_independent(unsatisfied_clauses) ;
        resample(independent_set, assignment) ;
    end
end
```

5 REFERENCES

REFERENCES

[1] Robin A. Moser and Gábor Tardos. 2009. A constructive proof of the general Lovasz Local Lemma. *CoRR* abs/0903.0544 (2009). arXiv:0903.0544 http://arxiv.org/abs/0903.0544

[2] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Annual Design Automation Conference (DAC '01)*. ACM, New York, NY, USA, 530–535. https://doi.org/10.1145/378239.379017