Designing an Efficient Parallel Presburger Arithmetic Solver in Haskell

Richard Li (YL5573)

December 2025

Introduction

This project refactors an existing Presburger Arithmetic solver library, Data.Integer.SAT, and provides a parallel alternative to the existing single-thread solution function. Testing on a 17-variable-weighted-sum statement yields ~3x sequential speedup, and on top of which, ~4.5x parallel speedup given effective parallel configurations.

Background & Context

Presburger Arithmetic (PrA) is a weak, completely axiomatisable logic theory first introduced in 1929. Despite including a weak axiom schema of induction, PrA is nevertheless a decidable theory (unlike more typical arithmetics like Peano Arithmetic or ZF(C), both of which are only semi-decidable). Specifically, one can design an algorithm such that, given any sentence, i.e. formula candidate, of finite length, the algorithm outputs correctly whether the sentence is provable from the axioms of PrA.

Typically, theories with induction encounter decidability challenges when universal and existential quantifiers are involved: to prove false an unbounded universal (Π_1) statement, a checking algorithm must enumerate all instances of the bounded variable (of which there are usually infinitely many) that follow from the $\forall$ sign, whereas, if the statement is true, the checking program ends up running forever.

Roughly speaking, PrA does not suffer from such difficulties by way of quantifier reduction: any unbounded universal or existential statement in PrA can be first written into conjunctive normal form (CNF), then unravelled into a long unbounded-quantifier-free (Δ_0) sentence, whose truth value is decidable via mechanical enumeration of all pertinent variables'

truth values. The reduction step is implemented via modulo arithmetic. The standard algorithm to decide the truth value of PrA statements is Cooper's Algorithm, whose time complexity is $O(n^3)$.

Project Overview

The objective of this project is to tackle the problem of efficiently deciding PrA statements. The project builds upon an old Haskell library, Data.Integer.SAT, which provides an existing framework that implements the primitives, propositions, and symbolic syntax of PrA, as well as a checkSat function that implements Cooper's Algorithm. Nevertheless, the library suffers from several detrimental challenges that make its deciding procedure impractical to use:

1. The library itself was last maintained in 2019, and thus does not support any GHC versions compatible with ARM-architecture chips - the library must be built locally to run on a modern version of Haskell
2. The algorithmic design of the original library prioritises the branching of coefficient exploration without effective pruning - a direct consequence of this design choice is that the program freezes whenever the proposition includes two-sided variable bound constraints, which quickly results in the program OOMing
3. The library does not support explicit parallel computation, which distributes workload across multiple cores to reduce runtime

This project addresses the aforementioned challenges in three steps:

1. Refactoring the existing library systematically to provide a basic, functioning single-thread solver as a baseline
2. Implementing an alternative deciding function that supports parallelisation
3. Designing benchmarks and text cases to quantitatively test the improvement of the parallel program
4. Exploring different parallelisation strategies and providing a set of outstanding configurations

Library Refactoring

Two changes that addressed bound pruning OOM are detailed below. The tested cases solve time were reduced from ∞ (impossible to solve) to within one minute after these changes.

1. Added a normalised bound procedure to mitigate coefficient blowup, which in the original library caused unnecessary branching and quick OOM:

```
None
normalizeBound :: BoundType -> Bound -> Bound
normalizeBound _ (Bound 1 t) = Bound 1 t
normalizeBound bt (Bound c t)
  | Just k <- isConst t =
      case bt of
        -- t < c*x  means  x > t/c  means  x >= ceil((t+1)/c) = floor(t/c) + 1
        -- So floor(t/c) < x, i.e., Bound 1 (floor(t/c))
        Lower -> Bound 1 (tConst (div k c))
        -- c*x < t  means  x < t/c  means  x <= floor((t-1)/c)
        -- So x < floor((t-1)/c) + 1, i.e., Bound 1 (floor((t-1)/c) + 1)
        Upper -> Bound 1 (tConst (div (k - 1) c + 1))
  | otherwise = Bound c t
```

2. Updated the gray case to avoid allocating performance-costly lists inside solveIsNeg’

```
None
- gray = [ ctEq (b |*| tVar x) (tConst i |+| beta)
-                                     | i <- [ 1 .. b - 1 ] ]
- solveIsNeg real
- foldl orElse (solveIsNeg dark) (map solveIs0 gray)

+ grayOrDark :: S ()
+ grayOrDark =
+      solveIsNeg dark `orElse` grayRange 1 (b - 1)
+      where
```

```
+        grayAt :: Integer -> S ()
+        grayAt i =
+          let eqi = ctEq (b |*| tVar x) (tConst i |+| beta)
+          in solveIs0 eqi
+        grayRange :: Integer -> Integer -> S ()
+        grayRange lo hi
+          | lo > hi = mzero
+          | lo == hi = grayAt lo
+          | otherwise =
+              let mid = (lo + hi) `div` 2
+              in grayRange lo mid `orElse` grayRange (mid + 1) hi
+ solveIsNeg real
+ grayOrDark
```

Sequential Optimisations

Data.Map Map was replaced with IntMap + modulo arithmetic pairing in multiple places resulting in ~30% time save. Furthermore, additional pruning was introduced and iApSubst (handles solved variable substitution) is revamped from direct list iteration to IntMap lookup + then substitution, which avoids iterating through unused variables.

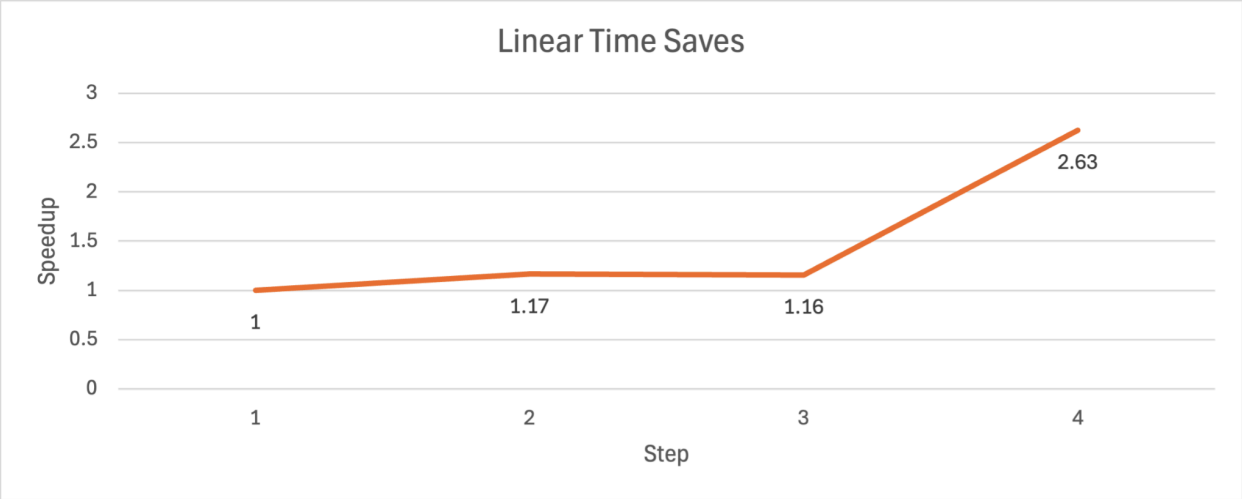
```
None
- iApSubst :: Inerts -> Term -> Term
- iApSubst i t = foldr apS t $ Map.toList $ solved i
-   where apS (x,t1) t2 = tLet x t1 t2

+ iApSubst :: Inerts -> Term -> Term
+ iApSubst is (T n m) =
+   IntM.foldlWithKey' step (T n IntM.empty) m
+   where
+     defs = solved is

 +    step :: Term -> Int -> Integer -> Term
```

```
+    step acc k c
+      | c == 0 = acc
+      | otherwise =
+          case IntM.lookup k defs of
+            Nothing   -> addCoeffK k c acc
+            Just defT -> addScaledTerm c defT acc
```

Together, they contributed to a ~3x speedup in solving the benchmark. See chart below for details: (Left to right: original -> IntMap -> Strict Fold -> iApSubst)



Parallel Optimisations

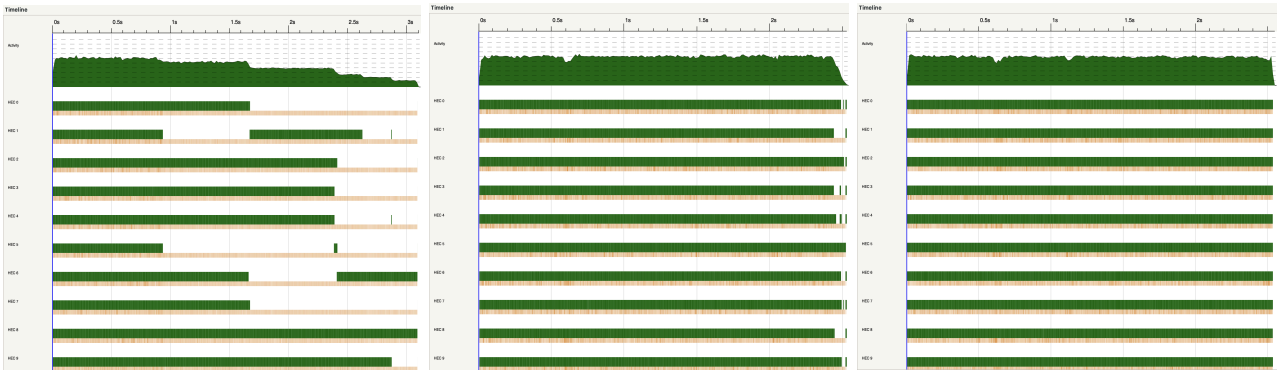
The structure of the checkSat function invites a parallel revision that can improve performance. Specifically, it is a DFS search along the branched out search tree of potential variable configurations, Choice (Answer a) (Answer a)

By sending one side of the choice to another core, we can achieve parallelism. However, since checkSat traverses the tree of many branches, it is important to decide how many times a spark should be created to do parallel work. Hence, the project implements checkSatPar (see below), which pars the right side of Choice, and also controls spark generation via a depth cutoff. Additionally, because of the way gray and grayRange is revamped, the Choice chain is more balanced to reduce the possibility of a very expensive disjunction chain.

```
None
checkSatPar :: Int -> PropSet -> Maybe [(Int, Integer)]
checkSatPar cutoff (State m) =
  go cutoff m
  where
    go _ None = Nothing
    go _ (One rw) = Just [(x, v) | (UserName x, v) <- iModel (inerts rw)]
    go d (Choice m1 m2)
      | d == cutoff =
          goChoice d m1 m2
      | otherwise = goChoice d m1 m2

    goChoice d m1 m2
      | d <= 0 = go 0 m1 <|> go 0 m2
      | otherwise =
          let r = go (d - 1) m2
              l = go (d - 1) m1
           in r `par` (l <|> r)
```

With logging and Threadscope, depth choices of 4, 8, and 16 are compared to explore efficient sparking configuration:

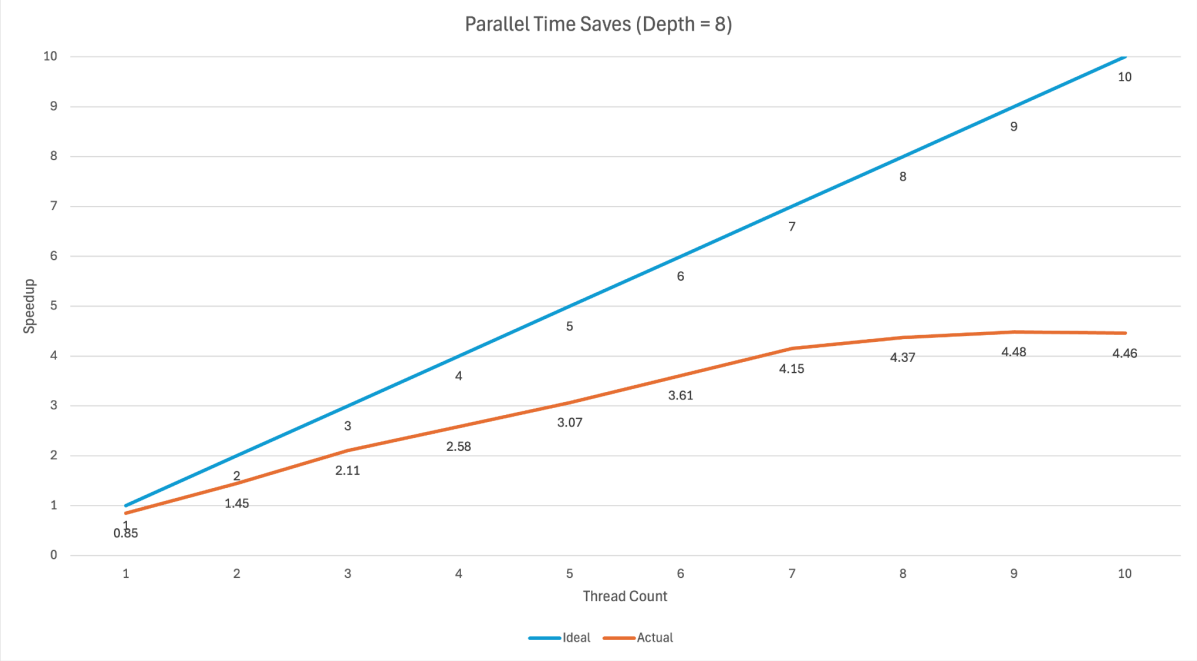


Left to right: Depth 4, 8, 16, respectively, on 10 cores.

Clearly, the low depth fails to spark sufficient work for all cores to execute simultaneously. On an average of 5 runs, it also took ~14% more time to execute than the other two cases (2.94s vs 2.58s)

Nevertheless, when examining the spark information details, around half (32k/65k) of the sparks were GC'ed in the depth-16 case. Furthermore, out of all remaining sparks, another 32k were fizzled, and <1k remaining sparks were converted. Meanwhile, in the more efficient depth-8 case, no sparks were GC'ed, ~30% were converted (79/255), and the remaining fizzled.

The project proceed to use 8 as depth to explore parallelisation behaviour on different numbers of cores:



Setting p = 0.87 and evaluating Amdahl's law equation:

$$S(N) = \frac{1}{(1-p) + \frac{p}{N}}$$

S(10) = 4.61, close to 4.46x observed speedup. suggesting that the library is around 87% parallelisable. Setting N = ∞ gives an optimal speedup under Amdahl's law at ~7.7x

Conclusion

This project explored Cooper's algorithm to decide the truth value of Presburger Arithmetic statements, and evaluated the performance of both sequential and parallel optimisation efforts.

Overall, the sequential approaches of implementing better data structures and more effective pruning brought the original library into a usable state. Nevertheless, parallelisation also substantially improved code efficiency. Furthermore, the optimisation strategies implemented by this project can serve as a reference for improvement efforts on similar projects that involve binary tree traversal.

References

1. 2025. Haskellorg. [accessed 2025 Dec 19]. https://hackage.haskell.org/package/presburger.
2. Cooper D.C. Theorem Proving in Arithmetic without Multiplication. https://www.cs.cmu.edu/~emc/spring06/home1_files/Cooper.pdf.