Parallel Congruence Closure SAT solver

Enrico Martini, VR445204

***Abstract*—In this report is presented a parallel implementation of a congruence closure algorithm for deduction in ground equational theories, able to solve a set of clauses in the quantifiers free fragment of first order logic, based on equality among variables, constants, function applications, recursive data structures with their elements and elements of arrays.**

I. Introduction

The first theory considered is the class of SMT problems is called EUF (Equality with Uninterpreted Functions), containing atoms that are equalities between terms built over uninterpreted function symbols. EUF (i.e., SAT modulo the theory of congruences) is important in applications such as the verification of pipelined processors, where, if the control is verified, the concrete data operations can be abstracted by uninterpreted function symbols. [1] It is the most important theory because its congruence closure algorithm is the core of the entire solver. The implemented algorithm also integrates the theory of lists $\mathcal{T}_{cons}$ and the theory of array without extensionality $\mathcal{T}_{A}$. In order to reduce the computational complexity of the alghoritm, a parallel version of the solver has been implemented.

II. Methodology

A. Algorithm

The most interesting feature of this implementation is the organization of the information within the data structures, shaped to be efficient.

a) Node: The design of the `Node` structure was mainly inspired by the interpretation of *'The Calculus of Computation'* [2], which describes a resolution procedure for the above mentioned theories. The `id` field holds the node's unique identification number; the `fn` field holds the constant or function symbol and the `args` field holds a list of identification numbers representing the function arguments. The `find` field holds the identification number of another node (possibly itself) in its congruence class. Following a chain of `find` references leads to the representative of the congruence class. A representative node's find field points to the node itself. If a node is the representative for its congruence class, then its `ccpar` (for congruence closure parents) field stores the set of all parents of all nodes in its congruence class.

```
class Node{
private:
    std::string         fn;
    int                 id;
    std::vector<int>    args;
    int                 find;
    std::vector<int>    ccpar;
};
```

b) Clause: The `Clause` class is used to save nodes while maintaining the given input relationship. It allows two nodes to be related but has no methods to compare them, as they are used in another class.

```
class Clause{
private:
    Node n1;
    Node n2;
    bool is_equal;
}
```

c) Formula: The `Formula` class contains a single vector of clauses. This structure is the exact transposition of the given input string to be resolved. Once the formula is created the initial string can be discarded because all relevant information has been saved.

```
class Formula{
private:
    std::vector<Clause> v_set;
};
```

d) Sat: The `Sat` class acts as a wrapper for the entire library. It contains methods for interfacing with the solver and methods for string parsing. Inside it is saved the initial translated formula and the set of nodes on which the congruence closure will be performed. There are also two index vectors, useful for checking the type of elements. In this case, since the array theory is also included, it has been mandatory to introduce a type checking system that is able to detect any errors in the provided string, such as the comparison between two arrays that is not possible to do in the array theory without extensionality, due to the fact that the decision procedure for T_A-satisfiability of quantifier-free $\sum_A$-formula $\mathcal{F}$ is based on a reduction to T_E-satisfiability via applications of the *(read-over-write)* axioms [2], shown in equations 1 and 2.

$$\forall a, v, i, j. i = j \rightarrow select(store(a, i, e), j) = e \quad (1)$$

$$\forall a, v, i, j. i \neq j \rightarrow select(store(a, i, e), j) = select(a, j) \quad (2)$$

There are two functions to check that the string is acceptable to the parser and then to the solver. While the `is_legal` function checks that an array type element is not in an equal relationship, the `well_formed` function checks that the syntax accepted as input is valid, for example by checking the number of open and closed brackets. The `transform_node`, `initialize_DAG`, `split` and `split_arguments` functions are used to populate the node vector and reconstruct the formula, preparing the necessary for resolution. The `classic_congruence_closure` function performs the congruence closure by calling the functions `FIND`, `UNION`, `CCPAR`, `CONGRUENT` and `MERGE`. These methods are a realization of the algorithm explained in chapter 9 of the Bradley-Manna book [2], which consists of calling the recursive function `MERGE` for each equation in order to build the congruence classes. Once the equalities are completed, it is checked that for each inequality the two elements that must be different are not in the same class of equivalence. The `list_congruence_closure` function makes strings that also belong to list theory acceptable to the solver. After initializing the DAG, for each node n such that `n.fn` $=$ *'cons'*, it adds `car(n)` to the DAG and merge `car(n)` with `n.args[1]` and `cdr(n)` to the DAG and merge `cdr(n)` with `n.args[2]`. After doing this pre-processing operation, the `classic_congruence_closure` is performed. If the procedure fails, then the formula is unsatisfable, but if the procedure is successful, there is another check to see if there are both atoms and lists in the same congruence class. In that case the formula will be unsatisfable by axiom *(atom)* shown in equation 3, otherwise satisfable.

$$\forall x, y. \neg atom(cons(x, y)) \qquad (3)$$

The `detect_store` function is executed before parsing the formula. According to the axioms, for each `select(store(a,i,e),j)` two formulas are created, one in case `i` is equal to `j` and one in case `i` is different from `j`. The recursive `solve` function after checking that the string is properly formatted, if the string does not contain the *store* keyword, the congruence closure is performed, while if the *store* keyword is detected, `solve` is called on the formulas generated by `detect_store`. The `SOLVE` function solves a formula in disjunctive normal form. A formula is in disjunctive normal form (DNF) if it is a disjunction of conjunctions of literals. The function divides the formula into a vector of disjointed conjunctions. Once the strings have been prepared, they are divided into batches of variable size depending on the processor on which the solver is running. Each core the processor is equipped with takes charge of a batch, going to call the `solve` function in parallel. If only one of these formulas is satisfable, the procedure ends.

```
class Sat{
private:
    Formula             f;
    std::vector<Node>   n_set;
    std::vector<int>    atoms;
    std::vector<int>    arrays;

    bool     is_legal();
    bool     well_formed(std::string s);

    int      transform_node(std::string n);
    void     initialize_DAG(std::string input);
    std::vector<std::string>
             split(std::string s);
    std::vector<std::string>
             split_arguments(std::string s);

    int               FIND(int index);
    void              UNION(int i1, int i2);
    std::vector<int>  CCPAR(int index);
    bool              CONGRUENT(int i1, int i2);
    void              MERGE(int i1, int i2);
    bool     classic_congruence_closure();
    bool     list_congruence_closure();

public:
    static std::vector<std::string>
             detect_store(std::string input);
    static bool solve(std::string s);
    static bool
    SOLVE(std::string input,std::string mode);
};
```

Finally, a heuristic was introduced trying to achieve further improvements in performance. After initializing the DAG, a forbidden list is created in which all the inequality clauses are present. Before performing the congruence closure, it is checked if equality and inequality with the same nodes are present. In this way it is possible to decide that the formula is unsatisfactory without even performing the congruence closure. In case the procedure starts, at each merge it is checked if the elements to be merged are part of the list of clauses contained in the forbidden list. If so, the procedure ends before it is even finished.

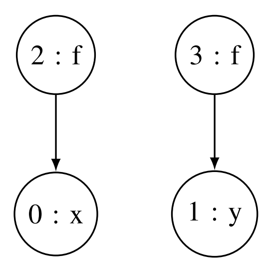
B. Equality theory congruence closure example

Let's see now with a practical example how congruence closure is performed. The formula 4 is passed as an argument to the `solve` function.

$$\mathcal{F} : x = y \wedge f(x) \neq f(y) \qquad (4)$$

```
x=y&f(x)!=f(y)
```

The string is well formatted and contains no *store* keyword, so the parser creates four nodes: two constants (x and y) and two functions ($f(x)$ and $f(y)$). At this point there are four congruence classes, one for each element. Every element is therefore representative of its own class.

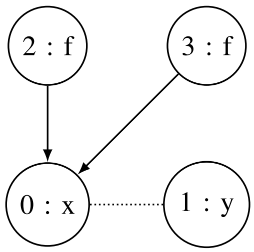


node	find	ccpar
0x	0	2
1y	1	3
2f->0	2	-
3f->1	3	-

Initially, the $x = y$ clause is taken into account. The `merge` between x and y is made.

```
MERGE 0 1
UNION 0 1
```

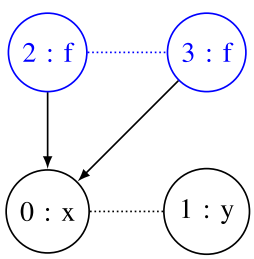
Now x and y are part of the same congruence class and the class representative is x. The `ccpar` of y have been moved to x, which now contains all the useful information for the congruence class.



node	find	ccpar
0x	0	23
1y	0	–
2f->0	2	–
3f->1	3	–

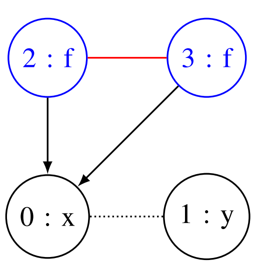
Indeed, the `merge` is recursively done between $f(x)$ and $f(y)$, since they are parents of x and y, have the same name, the same number of arguments and the arguments belong to the same congruence class.

```
MERGE 2 3 ?
CONGRUENT 2 3 = 1
MERGE 2 3
UNION 2 3
```



node	find	ccpar
0x	0	23
1y	0	–
2f->0	2	–
3f->1	2	–

At this point the equalities are solved. We begin to see for each inequality if its elements belong to the same congruence class. In this case $f(x)! = f(y)$, but from the last merge it can be seen that $f(x)$ and $f(y)$ belong to the same congruence class, so the formula is unsatisfable.




```
UNSAT
```

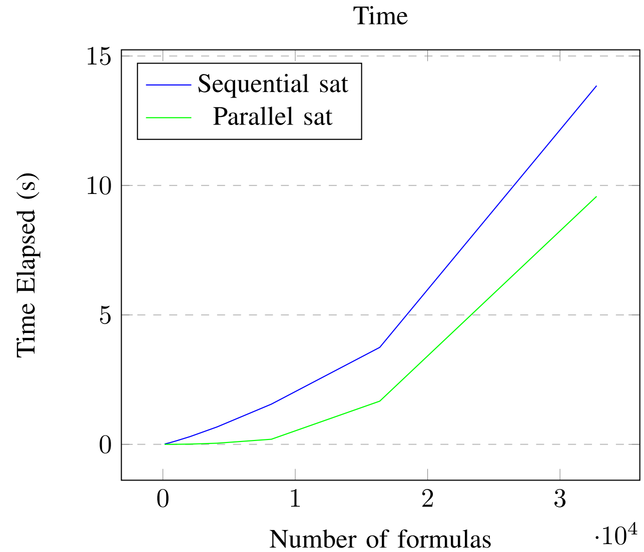
III. Validation

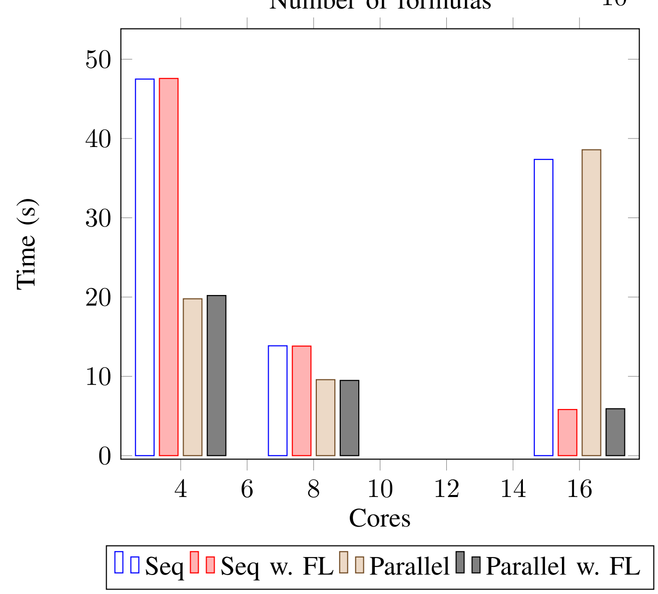
IV. Benchmarks

TABLE I
PERFORMANCE RESULTS WITH USE OF FORBIDDEN LIST

Test	#Formulas	#Equations	Sequential (s)	Parallel (s)	Speedup
7	128	1920	0,0128	0,0001	97,3
8	256	4352	0,0273	0,0003	107,6
9	512	9728	0,0605	0,0007	88,6
10	1024	21504	0,1341	0,0024	57,0
11	2048	47104	0,2964	0,0099	29,9
12	4096	102400	0,6639	0,0430	15,5
13	8192	221184	1,5309	0,1854	8,3
14	16384	475136	3,7426	1,7264	2,2
15	32768	1015808	13,8145	9,4844	1,5

V. Performance Analysis





VI. Conclusion

References

[1] R. Nieuwenhuis and A. Oliveras, "Fast congruence closure and extensions," *Information and Computation*, vol. 205, no. 4, pp. 557 – 580, 2007. Special Issue: 16th International Conference on Rewriting Techniques and Applications.

[2] A. R. Bradley and Z. Manna, *The Calculus of Computation: Decision Procedures with Applications to Verification*. Berlin, Heidelberg: Springer-Verlag, 2007.

APPENDIX

TABLE II

Theory	Source	Formula	Result
Equality	Bradley-Manna	f(x)=f(y)&x!=y	SAT
		x=y&f(x)!=f(y)	UNSAT
		f(a,b)=a&f(f(a,b),b)!=a	UNSAT
		f(f(f(a)))=a&f(f(f(f(f(a)))))=a&f(a)!=a	UNSAT
		f(f(f(a)))=f(f(a))&f(f(f(f(a))))=a&f(a)!=a	UNSAT
		f(x,y)=f(y,x)&f(a,y)!=f(y,a)	SAT
		f(g(x))=g(f(x))&f(g(f(y)))=x&f(y)=x&g(f(x))!=x	UNSAT
	IC Tests	b=d&f(b)=d&f(d)=a&a!=b	UNSAT
		a=b1&b1=b2&b2=b3&b3=c&f(a1,a1)=a&f(c1,c1)=c&a1=c1&a!=c	UNSAT
	Z3 Benchmark	f1!=f2&f3(f4,f5,f6,f7,f8(f9))!=f1&f3(f4,f5,f6,f7,f10)=f1&f10=f8(f9)&f10=f8(f9)&f3(f4,f5,f6,f7,f10)=f1	UNSAT
		f1!=f2&f3(f4,f5,f6,f7,f8(f9))!=f1&f3(f4,f5,f6,f7,f10)=f1&f10=f8(f9)&f3(f4,f5,f6,f7,f10)=f1	UNSAT
		f1!=f2&f3(f4,f5,f6,f7,f8(f9))!=f1&f3(f4,f5,f6,f7,f10)=f1&f10=f8(f9)	UNSAT
List	Bradley Manna	x1=x2&y1=y2&cons(x1,y1)!=cons(x2,y2)	UNSAT
		x=y&car(x)!=car(y)	UNSAT
		x=y&cdr(x)!=cdr(y)	UNSAT
		car(cons(x,y))!=x	UNSAT
		cdr(cons(x,y))!=y	UNSAT
		!atom(cons(x,y))	SAT
		atom(x)&cons(car(x),cdr(x))=x	UNSAT
		car(x)=car(y)&cdr(x)=cdr(y)&f(x)!=f(y)&!atom(x)&!atom(y)	UNSAT
		car(x)=y&cdr(x)=z&x!=cons(y,z)	SAT
	Intermediate exam	f(b)=b&f(f(b))!=car(cdr(cons(f(b),cons(b,d))))	UNSAT
Array	Bradley-Manna	i=k&select(store(x,i,v),k)!=v	UNSAT
		i!=k&select(store(x,i,v),k)!=select(x,k)	UNSAT
		i1=j&i1!=i2&select(a,j)=v1&select(store(store(a,i1,v1),i2,v2),j)!=select(a,j)	UNSAT
	Intermediate exam	e=select(store(a,i,e),j)&select(a,j)!=e	SAT