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 $\mathcal{T}_{cons}$.

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;
};
```

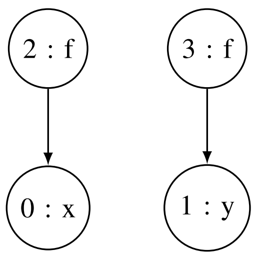
d) Sat: The `Sat` class acts as a wrapper for the entire program. It contains methods for interfacing with the solver and methods for string parsing. Inside it is saved the initial translated formula and the set of nodes on which the congruence closure will be performed. There are also two index vectors, 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 decision procedure for T_A-satisfiability of quantifier-free $\sum_A$-formula F is based on a reduction to T_E-satisfiability via applications of the (read-over-write) axioms.

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

    bool    is_legal();
    int     transform_node(std::string n);
    void    initialize_DAG(std::string input);
    bool    classic_congruence_closure();
    bool    list_congruence_closure();
    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);
public:
    static std::vector<std::string>
             split(std::string s);
    static bool well_formed(std::string s);
    static bool solve(std::string s);
    static std::vector<std::string>
          detect_store(std::string input);
};
```

B. Equality theory congruence closure example

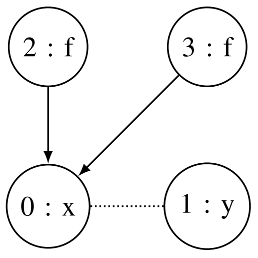
$$\mathcal{F} : x = y \wedge f(x) \neq f(y)$$

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



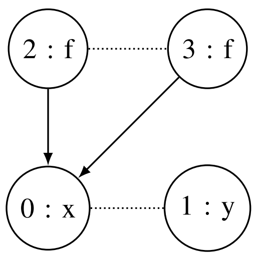
```
node                find                ccpar
0x                  0                   2
1y                  1                   3
2f->0               2                   -
3f->1               3                   -
```

```
MERGE 0 1
UNION 0 1
MERGE 2 3 ?
CONGRUENT 2 3 = 1
```

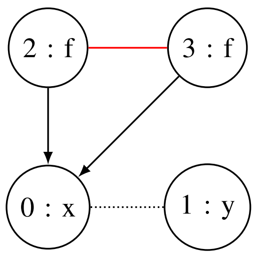


```
node                find                ccpar
0x                  0                   23
1y                  0                   -
2f->0               2                   -
3f->1               3                   -
```

```
MERGE 2 3
UNION 2 3
```



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

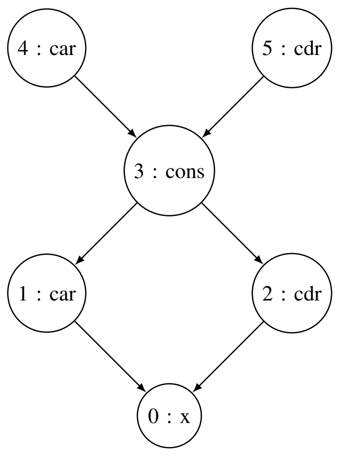


```
UNSAT
```

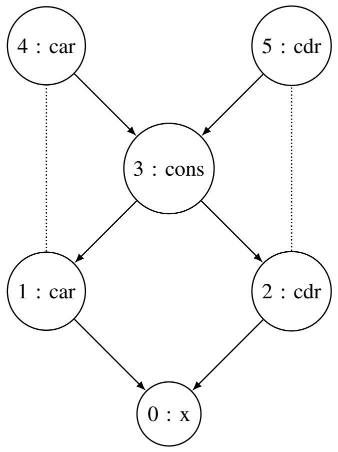
C. List theory congruence closure example

$$\mathcal{F} : cons(car(x), cdr(x)) = x \wedge atom(x)$$

```
atom(x)&cons(car(x),cdr(x))=x
```

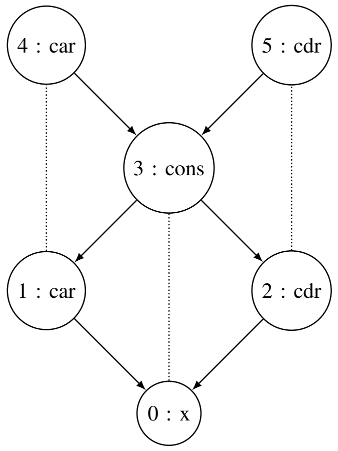


```
MERGE 4 1
UNION 4 1
MERGE 5 2
UNION 5 2
```

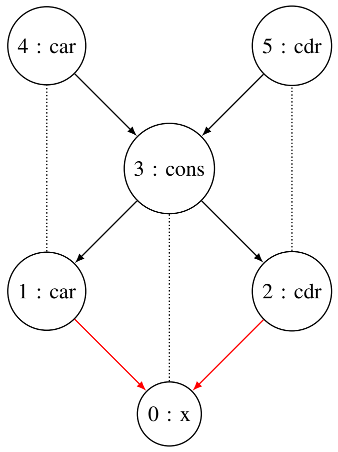


node	find	ccpar
0x	0	12
1car->0	4	–
2cdr->0	5	–
3cons->12	3	45
4car->3	4	3
5cdr->3	5	3

```
MERGE 3 0
UNION 3 0
MERGE 4 1 ?
MERGE 4 2 ?
CONGRUENT 4 2 = 0
MERGE 5 1 ?
CONGRUENT 5 1 = 0
MERGE 5 2 ?
```



node	find	ccpar
0x	3	–
1car->0	4	–
2cdr->0	5	–
3cons->12	3	4512
4car->3	4	3
5cdr->3	5	3



```
Euality theory passed
UNSAT
```

D. *Array theory congruence closure example*

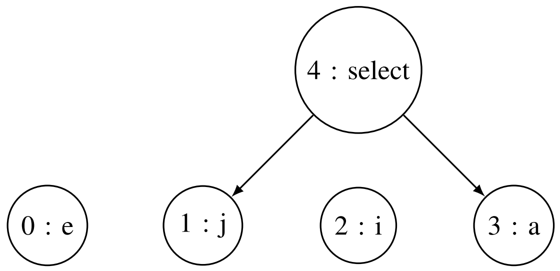
$$\mathcal{F} : e = \mathit{select}(\mathit{store}(a, i, e), j) \wedge \mathit{select}(a, j) \neq e$$

```
e=select(store(a,i,e),j)&select(a,j)!=e
```

```
detected store
```

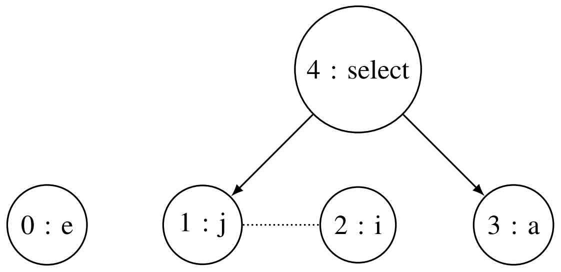
$$\mathcal{F}_1 : e = e \wedge j = i \wedge \mathit{select}(a, j) \neq e$$
$$\mathcal{F}_2 : e = \mathit{select}(a, j) \wedge j \neq i \wedge \mathit{select}(a, j) \neq e$$

```
1: e=e&j=i&select(a,j)!=e
2: e=select(a,j)&j!=i&select(a,j)!=e
```

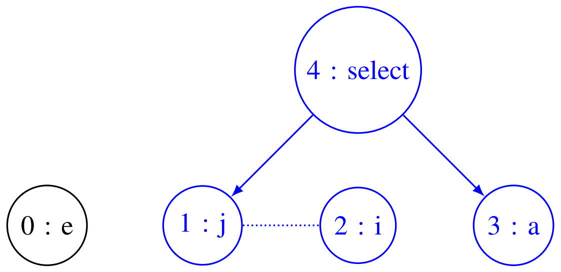


node	find	ccpar
0e	0	–
1j	1	4
2i	2	–
3a	3	4
4select->31	4	–

```
MERGE 0 0
MERGE 1 2
UNION 1 2
```



```
node                find                ccpar
──────────────────────────────────────────────────
0e                  0                   –
1j                  1                   4
2i                  1                   –
3a                  3                   4
4select->31         4                   –
```



```
Euality theory passed
SAT
```

III. Validation

IV. Benchmarks

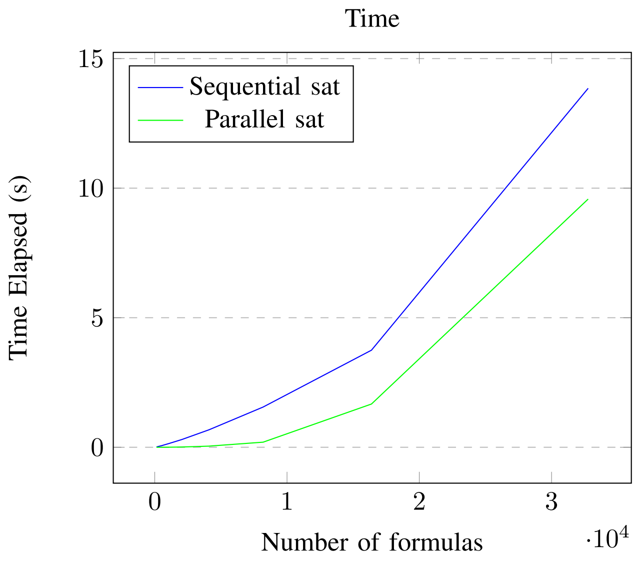
TABLE I
Performance results with simple algorithm.

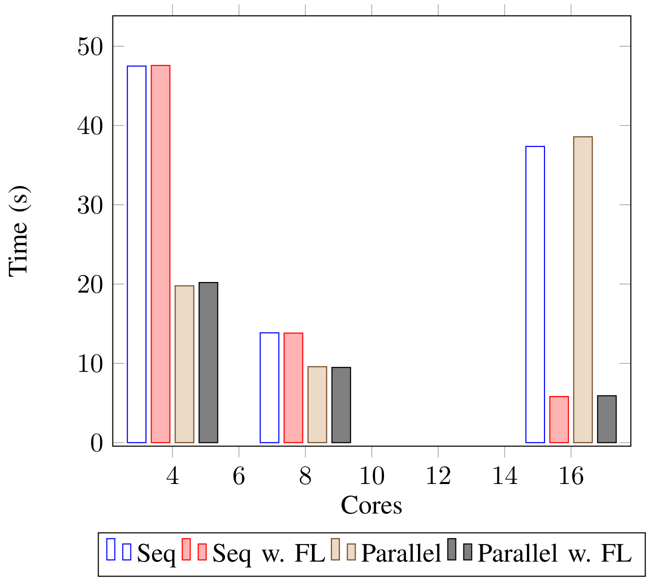
Test	#Formulas	Sequential (s)	Parallel (s)	Speedup
7	128	0,0117	0,0001	82,1
8	256	0,0290	0,0003	100,1
9	512	0,0612	0,0008	78,8
10	1024	0,1374	0,0026	52,1
11	2048	0,2988	0,0103	29,1
12	4096	0,6736	0,0442	15,3
13	8192	1,5526	0,1968	7,9
14	16384	3,7465	1,6690	2,2
15	32768	13,8530	9,5786	1,4

TABLE II
Performance results with use of forbidden list

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

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