Parallel SAT Solving - Using More Cores

Norbert Manthey

Norbert.Manthey@tu-dresden.de

18.06.2011

Outline

1. Introduction
2. Preliminaries
3. The Parallelization
4. Results
5. Open problems

Motivation

SAT solving is widely applied, however

- most solvers are sequential
- architecture becomes parallel
- a good scalable parallelization is still missing

How to parallelize the solving process?

CDCL Solvers

Modern SAT solver are based on DPLL with many improvements.

- Conflict analysis
  - Undo more decisions than only the last one (usually)
  - The variable order on the current path changes often
- Restarts
- Learned clause removal
- Advanced algorithms
  - Conflict resolution is linear (15%)
  - Two-Watched-Literal unit propagation (80%)

Current Approaches

- Run the same solver multiple times
  - sharing learned clauses and more
- Split the search
  - by splitting the search tree and solving subformulas
  - by splitting the formula and finding multiple models per subformula
- Run additional tasks in separat threads (e.g. autarky detection)

Problems:

- Higher memory bandwidth / more memory accesses
- Splitting might not lead to easier subformulas

Why not parallelize the solver itself?

Solver analysis

Runtime properties of an exemplary SAT solver

(all measurements are based on riss and the SAT09 industrial benchmark, 1h timeout)

- Unit propagation needs 80 %
- Propagating a literal reveals more implied literals
  - at least 2 more in 13 % of the cases
  - at least 4 more in 4 % of the cases

Suggestion:

Parallelize UP by splitting the formula,
share implied literals found in subformulas

Necessary steps

1. Separate formula into n partitions
   - Use function *Assign*: clause $\rightarrow$ partition
   - Currently: alternating
2. Create n threads that execute UP in parallel
   - create assignment, trail, propagation queue per thread
   - choose a master thread
   - initialize structures with according clause partition
3. Use *Assign* to distribute learned clauses
4. Do backjumping for all threads
5. Remove learned clauses in all threads
   - Can result in unbalanced load

Sequential propagation

```
traditionalPropagate(){
  C = 0;
  while( not myQueue.processed() ){
    l = myQueue.dequeue(); // keep on queue
    C = propagate( l );    // enqueues implied literals
    if( C != 0 ) break;
  }
  return C;
}
```

Sequential propagation uses its private propagation queue, assignment and trail.

The parallelization per thread

Master thread executes CDCL algorithm and wakes slave threads for UP

```
propagate(){
 while( (not all finished) and (no conflict signaled)){
   C = traditionalPropagate();
   if( C != 0 ){ signalConflict( C ); break; }
   check all other threads for new literals;
 }
 if( conflict not from master ){
  updateMaster();
 }
}
```

Reading other threads data is lock- and waitfree.

Example

Given scenario: Formula with 5 clauses, 2 threads

$$F = \langle [\neg 1, 2], [\neg 1, 4], [\neg 2, 3], [\neg 2, 5], [\neg 4, \neg 5] \rangle$$

Splitted formula: T_1: $[\neg 1, 2], [\neg 2, 3], [\neg 4, \neg 5]$ T_2: $[\neg 1, 4], [\neg 2, 5]$

Algorithm execution:

T_1:	Reason	-	C_1 C_3		C_2	C_5					C_4
	Queue	1	2 3	S	4	$\neg 5$				S	**5**
	Step	0	1	2	3	4	5	6	7	8	9
T_2:	Queue	1	4				S	2 3 $\neg 5$	**5**		
	Reason	-	C_2					C_1 C_3 C_5	C_4		

T_2 signals conflict in step 7

T_1 performs update in step 8 and 9

Propagation results in conflict

Results SAT Race 2010

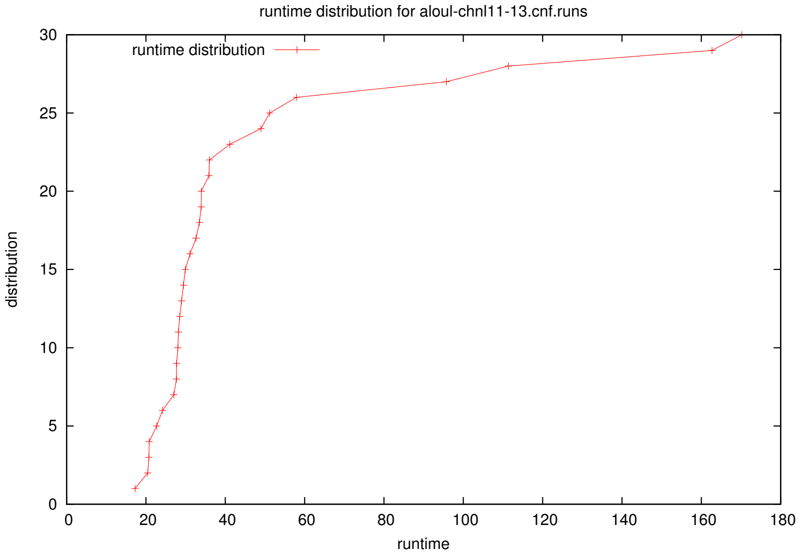
Conditions: 100 instances, 15 minutes timeout

Configuration	Seq1	D1	D2	D3
Solved instances	48	61	64	68
Average runtime	191.721	219.941	193.019	215.212

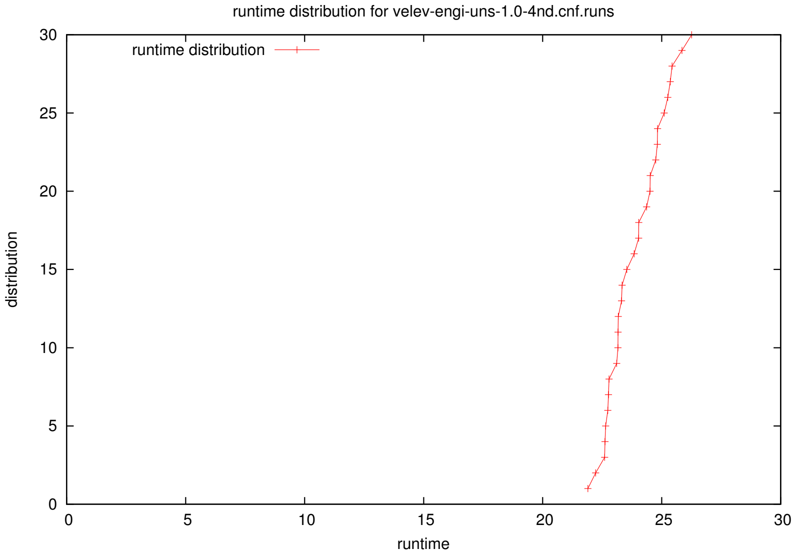
Speedup on the 41 commonly solved instances

Average	Maximal	D1	Median
1.091	1.578	1.28	1.3

Runtime distribution on a single instance



Runtime distribution on a single instance



Open problems

Problems:

- waiting times (4 % idle, 1.5 % system)
- no load balancing
- scales not beyond 2 threads

Possible Solutions:

- find a well working *Assign* function
- introduce load balancing to reduce idle times
- use spin locks instead of the sleep-state
- **combine presented approach with existing methods**

Conclusion

- Parallel UP is possible
- Not many additional memory accesses are needed
- Speedup is not yet optimal
- Further analysis has to be done

- Presented approach can be combined with existing solutions
- Used numer of cores can be doubled

Thanks

The solver is available at `https://gitorious.org/riss`