ECE 595Z Spring 2013

Digital Systems Design Automation

Course Project : Parallel implementation of Minisat SAT Solver on Threaded Building Blocks (TBB) platform.

Team : Sambit Mishra & Mahesh Ramesh

Abstract

This project aims to implement a parallelized version of a state-of-the-art sequential SAT solver. The Threaded Building Blocks (TBB) platform is used to realize this implementation. The parallel solver uses a portfolio of randomized sequential algorithms which are obtained by multiple realizations of the standard DPLL algorithm. Sharing of conflict clauses among worker threads is also implemented. Though the design methodology is simple, it is shown that the proposed solver performs better than the reference solver.

Introduction & Motivation

Current State-of-the-art SAT solvers are based on the standard DPLL algorithm with additional optimization techniques such as Non-chronological backtracking, Boolean Constraint Propagation and the concept of watched literals. However with the exponential increase in the input-instance size, it has become imperative that these need be solved as efficiently as possible. Their efficiency and speed hinges on the appropriate usage of the available hardware and low-level processes.

Thus there is huge scope for modern SAT solvers to utilize multi-core architectures. Being inherently sequential from a theoretical and worst-case analysis point of view, SAT indeed gives rise to many challenges in multi-core implementation.

Many parallel SAT solvers have been proposed earlier. Most of them are based on divide-and-conquer approach, where they divide the search space. Another class of parallel solvers are portfolio-based solvers which employ complementary strategies which compete and co-operate on the same formula. Since each worker works on the whole formula, there is no need to introduce load balancing overheads and cooperation is only achieved through exchange of learnt clauses. In this approach, the objective is to cover the space of good search strategies in the best possible way.

A qualitative survey was made on both the approaches and after a reasonable analysis considering the complexity of implementation, we realized that a solver employing multiple workers could be implemented within the given time and achieve superior performance to sequential solver.

TBBSat Implementation:

We call our modified solver as "**TBBSat**" based on the fact that Threaded Building Blocks (TBB) platform is used to build the solver. Minisat, one of the best sequential solvers available, is the core of TBBSat. In a nutshell, TBBSat is the parallel realization of multiple instances of Minisat core engine.

Overview of Minisat Implementation

Minisat is based on the DPLL algorithm, backtracking by conflict analysis and clause recording, and boolean constraint propagation using watched literals. Conceptually, the components of this solver can be divided into three categories:

- **Representation** – the SAT instance must be represented by internal data structures, as must any derived information.
- **Propagation** – Each literal may propagate unit information among its constraint clauses if the literal becomes TRUE. For clauses no unit information can be propagated until all the literals except one have become FALSE. Two unbound literals, p and q, called the watched literals are selected and said to be watched.
- **Search** –It involves selection of literal to be propagated, assuming a value for it and iteratively searching clauses at all decision levels. The consequences of a literal, say x=TRUE will then be propagated, possibly resulting in more variable assignments. All assignments will be stored on a stack in the order they were made; referred to as trail. The trail is divided into decision levels and is used to undo information during backtracking.
- **Learning** – conflict clauses are identified and added to the original clause database in this step. The step is invoked when a constraint becomes conflicting under the current assignment.

  Learnt clauses serve two purposes: they drive the backtracking and they speed up future conflicts by 'caching' the reason for conflict.
- Apart from the above mentioned steps, there are various optimization techniques that employed which will be explained later as and when required.

Pseudocode of Minisat:

```
parse the input clauses
perform trivial assignments and determine conflicts if any
loop
    propagate( )                        //propagate unit clauses
    if not conflict then
        if all variables assigned then
            return SATISFIABLE
        else
            decide( )                   //pick a new variable and assign it
    else
        analyze( )                      //analyze conflict and add conflict clause
        if top-level conflict found then
            return UNSATISFIABLE
        else
            backtrack()                 //undo all assignments until conflict clause is unit.
end loop
```

Modifications made to the existing code

Since the objective of this work was parallelizing Minisat solver, we looked at various blocks in the algorithm which were potential candidates for concurrent runs. The dependency between different blocks is significant which limits the possibility of extensive parallelism. Instead of trying to parallelize a single block, we decided to create multiple instances of the solver core and run them concurrently on different threads.

The reasoning behind this approach is as follows:

- The Solver core is a highly interdependent tree structure. This makes it challenging to parallelize a particular block of the core without neglecting data handling complexities.
- Parallelizing the entire solver core implies that worker threads compete against each other. The performance of such system would only be better than or equal to a single core solver.
- The proposed method would significantly increase memory. However, in most cases, execution time is more critical than the memory utilization factor. In these cases, the proposed solver is an ideal fit.
- The method derives motivation from earlier SAT solvers like CHAFF whose motto was to do simple calculations repetitively. Here instead of optimizing the individual blocks, the entire solver core is parallelized with careful optimizations.

The pseudocode of proposed algorithm is as follows:

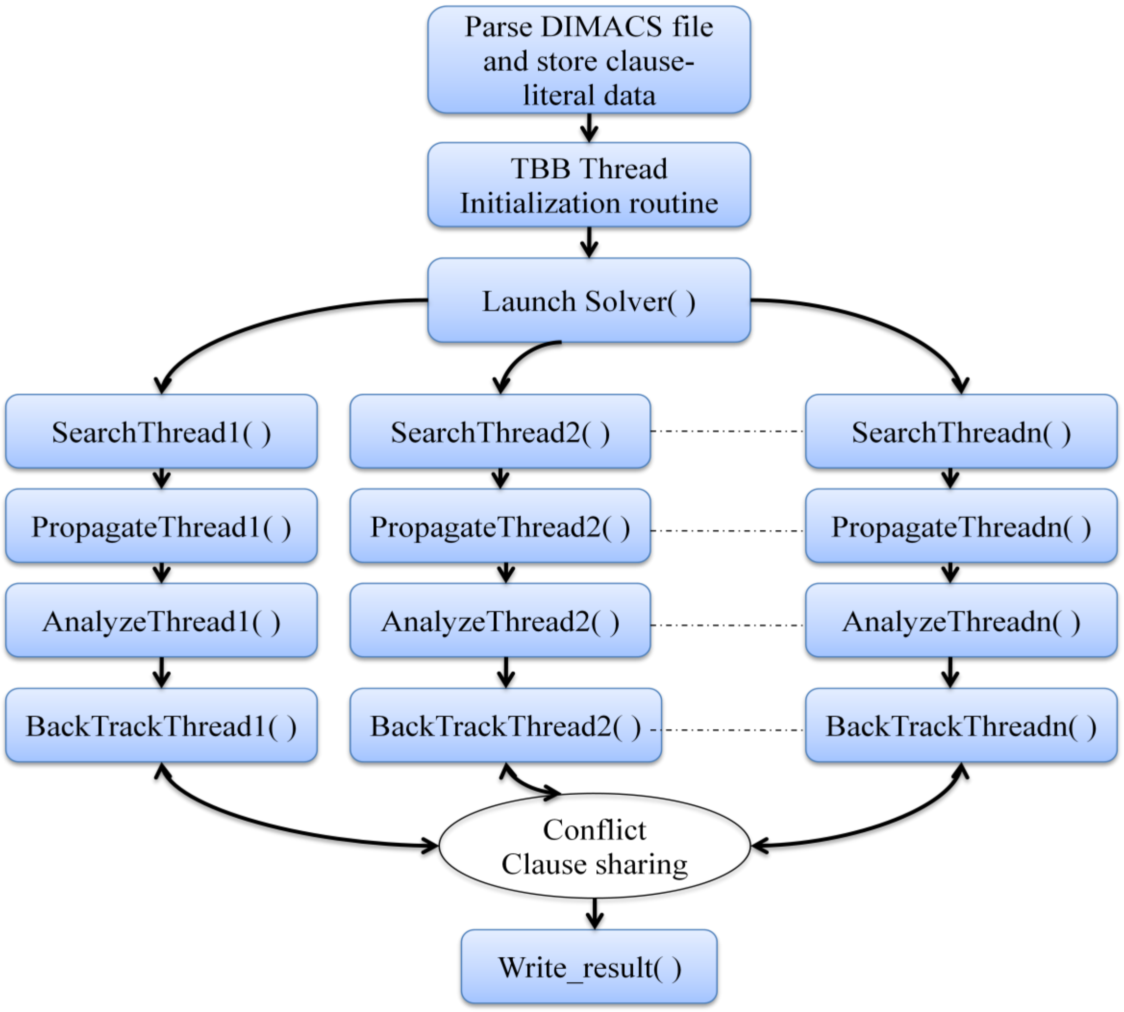
```
parse the input clauses
perform trivial assignments and determine conflicts if any
TBB_thread_schedule( )              //determine no of free threads and initialize them
loop                          //run for each thread
    Solver(thread_id)               //initialize worker for each thread
    loop
        propagate()                 //propagate unit clauses
        if not conflict then
            if all variables assigned then
                protected_result_write()
                return SATISFIABLE
            else
                decide( )           //pick a new variable and assign it
        else
            analyze( )              //analyze conflict and add conflict clause
            if top-level conflict found then
                return UNSATISFIABLE
            else
                backtrack( )        //undo all assignments until conflict clause is unit.
    end loop
end loop
```

TBB as a parallel platform:

Threaded Building Blocks is a library from Intel that supports scalable parallel programming using standard ISO C++ code. It can be easily integrated to existing languages or compilers. It fully supports nested parallelism, allowing to build larger parallel components from smaller parallel components. The library maps tasks onto threads in an efficient manner.

TBB was chosen as the platform for implementation because of its scalable features, ease of integration and ability to handle parallel tasking without having extensive knowledge of thread-level programming. Also to the best of our knowledge, TBB was not used earlier for parallel SAT solving, this motivated us to choose TBB as the platform for our implementation.

Flow diagram of our solver:



Handling data dependency between worker threads:

The solver core is launched on multiple threads and once each thread is finished executing the job (determining whether it is SAT or UNSAT), it writes to a common data structure and terminates. All other instantiated threads also terminate at this stage. This data structure is shared across threads and can lead to potential read/write errors. Hence a synchronization routine is developed in which a **mutex** *(similar to semaphore)* is used to handle the read/write operation to the above said data structure.

The solver class that is used is as follows:

```
class Solver – Public interface
        var       newVar ( )                              ……….add new variable to the clause list
        bool      addClause       (Vec<lit> literals)        ……..determine trivial conflict clauses and
                                                                  add clause
        bool      add…  (…)
        bool      simplifyDB ( )          ……………propagates unit information and removes
                                                                  satisfied constraints
        bool      solve           (Vec<lit> assumptions)
        Vec<bool> model                                   ……if found, this vector has the result ie model
```

Basis abstract data types used in the code:

```
Class Vec<T>  -Public interface
- Constructors:
Vec()
Vec (int size)
Vec (int size, T pad)

-Size operations:
int        size ()
void       shrink (int no of elements)
void       pop ()
void       growTo (int size)
void       growTo (int size, T pad)
void       clear ()

-stack interface:
Void       push ()
Void       push(T elem)
T          last ()

-vector interface:
T op[] (int index)
```

```
Class lit –Public interface
   lit (var x)

-Global functions:
lit        op (lit p)
bool       sign (lit p)
int        var (lit p)
int        index (lit p)
```

```
class lbool –Public interface
lbool () lbool (bool x)

-Global functions:
lbool op (lbool x)

-Global constants:
lbool FALSE, TRUE
```

```
Class Queue<T> –Public interface
Queue ()

Void       insert (T x)
T          dequeue ()
void       clear ()
int        size ()
```

```
Collect
-----------------------------
private:
static Collect* mCollect;
public:
Solver* s
-----------------------------
Collect()
static Collect* getInstance()
virtual ~Collect()
```

```
class Solver
    –   Constraint database
--------------------------------------------------------------------------------
//tbb_vars & methods
static int          index;
unsigned int        seed_value;
bool                random_pick;
typedef             struct
{
        int learnt_size;
        int *clause_ptr;
}cs;
tbb::concurrent_queue <cs> scq;
void     pushLearntClause (vec<Lit>&);
void     popLearntClause ( );
unsigned int maxprops;
--------------------------------------------------------------------------------
Vec<Constr>    constrs         -List of problem constraints
Vec<Clause>    learnts         - List of learnt clauses
double         cla_inc         - Clause activity increment – amount to bump with.
double         cla_decay       -Decay factor for clause activity



-variable order
vec<double>    activity        - Heuristic measurement of the activity of a variable
double         var_inc         - Variable activity increment – amount to bump with.
double         var_decay       - Decay factor for variable activity

-Propagation
Vec<Vec<Constr>> watches       - for each literal 'p', a list of constraints watching 'p'. A constraint will
                               be inspected when 'p' becomes true.
Vec<Vec<Constr>> undos         - For each variable 'x', a list of constraints that need to update when 'x'
                               becomes unbound by backtracking.
Queue<lit> propQ               - Propagation queue.

-Assignments
Vec<lbool> assigns             - The current assignments indexed on variables.
Vec<lit>     trail             - List of assignments in chronological order.
Vec<int>     trail_lim         - Separator indices for different decision levels in 'trail'.
Vec<int>     level             - For each variable, the decision level it was assigned.
```

A critical section of the code is the propagate () which performs the Boolean constraint propagation of watched literals. Its pseudocode is as follows:

```
Constr Solver.propagate()
while (propQ.size()>0)
        lit p = propQ.dequeue()                  - 'p' is now the enqueued fact to propagate.
        Vec<Constr> tmp                          - 'tmp' will contain the watcher list for 'p'
        watches [index(p)].moveTo(tmp)

        for (int i=0; i< tmp.size(); i++)
                if(!tmp[i].propagate (this, p))
                ---constraint is conflicting; copy remaining watches to 'watches[p]' and return constraint
                for (int j=i+1; j< tmp.size(); j++)
                        watches[index(p)].push(tmp[j])
                propQ.clear()
                return tmp[i]
return NULL
```

Key steps:

- The class ‘Solver’ forms the core of SAT engine which carries out directed steps as explained earlier.
- Multiple objects of this solver class is called depending upon the number of threads available and then launched.
- The worker objects in turn call search, propagate, analysis and clause learning functions.
- A new literal is picked based on past activity and randomized samples.
- The results are then collected synchronously through careful handling of parallel scheduling.

Some of the key features of both minisat and our implementation, TBBSat are as follows:

Restart strategies:

Restart policies represent an important component of modern SAT solvers. These are used to compact the assignment stack and improve the order of assumptions. Here the policy is based on the number of conflict clauses at a time. If the number is more than a threshold, the solver terminates the search and begins a new iteration with a randomized initial literal. In our implementation, the same restart policy is adopted by all worker threads.

Conflict clause sharing among threads: (Our optimization technique)

A conflict clause is determined by the Analyze() section. A breadth-first traversal is achieved by inspecting the literal queue backwards. Along with the conflict clause, backtracking level is obtained. This conflict clause now becomes a learnt clause for the solver of the corresponding worker thread. The conflict cause thus obtained is then propagated to the other worker threads.

Each solver object has a queue ,*tbb::concurrent_queue<cs> scq*. The learnt clauses are pushed onto this queue. A concurrent_queue allows multiple threads to concurrently push and pop items and thus we do not need to explicitly use mutexes.

After propagating its learnt clause to other threads, a thread checks to see if any of the clauses has been propagated to it. If it finds a clause it does one of the following things:

- See if the clause is UNSAT under current literal assignments, if yes, then solver thread returns with the answer.
- If the clause is a unit clause then we backtrack until the zeroth decision level and then propagate it.
- Analyse the clause and then propagate it in the normal way, meaning the next *propagate()* call in *search()*. Our current implementation for this part has memory and synchronization issues. When we tested it for more than 4 cores we sometimes got the wrong answer and the program crashed due to memory corruption. Hence we have limited the usage of this part to the case when we are running it on less than equal to 4 cores.

Picking a Branch literal:

At any stage of *solve()*, if there is no conflict found, the solver picks a new unassigned literal for subsequent iterations. The strategy to pick a literal arises from two paradigms:

- *Random pick* : A literal is selected randomly for reach worker thread.

-*Activity based pick* : For every learnt clause that is recorded, the activity factor of each of its literals is updated with a score. A new literal is picked based on this activity.

Experimentation:

The modified solver, TBBSat was tested on a set of benchmarks taken from SAT 2008 Competition. The benchmarks consisted of both SAT and UNSAT cases. The number of clauses ranged from 60,000 to 1,200,000 in the benchmarks. Also the number of literals varied between 25,000 to 200,000. The large size of benchmarks gives us opportunity to test TBBSat on real-world problems and allocate different number of cores for experimentation.

The solver was run on the machine "**aristotle.ecn.purdue**" which had 48 cores. This gave us the flexibility to choose the number of threads/cores depending on problem size and perform a true test of parallelization.

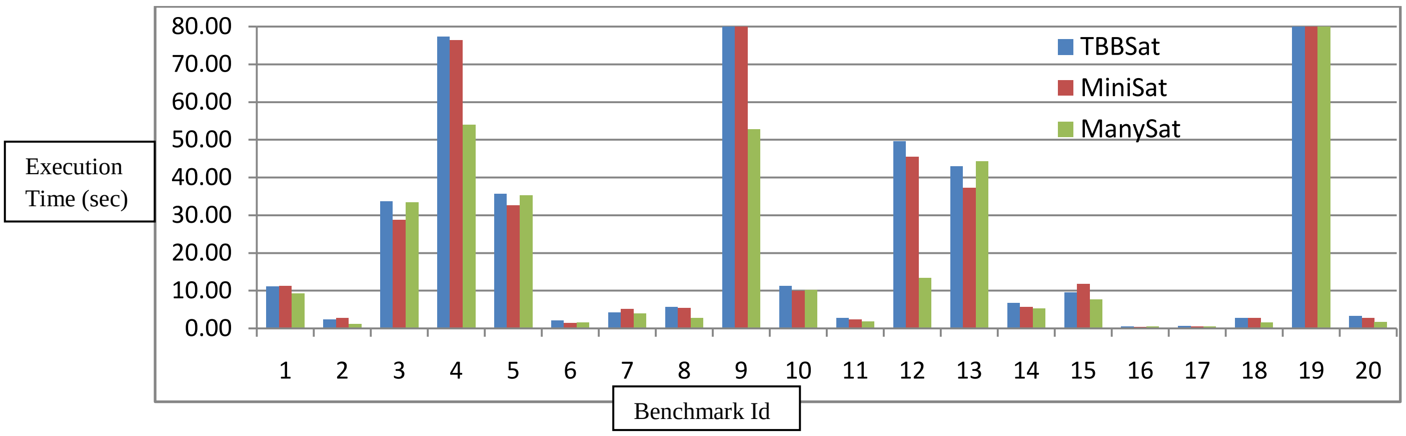
Also, the existing solver, Minisat and a recently developed parallel solver "ManySat" were run on the same benchmarks to compare the performance of TBBSat.

Results:

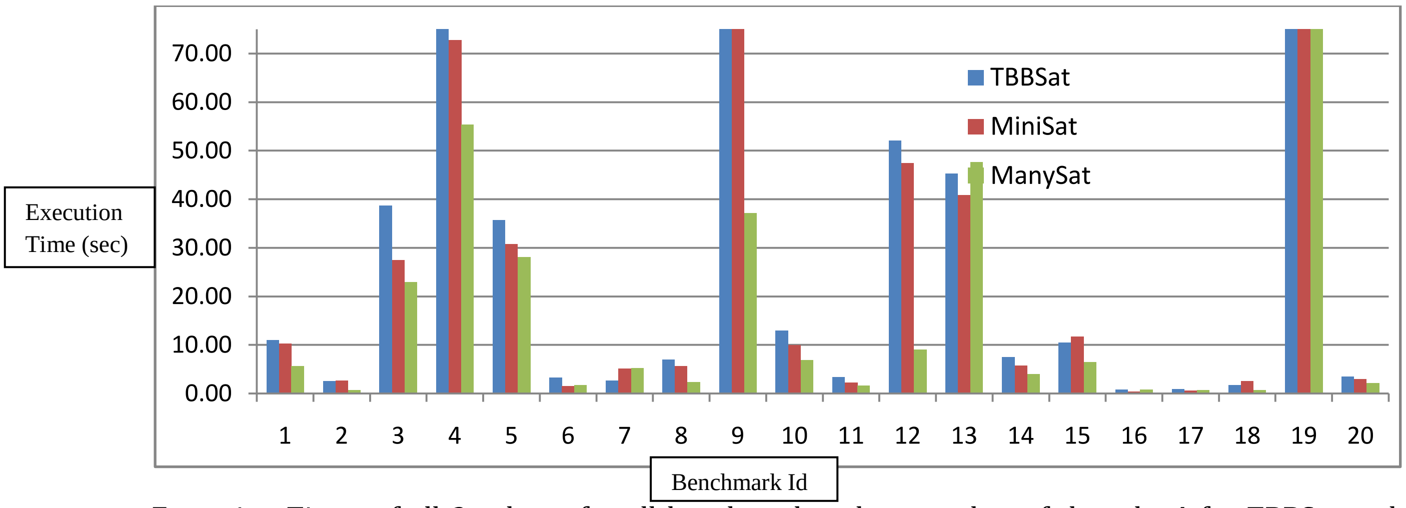
Some of the experimental data obtained are tabulated and plotted as follows:

No	Benchmark	Satisfiability	Number of literals	Number of clauses	TBBSat	MiniSat	ManySat
1	ibm-2002-05r-k90.cnf	SAT	180140	964638	0:11.14	0:11.19	0:09.25
2	ibm-2002-07r-k100.cnf	UNSAT	24767	61175	0:02.25	0:02.68	0:01.09
3	ibm-2002-11r1-k45.cnf	SAT	156626	633125	0:33.62	0:28.70	0:33.36
4	ibm-2002-19r-k100.cnf	SAT	310152	1189123	1:17.39	1:16.36	0:53.94
5	ibm-2002-21r-k95.cnf	SAT	191522	783592	0:35.64	0:32.64	0:35.26
6	ibm-2002-26r-k45.cnf	UNSAT	224477	1195347	0:02.07	0:01.41	0:01.56
7	ibm-2002-27r-k95.cnf	SAT	75824	297575	0:04.25	0:05.05	0:03.98
8	ibm-2004-03-k70.cnf	SAT	69839	275980	0:05.59	0:05.43	0:02.76
9	ibm-2004-04-k100.cnf	SAT	120445	466313	3:13.29	3:07.93	0:52.74
10	ibm-2004-06-k90.cnf	SAT	112376	481553	0:11.29	0:09.86	0:10.22
11	ibm-2004-1_11-k25.cnf	UNSAT	78503	289146	0:02.75	0:02.30	0:01.80
12	ibm-2004-1_31_2-k25.cnf	UNSAT	31125	106016	0:49.49	0:45.47	0:13.43
13	ibm-2004-19-k90.cnf	SAT	227897	875361	0:42.97	0:37.29	0:44.30
14	ibm-2004-2_02_1-k100.cnf	UNSAT	68848	239027	0:06.68	0:05.69	0:05.26
15	ibm-2004-2_14-k45.cnf	UNSAT	65032	247948	0:09.52	0:11.71	0:07.62
16	ibm-2004-26-k25.cnf	UNSAT	125791	633816	0:00.50	0:00.32	0:00.50
17	ibm-2004-3_02_1-k95.cnf	UNSAT	61780	211666	0:00.64	0:00.52	0:00.40
18	ibm-2004-3_02_3-k95.cnf	SAT	73525	269141	0:02.81	0:02.82	0:01.50
19	ibm-2004-3_11-k60.cnf	UNSAT	194649	751312	8:58.76	8:32.21	3:02.52
20	ibm-2004-6_02_3-k100.cnf	UNSAT	80370	293162	0:03.34	0:02.77	0:01.73

Execution times (in seconds) of three solvers for all benchmarks for number of cores = 2



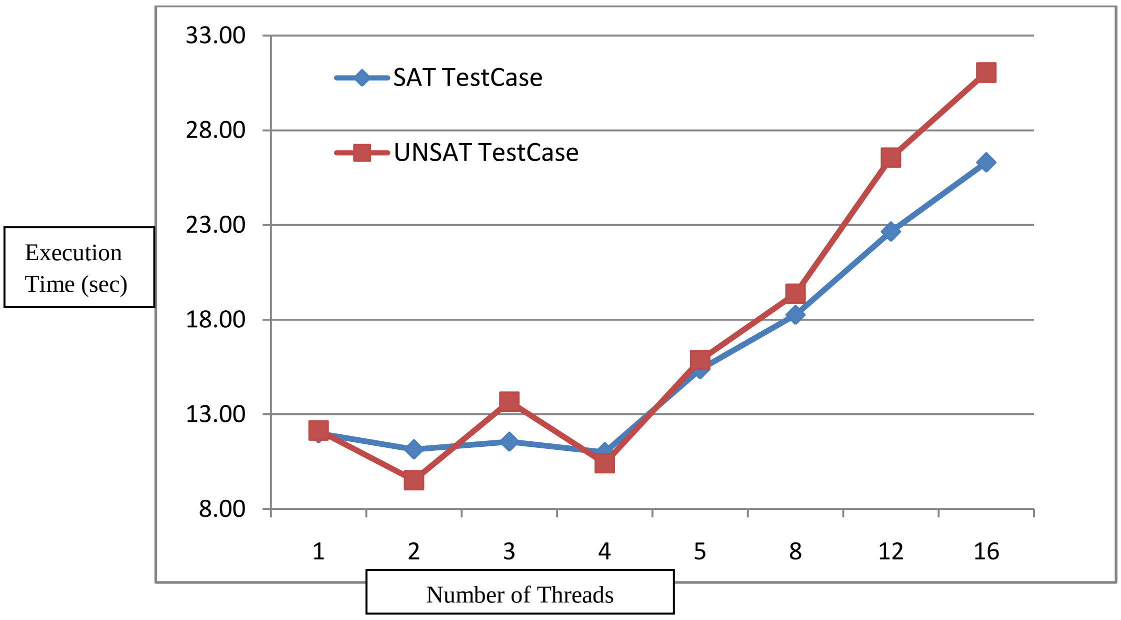
Execution Times of all 3 solvers for all benchmarks when number of threads=**2** for TBBSat and Manysat.



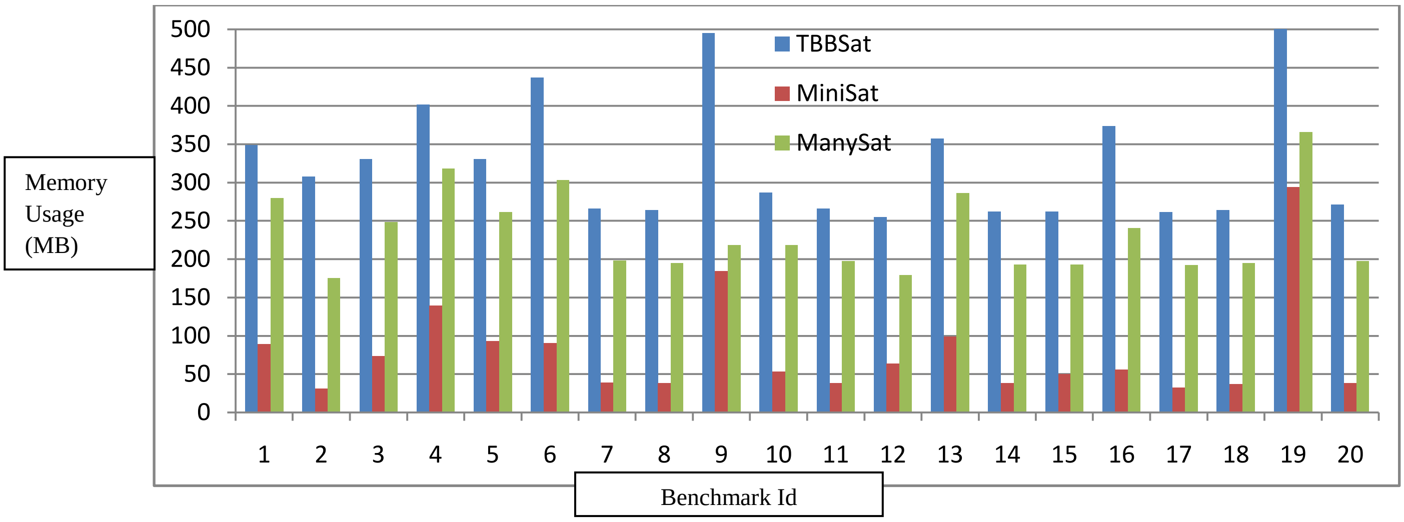
Execution Times of all 3 solvers for all benchmarks when number of threads=**4** for TBBSat and Manysat.

TBB_Sat different runs		**1 thread**	**2 thread**	**3 thread**	**4 thread**	**5 thread**	**8 thread**	**12 thread**	**16 thread**
ibm-2002-05r-k90.cnf	SAT	0:11.99	0:11.14	0:11.55	0:10.99	0:15.37	0:18.25	0:22.64	0:26.29
ibm-2002-07r-k100.cnf	UNSAT	0:02.71	0:02.25	0:03.11	0:02.48	0:03.39	0:03.93	0:04.84	0:05.61
ibm-2002-11r1-k45.cnf	SAT	0:31.60	0:33.62	0:37.02	0:38.70	0:43.34	0:51.67	1:08.80	1:20.07
ibm-2002-19r-k100.cnf	SAT	1:14.80	1:17.39	1:25.55	1:17.48	1:30.51	1:58.11	2:33.16	3:04.71
ibm-2002-21r-k95.cnf	SAT	0:33.19	0:35.64	0:38.22	0:35.70	0:46.32	0:56.72	1:07.13	1:23.31
ibm-2002-26r-k45.cnf	UNSAT	0:01.71	0:02.07	0:02.45	0:03.23	0:03.23	0:04.40	0:06.21	0:08.23
ibm-2002-27r-k95.cnf	SAT	0:05.09	0:04.25	0:05.98	0:02.61	0:07.18	0:09.07	0:10.99	0:13.84
ibm-2004-03-k70.cnf	SAT	0:05.35	0:05.59	0:06.13	0:06.91	0:07.17	0:08.62	0:11.22	0:13.60
ibm-2004-04-	SAT	2:54.64	3:13.29	3:18.25	3:26.02	3:44.82	4:35.84	5:01.58	5:28.10

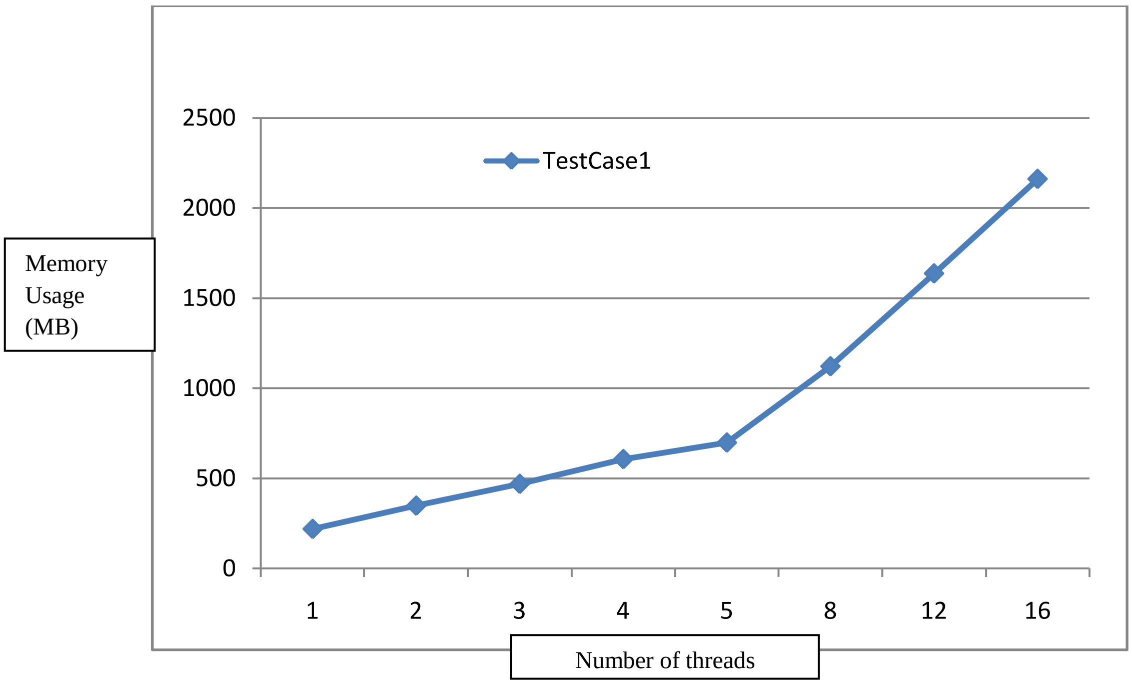
k100.cnf									
ibm-2004-06-k90.cnf	SAT	0:10.56	0:11.29	0:12.15	0:12.89	0:14.10	0:18.87	0:21.97	0:27.10
ibm-2004-1_11-k25.cnf	UNSAT	0:02.47	0:02.75	0:03.03	0:03.34	0:03.67	0:03.96	0:07.26	0:08.08
ibm-2004-1_31_2-k25.cnf	UNSAT	0:46.48	0:49.49	0:50.15	0:52.05	0:53.45	1:03.55	1:25.41	1:26.85
ibm-2004-19-k90.cnf	SAT	0:40.87	0:42.97	0:47.29	0:45.17	0:55.64	1:05.78	1:32.25	1:48.50
ibm-2004-2_02_1-k100.cnf	UNSAT	0:06.03	0:06.68	0:07.15	0:07.41	0:08.20	0:09.79	0:10.28	0:15.69
ibm-2004-2_14-k45.cnf	UNSAT	0:12.15	0:09.52	0:13.67	0:10.42	0:15.86	0:19.37	0:26.55	0:31.04
ibm-2004-26-k25.cnf	UNSAT	0:00.49	0:00.50	0:00.66	0:00.81	0:01.02	0:01.57	0:02.48	0:03.26
ibm-2004-3_02_1-k95.cnf	UNSAT	0:00.51	0:00.64	0:00.75	0:00.85	0:00.97	0:01.29	0:01.78	0:02.40
ibm-2004-3_02_3-k95.cnf	SAT	0:02.67	0:02.81	0:03.16	0:01.69	0:03.90	0:04.87	0:06.15	0:07.96
ibm-2004-3_11-k60.cnf	UNSAT	8:20.76	8:58.76	7:36.76	6:03.06	9:42.03	10:41.50	12:26.37	15:39.17
ibm-2004-6_02_3-k100.cnf	UNSAT	0:03.00	0:03.34	0:03.58	0:03.50	0:04.35	0:05.60	0:06.72	0:08.85



Performance of TBBSat on two test cases for different number of threads.



Memory utilization for 3 solvers for all benchmarks.



Memory utilization by TBBSat for different number of threads.

From the table and graphs, some of key observations are as follows:

The execution time of TBBSat is comparable to Minisat and in several cases, it outperforms Minisat. However Manysat performs better than our implementation. Manysat is based on OpenMP architecture and performs several optimization techniques which boost its performance significantly.

For smaller size benchmarks, TBBSat performs relatively better than Minisat for a fixed number of threads. For larger size, because of tremendous load on memory resources, TBBSat is not able to give the required performance.

When the number of threads is increased monotonically from 2 to 16, it is seen that the performance of TBBSat goes down. This is due to the fact that as the number of cores increase, cache contention becomes common and there is huge overhead in thread scheduling and initialization.

Memory usage: The memory usage for TBBSat is significantly huge when compared to other solvers. This is again due to the entire replication of solver core, thus incurring large memory resource requirement. It increases even more when we share clauses across all threads.

Determination of number of threads: Ideally the number of threads should be a function of the number of clauses. However we have fixed number of threads for a particular run. It can be provided from the command line.

We believe the reason for performance degradation is due to the fight for simultaneous cache access by worker threads. This results in stalling of the task, thus effectively slowing down the program.

Also there is significant scheduling overhead causing by TBB thread handlers. Since TBB acts on a task level, and not at the thread level, there are additional processes which slow down the system.

Cases when it works well: Usually we get good results with 4 cores. Going beyond 4 cores results in performance degradation.

We believe it could perform well in a scenario where we can specify the nodes or cores we want the threads to run on. However currently TBB has no such feature.

Conclusion

We have examined TBB as a parallel framework for SAT solving domain and successfully implemented an existing solver onto TBB. We believe we were not able to harness the entire potential of TBB. For instance, we could use task-based approach than a thread-based approach *(tbb::task instead of tbb::threads)* so as to make use of the strength of TBB, which lies in parallelizing tasks, not threads. We also have implemented a limited version of clause sharing technique in TBBSat. When implemented comprehensively, the clause sharing could benefit from more number of cores.

Acknowledgements:

We would like to thank our professor Anand Raghunathan for providing us access to a multi-core machine because of which we were able to run all our simulations.

References

[1] “Minisat – An Extensible SAT Solver”, Niklas Een and Niklas Sorensson. – (Source code obtained from their website).

[2] “MANYSAT: A parallel SAT Solver”, Y. Hamadi.