SatX10: A Scalable Plug&Play Parallel SAT Framework
(Tool Presentation)

Bard Bloom, David Grove, Benjamin Herta, Ashish Sabharwal, Horst Samulowitz, and Vijay Saraswat

IBM Watson Research Center, New York, USA
{bardb,bherta,groved,ashish.sabharwal,samulowitz,vsaraswa}@us.ibm.com

Abstract. We propose a framework for SAT researchers to conveniently try out new ideas in the context of parallel SAT solving without the burden of dealing with all the underlying system issues that arise when implementing a massively parallel algorithm. The framework is based on the parallel execution language X10, and allows the parallel solver to easily run on both a single machine with multiple cores and across multiple machines, sharing information such as learned clauses.

1 Introduction

With tremendous progress made in the design of Boolean Satisfiability (SAT) solvers over the past two decades, a wide range of application areas have begun to exploit SAT as a powerful back end for declarative modeling of combinatorial (sub-)problems which are then solved by off-the-shelf or customized SAT solvers. Many interesting problems from areas such as software and hardware verification and design automation often translate into SAT instances with millions of variables and several million constraints. Surprisingly, such large instances are not out of reach of modern SAT solvers. This has led practitioners to push the boundary even further, resulting in plenty of harder instances that current SAT solvers cannot easily tackle.

In order to address this challenge, SAT researchers have looked to exploiting parallelism, especially with the advent of commodity hardware supporting many cores on a single machine, and of clusters with hundreds or even thousands of cores. However, the algorithmic and software engineering expertise required to design a highly efficient SAT solver is very different from that needed to most effectively optimize aspects such as communication between concurrently running solvers or search threads. Most SAT researchers do not possess deep knowledge of concurrency and parallelism issues (message passing, shared memory, locking, deadlocking etc). It is thus no surprise that state-of-the-art parallel SAT solvers often rely on a fairly straightforward combination of diversification and limited knowledge sharing (e.g., very short learned clauses), mainly on a single machine.

The goal of this work is to bridge this gap between SAT and systems expertise. We present a tool called **SatX10**, which provides a convenient plug&play

framework for SAT researchers to try out new ideas in the context of parallel SAT solving, without the burden of dealing with numerous systems issues. `SatX10` is built using the X10 parallel programming language [2]. It allows one to incorporate and run any number of diverse solvers while sharing information using one of various communication methods. The choice of which solvers to run with what parameters is supplied at run-time through a configuration file. The same source code can be compiled to run on one node, across multiple nodes, and on a variety of computer architectures, networks, and operating systems. Thus, the `SatX10` framework allows SAT researchers to focus on the solver design aspect, leaving an optimized parallel implementation and execution to X10.

We demonstrate the capabilities of `SatX10` by incorporating into it four distinct DPLL-based SAT solvers that continuously exchange learned clauses of a specified maximum size while running on single or multiple nodes of a cluster.

The goal of this paper is not to present a state-of-the-art parallel SAT solver. Rather, we discuss the design of `SatX10` and the API that must be implemented by any SAT solver to be included in `SatX10`. The `SatX10` harness is available at `http://x10-lang.org/satx10` under an appropriate open source license.

2 Background

We assume familiarity with the SAT problem and DPLL-based sequential systematic SAT solvers, which essentially are carefully designed enhancements of tree search, in particular *learning clauses* when they infer that a partial truth assignment cannot be extended to a full solution. These solvers typically make thousands of branching decisions per second and infer hundreds to thousands of new clauses per second. Most of the successful parallel SAT solvers are designed to run on a single machine. They exploit *diversification*, by simply launching multiple parameterizations of the same solver or of different solvers, and a very limited amount of *knowledge sharing*, typically through learned clauses. Three prominent examples are: `ManySat` [3], which won the parallel track in the SAT 2009 Competition and is based on different parameterizations of `MiniSat` and clause sharing; `Plingeling` [1], which was a winner in the 2011 SAT Competition (wall-clock category, Application instances) and runs multiple variations of `lingeling` while sharing only unit clauses; and `ppfolio` [4], which was placed first and second in the 2011 Competition and simply runs certain five solvers.

2.1 X10: A Parallelization Framework

X10 [2, 5, 7] is a modern programming language designed specifically for programming multi-core and clustered systems easily. Unlike C++ or Java, where threads and network communications are API calls, in X10, parallel operations are integral to the language itself, making it easy to write a single program that makes full use of the resources available in a cloud, GPUs, or other hardware.

X10 is a high-level language that gets compiled down into C++ or Java. Specifically, it runs on Java 1.6 VMs and Ethernet. When compiled to C++, X10 runs on x86, x86_64, PowerPC, Sun CPUs, and on the BlueGene/P, and Ethernet and Infiniband interconnects (through an MPI implementation). It runs on the Linux, AIX, MacOS, Cygwin operating systems. Note that the *same* source program can be compiled for all these environments.

Importantly for SatX10, an X10 program can use existing libraries (e.g., sequential SAT solvers) written in C++ or Java. This permits us to build a parallel SAT solver by using many sequential SAT solvers (changed in modest ways) and using a small X10 program to launch them and permit communication between them. The X10 runtime handles all the underlying communications, thread scheduling, and other low-level system functions.

X10 follows the APGAS (Asychronous Partitioned Global Address Space) programming model. This model says that there are independent memory spaces available to a program, and the program can move data asynchronously between these memory spaces. In X10, these memory spaces are called *places*. Functions and other executable code in the form of closures can also move across places, to process the data where it resides.

X10's basic concurrency and distribution features most relevant for SAT solver designers include a `finish` block, an `async` block, and an `at` block:

- All activities, including parallel activities, inside of a `finish` block must complete before moving on past the finish block.
- The contents of an `async` block can execute in parallel with anything outside of the `async` block. This lets the programmer take advantage of multiple cores within the same place.
- `at` is a place-shifting operation. Code inside the at block is executed at place `p`, not locally. Any referenced data is automatically copied to `p`.

The X10 language has many other features, such as constrained types, GPU acceleration, atomic blocks, clocked computations, collectives, and others. These are not critical to the creation of SatX10 and will not be discussed here.

3 Building Parallel SAT Solvers with X10

The architecture of `SatX10` is shown in Fig. 1 as a high level schematic diagram. Its main components, discussed below, are designed with the goal of providing a generic way to incorporate a diverse set of SAT solvers and support information sharing across solvers. The basic integration of a SAT solver in `SatX10` requires only minimal changes to the SAT solver code and one additional header file to be created. Each constituent SAT solver is enhanced with an X10 part, which is then used for all parallelization and communication, done transparently by the X10 back end. We assume below that the solvers are written in C++.

A. `SatX10.x10` is the main X10 file that, at runtime, reads in the desired solver configuration from a "`.ppconfig`" file (indicating which solvers to launch with which parameters), sets up solvers at various "places", executes them, and sends a "kill" signal to all other solvers when one solver finishes. This

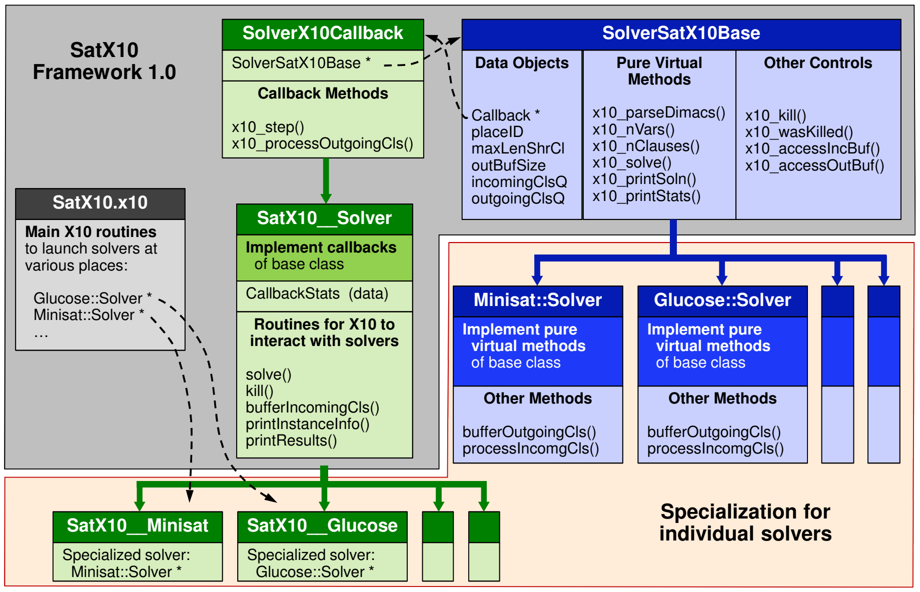

Fig. 1. A schematic diagram depicting the architecture of `SatX10`

file attaches to the solver at each place user-defined X10 "callbacks" that the solver can use to send messages to (or execute methods at) other places, such as sending a newly learned clause. While our current implementation uses a one-to-all, asynchronous communication strategy, X10 provides extensive support for other parallelization schemes such as clocked synchronization and periodic all-to-all information reduction and communication.

B. `SolverSatX10Base.h` provides the base C++ class that every SAT solver's main "solver" class must inherit from and implement some virtual methods for. These simple virtual methods include `x10_solve()`, which solves the instance and returns -1/0/1 based on whether the instance was found to be unsatisfiable/unknown/satisfiable; `x10_printSoln()`, which prints a satisfying assignment; `x10_printStats()`, which prints solver statistics; etc.

C. `SatX10__Solver.h` provides the generic C++ base class that, appropriately extended, acts as the interface between each solver and X10. It provides the implementation of the callback methods solvers use for communication. For each solver, a new header file is built with a solver-specific derived class that has a reference to the main "solver" object. It provides solver-specific routines, such as converting knowledge to be shared from solver-specific data types to the generic data types used in `SatX10__Solver.h` (e.g., converting a learnt clause in the solver's internal format to `std::vector<int>`). The main X10 routine creates at each place an object of such a derived class.

D. *Modifications to solver class*: As mentioned above, the main "solver" class in each SAT solver must inherit from `SolverSatX10Base` and implement its pure virtual methods. The entire code for each solver must also be put inside a unique `namespace` so as to avoid conflicting uses of the same object name across different solvers. Further, the main search routine should be modified to (i) periodically check an indicator variable, `x10_killed`, and abort search if it is set to `true`, (ii) call the appropriate "callback" method whenever it wants to send information to (or execute a method at) another place, (iii) periodically call `x10_step()` to probe the X10 back end for any incoming information (e.g., learned clauses or kill signal) sent by other places, and (iv) implement `x10_processIncomingClauses()`, which incorporates into the solver a list of clauses received from other concurrently running solvers. Note that depending on when in the search process the incoming clause is incorporated, one may need to properly define "watch" literals, possibly backtrack to a safe level, and satisfy other solver specific requirements.

To share other kinds of information, one can define methods similar to `x10_processIncomingClauses` and `x10_bufferOutgoingClauses`. The X10 compiler `x10c++` is used to compile everything into a single executable. The communication back end (shared memory, sockets, etc.) is specified as a compilation options, while solver configurations, number of solvers to run, hostnames across a cluster, etc., are specified conveniently at runtime.

4 Empirical Demonstration

The main objective of this section is to show that `SatX10` provides communication capabilities at a reasonable cost and that it allows effective deployment of a parallel SAT solver on a single machine as well as across multiple machines. As stated earlier, designing a parallel solver that outperforms all existing ones is not the goal of this work.

We used the `SatX10` framework to build a parallel SAT solver `MiMiGlCi`, composed of `Glucose 2.0`, `Cir_Minisat`, `Minisat 2.0`, and `Minisat 2.2.0`, with additional parameterizations (e.g., different restart strategies). As a test bed, we chose 30 instances of medium difficulty from various benchmark families from the application track of SAT Competition 2011 [6]. All experiments were conducted on 2.3 GHz AMD Opteron 2356 machines with two 4-core CPUs and 16 GB memory, InfiniBand network, running Red Hat Linux release 6.2.

The results of the evaluation are summarized in Fig. 2 in the form of the standard "cactus" plot, showing the maximum time (y-axis) need to solve a given number of instances (x-axis).

The two curves in the left-hand-side plot show the comparison on a single machine when `MiMiGlCi` is run on 8 places (i.e., with 8 sequential solvers running in parallel) while sharing clauses of maximum lengths 1 and 8, respectively. Here we see a clear gain in performance when sharing clauses of size up to 8, despite the overhead both on the communication side and for each solver to incorporate additional clauses. In general, what and how much to share must, of course, be carefully balanced out to achieve good performance.

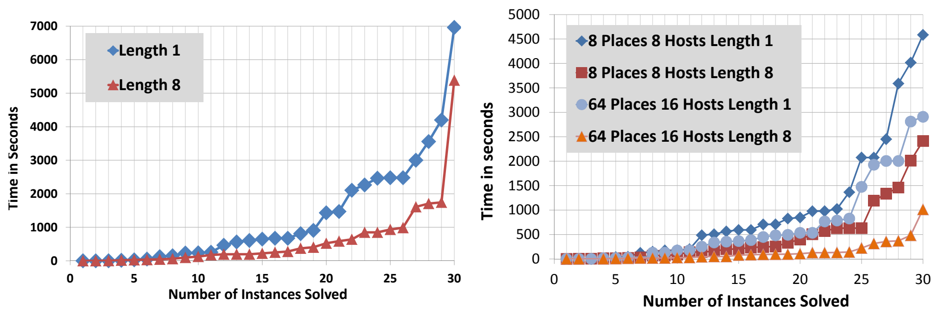

Fig. 2. Performance on a single machine (left) and on multiple machines (right)

The right-hand-side plot in Fig. 2 shows the results on multiple machines. Specifically, we report numbers for 8 places running on 8 different machines, and 64 places running on a total of 16 machines. The benefit of sharing clauses of length up to 8 is clear in this setting as well. In fact, 8 places sharing clauses of length at most 8 performed better than 64 places sharing only unit clauses, indicating that the ability to conveniently share more can be much more powerful than simply running more solvers. Not surprisingly, 64 places sharing clauses of length up to 8 performed the best, solving all instances in 1,000 seconds.

In summary, `SatX10` provides a framework to easily build parallel SAT solvers composed of a diverse set of constituent solvers, with the capability of sharing information while executing various parameterizations of the constituent solvers on a single machine or multiple machines. The rich language of X10 underlying `SatX10` handles all parallelization aspects, including parallel execution, and provides a uniform interface to all constituent SAT solvers. We hope this will serve as a useful tool in pushing the state of the art in parallel SAT solving.

Acknowledgement. `SatX10` originated from an X10-based SAT solver named `PolySat`, which was developed in collaboration with David Cunningham.

References

[1] A. Biere. Lingeling and friends at the sat competition 2011. Technical report, Johannes Kepler University, 2011.
[2] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05*, pp. 519–538, San Diego, CA, USA, 2005.
[3] Y. Hamadi and L. Sais. ManySAT: a parallel SAT solver. *JSAT*, 6, 2009.
[4] O. Roussel. Description of ppfolio. Technical report, Artois University, 2011.
[5] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. Report on the experimental language, X10. Technical report, IBM Research, 2011.
[6] SAT Competition. www.satcompetition.org, 2011.
[7] X10. X10 programming language web site. http://x10-lang.org/, Jan. 2010.