Massive Parallelization of SAT Solvers

[CS262A Project Report]

Nishant Totla
UC Berkeley
nishant@eecs.berkeley.edu

Aditya Devarakonda
UC Berkeley
aditya@eecs.berkeley.edu

ABSTRACT

Boolean Satisfiability (SAT) is a well known NP-complete problem that forms the basis of modern constraint solving. The canonical problem consists of a conjunction of boolean clauses where each clause is a disjunction of literals. The problem is to determine whether there exists a satisfying assignment (and find it if it does). Despite the theoretical hardness of the SAT problem, modern solvers are extremely efficient and sophisticated in terms of the algorithms and heuristics they use. This is especially true for industrial instances generated from practical applications. Even so, further performance gains are harder to produce, while there still exist instances that are unsolvable by modern solvers. Fortunately, this is also a time when the generalization and availability of multicore hardware makes it easier to take advantage of parallel processing capabilities for further speedups. In this paper, we focus on massive parallelization (very large number of processors) of SAT solvers. The goal is to obtain a detailed understanding of the factors that affect performance, communication overheads and efficient ways to encode domain specific knowledge.

General Terms

SAT Solving, Parallelism

Keywords

distributed memory, shared memory, portfolio, divide and conquer, MPI, OpenMP, ManySAT

1. INTRODUCTION

In addition to the traditional hardware and software verification domains, SAT solvers are increasingly popular in newer domains such as computational biology and general theorem proving. This widespread adoption is mainly due to the significant efficiency gains made during the last decade. Modern SAT solvers are capable of handling instances with hundreds of thousands of variables and millions of clauses, and solving them in the order of a few minutes. This is due to both algorithmic improvements in the theory of SAT solving, and better understanding of the structure of instances.

Despite these impressive improvements, new instances consistently challenge the best modern solvers. Each annual SAT Competition [2] sees instances that are unsolvable by any solver within a reasonable amount of time. At the same time, it is becoming harder to discover low level or high level algorithmic adjustments for increasing efficiency. In this scenario, it is not only natural, but crucial that SAT solvers should be extended to parallel and/or distributed hardwares that are now much easier to access and relatively programmer friendly. At the core, SAT solvers perform search over a well defined search space, and there are interesting challenges to be tackled for parallelizing this search.

Several parallel SAT solvers have been proposed in the past few years. A brief survey is presented in Section 7. Broadly, parallel SAT solvers are based on one of the following approaches

1. *Divide and Conquer*: These solvers either divide the search space using certain heuristics, or decompose the formula itself using decomposition techniques. The major challenge with this approach is the difficulty of workload balancing between different processor units. Appropriate search space division is yet another challenge. Sharing information could also get tricky.

2. *Portfolio*: Portfolios take advantage of the main weakness of modern solvers - their sensitivity to parameter tuning. Each processor unit runs a version of the solver with predetermined parameters, working on the full problem instance. This combination of solvers is

designed to represent orthogonal yet complementary strategies. In addition, solvers can exchange learned information to further speed up the search. The main challenge with this approach is ensuring that the portfolio is diverse.

In general, the workload of each processor unit can be visualized as *INSTANCE(Problem instance, Solver, Search space)* - a function of three parameters. Varying the search space results in the divide and conquer approach, and varying the solver results in the portfolio. Any of the three parameters could be varied to obtain a valid parallelization.

The rest of the paper is organized as follows. In Section 2, we describe the main goals that this paper aims for in terms of improvement over state of the art in parallel SAT solving. Section 3 describes some commonly used modern techniques that SAT solvers use to perform search, and heuristics to make it efficient and fast. In Section 4, we present the structure of our parallel SAT solver and some reasoning for our design decisions. Sections 5 and 6 describe the environment for our experiments and our results, respectively. Section 7 presents previous relevant work in the area, and Section 8 concludes with the insights obtained from our experiments and some future work.

2. OBJECTIVES

In this section, we describe the contributions we intend to present with this work. This includes both improvements over existing parallel SAT solvers and a better understanding of how parallel SAT solving could be made to scale more efficiently. In recent competitions, portfolio solvers have achieved much more success than their divide and conquer counterparts. While the idea of portfolios is appealing, most implementations use minimal to no sharing of learned information, and there is little insight (except for some empirical data) on which parameters work best.

Additionally, an important aspect lacking from successful parallel solvers like ManySAT [22] and Plingeling [1] is massive scaling experiments. While massively parallel hardware is easy to access, these solvers use shared memory, which restricts the degree of parallelism that can be attained by restricting them to cores on a single node. This means that little has been explored in the domain of scaling the size of parallel solvers up to a few hundred or thousand cores. There is some work on scaling simple portfolios (without sharing) to massively parallel environments [5, 19], but with limited success. In this paper, we evaluate scaling results for our implementation of a parallel SAT solver intended to scale to hundreds or thousands of cores, using a combination of shared and distributed memory. The implementation is described in detail in Section 4. We evaluate our work based on trying to answer the following questions

1. *Scaling to a large number of cores*: How many cores is it possible to scale our solver to?
2. *Demonstrating parallel scaling*: Does adding more cores make the solver faster?
3. *Speedups over existing solvers*: Can we do better than existing parallel solvers?
4. *Solve intractable instances*: Can we solve instances that other parallel solvers cannot?

3. BACKGROUND

In this section, we briefly describe the high-level algorithms and heuristics which modern SAT solvers use.

3.1 Conjunctive Normal Form

Conjunctive Normal Form (CNF) has been accepted as the canonical way of representing a SAT instance. Any propositional formula can be written in CNF form. A formula in CNF form consists of variables and clauses. Formally, a formula ϕ is in CNF form if it is a conjunction of clauses i.e. $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$. Each clause is a disjunction of literals i.e. $C_i = l_1 \vee l_2 \vee \cdots \vee l_{t_i}$. A literal is simply a variable v or its negation $\neg v$. The formula is satisfiable if there exists an assignment of True/False values to the variables such that all the clauses are satisfied.

3.2 DPLL Search

Most of the state of the art solvers are based on the Davis, Putnam, Logemann and Loveland procedure, commonly known as DPLL [9]. DPLL is a backtracking search procedure. At each node of the search tree, a *decision* literal is chosen according to some branching heuristics. It is assigned one of the possible values (True or False), which is followed by an *inference* step that deduces and propagates some forced literal assignments such as unit and monotone literals. All assignments made by a decision and the resulting propagations are at the same decision level. The decision level is increased with each new decision, increasing until either a satisfiable assignment or conflict is found. In the first case, the solver returns the found assignment, and in the second case, the solver backtracks to the last decision level and reverses the decision at that level (by assigning it the opposite value). The formula is declared to be unsatisfiable when a backtrack to level 0 occurs. While the basic DPLL procedure is simply backtracking search, many improvements and optimizations have been proposed over the years.

3.3 Modern SAT solvers

Modern SAT solvers [16, 14] are based on the classical DPLL search procedure, combined with heuristics such as

1. Restart policies [6, 12] which prevent the search from getting stuck in unsatisfiable parts of the search space at deep decision levels. During a restart, the solver effectively leaves its search and starts afresh, still keeping learned information from the earlier part of the search. Rapid restarts have been shown to be very effective in reducing running times [7].

2. Activity-based variable selection heuristics (e.g. VSIDS) [14] keep track of the frequency of occurrence of variables during the search. This provides more information about variables which are potentially important to the search, thereby helping to intensify the search.

3. Conflict Driven Clause Learning (CDCL) [17] is a key technique which generates additional clauses whenever the search reaches a conflict. Clauses are crucial in helping to prune the search space so that similar conflicts are not repeated, and the solver does not delve into parts of the search space that have caused a conflict before. Learned clauses could also trigger additional unit propagations, and help in non-chronological backtracking.

4. Progress saving [8], which can be seen as a partial component caching technique, saves polarities of variables assigned between two decision levels. Similar polarities are used in a subsequent search, which avoids solving some components multiple times.

Efficient data structures are implemented to manage and update information for these heuristics. Modern SAT solvers are especially efficient with "structured" SAT instances coming from industrial applications. There are several other heuristics that have been tried, many of which try to optimize the solver to solve problems from a specific domain.

4. SOLVER DESCRIPTION

In this section, we describe the structure of our parallel SAT solver. Our solver implements a combined divide and conquer plus portfolio approach. At the high level, our solver is a divide and conquer solver. The search space is divided into multiple disjoint parts and a separate process is launched to handle each such part. Each process, however, is a parallel portfolio of multiple threads running independent solvers to solve the assigned part of the search space. Threads in a single process are implemented using OpenMP, so they use shared memory. This makes sharing learned clauses within a single process fast. Processes, on the other hand, communicate using message passing (implemented using MPI). The number of processes as well as the number of threads per process can both be varied to tune for optimal performance.

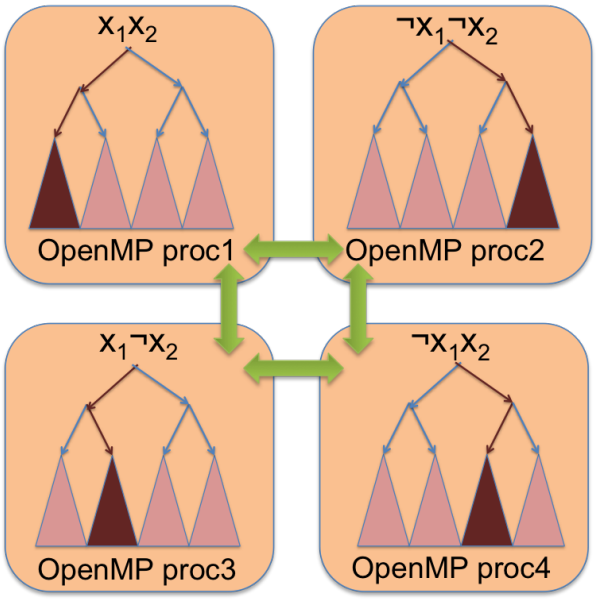

Figure 1: Visualization of the divide and conquer guiding paths approach.

This method of parallelization does not require large portfolios, since the size of the portfolio is restricted to the number of cores in a node (in our experiments, this is 24. See Section 5). Scaling is dependent on division of search space, which is performed using the notion of *guiding paths* [11, 20, 10]. Given a problem instance ϕ, a guiding path $\mathcal{G}$ is a set of constraints created by forcing True/False values on a small set of variables of ϕ. This results in the smaller (more constrained) search space defined by $\phi \wedge \mathcal{G}$. An exhaustive set of guiding paths $\mathcal{G}_1, \mathcal{G}_2, \cdots, \mathcal{G}_t$ is created when the solver is started, and the corresponding search spaces are assigned to each process. Figure 1 illustrates our design with 4 processes created using guiding paths of length 2, using variables x_1 and x_2.

The choice of guiding paths is crucial to solver performance. Picking arbitrary variables will divide the search space, but might provide negligible speedups in performance. An important parameter to optimize the solver is to consider better guiding paths - ones that contain "important" variables. We currently pick guiding paths based on the first few decisions made by the VSIDS heuristic of the solver. A potentially more optimal way could be to obtain information about variables from the high-level domain (such as syn-

thetic biology, model checking etc.) from which the problem instance was generated.

We note that parallelizing in this way enables us to scale our solver to a large number of cores. An alternative would be to scale the portfolio directly, having a large number of parallel instances of the solver that operate on the same instance [5]. Since solver parameters are limited in number, creating a large portfolio needs additional heuristics which are not readily evident. In addition, communication could become a bottleneck at a large scale, since it would have to be done using message passing while ensuring that solvers are not burdened with the very large number of learned clauses that would be generated. The advantages of scaling would vanish beyond a point. On the other hand, having a fully divide and conquer solver would just allow us to use longer guiding paths (but not much longer), and this could fail if the wrong paths were chosen, taking a very long time on spaces that need to be explored to greater depths.

5. EXPERIMENTAL SETUP

Our hybrid parallelization approach is well-suited for architectures with multiple processors per node and several nodes executing in parallel. This two-level approach where OpenMP threads run within a node (shared memory parallelism) and MPI processors run across nodes (distributed memory parallelism) gives us great flexibility across many different architectures. In this paper, we provide experimental data collected on NERSC Hopper.

Hopper[1] is a Cray XE6 system with 153,216 processors, 217 Terabytes of memory, 2 Petabytes of disk storage and has a peak performance of 1.28 Petaflops/sec. Each node of Hopper contains 24 AMD MagnyCours processors and 32 GB of DDR3 memory – some Hopper nodes have 64 GB of DDR3 memory to support applications with high memory requirements. Each compute node has the processor-memory layout shown in Figure 2. The compute node contains 2 MagnyCours chips with 12 cores each, each chip is further divided into 2 dies with 6 cores.

Communication between the cores is supported through a HyperTransport 3.0 bidirectional interconnect. It should be noted that interconnect latencies and bandwidths differ for the on-chip interconnect (between two dies on the same chip) and the off-chip interconnect (between two dies on different chips). The memory layout is also non-uniform; each die has 2 sockets to memory and as a result data layout and data locality play an important role in performance tuning. For our experiments we do not attempt to tune the SAT solver

[1]http://www.nersc.gov/users/computational-systems/hopper/

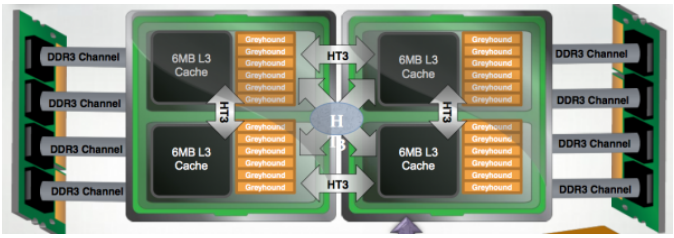

Figure 2: NERSC Hopper hardware layout

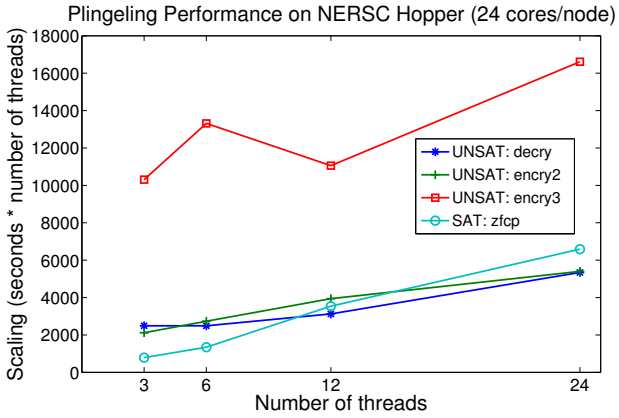

Figure 3: Strong scaling of Plingeling on several SAT/UNSAT instances

for efficient data access. We take a more high-level approach and simply vary the number of MPI processes per node and OpenMP threads per process to see whether NUMA effects have a non-negligible effect on performance.

6. PERFORMANCE EVALUATION

In this section, we evaluate the performance of existing SAT solvers and compare the best existing solver to our distributed memory parallel version.

6.1 Performance of Existing Solvers

We begin by first comparing the scaling of two existing SAT solvers, Plingeling and ManySAT 2.0. This data was crucial in selecting a good solver to modify and extend to distributed memory, parallel machines. Ideally we would have written an optimized SAT solver for distributed memory. However that would have been a massive software engineering effort due to the complexity of SAT solvers. Therefore, we decided to extend one of the existing SAT solvers written to take advantage of shared memory parallelism.

Figures 3, 4 show strong scaling performance of Plingeling and ManySAT, respectively, for 3 UNSAT instances and 1 SAT instance. These results were used to determine which SAT solver to modify for our MPI-based, distributed mem-

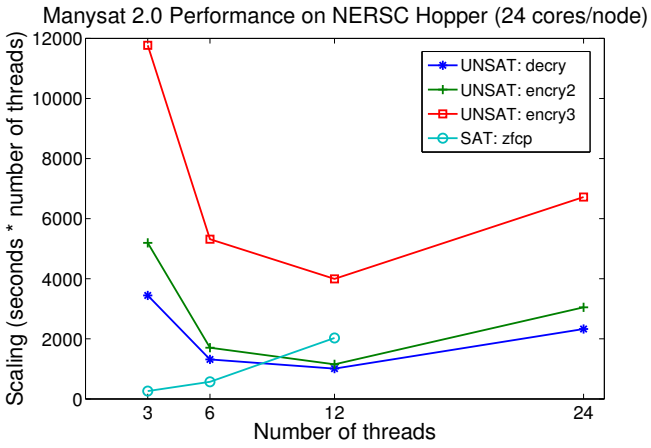

Figure 4: Strong scaling of ManySat on several SAT/UNSAT instances

ory SAT solver. Before discussing the performance of Plingeling and ManySAT, we briefly describe how to interpret Figures 3 and 4. The y-axes (scaling factor) represents how well each solver scales across increasing number of threads used for each dataset. The scaling factor[2] is in units of (seconds) $\times$ (number of threads), which means that a line with zero slope shows perfect, linear scaling, positive slopes represent poor scaling and negative slopes represent super linear scaling.

Figure 3 shows the performance of Plingeling for each SAT/ UNSAT instance on a single node of Hopper while varying the number of threads used by the solver. We see that Plingeling doesn't scale very well for any instance, except for the Encry3 dataset while running with 12 threads. Encry3 is a large UNSAT instance and the speedup at 12 threads might be attributed to having a large enough solver portfolio where clause sharing leads to rapid convergence of the solver. Beyond 12 threads we see that communication overhead of clause sharing causes a slowdown. In general, Figure 3 suggests that Plingeling does not scale very well on a single node of Hopper and may not be the best solver to use for our distributed memory implementation.

Figure 4 shows the strong scaling performance of ManySAT and we see that this solver performs much better on Hopper. In fact, ManySAT exhibits super linear scaling for a certain range of threads on all of the UNSAT instances. Once again, we believe that this is due to having a diverse portfolio where clause sharing leads to faster convergence. Beyond 12 threads the communication overhead of clause sharing introduces inefficiencies and results in poor scaling at 24 threads. ManySAT performs just as poorly for the SAT instance between 3 and 12 threads and we were unable to run with 24 threads due to memory constraints. Plingeling was able to scale to 24 threads primarily because it uses a soft memory limit in the software where threads will not be created when current memory usage grows beyond 50% total node memory – ManySAT does not implement this soft limit. Based on these results we chose to modify ManySAT for our distributed memory implementation.

[2]The magnitude in the y-direction is not an important measure. The trend (slope) across the number of threads is key.

6.2 MPIManySAT Performance

In this section we compare the performance of MPIManySAT to the performance of ManySAT and gauge whether our distributed memory implementation has advantages over existing shared memory implementations. Before describing our results we would like to briefly describe a feature provided by ManySAT. When launching ManySAT the user can specify whether the solver runs in deterministic mode or non-deterministic mode. These two modes have a large impact on running times, therefore we evaluate the performance of ManySAT and MPIManySAT under both modes.

The primary difference between the two solver modes is that the deterministic setting utilizes OpenMP barriers to synchronize thread communication (during clause sharing) whereas the non-deterministic setting does not. In the non-deterministic mode, threads can potentially miss important clauses that may lead to faster convergence. Conversely, a thread making fast progress towards the solution can easily reach it without unnecessary OpenMP barrier synchronizations. For the non-deterministic mode we repeat the runs three times and plot all three running times, in order to show the variability in running times.

We collected running time data for ManySAT and MPI-ManySAT on Hopper using two datasets: an easy SAT instance and an easy UNSAT instance. Figures 5 and 6 show the running time performance on the easy SAT and easy UNSAT instances, respectively, for both solvers. The figures show results from both solvers and use the following convention: the left half of the x-axis (from 3 to 24 threads) show results for ManySAT on a single node of Hopper and the right half of the x-axis (from 192 to 768 processors) show results for MPIManySAT on several nodes of Hopper. For MPIManySAT we launch 2 MPI processes per node and 12 OpenMP threads per MPI process.

Deterministic ManySAT performance in Figure 5 suggests that running times are quite variable as we scale up the number of threads. In particular, there is a slowdown between 3 threads and 6 threads which we believe is attributable to the fact that the portfolio is not diverse enough to converge

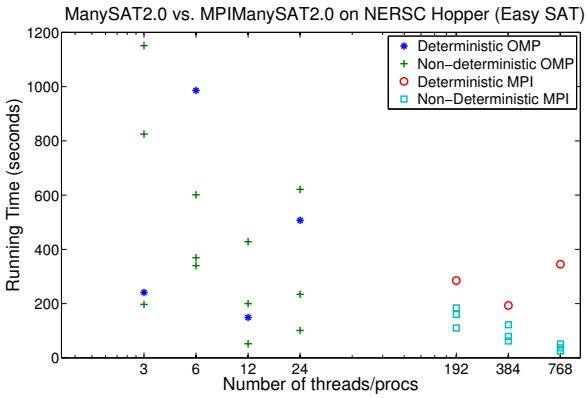

Figure 5: Comparison of ManySAT and MPI-ManySAT on an easy SAT instance

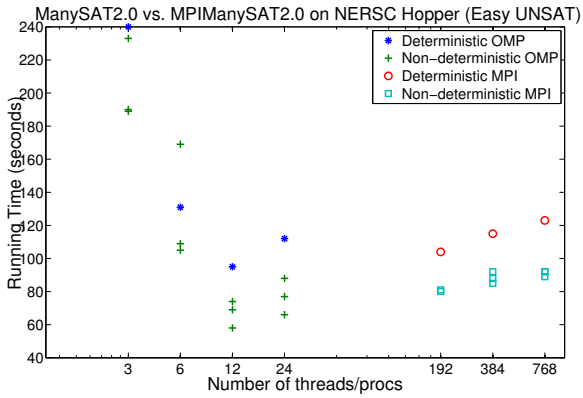

Figure 6: Comparison of ManySAT and MPI-ManySAT on an easy UNSAT instance

faster than 3 threads and we pay for the OpenMP barrier synchronization. However, at 12 threads the portfolio becomes diverse enough that one or more of the threads are on promising search paths and the clause learning (synchronization) helps. Finally, at 24 threads we see synchronization overhead dominating any gains from portfolio diversity. In the non-deterministic case for ManySAT we see large variability in running times, across all thread ranges. Even though we collect only 3 data points for non-deterministic ManySAT, the running times have large variability. It is clear that non-determinism could potentially solve the problem faster. But without luck, the problem could see large slowdowns. The primary reason for the large slowdowns is the fact that threads might miss important synchronization events where learned clauses can accelerate convergence.

Deterministic MPIManySAT performance in Figure 5 exhibits much more predictable scaling behavior where we see a speed up from 192 cores to 384 cores but then slowdown at 768 cores due to too much synchronization overhead. Note that because our dataset is an easy SAT instance it is not entirely necessary to use so many cores, as a result the MPIManySAT running times are slightly slower than the ManySAT running times. The most interesting results are the running times for non-deterministic MPIManySAT. We see that the running times are clustered much more closely than ManySAT. This result should be expected given that the MPI version uses a divide and conquer strategy based on guiding paths that divides the search space among the MPI processes. Due to the search space division, clause sharing between MPI processes can be eliminated since the search space is different for each MPI process. In addition, if the SAT instance has several satisfiable assignments then multiple MPI processes can reach such an assignment thus leading to a race-to-halt condition. This is precisely why the non-deterministic running times for the MPI version are clustered closer together and faster than the running times for non-deterministic and deterministic ManySAT.

Figure 6 shows running time performance of ManySAT and MPIManySAT for an easy UNSAT instance. Deterministic ManySAT scales well from 3 threads to 12 threads and the reason for this is because UNSAT instances require exploration of the entire search space. Therefore, clause sharing has a lower overhead in comparison to the time taken to explore the search space. However, it should be noted that eliminating the synchronization overhead is, for the most part, much faster in the non-deterministic case. The reason for the speed up without synchronization is because this is an UNSAT instance.

MPIManySAT does not scale very well for the UNSAT instance. This is due to the MPI startup overhead – note the linear increase in running time – and because the UNSAT instance is easy to solve. Another difference between the MPI performance on the SAT instance and the UNSAT instance is that the MPI version can exit immediately after one process finds a satisfiable solution. However, in the UNSAT case we must wait until all MPI processes have finished before concluding that the problem is unsatisfiable. As before, the non-deterministic version of MPIManySAT shows much closer clustering than non-deterministic ManySAT.

From these experiments we can conclude that MPIManySAT performs well on the SAT instance in comparison to ManySAT. We are less successful on the UNSAT instance but this is due to the fact that the both instances are easy to solve. Even with this drawback, our solver is much more reliable under the non-deterministic case than ManySAT. This suggests that our solver might be able to solve hard SAT instances

Hard SAT (deterministic, MPIManySAT)	
Number of cores	Running Time (sec)
384	1984.0
768	511.0
Hard SAT (non-deterministic, ManySAT)	
Number of threads	Running Time (sec)
12	2067.0
24	1730.0
Speedup: 3.3× (24 threads vs. 768 cores)	

Table 1: Running time results for MPIManySAT and ManySAT for a hard SAT instance.

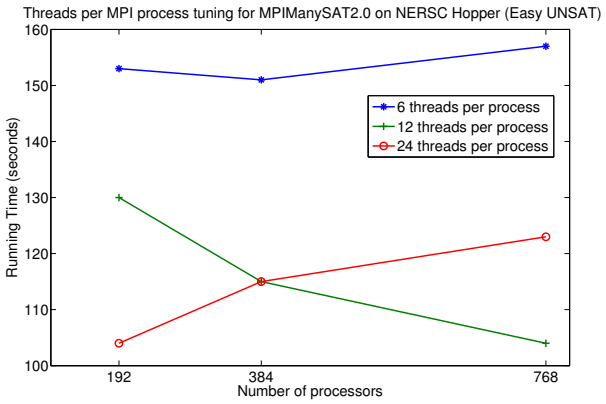

Figure 7: OpenMP threads per MPI process tuning of MPIManySAT for an easy UNSAT instance

that ManySAT may not be able to solve. Being able to do so would, in and of itself, be a success.

6.3 MPIManySAT Tuning

Our hybrid parallelization approach offers us the opportunity to tune the performance of our SAT solver by varying the number of OpenMP threads used by each MPI process. Optimizing the number of threads per MPI process should allow us to better utilize the hardware by making efficient use of NUMA by mapping one MPI process to each NUMA region. As described in Section 5, Hopper has 4 NUMA regions with 6 cores each. We experiment with different MPI process to threads ratios, beginning with launching one MPI process per NUMA region (with 6 OpenMP threads each) and ending with one MPI process per node (with 24 OpenMP threads). Note that a tradeoff exists as we begin to increase the number of MPI processes per node. If each node runs multiple MPI processes then our guiding paths can be longer. For example, two MPI processes on a single node of Hopper can divide the search space using a single variable. However, four MPI processes on a single node of Hopper can divide the search space using two variables. Naturally, this is an important tradeoff to explore because striking the right balance between search space division, efficient hardware use and portfolio diversity is critical.

Figure 7 shows the running time of MPIManySAT for 3 processor ranges and varies the number of OpenMP threads assigned to each MPI process from 6 threads per process to 24 threads per process. We use an easy UNSAT instance so that the race-to-halt condition for SAT instances does not affect the experiments. We also run the experiments under deterministic mode so that reproducibility is guaranteed.

Figure 7 suggests that mapping 6 threads to each MPI process is inefficient. We believe that this is due to insufficient diversity in the solver portfolio which leads to inefficient exploration of the search space assigned to each MPI process. With 6 threads per MPI process the solver can use longer guiding paths, however, the lack of diversity leads to inefficient exploration of the assigned search space. Assigning 12 threads to each MPI process seems to perform much better and lead to speedup as we scale the number of processors. At 24 threads per MPI process we experience a slowdown as we scale the number of processors, however, we find that at 192 cores we perform just as well as 12 threads at 768 cores. These two points suggest that longer guiding paths are an important factor for scalability but not an important factor for running time (at least for this easy UNSAT instance). At 24 threads and 192 processors we have a large solver portfolio but short guiding paths whereas at 12 threads and 768 processors we have a smaller solver portfolio but longer guiding paths. Clearly, this shows that a tradeoff exists between the diversity of the portfolio and the length of the guiding paths.

We can also draw conclusions regarding the importance of variables chosen for the guiding paths. Comparing the data points for 12 threads, 192 cores and 24 threads, 192 cores the main difference is that at 12 threads per process we choose a fourth variable for the guiding path. Based on the running times it is clear that the fourth variable is not as important given that we do not see a speedup after choosing it for the guiding path. This means that the heuristic for choosing a variable for the guiding path is just as important as the diversity of the portfolio.

6.4 Hard SAT Instance Performance

One of our main metrics of success is to be able to solve hard SAT/UNSAT instances that existing solvers cannot currently solve. We use deterministic ManySAT as our point of comparison due to the running time reproducibility guar-

anteed under deterministic solver mode. We also ran MPI-ManySAT under deterministic mode as a fair comparison, with the only difference being the search space division for the MPI version.

Table 1 shows the results of our experiments. We chose to limit running times to a maximum of 90 minutes, after which we killed the solver and called that run a failure. We believe that this is a reasonable limit to place given that allocations on supercomputing resources are finite and conservation is a legitimate concern. Under these limits, we found that MPIManySAT was able to successfully solve the hard SAT instances for 384 cores and 768 cores. Interestingly, the MPI version shows super linear speedup (a 1.9× speedup) between 384 cores and 768 cores. The main difference between the two runs is that at 768 cores we subdivide the search space using another variable and that variable is key to reaching the solution faster. This acceleration is quite promising because it suggests that selecting the "right" variables to prune the search space can have a large impact on time to solution.

In comparison, ManySAT did not perform well while running in deterministic mode. ManySAT was unable to finish within the allotted 90 minutes for any number of threads. This shows that search space division is an important component in being able to solve hard SAT instances. However, to be completely fair we re-ran the hard SAT instance with ManySAT running in non-deterministic mode. We replicated the run 3 times to get a fair spread of the running time and found that most runs did not finish in 90 mins. However, table 1 shows the running times of the runs that did finish (we only show the best running times in the table). Even then, our MPI version (running in deterministic mode) performs much better than the best non-deterministic ManySAT run and achieves a speedup of 3.3×. Once again, we stress that MPIManySAT was able to finish deterministically while ManySAT was not able to do so. It should also be noted that we can reasonably expect MPIManySAT to perform even better under non-deterministic mode given the results in Figures 5 and 6.

7. RELATED WORK

We present some noticeable related work in the area of parallel SAT solving.

ManySAT [22] was the first portfolio solver, that created portfolios based on varying parameters such as restart policies, randomness in the decision heuristic and variable polarity heuristic. It is built on top of MiniSAT [16] with a shared-memory model using OpenMP. The communication between the solvers is organized through lockless queues which contain all the learned clauses that a particular core wants to share. Since it uses MiniSAT, each core runs a CDCL-based DPLL solver that broadcasts all learned clauses of up to a fixed length (this length is determined empirically). Our solver is built on top of ManySAT.

Plingeling [1] is another successful portfolio based parallel solver that uses the POSIX pthreads library for parallelization. It is built on top of Lingeling [1]. While there are differences in the approaches and heuristics used compared to ManySAT, the fundamental architecture and design is similar.

PSATO [11] is based on the SATO (SAtisfiability Testing Optimized) sequential solver. Like SATO, it uses a *trie* data structure to represent clauses. Like our solver, PSATO uses the notion of *guiding-paths* to divide the search space of the problem. Although in this case, the exploration is organized in a master/slave model. The master assigns guiding paths to workers, and performs load balancing. Gradsat [20] is a parallel solver based on zChaff, and is similar in structure to PSATO, except that it also allows for sharing of learned clauses between slave workers. A client incorporates a foreign clause when it backtracks to level 1 (top level). In [21], the architecture is similar to Gradsat, but a client incorporates a foreign clause if it is not subsumed by the current guiding-path constraints. This approach is supposed to scale well on computational grids.

PSatz [4] is the parallel version of the Satz solver, and uses a dynamic load-balancing mechanism based on work-stealing techniques. This is similar to PSATO, without the presence of a master. Several other parallel solvers [18, 3, 13, 15] use ideas similar to PSATO, PSatz and Gradsat, along with some optimizations and heuristics of their own.

pMiniSAT [10] is a divide and conquer based parallel SAT solver that uses guiding paths. However, it additionally exploits the knowledge on these paths to improve clause sharing. Clauses can be large when constrained by the guiding path, but when considered with the knowledge of the guiding path of a particular thread, the clause can be shortened and therefore made smaller and more useful.

Clearly, there has been a lot of effort in trying to parallelize SAT solving. In practice, divide and conquer approaches have performed poorly compared to their portfolio counterparts, due to load balancing issues and difficulty of sharing learned clauses. On the other hand, divide and conquer approaches make it easy to distribute search and scale it up to a large number of cores. Portfolios need to be carefully crafted and are not readily scalable. There have been di-

vide and conquer solvers written for computational grids, but few attempts to massively parallelize portfolios. One such attempt [5] shows some gains with a simple portfolio that does not perform clause sharing. In general, both approaches have their merits, and our solver is an attempt to bring the two together and obtain further improvements.

8. CONCLUSIONS

We have developed a distributed memory, parallel SAT solver build on top of ManySAT that can scale to hundreds of cores through the use of a divide and conquer strategy based on guiding paths. We have shown that our MPI version can deterministically solve hard SAT instances while existing shared memory, parallel SAT solvers cannot under a reasonable time constraint. Building our MPI solver on top of ManySAT has enabled us to develop a hybrid, parallel code which can use MPI processes for parallelism across nodes and OpenMP threads for parallelism with a node. Hybrid parallelization provides greater flexibility when attempting to efficiently map software to hardware. In particular, we feel that this hybrid approach is especially useful for NUMA architectures where we can launch one MPI process per NUMA region and several threads within a NUMA region. However, using ManySAT as our base solver has its drawbacks. The biggest drawback is that ManySAT uses an object-oriented approach which makes it difficult to write MPI code that supports complex communication patterns (other than the current divide and conquer solution) due to the considerable software rearchitecting needed to make the ManySAT data structures MPI-friendly.

In addition to the performance tuning opportunities and more efficient software-hardware mapping, there are other application-level optimizations that can potentially make our solver much faster than existing SAT solvers. In Section 6 we show that choice of variables used for search space division can be critical to faster solvers. In fact, we saw a super linear speedup just by splitting the search space further by one variable. Our current heuristic is rather simplistic in that we run one instance of ManySAT and extract the first few variables that the instance makes decisions on. Naturally, this guess is based upon the VSIDS heuristic within ManySAT and may not be good enough for search space splitting at the higher level of guiding paths. One approach would be to fix the problem domain from which the SAT/UNSAT datasets are derived from and specialize the variable selection heuristic for that domain.

Good choices of variables will enable the solver to quickly prune out uninteresting search paths, however, this has the disadvantage of causing load imbalance between the processors. In fact, the divide and conquer strategy will cause processors to go idle whenever the assigned search spaces are easily eliminated. In these situations, it would be much more effective to employ a work-stealing strategy where exiting MPI processes replicate a running process and make additional decisions to subdivide the running process' search space. The work stealing optimization should further reduce running times since all processors will be kept active and deeper levels of the search tree will continue to be subdivided.

Many of the existing SAT solvers use randomness to diversify the solver portfolio. However, the degree of diversification is limited by the number of processors in a shared memory system. Our MPI version is not limited by this and can launch a large number of solvers each with different startup parameters (random seeds, restart frequency, clause sharing ratio, etc). These are all optimizations that existing SAT solvers cannot implement or take advantage of due to their limited ability to scale. Although, we have not implemented these optimizations we believe that our distributed memory parallel SAT solver has the potential to efficiently and quickly solve hard SAT/UNSAT problems.

In this paper, we have implemented a distributed memory SAT solver that can scale to hundreds of cores and do so efficiently with a divide and conquer strategy based on guiding paths. Our solver was able to successfully solve a hard SAT instance deterministically while ManySAT failed under the same conditions. We were also able to attain a speedup of 3.3× over non-deterministic ManySAT on the same hard SAT instance. There are still many challenges to address and optimizations to explore, however, we have shown that a distributed memory SAT solver is feasible and necessary for hard SAT/UNSAT instances where existing shared memory SAT solvers fail.

9. ACKNOWLEDGEMENTS

The authors would like to thank James Demmel, Anthony Joseph, John Kubiatowicz and Sanjit Seshia for their helpful comments and suggestions throughout the semester. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

10. REFERENCES

[1] Lingeling, Plingeling and Treengeling. `http://fmv.jku.at/lingeling/`.

[2] The International SAT Competitions web page. `http://www.satcompetition.org/`.

[3] Alberto Maria Segre and Sean L. Forman and Giovanni Resta and Andrew Wildenberg. Nagging: A

scalable fault-tolerant paradigm for distributed search. *Artif. Intell.*, 140(1/2):71–106, 2002.

[4] Bernard Jurkowiak and Chu Min Li and Gil Utard. A Parallelization Scheme based on Work Stealing for a class of SAT Solvers. *Journal of Automated Reasoning*, 34, 2005.

[5] Bordeaux, Lucas and Hamadi, Youssef and Samulowitz, Horst. Experiments with Massively Parallel Constraint Solving. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, pages 443–448, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[6] Carla P. Gomes. Boosting combinatorial search through randomization. pages 431–437. AAAI Press, 1998.

[7] Carla P. Gomes and Bart Selman and Nuno Crato and Henry Kautz. Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems. *Journal of automated reasoning*, 24:2000, 2000.

[8] Daniel Frost and Rina Dechter. In Search of the Best Constraint Satisfaction Search. In *In Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 301–306, 1994.

[9] Davis, Martin and Logemann, George and Loveland, Donald. A Machine Program for Theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[10] Geoffrey Chu and Peter J. Stuckey and Aaron Harwood. PMiniSAT: A Parallelization of MiniSAT 2.0. Technical report, 2008.

[11] Hantao Zhang and Maria Paola Bonacina and Maria Paola and Bonacina and Jieh Hsiang. PSATO: a Distributed Propositional Prover and Its Application to Quasigroup Problems. *Journal of Symbolic Computation*, 21:543–560, 1996.

[12] Kautz, Henry and Horvitz, Eric and Ruan, Yongshao and Gomes, Carla and Selman, Bart. Dynamic Restart Policies. In *Eighteenth National Conference on Artificial Intelligence*, pages 674–681, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[13] Luis Gil and Paulo Flores and Luis Miguel Silveira. PMSat: a parallel version of minisat. *Journal on Satisfiability, Boolean Modeling and Computation*, page 2008.

[14] Matthew W. Moskewicz and Conor F. Madigan and Ying Zhao and Lintao Zhang and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Annual ACM IEEE Design Automation Conference*, pages 530–535. ACM, 2001.

[15] Max Böhm and Ewald Speckenmeyer. A Fast Parallel SAT-Solver - Efficient Workload Balancing. In *Annals of Mathematics and Artificial Intelligence*, pages 40–0, 1996.

[16] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *SAT*, pages 502–518, 2003.

[17] Silva, João P. Marques and Sakallah, Karem A. GRASP - a New Search Algorithm for Satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[18] Tobias Schubert and Matthew Lewis and Natalia Kalinnik and Bernd Becker. MiraXT âĂŞ A Multithreaded SAT Solver.

[19] Tomohiro Sonobe and Mary Inaba. Portfolio with Block Branching for Parallel SAT Solvers. In *LION*, pages 247–252, 2013.

[20] Wahid Chrabakh and Rich Wolski. GrADSAT: A Parallel SAT Solver for the Grid, 2003.

[21] Wolfgang Blochinger and Carsten Sinz and Wolfgang KÃijchlin. Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Computing*, 29:969–994, 2003.

[22] Youssef Hamadi and Lakhdar Sais. ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 6, 2009.