Experiments with Massively Parallel Constraint Solving

Lucas Bordeaux, Youssef Hamadi and **Horst Samulowitz**
Microsoft Research Cambridge
{lucasb,youssefh,horsts}@microsoft.com

Abstract

The computing industry is currently facing a major architectural shift. Extra computing power is not coming anymore from higher processor frequencies, but from a growing number of computing cores and processors. For AI, and constraint solving in particular, this raises the question of how to scale current solving techniques to massively parallel architectures.

While prior work focusses mostly on small scale parallel constraint solving, we conduct the first study on scalability of constraint solving on 100 processors and beyond in this paper. We propose techniques that are simple to apply and show empirically that they scale surprisingly well. These techniques establish a performance baseline for parallel constraint solving technologies against which more sophisticated parallel algorithms need to compete in the future.

1 Context and Goals of the Paper

A major achievement of the digital hardware industry in the second half of the 20th century was to engineer processors whose frequency doubled every 18 months or so. It has now been clear for a few years that this period of "free lunch", as put by [Sutter, 2005], is behind us. The forecast of the industry is still that the available computational power will keep increasing exponentially, but the increase will from now on be in terms of number of processors available, not in terms of frequency per unit. This shift from *ever higher frequencies* to *ever more processors*[1] is perhaps the single most significant development in the computing industry today. Besides the high-performance computing facilities readily accessible by many AI practitioners in academia and the industry, novel architectures provide large-scale parallelism:

- *Multi-Core processors* are now the norm. Chip makers are predicting that the trend will from now on intensify from just a few cores to many [Held *et al.*, 2006], a shift which raises significant challenges for software development [Sutter, 2005].
- *Data-centers* now offer truly massive infrastructures for rent: Amazon offers a range of web services through which computing and storage can be used on demand, and Microsoft launched in 2008 its "operating system in the cloud", Azure. With such facilities anyone can now gain access to super-computing facilities at a moderate cost[2].
- *Distributed Computing* offers possibilities for groups of users to put their computational resources in common and effectively obtain massive capabilities. Examples include Seti@home and Sharcnet.

The number of processors provided in each scenario is already high, and it is bound to grow significantly in the near future. The main challenge raised by this new hardware is therefore to *scale*, *i.e.,* to cope with this growth.

Consequences for AI Compute-Intensive Techniques

How to utilize the computational power provided by new hardware is a question of technology and engineering, and one could argue that it is largely orthogonal to the *scientific* questions that should be at the center of AI research—larger machines will not magically provide us with a science of human intelligence.

However, what new hardware provides is increased scalability, and this increased scalability has in the past years directly contributed to the scalability of the *core techniques* of AI. In return this contributed to a broader application of AI and to some notable achievements such as Deep Blue, a spectacular example in which powerful hardware was used together with algorithmic innovations to defeat human intelligence at chess.

In the context of Linear Programming, Bixby [Bixby, 2002] clarifies the respective roles of software and hardware in the progress made from 1988 to 2002: software and hardware both contributed a speed improvement of roughly 1,000. For other constraint solving techniques, and other *compute-intensive* methods in general, it seems safe to say that hardware contributed *at least* as much as software to the progress

[1] Unless otherwise stated in this paper the word *processor* refers to an independent computing unit, whether it is a CPU or a core.

[2] At the time of this writing Amazon's price calculator (`http://aws.amazon.com/ec2/`) indicates for example that a medium, high-CPU instance can be used for $0.20 per hour. Occasionally renting resources can therefore become orders of magnitude cheaper than buying and maintaining a dedicated cluster.

observed in the past decades[3]. In other words progress would be considerably slower should we stop gaining from hardware improvements; this in our view gives a strong motivation for studying the question of parallel scalability.

Contributions and Outline of the Paper

There exists substantial prior work in parallel AI, *e.g.,* [Waltz, 1993] but little work on massively parallel constraint solving [Jaffar *et al.*, 2004]. Our paper extents the experimental study of massively parallel constraint solving and displays both novel and promising results. We start by a brief overview of parallel constraint solving in Section 2. From this overview two particularly promising approaches are identified; we treat them in order, propose new techniques for each approach, and conduct experiments. These are reported in Section 3 for the first selected approach (search-space splitting); and in Section 4 for the second (portfolios). We draw conclusions and elaborate upon the perspectives that arise from this work for massively parallel constraint solving in Section 5.

2 Parallel Constraint Solving

We assume that the reader is familiar with constraint solving in general. We are interested in all variants of constraint solving problems but will focus in most of the paper on propositional satisfiability (SAT). The reason is that the complex mix of dynamic heuristics and other techniques used by SAT solvers raises particularly interesting challenges for parallelization.

The main approaches to parallel constraint solving can roughly be divided into the following main categories:

Search Space Splitting strategies explore the parallelism provided by the *search space* and are probably the approach most commonly used [Pruul and Nemhauser, 1988], when a branching is done the different branches can be explored in parallel ("OR-parallelism"). One challenge with this approach is load balancing: the branches of a search tree are typically extremely imbalanced and require a non-negligible overhead of communication for work stealing. Recent works based on this approach are *e.g.,* [Zoeteweij and Arbab, 2004; Jaffar *et al.*, 2004; Michel *et al.*, To appear].

Portfolios explore the parallelism provided by *different viewpoints* on the same problem, for instance different algorithms or parameter tunings. This idea has also been exploited in a non-parallel context, *e.g.,* [Gomes and Selman, 2001; Xu *et al.*, 2007]. One challenge here is to find a scalable source of diverse viewpoints that provide orthogonal performance and are therefore of complementary interest. A recent solver based on this approach is [Hamadi *et al.*, 2008].

Problem Splitting is another idea that relates to parallelism, where the instance itself is split into pieces to be solved by each processor. One challenge here is that because no processor has a complete view on the problem, it typically becomes much *more difficult* to solve than in the centralized case as reconciling the partial solutions obtained for each subproblem becomes challenging. Instance splitting typically relates to *distributed CSPs*, a framework introduced in [Yokoo *et al.*, 1990] which assumes that the instance is naturally splitted between agents, for instance for privacy reasons.

Other Approaches can be thought of, typically based on the parallelization of one key algorithm of the solver, for instance constraint propagation. However parallelizing propagation is challenging [Kasif, 1990] and the the scalability of this approach is limited by Amdahl's law: if propagation consumes 80% of the runtime, then by parallelizing it, even with a massive number of processors, the speed-up that can be obtained will be under 5. Some other approaches focus on particular topologies or make assumptions on the problem.

Based on these observations the search-space splitting and portfolio approaches are in our opinion the most likely to offer scalable speed-ups. In the following we propose an experimental setting for both approaches. Note that even for these approaches scalability issues are yet to be investigated: most related works use a number of processors between 4 and 16; the only exception we are aware of is [Jaffar *et al.*, 2004] in the context of search-space splitting.

One issue with parallelism is that all approaches may (and a few must) resort to communication. Communication between parallel agents is of course costly in general: in shared-memory models such as multi-core this typically means an access to a shared data-structure for which some form of locking is usually unavoidable; the cost of message-passing cross-CPU is even significantly higher. Communication additionally makes it difficult to get *insights* on the solving process since the executions are highly inter-dependent and understanding parallel executions is notoriously complex. In this paper we simplify this by considering approaches in which *communication can be avoided.*

3 Search-Space Splitting Strategies

Here we propose a simple approach to search-space splitting that we call *splitting by hashing*. Let C denote the set of constraints of the problem. To split the search space of a problem into p parts one approach is to assign each processor i an extended set of constraints $C \cup H_i$ where H_i is a hashing constraint, which constrains processor i to a particular subset of the search space. The hashing constraints must necessarily satisfy the following property:

sound: The hashing constraints must *partition* the search space: $\bigcup_{i \in 0..p-1} H_i$ must cover the entire initial search space, and $\bigcap_{i \in 0..p-1} H_i$ should preferably be empty.

We claim that the following qualities are desirable:

effective: The addition of the hashing constraints should effectively allow each processor to efficiently skip the portions of the search space not assigned to it. Each processor should therefore solve a problem *significantly easier* than the original problem.

[3] We feel that our remarks and conclusions on constraint solving largely apply to a broader class of *compute-intensive* AI methods. There is another category of AI techniques for which massive parallelism is of considerable interest, namely *data-intensive* techniques such as those of data-mining, and these are clearly beyond the scope of this paper. Let us also insist that this paper is about one aspect of AI progress only: the measurable progress in terms of speed of these compute-intensive algorithms.

balanced: The splitting should be *balanced*, *i.e.,* all processors should be given about the same amount of work.

The most natural way to define such hashing constraints is the following: we select a subset S of the variables of the problem and define H_i as follows:

$$\sum_{x \in S} x \equiv i \pmod{p}$$

This effectively decomposes a problem into p problems. Note that p has to be within reasonable limits, *e.g.,* it should be smaller than the cardinality of the domain. (If no better choice is obvious one can always choose $p = 2$, in which case the sum imposes a parity condition, or XOR constraint.) Therefore one hashing constraint does not necessarily suffice to scale-up to an arbitrary number of processors—for this the splitting can be repeated in an obvious way: each of the subproblems obtained by hashing can itself be hashed again. For instance, by adding n parity constraints we scale to 2^n processors.

The splitting obtained with this approach is obviously *sound* and it is important to note that it is also most likely to be *balanced*: if we pick the variables in S at random we have a good chance of summing variables that have no significant correlation and for which the work on the even and odd sides is comparable. In fact, balance is a well-known theoretical property of parity constraints: if S is chosen large enough it can be proved that the two sides are balanced with very high probability [Valiant and Vazirani, 1985]. This property was used successfully by recent SAT/CP research to obtain fast approximate solution counts [Gomes *et al.*, 2006]. It seems to us that this technique is of interest in the context of parallelism, but this has to our knowledge not been investigated. One question that remains is nonetheless whether the approach satisfies the third criterion and is *effective*; we can answer this by experiments.

Experiments

In a first experiment[4] we verify the effectiveness of this technique in a classic CSP setting: we consider the well-known N-Queens problem (here $N = 17$) with a lexicographical variable ordering strategy and with values enumerated in increasing order. Here we enumerate the solutions to this problem and for enumeration the chosen heuristics are a reasonable choice. We impose a hashing constraint on 30% of the variables, picked at random, and we bias the static search heuristic so that it branches on these variables first.

What is striking on this result is that our simple splitting technique gives excellent results, with a linear speed-up for up to 30 processors. It is interesting to note that most other works that report similar results (*e.g.,* [Zoeteweij and Arbab, 2004], where the maximum number of processors discussed is 16; [Jaffar *et al.*, 2004] who scale up to 60 processors) use considerably more sophisticated parallel techniques and require communication. Beyond 30 processors our approach ceases to scale-up on this specific problem, but this is to a great extent explained by the fact that the instance consists only of a few variables and the chosen XOR constraints are not effective anymore.

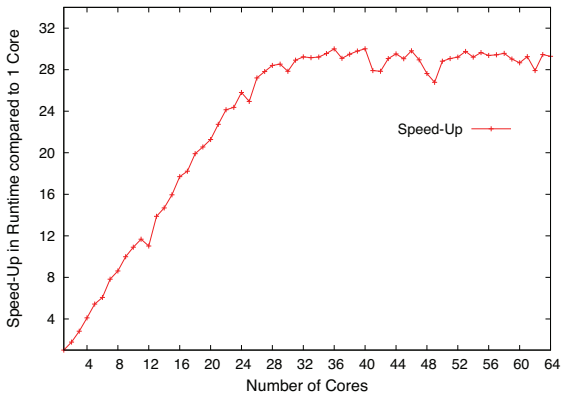

Figure 1: Search-Space Splitting Strategy: The Queens

The previous experimental setting is obviously in favor of *splitting by hashing* for three main reasons: 1) the variable ordering is static, 2) we are exploring a search space exhaustively, and 3) the problem is highly symmetric.

For a more thorough assessment of the scalability of the technique we use it in a very different setting: we use a recent SAT solver[5] and a recent selection of 100 industrial SAT instances (those of the latest SAT race). Since the variable ordering in SAT is extremely dynamic, SAT solvers do not exactly prove unsatisfiability by exhausting a search tree (clause learning is also used), and industrial SAT instances normally do not have a balanced search space, we avoid the three biases.

In this experiment we split each problem into 64 parts by using six parity constraints. Each of them is encoded into clauses in the natural way, and uses 3 variables picked at random. This was tuned experimentally: constraints of size 2 prove too likely to select variables with natural correlations, in which case hashing becomes *imbalanced* (one side is trivially unsatisfiable); high sizes tend to decrease *effectiveness* because propagation methods have difficulties with congruence constraints of large arity (*e.g.,* [Gomes *et al.*, 2006]).

In Figure 2 we display the performance of MiniSAT without and with added XOR constraints (curve "XOR") in terms of number of instances solved in dependence of time. For further comparison the figure also shows the curve obtained when using the technique we introduce in the next section ("fixed order"). We observe that even in an apparently unfavorable setting this method is able to improve the performance of the standard MiniSat solver (+7 instances solved), albeit significantly less than the portfolio approach that we discuss next. One reason why the gains are moderate is that we cannot afford to bias the variable ordering, and may lose *effectiveness*: dividing the search space by p using XORs does not always make the problem p times easier.

[4] All experiments in this paper were run on a 16 node cluster; each node having 2 Quad-Core AMD 2350 processors ($2.0GHz$) and 16GB of memory.

[5] The experiments we report use *Microsoft Solver Foundation* for CSP and *MiniSAT 2.0* for SAT. Note also that MiniSAT is for now used *without preprocessor (Satelite)*.

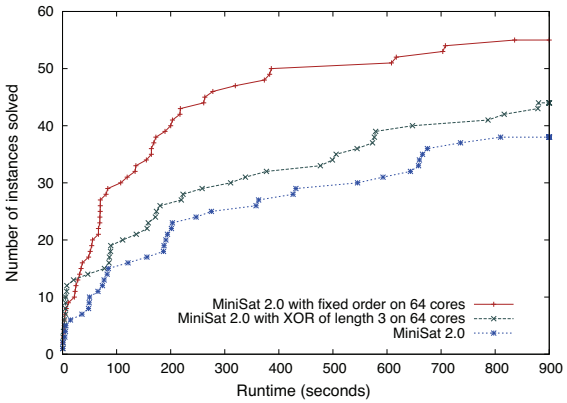

Figure 2: Search-Space Splitting Strategy on 100 SAT industrial instances

Conclusions

Splitting strategies are perhaps the most natural approach to parallel constraint solving and they remain of interest for massive parallelism. Our view is however that we should distinguish between 2 cases.

A relatively simple case is when the heuristic is static, as is most often assumed in parallel CSP works. Then the simpler search tree structure is not too difficult to break down into pieces, and this is therefore the setting in which most of the positive results are reported [Perron, 1999; Zoeteweij and Arbab, 2004; Michel *et al.*, To appear]. In this case our initial observation is that the speed-up can sometimes be reproduced using a simple hashing technique; this technique is therefore valuable as a baseline well worth considering before trying a more complex approach. We are confident that the technique should scale-up, for instance for applications like model counting.

A more challenging case however is with highly dynamic heuristics, as used by SAT solvers but also increasingly by sequential CP solvers. Our view is that splitting the search space explored using such strategies is much more difficult. Here the state-of the art is represented by the pMiniSAT solver of [Chu and Stuckey, 2008], which also incorporates other techniques such as an improved propagation. Judging from the SAT race 08 results, pMiniSAT is a powerful solver which outperforms the best sequential solver, but is itself outperformed by portfolio approaches (pMiniSAT ranked second of the race). Interestingly our experiments for fairly large-scale parallelism show similar conclusions to those obtained in the SAT race for small-scale parallelism (4 cores): similarly we obtain moderate speed-ups with our splitting approach.

The particular technique we have experimented is simple to apply and could conceivably be improved using communication to enable *e.g.,* work stealing [Chu and Stuckey, 2008; Jaffar *et al.*, 2004]. However, the challenges raised by communication, already complex and costly with small-scale parallelism, become daunting when the number of communicating processors gets large (*e.g.,* 10,000 cores).

4 Portfolio Approaches

Portfolio approaches exploit the variability of performance that is observed between several solvers, or several parameter settings for the same solver. We find it convenient to distinguish between two aspects of parallel portfolios:

- If assumptions can be made on the number of processors available then it is possible to handpick a set of solvers and settings that complement each other optimally. The SAT race 08 was using 4-core machines and the winner, ManySAT [Hamadi *et al.*, 2008], followed this approach, with some other improvements such as clause exchange.
- If we want to face an arbitrarily high number of processors we need *automated* methods able to generate a portfolio of any size on demand. Since there is always only a finite set of solvers this calls for an approach that exploits complementary settings of another kind: the challenge is to find a source of *variability* that can be favorably explored in parallel.

These two aspects are not totally orthogonal and should, in fact, be seen as complementary: one could use an optimized fixed-size portfolio as a basis which, given more processors, could be scaled-up using an automated method suggested in the second point. In this paper we focus on the second aspect.

While in general many sources of variability exist in constraint solvers, we are looking for variability sources with the following qualities:

scalable: many settings should give many different runtimes (otherwise there exist limitations to the number of settings that can be tried in parallel).

favorable: moving away from the default sequential setting should not systematically worsen the performance; it should have a probability of improving it.

solver-independent: each and every solver usually has a number of specific parameters (ratios, limits, frequencies, etc.) whose setting affects the performance. We do not explore this direction for purposes of reusability.

The best study of variability we are aware of is recent work on mixed integer programming [Danna, 2008], and we have essentially considered the same sources: permutating some of the data of the instance, and changing the random seed used by the solver. Our experience is that the variability offered by different random seeds is not scalable enough and this parameter is also hardly solver-independent (the use of random seeds varies among solvers—some do not employ randomness at all).

While permutations appear to be more promising with respect to solver-independence and scalability, it is the case that several types of permutations are clearly not very *favorable*: for example permutations of the constraints of an instance tend to separate constraints that are naturally grouped together in the encoding, leading to disadvantageous memory/caching and heuristic effects [Danna, 2008]. However, permutations of variable orderings are closer to the desired effect of exploring diverse viewpoints on a problem. It is well-known that variable ordering heuristics have a dramatic

impact on the performance of resolution. While we can expect a high variability, it remains unclear if this variability can be exploited *favorably*. One challenge that we want to test is if this source of variability works well with solvers that use complex and dynamic variable ordering heuristics: we cannot simply randomize the variable ordering and force the solver to follow it statically. In our studies the most effective variant[6] of permutation we discovered is to *partially fix the variable ordering*: we pick a small quantity q of variables at random and bias the search process so that these variables are always branched on first (using *e.g.,* a randomized polarity); the solver then continues branching with its normal strategy. It is the case that the solver can backtrack over those fixed variables if this is logically implied. Hence, the solver remains *complete*.

In Figure 3 we show for a sample of instances the variability in runtimes obtained when trying 500 different ways to fix just $q = 1$ variable (note the log-scale). Clearly, the variability is extremely high: by fixing a variable we obtain many runtimes significantly lower and higher than without. The technique seems to be *favorable* and satisfy the qualities we are looking for.

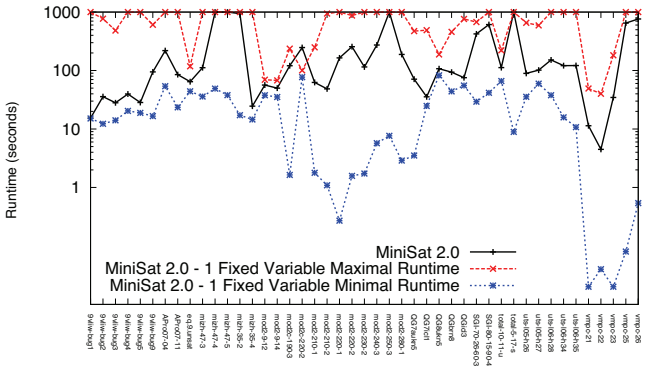

Figure 3: Variability of runtimes when using partially fixed variable orderings

The reasons why fixing a particular variable speed-up the resolution seem difficult to analyze: we tried to correlate the variable responsible for the best runtime with syntactic and topological features of the instance—without success. This absence of correlation made our idea to employ machine learning techniques to predict "good" variables fail.

Experiments

To evaluate our approach we conducted experiments in the context of SAT (same instances as before). Here we use a partially fixed variable ordering of $q = 3$ variables, which was set empirically. The obtained results are in our view very encouraging and for this technique we have therefore pushed the experiments up to 128 processors, following our initial goal of experimenting beyond 100 processors.

[6]A number of other variants remain of potential interest. Notably with solvers that use activity-based heuristics one can force the initial ordering to follow a randomly generated order by *assigning initial activities at random*. The solver then continues unimpaired.

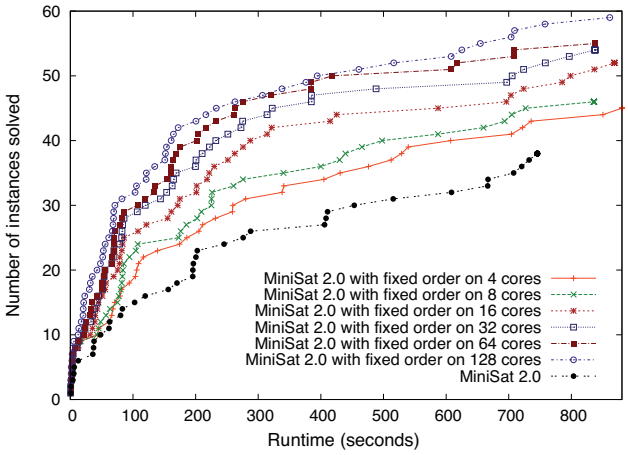

Figure 4: Partially Fixed Variable Ordering on 100 SAT industrial instances

In Figure 4 we compare the standard MiniSat solver with MiniSat employing the fixed variable strategy on $4, 8, .., 128$ processors. Again, we display the number of solved instances in dependence of time. We note that the technique scales very well in the sense that utilizing more processors consistently helps solving more instances. In addition, it is also the case that the increase in performance is quite significant. For example, on 128 cores we are able to solve 21 (+55%) more instances than standard MiniSat. In order to further analyze the quality of this techniques we address the following questions. First, one natural claim could be that the technique is strongly biased towards improving the solver on *satisfiable* instances. Detailed figures show that such is not the case, and that the gains of parallelism are quite balanced between satisfiable and unsatisfiable instances:

Number of cores	1	4	8	16	32	64	128
♯solved SAT instances	19	25	25	28	29	30	32
♯solved UNSAT inst.	19	20	21	24	25	25	27

Second, it is a legitimate question whether the effects of this approach are easily subsumed by other solving techniques. To address this point we considered the employment of the state of the art "Satelite" preprocessor for MiniSat. The corresponding results are displayed in Fig. 5 (l). While the improvements slightly decrease in this setting, we still observe a significant improvement in performance (+33% solved). Third, although our primary goal is to speed-up the resolution of practical problems we have experimented with random instances to verify if our conclusions on the respective merits of our techniques would be fundamentally different. The results are reported in Fig. 5 (r), which shows similar trends to Fig. 2. However, one important difference is that on random instances both of the techniques we propose seem to bring gains on satisfiable instances only.

Conclusions

The interest of portfolio approaches for parallel constraint solving was if needed evidenced by the fact that the winner of the parallel SAT race used this approach. Our findings confirm this and reveal that portfolios can be also scale-up to high

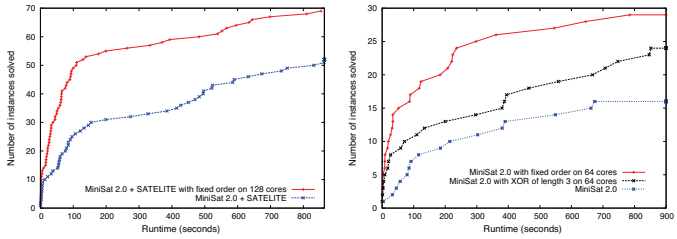

Figure 5: Detailed analysis: Satelite (l); random instances (r)

numbers of processors.

A major advantage of portfolio approaches is that each processor has a global view on the whole problem and its entire search space. Since all processors run a *complete* search with different viewpoints, the parallel search terminates as soon as one processor has finished, whether the instance is satisfiable or unsatisfiable. This is an important difference to the splitting approach, where the unsatisfiability of all parts have to be proven before we can terminate the resolution of an unsatisfiable instance. This makes the portfolio approach appealing when the goal is to determine satisfiability, as opposed to enumerating solutions. Our technique has the important advantage of obviously being extremely fault-tolerant: failed machines do not prevent the others from making progress.

Last, note that our approach wants to establish a bottom line performance which can be improved; in particular we have stated that adding communication does not come without major challenges—but it is surely worth further research.

5 Conclusion

The significant improvements obtained in Section 4 should ideally be contrasted with the fact that 128 processors are mobilized; this is obviously vastly more energy-consuming and the gains have to be compared against this consumption. Unfortunately when we reach a certain scale it becomes difficult to evaluate *speed-ups*: for the more challenging problems estimating the resolution times on a single processor is simply not always feasible. Our approach was therefore to focus on numbers of solved instances. This is justified by the fact that in general solving new problems matters more than reducing the runtimes of instances already solvable. However our feeling is that energy efficiency will become a growing concern in massively parallel computing (it is already a major one for data-centers); future research could develop methodologies and metrics to measure energy efficiency.

We wanted this paper to represent a step towards answering the question: *Which approaches scale with massively parallel architectures?* A promising approach, based on our experiments, are portfolios: the technique we proposed of *partially fixing variables* showed very encouraging results and there seems to be plenty of room to improve it. From our conclusions, whether search-space splitting techniques can scale-up well is unclear. These techniques have potential when reasonably static heuristics are used, and in this respect one of our contributions was to show that a simple technique of *hashing* sometimes suffices to obtain an excellent scalability. With complex heuristics we obtain encouraging results but the approach raises challenges: naturally improving our baseline approach would require communication, which can get intractable for massive parallelism.

References

[Bixby, 2002] R. Bixby. The new generation of Integer Programming codes. In *CP-AI-OR*, 2002. Invited talk.

[Chu and Stuckey, 2008] G. Chu and P. J. Stuckey. PMiniSAT: a parallelization of MiniSAT 2.0. In *SAT race*, 2008.

[Danna, 2008] E. Danna. Performance variability in mixed integer programming. In *5th Workshop on Mixed Integer programming (MIP)*. 2008.

[Gomes and Selman, 2001] C. Gomes and B. Selman. Algorithm portfolios. *Artif. Intel.*, 126(1-2):43–62, 2001.

[Gomes *et al.*, 2006] C. Gomes, A. Sabharwal, and B. Selman. Model counting: A new strategy for obtaining good bounds. In *Nat. Conf. on AI (AAAI)*, 2006.

[Hamadi *et al.*, 2008] Y. Hamadi, S. Jabbour, and L. Saïs. ManySAT: Solver description. In *SAT race*, 2008.

[Held *et al.*, 2006] J. Held, J. Bautista, and S. Koehl. From a few cores to many: A tera-scale computing research overview. Technical report, Intel Corp., 2006.

[Jaffar *et al.*, 2004] J. Jaffar, A. Santosa, R. Yap, and K. Zhu. Scalable distributed depth-first search with greedy work stealing. In *Tools with AI (ICTAI)*, pages 98–103, 2004.

[Kasif, 1990] S. Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artif. Intel.*, 45(3):99–118, 1990.

[Michel *et al.*, To appear] L. Michel, A. Su, and P. Van hentenryck. Transparent parallelization of constraint programming. *INFORMS J. on Computing*, To appear.

[Perron, 1999] L. Perron. Search procedures and parallelism in constraint programming. In *Principles and Practice of Constraint Programming (CP)*, pages 346–360. 1999.

[Pruul and Nemhauser, 1988] E. A. Pruul and G. L. Nemhauser. Branch-and-bound and parallel computation: A historical note. *O. R. Letters*, 7(2), 1988.

[Sutter, 2005] H. Sutter. The free lunch is over: A fundamental turn towards concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[Valiant and Vazirani, 1985] L. Valiant and V. Vazirani. NP is as easy as detecting unique solutions. In *ACM Symp. on Theory of Computing (STOC)*, pages 458–463, 1985.

[Waltz, 1993] D. L. Waltz. Massively parallel AI. *Int. J. of High Speed Computing*, 5(3):491–501, 1993.

[Xu *et al.*, 2007] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *J. of AI Research (JAIR)*, 32:565–606, 2007.

[Yokoo *et al.*, 1990] M. Yokoo, T. Ishida, and K. Kubawara. Distributed constraint satisfaction for DAI problems. In *Workshop on Distributed Artif. Intel.*, 1990.

[Zoeteweij and Arbab, 2004] P. Zoeteweij and F. Arbab. A component-based parallel constraint solver. In *Coordination Models and Languages*, pages 307–322. 2004.