Insights into Parallelism with Intensive Knowledge Sharing

Ashish Sabharwal[1] and Horst Samulowitz[2]

[1] Allen Institute for Artificial Intelligence (AI2), Seattle, WA 98103, USA
AshishS@allenai.org
[2] IBM Watson Research Center, Yorktown Heights, NY 10598, USA
samulowitz@us.ibm.com

Abstract. Novel search space splitting techniques have recently been successfully exploited to paralleliz Constraint Programming and Mixed Integer Programming solvers. We first show how universal hashing can be used to extend one such interesting approach to a generalized setting that goes beyond discrepancy-based search, while still retaining strong theoretical guarantees. We then explain that such static or explicit splitting approaches are not as effective in the context of parallel combinatorial search with intensive knowledge acquisition and sharing such as parallel SAT, where implicit splitting through clause sharing appears to dominate. Furthermore, we show that in a parallel setting there exists a surprising tradeoff between the well-known communication cost for knowledge sharing across multiple compute nodes and the so far neglected cost incurred by the computational load per node. We provide experimental evidence that one can successfully exploit this tradeoff and achieve reasonable speedups in parallel SAT solving beyond 16 cores.

1 Introduction

There have recently been several successful proposals for parallelizing combinatorial search and optimization, especially in the context of Constraint Programming (CP) and Mixed Integer Programming (MIP), such as by Régin et al. [25], Moisan et al. [22, 23], and Fischetti et al. [13]. A desirable strength of these and prior approaches such as the *guiding path* heuristic [31] is that they achieve parallelization without any communication between the compute cores. In one way or another, they split the underlying search space upfront or *statically* amongst the k available compute cores, which obviates the need for communication. Unlike search schemes based on global load-balancing or work-stealing [10, 21, 26, 27], these communication-less approaches compute a static assignment of subproblems (or of subtrees induced by a static assignment of search tree leaves) to each compute core.

We begin by discussing these static splitting, communication-less approaches and proposing a novel generalized static search space splitting scheme that, unlike some of these recent proposals, is not limited to a particular search strategy (e.g., discrepancy-based search) or a class of problem instances. This general

scheme uses randomly generated XOR or parity constraints and relies on their desirable universal hashing properties in order to achieve a dynamic balanced split of the search space amongst the compute cores. Prior works by Bordeaux et al. [8] and Plaza et al. [24] have alluded to XOR-based splitting, but only from an empirical perspective and without a formal analysis, especially with respect to the balanced effect of pruning *any* large-enough subtree of the whole space.

The formal correctness argument for our XOR-based splitting scheme highlights certain assumptions crucial to the success of some of the recent parallelization proposals. We explain why these assumptions, unfortunately, often fail in the context of search algorithms that dynamically learn from failures, such as CDCL (conflict-directed clause learning) solvers for propositional satisfiability (SAT) and Lazy Clause Generating CP solvers. In combinatorial search algorithms that support knowledge acquisition from failures or conflicts, intensively sharing that learned knowledge can provide an *implicit* way of splitting the search space explored by each core. As long as the solvers running at each core are sufficiently different, one of them will encounter a certain failed state before others and, by informing others of the reason of its failure, will indirectly prevent them from exploring this failed state as well as any search sub-space that fails for the same reason.

When it comes to knowledge sharing, it is common wisdom that communication across a network is costly. When designing distributed constraint solvers running on multiple machines and sharing information, it is deemed desirable to pack as many solvers on compute cores on individual machines as possible, so that inter-machine network latency does not hurt performance. This common wisdom stems from the undisputed fact that communicating across processes (or threads) on a single machine is much faster than communicating across different machines. While true, this reasoning ignores the *memory bandwidth* aspect, which can in principle have a negative impact on solvers. For example, it is folklore knowledge that running multiple copies of the MIP solver CPLEX [18] on the same machine notably degrades performances. The main reason is that more threads running memory intensive applications lead to more *cache misses* and *involuntary context switches*, both of which negatively impact performance. This happens even if one uses fewer threads than the number of available compute cores, because multiple cores tend to share at least the L3 cache. While cache performance of SAT solvers on a single compute node has been analyzed earlier [1, 32], its study in the context of multiple compute nodes and the resulting trade-offs remain unexplored.

As a motivating example, we consider the performance of the state-of-the-art parallel SAT solver Plingeling [6] across $k = 1, 2, 4, \ldots, 32$ cores of a single 32-core machine. Plingeling, like most current parallel SAT solvers, implements *implicit search space partitioning*. The results (geometric average runtime on our dataset, discussed later) are shown in Figure 1. We observe a sharp decline in performance when going from 16 to 32 cores. In this work, we ask: *Is this decline in performance caused mainly by duplication of work by cores given the lack of a static search space splitting mechanism, or by over-utilization of the memory*

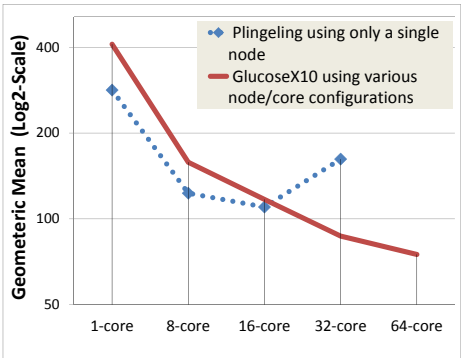

Fig. 1. Performance of Plingeling across $k = 1, 2, 4, \ldots, 32$ core on a single node compared to GlucoseX10 using various node-per-core configurations up to 64 cores.

bus? We find that reduced performance can be attributed to the latter and, more surprisingly, to such a large extent that one can, in fact, even successfully trade off the intra-node memory bandwidth bottleneck with the presumably high inter-node network communication cost.

Can this surprising trade off be successfully exploited in practice? To assess this, we first implement as a baseline two *static splitting* strategies that are promising in the context of SAT, one of them based on universal hashing through XOR constraints with strong theoretical guarantees as mentioned earlier. While a careful implementation of these strategies allows full knowledge sharing amongst the compute cores as well as the utilization of the highly effective dynamic search heuristics embedded in SAT solvers, we find that static splitting strategies have limited success in the context of SAT. However, exploiting our findings about the communication vs. node utilization trade-off, we show that a simple distributed variant of the Glucose 3.0 solver [3], created using the SatX10 framework [7] and performing *implicit search space splitting* by sharing the shortest 5% of the learned clauses, can continue to scale (i.e., have increasing speedups) on up to 64 cores when the copies of the solver are split carefully across multiple compute nodes. As shown in Figure 1 on our dataset, this simple distributed solver is more than competitive with Plingeling, the winner of the parallel track in the Application instance category of the 2013 SAT Competition [4]. Even though slower by as much as 1.5x when using 1 core, it clearly outperforms Plingeling as one moves to 32 or more cores, demonstrating that our insights can indeed be successfully exploited in practice.[3]

[3] We emphasize that this comparison is not meant to argue that GlucoseX10 is superior to Plingeling, but rather to illustrate that there exist unusual yet successful ways of making use of available compute resources.

2 Generalizing Static Search Space Splitting

We begin this section with a brief recapitulation of recently proposed successful parallelization strategies for CP and MIP search and optimization using what we will refer to as *static* search space splitting. With k compute cores, the search space is split upfront into k disjoint subspaces and then each core proceeds to search in the subspace it is responsible for. The novelty here is to achieve such a splitting in a way that requires no communication between the compute cores what-so-ever, which means communication cost never becomes a bottleneck as the number of compute cores is increased.

An interesting recent example of explicit search space splitting is the so-called *embarrassingly parallel search* (henceforth referred to as EPS) [25]. Suppose the problem instance has n variables, all of which are binary. The idea is to simply split the entire search space of size 2^n into $2^{\tilde{n}}$ disjoint subspaces, by fixing the value of some $\tilde{n}$ variables, $\tilde{n} < n$, in all possible ways. The resulting $2^{\tilde{n}}$ subproblems are then divided up equally amongst k compute cores. As long as $2^{\tilde{n}} \gg k$, by the law of large numbers, one expects the distribution of overall computation load across the compute cores to be roughly uniform. A related scheme [13] recently proposed in the context of MIP makes each core branch on the top $\tilde{n}$ variables and then choose a $1/k$ fraction of the resulting child nodes in a round robin fashion.

Another explicit parallelization technique—which in fact inspired some of our work—is the recent proposal by Moisan et al. [22] who study parallelization of a particular class of search heuristics in the context of CP, namely limited discrepancy-based search (LDS) [16]. We will refer to this technique as PLDS. It was shown that it is possible to assign the 2^n leaves of the entire search tree to the k available cores such that the leaves are visited by the k cores jointly in roughly the same order as a sequential LDS and the total number of search nodes visited by each core c (in the subtree induced by the leaves assigned to c) is no more than $(2^n/k) \log k$. Further, and more importantly, each of the cores is guaranteed to benefit roughly equally from any dynamic pruning of a subtree T of the entire search space by constraint propagators. All this is achieved by PLDS notably without any communication between the processors. This idea can also be extended to Depth-bounded Discrepancy based search (DDS) [23].

In the context of SAT, earlier work by Bordeaux et al. [8] suggested a completely distributed strategy where each core fixes a small number $\tilde{n}$ of variables (e.g., at random) without coordinating with other cores. Each core is allowed to independently select which $\tilde{n}$ variables it wants to fix and the values it wants to assign to them. To ensure completeness as well as to address the high likelihood of load imbalance in this context, the authors employed an interesting strategy of allowing the solver at each core to backtrack over the top $\tilde{n}$ variables—but only after it had proved its restricted sub-formula to be unsatisfiable. The authors also suggested more traditional search space splitting strategies that added new constraints to split the search space into disjoint subspaces, but, perhaps in part because of no knowledge sharing, they found the distributed variable fixing strategy to be the most effective. More recently, Heule et al. [17] and van der

Tak et al. [30] have proposed the use of more complex inference techniques than unit propagation to split the search space in the first phase of search and then solve the resulting sub-problems in parallel without knowledge sharing.

2.1 Generalized Splitting Using Universal Hashing

The theoretical balancing guarantees provided by the PLDS approach can in fact be extended to a more general setting for dynamic search heuristics that go well beyond LDS and DDS, including the conflict analysis driven heuristics employed by SAT solvers (e.g., VSIDS) as well as impact based search (IBS) in CP. We discuss here a novel way to achieve this in a search-independent and problem-independent manner, using parity or XOR constraints.

XOR constraints of length ℓ over binary variables x_i are constraints of the form $\sum_{i=1}^{\ell} x_i = p \mod 2$, where $p \in \{0,1\}$ is referred to as the *parity* of the constraint. When generated at random by choosing the set of ℓ variables as well as the parity p uniformly amongst all choices, XOR constraints (of large enough length) act as a family of uniform hash functions, resulting in desirable search space splitting properties that have been exploited in theoretical computer science [5, 29] as well as in the design of practically efficient approaches for approximating the number of solutions of a combinatorial problem and for probabilistic inference [9, 11, 14]. In the interest of space, we refer the reader to any of these other works for formal properties of XOR constraints. In our context, they have precisely the key properties exploited by the PLDS approach. We discuss below how this observation can be exploited.

Consider a sequential search algorithm S. Given a problem instance I on n binary variables, let σ be the ordered sequence of the subset of the 2^n leaves (at depth n) of the underlying search tree $\mathcal{T}$ (of size 2^n) that S visits when operating on I. To create a parallel version S^k of S that utilizes k compute cores, where for simplicity of exposition we assume k is a perfect power of 2, we generate at random $\log k$ sets $X_j, 1 \leq j \leq \log k$, of ℓ variables each and restrict the search space explored by core $i, 1 \leq i \leq k$, to the sub-space determined by XOR constraints C_{ij} defined as $\sum_{x \in X_j} x = b_{ij} \mod 2$, where b_{ij} is the j-th bit of the $\log k$ bit binary representation b_i of the integer i. Let $C_i = \bigwedge_j C_{ij}$. The i-th solver S_i of S^k running on core i operates on the restricted problem instance $I \wedge C_i$. S_i follows the original leaf sequence σ, but simply skips the leaves that do not satisfy its XOR constraints C_i. For efficiency, if there is a subtree T of $\mathcal{T}$ none of whose leaves satisfy C_i, S_i must identify this fact and not waste time exploring T at all. This, in our case, is easily achieved as at least one constraint C_{ij^*} for some j^* must be violated by the partial truth assignment that defines the root node of T, which means that the XOR propagator for C_{ij^*} would make S_i fail immediately as soon as it reaches T.

This restriction scheme ensures that every pair $S_i, S_{i'}$ of solvers operates in disjoint subspaces of $\mathcal{T}$, and that the k cores together cover all of $\mathcal{T}$. Formally:

Proposition 1. *For constraints C_i as defined above and for $i \neq i'$, $C_i \wedge C_{i'} = \bot$ and $\bigvee_i C_i = \top$. Further, $(I \wedge C_i) \wedge (I \wedge C_{i'}) = \bot$ and $\bigvee_i (I \wedge C_i) = I$.*

Proof. Let j^* be a bit in which the $\log k$ bit binary representations of i and i' differ, i.e., $b_{ij^*} \neq b_{i'j^*}$. Then $(C_i \wedge C_{i'}) = (\bigwedge_j C_{ij}) \wedge (\bigwedge_j C_{ij}) \Rightarrow (C_{ij^*} \wedge C_{i'j^*}) \Rightarrow (\sum_{x \in X_{j^*}} x = b_{ij} \mod 2) \wedge (\sum_{x \in X_{j^*}} x = b_{i'j^*} \mod 2) \Rightarrow (b_{ij^*} = b_{i'j^*})$, which, by the choice of j^*, is never the case. Hence, $C_i \wedge C_{i'} = \bot$.

On the other hand, $\bigvee_i C_i = \bigvee_i \bigwedge_j C_{ij} = \bigwedge_j \bigvee_i C_{ij} = \bigwedge_j \bigvee_i (\sum_{x \in X_j} x = b_{ij} \mod 2)$. Since i spans the range $\{0, 1, \ldots, k-1\}$ and we are working with $\log k$ bit representations, for each j there must exist an i such that $b_{ij} = 0$ and an i' such that $b_{i'j} = 1$. Hence, $\bigvee_i (\sum_{x \in X_j} x = b_{ij} \mod 2) = \top$ for every j, implying $\bigwedge_j \bigvee_i (\sum_{x \in X_j} x = b_{ij} \mod 2) = \top$ and finishing the proof that $\bigvee_i C_i = \top$.

The remaining claims now follow from these results. First, $(I \wedge C_i) \wedge (I \wedge C_{i'}) = I \wedge (C_i \wedge C_{i'}) = \bot$. Next, $\bigvee_i (I \wedge C_i) = I \wedge \bigvee_i C_i = I \wedge \top = I$. □

We thus have a static partition of the search space amongst the k solvers. Moreover, the partition is balanced in the sense that each solver S_i gets precisely a $1/k$ fraction of the overall 2^n size search space $\mathcal{T}$ (irrespective of I). Most interestingly, the uniform hashing properties of XORs guarantee that, with large enough ℓ, with high probability, *every large-enough subspace of $\mathcal{T}$ has roughly equal representation in each of the k compute cores*, which act as k "buckets" for the underlying hash function. This is formalized in the following theorem, which notably is independent of the properties of the search algorithm S (e.g., using LDS or not) or of the problem instance I.

Theorem 1. *Let S^k be the parallel constraint solver for k cores as described above, operating on a problem instance I over n binary variables forming the 2^n size search space $\mathcal{T}$. For $\ell = n/2, \epsilon \in (0,1), \delta > 0$, and $k \leq 2^{n/(2+\delta)}$,*

1. *the entire subtree $\mathcal{T}_i$ of $\mathcal{T}$ induced by the leaves assigned to S_i contains no more than $(2^{n+1}/k) \log k$ internal and leaf nodes combined; and*
2. *for any arbitrarily chosen subtree T of $\mathcal{T}$ with $L \geq k^{2+\delta}/\epsilon$ leaves, with probability at least $1 - \epsilon$ over the choice of the random XOR constraints, the following holds: For any core i, the number of leaves of T that are assigned to S_i lies within $\mu \cdot (1 \pm k^{-\delta/2})$ where $\mu = L/k$ is the expected value.*

Proof. To argue that the first claim holds, we observe that $\mathcal{T}_i$ has exactly $2^n/k$ leaves, which implies that the number of internal nodes of $\mathcal{T}_i$ with two children must be exactly $2^n/k - 1$. Thus, the total number of nodes in $\mathcal{T}_i$ is higher precisely when it has more internal nodes with only one child. As can be seen from a tree rotation argument, the number of internal nodes with only one child is maximized when the leaves of $\mathcal{T}_i$, all at depth n of $\mathcal{T}$, are uniformly spread apart at distance k from each other. In this case, $\mathcal{T}_i$ contains *all* internal nodes of $\mathcal{T}$ up to depth $n - \log k$, for a total of $2^{n - \log k + 1} - 1$ nodes, each of which is extended to depth n by a unique path of length $\log k$ containing nodes with one child. It follows that the number of nodes in $\mathcal{T}_i$ is upper bounded by $(2^{n - \log k + 1} - 1) \log k < (2^{n+1}/k) \log k$ as desired.

In order to prove the second claim, we capitalize on the known fact that $\log k$ random XORs of length $n/2$ act as a universal family of hash functions on the 2^n leaves of $\mathcal{T}$, placing the leaves pairwise independently into k different "buckets",

which correspond to our k cores. Let L_i be a random variable (with randomness over the choice of XORs) capturing the number of leaves of T assigned to core i. Pairwise independence of the assignment of leaves to cores implies that the variance $\mathrm{Var}(L_i)$ of L_i is no more than its expected value $\mathbb{E}(L_i) = L/k = \mu$. Applying first the Chebychev inequality and then the union bound,

$$\Pr\left[|L_i - \mu| \geq \mu k^{-\delta/2}\right] \leq \frac{\mathrm{Var}(L_i)}{\mu^2 k^{-\delta}} \leq \frac{\mu}{\mu^2 k^{-\delta}} \leq \frac{1}{k}\frac{k^{2+\delta}}{L} \leq \frac{\epsilon}{k}$$

$$\Rightarrow \Pr\left[\exists i.\ |L_i - \mu| \geq \mu k^{-\delta/2}\right] \leq \sum_{i=1}^{k} \Pr\left[|L_i - \mu| \geq \mu k^{-\delta/2}\right] \leq \epsilon$$

Taking the complement of this probability finishes the proof. □

We note that although the theorem is stated for $\ell = n/2$, Ermon et al. [12] have recently shown that certain desirable hashing properties still hold with XORs of length $\ell \ll n/2$, which are often much easier to propagate.

As an illustration of the result, suppose $\epsilon = 1/n, \delta = 1$, and T is *any* subtree of the search space with $L \geq nk^3$ leaves. Then the theorem states that with probability at least $1 - 1/n$ over the choice of random XORs, the number of leaves of T assigned to S_i will be within $L/k \cdot (1 \pm 1/\sqrt{k})$, i.e., very close to the ideal balancing value of L/k. As a consequence, each core will benefit roughly equally if a constraint propagator prunes T.

2.2 Implementing XOR-Based Splitting with Knowledge Sharing

While the above reasoning shows that adding randomly generated XOR constraints can, in principle, qualitatively provide the guarantees of PLDS in a much more generic setting, it is not obvious how best to implement this strategy. One of the parallelization suggestions by Bordeaux et al. [8] was in fact to add random XOR constraints by converting then into a CNF formulation. An XOR constraint with ℓ variables, however, requires adding $2^{\ell-1}$ clauses, which quickly becomes impractical as ℓ grows.[4] Thus, for practical reasons, we limit our evaluation to small values of ℓ. Bordeaux et al. did not find this to be effective, but their tests were performed without communication while we now test the approach in the presence of knowledge sharing, in the context of SAT. This, however, immediately raises an implementation challenge, which we discuss next.

Since each S_i operates on the original instance conjoined with new constraints that differ from core to core, clauses learned by one core may not be valid for other cores. In principle, one can label each learned clause C as sharable or not based on the information used to derive C. However, due to subtleties in the implementation of modern SAT solvers, it is insufficient to simply check whether the conflict analysis that led to the derivation of C involved one of the clauses encoding an XOR constraint. Other operations in the solver, such as propagation

[4] One could alternatively use $\mathrm{O}(\ell)$ new variables to encode the XOR constraint, but this is known to slow down the search [14].

of unit literals learnt based on XOR constraints and clause base reductions, must also be appropriately altered to take the effect of the XOR constraints that differ from core to core.

An interesting alternative to explicitly adding new constraints $\mathcal{X}$ to the formula F is to alter the branching heuristic such that the solver *automatically* searches only in the assignment subspace that satisfies the constraints $\mathcal{X}$. The idea is to pre-compute all solutions to $\mathcal{X}$ over the set of variables appearing in $\mathcal{X}$ and add them to a solution pool $\mathcal{S}$. Note that each $\sigma \in \mathcal{S}$ is a partial assignment for F. Now one can iterate through these partial assignments $\sigma_1, \sigma_2, \ldots, \sigma_{|S|} \in \mathcal{S}$, moving from σ_i to σ_{i+1} as soon as the solver refutes the subtree under σ_i. Since we do not add XORs explicitly as constraints and instead just branch in a way that is consistent with XORs, the original formula must logically entail any learnt clause. Furthermore, since we enumerate the solution pool $\mathcal{S}$ exhaustively the approach is both sound and complete.

A notable advantage of this approach is that *all* clauses learnt by any core are valid for all other cores as well, and can thus be freely shared.[5] This obviates the need to implement mechanisms to decide which clauses are safe to share and which aren't. On the other hand, for the approach to be practical, the solution pool $\mathcal{S}$ must have a succinct representation that the solvers can exploit. For example, if $\mathcal{X}$ is the conjunction of $\log k$ XOR constraints of length ℓ each on disjoint sets of variables, then $|\mathcal{S}| = (2^{\ell-1})^{\log k} = k^{\ell-1}$, which can quickly become huge as k grows. To avoid this blow up, we instead use $\log k$ solution sub-pools of size $2^{\ell-1}$, one for each XOR constraint. The branching heuristic first fixes all variables from one sub-pool before moving on to the next sub-pool.

Our implementation includes a range of variations and extensions. For instance, we experimented with the setting where one core remains completely unaltered while all other cores employ XOR based branching. This strategy was motivated by the fact that altering branching decisions can have a tremendous impact on the search, and adding one unchanged solver to the pool of solvers would retain some of the original search pattern and the solver's flexibility. Furthermore, when branching according to the XOR variables at the top of the search tree we take propagation results directly into account so that an implication that falsifies the current XOR assignment causes us to directly move on to the next assignment. Since we have a sequence of XORs to branch on, this often allows us to skip entire sets of assignments.

2.3 Limits of Static Splitting

To our surprise, none of the approaches discussed above was truly effective in parallelizing SAT solvers. We tried several variations and parameter settings and will describe some representative results in Section 3, Table 1. As we discuss next, the reason might lie in the "rigid" static search space splitting interfering with the highly dynamic conflict-directed search performed by SAT solvers.

[5] Clauses learnt at core i could be filtered based on the alignment of XORs between cores i and j so that only the ones that have a chance of ever being triggered at core j are shared with it.

In general, the recent static splitting proposals discussed at the beginning of this section and which inspired our XOR-based splitting mechanism, unfortunately, have significant limitations in the context of search strategies that apply aggressive search space pruning through powerful propagators or that use information learned from failures to guide future search.

For example, the EPS approach and its variation for MIP implicitly assume that once the huge pool of $2^{\tilde{n}}$ subproblems has been created, one simply must resolve each subproblem independently and knowledge learned from solving one subproblem cannot significantly help inference on other subproblems. This is clearly not the case for CDCL solvers which are heavily guided by clauses learned from conflicts and are in fact able to quickly prune new subproblems based on experience from previously encountered subproblems. The same applies also to CP solvers that perform conflict analysis and lazy clause generation [28].

The PLDS and XOR-splitting approaches, on the other hand, do not address the rather common case where the number of nodes s visited by a sequential search algorithm is significantly less than 2^n. In fact, s being vastly smaller than 2^n on real-world instances of interest is precisely what allows us to tackle large NP-complete problems in a reasonable amount of time. For example, consider an infeasible instance where fixing well-selected $\tilde{n} \ll (n - \log k)$ variables lets constraint propagators already deduce infeasibility. With $k = 1024$, this condition holds whenever $\tilde{n}$ is much smaller than $n - 10$, which most SAT and CP solvers will guarantee fairly easily. While it does holds that each S_i will not process more than $(2^{n+1}/k) \log k$ nodes and subtree pruning will positively impact each S_i roughly equally, this is not a very useful guarantee as even a well-guided sequential search would take only $2^{\tilde{n}} \ll 2^n/k$ steps to begin with! In general, *uniform splitting of the naïve search space of size* 2^n *across* k *cores does* not *say much about speedups being close to* k *unless the underlying search algorithm works in a rather brute force manner.*

Further, PLDS has so far been demonstrated to be effective in a setting where the leaves of the search tree are much more costly to process than internal nodes. This, however, is usually not the case in most SAT, CP, and MIP applications, where node processing time often *decreases* as one goes deeper in the search tree.

Finally, forcefully fixing variables at the top of the search tree, as is the case in our XOR implementation and the random prefix method [8], seems to often interfere with dynamic branching choices that tend to make search more effective. This is especially true for SAT solvers, which heavily rely on recent conflicts for branching decisions, and variable activities maintained by them often change very rapidly. This behavior is also reflected in improved performance that we observed when using shorter XORs and random prefixes.

3 Implicit Splitting through Intensive Knowledge Sharing

An alternative to static splitting is *implicit* search space splitting, where the k compute cores start exploring the entire search space independently but dynamically communicate to each other—ideally in a succinct fashion—which subspaces

Table 1. Performance of various search space splitting and knowledge sharing approaches for SAT, using $k = 32$ cores

Search Space Splitting		Clause Sharing	Runtime	#Solved
Approach	Length	(%)	(sec)	(count)
Implicit	–	2	139	62
Implicit	–	5	119	63
Implicit	–	8	113	64
Static, 5 XORs	2	5	132	60
Static, 5 XORs	3	5	161	59
Static, Random Prefix	5	5	141	61
Static, Random Prefix	10	5	173	59

they have already explored. We argue that in combinatorial search algorithms that learn from failures, such as CDCL SAT solvers and CP solver employing Lazy Clause Generation (LCG), *implicit search space splitting achieved by sharing succinct reasons of failures can be much more effective than explicit search space splitting schemes* such as those discussed above. When such solvers encounter a "conflict" or a failed state, they analyze the reason of the failure in terms of constraint propagations and learn a clause (or a small set of clauses) that would prevent the solver from wasting time in other states that fail for a similar reason. By sharing such learned clauses with other cores, one can implicitly achieve search space splitting—as long as there is enough variation among the solvers executing on different cores that one of them encounters a particular failed state before others do. While simply having different random seeds at each core can make the search sufficiently different, in our experiments we vary also some of the solver's parameters, such as activity decay rate and restart frequency, across various cores.

When viewing knowledge sharing as implicit search space splitting, one must revisit the heuristics commonly used to decide how much to share and when. Currently employed heuristics, again guided by the common wisdom of communication latency being the main hurdle to avoid, tend to share perhaps too little information [19]. As the representative results in Table 1 (to be discussed shortly) demonstrate, it can be better to pay the price of additional communication for implicit search space splitting than use explicit splitting.

The results throughout the paper are for experiments on a cluster of 32-core compute nodes. Each node is a 4x8 3.8 GHz Power7 machine (CHRP IBM 9125-F2C), 4 MB cache per CPU, and 128 GB of RAM. The nodes are connected via a network that supports the PAMI message passing interface [20]. The evaluation is on SAT Race 2010 instances (all industrial/application category) restricted to the 73 that 1-core Glucose 3.0 could not solve within 10 seconds. The timeout (wall-clock) used was 5,000 seconds; our results remain qualitatively unchanged for smaller timeouts as well.

In Table 1, we compare (a) the implicit splitting approach with each core sharing the shortest 2%, 5%, and 8% of its learned clauses;[6,7] (b) the XOR-based explicit splitting approach with XORs of lengths 2 and 3; and (c) the "random prefix" approach of Bordeaux et al. [8] fixing 5 or 10 variables and enhanced with sharing the shortest 5% of the learned clauses. For the XOR-based splitting approach, longer XORs achieved more load balancing as expected but worsened per-core performance as XOR constraints do not unit propagate very well. Shorter XORs led to significantly increased load imbalance, but we could counter this and somewhat improve the overall performance by (i) restarting an early finishing core on the full problem instance (no XORs) or (ii) using one additional core in parallel on this full instance, in both cases continuing to share information. The table reports numbers for the latter, which, as we see, was still insufficient to outperform implicit search space splitting.

Implicit sharing was nearly always the best and sharing 5% yielded good, stable performance for various values of k. This is the setup we use henceforth.

4 The Communication-Utilization Tradeoff

Suppose we have two machines M_1 and M_2 with 32 compute cores each. Let S be a solver that we can run in parallel on one or both of these machines, and we can share information learned from failures across copies of S running on these machines. We are interested in minimizing the time the slowest of the parallel copies of S takes. When running $k \leq 32$ copies of S on a problem instance I, is it better to run k copies on M_1 or $k/2$ copies each on M_1 and M_2?

The answer, it turns out, is not that straightforward. It depends on k (in relation to the number of cores, 32, on each machine), the amount c of communication between each pair of copies of S, and the intensity of memory accesses (and the resulting cache hits and misses) performed by S. As one might expect, when $k \ll 32$ and c is small, either option results in about the same performance. As one might also expect, when $k \ll 32$ but c is substantial, network latency does play a significant role and it is better to run k copies on M_1 rather than communicate across machines. On the other hand, when $k \sim 32$ but the amount c of communication is very small, the latency of inter-machine communication does not play a significant role and it is actually better to run $k/2$ copies on two machines because constraint solvers often require high memory bandwidth and interfere (at the system level) more with each other the more copies of them are run in parallel on a single machine (we quantify this interference later in the

[6] Sharing of the shortest x% learned clauses is implemented by maintaining a dynamic cutoff length L such that all learned clauses of length up to L are shared. L is adjusted periodically to achieve the x% sharing target. In principle, x itself could be adjusted based on the total number k of cores or properties of the problem instance.

[7] We also experimented with more sophisticated sharing schemes such as those based on the LBD level of the learned clause [3], but our main findings remained unaffected. To avoid unintended consequences of complex sharing mechanisms, we report results on the simplest setting which still achieved state-of-the-art performance on 64 cores.

paper). Remarkably, we find that this trend continues to hold for $k \sim 32$ *even when the amount c of communication is high*. Specifically, even if each copy of solver S shares with every other $k-1$ copy as many as 5% of the clauses that it learns, it is better to run $k/2$ copies each on M_1 and M_2 and pay the price of inter-machine network communication than have k copies compete for memory bandwidth on M_1. E.g., it is often better to run 16 copies on 2 machines, and sometimes even better to run 8 copies on 4 machines, than run 32 copies on a single machine with 32 cores. The best allocation of cores across nodes is clearly dependent on the machine type. However, our empirical observations suggest that the configuration process does not have to be very fine grained.

We note that with $k = 32$ copies, sharing 5% of the learned clauses results in a significant amount of communication as each copy of S, when generating m learned clauses per second on its own, is expected to receive $m*(k-1)*5/100 = 1.55m$ clauses from other copies of S. In other words, each copy listens more to others than spend time doing its own deductions.

4.1 Time Profile of SAT Solvers

When working in a parallel setting where solvers run possibly on different machines and intensively share information, it is important to understand where these solvers spend most of their time and what role does the communication cost play. For CDCL SAT solvers, the total time T can be divided up as follows:

$$\begin{aligned} T &= T_{\text{conf}} + T_{\text{prop}} + T_{\text{comm}} + T_{\text{misc}} \\ &= \frac{N_{\text{conf}}}{R_{\text{conf}}} + \frac{N_{\text{prop}}}{R_{\text{prop}}} + \frac{N_{\text{comm}}}{R_{\text{comm}}} + T_{\text{misc}} \end{aligned}$$

Here T_{conf} represents the total time spent performing conflict analysis, N_{conf} represents the number of times conflict analysis is performed, and $R_{\text{conf}} = N_{\text{conf}}/T_{\text{conf}}$ is the associated rate, i.e., time per conflict analysis. The other symbols similarly correspond to the time, number, and rate of unit propagations, and of communication between cores. Since our main interest is in studying parallelization, we will compute and report all times in terms of wall-clock time.

There have been studies suggesting that SAT solvers spend a substantial amount of time performing unit propagation, and this observation has been used to help understand the behavior and limitations of parallel SAT solvers [15]. To make the numbers concrete in our setting and with our solver, we computed the values of T_{conf} and T_{prop} for Glucose 3.0 on our dataset. Figure 2 shows the result in relative terms as a percentage of T. To make relative numbers meaningful, the dataset was restricted to instances needing at least 30 seconds to solve. Clearly, unit propagation does dominate the time Glucose spends solving most instances, 71.01% on average[8] and never less than 40% on our dataset. Conflict analysis is the next most expensive operation. It can vary from 5% to 45%, with the geometric average being 12.45%. We will return to these observations later.

[8] All results are reported as a geometric mean, which is often more robust to high outliers than arithmetic mean. Results remain qualitatively unchanged either way.

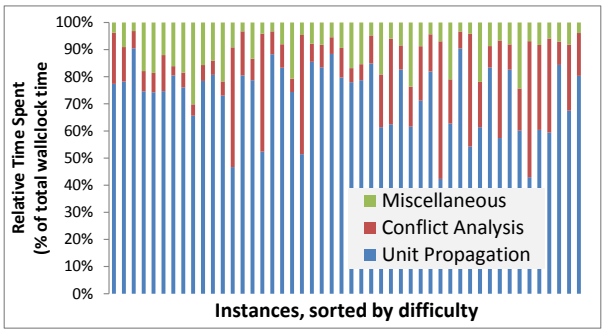

Fig. 2. Relative split of the total time spent by Glucose into T_{conf}, T_{prop}, and T_{misc}

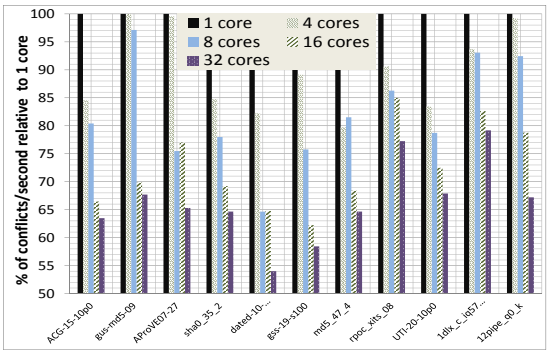

Fig. 3. Impact on R_{conf} when running $k = 4, 8, 16, 32$ independent copies of Glucose. Shown, for a selected set of instances, is R_{conf} as a percentage of the baseline R_{conf} obtained by running only a single copy of Glucose.

While it is folklore knowledge that running k independent processes on a single compute node with m cores can slow down each process as k approaches m, the large extent of slow down is rather surprising for SAT solvers. To quantify this effect, we ran k independent copies of the 1-core solver on a compute node with 32 cores. The plot in Figure 3, which shows the results on individual instances where Glucose on 1 core took between 600 and 1,800 seconds, demonstrates that the rate R_{conf} of conflict analysis is significantly reduced as k increases, by as much as 45% when increasing k from 8 independent copies to 32 copies. We obtained similar results for R_{prop} (omitted due to space limitation).

4.2 Communication Cost vs. Node Utilization

Now suppose we run k cooperating copies of a solver in a parallel setting with information sharing among the copies. How does the slow down seen when run-

ning $k \approx m$ copies on a single compute node compare with the communication cost incurred when running, say, $k/2$ copies on two different compute nodes?

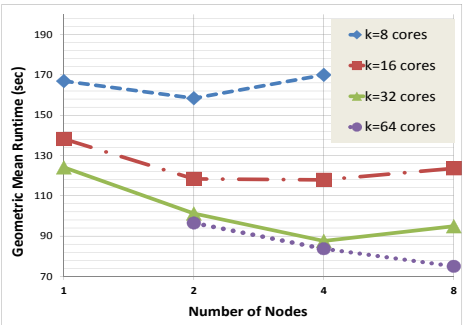

Fig. 4. Trade-off: communication cost vs. node utilization

Figure 4 depicts this trade-off, where on the horizontal axis we have the number N of compute nodes used, on the vertical axis is the performance (measured as the geometric mean of the runtimes across instances), and each curve corresponds to a different total number k of cores used in parallel. Each compute node thus runs k/N solvers in parallel.

While for small k (e.g., $k = 8$) performance, as expected, drops when increasing N due to the communication overhead, for larger values of k, increasing N surprisingly leads to significantly better performance. For example, the data shows that when wanting to run $k = 32$ solvers in parallel, it is substantially better to run only $k/N = 16$ or even 8 solvers per compute node than to fully utilize the node by running $k = 32$ solvers on it. Based on these findings, we used the following configurations of N nodes with k/N cores per node (for a total of $N \times k/N = k$ cores) produce data for the GlucoseX10 curve in Figure 1: $1 \times 1 = 1, 2 \times 4 = 8, 2 \times 8 = 16, 4 \times 8 = 32$, and $8 \times 8 = 64$.[9] The figure shows that GlucoseX10, while worse than Plingeling for $k = 1$, continues to scale reasonably well even up to $k = 64$ and clearly outperforms Plingeling for $k > 16$.

To understand this behavior, let us consider the time profile of SAT solvers discussed earlier. By increasing N and keeping everything else unchanged, we must clearly decrease the communication rate R_{comm}. Assuming T_{misc} is not affected significantly and neither are the total numbers N_{conf} and N_{prop} of conflicts and unit propagations, respectively, the only way the overall time T can decrease, is for one or both of the rates R_{conf} and R_{prop} to significantly increase. This, indeed, is the case. Across all problem instances N_{conf} is not systematically altered one way or another by changing N from 1 to 4 with $k = 32$ cores in total.

[9] While the specific numbers reported here are based on evaluation on our compute hardware, our qualitative findings are likely to be applicable to other compute systems as well. Simple experimentation can be used to identify the most effective split of k cores of a parallel solver across multiple compute nodes.

However, as expected, R_{conf} increases quite consistently across all instances and the geometric mean of R_{conf} is roughly 20% larger when using 4 nodes compared to 1 node. A similar trend holds also for N_{prop} and R_{prop}.

These results highlight the rather surprising tradeoff between the utilization of each compute node and the communication cost across multiple nodes. They also show that this tradeoff can be fruitfully exploited.

5 Concluding Remarks

Limited intra-node memory bandwidth has a substantial impact on the performance of today's combinatorial search methods when several such solvers, or their parallel versions, operate on a single machine. One may naïvely consider this impact smaller than the usually high latency of communication across a network. Our results, however, demonstrate that one can significantly gain in performance by distributing a parallel solver across multiple machines *even when the solver employs extensive knowledge acquisition and sharing.* For example, a SAT solver learning around 1,000 clauses per second and sharing 5% of what it learns with other 31 solvers in turn receives $1{,}000 \times 31 \times 0.05$, or over 1,500 clauses per second. Even then, as our results show, distributing 32 cores across 4 or 8 compute nodes pays off. A testament to the practical importance of this insight is that one is able to significantly outperform the state of the art in parallel SAT solving on 32 or more cores.

This is, of course, not a complete solution to effective parallelization of SAT solvers or CP solvers with lazy clause generation. By using only a subset of the available cores on a machine and letting others idle, we are essentially wasting resources. However, our results suggest that, rather than allocating idle cores to other solvers running in parallel, one should consider other uses of the idle cores. In particular, it may be worthwhile revisiting operations such as unit propagation and conflict analysis, which often take up nearly 90% of the solver's time and must be performed in any case. Our results motivate parallel propagation and conflict analysis schemes as was also suggested earlier by Hamadi and Wintersteiger [15]. In this context, it is important to note that while unit propagation is P-complete [15], the only known theoretical consequence of this completeness observation is that a q-step unit propagation sequence cannot be parallelized to $\log^i q$ parallel steps for any constant i. However, this does not preclude reductions by large constant factors or even asymptotic reductions to, say, $\sqrt{q}$ parallel steps. This may be a more promising use of idle cores than running additional copies of the solver as this is likely to have a more coherent memory footprint across cores and thus be more amenable to better cache performance.

As the total number of compute cores grows, the communication cost must eventually become dominant. It, therefore, remains important to consider more sophisticated knowledge sharing schemes [2] or more parallelizable proofs [19], both of which are promising research directions orthogonal to our findings. Any improvements along these lines will affect our approach positively as well and help it scale to more compute cores.

Bibliography

[1] M. Aigner, A. Biere, C. M. Kirsch, A. Niemetz, and M. Preiner. Analysis of portfolio-style parallel SAT solving on current multi-core architectures. In *POS-2013: Intl. Workshop on Pragmatics of SAT*, Helsinki, Finland, 2013.

[2] G. Audemard, B. Hoessen, S. Jabbour, J.-M. Lagniez, and C. Piette. Revisiting clause exchange in parallel sat solving. In *SAT*, pp. 200–213, Trento, Italy, 2012.

[3] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In *21st IJCAI*, pp. 399–404, Pasadena, CA, July 2009.

[4] A. Balint, A. Belov, M. Heule, and M. Järvisalo. SAT competition, 2013.

[5] M. Bellare, O. Goldreich, and E. Petrank. Uniform generation of NP-witnesses using an NP-oracle. *Information and Computation*, 163(2):510–526, 2000.

[6] A. Biere. Lingeling, Plingeling and Treengeling entering the SAT Competition 2013. In *Proc. of SAT Competition 2013*, vol. B-2013-1 of *Dept. of Computer Science Series of Publications B, Univ. of Helsinki*, pp. 51–52, 2013.

[7] B. Bloom, D. Grove, B. Herta, A. Sabharwal, H. Samulowitz, and V. A. Saraswat. SatX10: A scalable plug&play parallel SAT framework - (tool presentation). In *SAT*, pp. 463–468, 2012.

[8] L. Bordeaux, Y. Hamadi, and H. Samulowitz. Experiments with massively parallel constraint solving. In *21st IJCAI*, pp. 443–448, 2009.

[9] S. Chakraborty, K. Meel, and M. Vardi. A scalable and nearly uniform generator of SAT witnesses. In *25th CAV*, pp. 608–623, July 2013.

[10] G. Chu, C. Schulte, and P. J. Stuckey. Confidence-based work stealing in parallel constraint programming. In *15th CP*, pp. 226–241, May 2009.

[11] S. Ermon, C. Gomes, A. Sabharwal, and B. Selman. Taming the curse of dimensionality: Discrete integration by hashing and optimization. In *30th ICML*, pp. 334–342, June 2013.

[12] S. Ermon, C. Gomes, A. Sabharwal, and B. Selman. Low-density parity constraints for hashing-based discrete integration. In *31st ICML*, 2014.

[13] M. Fischetti, M. Monaci, and D. Salvagnin. Self-splitting of workload in parallel computation, May 2014.

[14] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting: A new strategy for obtaining good bounds. In *21st AAAI*, pp. 54–61, Boston, MA, July 2006.

[15] Y. Hamadi and C. M. Wintersteiger. Seven challenges in parallel sat solving. In *26th AAAI*, 2012.

[16] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *14th IJCAI*, pp. 607–615, Montreal, Canada, Aug. 1995.

[17] M. Heule, O. Kullmann, S. Wieringa, and A. Biere. Cube and conquer: Guiding cdcl sat solvers by lookaheads. In *Haifa Verification Conference*, pp. 50–65, 2011.

[18] IBM ILOG. IBM ILOG CPLEX Optimization Studio 12.6, 2013.

[19] G. Katsirelos, A. Sabharwal, H. Samulowitz, and L. Simon. Resolution and parallelizability: Barriers to the efficient parallelization of sat solvers. In *27th AAAI*, 2013.

[20] S. Kumar, A. R. Mamidala, D. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burrow. PAMI: A parallel active message interface for the Blue Gene/Q supercomputer. In *IPDPS-2012: 26th IEEE International Parallel & Distributed Processing Symposium*, pp. 763–773, 2012.

[21] L. Michel, A. See, and P. V. Hentenryck. Transparent parallelization of constraint programming. *INFORMS Journal on Computing*, 21(3):363–382, 2009.

[22] T. Moisan, J. Gaudreault, and C.-G. Quimper. Parallel discrepancy-based search. In *19th CP*, vol. 8124 of *LNCS*, pp. 30–46, Uppsala, Sweden, Sept. 2013.
[23] T. Moisan, C.-G. Quimper, and J. Gaudreault. Parallel depth-bounded discrepancy search. In *11th CPAIOR*, vol. 8451 of *LNCS*, Cork, Ireland, May 2014.
[24] S. M. Plaza, I. L. Markov, and V. Bertacco. Low-latency sat solving on multicore processors with priority scheduling and xor partitioning. In *International Workshop on Logic Synthesis (IWLS)*, 2008.
[25] J.-C. Régin, M. Rezgui, and A. Malapert. Embarrassingly parallel search. In *19th CP*, vol. 8124 of *LNCS*, pp. 596–610, 2013.
[26] C. C. Rolf and K. Kuchcinski. Load-balancing methods for parallel and distributed constraint solving. In *IEEE Conf. on Cluster Computing*, pp. 304–309, Sept. 2008.
[27] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, and T. Koch. ParaSCIP – a parallel extension of SCIP. In *Competence in High Performance Computing 2010*, pp. 135–148. Springer, Feb. 2012.
[28] P. J. Stuckey. Lazy clause generation: Combining the power of SAT and CP (and MIP?) solving. In *7th CPAIOR*, pp. 5–9, 2010.
[29] L. G. Valiant and V. V. Vazirani. NP is as easy as detecting unique solutions. *Theoretical Comput. Sci.*, 47(3):85–93, 1986.
[30] P. van der Tak, M. Heule, and A. Biere. Concurrent cube-and-conquer - (poster presentation). In *SAT*, pp. 475–476, 2012.
[31] H. Zhang, M. P. Bonacina, and J. Hsiang. PSATO: a distributed propositional prover and its application to quasigroup problems. *J. Symb. Comput.*, 21(4): 543–560, 1996.
[32] L. Zhang and S. Malik. Cache performance of SAT solvers: a case study for efficient implementation of algorithms. In *6th SAT*, pp. 287–298, Santa Margherita Ligure, Italy, May 2003.