Adaptive Parallelization for Constraint Satisfaction Search

Xi Yun\(^1\) and Susan L. Epstein\(^{1,2}\)

Department of Computer Science
\(^1\)The Graduate Center and \(^2\)Hunter College of The City University of New York
New York, NY 10065 USA
xyun@gc.cuny.edu, susan.epstein@hunter.cuny.edu

Abstract
This paper introduces two adaptive paradigms that parallelize search for solutions to constraint satisfaction problems. Both are intended for any sequential solver that uses contention-oriented variable-ordering heuristics and restart strategies. Empirical results demonstrate that both paradigms improve the search performance of an underlying sequential solver, and also solve challenging problems left open after recent solver competitions.

Introduction
As parallel processing resources become increasingly available, the challenge for modern sequential constraint solvers is how best to make use of them. This paper details extensive experiments that test a variety of reasonable parallelization approaches for an underlying sequential solver (henceforward, a solver). This work uses three sets of carefully curated benchmarks from a broad variety of problem classes, and explores several reasonable premises about how parallelization might proceed. The thesis of our work is that knowledge accrued during search can substantially enhance its parallelization. We introduce two new paradigms that address knowledge sharing among processors: ELF and SPREAD. The principal result reported here is that, on particularly difficult problems, SPREAD outperforms all the tested alternatives, because it shares knowledge and balances processor workload adaptively.

Both ELF and SPREAD make two assumptions about a solver. First, it must have a variable-ordering heuristic that prioritizes contention, variables whose constraints are more likely to cause wipeouts. (We used learned variable weights (Boussemart et al., 2004), but impact (Refalo, 2004) would be an alternative.) Second, the solver extricates search from early unproductive assignments with a restart strategy (Gomes et al., 2000). Most modern sequential solvers satisfy both these conditions, and so this work is applicable to them.

For multiple processors to share knowledge as they search on the same problem, they must either share memory or pass messages to one another. Here we focus on the latter, where a manager on one processor coordinates messages among the other processors (the workers), under a popular parallel communication standard. Such a manager-worker framework can support knowledge sharing in several ways. For example, workers could repeatedly interrupt their searches to check for messages, or a manager with a deep knowledge of its workers’ behavior could choose an opportune moment to interrupt them. To preserve the integrity of the underlying sequential solver, however, our paradigms instead reflect shared knowledge in the way the manager assigns tasks to the workers.

ELF (ExpLorer and Followers) has a sequential solver (its explorer) search the full problem with restart on a designated processor, while its workers (followers) search without restart on subproblems identified based on feedback from the explorer. SPREAD (Search by Probing and REcursive Adaptive Domain-splitting) first races all its workers on the full problem with restart during a time-limited phase. Then SPREAD’s manager partitions the full problem based on contention identified thus far, and distributes subproblems to the workers with resource limits based on the search effort during the race. If a subproblem is returned unsolved within that limit, the manager further partitions it, and re-distributes the resultant subproblems with a higher search limit. This naturally directs computational power to challenging subproblems. After background and related work in the next section, we introduce a generic partition mechanism, describe ELF and SPREAD, evaluate them empirically, and discuss their advantages and limitations.
**Background and related work**

A constraint satisfaction problem (CSP) $P = \langle X, D, C \rangle$ is represented by a set of variables $X = \{X_1, \ldots, X_n\}$, each with an associated discrete domain $D = \{d_1, \ldots, d_n\}$, and a set $C$ of constraints that must be satisfied by any solution. Here, the solver does systematic backtracking that sequentially assigns values to variables and validates the consistency of each assignment. From the domains of as-yet-unassigned variables, propagation removes some values inconsistent with $C$ and current assignments. This causes a wipeout when a domain becomes empty. A level-based constraint weight assigns higher weights to constraints that lead to wipeouts more frequently at the top of the search tree. Crucial in ELF and SPREAD is a variable’s weight, the sum of the weights on the constraints in which it participates.

As massive computing resources become increasingly available, parallelization of search for solutions to CSPs becomes increasingly attractive. Research has considered a variety of platforms: a single node (Chu et al., 2009; Martins et al., 2010), a cluster (Schubert et al., 2009; Xie and Davenport, 2010), and a grid (Hyvärinen et al., 2011). It has considered various parallel models as well, including OpenMP (Chu et al., 2009; Martins et al., 2010; Michel et al., 2007), and Message Passing Interface (MPI) (Xie and Davenport, 2010; Zhang et al., 1996). MPI offers portability for applications on many processors, and has become the de facto standard for a variety of technological platforms. Here we use MPI on a cluster with 64 processors (unless otherwise specified), a number widely available on modern computing clusters and powerful workstations.

A portfolio-based parallelization method for CSP solvers (Gomes and Selman, 1997; Huberman et al., 1997) schedules a set of algorithms (its portfolio) on one or more processors in an attempt to outperform any of its constituent algorithms. This approach can prove beneficial when information is shared among processors, as when parallel SAT solvers share clauses among processors (Hamadi and Sais, 2009). Most portfolio-based methods for CSPs, however, do not share information (Bordeaux et al., 2009; Xu et al., 2008; Yun and Epstein, 2012).

Another popular parallelization method is search space splitting, which investigates different search subspaces on different processors. For SAT instances, this method usually exploits a guiding path, a sequence of variables whose assignment partitions the full search space (Zhang et al., 1996). Formally, given a binary search tree for systematic backtracking on a SAT instance, a guiding path from the root (at level 1) to any node (at level k) is a sequence of k states (\(<L_1, \delta_1>, \ldots,<L_k, \delta_k>\), where each state \(<L_i, \delta_i>\) records the truth value $L_i$ assigned to the $i$th variable on the path, and a boolean flag $\delta_i$ indicates whether both values have been attempted (e.g., Figure 1(a)). An open node ($\delta_i = 1$) is one whose left subtree is still under exploration; otherwise the node is closed ($\delta_i = 0$). Identification of a helpful guiding path is non-trivial for SAT problems; variables that often appear in learned clauses have proved effective (Chu et al., 2009; Martins et al., 2010). Iterative partitioning with clause learning, where subproblems’ search spaces may not be mutually exclusive, can also be surprisingly efficient (Hyvärinen et al., 2011).

With non-boolean domains, search space splitting for CSPs is generally more complex. For a single processor, extraction of networks (EN) iteratively splits the domains of a set of variables in a predetermined order (Mehta et al., 2009). Such a split on the $i$th variable, as in Figure 1(b), produces two subproblems that differ only in the variable’s domain; one has the same domain as the $i$th variable, and the other has the remainder. EN can extract visited search spaces to avoid duplicate search after restart.

Other parallelization schemes include a SAT solver that uses a portfolio phase followed by a splitting phase (Martins et al., 2010) without re-partitioning the problem as we do here. Additional related work includes problem decomposition (Singer and Monnet, 2007) and collaborative search (Segre et al., 2002; Vander-Swalmen et al., 2009). Other workload-balancing approaches include work stealing (Chu et al., 2009; Jurkowski et al., 2001; Michel et al., 2007; Schubert et al., 2009) and work sharing (Xie and Davenport, 2010).

![Fig. 1.](image)

Fig. 1. (a) A guiding path with open nodes at $X_1$, $X_2$, and $X_4$. (b) Extraction of subproblem $P_2$ from $P_1$ (under variables $X$, $Y$, $Z$, in that order) produces subproblems $R_1$, $R_2$, $R_3$. Together the search spaces for $P_2$, $R_3$, $R_2$, and $R_1$ partition the search space for $P_1$.

![Fig. 2.](image)

Fig. 2. Bisection partitioning on $X$ and $Y$ creates a virtual binary search tree with bit representations for the four subproblems.
Bisection partitioning

This section introduces iterative bisection partitioning, an extension of the way that a guiding path splits a SAT problem and EN splits a CSP. A bisection partition (BP) on decision variable \( X \) with domain \(d \) replaces \( X \) with two variables \( X' \) and \( X'' \) whose respective domains, \(d'\) and \(d''\), partition \(d\). To generate subproblems with search spaces of similar sizes, we adopt an (almost) even partitioning:

\[
d' = \{v_1, \ldots, v_{\lfloor d/2 \rfloor}\} \quad \text{and} \quad d'' = \{v_{\lfloor d/2 \rfloor +1}, \ldots, v_d\}, \quad \text{where} \quad \chi = \lceil d/2 \rceil.
\]

Given an ordered sequence of \( v \) decision variables, iterative bisection partitioning (IBP) repeats BP on them to generate \(2^v\) subproblems. Figure 2 illustrates how IBP on variables \(X\) and \(Y\) in \(P_1\) generates the subproblems \(R_1, R_2, R_3,\) and \(R_4\). Intuitively, processing those subproblems on different processors in parallel could improve overall search performance on \(P_1\).

IBP is a version of EN that creates subproblems with solution spaces of comparable sizes (for load balance) on variables with high (contention-based) weights as defined in the previous section. Moreover, since IBP splits any search space much the way a guiding path splits \(\{0,1\}\) for SAT problems, an IBP-generated subproblem can be represented by a guiding path \((L_1, \delta_1)\ldots (L_i, \delta_i)\), where \(L_i\) indicates whether the \(i\)th decision variable \(X_i\) is associated with \(d'\) \((L_i = 0)\) or with \(d''\) \((L_i = 1)\). This simplified (bit string) representation can reduce the communication effort required to pass subproblems to workers.

ELF

Given problem \(P\), ELF uses a manager on a dedicated processor to race an explorer on a second processor against a set of followers, each on its own processor. The explorer uses a sequential solver to search for a solution to \(P\). On each restart, the explorer also reports to the manager its most recent variable weights and restart resource cutoff (in number of backtracks). Whenever all the followers are idle (have completed their assigned subproblems), the manager partitions \(P\) with IBP on the highest-weight variables, and distributes the subproblems to the followers, along with the explorer’s last reported resource cutoff and constraint weights. Each follower then initializes the weights of the variables in its new subproblem accordingly, and searches under the cutoff as instructed by the manager.

Intuitively, ELF discretizes search by multiple rounds of restarts, during which the explorer and the followers race to find a solution, while the manager and the followers exploit the explorer’s reports to split the search space and to search. The race terminates when the explorer finds a solution, a follower finds a solution, or the followers together prove all the subproblems unsatisfiable. We emphasize, however, that relatively equal search space sizes do not at all promise relatively equal search effort.

ELF’s architecture is similar to collaborative search by nagging, a scalable fault-tolerant scheme for distributed search (Segre et al., 2002). Nagging has a manager that performs a standard search algorithm (as ELF’s explorer does) and one or more naggers (i.e., assistants) that execute the same algorithm on transformed problems or subproblems. The most significant difference between ELF and nagging is that, with restart, ELF executes search space splitting on dynamically changing search trees, but nagging uses an algorithm portfolio to transform the problem (e.g., reorder the unassigned variables) in a static search tree. A nagger can also force the manager to backtrack if the nagger proves its subproblem unsatisfiable more quickly.

ELF’s manager could partition the problem several ways; we use IBP because it is convenient and uniform for finite domains. Rather than force the followers to check repeatedly for messages, the manager waits until they are all idle. Because there is no guarantee that every subproblem will be equally difficult (i.e., consume backtracks at an equal rate), some workers may become idle earlier. To provide followers with the explorer’s most current information, ELF’s manager uses a discount factor to reduce the explorer’s reported resource cutoff before passing it on to the followers. A somewhat lower resource allocation makes it more likely that the followers will finish before the explorer’s next restart.

This synchronization of the explorer with the followers can be imperfect. An alternative would have the manager accumulate the explorer’s information in MPI message buffers and proceed asynchronously, with workers subject to different weights and cutoffs. As the weights stabilize and the cutoff becomes large, this could prove a better choice. As reported here, however, some of ELF’s workers are likely to wait for their next assignment until the others are finished. That relatively weak ability to balance workload motivated the development of SPREAD.

SPREAD

Under a manager-worker framework, SPREAD combines and extends two effective methods from SAT to parallelize search for CSPs: an algorithm portfolio with search space splitting (Martins et al., 2010), and iterative partitioning (Hyvärinen et al., 2011). Given CSP \(P\) with portfolio resource limit \(l\) and restart schedule policy, SPREAD tries to solve \(P\) under \(l\) during a portfolio phase (Algorithm 1), and then uses recursive splitting with IBP (RS-IBP) to exploit information from that phase during a splitting phase under policy (Algorithm 3). Algorithm 2 controls the \(w\) workers.

Algorithm portfolio phase. In Algorithm 1, the manager sends each worker the signal 0 to initiate a portfolio phase (line 2). On receipt of that message, each worker attempts to solve \(P\) within search limit \(l\) with a different random
Algorithm 1: Portfolio (Manager)
\textbf{Input:} $P$, policy
\textbf{Output:} variable weights and backtrack counts
1: $\text{tmp\_weights} \leftarrow 0$, $\text{bt\_num} \leftarrow 0$, signal $\leftarrow 0$
2: \textbf{for} $i = 1$ to $w$ \textbf{do} $\text{MPI\_Send(signal, } i)$
3: \textbf{while} $i > 0$ \textbf{do}
4: $\text{MPI\_Recv(<tmp\_weights, bt\_num>)}$
5: $i \leftarrow i - 1$
6: Compute weights and base by average based on all received $\text{tmp\_weights}$ and $\text{bt\_num}$
7: \textbf{return} $<\text{weights, base}>$

split seed, a kind of weak algorithm portfolio. If a worker succeeds (i.e., finds a solution or proves $P$ unsatisfiable), it reports that immediately and terminates the MPI environment, including execution on all other workers (Algorithm 2, line 6). Otherwise, the worker has exhausted $i$, and reports to the manager the variable weights it has learned and how many backtracks it used (Algorithm 2, line 7). Finally, the manager averages across all workers the variable weights and backtrack counts, and passes them on to the search space splitting phase (Algorithm 1, lines 6-7).

Splitting phase. Intuitively, RS-IBP attempts to balance the workload assigned to multiprocessors when a CSP is partitioned (Algorithm 3). RS-IBP maintains a queue $Q$ of subproblems to search (each as a bit string for the partition that gave rise to it), with their corresponding backtrack limits in queue $L$. RS-IBP determines the number $v$ of initial decision variables based on $w$ (here, the smallest number such that $2^v \geq 2w$, line 1). It then chooses the $v$ variables with the highest average weights learned for $P$ during the portfolio phase. The manager partitions $P$ on those variables, tracks the resultant subproblems and their respective backtrack limits (lines 3-4), and then distributes subproblems to workers with backtrack limits and variable weights (line 6). Task distribution (line 6) notifies the worker on processor $i$ with signal 1 that it is about to send a subproblem, and then dequeues and sends the first subproblem on $Q$ with its corresponding weights and backtrack limit from $L$. The manager then awaits its workers' feedback.

In this phase too, a worker has three possible behaviors. It reports any detected solution to the manager and terminates the MPI environment (Algorithm 2, line 12); it reports that its subproblem is unsatisfiable with message 0 (line 13); or it reports that it has exhausted its resources and returns its subproblem with message 1 (line 14). When a subproblem is returned and $Q$ has fewer than $2^v$ subproblems, the manager re-partitions the returned subproblem on $\xi$ additional decision variables, and enqueues the resultant subproblems and resource limits (Algorithm 3, lines 14-18). Otherwise, the manager re-enqueues the subproblem without repartitioning (Algorithm 3, lines 20-22). As they become available, the manager also distributes subproblems to any idle workers (Algorithm 3, lines 11 and 23-25). RS-IBP terminates when some worker finds a solution, or when all subproblems are proved unsatisfiable. When eventually distributed, an unresolved subproblem that returned to the queue intact will almost certainly begin with a different random seed, and may therefore have a very different search experience. $\xi$ is chosen so as to bound
the RS-IBP queue length by $2^v + 2^{v+1} - 1$ for $v$ initial decision variables. Nonetheless, IBP’s relatively short bit strings would in practice allow large queues. Indeed, in the work reported next, the average number of generated problems in every problem set was always under 200.

**Experimental design and results**

All experiments ran on a Cray XE6m. Each of its 160 dual-socket compute nodes contains two 8-core AMD Magny-Cours processors running at 2.3 GHz. (Here, a processor corresponds to a core of the Cray.) The solver was Mistral-1.331 (C++ source code, (Competition, 2008)), chosen for its compatibility with MPI on the Cray. This allows us to assemble sets of difficult problems and to evaluate the performance improvement by SPREAD. Note that, on a Cray XE6m, Mistral compiled under the GCC compiler (Mistral-GCC) runs 2 to 3 times faster than under the CC compiler (Mistral-CC). This gives the Mistral-GCC benchmark a considerable advantage over all our parallel solvers, which require the CC compiler for MPI.

Our experiments evaluate parallel approaches on three sets of difficult problems. From the repository of over 7000 problems in the two most recent CSP solver competitions (Competition, 2009; Competition, 2008), we selected 51 representative classes. (See Figure 3.) They cover a broad variety of CSPs, including binary and non-binary problems, both real-world and artificial. To avoid bias toward classes with many problems, we stratified selection by class to re-assemble sets of difficult problems and to evaluate the performance improvement by SPREAD. Note that, on a Cray XE6m, Mistral compiled under the GCC compiler (Mistral-GCC) runs 2 to 3 times faster than under the CC compiler (Mistral-CC). This gives the Mistral-GCC benchmark a considerable advantage over all our parallel solvers, which require the CC compiler for MPI.

Our experiments evaluate parallel approaches on three sets of difficult problems. From the repository of over 7000 problems in the two most recent CSP solver competitions (Competition, 2009; Competition, 2008), we selected 51 representative classes. (See Figure 3.) They cover a broad variety of CSPs, including binary and non-binary problems, both real-world and artificial. To avoid bias toward classes with many problems, we stratified selection by class to reflect any pre-specified subclasses and naming conventions, and chose a subset from each class in proportion to the original class sizes. This resulted in 1765 CSPs in 51 classes of 7 to 65 problems each. Next, we solved each of the 1765 problems with Mistral-GCC, eliminated any that Mistral-GCC could solve in less than a minute (1398 in all), and then partitioned those remaining into two sets. The hard set is the 119 CSPs that Mistral-GCC could solve within 1 to 30 minutes each, running sequentially on the Cray. The harder set is the 248 CSPs that remained. Finally, the challenge set is the 133 competition problems never solved by any solver within 30 minutes during the same competitions, and not already present in the harder set.

In addition to ELF, SPREAD, and Mistral, we tested:

- **NR** (Naïve Random), a naïve random algorithm portfolio that races 63 copies of Mistral with random seeds.
- **NV** (Naïve Variable), an algorithm portfolio that races 63 copies of Mistral, each of which randomly selects and orders the first 3 variables it assigns (but not their values) and retains them on every restart (Bordeaux et al., 2009).
- **PP** (Parallel Portfolio), an algorithm portfolio that races in parallel 63 combinations of heuristics and restart strategies. Candidate heuristics were impact, dom/wddeg, dom/wdeg, and impact/wddeg, as implemented in Mistral. Candidate restart policies were impact, arithmetic, geometric, and dynamic. (Luby-k used $k$ backtracks per unit, $k \in \{128, 256, 512, 1024, 2048, 4096\}$ (Luby et al., 1993). Geometric used restart limit $x(n) = 100p^n$ at step $n$, where $p = 1.3, 1.5$ or 2.0; arithmetic on step $n$ used $x(n) = 16000n$, $8000n$, $1000n^2$, or $500n^2$. Dynamic adaptively determined whether to execute geometric restart with exponent 1.3, 1.5, or 2.0 based on the problem formulation (e.g., number of predicates), and restarted on the minimum of 1000 and number of variables.
- **RP** (Random Partitioning) splits on 7 randomly-chosen decision variables, and assigns those 128 subproblems to 63 workers. (Some workers may process more than one.)
- **MISTRAL** (Mistral) is the 119 CSPs that Mistral-GCC could solve within 1 to 30 minutes each, running sequentially on the Cray. The harder set is the 248 CSPs that remained. The challenge set is the 133 competition problems never solved by any solver within 30 minutes during the same competitions, and not already present in the harder set.

All runs used a 30-minute per problem time limit. SPREAD used 64 processors (including one for the manager); ELF used 64 for its followers plus 2 for its explorer and manager, a slight advantage. ELF used geometric restart with base 100 and discount factor 0.8. SPREAD’s portfolio phase was 100 seconds (unless otherwise specified) with dynamic restart. Backtrack limits for the first subproblems (generated in Algorithm3, line 3) used the base from the portfolio phase. When a subproblem was further partitioned on $\xi$ more decision variables, the backtrack limit was multiplied by $(1.5)^\xi$. On all approaches, we report the median of 3 runs, as in recent parallel SAT solver competitions (SAT competition).

Figure 4 compares SPREAD’s runtime to the others for the hard problem set. Although a few instances went unsolved under SPREAD, it clearly outperformed most of the others, including the faster sequential version of Mistral, and PP (with its different combinations of search strategies). SPREAD solved 43.70% of these problems within 100 - 200 seconds, the time when SPREAD’s splitting phase on critical decision variables has just begun, while PP tries a complementary algorithm portfolio instead. This is a clear demonstration that the portfolio phase informs IBP. Moreover, NWSREAD’s pure search space splitting without that knowledge was dramatically inferior; it could not solve 75.63% of the hard problems. In contrast, even NR had solved 17.65% problems within the first 100 seconds. NWSREAD was therefore excluded from subsequent test-
ing, as were both versions of Mistral alone.

Figure 5 evaluates the remaining approaches on the harder problem set. Within 1800 seconds, SPREAD solved 56 problems (44 satisfiable), 16 more (a 40.00% improvement) than the best benchmark method PP (which solved 40), and 31 more (a 124.00% improvement) than the worst, NV (which solved 25). As one would expect, SPREAD behaved early on much like the portfolio-based methods NR and PP; it solved 10 (all satisfiable) within 100 seconds, that is, in its portfolio phase. SPREAD also consistently solved more problems that required more time, 18 of them (11 satisfiable) in the last 800 seconds.

Table 1 compares ELF, SPREAD, and the four methods from Figure 5 on 32 problems from the challenge set, those solved by at least one method. (None was solved during a portfolio phase.) Here we report on two versions of SPREAD, that differ only in their portfolio phase times: a 10-second SPREAD-10-pf, and the 100-second SPREAD-100-pf used in the remainder of this paper. SPREAD significantly outperformed the other parallelization methods, even though we had hand tuned ELF’s explorer to do well on this set (geometric restart with base 100, factor 1.3333, and discount factor 0.8). Although SPREAD did best with rand problems, it also solved varied problems from such classes as langford, crossword, and queenAttacking.

To investigate resource usage, let the idle ratio of the $i$th worker be the fraction of overall runtime that it was idle on a problem. On problems solved during its partitioning phase, SPREAD’s average idle ratio (across its processors in 3 runs) rose as high as 0.8251 on hard, 0.8793 on harder, and 0.5015 on challenge problems, probably due to high backtrack limits on extremely unbalanced search trees. Overall, however, SPREAD still often managed an idle ratio under 0.1: on 56.92% of the hard problems, 69.92% of the harder problems, and 75.00% of the challenge problems.

Finally, Figure 6 plots, as a median with cutoff 1800 seconds, SPREAD’s runtimes to solve a difficult unsatisfiable instance on different numbers of processors. Similar improvements were also observed on satisfiable problems, where search performance typically has greater variance.

**Discussion and conclusions**

ELF and SPREAD have several advantages. In the search for a single solution, they are complete, and make no assumption about domains or constraint types; as long as their solver can address a problem, they can too. Both represent search space splitting as a queue of bit strings that repre-
Several factors account for SPREAD’s ability to outperform ELF on the harder problems and the challenge problems. Although SPREAD’s search is influenced by the variables it splits on and their order, its portfolio-phase search limit also has a strong effect. Because the splitting-phase search limit is proportional to the number of backtracks in the portfolio phase, and because restarts in the portfolio phase are geometric, the longer portfolio phase in SPREAD-100-pf is more likely to produce higher resource limits during the splitting phase and possibly more idle time. Nonetheless, SPREAD-100-pf has additional portfolio time, which could produce more reliable weights. On the other hand, if weights do not provide good guidance, SPREAD-10-pf can devote that additional 90 seconds to subproblem search.

Table 1: Runtimes for different parallelization strategies on the challenge problem set. Best results in boldface.

<table>
<thead>
<tr>
<th>Problem Name</th>
<th>Satisfiable?</th>
<th>NR</th>
<th>NV</th>
<th>PP</th>
<th>RP</th>
<th>ELF</th>
<th>SPREAD-10-pf</th>
<th>SPREAD-100-pf</th>
</tr>
</thead>
<tbody>
<tr>
<td>crossword-m1-words-21-10</td>
<td>yes</td>
<td>520.47</td>
<td>-</td>
<td>-</td>
<td>721.84</td>
<td>-</td>
<td>-</td>
<td>846.01</td>
</tr>
<tr>
<td>crossword-m1c-ogd-vg10-13_ext</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>744.89</td>
<td>583.22</td>
</tr>
<tr>
<td>crossword-m1c-ogd-vg10-14_ext</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1302.41</td>
<td>402.33</td>
</tr>
<tr>
<td>crossword-m1c-ogd-vg12-12_ext</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>461.62</td>
<td>586.02</td>
</tr>
<tr>
<td>crossword-m1c-uk-vg11-12_ext</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1081.71</td>
<td>-</td>
</tr>
<tr>
<td>frb53-24-2-mgd_ext</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>1240.12</td>
<td>361.06</td>
<td>749.33</td>
<td>329.85</td>
<td></td>
</tr>
<tr>
<td>frb53-24-5_ext</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>748.17</td>
<td>331.19</td>
<td>281.01</td>
<td>63.04</td>
<td>255.86</td>
</tr>
<tr>
<td>frb56-25-2-mgd_ext</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>470.45</td>
<td>661.94</td>
<td>822.12</td>
<td></td>
</tr>
<tr>
<td>langford-2-14</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>485.81</td>
<td>-</td>
<td>187.39</td>
<td>401.30</td>
</tr>
<tr>
<td>langford-3-16</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>567.92</td>
<td>-</td>
<td>659.73</td>
<td>446.50</td>
</tr>
<tr>
<td>queenAttacking-8</td>
<td>yes</td>
<td>-</td>
<td>1065.19</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>rand-3-24-24-76-632-17_ext</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>358.80</td>
<td>430.53</td>
<td>240.02</td>
<td>326.82</td>
</tr>
<tr>
<td>rand-3-24-24-76-632-fcd-47_ext</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>823.91</td>
<td>988.58</td>
<td>693.89</td>
<td>207.30</td>
</tr>
<tr>
<td>rand-3-24-24-76-632-fcd-50_ext</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>692.31</td>
<td>410.21</td>
<td>59.71</td>
<td>168.40</td>
</tr>
<tr>
<td>rand-3-28-28-93-632-16_ext</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1452.27</td>
<td>1511.52</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>rand-3-28-28-93-632-23_ext</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1077.27</td>
<td>551.02</td>
<td>758.57</td>
<td></td>
</tr>
<tr>
<td>rand-3-28-28-93-632-25_ext</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>448.20</td>
<td>464.58</td>
<td></td>
</tr>
<tr>
<td>rand-3-28-28-93-632-3_ext</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1306.23</td>
<td>648.04</td>
<td></td>
</tr>
<tr>
<td>rand-3-28-28-93-632-30_ext</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>893.93</td>
<td>1061.22</td>
<td></td>
</tr>
<tr>
<td>rand-3-28-28-93-632-35_ext</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1186.84</td>
<td>1321.97</td>
<td></td>
</tr>
<tr>
<td>rand-3-28-28-93-632-37_ext</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>238.10</td>
<td></td>
</tr>
<tr>
<td>rand-3-28-28-93-632-8_ext</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1126.08</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>rand-3-28-28-93-632-fcd-16_ext</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1531.64</td>
<td>530.76</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>rand-3-28-28-93-632-fcd-20_ext</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>24.79</td>
<td>428.46</td>
<td>299.64</td>
<td>314.52</td>
</tr>
<tr>
<td>rand-3-28-28-93-632-fcd-21_ext</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1322.25</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>rand-3-28-28-93-632-fcd-24_ext</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1122.49</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>rand-3-28-28-93-632-fcd-27_ext</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1349.44</td>
<td></td>
</tr>
<tr>
<td>rand-3-28-28-93-632-fcd-31_ext</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>700.22</td>
<td>211.54</td>
<td></td>
</tr>
<tr>
<td>rand-3-28-28-93-632-fcd-35_ext</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>494.40</td>
<td>616.61</td>
<td></td>
</tr>
<tr>
<td>rand-3-28-28-93-632-fcd-40_ext</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>202.85</td>
<td>137.29</td>
<td>219.16</td>
<td></td>
</tr>
<tr>
<td>rand-3-28-28-93-632-fcd-42_ext</td>
<td>yes</td>
<td>-</td>
<td>1618.35</td>
<td>-</td>
<td>120.04</td>
<td>144.94</td>
<td>124.55</td>
<td></td>
</tr>
<tr>
<td>rand-3-28-28-93-632-fcd-46_ext</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1410.23</td>
<td>199.73</td>
<td>168.30</td>
<td>159.21</td>
</tr>
</tbody>
</table>
er’s own search algorithm.

To split a search space, ELF and SPREAD use IBP, which has no knowledge about problem domains. One could, however, exploit domain characteristics, for example, partition a critical variable’s large domain into more subproblems, rather than bisect it, or group particular values into split domains rather than simply split at the median. In contrast, for extremely small domains (e.g., binary in SAT), parity constraints could lead to more balanced and effective partitioning than IBP (Bordeaux et al., 2009).

RS-IBP must first fail on a subproblem to partition it; that first effort on it is wasted. Moreover, because RS-IBP initially depends on data from the portfolio phase, it can be misled. For example, in the queens-knights (QK) problems, the knight variables are the true contention, but weights initially prefer the queen variables. SPREAD’s portfolio phase is similarly misled, so RS-IBP partitions on queen variables, and the results prove unsatisfactory. Indeed, in a test run, SPREAD with a portfolio time of 10 seconds failed on all five QK problems in the hard problem set. A longer (100-second) portfolio phase however, provided more accurate weights, and SPREAD managed to solve 2 of them.

Both ELF and SPREAD are adaptive; ELF learns weights and cutoffs for its workers from its explorer, while SPREAD iteratively partitions hard subproblems. Under RS-IBP, however, SPREAD gradually allocates more computing cycles to the hard parts of a problem, and thereby balances workload assignments for processors better. Current work includes a dynamically-chosen duration for the portfolio phase, and investigation of methods that maintain the decision path and change only the variables that further partition a returned subproblem. Future work will also combine adaptive decision variable selection with nogood learning.

Meanwhile, SPREAD provides a well-tested and promising paradigm for parallel search on CSPs. SPREAD is compatible with the widely-used MPI communication standard and requires no alteration of its underlying solver. SPREAD combines and extends methods for workload partitioning and balancing from SAT to CSP solution. This paper confirms its ability to conveniently and efficiently parallelize a modern sequential CSP solver.

Acknowledgements

This work was supported by the National Science Foundation under IIS-0811437, CNS-0958379 and CNS-0855217, and CUNY’s High Performance Computing Center.

References


