A Hybrid Paradigm for Adaptive Parallel Search

Xi Yun[1] and Susan L. Epstein[1,2]

[1] Department of Computer Science,
The Graduate Center of the City University of New York, New York, NY 10016, USA
[2] Department of Computer Science,
Hunter College of the City University of New York, New York, NY 10065, USA
xyun@gc.cuny.edu, susan.epstein@hunter.cuny.edu

Abstract. Parallelization offers the opportunity to accelerate search on constraint satisfaction problems. To parallelize a sequential solver under a popular message passing protocol, the new paradigm described here combines portfolio-based methods and search space splitting. To split effectively and to balance processor workload, this paradigm adaptively exploits knowledge acquired during search and allocates additional resources to the most difficult parts of a problem. Extensive experiments in a parallel environment show that this paradigm significantly improves the performance of an underlying sequential solver, outperforms more naive approaches to parallelization, and solves many difficult problems left open after recent solver competitions.

1 Introduction

Spread (Search by Probing and REcursive Adaptive Domain-splitting) is an adaptive paradigm that harnesses parallel computation to enhance an underlying sequential constraint solver (henceforward, a *solver*). Because Spread does not alter its solver, only minimal programming for message passing is required for use with modern solvers. Our thesis is that, on difficult problems, parallelization that combines efficient task assignment with effective exploitation of information can significantly improve performance. The principal results reported here are that Spread significantly improves the performance of its solver, outperforms a variety of reasonable alternatives, and solves many difficult constraint satisfaction problems left open after recent solver competitions.

Spread makes only two assumptions about its solver. First, the solver directs search with a variable-ordering heuristic toward *contention*, variables whose constraints are more likely to cause wipeout [1]. (Here, we used learned variable weights [2], but variable impact would be an alternative [3].) Second, the solver uses a restart strategy to extricate search from early unproductive assignments [4]. Most modern solvers satisfy both conditions.

Spread uses a *manager-worker* framework, where a *manager* assigns tasks and coordinates messages among all the other processors (the *workers*). Spread has two phases: a time-limited portfolio phase followed by a splitting phase. In the *portfolio phase*, Spread's multiple workers search in parallel from random

M. Milano (Ed.): CP 2012, LNCS 7514, pp. 720–734, 2012.

seeds; if any worker reports a solution or proves that there is none, the problem is solved. Otherwise, once the portfolio phase exhausts its time allocation, SPREAD begins its *splitting phase*, where the manager partitions the original problem into subproblems based on weights learned thus far. The manager distributes the subproblems to the workers with search limits based on the search effort during the portfolio phase. If any worker reports a solution, or if all the subproblems are proved to have no solution, the problem is solved. Any subproblem returned unsolved to the manager undergoes further partitioning. Those new subproblems are enqueued and eventually re-distributed with larger search limits as workers become available. This recursive partitioning mechanism naturally directs computational power to difficult subproblems.

SPREAD facilitates parallelization. It accepts any constraint supported by its solver. To partition problems, SPREAD manipulates domains rather than constraints, so that users need not learn propagators provided by the solver or implement new ones. Because its domain splitting method is general, SPREAD could be extended to continuous variable domains. SPREAD's portfolio phase solves easy problems quickly and stably. For more difficult problems, the portfolio phase also learns weights that determine how the manager in the subsequent splitting phase generates subproblems. Moreover, workers can exploit those same weights during their search on subproblems. In practice, the variables used to generate subproblems can be statically chosen before the splitting phase (*SPREAD-S*), or determined dynamically from weights learned during search on the corresponding subproblem (*SPREAD-D*). (For clarity, we refer to the paradigm here as SPREAD, and the individual implementations as SPREAD-S and SPREAD-D.) After relevant background and related work in the next section, we describe SPREAD, offer some reasonable alternatives, evaluate SPREAD-S and SPREAD-D against them, and discuss their advantages and limitations.

2 Background and Related Work

A constraint satisfaction problem (*CSP*) $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ is defined by a set of variables $\mathcal{X} = \{X_1, ..., X_n\}$, each with an associated domain $\mathcal{D} = \{d_1, ..., d_n\}$, and a set of constraints $\mathcal{C} = \{c_1, ..., c_m\}$. A *solution* to P assigns a value to each variable in $\mathcal{X}$ from its respective domain so that it satisfies $\mathcal{C}$. If P has a solution, it is *satisfiable*; otherwise it is *unsatisfiable*. The solver here is assumed complete; it executes systematic backtracking, traditionally envisioned as a search tree. There, after each value assignment, inference removes from the domains of the as-yet-unbound variables all values that it shows inconsistent with $\mathcal{C}$. If a domain becomes empty (a *wipeout*), *weight learning* increases the weight of the constraint that removed the last value. Once search stops, the weight of a variable is taken as the sum of the weights on the constraints that restrict it [2].

Parallelization seeks to exploit the massive computing resources increasingly available on multicore computers, and in clusters, grids, and clouds. Research on parallelization for CSP solvers includes a broad spectrum of parallel programming models (e.g., OpenMP [5,6,7], Message Passing Interface (*MPI*) [8,9]) and

a variety of platforms (e.g., single node [5,7], cluster [9,10], and grid [11]), on a scale from a few processors to thousands. In particular, MPI is intended for high-performance parallel computing on platforms without shared memory. Its convenience and portability have made it the *de facto* standard for a variety of technological platforms. This work uses MPI on a cluster and executes extensive experiments on up to 256 processors, a number widely available in modern computing environments.

Search space splitting can explore different search subspaces on different processors. For SAT instances, with their boolean domains, search space splitting usually relies on a guiding path (e.g., Fig. 1(a)). A boolean flag δ_i indicates whether a node is *closed* (both values attempted, $\delta_i = 0$, black circle) or *open* (one value attempted, $\delta_i = 1$, white circle) [8]. Although identification of a helpful guiding path is non-trivial (as in [5]), variables with particular properties have proved effective for splitting [7]. Iterative partitioning with clause learning, where search spaces of SAT subproblems may overlap, can also be an effective strategy [11].

Given non-boolean domains and various kinds of constraints, search space splitting for CSPs becomes more complex. A SAT solver can conveniently partition its search space by adding new clauses (e.g., parity constraints [12]), without any modification to its search strategy or propagation methods. A CSP solver that tries to add new constraints to split a search space, however, might confront constraints it could not directly process. Even splitting only with already existing constraints (as in [13]) might have to contend with different formulations and different models for the same problem under different solvers. Partitioning by domain manipulation avoids such difficulties. One approach, *network extraction* (NE), performs a sequence of domain splits on a subset of $\mathcal{X}$ under a given variable ordering for a single processor [14], as in Fig. 1(b). A split on the ith variable produces subproblems P_i^1 and P_i^2 that differ only in the ith variable's

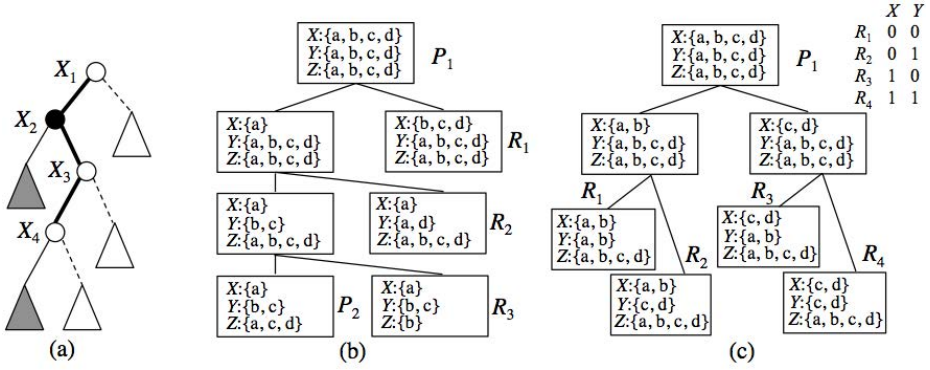

Fig. 1. (a) A guiding path with open nodes at X_1, X_3, and X_4. (b) Extraction of subproblem P_2 from P_1 (under variable order X, Y, Z) produces subproblems R_1, R_2, R_3. (c) Iterative bisection partitioning on X and Y creates a virtual binary search tree of subproblems, shown with their bit-string representations.

domain: P_i^1 has some values for the ith variable, and P_i^2 has the rest. NE was developed to avoid duplicate search on visited search spaces after restart [14].

Another prevalent parallelization approach for CSP solvers uses an algorithm portfolio [15]. A *portfolio-based* method schedules a set of algorithms (its $portfolio$) on one or more processors, hoping to outperform any of its constituent algorithms [12,16]. Although this approach can benefit from information shared among processors, as when parallel SAT solvers share clauses [17], most portfolio-based methods for CSPs do not share information [12].

Additional parallelization methods include various workload-balancing mechanisms, such as work stealing [5,6,10] and work sharing [9]. The SAT-solver parallelization methods most relevant to SPREAD are described in [7] and [11]. The first passes information from a portfolio phase to a subsequent splitting phase for effective partitioning; the second iteratively partitions a SAT problem with learned clauses. SPREAD combines and extends them to parallelize adaptive search for CSPs. (An earlier version of SPREAD appeared as a modification to an explore-and-follow parallelization paradigm [18].) The CSP work most relevant to SPREAD is [13]. SPREAD, however, better utilizes its computing resources initially, splits the search space by domain manipulation, avoids nogood learning, and proves convenient for systematic experimental evaluation.

3 The SPREAD Paradigm

SPREAD uses its manager to partition and distribute tasks, and leaves search entirely to its workers. Each worker executes the solver exactly as it would on a single processor, but may receive different parameter values from the manager. SPREAD starts with a weak portfolio, where different workers execute the same solver from different random seeds on the full problem. The subsequent splitting phase formulates subproblems, and recursively partitions those that prove difficult to direct additional computing resources to them.

Given problem P with search limit ℓ and restart schedule *policy*, the SPREAD-S manager executes the portfolio phase with Algorithm 1 on workers under the control of Algorithm 2. The manager then uses Recursive Splitting with Iterative Bisection Partitioning (*RS-IBP*) during the splitting phase with Algorithm 3. As is traditional in MPI, the manager executes on processor 0, and the other k processors are workers. We extend SPREAD-S to SPREAD-D at the end of this section.

Portfolio Phase. To begin, the manager sends the initialization signal 0 to each of the k workers (Algorithm 1, lines 2-4). On receipt of that message, workers (Algorithm 2) execute a weak algorithm portfolio, attempting to solve P within ℓ with different random seeds. Any proof of either P's satisfiability or unsatisfiability within ℓ leads to an immediate report as well as the termination of the MPI environment, including execution on all workers (Algorithm 2, lines 5-6). Otherwise, worker i has exhausted ℓ, and reports to the manager its learned weights w_l and backtrack count b_l (Algorithm 2, line 8). The manager receives w_p and b_p from worker p (Algorithm 1, line 6), possibly in non-numerical order.

Finally, the manager forwards to the splitting phase *weights*, the average of the variable weights w_i received from all the workers, and *base*, the average of the backtrack counts b_i received from them, where they will be used to guide search on the subproblems (Algorithm 1, lines 8-9).

Algorithm 1 Portfolio (Manager)
Input: P, *policy*
Output: weights, backtrack counts
1: $signal \leftarrow 0$
2: **for** $i = 1$ to k
3: $w_i \leftarrow 0$, $b_i \leftarrow 0$
4: MPI.Send($signal$, i)
5: **while** $i > 0$
6: MPI.Recv($\langle w_p, b_p \rangle$, p)
7: $i \leftarrow i - 1$
8: Compute *weights* from all w_i and *base* from all b_i
9: **return** $\langle weights, base \rangle$

Algorithm 2 Worker
Input: P, ℓ, *policy*
Output: result of search on P
1: **while** TRUE
2: MPI.Recv($signal$, 0)
3: **if** $signal$ = -1 **break**
4: **if** $signal = 0$ // portfolio phase
5: **if** solve(P, ℓ, *policy*, *rand_seed*)
6: Output result and abort MPI
7: **else**
8: MPI.Send($\langle w_l, b_l \rangle$, 0)
9: **else** // splitting phase
10: MPI.Recv($\langle P^S, \ell_r$,*weights*$\rangle$,0)
11: Initialize variable weights of P^S
12: **if** solve($P^S, \ell_r, policy, rand\_seed$)
13: **if** P^S is satisfiable
14: Output *sol* and abort MPI
15: **else do** MPI.Send($\langle 0, P^S \rangle$, 0)
16: **else do** MPI.Send($\langle 1, P^S \rangle$, 0)

Algorithm 3 RS-IBP (Manager)
Input: P, weights w, *base*, *policy*
Output: solution of P
1: $v \leftarrow get\_splitting\_number(k)$
2: $threshold \leftarrow compute\_threshold(v)$
3: $splits \leftarrow choose(v, P, w)$
4: **for** P^S in IBP(P, *splits*)
5: Q.push(P^S)
6: **for** $i = 1$ to 2^v **do** L.push(*base*)
7: $\#enqueued \leftarrow Q.size()$
8: $\#feedback \leftarrow 0$, $signal \leftarrow 1$
9: **for** $i = 1$ to k **do** $distribute(i)$
10: **while** $\#feedback < \#enqueued$
11: MPI.Recv($\langle feedback, P^S \rangle$, p)
12: $\#feedback \leftarrow \#enqueued + 1$
13: **if** $feedback = 0$
14: **if** !$Q.empty()$
15: $distribute(p)$
16: **else**
17: **if** $Q.size() < threshold$
18: $splits \leftarrow choose(\xi, P^S, w)$
19: **for** P_i^S in IBP(P^S, *splits*)
20: Q.push(P_i^S)
21: L.push($get\_limit(P_i^S, base)$)
22: $\#enqueued \leftarrow \#enqueued+1$
23: **else**
24: Q.push(P^S)
25: L.push($get\_limit(P^S, base)$)
26: $\#enqueued \leftarrow \#enqueued+1$
27: **for** $i = 1$ to k
28: **if** i is idle && !$Q.empty()$
29: $distribute(i)$
30: Send $signal$ -1 to all processors

A portfolio-based method that shares information must address the trade-off between diversification and intensification [19]. Diversification uses dramatically different search strategies, and expects its searchers to proceed independently. In contrast, intensification explores with relatively small variations around a single strategy, and expects to share the information it gathers among all its searchers. Because SPREAD is intended to solve difficult CSPs, it does

intensification in its portfolio phase, as recommended in [19]. Indeed, the primary purpose of SPREAD's portfolio phase is to glean information to support search space splitting, not to solve P.

Iterative Bisection Partitioning. A *bisection partition* (BP) on variable X_i with domain d_i replaces X_i with two variables, X_i' and X_i'', whose respective domains d_i' and d_i'' partition d_i. To generate subproblems with search spaces that may have similar sizes, without bias toward particular domain values, we adopt an (almost) even bisection partition where $d_i' = \{v_1, ..., v_\chi\}$, $d_i'' = \{v_{\chi+1}, ..., v_{|d|}\}$, and $\chi = \lceil |d|/2 \rceil$. *Iterative bisection partitioning* (IBP) repeats BP on v ordered *splitting variables* to generate 2^v subproblems. Fig. 1 (c) illustrates IBP on variables X and Y of P_1 to generate subproblems R_1, R_2, R_3, and R_4. Intuitively, overall search performance on P_1 could be improved by processing such subproblems on different processors in parallel.

In SPREAD, the manager chooses as splitting variables those with the highest weights. This conserves the promising variable ordering already found effective in the portfolio phase by a solver that exploits those weights (e.g., variable-ordering heuristic *dom/wdeg* [2]). Moreover, since IBP splits domains of CSP instances much the way a guiding path splits {0,1} for SAT problems, an IBP-generated subproblem can analogously be represented by a guiding path, where L_i indicates whether the ith splitting variable X_i is associated with d_i' ($L_i = 0$) or with d_i'' ($L_i = 1$). (See Fig. 1(c).) This simple bit-string representation reduces the communication effort required to pass subproblems to workers.

Splitting Phase. In its splitting phase, SPREAD recursively splits the search space with IBP (Algorithm 3). Initially, the manager partitions P into several subproblems, each represented as a bit string for the partition that gave rise to it, and allocates *base* backtracks to each one. Subproblems and their backtrack limits are maintained in queues $\mathcal{Q}$ and $\mathcal{L}$, respectively. For k workers, the manager determines how many initial splitting variables to use (here, $v = \lceil \log_2 2k \rceil$), computes the queue length *threshold* (here, 2^v), and then chooses *splits*, the v variables with the highest weights in *weights* learned for P during the portfolio phase (lines 1-3). Next the manager partitions P on *splits* in descending order of weight, and tracks the resultant subproblems and their respective backtrack limits (lines 4-6). Before it distributes subproblems to workers with backtrack limits and variable weights (line 9), the manager notifies the ith worker with signal 1 that it is about to do so. The manager then dequeues and sends the first k subproblems on $\mathcal{Q}$ with their corresponding weights and backtrack limits from $\mathcal{L}$, and awaits feedback.

As in the portfolio phase, a worker immediately reports any detected solution to the manager, and terminates the MPI environment (Algorithm 2, line 14). If a worker proves its subproblem P^S unsatisfiable, however, it notifies the manager with message 0 (Algorithm 2, line 15). The manager replies with a new subproblem from $\mathcal{Q}$ (if any is waiting, Algorithm 3, lines 14-15). Otherwise, the worker has exhausted its resources ℓ_r and returns its subproblem to the manager with message 1 (Algorithm 2, line 16). If the subproblem queue has fewer

than *threshold* subproblems, the manager recursively partitions the returned subproblem on new splitting variables, and enqueues the resultant subproblems with their resource limits (Algorithm 3, lines 18-22). If there is insufficient space on the queue to repartition the subproblem, the manager re-enqueues it as it was, but with a larger resource limit (Algorithm 3, lines 24-26). Whether or not it repartitions returned subproblems, the manager continues to distribute subproblems from $\mathcal{Q}$ to any idle worker (Algorithm 3, lines 27-29). RS-IBP terminates when some worker finds a solution, or when all subproblems are proved unsatisfiable.

When eventually distributed, an unresolved subproblem (even without repartitioning) will break ties with a random seed, and may therefore have a different search experience. To bound the size of $\mathcal{Q}$, for each split, SPREAD-S here chooses ξ as $\max\{\lceil \log_2(threshold - \mathcal{Q}.size()) \rceil, 1\}$ (Algorithm 3, line 18). This bounds the length of $\mathcal{Q}$ at $2^v + 2^{v-1} - 1$, which happens only when an unresolved subproblem confronts a queue of length $2^{v-1} - 1$. Nonetheless, IBP's concise bit-string representation makes it space-efficient, and in practice allows large queues.

Spread-D. SPREAD-S always uses the same splitting variables in the same order, determined by the weights first learned during its portfolio phase. Intuitively, for a returned subproblem, it could be more accurate to determine splitting variables dynamically, with weights learned during search on that subproblem. SPREAD-D is an extension of SPREAD-S that dynamically chooses its splitting variables. When a SPREAD-D worker fails to solve a subproblem within the allocated resource, it returns to the manager both the subproblem and the weights learned on it (i.e., received initially from the manager and modified during this search). This requires modification of only Algorithm 2, line 16 and Algorithm 3, line 11. The manager then chooses, in line 18, additional splitting variables with the highest weights acquired during search on the returned subproblem. Because SPREAD-D never changes *splits*, which originally designated the subproblem returned in line 2, it guarantees mutually exclusive subproblems.

In the portfolio phase, SPREAD-S and SPREAD-D terminate only when some worker finds a solution or proves the problem unsatisfiable. Otherwise they enter the splitting phase where, without a search limit, they terminate only when a solution is found or all subproblems are proved unsatisfiable. SPREAD is complete, because IBP generates subproblems with mutually exclusive, collectively exhaustive search spaces, and a subproblem is always partitioned or receives larger search limits. Section 5 demonstrates that SPREAD is also effective.

4 Experimental Design

The experiments reported here evaluate parallelization methods on their ability to solve both problems difficult for the underlying solver and problems difficult for all the solvers in the two most recent international CSP solver competitions [20,21]. From the repository of more than 7000 problems in those competitions, we selected 51 representative classes that cover a broad variety of CSPs with relatively uniform population distribution, shown in Fig. 2. To avoid any bias

toward large classes, we stratified selection from each class to reflect any prespecified subclasses and naming conventions, and chose a subset from each class in proportion to its original subclass sizes. This produced 1765 problems in classes of sizes from 7 to 65.

The experimental platform was a Cray XE6m system with 160 dual-socket compute nodes. Each node contains two 8-core AMD Magny-Cours processors running at 2.3 GHz. (Here a SPREAD processor corresponds to a Cray core.) Without a readily-available parallel CSP solver as a benchmark, this paper compares the performance of SPREAD-S and SPREAD-D to a variety of parallelization methods inspired by relevant work. We chose to parallelize the solver Mistral-1.331 (with C++ source code from [20]) because it is compatible with MPI on the Cray, and allows us to curate sets of difficult problems from recent CSP solver competitions and to evaluate the performance improvement under SPREAD-S and SPREAD-D. Mistral can be compiled to run sequentially on the Cray XE6m either under the GCC compiler (*Mistral-GCC*) or the CC compiler (*Mistral-CC*), but Mistral-GCC runs about two to three times faster than Mistral-CC. This gives the Mistral-GCC benchmark a considerable advantage over all our parallel solvers, which require the CC compiler for MPI.

We solved each of the 1765 problems with Mistral-GCC, and eliminated the 1398 problems solved by Mistral-GCC in less than one minute. The 119 that could be solved by sequential Mistral-GCC within 1 to 30 minutes on the Cray became the *hard set*; the remaining 248 became the *harder set*. Finally, the *challenge set* consists of the 133 problems never solved by any solver within 30 minutes in either competition, and not already included in the harder set.

We tested Mistral-GCC and Mistral-CC alone, as well as SPREAD-S and SPREAD-D with Mistral-CC, and the following parallelization approaches:

- *Naive Random* (NR) races 63 copies of Mistral-CC with random seeds.
- *Parallel Portfolio* (PP) races 63 combinations of heuristics and restart policies. The heuristics were *impact*, *dom/wldeg*, *dom/wdeg*, and *impact/wdeg*. The restart policies were Luby-k (k (backtracks per unit) $\in$ {128, 256, 512, 1024, 2048, 4096}), geometric (restart limit $x(n) = 100p^n$ at step n, where

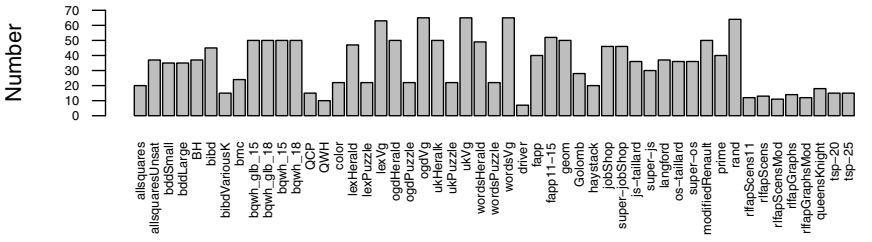

Fig. 2. Population distribution of the 1765 problems in the hard and harder problem sets, identified under stratified selection from 51 CSP competition classes

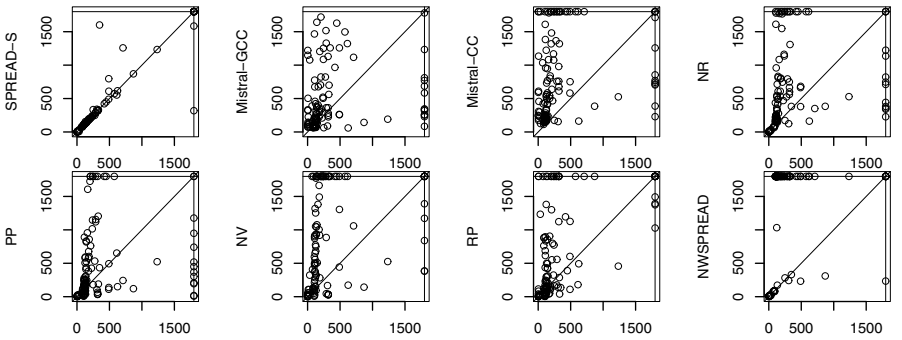

Fig. 3. On the 119-problem hard set, solution time in seconds for SPREAD-D (x-axis) plotted against that for other methods (y-axis). Each circle represents a problem; black areas indicate a heavy concentration of problems. Circles at the top and far right represent unsolved problems.

$p = 1.3$, 1.5 or 2.0), arithmetic ($x(n) = 16000n$, $8000n$, $1000n^2$, and $500n^2$), and dynamic. Dynamic adaptively determines whether to execute geometric restart with exponent 1.3, 1.5, or 2.0 based on the problem formulation, and restarts on the minimum of 1000 and the number of variables. Given these 4×16 possibilities but only 63 workers, PP did not execute *impact/wdeg* with dynamic restart and exponent 2.0.

- *Naive Variable* (*NV*) partially fixes the variable orders, as suggested in [16]. NV races 63 copies of Mistral, each of which randomly selects and orders the first 3 variables it assigns (but not their values) and reuses those variables on every restart.
- *Random Partitioning* (*RP*) splits on 7 randomly-chosen splitting variables, and enqueues those 128 subproblems for distribution to 63 workers, which run them to completion. Some workers process more than one subproblem.
- *No-Weight* SPREAD (*NW*SPREAD) is an ablated version of SPREAD intended to gauge the impact of learned weights. NWSPREAD does not use the weights from the portfolio phase for the workers, either to split or to search.

5 Experimental Results

Unless otherwise stated, all results reported here use the median of the values from three runs (as in recent parallel SAT solver competitions [22]), under a 30-minute per problem time limit, with the portfolio phase in both versions of SPREAD limited to 100 seconds. The initial backtrack limit was *base*, the average generated in the portfolio phase (Algorithm 1, line 8). When a subproblem was partitioned on ξ additional splitting variables, this limit was multiplied by $(1.5)^{\xi}$.

On the Hard Problem Set. Fig. 3 compares SPREAD-D's runtime to that of the other approaches in Section 4. Although a few instances (along the right margin) went unsolved under SPREAD-D, Fig. 3 shows that both versions of SPREAD

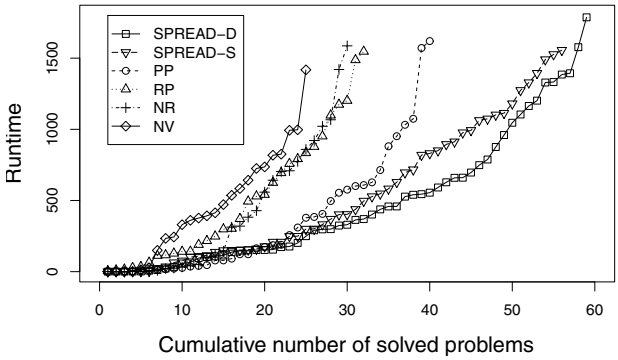

Fig. 4. On the 248 problems in the harder problem set, cumulative number of solved problems across 1800 seconds for SPREAD-D, SPREAD-S, and four competitors

clearly outperform most of the other benchmark methods. Indeed, on the problems solved both by SPREAD-D and each competitor, SPREAD-D achieved average speedups of 19.08 ($\sigma = 79.41$) over Mistral-GCC, 27.91 ($\sigma = 148.32$) over Mistral-CC, 2.65 ($\sigma = 2.77$) over NR, 1.98 ($\sigma = 1.92$) over PP, 4.03 ($\sigma = 3.89$) over NV, 3.34 ($\sigma = 6.48$) over RP, and 1.59 ($\sigma = 1.61$) over NW. Both SPREAD-S and SPREAD-D solved 43.70% of the hard set within 100 – 200 seconds. This is the time when both SPREAD versions have just begun to use critical splitting variables, while PP tries a complementary algorithm portfolio instead. The plot for PP on the lower left is a clear demonstration that search space splitting is essential. Moreover, search space splitting without the knowledge from the portfolio phase (NWSPREAD, on the lower right) was dramatically inferior; it could not solve 75.63% of these problems in 30 minutes, even though NR solved 17.65% of them within 100 seconds. Given their performance, NWSPREAD, sequential Mistral-CC, and Mistral-GCC were excluded from further comparisons.

On the Harder Problem Set. Fig. 4 compares both versions of SPREAD to the remaining parallelization methods. Given 30 minutes per problem, SPREAD-S solved 56 problems (44 satisfiable), 16 more (40.00% improvement) than the best benchmark method PP (which solved 40), and 31 more (124.00%) than the worst, NV (which solved 25). In addition, SPREAD-D solved 59 (46 satisfiable), 3 more than SPREAD-S. As one would expect, both versions of SPREAD behaved early on much like the portfolio-based methods NR and PP. SPREAD-S solved 10 (all satisfiable) within the first 100 seconds, its portfolio phase, while SPREAD-D solved 12 (all satisfiable). Both versions of SPREAD also solved more problems that required more time. In the last 800 seconds, SPREAD-S solved 18 (11 satisfiable) and SPREAD-D solved 12 (6 satisfiable).

On the Challenge Set. SPREAD-S and SPREAD-D significantly outperformed the other parallelization methods. Table 1 compares runtimes for SPREAD-S, SPREAD-D, and RP on the 35 problems solved by at least one of them more than once. (The other approaches from Fig. 4, NR, NV, and PP, solved 1, 1,

Table 1. Challenge problem solution times for SPREAD-S (S-S) and SPREAD-D (S-D), with best in boldface. 10 denotes a 10-second (rather than 100-second) portfolio phase. – denotes failure to solve in 30 minutes.

Problem	SAT	RP	S-S-10	S-S-100	S-D-10	S-D-100
costasArray-20	yes	**721.84**	–	–	876.13	1120.20
crossword-m1-words-21-10	yes	–	–	846.01	**746.36**	795.21
crossword-m1c-ogd-vg10-13_ext	no	–	744.89	**583.22**	748.62	750.28
crossword-m1c-ogd-vg10-14_ext	no	–	1302.41	**402.33**	1264.02	1280.17
crossword-m1c-ogd-vg12-12_ext	no	–	461.62	586.02	450.51	**450.22**
crossword-m1c-uk-vg11-12_ext	no	–	–	**1081.71**	–	–
frb53-24-2-mgd_ext	yes	–	749.33	**329.85**	749.25	330.59
frb53-24-5_ext	yes	748.17	63.04	255.86	**62.74**	256.10
frb56-25-2-mgd_ext	yes	–	661.94	822.12	**661.32**	822.18
graphcoloring-myciel6-6	no	–	–	–	–	**1185.96**
graphcoloring-myciel7-6	no	–	–	–	–	**1178.54**
langford-2-14	no	485.81	187.39	401.30	**150.96**	–
langford-3-16	no	567.92	659.73	446.50	537.89	**129.86**
rand-3-24-24-76-632-17_ext	yes	358.80	240.02	326.82	240.18	**239.56**
rand-3-24-24-76-632-fcd-47_ext	yes	823.91	693.89	**207.30**	691.45	697.14
rand-3-24-24-76-632-fcd-50_ext	yes	692.31	59.71	168.40	59.63	**58.01**
rand-3-28-28-93-632-16_ext	yes	–	1551.52	–	1551.47	**1541.81**
rand-3-28-28-93-632-23_ext	yes	–	551.02	758.57	**550.41**	592.62
rand-3-28-28-93-632-25_ext	yes	–	448.20	464.58	**448.14**	449.93
rand-3-28-28-93-632-3_ext	yes	–	1306.23	**648.04**	1305.86	1304.37
rand-3-28-28-93-632-30_ext	yes	–	**893.93**	1061.22	894.57	897.32
rand-3-28-28-93-632-35_ext	no	–	**1186.84**	1321.97	1192.83	1189.95
rand-3-28-28-93-632-37_ext	yes	–	–	**238.10**	–	–
rand-3-28-28-93-632-8_ext	no	–	1126.08	–	1126.21	**1118.44**
rand-3-28-28-93-632-fcd-16_ext	yes	–	1531.64	**530.76**	1529.82	1519.74
rand-3-28-28-93-632-fcd-20_ext	yes	**24.79**	299.64	314.52	299.73	295.269
rand-3-28-28-93-632-fcd-21_ext	yes	–	1322.25	–	1322.01	**1221.21**
rand-3-28-28-93-632-fcd-24_ext	yes	–	1122.49	–	1116.40	**1116.19**
rand-3-28-28-93-632-fcd-27_ext	yes	–	–	**1349.44**	–	–
rand-3-28-28-93-632-fcd-31_ext	yes	–	700.22	**211.54**	690.09	684.804
rand-3-28-28-93-632-fcd-35_ext	yes	–	494.40	616.61	492.14	**489.88**
rand-3-28-28-93-632-fcd-40_ext	yes	–	**137.29**	219.16	137.32	197.24
rand-3-28-28-93-632-fcd-42_ext	yes	–	144.94	**124.55**	152.84	138.62
rand-3-28-28-93-632-fcd-46_ext	yes	1410.23	168.30	**159.21**	171.66	166.41
super-js-taillard-20-20	no	–	–	–	**1142.61**	–
Problems solved least twice	–	9	27	27	30	30
Problems solved at least once	–	20	31	30	33	32

and 2 problems, respectively, all among these 35.) SPREAD-S and SPREAD-D are shown with both 10-second and 100-second portfolio phases; neither ever solved a problem during the portfolio phase. Although SPREAD did best with *rand* problems, it also solved problems in such categories as *Langford*, *crossword*, *super-jobshop*, and *graph-coloring*.

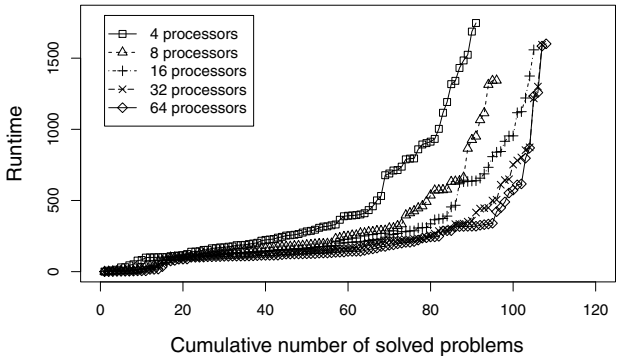

Fig. 5. Cumulative number of problems from the hard set solved by SPREAD-S

SPREAD's search is influenced by the variables it splits on and by their order, but the portfolio-phase search limit also has a strong effect. (Recall that the splitting-phase search limits are proportional to the backtracks consumed in the portfolio phase.) Because we report a median of three runs, to record a problem on any but the last line in Table 1, a program must have solved it at least twice. Both versions of SPREAD actually solved more problems; the last row indicates how many different problems they solved at least once in the three runs. Solved problems not listed in Table 1 include the satisfiable queenAttacking-8 and tdsp-C5-3-91, and the unsatisfiable pseudo-par-32-3-c, super-js-taillard-20-12, and super-js-taillard-20-22. Were the splitting-phase search limit infinite, SPREAD would partition only once and would probably benefit from a longer portfolio phase, but could readily be modified to search for all solutions.

Scalability. Fig. 5 shows that, given more processors, SPREAD consistently solved more problems from the hard set. More than 64 processors, however, introduced only marginal improvement on these problems. (Data omitted.) Because we did not tune SPREAD specifically for Mistral, we would expect similar improvement with other CSP solvers. Recall that, among our curated problems, the hard set contains the easiest ones, where further improvement by SPREAD is relatively difficult. In contrast, Fig. 6 shows how SPREAD scales on two typical problems from the harder problem set, given one hour. With more processors, SPREAD was significantly more likely to succeed within the time limit, and its runtime variance decreased, which produced more stable performance.

Other Statistics. When a worker completes its subproblem but no subproblems remain in the queue, that worker becomes *idle*. To investigate how well SPREAD uses its computing resources, let the *idle ratio* of the ith worker be the fraction of overall runtime that it was idle. SPREAD-S's average idle ratio on problems solved during the splitting phase rose as high as 0.8251 on hard, 0.8793 on harder, and 0.5015 on challenge problems. Large idle ratios were likely caused by a high backtrack limit on an extremely unbalanced search tree, which forced most other workers to wait for a new assignment. The idle ratio could be improved by a backtrack limit tailored to a particular problem class. Overall,

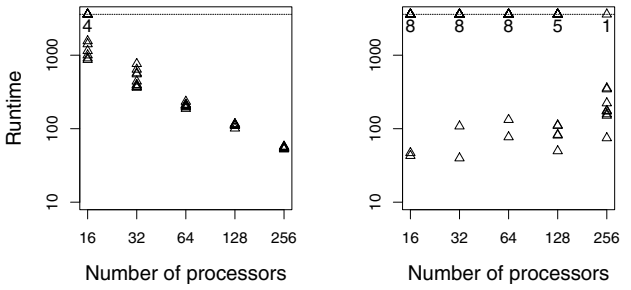

Fig. 6. Runtimes to solution within 1 hour across 10 runs of SPREAD-S with different numbers of processors for (a) rlfapScens11-f1 (unsatisfiable) (b) js-taillard-20-15-105-4 (satisfiable). Numbers with the uppermost triangles count failed runs.

however, SPREAD's idle ratio was under 0.1 on 56.92% of the hard, 69.92% of the harder, and 75.00% of the challenge problems. SPREAD-D's idle ratio was similar: 58.50%, 76.81%, and 75.00% under 0.1, respectively. Finally, Table 2 provides data on subproblems generated during the splitting phase.

6 Discussion

There are many plausible ways to parallelize a solver. One might perturb initial assignments, to vary the top of the search tree, using the same variables with different values. That was tested here as NV, and shown adequate only for the easiest of our test problems. Given the success of restarts and the ability of solvers to learn about contention, one might race the solver against copies of itself with different seeds. That was tested here as NR, and shown only slightly more effective. Given the success of some splitting and portfolio approaches, one might execute random partitioning, or race different solvers against one another. That was tested here as RP and PP, respectively, and shown adequate for some problems, but significantly less so for more difficult ones.

SPREAD could define its phases' search limits in number of backtracks, consistency checks, or search tree size. In the portfolio phase, time is the limiting factor because it forces all the workers to finish at once. In the splitting phase, however, there is a backtrack limit, to reduce the likelihood that all the workers will communicate with the manager at once.

To split a search space, SPREAD uses IBP, which, for generality, assumes no knowledge about problem domains. It could, however, be profitable to exploit domain characteristics. For example, one might partition the large domain of a critical variable into more subproblems, or partition extremely small domains (e.g., binary, as in SAT) with parity constraints [12].

Our work now proceeds along three lines. First, IBP may be misled by information collected during the portfolio phase. A typical example comes from the *queens-knights* (QK) problems. Although the contention in QK lies with the knights, weight-based variable-ordering heuristics prefer the queens variables at

Table 2. During the splitting phase, mean split subproblems (#), average maximum subproblem queue length (Max), and average maximum split variable number (μ)

	Hard			Harder			Challenge		
Implementation	#	Max	μ	#	Max	μ	#	Max	μ
Spread-S	156	129	7.6	518	136	13.6	244	132	9.6
Spread-D	146	129	7.5	374	137	11.3	211	130	8.6

the beginning of search, when weight-based heuristics (e.g., $dom/wdeg$) are close to those not based on weights (e.g., dom/deg). We are exploring adaptive methods that dynamically choose duration for the portfolio phase. Second, Spread-D did not always outperform Spread-S. In Spread-D, weights emphasize the local perspective of the subproblem, and preserve the portfolio phase's global perspective on the full problem only at the top of the search tree. We suspect that the initial partitioning is effective because it is based on parallel probing, and that repartitioning is less effective because it lacks the benefit of restart within the subproblem. We are therefore exploring restart strategies for Spread. Finally, nogood learning (as clause learning) has proved crucial in SAT, but has thus far received relatively little attention in parallel CSP solvers, including Spread. Future work includes combinations of adaptive splitting variable selection with nogood learning to avoid the loss of useful information.

Meanwhile, Spread offers a complete and effective method to parallelize a CSP solver. As a parallelization paradigm, Spread makes no assumption about domains or constraint types, and so accepts any class of CSPs that its solver can handle. Its bit-string representation permits programmers to ignore the implementation details of the solver, significantly simplifying parallelization, and dramatically reduces communication effort. Its portfolio phase solves easy problems quickly, and informs the splitting phase for effective partitioning. By recursively partitioning difficult subproblems with RS-IBP, it gradually allocates more computing cycles to the difficult parts of a problem, and thereby adaptively balances processor workload. Finally, Spread provides a natural way to embed restart policies into an MPI environment without recoding its underlying solver.

Acknowledgements. This work was supported in part by the National Science Foundation under IIS-0811437, CNS-0958379 and CNS-0855217, and the City University of New York's High Performance Computing Center. The authors thank the referees for their thoughtful suggestions.

References

1. Grimes, D., Wallace, R.J.: Sampling Strategies and Variable Selection in Weighted Degree Heuristics. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 831–838. Springer, Heidelberg (2007)
2. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: Proceedings of Sixteenth European Conference on Artificial Intelligence, pp. 146–149. IOS Press, Amsterdam (2004)

3. Refalo, P.: Impact-Based Search Strategies for Constraint Programming. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 557–571. Springer, Heidelberg (2004)
4. Gomes, C.P., Selman, B., Crato, N.: Heavy-Tail Distributions in Combinatorial Search. In: Smolka, G. (ed.) CP 1997. LNCS, vol. 1330, pp. 121–135. Springer, Heidelberg (1997)
5. Chu, G., Schulte, C., Stuckey, P.J.: Confidence-Based Work Stealing in Parallel Constraint Programming. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 226–241. Springer, Heidelberg (2009)
6. Michel, L., See, A., Van Hentenryck, P.: Parallelizing Constraint Programs Transparently. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 514–528. Springer, Heidelberg (2007)
7. Martins, R., Manquinho, V., Lynce, I.: Improving search space splitting for parallel SAT solving. In: Proceedings of Twenty-Second International Conference on Tools with Artificial Intelligence, pp. 336–343 (2010)
8. Zhang, H., Bonacina, M.P., Hsiang, J.: PSATO: a distributed propositional prover and its application to quasigroup problems. Journal of Symbolic Computation 21, 543–560 (1996)
9. Xie, F., Davenport, A.: Massively Parallel Constraint Programming for Supercomputers: Challenges and Initial Results. In: Lodi, A., Milano, M., Toth, P. (eds.) CPAIOR 2010. LNCS, vol. 6140, pp. 334–338. Springer, Heidelberg (2010)
10. Schubert, T., Lewis, M.D.T., Becker, B.: PaMiraXT: Parallel SAT solving with threads and message passing. Journal of Satisfiability, Boolean Modeling and Computation 6, 203–222 (2009)
11. Hyvärinen, A.E.J., Junttila, T., Niemelä, I.: Grid-Based SAT Solving with Iterative Partitioning and Clause Learning. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 385–399. Springer, Heidelberg (2011)
12. Bordeaux, L., Hamadi, Y., Samulowitz, H.: Experiments with massively parallel constraint solving. In: Proceedings of Twenty-First International Joint Conference on Artificial Intelligence, pp. 443–448 (2009)
13. Kotthoff, L., Moore, N.: Distributed solving through model splitting. In: Proceedings of Third Workshop on Techniques for Implementing Constraint Programming Systems, pp. 26–34 (2010)
14. Mehta, D., O'Sullivan, B., Quesada, L., Wilson, N.: Search Space Extraction. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 608–622. Springer, Heidelberg (2009)
15. Gomes, C., Selman, B.: Algorithm portfolio design: theory vs. practice. In: Proceedings of Thirteenth Conference on Uncertainty in Artificial Intelligence, pp. 190–197 (1997)
16. Kadioglu, S., Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M.: Algorithm Selection and Scheduling. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 454–469. Springer, Heidelberg (2011)
17. Hamadi, Y., Sais, L.: ManySAT: a parallel SAT solver. Journal on Satisfiability, Boolean Modeling and Computation 6, 245–262 (2009)
18. Yun, X., Epstein, S.: Adaptive parallelization for constraint satisfaction search. Accepted by SoCS (2012)
19. Guo, L., Hamadi, Y., Jabbour, S., Sais, L.: Diversification and Intensification in Parallel SAT Solving. In: Cohen, D. (ed.) CP 2010. LNCS, vol. 6308, pp. 252–265. Springer, Heidelberg (2010)
20. CSP Competition (CPAI 2008), `http://www.cril.univ-artois.fr/CPAI08/`
21. CSP Competition (CSC 2009), `http://www.cril.univ-artois.fr/CPAI09/`
22. SAT Competitions, `http://www.satcompetition.org`