

Parallelizing Constraint Programs Transparently

Laurent Michel¹, Andrew See¹, and Pascal Van Hentenryck²

¹ University of Connecticut, Storrs, CT 06269-2155

² Brown University, Box 1910, Providence, RI 02912

Abstract. The availability of commodity multi-core and multi-processor machines and the inherent parallelism in constraint programming search offer significant opportunities for constraint programming. They also present a fundamental challenge: how to exploit parallelism transparently to speed up constraint programs. This paper shows how to parallelize constraint programs transparently without changes to the code. The main technical idea consists of automatically lifting a sequential exploration strategy into its parallel counterpart, allowing workers to share and steal subproblems. Experimental results show that the parallel implementation may produce significant speedups on multi-core machines.

1 Introduction

Recent years have witnessed a transition in processor design from improvements in clock speed to parallel architectures. Multi-core and multiprocessor machines are now commodity hardware, a striking example of which is the recent announcement of the 80 core prototype developed by Intel [3]. Constraint programming (CP) search naturally offers significant opportunities for parallel computing, yet very little research has been devoted to parallel constraint programming implementations. Notable exceptions include CHIP/PEPSys [12] and its successors ECLiPSe [6], Parallel Solver [7], and Mozart [10].

The paper tackles the challenge eloquently stated by Schulte and Carlsson [11]: *how to exploit the resources provided by parallel computers and making their useful exploitation simple*. One of the difficulties here is the rich search languages typically supported by modern constraint programming languages (e.g., [9,7,13]). Indeed, CP search is best described by the equation

$$\text{CP Search} = \text{Nondeterministic Program} + \text{Exploration Strategy}$$

indicating that a CP search procedure consists of a nondeterministic program implicitly describing the search tree and an exploration strategy specifying how to explore the search space. For instance, in COMET, the nondeterministic program is expressed using high-level nondeterministic instructions such as `tryall`, while the exploration strategy is specified by a search controller, i.e.,

$$\text{CP Search in COMET} = \text{Nondeterministic Program} + \text{Search Controller.}$$

This research shows how to transparently parallelize such rich and expressive search procedures. The parallelization is purely built on top of the COMET system and involves no modification to its runtime. The key technical idea is to

```
1  CPSolver cp();
2  LDS lds(cp);
3  range S = 1..8;
4  var<CP>{int} q[S](m,S);
5  m.post(allDifferent(all(i in S) q[i] + i));
6  m.post(allDifferent(all(i in S) q[i] - i));
7  m.post(allDifferent(q));
8  exploreall<cp> {
9    forall(i in S) by q[i].getSize()
10     tryall<cp>(v in S : q[i].memberOf(v))
11       label(q[i],v);
12 }
```

Fig. 1. A Constraint Program for the N-Queens Problem

automatically lift the search controller of a constraint program into a parallel controller implementing the same exploration strategy in parallel. This parallelization does not require any change to the CP program and reuses the very same nondeterministic program and search controller. More precisely, the parallelization creates a number of workers, each of which executes the original constraint program with the generic parallel controller instead of the original search controller. The generic parallel controller, which encapsulates the original search controller, implements a work-stealing strategy in which an idle worker steals CP subproblems from other workers. Experimental results show that such a transparent parallelization may produce significant speedups compared to the sequential implementation.

The paper briefly reviews the concepts of nondeterministic programs and search controllers from the COMET system, since these are at the core of the parallel implementation. The paper then presents the transparent parallelization of a constraint program and its implementation. It reports experimental results showing the benefits of the approach and contrasts this implementation with earlier work, emphasizing the transparency and genericity of the approach.

2 Nondeterministic Programs and Search Controllers

Nondeterministic Programs COMET features high-level abstractions for nondeterministic search such as `try` and `tryall`, preserving an iterative programming style while making nondeterminism explicit. Figure 1 shows a constraint program for finding all solutions to the queens problem, illustrating some of the nondeterministic abstractions. Line 1 declares a CP solver and line 4 defines the decision variables. Lines 5–7 state the problem constraints, while lines 8–12 specify the nondeterministic search. The `exploreall` instruction in line 8 specifies that all solutions are required. The search procedure iterates over all variables ordered by smallest domains (line 9), nondeterministically chooses a value for the selected variable (line 10), and assigns the value to the variable (line 11).

```

1 interface SearchController {
2   void start(Continuation s, Continuation e);
3   void addChoice(Continuation c);
4   void fail();
5   void startTry();
6   void label(var<CP>{int} x, int v);
7   ...
8 }

```

Fig. 2. Interface of Search Controllers

The nondeterministic choice is expressed using a `tryall` instruction. Note that lines 8–12 specify a nondeterministic program that describes the search tree to explore. The nondeterministic instructions are concerned with the control flow only and are implemented using continuations.

Search Controllers. It is the role of the search controller to implement the exploration strategy (i.e., in which order to explore the search nodes), as well as how to save and restore search nodes. In Figure 1, the search controller is declared in line 2 and implements an LDS strategy. It is then used (through the CP solver) in the nondeterministic instructions in lines 8 and 10. The interface of search controllers is partially specified in Figure 2 and it is useful to review the semantics of some of its methods. Method `start` is called by instructions starting a nondeterministic search (e.g., the `exploreal` or `minimize` instructions). It receives two continuations `s` and `e` as parameters. Continuation `e` (for exit) is called when the search is complete. Continuation `s` (for start) can be called each time a restart is executed: When called, it restarts execution at the beginning of the nondeterministic search. Method `addChoice` adds a new choice point (described by a continuation) and method `fail` is called upon failure. Method `startTry` is called when a `try` or a `tryall` instruction is executed. Finally, method `label` is called to instantiate a variable to a specific value.

The connection between nondeterministic programs and controllers relies on source-to-source rewritings. For instance, the rewriting for the `try` instruction

```

try<sc> LEFT | RIGHT
    → sc.startTry();
      bool rightBranch = true;
      continuation c { rightBranch=false;
                      sc.addChoice(c);
                      LEFT; }
      if (rightBranch) RIGHT;

```

first invokes the `startTry` method of the controller. It then creates a continuation `c` to represent the right choice, adds the choice point to the controller, and executes the left branch. When the system wishes to return to the right choice, it calls the continuation `c`, executing the conditional statement. Since `rightBranch` is true in the continuation, the body `RIGHT` is executed.

An LDS Controller. The LDS controller is shown in Figure 3 and is organized around a queue of search node. A search node is represented by a pair (c, p) , where c is the continuation produced by the nondeterministic program and p is a semantic path [2,4], i.e., the sequence of constraints added on the path from the root to the search node. The most significant methods are `addChoice` (lines 7–9) and `fail` (lines 10–17). Method `addChoice` enqueues a search node composed of the continuation and a *checkpoint* which encapsulates the semantic path from the root to the node. Method `fail` calls the exit continuation if the queue is empty (line 11). Otherwise, it dequeues a search node (line 13), restores the search state (line 14), and calls the continuation (line 15) to resume execution. To restore a search state j from node i , the implementation finds the longest common prefix k in the sequence of i and j , backtracks to k , and posts all the remaining constraints of j . Note that the state restoration may fail in optimization applications when new bounds have been found elsewhere in the tree, in which case the `fail` method is called recursively.

```

1  class LDS implements SearchController {
2      CPSolver _cp; Continuation _exit; CPNodeQueue _Q;
3      LDS(CPSolver cp) { _cp = cp; _Q = new CPNodeQueue(); }
4      void start(Continuation s, Continuation e) { _exit = e; }
5      void exit() { call(_exit); }
6      void startTry() {}
7      void addChoice(Continuation c) {
8          _Q.enqueue(new CPNode(new CheckPoint(_cp), c));
9      }
10     void fail() {
11         if (_Q.empty()) exit();
12         else {
13             CPNode next = _Q.dequeue();
14             if (next.restore(_cp)==Failure) fail();
15             else call(next.getContinuation());
16         }
17     }
18     void label(var<CP>{int} x, int v) {
19         if (_m.label(x, v)==Failure) fail();
20     }
21 }

```

Fig. 3. The Controller for Limited Discrepancy Search

3 Transparent Parallelization of Constraint Programs

The parallelization of constraint programs is illustrated in Figure 4, which shows a parallel constraint program to solve the n-queens problem with 4 workers. It features a `ParallelCPSolver` instruction specifying the number of workers and enclosing the constraint program. Observe that the parallelization is entirely transparent and does not require any modification to the constraint program besides replacing the sequential CP solver by its parallel counterpart. In this

program, all the workers use an LDS exploration strategy. It is also possible to design a similar parallelization in which different workers implement different exploration strategies by using, say, a factory to create search controllers. The next two sections describes the implementation, starting with the high-level architecture before diving into the implementation.

```

1 ParallelCPSolver cp(4) {
2   LDS lds(cp);
3   range S = 1..8;
4   var<CP>{int} q[S](m,S);
5   m.post(allDifferent(all(i in S) q[i] + i));
6   m.post(allDifferent(all(i in S) q[i] - i));
7   m.post(allDifferent(q));
8   exploreall<cp> {
9     forall(i in S) by q[i].getSize()
10      tryall<cp>(v in S : q[i].memberOf(v))
11        label(q[i],v);
12   }
13 }
```

Fig. 4. Parallelizing the Constraint Program for the N-Queens Problem

4 High-Level Description of The Parallel Architecture

Each worker is associated with a thread and executes its own version of the constraint program. The workers collaborate by sharing subproblems, naturally leveraging the semantic paths used in the underlying search controllers. The collaboration is based on work stealing: When a worker is idle, it steals subproblems from active workers. Work stealing has some strong theoretical guarantees [1] and was successfully used in parallel (constraint) logic programming systems as early as in the 1980s (e.g., [14,12]) and in recent systems. Figure 5 illustrates work stealing. The left part depicts the search tree explored by a busy worker. The right part shows the new search tree after a steal operation: The stolen subproblems 4, 5, and 6 are placed in a problem pool (top right) and the search tree of the active worker is updated accordingly (bottom right).

In a continuation-based implementation of nondeterminism, the search nodes corresponding to open subproblems (e.g., subproblems 4–6 and 7–9 in Figure 5) have not been created yet: for instance, these nodes are created by lines 13–15 using continuation C_1 in the LDS controller in Figure 3. This lazy creation of search nodes is important for efficiency reasons. However, when an idle worker wants to steal these nodes, the busy worker has to generate them itself (since the continuation references its own stack). As a result, each worker is organized as a simple finite state machine, alternating the exploration of its search tree and the generation of subproblems to be stolen (see Figure 6). The transition from exploration to generation occurs when an idle worker requests a node, in which case the busy worker explicitly generates the top-level subproblems and

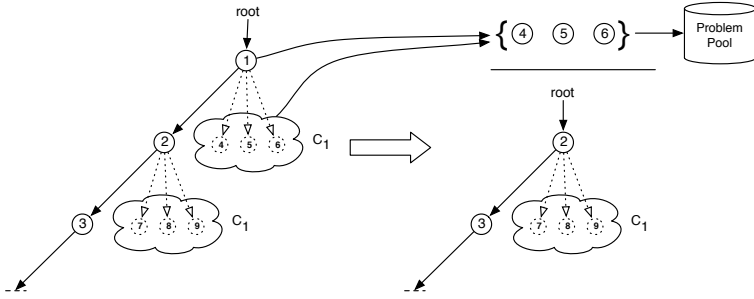


Fig. 5. Work Stealing in the Parallel Architecture

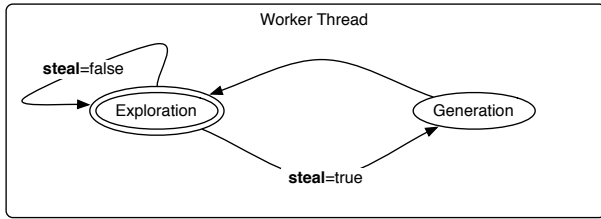


Fig. 6. The Finite State Machine for Workers

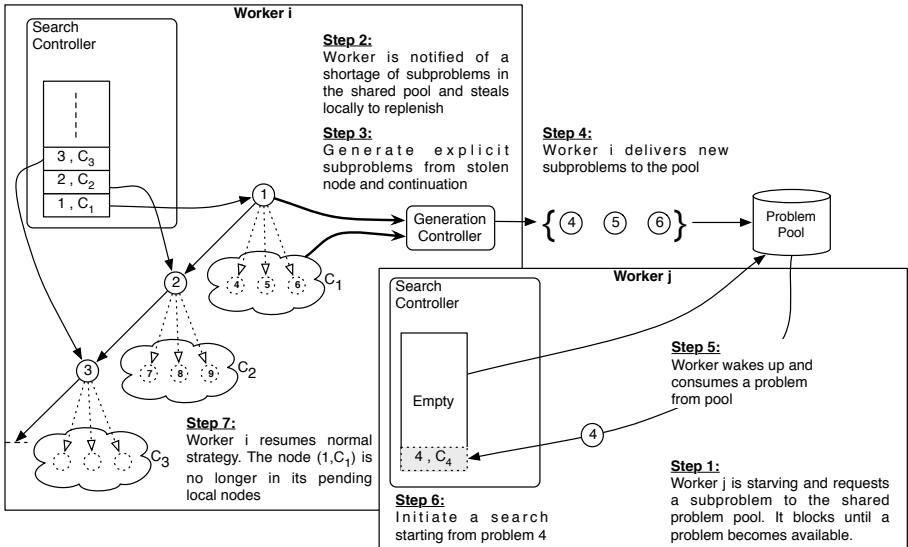


Fig. 7. The Work-Stealing Protocol

returns to its exploration. The work-stealing protocol is depicted in Figure 7 which describes all the steps in detail. The workers interact through a problem pool which store subproblems available for stealing. The pool may be distributed

across the workers or centralized. A centralized problem pool did not induce any noticeable contention in our parallel implementation, as it is organized as a producer/consumer buffers manipulating only pointers to subproblems.

The overall architecture has several fundamental benefits. First, it induces only a small overhead in exploration mode, since the workers execute a sequential engine with a lightweight instrumentation to be discussed in the next section. Second, the workers only exchange subproblems, making it possible for them to execute different search strategies if desired. This would not be possible if syntactic paths were used instead. Third, because the workers are responsible for generating subproblems available for stealing, the architecture applies directly to any exploration strategy: the strategy is encapsulated in the search controller and not visible externally.

5 Implementation

This section explains how to transparently parallelize a constraint program for the architecture presented above. *The fundamental idea is to instrument the constraint program with a generic parallel search controller which lifts any exploration strategy into a parallel exploration strategy.*

The Source to Source Transformation. The first step consists of rewriting the `ParallelCPSolver cp(4)` instruction to make the threads, the problem pool, and the parallel controller explicit. Figure 8 depicts the result of the transformation. The changes are located in Lines 1–4. Line 1 creates the problem pool. Line 2 uses the `parall` instruction of COMET [5] to create a thread for each of the four workers. Each thread executes the body of the loop (a closure) and has its own CP solver. Line 4 is important: It creates a parallel search controller. This parallel controller becomes the controller for the CP solver and is used by the nondeterministic instructions (through the CP solver) during the search. Note that line 5 adds the LDS controller to the parallel controller.

The Generic Parallel Controller. The implementation core is the parallel search controller whose architecture is depicted in Figure 9. The figure shows that the CP solver references the parallel controller which encapsulates two sub-controllers: the *exploration controller* for the search exploration (e.g., a DFS or an LDS controller) and the *generation controller* for generating subproblems. Subproblems are simply specified by semantic paths. Indeed, the initial constraint store and a semantic path define a subproblem, which can be solved by the search procedure. *This is exactly the idea of semantic decomposition proposed in [4], which we apply here to the parallelization of constraint programs.* We now describe the parallel controller in three steps to cover (1) the search initialization, (2) the transition between exploration and generation, and (3) subproblem generation. Observe that workers execute their own parallel controller: They only share the problem pool which is synchronized.

The Search Initialization. Figure 10 illustrates the two methods at the core of the initialization process. Recall that method `start` is called at the beginning

```

1 ProblemPool pool();
2 parall(i in 1..4) {
3   CPSolver cp();
4   ParallelController par(cp,pool);
5   LDS lds(cp);
6   range S = 1..8;
7   var<CP>{int} q[S](cp,S);
8   m.post(allDifferent(all(i in S) q[i] + i));
9   m.post(allDifferent(all(i in S) q[i] - i));
10  m.post(allDifferent(q));
11  exploreall<cp> {
12    forall(i in S) by q[i].getSize()
13      tryall<cp>(v in S : q[i].memberOf(v))
14        label(q[i],v);
15  }
16 }

```

Fig. 8. The Parallel Constraint Program after Source to Source Transformation

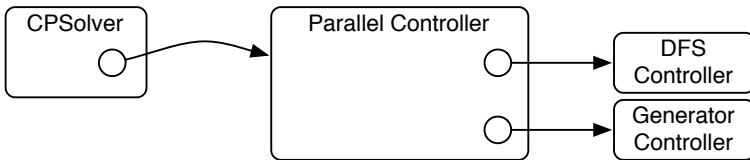


Fig. 9. The Architecture of the Parallel Controller

of the search (e.g., when the `exploreall` or `minimize` instructions are called). The `start` method of the generic parallel controller has two functionalities: (1) it must ensure that a single worker starts its exploration to avoid redundant computation; (2) it must allow all workers to steal work once they have completed their current execution. The `start` method solves the second problem by creating a continuation `steal` (lines 3–6) used to initialize the exploration controller in line 5. This means that the exploration controller, when calling its `exit` continuation, will now execute the instruction in line 7 to steal work instead of leaving the search. The first problem mentioned above is solved by the instruction in line 7. The problem pool is used to synchronize all the workers. A single worker is allowed to start exploring; the others start stealing. Note also that, when the first worker has finished its first exploration, it will also return to line 7 and start stealing work, since the test in line 7 succeeds exactly once.

Stealing work is depicted in lines 9–17. The worker requests a problem from the pool (line 10), i.e., it tries to steal problems from other workers. This call to a method of the problem pool only returns when the worker has found a subproblem or when all the workers are looking for work. When a subproblem is available, the parallel controller uses *semantic decomposition*: It restores the subproblem for its worker and restarts an entirely new search. That means (1) executing the `start` method on the subcontroller (line 13) and (2) restarting the

search by calling the `_start` continuation received by the `start` method (line 14) to restart execution from scratch. If no search problem is available, no work can be stolen anywhere and the worker exits (which means that its thread waits on a barrier to meet all the other threads in the `parall` instruction [5]).

```

1 void ParallelController::start(Continuation s, Continuation e) {
2   _start = s; _exit = e;
3   continuation steal {
4     _steal = steal;
5     _explorationController.start(null, _steal);
6   }
7   if (!_pool.firstWorker()) stealWork();
8 }
9 void ParallelController::stealWork() {
10  Checkpoint nextProblem = _pool.getProblem();
11  if (nextProblem != null) {
12    nextProblem.restore(_cp);
13    _explorationController.start(null, _steal);
14    call(_start);
15  }
16  else call(_exit);
17 }

```

Fig. 10. The Search Initialization in the Parallel Controller

```

1 void ParallelController::startTry() {
2   if (_generatingMode) _generationController.startTry();
3   else if (!_stealRequest) _explorationController.startTry();
4   else transition();
5 }
6 void ParallelController::transition() {
7   CPNode node = _explorationController.steal();
8   if (node != null) {
9     _generatingMode = true;
10    Checkpoint currentState = new Checkpoint(_cp);
11    continuation resume {
12      node.restore(_cp);
13      _generationController.start(null, resume);
14      call(node.getContinuation());
15    }
16    currentState.restore(_cp);
17    _generatingMode = false;
18  }
19  _explorationController.startTry();
20 }

```

Fig. 11. Mode switching in the Parallel Controller

The Transition Between Exploration and Generation. The transition between exploration and generation, as well as method delegation to the appropriate sub-controller, is depicted in Figure 11. Method `startTry` explains what happens at the beginning of every nondeterministic instruction. In generation mode, the parallel controller delegates to the generation controller (line 2); in exploration mode, as long as there are no stealing request, it delegates to the exploration controller (line 3). Otherwise, the parallel controller transitions to generation mode (line 4). To transition, the parallel controller first tests whether there is something to steal, in which case the instructions in lines 9–15 are executed. These instructions are responsible to start the generation of the nodes as discussed in Section 4. The parallel controller first steals the local node (line 10) (e.g., checkpoint 1 and continuation C_1 in Figure 7), restores the subproblem (line 12), calls the `start` method on the generation controller to initiate the generation process (line 13), and executes the continuation to produce the subproblems (line 14). Observe that the generation controller is called with a new exit continuation `resume`; upon exiting, the generation controller executes this continuation to return to line 16 allowing the worker to return to exploration mode. Indeed, line 16 restores the current state, while line 17 indicates that the parallel controller is in exploration mode. Finally, line 19 is executed to delegate the `startTry` call to the exploration controller.

```

1 void GenerationController::label(var<CP>{int} x,int val) {
2     if (_store.label(x,val)==Failure)
3         fail();
4     else {
5         _pool.add(new CheckPoint(_cp));
6         fail();
7     }
8 }

```

Fig. 12. The Implementation of the Generation Controller

The Generation Controller. The generation controller explores the children of a node, sending their subproblems to the problem pool. It can be viewed as a simple DFS controller failing each time a new subproblem is created. For instance, Figure 12 depicts the implementation of method `label` for the generation controller. Compared to the LDS controller in Figure 3, it adds lines 4–7 to produce the child subproblem to the problem pool and then fails immediately.

Optimization. In the description so far, workers only exchange subproblems. In optimization applications, it is critical for the workers to communicate new bounds on the objective function. The parallel workers use events to receive new bounds asynchronously. The instruction

```

whenever _pool@newBound(int bound)
    _bestFound = min(_bestFound,bound);

```

```

1 void ParallelController::addChoice(Continuation f) {
2     if (!_cp.setBound(_bestBound)) fail();
3     if (_generatingMode) _generationController.addChoice(f);
4     else                  _explorationController.addChoice(f);
5 }

```

Fig. 13. Using Bounds in the Parallel Controller

updates the best found solution in the parallel controller. The bound can then be used in the `addChoice` method to fail early as shown in Figure 13.

6 Experimental Results

We now describe the experimental results showing the feasibility of transparent parallelization. Before presenting the results, it is important to point out some important fact of parallel implementations. First, optimization applications may exhibit interesting behaviors. Some are positive: *superlinear* speedups may occur because the parallel version finds better solutions quicker and thus prunes the search space earlier. Some are negative: the parallel implementation may explore some part of the search tree not explored by the sequential implementation, reducing the speed-up. Typically, in optimization applications, we also give results factoring these behaviors by reporting the time to prove optimality given an optimal solution at the start of the search. Second, the parallel implementation of a search strategy produces a different search strategy because of work stealing. This strategy may be less or more effective on the problem at hand.

Experimental Setting. The results use an Apple PowerMac with two dual-core Intel Xeon 2.66 GHz processors (4 cores total), 4MB L2 Cache per processor, and 2GB of RAM running Mac OSX 10.4.9. Our implementation links our own in-house dynamic library for finite domains to the COMET system and implements the parallel controller in COMET itself. No change to the COMET runtime system were requested from its implementors. Results are the average of 10 runs. It is important to mention that parallel threads are in contention for the cache and for memory on multicore machines, which may have adverse effects on runtime.

Table 1. Experimental Results on the Queens Problem using DFS and LDS

	N	seq	Runtime (s)				Speedup (vs. seq)				Speedup (vs. 1w)			
			1w	2w	3w	4w	1w	2w	3w	4w	2w	3w	4w	
DFS	12	1.25	1.46	0.81	0.57	0.47	0.85	1.55	2.19	2.67	1.81	2.56	3.11	
DFS	14	36.6	41.9	22.8	16.6	13.5	0.87	1.60	2.19	2.70	1.84	2.52	3.09	
DFS	16	1421	1611.7	863.19	653.88	488.22	0.88	1.65	2.17	2.91	1.87	2.46	3.30	
LDS	12	2.12	2.33	1.47	1.15	0.91	0.91	1.45	1.84	2.33	1.59	2.02	2.56	
LDS	14	178	73.05	42.46	39.26	33.06	2.4	4.2	4.5	5.3	1.72	1.86	2.21	

Table 2. Experimental Results on a Scene Allocation Problem

	Runtime (s)					Speedup (vs. seq)				Speedup (1w)		
	seq	1w	2w	3w	4w	1w	2w	3w	4w	2w	3w	4w
DFS	257	267	138	96	74	0.96	1.86	2.70	3.49	1.94	2.80	3.62
DFS-P	248	264	138	94	72	0.94	1.80	2.64	3.45	1.92	2.82	3.68
LDS	435	376	200	164	121	1.16	2.18	2.66	3.60	1.88	2.30	3.11
LDS-P	388	362	190	160	115.7	1.07	2.04	2.43	3.35	1.90	2.27	3.13

Table 3. Experimental Results on Graph Coloring using DFS

Runtime						Speedup (seq)				Speedup (1w)		
	seq	1w	2w	3w	4w	1w	2w	3w	4w	2w	3w	4w
0	18.0	21.4	16.7	8.3	7.7	.84	1.07	2.16	2.33	1.28	2.57	2.78
1	39.8	46.9	16.3	12.3	10.5	.85	2.44	3.23	3.79	2.87	3.8	4.46
2	148.5	176.0	93.2	65.3	51.3	.84	1.59	2.27	2.89	1.89	2.70	3.40
3	323.8	385.6	198.1	141.9	111.1	.84	1.63	2.28	2.91	1.94	2.72	3.47
0-P	9.7	11.6	6.1	4.4	3.4	.84	1.60	2.20	2.85	1.9	2.63	3.41
1-P	22.6	26.9	14.1	10.0	7.9	.84	1.60	2.26	2.86	1.9	2.69	3.40
2-P	147.9	178.2	92.7	64.9	51.1	.83	1.60	2.28	2.89	1.92	2.75	3.49
3-P	324.0	390.0	199.1	141.2	111.0	.83	1.62	2.29	2.91	1.95	2.76	3.51

The Queens Problem. Table 1 depicts the results for finding all solutions to the queens problem using DFS. The table reports the times in seconds for the sequential program and for the parallel version with 1–4 workers. Columns 7–10 give the speedups with respect to the sequential version and columns 11–13 with respect to a single worker. The results show that the speedups increase with the problem sizes, reaching almost a speedup of 3 with 4 processors over the sequential implementation and 3.30 over a single worker. The overhead of a single worker compared to the sequential implementation is reasonable (about 13% for 16-queens). The last two lines depict the results for LDS: these results were surprising and show superlinear speedups for 14 queens due to the different exploration strategy induced by work stealing. Indeed, each time a node is stolen, a new LDS search is restarted with a smaller problem generating fewer waves. As a result, the size of the LDS queue is smaller, inducing time and space benefits.¹

Scene Allocation. Table 2 reports experimental results on a scene allocation problem featuring a global cardinality constraint and a complex objective function. The search procedure does not use symmetry breaking and the instance features 19 scenes and 6 days. The first two lines report results on DFS and the last two on LDS. The second and fourth lines report times for the optimality proof. DFS exhibits nice speedups reaching 3.45 for the proof and 3.49 overall. Observe also the small overhead of 1 worker over the sequential implementation. LDS exhibits superlinear speedups for 1 and 2 workers and nice speedups overall.

¹ The first worker always generates some problems in the pool (its steal request flag is initialized to true) which explains the superlinear speedup even with one worker.

Table 4. Experimental Results on Golomb Rulers

	N	seq	1w	2w	3w	4w
DFS	12	209/305	217/313 (0.97)	35/99 (3.08)	35/76 (4.01)	35/64 (4.76)
DFS	13	2301/5371	2354/5421 (0.99)	254/2228 (2.41)	254/1536 (3.49)	254/1195 (4.49)
LDS	12	575/1494	411/757 (1.97)	135/427 (3.49)	129/379 (3.94)	131/334 (4.47)

Graph Coloring. Table 3 illustrates results on four different instances of graph coloring with 80 vertices and edge density of 40%. The search procedure uses symmetry breaking. The overhead of the single worker is more significant here (about 20%) since there is little propagation and the cost of maintaining the semantic paths is proportionally larger. The top half of the table shows the total execution time, while the bottom part shows the time for proving optimality (given the optimal solution). Once again, the parallel implementation shows good speedups for a problem in which maintaining semantic paths is costly.

Golomb Rulers. The Golomb rulers problem is to place N marks on a ruler such that the distances between any two marks are unique. The objective is to find the shortest such ruler for N marks. The propagation is more time-consuming on this benchmark. Table 4 shows the speedups for Golomb 12–13 using DFS and Golomb 12 using LDS. The table reports, for each entry, the time to find the optimal solution, the total time, and the speedup. The speedups are always superlinear in this benchmark and the overhead of a single worker is small compared to the sequential implementation. Moreover, for DFS which dominates LDS here, the speedups scale nicely, moving from 3.49 with three processors to 4.49 with 4 processors ($N=13$), although the optimal solution is found after about 254 seconds in both. This is particularly satisfying since complex applications are likely to spend even more time in propagation than this program, showing (again) the potential of parallel computing for constraint programming. It is also important to mention that the sequential implementation is only about 20% slower than GECODE for the same number of choice points on this benchmark, although COMET is a very high-level garbage-collected language which imposes some overhead on our finite-domain library.

7 Related Work

The main novelty of this paper is the transparent parallelization of advanced constraint programming searches. To our knowledge, no other systems feature a similar functionality which addresses the challenge recently formulated by Schulte and Carlsson [11]. We review related work in constraint programming to emphasize the originality of our proposal.

Parallel constraint logic programming was pioneered by the CHIP/PEPSys system [12] and elaborated in its successors [6]. These systems offer transparent parallelization. However, they were implemented at a time when there was no clean separation between the nondeterministic program and the exploration

strategy (DFS was the exploration strategy). Moreover, these systems induce a performance penalty when parallelism is not used, since they must use advanced data structures (hash-windows, binding arrays, enhanced trailing) to support parallelism transparently [12]. *The proposal in this paper is entirely implemented on top of COMET and requires no change to its runtime system.*

Perron describes Parallel Solver in [7] and reports experimental results in [8], but he gives almost no detail on the architecture and implementation. Parallel Solver builds on Ilog Solver which supports exploration strategies elegantly through node evaluators and limits [7]. However, Ilog Solver uses syntactic paths which induce a number of pitfalls when, say, global cuts and randomization are used. It is not clear how work is shared and what is involved in parallelizing a search procedure, since no code or details are given.

Oz/Mozart was a pioneering system in the separation of nondeterministic programs and exploration strategies, which are implemented as search engines [9]. Schulte [10] shows how to exploit parallelism in Oz/Mozart using a high-level architecture mostly similar to ours. At the implementation level, workers are implemented as search engines and can generate search nodes available for stealing from a master. The actual implementation, which consists of about 1,000 lines of code, is not described in detail. It seems that syntactic paths are used to represent subproblems. Moreover, since search engines implement an entire search procedure, the worker cannot transparently lift another engine into a parallel search engine without code rewriting. In contrast, search controllers in COMET do not implement the complete search procedure: they just encapsulate the exploration order (through the `addNode` and `fail` methods) and save/restore search nodes. *It is the conjunction of the nondeterministic program (control flow through continuation) and search controllers (exploration order and node management) which implements the search procedure and makes transparent parallelization possible.* Note also that the code of the parallel controller is very short.

8 Conclusion

This paper showed how to transparently parallelize constraint programs addressing a fundamental challenge for CP systems. The key technical idea is to lift a search controller into a parallel controller supporting a work-stealing architecture. Experimental results show the practicability of the approach which produces good speedups on a variety of benchmarks. The parallel controller is built entirely on top of the COMET system and hence the approach induces no overhead when parallelism is not used. The architecture also allows different workers to use different strategies. Advanced users can write their own parallel controllers for specific applications, as the architecture is completely open. Future research will be concerned with a distributed controller: This requires distributing the problem pool which must become a shared object and replacing threads by processes, but these abstractions are also available in COMET.

Acknowledgments. This research is partially supported by NSF awards DMI-0600384 and IIS-0642906 and ONR Award N000140610607.

References

1. Blumhofs, R.D., Leiserson, C.E.: Scheduling Multithreaded Computations by Work Stealing. In: FOCS '94 (1994)
2. Choi, C.W., Henz, M., Ng, K.B.: Components for State Restoration in Tree Search. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239. Springer, Heidelberg (2001)
3. Intel Corp.: Teraflops research chip (2007)
4. Michel, L., Van Hentenryck, P.: A Decomposition-Based Implementation of Search Strategies. *ACM Transactions on Computational Logic* 5(2) (2004)
5. Michel, L., Van Hentenryck, P.: Parallel Local Search in Comet. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709. Springer, Heidelberg (2005)
6. Mudambi, S., Schimpf, J.: Parallel CLP on Heterogeneous Networks. In: ICLP-94 (1994)
7. Perron, L.: Search Procedures and Parallelism in Constraint Programming. In: Jaffar, J. (ed.) Principles and Practice of Constraint Programming – CP'99. LNCS, vol. 1713. Springer, Heidelberg (1999)
8. Perron, L.: Practical Parallelism in Constraint Programming. In: CP-AI-OR'02 (2002)
9. Schulte, C.: Programming Constraint Inference Engines. In: Smolka, G. (ed.) Principles and Practice of Constraint Programming - CP97. LNCS, vol. 1330. Springer, Heidelberg (1997)
10. Schulte, C.: Parallel Search Made Simple. In: TRICS'2000 (2000)
11. Schulte, C., Carlsson, M.: Finite Domain Constraint Programming Systems. In: Handbook of Constraint Programming. Elsevier, Amsterdam (2006)
12. Van Hentenryck, P.: Parallel Constraint Satisfaction in Logic Programming: Preliminary Results of CHIP within PEPSys. In: ICLP'89 (1989)
13. Van Hentenryck, P., Michel, L.: Nondeterministic Control for Hybrid Search. In: CP-AI-OR'05 (2005)
14. Warren, D.H.D.: The SRI Model for Or-Parallel Execution of Prolog. Abstract Design and Implementation Issues. In: ISLP-87 (1987)