Transparent Parallelization of Constraint Programming

Laurent Michel, Andrew See
University of Connecticut, Storrs, CT 06269-2155
Pascal Van Hentenryck
Brown University, Box 1910, Providence, RI 02912

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 sequential 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.

Key words: constraint programming, parallel computing, distributed computing, programming languages

---

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 [6]. 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 [25] and its successors ECLiPSe [17], Parallel Solver [19], and Mozart [22].

The paper tackles the challenge eloquently stated by Schulte and Carlsson [23]: *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., [19, 21, 28]). Indeed, constraint-programming search is best described by the equation

CP Search = Nondeterministic Program + Exploration Strategy

indicating that a constraint-programming search procedure consists of a nondeterministic program implicitly describing the search tree and an exploration strategy specifying how to

explore the search tree. 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. In other words, COMET implements the "equation"

CP Search in COMET = Nondeterministic Program + 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 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 constraint 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 subproblems from other workers. Experimental results on a variety of benchmarks show that such a transparent parallelization may produce significant speedups compared to the sequential implementation.

The rest of this paper is organized as follows. Section 2 briefly reviews nondeterministc search in COMET and, in particular, the separation between nondeterministic programs and exploration strategies which is fundamental for transparent parallelism. Section 3 reviews two basic parallel programming abstractions of COMET: parallel loops and thread pools, which are used in parallelizing constraint programs. Section 4 shows parallel implementations of constraint programs, highlighting the small distance between the sequential and parallel versions. Section 5 describes the implementation in detail, starting with a high-level overview before proceeding to the implementation of the parallel controller. Section 6 presents the experimental results on a variety of applications. Finally, section 7 presents the related work and section 8 concludes the paper.

2. Nondeterministic Search

This section describes how to implement nondeterministic search for constraint programming within the COMET system. COMET cleanly separates the search tree to explore from the exploration strategy and its design is based on the equation

Nondeterministic Search = Nondeterministic Program + Search Controller.

In particular, the search tree is specified by a nondeterministic program which uses high-level nondeterministic instructions such as `try` and `tryall`. The exploration strategy is specified

by a search controller. This section reviews both aspects in some detail.

2.1 Nondeterministic Programs

Comet provides high-level abstractions to specify nondeterministic programs naturally. These abstractions make nondeterminism explicit, while preserving a natural iterative programming style. They are best illustrated through some examples.

2.1.1 Perfect Squares

```
1  int    s     = 112;
2  range Side = 1..s;
3  range Squares = 1..21;
4  int side[Squares] =   [50,42,37,35,33,29,27,25,24,19,18,17,16,15,11,9,8,7,6,4,2];
5
6  SolutionPool Solutions();
7  CPSolver CP();
8  var<CP>{int} x[i in Square](CP,1..s-side[i]+1);
9  var<CP>{int} y[i in Square](CP,1..s-side[i]+1);
10
11 exploreall<CP> {
12   forall(i in Squares,j in Squares: i<j)
13     CP.post(x[i] + side[i] <= x[j] || x[j] + side[j] <= x[i] ||
14             y[i] + side[i] <= y[j] || y[j] + side[j] <= y[i]);
15   forall(p in Side) {
16     CP.post(sum(i in Squares) side[i]*((x[i]<=p) && (x[i]>=p-side[i]+1)) == s);
17     CP.post(sum(i in Squares) side[i]*((y[i]<=p) && (y[i]>=p-side[i]+1)) == s);
18   }
19 } using {
20   forall(p in Side, i in Squares)
21       try<CP> CP.post(x[i] == p); | CP.post(x[i] != p);
22   forall(p in Side, i in Squares)
23       try<CP> CP.post(y[i] == p); | CP.post(y[i] != p);
24   Solutions.add(new Solution(CP));
25 }
```

Figure 1: A Constraint Program for the Perfect Square Problem.

Figure 1 depicts a simple constraint program to find all solutions to the perfect square problem (problem 009 in CSPLib [8]). The problem consists of placing 21 squares, all of different sizes, into a 112×112 master square, so that the squares do not overlap and fill the entire master square. Lines 1–4 declare the input data, that is the size **s** of the master square (line 1) and the sizes of the squares (line 4). The rest of the statement mainly describes the decision variables, the constraints, and the search procedures.

Decision Variables The constraint solver and the decision variables are declared in lines 7–9. Every square `i` is associated with two decision variables, `x[i]` and `y[i]`, specifying the x and y coordinates of its bottom-left corner. The domains of these variables represent all the positions where square `i` can be placed in the master square. Note also that line 5 declares a variable (of type `SolutionPool`) to hold all the solutions to the problem.

Constraints The constraints are specified in lines 11–18. The non-overlapping constraints are expressed in lines 12–14. Given two squares `i` and `j`, they specify that `i` is placed on the left, on the right, below, or above `j`. The constraints in lines 15–18 specify that the squares fill the entire master space. In particular, line 16 (resp. line 17) states that the sizes of the squares intersecting a vertical (resp. horizontal) line `p` sum to the size of the master square.

Search procedure The nondeterministic program is described in lines 20–24. It first assigns the x-coordinates (lines 20–21) and then the y-coordinates (lines 22–23). Each of these assignments is a value/variable labeling: it considers a position `p` and determines the squares whose bottom-left corners have `p` as their x-coordinates (resp. y-coordinates). The nondeterministic program uses the nondeterministic `try` instruction which specifies several choices. In this application, the first choice assigns a position to a decision variable, say `x[i]`. The second choice imposes the constraint `x[i] != p`. Note also the `exploreall` instruction in line 11 that specifies that all solutions must be found and the instruction in line 24 which adds each solution to the pool declared in line 5. If only one solution is desired, the `explore` instruction can be used instead of `exploreall`.

2.1.2 Scene Allocation

Finding perfect squares is a constraint satisfaction problem. The second application, scene allocation [26], illustrates a nondeterministic program for an optimization problem. The goal is to minimize the cost of producing a movie consisting of several scenes. The scenes can be shot on various days, with the constraint that at most 5 scenes can be filmed every day. Each actor has a day fee, appears in various scenes, and is paid according to the number of days he/she spent on site.

Figure 2 depicts a constraint program for the scene allocation problem. The input data is declared in lines 1–10. The number of scenes is specified in line 1, the days in line 3, the actors in lines 4–5, and their fees in line 6. The array `appear` specifies the set of actors playing in each scene. For simplicity, its initialization is not shown (see line 8). The set of scenes in which an actor appears is computed in line 9, while line 10 specifies the maximum number of scenes (that is, five) that can be shot on any given day.

```
1  int    maxScene = 19;
2  range Scenes = 1..maxScene;
3  range Days = 0..6;
4  enum Actor = {Patt,Casta,Scolaro,Murphy,Brown,Hacket,Anderson,McDougal,
5                   Mercer,Spring,Thompson};
6  int    fee [Actor] =[26481,25043,30310,4085,7562,9381,8770,5788,7423,3303,9593];
7  set{Actor} appears[Scenes];
8  ...
9  set{int} which[a in Actor] = setof(i in Scenes) member(a,appears[i]);
10 int up[Days] = 5;
11
12 CPSolver CP();
13 var<CP>{int} shoot[Scenes](CP,Days);
14 minimize<CP> sum(a in Actor) sum(d in Days) fee[a] * or(s in which[a]) (shoot[s]==d)
15 subject to
16    CP.post(atmost(up,shoot));
17 using
18    forall(s in Scenes) by (shoot[s]. getSize(),− sum(a in appears[s]) fee[a])
19       tryall<CP>(d in Days)
20          CP.post(shoot[s] == d);
```

Figure 2: A Constraint Program for the Scene Allocation Problem.

Decision Variables There is a decision variable `shoot[s]` for every scene `s` (line 13), representing the day the scene is shot.

Objective and Constraints The objective function (line 14) specifies that the solution must minimize the sums of the actor costs, each actor being paid his/her fee times the number of days in which he/she shoots at least a scene. There is only one global cardinality constraint specifying that there are at most 5 scenes shot every day.

The Nondeterministic Program In this application, the nondeterministic program is a traditional labeling exploiting the first-fail principle. More precisely, the nondeterministic program iterates over the scenes and always chooses to assign first the scene with the smallest domain (line 19). Ties are broken by choosing the most costly scene first. Once a scene `s` is selected, the search nondeterministically assigns the days to `s` (lines 20–21). The nondeterministic program uses the `tryall` instruction which generalizes the `try` construct for situations in which the number of choices is not known a priori.

2.2 Search Controllers

As mentioned earlier, a search procedure in Comet is specified by a nondeterministic program and a search controller. Nondeterministic programs implicitly specify the search tree to explore, while search controllers specify the exploration strategy. Typically users will not

```
1  function int fact(int n) {
2     if (n==0) return 1;
3     else return n*fact(n−1);
4  }
5  int i = 4;
```

```
6  continuation c { i = 5; }
7  int r = fact(i);
8  cout<<"fact(" << i << ") =" << r << endl;
9
10 if (i == 5) call(c);
```

Figure 3: Continuations in COMET.

write controllers. COMET provides a default search controller implementing a depth-first exploration, as well as implementations of a variety of controllers. However, users retain the ability to write custom controllers whenever their applications impose some requirements not covered by existing search controllers.

Because they play a fundamental role in the parallelization of constraint programs, this section reviews search controllers in some detail. It starts by giving an overview of continuations, which are used to implement nondeterminism. It then presents the interface of search controllers, the link between nondeterministic programs and search controllers, and a number of search controllers. More information can be found in [28, 27].

2.2.1 Continuations

Continuations provide a flexible control structure to implement several higher-level abstractions such as exceptions, coroutines, and nondeterminism. Informally speaking, a continuation is a snapshot of the runtime data structures that allows the execution to restart from this point at a later stage of the computation. More precisely, a continuation is a pair $\langle I, S \rangle$, where I is an instruction pointer and S is a stack to execute the code starting at I. In COMET, continuations are obtained through instructions of the form

```
continuation c ⟨body⟩
```

that binds `c` to a continuation $\langle I, S \rangle$, where I is the next instruction in the code and S is the stack when the continuation is captured. It then executes its body and continues in sequence. The resulting continuation can be invoked with a `call(c)` that restores the stack S and restarts execution from I. Consider the code displayed in Figure 3. The code outputs

```
fact(5) = 120
fact(4) = 24
```

Indeed, the continuation `c` in line 5 consists of an instruction pointer to line 6 and a stack whose entry for `i` stores the value 4. The COMET implementation first calls the factorial function with argument 5 (since `i = 5` is executed when the continuation is taken). Since `i` has value 5, the implementation calls the continuation (line 8), which restarts execution

```
1 function int fact(int n) {
2    if (n==0) return 1;else return n*fact(n−1);
3 }
4 function Continuation getContinuation() {
5   continuation c { i = 5; }
6   return c;
7 }
```

```
8  int i = 4;
9  Continuation c = getContinuation();
10 int r = fact(i);
11 cout << "fact(" << i << ") = " << r << endl;
12
13 if (i == 5)
14    call(c);
```

Figure 4: Continuation in COMET Again.

in line 6 with a stack whose entry for `i` has value 4. The COMET implementation thus calls `fact(4)`, displays its result, and terminates (since `i` is 4).

Consider now the code displayed in Figure 4. The code has the same effect but it illustrates the complex control/stack patterns that may be induced by continuations. Indeed, the continuation is taken in line 5, i.e., inside the function `getContinuation`. The instruction pointer is on line 6 (the `return`) and the stack contains two frames for the global and the function scopes. When the continuation is called on line 14, the stack is restored, the execution restarts in line 6, returns the correct continuation `c`, and proceeds to compute `fact(4)`, displays its results, and terminates. Note that continuations, like closures, are first-class objects that can be stored in data structures, used as arguments, and returned as values.

2.2.2 The Interface of Search Controllers

Since the implementation of nondeterminism in COMET is based on continuations, it is thus not surprising that search controllers implement the search strategy by manipulating continuations. The interface of search controllers is depicted in Figure 5 and we now review the role of each method informally.

The `setExit` method is called by the `explore`, `exploreall`, and `minimize` instructions and receives as argument a continuation `e` which specifies what to do after the search is completed. The `start` method is called by the `explore`, `exploreall`, and `minimize` instructions to initiate the nondeterministic execution. It receives a continuation `s` which is captured at the beginning of the nondeterministic program. The `restart` method is used for restarting strategies and typically calls the continuation `s` above (among other treatments). The `exit` method typically calls the continuation `e` to terminate the search. The `addChoice` and `fail` methods are particularly important in that they typically specify the search strategy. Method `addChoice` receives a continuation `c` representing a nondeterministic choice and stores `c` in some data structure for future use. Method `fail` is called upon a failure in the search and typically restarts execution by executing a continuation received in `addChoice`.

```
interface SearchController {
  void setExit(Continuation e);
  void start(Continuation s);
  void restart ();
  void exit ();
  void addChoice(Continuation c);
  void fail ();
  void startTry();
  void exitTry();
  void startTryall ();
  void exitTryall ();
  CPNode steal();
  void solve(CPNode n);
}
```

Figure 5: The Interface of Search Controllers.

```
try<sc> LEFT | RIGHT
```

↓

```
sc.startTry ();
bool rightBranch = true;
continuation c {
  rightBranch = false;
  sc.addChoice(c);
  LEFT;
}
if (rightBranch) RIGHT;
sc.exitTry ();
```

Figure 6: Compiling the `try`.

```
exploreall<sc> CSTR using SEARCH
```

↓

```
continuation e {
   sc.setExit(e);
   CSTR
   continuation s {
     cc.start(s);
   }
   SEARCH
   sc. fail ();
}
```

Figure 7: Compiling the `exploreall`.

The remaining instructions are executed before and after a `try` (resp. `tryall`) instruction.

2.2.3 The Link between Nondeterministic Instructions and Search Controllers

Nondeterministic instructions in COMET are implemented in terms of source-to-source transformations. Figures 6 and 7 illustrate these transformations on the `try` and `exploreall` instructions; other nondeterministic instructions are rewritten similarly (see [28] for details).

Consider first Figure 6 which depicts the rewriting for the `try`. The resulting code first invokes the `startTry` method of the controller (line 1). It then creates a continuation c to represent the right choice (line 3), adds the choice point to the controller (line 5), executes the left branch (line 6), and invokes method `exitTry` (line 9). The right branch in line 9 is not executed, since the boolean `rightBranch` was set to false in line 4. 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 (see line 2), the body `RIGHT` is executed in line 8.

Consider now Figure 7 that depicts the rewriting for the `exploreall`. An `exploreall` instruction first takes a continuation `e` representing what must be executed when no more choice points are left unexplored, i.e., when all the solutions of its body have been explored (line 1). It first stores the continuation in the search controller and posts the constraints (lines 2–3). It then takes a continuation `s` which captures the beginning of the nondeterministic program, and stores it in the controller as well (lines 4–5). It then executes the nondeterministic program and fails, which induces the search controller to consider unexplored choices (lines 7–8). When no such choices exist, the search controller will call continuation `e`.

2.2.4 A Controller for Depth-First Search with Checkpoints

We now turn to the implementation of several search controllers used in the experimental results. Our first controller is an implementation of depth-first search using checkpoints. Checkpoints are first-class objects in COMET and represent the constraints added from the start of the search to the specific computation point where the checkpoint was taken. The use of checkpoints is critical in parallelizing constraint programs, since they precisely define nodes of the search tree. (Checkpoints are called semantic paths [4, 15]). Figure 8 depicts the implementation of the controller. It inherits from `CPSearchController` which has default implementations of a number of methods. The DFS controller maintains a stack of continuations and a stack of checkpoints to store unexplored branches of the search tree, which automatically grows as needed. The method `addChoice` (line 12–16) pushes the continuation (argument `f`) onto the stack. It also captures a checkpoint, which is pushed onto the checkpoint stack. This pair (continuation,checkpoint) specifies a node to explore: The continuation specifies where to restart execution, while the checkpoint may be used to restore the state of the solver. Together, the continuation and the checkpoint are an implicit representation of the unexpanded siblings of the current search node which are set aside for later expansion. The `fail` method (line 17–28) is called upon failure, either when the solver discovers an infeasibility or when a solution has been found in the `exploreall` or `minimize` instructions. If the stacks are empty, it calls the `exit` method to conclude the search. Otherwise, it pops the continuation and the checkpoint from the top of the stacks (lines 23–24), restores the state of the constraint solver (line 23), and calls the continuation (line 25) if the checkpoint restoration was successful.[1]

[1]The restoration may fail in minimization problems when the restored state may not improve upon the best upper bound so far.

```
1  class DFSearch extends CPSearchController implements SearchController {
2     Continuation[] _cont;
3     Checkpoint[]  _chp;
4     int               _maxSize;
5     int               _top;
6     DFSearch(CPSolver cps) : CPSearchController(cps) {
7        _maxSize = 100;
8        _cont  = new Continuation[0.._maxSize−1];
9        _chp   = new Checkpoint[0.._maxSize−1];
10    }
11    void start(Continuation s) { super.start(s);_top = 0;}
12    void addChoice(Continuation f) {
13       resize ();
14       _cont[_top++] = f;
15       _chp[_top]  = new Checkpoint(_cps.getStore());
16    }
17    void fail () {
18       do {
19          if (_top == 0)
20             exit ();
21          else {
22             Continuation next = _cont[_top − 1];
23             bool ok = _chp[_top−1].restore(_cps.getStore ());
24             _chp[−−_top]=null;
25             if (ok) call(next);
26          }
27       } while (true);
28    }
29 }
```

Figure 8: A Controller for Depth-First Search with Checkpoints.

2.2.5 A Controller for Limited Discrepancy Search

Limited discrepancy search (LDS) [10] is a search strategy relying on a good heuristic. It explores the search space by trusting the heuristic early in the search and less and less so over time. More precisely, LDS can be thought of as exploring the search tree as a series of waves. In the first wave, LDS follows the heuristic to a leaf. In the second wave, it explores all the leaves that can be reached when the search path differs from the heuristic on exactly one decision (that is, the search path takes a single different choice in the nondeterministic program). In wave i, LDS explores leaves that can be reached when the search path exhibits i discrepancies with respect to the path followed by the heuristic.

A wave is a collection of nodes that are represented by a continuation, a checkpoint, as well as additional information such as the discrepancy or the depth in the tree. At any point in time, the implementation must manipulate both the current and the next wave. There are many ways of implementing LDS and Figure 9 depicts an implementation organized

```
1  class LDSearch extends CPSearchController implements SearchController {
2     CPNodeStack _curStack,_nextStack;
3     int              _discrep , _maxDiscrep;
4     LDSearch(CPSolver cps) : CPSearchController(cps) {
5        _curStack  = new CPNodeStack();
6        _nextStack = new CPNodeStack();
7     }
8     void start(Continuation enter) { super.start(enter);_discrep=0; _maxDiscrep=0; }
9     void addChoice(Continuation f) {
10      Checkpoint cp = new Checkpoint(_cps.getStore());
11      if ( _discrep + 1 < _maxDiscrep)
12         _curStack.push(CPNode(f,cp,_discrep+1));
13      else _nextStack.push(CPNode(f,cp,_discrep+1));
14    }
15    void fail () {
16       do {
17          if (_curStack.empty() && _nextStack.empty()) {
18             exit ();
19          } else {
20             if (_curStack.empty()) {
21               _maxDiscrep += 1;
22                CPNodeStack tmp = _curStack;
23                _curStack = _nextStack;
24                _nextStack = tmp;
25             }
26             CPNode p = _curStack.pop();
27             _discrep  = p.getDiscrepancies();
28             bool ok = p.getCheckpoint().restore();
29             if (ok) call(p.getContinuation());
30          }
31       } while (true);
32    }
33 }
```

Figure 9: The Controller for Limited Discrepancy Search.

around two stacks of nodes, for the current and the next wave. Stack `_curStack` holds the unexplored nodes in the current wave, while `_nextStack` holds the nodes of the next wave. Like in the DFS controller, method `addChoice` creates a checkpoint (line 10). Then it inserts a node in the current wave if it does not exceed the limit on discrepancies (lines 11–12) or in the next wave otherwise (line 13). The method `fail` pops the node from the current wave if possible (lines 26–29), in which case it also restores the node discrepancies (line 27). If the current wave is empty, the two stacks are swapped and the number of allowed discrepancies is increased (lines 20–25). The search completes when both stacks are empty.

To use LDS in the scene allocation program in Figure 2, it suffices to add the instruction

```
CP.setController(LDSearch(CP));
```

after line 12 where the CPSolver `CP` is created. This clear separation between the nonde-

terministic program and the search strategy is one of the appealing features of constraint programming. It is one of the main contributions of this paper to show that constraint programs can be parallelized automatically, while preserving this functionality.

2.2.6 A Controller for Bounded Discrepancy Search

We conclude this section with a controller for bounded discrepancy search. The controller is depicted in Figure 10 and consists of limiting the search to those that have at most k discrepancies compared to the heuristic. Note that this is an incomplete search controller as it would be necessary to progressively increase the limit k on the number of discrepancies to obtain a complete search. The controller is essentially similar to the DFS controller but is organized around a stack of nodes. It also maintains the discrepancies of the current node (lines 10 and 19) and differs in how method `addChoice` is implemented. Indeed, instead of systematically pushing a node on the stack, the controller does so only whenever the number of discrepancies is not above the threshold (lines 10–11). Observe also that the node pushed on the stack has `_discrep+1` discrepancies.

```
1  class BDSearch extends CPSearchController implements SearchController {
2     CPNodeStack _curStack;
3     int           _discrep , _maxDiscrep;
4     BDSearch(CPSolver cps,int di) : CPSearchController(cps) {
5         _curStack = new CPNodeStack();
6         _maxDiscrep=di;
7     }
8     void start(Continuation enter) { super.start(enter);_discrep=0;}
9     void addChoice(Continuation f) {
10       if ( _discrep + 1 < _maxDiscrep)
11          _curStack.push(CPNode(f,Checkpoint(_cps.getStore()),_discrep+1));
12    }
13    void fail () {
14       do {
15          if (_curStack.empty()) {
16             exit ();
17          } else {
18             CPNode p = _curStack.pop();
19             _discrep = p.getDiscrepancies();
20             bool ok = p.getCheckpoint().restore();
21             if (ok)
22                call(p.getContinuation());
23          }
24       } while (true);
25    }
26 }
```

Figure 10: The Controller for Bounded Discrepancy Search.

3. Parallel Abstractions of Comet

This section gives a brief overview of some parallel abstractions of Comet used later in the paper. More detail on these and other abstractions can be found in [16, 13].

3.1 Parallel Loops

Comet provides a `parall` construct, as the parallel counterpart to the sequential `forall`. Consider the Comet snippet for a parallel multi-start local search for jobshop scheduling.

```
1 Solution sol[1..nbStarts];
2 parall(i in 1..nbStarts) {
3    JobshopLocalModel js();
4    sol[i] = js.solve();
5 }
6 selectMax(i in 1..nbStarts)(sol[i].getValue()) cout << "Sol. cost:" << sol[i].getValue() << endl;
```

Each execution of the body creates a jobshop model (line 3) and searches for a high-quality solution from a randomly generated initial solution (line 4). For iteration `i`, the returned solution is stored in `sol[i]` (line 4 again). Once all iterations are completed, the value of the best found solution is displayed (line 6). The parallel abstraction is illustrated in line 2: The `parall` instruction is the parallel counterpart to the sequential `forall` construct of Comet. Operationally, the `parall` creates a thread to execute the loop body for each value of `i`. These threads are joined after the loop, i.e., the instruction following the loop is only executed after all threads completed their execution. Observe the easy transition from a sequential to a parallel multi-start algorithm.

3.2 Thread Pools

The implementation of the `parall` instruction associates a thread with each iteration of the parallel loop. Since each thread has its own runtime control blocks and stacks[2], this implementation may induce some non-negligible overhead when the number of iterations is large. Thread pools overcome this limitation and allow Comet programs to map nicely onto the underlying architecture. The following snippet

```
1 ThreadPool tp(4);
2 Solution sol[1..nbStarts];
3 parall<tp>(i in 1..nbStarts) {
4    JobshopLocalModel js();
5    sol[i] = js.solve();
6 }
7 selectMax(i in 1..nbStarts)(sol[i].getValue()) cout << "Sol. cost: " << sol[i].getValue() << endl;
```

[2]Each thread has its native runtime control block and stack, as well as equivalent data structures for the Comet runtime.

reconsiders the parallel multistart algorithm but now uses a thread pool. The thread pool is declared in line 1 and used to parameterize the `parall` instruction in line 3. The semantics is the same as the traditional `parall` but the loop only uses the four threads in the pool to execute the iterations. The fragment shows how to collect all the solutions in a solution pool (declared in line 2) and how to retrieve the best solution in line 7. Observe the small distance between the sequential and parallel code and the clean separation between the model, the parallel code, and the mapping on the target architecture. Note also that the size of the parallel pool can be determined statically (or dynamically with a runtime call to `System.getNumberOfCores()`) according to the number of available processors.

3.3 Early Termination

Sometimes, it is desirable to terminate all threads as soon as some condition holds, e.g., as soon as a solution is found. For these situations, COMET provides an abstraction for early termination which we illustrate with a multi-start local search model for the the Progressive Party Problem based on a parallel loop. The code snippet on the left

```
1  ThreadPool tp(4);
2  Boolean found(false);
3  Solution sol;
4  parall<tp>(i in 1..nbStarts) {
5     ProgressivePartyModel pp();
6     if (pp.search()){
7        sol = pp.getSolution();
8        found := true;
9     }
10 } until found;
```

```
1  ThreadPool tp(4);
2  Boolean found(false);
3  Solution sol;
4  parever<tp>{
5      ProgressivePartyModel pp();
6      if (pp.search()){
7           sol = pp.getSolution();
8           found := true;
9      }
10 } until found;
```

uses a parallel loop and a condition `until found` on line 10 which terminates all threads as soon as one of them sets `found` to true on line 8. When the number of iterations to run is a priori unbounded, a `parever` construct can be used instead as shown on the right hand side. It runs the same search as before but no longer bounds the number of iterations. Similar constructs can be used in many other contexts.

4. Transparent Parallelization of Constraint Programs

The parallelization of constraint programs is illustrated in Figures 11 and 12, which depict parallel versions of the square and scene programs with one worker per core.

Consider first Figure 11. The modifications for obtaining a parallel version of the algorithm are in lines 7–9 and 29. They do not affect the model, that is the decision variables, the constraints, and the search procedure (lines 10–27). The parallelization first requires

```
1  int    s    = 112;
2  range Side = 1..s;
3  range Squares = 1..21;
4  int side[Squares] =   [50,42,37,35,33,29,27,25,24,19,18,17,16,15,11,9,8,7,6,4,2];
5
6  SolutionPool Solutions();
7  CPProblemPool rep(System.getNumberOfCores());
8  parall<rep>(i in 1..rep.getSize()) {
9    ParallelCPSolver CP(rep);
10   var<CP>{int} x[i in Square](CP,1..s−side[i]+1);
11   var<CP>{int} y[i in Square](CP,1..s−side[i]+1);
12
13   exploreall<CP> {
14     forall(i in Squares,j in Squares: i<j)
15       CP.post(x[i] + side[i] <= x[j] || x[j] + side[j] <= x[i] ||
16               y[i] + side[i] <= y[j] || y[j] + side[j] <= y[i]);
17     forall(p in Side) {
18       CP.post(sum(i in Squares) side[i]∗((x[i]<=p) && (x[i]>=p−side[i]+1)) == s);
19       CP.post(sum(i in Squares) side[i]∗((y[i]<=p) && (y[i]>=p−side[i]+1)) == s);
20     }
21   } using {
22     forall(p in Side, i in Squares)
23       try<CP> CP.post(x[i] == p); | CP.post(x[i] != p);
24     forall(p in Side, i in Squares)
25       try<CP> CP.post(y[i] == p); | CP.post(y[i] != p);
26     Solutions.add(Solution(CP));
27   }
28 }
29 rep.close();
```

Figure 11: A Parallel Constraint Program for the Perfect Square Problem.

the declaration of a problem pool, called the *repository*, which contains subproblems shared by the workers (line 7). The repository encapsulates a thread pool with, in this particular program, as many threads as physical cores. The COMET program then features a parallel loop with as many iterations as there are workers (line 8). The repository is then closed in line 29, releasing its threads. The only modification is in line 9, where the constraint solver is replaced by a parallel constraint solver.

Consider now Figure 12 that depicts a parallel program for scene allocation. The parallelization proceeds exactly as for the perfect square program, although this is a minimization problem and the search strategy uses an LDS controller. Once again, it illustrates that the parallelization of constraint programs in COMET is essentially transparent.

```
1  int    maxScene = 19;
2  range Scenes = 1..maxScene;
3  range Days = 0..6;
4  enum Actor = {Patt,Casta,Scolaro,Murphy,Brown,Hacket,Anderson,McDougal,
5                 Mercer,Spring,Thompson};
6  int    fee [Actor] =[26481,25043,30310,4085,7562,9381,8770,5788,7423,3303,9593];
7  set{Actor} appears[Scenes];
8  ...
9  set{int} which[a in Actor] = setof(i in Scenes) member(a,appears[i]);
10 int up[Days] = 5;
11
12 CPProblemPool rep(System.getNumberOfCores());
13 parall<rep>(i in 1..rep.getSize()) {
14    ParallelCPSolver CP();
15    CP.setController(LDSearch(CP));
16    var<CP>{int} shoot[Scenes](CP,Days);
17    minimize<CP> sum(a in Actor) sum(d in Days) fee[a] * or(s in which[a]) (shoot[s]==d)
18    subject to
19      CP.post(atmost(up,shoot));
20    using
21      forall(s in Scenes) by (shoot[s].getSize(),- sum(a in appears[s]) fee[a])
22         tryall<CP>(d in Days)
23            CP.post(shoot[s] == d);
24 }
25 rep.close();
```

Figure 12: A Parallel Constraint Program for Scene Allocation using an LDS Controller.

5. The Implementation

This section describes the implementation of the transparent parallelization. It first reviews the parallel model, before presenting the parallel search controller.

5.1 High-Level Description of The Parallel Architecture

In the parallel implementation, 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., [29, 25]) and in recent systems. Figure 13 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).

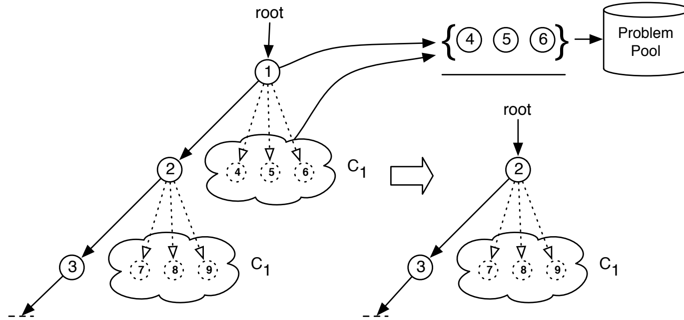

Figure 13: Work Stealing.

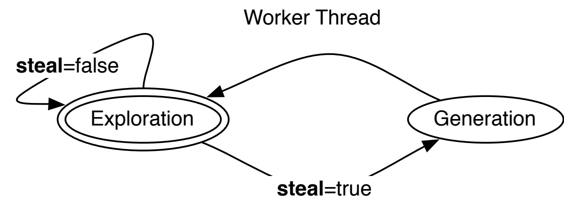

Figure 14: The Workers' FSA.

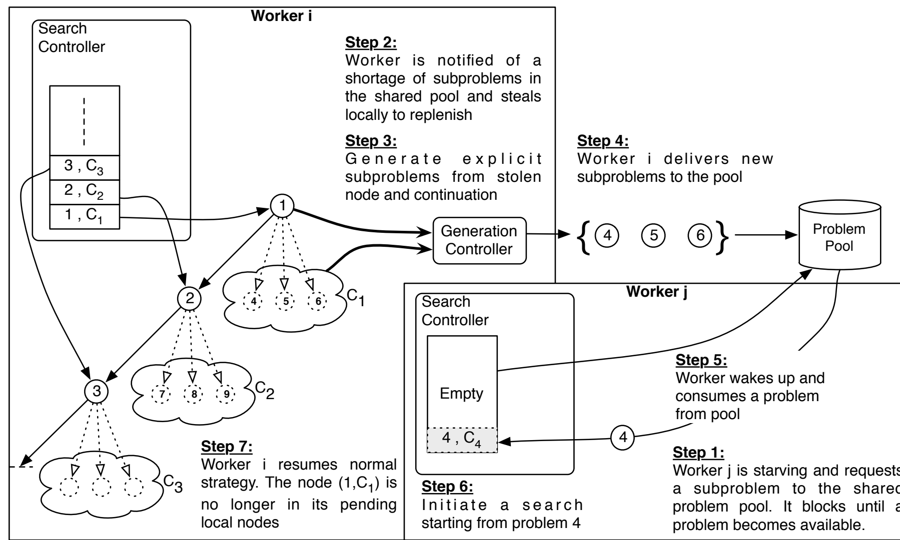

Figure 15: The Work-Stealing Protocol.

In the Comet implementation, the search nodes corresponding to open subproblems (e.g., subproblems 4–6 and 7–9 in Figure 13) have not been created yet: for instance, these nodes are created by lines 17–28 in the DFS controller of Figure 8. 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, 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 14). The transition from exploration to generation occurs when an idle worker requests a node, in which case the busy worker explicitly generates subproblems before returning to its exploration. In the current implementation, busy workers periodically poll the problem pool to check whether more subproblems must be shared, in which case they switch to generation.

The work-stealing protocol is depicted in Figure 15 which describes all the steps in detail. The workers interact through a problem pool which stores subproblems available for stealing.

The pool may be distributed across the workers or centralized. A centralized problem pool does not induce any noticeable contention in our parallel implementation, as it is organized as a producer/consumer buffer manipulating only pointers to subproblems. In the first step of Figure 15, worker i is busy while worker j is idle. Since the problem pool is empty, the request from worker j for a new subproblem blocks. In steps 2 and 3, worker j polls the pool, switch to generating mode, and steals the root node, to produce nodes 4–6. In step 4, nodes 4–6 enter the problem pool. In step 5–6, worker i wakes up to receive a subproblem from the pool (node 4) and starts a new search. Finally, in step 7, worker i resumes its own search. Note that this example shows a somewhat extreme case since, in general, busy workers switch to generation mode before the pool is empty, so idle workers rarely starve.

The overall architecture has several 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. 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.

5.2 The Interface of Search Controller Revisited

To parallelize constraint programs transparently, it is necessary to enhance the interface of search controllers slighly. Figure 5 depicted the full interface. The last two methods, namely `steal` and `solve`, augment the core functionalities described earlier with work stealing capabilities: `steal` is used to steal a search node from a controller and `solve` is used to solve a subproblem. Figure 16 shows an implementation of these two methods for a DFS controller. Method `steal` simply steals the bottom node of the stack (if any). For DFS, the bottom node represents the root of the current search tree and tends to produce large subproblems. Note that, for other strategies, it might be beneficial to steal a different node. This is easily achieved by overriding the `steal` method. Method `solve` restores the checkpoint stored in the subproblem and restarts the search procedure by executing the "start" continuation. Observe that method `solve` does not use the continuation stored in the search node, since it generally orginates from another thread. Instead, it simply restores the checkpoint, thus creating a subproblem which is solved using the search procedure.

5.3 The Parallel Search Adapter

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*

```
1  CPNode DFSearch::steal() {
2    if (_top > 0) {
3      CPNode rv(_cont[0],_chp[0]);
4      forall(i in 1.._top-1) {
5        _cont[i-1] = _cont[i];
6        _chp[i-1] = _chp[i];
7      }
8      --_top;
9      return rv;
10   } else return null;
11 }
```

```
12 void DFSearch::solve(CPNode n) {
13   Checkpoint cp = n.getCheckpoint();
14   bool success = cp.restore(_cps.getStore());
15   if (success)
16     call(_start);
17 }
18
19
20
21
22
```

Figure 16: The `steal` and `solve` Methods of the DFS Controller.

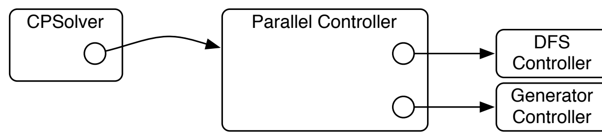

Figure 17: The Architecture of the Parallel Controller.

generic parallel search adapter that lifts any exploration strategy into a parallel exploration strategy. The core of the implementation is the parallel search adapter created when the parallel solver is declared (for instance, line 9 in Figure 11). The architecture of the parallel adapter is depicted in Figure 17. 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., the default or user-supplied controller) and the *generation controller* for generating subproblems. Subproblems are obtained from search nodes by extracting the checkpoints and other information such as discrepancies. Indeed, the initial constraint store and a checkpoint define a subproblem, which can be solved by the original search procedure. *This is exactly the idea of semantic decomposition proposed in [15], which we apply here to the parallelization of constraint programs.* Observe also that each worker has its own constraint solver and thus its own parallel adapter. They only share the repository which contains search nodes available for stealing. The repository is a monitor (like solution pools) and all its methods are executed in mutual exclusion. The parallel adapter is now described in two steps to cover (1) the search initialization and (2) the exploration/generation transition.

5.3.1 The Search Initialization

Figure 18 illustrates the two methods at the core of the initialization process. Recall that method `start` is called at the beginning of the search (e.g., when the `exploreall` or `minimize` instructions are called). The `start` method of the parallel adapter has two functionalities: (1) it must ensure that a single worker starts its exploration to avoid redun-

```
1  void ParallelAdapter::start(Continuation s) {
2      _start = s;
3      continuation steal {
4          _steal = steal;
5          _exploration.setExit( _steal );
6          _exploration.start( _start );
7      }
8      if (!_rep.firstWorker()) stealWork();
9  }
```

```
10 void ParallelAdapter::stealWork() {
11     CPNode nextProblem = _rep.getProblem();
12     while (nextProblem != null) {
13         _exploration.solve(nextProblem);
14         nextProblem = _rep.getProblem();
15     }
16     call( _exit );
17 }
18
```

Figure 18: The Search Initialization in the Parallel Controller.

dant computations; (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 2–7) used to initialize the `exit` continuation of exploration controller in line 5. This means that the exploration controller, when calling its `exit` continuation, will now execute the instruction in line 8 to steal work instead of leaving the search. The first problem mentioned above is solved by the instruction in line 8. The repository 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 8 and start stealing work, since the test in line 8 succeeds exactly once.

Stealing work is depicted in lines 10–17. The worker requests a subproblem from the repository (line 11), that is, it tries to steal problems from other workers. This call to a method of the repository 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 adapter asks the exploration controller to solve the subproblem (line 13). In other words, the exploration controller starts an entirely new search to solve the subproblem. In other words, workers exchange subproblems, which are solved by applying the original search procedure.

Note that it may be the case that the subproblem cannot be restored successfully (in optimization problems), in which case line 14 is executed to obtain a new subproblem. If nothing 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 [16]).

5.3.2 The Transition Between Exploration and Generation

The transition between exploration and generation, as well as the delegation to the appropriate sub-controller, is depicted in Figure 19. Method `addChoice` illustrates the delegation process: In generation (resp. exploration) mode, the call is delegated to the generation (resp. exploration) controller. Method `startTry` and `exitTry` follow the same schema. However, method `startTry` also "publishes" work when requested by the repository (line

```
1  void ParallelAdapter::addChoice(continuation f) {
2    if (_generatingMode)
3        _generation.addChoice(f);
4    else _exploration.addChoice(f);
5  }
6  void ParallelAdapter::startTry() {
7    publish();
8    if (_generatingMode)
9        _generation.startTry();
10   else _exploration.startTry();
11 }
12 void ParallelAdapter::exitTry() {
13   if (_generatingMode)
14       _generation.exitTry();
15   else _exploration.exitTry();
16 }
```

```
17 void ParallelAdapter::publish() {
18   if (!_generatingMode)
19     if (_rep.wantMore()) {
20       CPNode node = _exploration.steal();
21       if (node != null) {
22         _generatingMode = true;
23         Checkpoint now(_csp.getStore());
24         continuation resume {
25           _generation.setExit(resume);
26           _generation.solve(cpp);
27         }
28         now.restore(_cps.getStore());
29         _generating = false;
30     }
31 }
32
```

Figure 19: Mode switching in the Parallel Controller.

7). Method `publish` is responsible to start the generation of the nodes as discussed in Section 5.1. Method `wantMore` of the problem pool implements polling and is called in line 19 to determine whether the pool needs more subproblems. When such condition holds, the worker switches to generation mode. The resulting protocol is similar to randomized work stealing [1], but possibly resulting in more than one worker producing subproblems. Once again, heuristics for worker selection and subproblem ordering can be implemented naturally by overriding the relevant methods. The parallel controller first steals a local node (line 20) from the exploration controller (e.g., checkpoint 1 and continuation C_1 in Figure 15). If such a problem is available for stealing, the code in lines 21–29 moves the adapter into generation mode (line 22), generates subproblems (line 26) by calling the generation explorer, and returns in exploration mode (line 29). To return in generation mode, the parallel adapter creates a new continuation `resume` and a checkpoint `now`. The generation controller is called with its exit continuation set to `resume`, ensuring that, when the generation phase is completed, execution returns in line 28. Similarly, the checkpoint `now` is restored so that the constraint solver returns to its exploration state.

5.4 The Generation Controller

It remains to describe the generation controller whose role is to explore 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 20 depicts the implementation of method `exitTry` for the generation controller. This method is called at the end of the nondeterministic instruction `try`. As a result, when it reaches this instruction, the

```
1 void GenerationController::exitTry() {
2   Checkpoint toPub(_cps.getStore());
3   _rep.addProblem(CPNode(null,toPub,_discr);
4   fail ();
5 }
```

Figure 20: The Generation Controller.

```
1 void ParallelAdapter::addChoice(Continuation f){
2    if (!_cp.setBound(_bestBound)) fail();
3    if (_generatingMode) _generation.addChoice(f);
4    else _exploration.addChoice(f);
5 }
```

Figure 21: Bounds in the Parallel Controller.

controller has created a subproblem. The code thus creates a checkpoint `toPub` capturing this subproblem (line 2) and creates a node with this checkpoint and possibly other information as the number of discrepancies.[3] It then fails, generating other nodes until its stack is empty, in which case its exit continuation is called and execution resumes in exploration mode.

5.5 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 _rep@newBound(int bound) _bestFound = min(_bestFound,bound);
```

updates the best found solution in the parallel adapter. The bound can then be used in the `addChoice` method to fail early as shown in Figure 21.

5.6 Distributed Computing

This section discusses how to implement a distributed search controller to explore the search space on a network of workstations. The main difference between the parallel and distributed code is the pool. Indeed, sub-problems produced by workers can no longer be shared easily since the workers now have different address spaces. Instead, they must be shipped over the network back to the master (using sockets in our implementation) and then transmitted to starving workers (also across the network). Note also that, on a parallel machine, all the workers can probe the problem pool regularly to decide whether to *push* new sub-problems into the pool. Since they share the same address space, it induces negligible overhead. On a distributed platform, however, the higher communication latency makes it necessary to use a *pull* model in which the master requests subproblems whenever it wishes to replenish its pool. This communication protocol is once again implemented using events. The instruction

```
whenever _rep@needMore() _wantGenerate = true;
```

[3] The continuation is irrelevant since it cannot be used by other workers.

informs the worker that the pool size is low and new work must be published. The call to `_rep.wantMore()` in `publish()` is then replaced with a test of the flag `_wantGenerate`. This pull model works equally well in parallel, but was not used initially for simplicity.

5.7 Portability to Other Systems

It is important to stress that the abstractions proposed herein are independent of the COMET system. They could be implemented in any system, albeit more tediously. They are based on a number of abstractions, each of which could be reimplemented in other languages or even in C++ libraries. COMET makes heavy use of continuations and closures, but these could be implemented or simulated in languages that do not support them natively [14]. Similarly, parallel loops have been implemented in systems such as OpenMP [3] and work stealing has been implemented in parallel logic programming systems such as CHIP [25] and in Cilk [5]. One of the main novelties in this paper is to show that a careful selection of abstractions enables a transparent parallelization of constraint programs and this contribution is not restricted to COMET, but would generalize to other systems and libraries.

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 facts 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-ups. 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. Indeed, the stolen subproblems are solved with the search procedure and the heuristic may now be different. This strategy may be less or more effective on the problem at hand. This is also the case when a single worker is used with the parallel abstractions since this worker still publishes work to the pool and solves them later, starting from the beginning of the search procedure. Testing one worker in this way serves to measure the overhead/speedup of the implementation without any benefits of parallelization.

	Runtime (s)					Speedup (vs. seq)				Speedup (1w)		
	seq	1w	2w	3w	4w	1w	2w	3w	4w	2w	3w	4w
DFS	448	493	262	180	144	0.91	1.71	2.49	3.11	1.88	2.74	3.42
DFS-P	395	428	222	150	117	0.92	1.78	2.64	3.37	1.93	2.86	3.66
LDS	1375	594	364	277	225	2.32	3.78	4.96	6.10	1.63	2.14	2.64
LDS-P	1192	516	311	229	194	2.31	3.84	5.20	6.15	1.66	2.25	2.66

Table 1: A Scene Allocation Problem.

6.1 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 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 necessary. 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.

The experimental results always compare the parallel and the sequential implementations. In all cases but depth-first search, the exploration and the sequential controllers are the same. For depth-first search, it is possible, in a sequential implementation, to obtain a more efficient implementation than the controller depicted in Figure 8. Indeed, since backtracking is chronological, it suffices to capture the trail pointer instead of a checkpoint, which may be significant when the constraint propagation does not take much time.

6.2 Scene Allocation

Table 1 reports experimental results on a scene allocation problem featuring a global cardinality constraint and a complex objective function. These results are the average of 50 runs. 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 line report times for the optimality proof. DFS exhibits nice speedups reaching 3.37 for the proof and 3.11 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. Recall that the parallelization changes the strategy, even with 1 worker, which happens to be beneficial in this problem, causing these superlinear speedups.

6.3 Graph Coloring

Tables 2 and 3 illustrate results on four different instances of graph coloring with 80 vertices and edge density of 40%. These tables report averages over 25 runs. The search procedure

Graph	P	time(s)				Speedup		#fails	# choices
		ave	stdev	min	max	vs seq	vs 1w		
0	seq	16.43	0.00	16.43	16.43	–	–	802556.00	491736.00
	1w	20.71	0.00	20.71	20.71	0.79	1.00	802543.00	492013.00
	2w	17.37	0.02	17.33	17.42	0.95	1.19	1194385.56	733692.80
	3w	5.89	0.01	5.88	5.91	2.79	3.52	537093.60	326507.32
	4w	5.62	0.09	5.32	5.71	2.92	3.69	580073.40	354978.72
1	seq	35.43	0.00	35.43	35.43	–	–	1731518.00	1062397.00
	1w	44.66	0.00	44.66	44.66	0.79	1.00	1731480.00	1062894.00
	2w	16.57	0.02	16.54	16.63	2.14	2.69	1109595.84	671722.40
	3w	12.93	0.04	12.85	12.99	2.74	3.45	1192087.40	724050.84
	4w	11.74	0.08	11.63	11.86	3.02	3.80	1272640.40	775273.36
2	seq	139.54	0.00	139.54	139.54	–	–	6713460.00	4165273.00
	1w	175.96	0.00	175.96	175.96	0.79	1.00	6713416.00	4165715.00
	2w	100.01	0.22	99.69	100.92	1.40	1.76	6714167.72	4166947.44
	3w	73.09	0.09	72.90	73.23	1.91	2.41	6714841.44	4168152.48
	4w	61.83	0.17	61.54	62.08	2.26	2.85	6715623.24	4169339.76
3	seq	9.36	0.00	9.36	9.36	–	–	459005.00	289468.00
	1w	11.86	0.00	11.86	11.86	0.79	1.00	458995.00	289701.00
	2w	38.23	0.10	38.06	38.57	0.24	0.31	2603176.04	1623075.64
	3w	35.53	13.08	4.13	48.35	0.26	0.33	3319330.64	2069677.76
	4w	7.51	3.84	2.84	11.79	1.25	1.58	793073.84	498238.16
4	seq	292.39	0.00	292.39	292.39	–	–	13906376.00	8459797.00
	1w	367.71	0.00	367.71	367.71	0.80	1.00	13906352.00	8460236.00
	2w	208.35	0.19	208.01	208.63	1.40	1.76	13903257.52	8458907.48
	3w	152.48	0.14	152.22	152.71	1.92	2.41	13901763.92	8458895.64
	4w	128.59	0.19	128.28	128.97	2.27	2.86	13902145.52	8460042.64
5	seq	40.73	0.00	40.73	40.73	–	–	2014059.00	1269648.00
	1w	51.40	0.00	51.40	51.40	0.79	1.00	2014039.00	1269955.00
	2w	12.14	0.03	12.09	12.19	3.35	4.23	814057.56	505484.28
	3w	10.90	0.05	10.80	10.98	3.74	4.72	1005267.80	622130.52
	4w	8.47	0.32	8.06	9.08	4.81	6.07	898695.40	556901.56

Table 2: Graph Coloring using DFS.

uses symmetry breaking. The overhead of the single worker is more significant since there is little propagation and the cost of maintaining the semantic paths is proportionally larger.

The first two columns indicate the problem number and the number of workers respectively. Columns 3–5 show the average, standard deviation, and minimum and maximum runtimes across all runs. The next two columns show the speedups with respect to the sequential and one worker implementations. Finally, the last two columns show the average number of failures and choice points over all workers. In Table 2, it is interesting to note that the number of failures and choice points stay approximately the same as the number of workers increases, for some instances, indicating that the total work done is about the same as in the sequential implementation. Note that there is very little variance in the execution times regardless of the number of workers, except on instance 3 where the number of choice points increases with 3 workers –the manifestation of a negative effect where more

Graph	P	time(s)				Speedup		#fails	# choices
		ave	stdev	min	max	vs seq	vs 1w		
0	seq	8.8	0.00	8.80	8.80	–	–	407960.00	243755.00
	1w	10.9	0.00	10.93	10.93	0.80	–	407947.00	244032.00
	2w	6.2	0.02	6.19	6.25	1.42	1.76	407956.04	244649.40
	3w	4.6	0.01	4.54	4.58	1.93	2.40	408012.92	245194.20
	4w	4.1	0.07	4.02	4.19	2.13	2.65	408064.00	245808.60
1	seq	20.2	0.00	20.19	20.19	–	–	945124.00	567346.00
	1w	25.3	0.00	25.27	25.27	0.80	–	945086.00	567843.00
	2w	14.3	0.01	14.23	14.29	1.42	1.77	945136.76	568329.80
	3w	10.4	0.02	10.38	10.45	1.94	2.43	945188.16	568928.32
	4w	8.9	0.11	8.60	9.03	2.27	2.84	945281.40	569769.00
2	seq	139.9	0.00	139.95	139.95	–	–	6712804.00	4164746.00
	1w	176.3	0.00	176.31	176.31	0.79	–	6712760.00	4165188.00
	2w	100.1	0.08	99.98	100.35	1.40	1.76	6712861.20	4165899.64
	3w	73.4	0.09	73.24	73.61	1.91	2.40	6712953.60	4166704.80
	4w	61.9	0.15	61.47	62.09	2.26	2.85	6713038.00	4167494.08
3	seq	1.4	0.00	1.40	1.40	–	–	62754.00	37028.00
	1w	1.7	0.00	1.74	1.74	0.80	–	62744.00	37261.00
	2w	1.0	0.01	1.01	1.03	1.37	1.70	62741.24	37623.64
	3w	0.8	0.01	0.77	0.79	1.80	2.23	62781.08	37966.00
	4w	1.0	0.09	0.68	1.09	1.39	1.72	62777.68	38314.44
4	seq	293.1	0.00	293.08	293.08	–	–	13900251.00	8455791.00
	1w	367.1	0.00	367.09	367.09	0.80	–	13900227.00	8456230.00
	2w	208.7	0.39	208.18	210.02	1.40	1.76	13900374.04	8457046.92
	3w	153.1	0.15	152.75	153.38	1.91	2.40	13900480.80	8457938.60
	4w	128.6	0.16	128.29	129.07	2.28	2.85	13900569.64	8458728.20
5	seq	9.3	0.00	9.29	9.29	–	–	428063.00	257784.00
	1w	11.6	0.00	11.59	11.59	0.80	–	428043.00	258091.00
	2w	6.6	0.01	6.54	6.58	1.42	1.77	428071.52	258583.24
	3w	4.8	0.01	4.79	4.83	1.93	2.41	428079.76	259131.04
	4w	4.3	0.11	4.00	4.46	2.14	2.68	428100.68	259638.44

Table 3: Graph Coloring using DFS, proof only.

work is done in parallel–. When the number of choice points decreases, as in instance 5, the speedup improves significantly. In Table 3, the number of choicepoint remains roughly the same as the number of workers increase. Once again, the parallel implementation shows good speedups for a problem in which maintaining semantic paths is costly.

6.4 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 the runtime, speedup, failures and choicepoints for DFS and LDS, showing the full search and proof separately, averaging over 25 runs. The speedups are good, and in some cases superlinear.

N	P	time(s)				Speedup		#fails	# choices
		ave	stdev	min	max	vs seq	vs 1w		
12 DFS	seq	304	0.00	304	304	–	–	2656744.00	792993.00
	1w	331	0.00	331	331	0.92	1.00	2656730.00	792981.00
	2w	118	43.57	105	272	2.59	2.81	1820556.00	537251.52
	3w	163	13.27	139	178	1.87	2.03	3783363.00	1138224.32
	4w	71	23.15	59	147	4.26	4.63	2142895.76	635456.24
13 DFS	seq	5361	0.00	5361	5361	–	–	35497987.00	10525412.00
	1w	5758	0.00	5758	5758	0.93	1.00	35497968.00	10525394.00
	2w	3422	119.07	3204	3498	1.57	1.68	42801271.72	12729656.60
	3w	2068	178.72	1622	2329	2.59	2.78	38348351.44	11392749.64
	4w	1383	241.69	1276	2026	3.88	4.16	33158246.36	9750232.08
12 LDS	seq	1932	0.00	1932	1932	–	–	2137797.00	2377081.00
	1w	846	0.00	846	846	2.28	1.00	2676625.00	2699015.00
	2w	636	53.16	457	661	3.04	1.33	3381710.04	3457495.20
	3w	512	47.26	438	600	3.77	1.65	2641990.56	2738050.24
	4w	413	98.91	287	594	4.68	2.05	2194304.56	2295617.72
12 DFS-P	seq	141	0.00	141	141	–	–	1100901.00	314148.00
	1w	148	0.00	148	148	0.95	1.00	1100874.00	314125.00
	2w	76	0.07	76	76	1.86	1.95	1100852.12	314111.52
	3w	52	0.06	52	52	2.73	2.86	1100828.40	314091.28
	4w	39	0.05	39	39	3.61	3.78	1100747.96	314026.64
13 DFS-P	seq	4089	0.00	4089	4089	–	–	26138742.00	7549948.00
	1w	4296	0.00	4296	4296	0.95	1.00	26138707.00	7549916.00
	2w	2174	2.48	2170	2179	1.88	1.98	26138666.68	7549883.04
	3w	1480	8.35	1455	1488	2.76	2.90	26138640.12	7549865.84
	4w	1113	1.26	1110	1115	3.67	3.86	26138604.88	7549836.24
12 LDS-P	seq	420	0.00	420	420	–	–	1100901.00	1100901.00
	1w	220	0.00	220	220	1.91	1.00	1100874.00	1100877.00
	2w	116	0.17	116	116	3.61	1.89	1100843.92	1100847.92
	3w	139	4.36	120	144	3.02	1.58	1076639.16	1085666.76
	4w	144	3.77	137	148	2.92	1.53	1051037.60	1067577.48

Table 4: Golomb Rulers.

Moreover, for DFS which dominates LDS here, the speedups scale nicely, moving from 2.59 with three processors to 4.26 with 4 processors (N=13). This is particularly satisfying since complex applications are likely to spend even more time in propagation, 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 [11] 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.

6.5 Jobshop Scheduling

We conclude this section by reporting a number of experimental results on jobshop scheduling. Scheduling is one of the successful application area of constraint programming and these results assess the benefits of parallel computing on these problems.

6.5.1 The Basic Constraint Program

The constraint programs evaluated in this section all use edge-finding and not-first/not-last algorithms [2, 24, 18]. Their search procedures rank the machines one at a time, using a lexicographic heuristic to choose the machine to schedule next. The heuristic first selects the machine with the smallest local slack and breaks ties using the global slack. Ranking a machine consists of sequencing the activities. Figure 22 depicts the constraint program stating the traditional precedence and resource constraints and ranking the machines successively, while Figure 23 shows the ranking procedure. The ranking of a machine consists of finding the first available rank `r` (line 3), selecting an activity `a` that can be scheduled in position `r` (line 4), and trying to schedule `a` in position `r` (lines 5–7). Observe once again that the parallelization is completely transparent: the constraints and the search procedure are left unchanged by the parallelization.

```
1  Repository rep(nbt);
2  parall<rep>(i in 1..rep.getSize()) {
3     ParallelCPScheduler CP(horizon,rep);
4     CPActivity a[j in Jobs,t in Tasks](CP,duration[j,t]);
5     CPActivity makespan(CP,0);
6     CPUnaryResource r[Machines](CP);
7     minimize<CP> makespan.start()
8     subject to {
9        forall(j in Jobs,t in Tasks: t != Tasks.getUp())
10          a[j,t].precedes(a[j,t+1]);
11       forall(j in Jobs)
12          a[j,Tasks.getUp()].precedes(makespan);
13       forall(j in Jobs, t in Tasks)
14          a[j,t].requires(r[machine[j,t]]);
15    } using {
16       forall(m in Machines) by (r[m].localSlack(),r[m].globalSlack())
17          r[m].rank();
18       CP.post(makespan.start() == makespan.start().getMin());
19     }
20 }
21 rep.close();
```

Figure 22: A Parallel Constraint Program for Jobshop Scheduling.

Note that execution times in scheduling may often vary significantly according to the heuristic or search strategy. Thus, the rest of this section reports results on several settings.

6.5.2 Basic Results

We first report results on the ORB 1–5 benchmarks which are reasonably easy at this point. Tables 5 and 6 report these results for the DFS and LDS search controllers presented earlier.

```
1 void rank() {
2   while (isRanked()) {
3     int r = getRank();
4     selectMin(a in Activities: pos[a]. memberOf(r))(activity[a].getECT(),activity[a]. getEST())
5       try<CP>
6           CP.post(pos[a] == r);
7         |   CP.post(pos[a] != r);
8 }
```

Figure 23: Ranking a Resource in Jobshop Scheduling.

Runtime						Speedup (seq)				Speedup (1w)		
	seq	1w	2w	3w	4w	1w	2w	3w	4w	2w	3w	4w
ORB 1	15.53	16.45	9.76	5.69	4.48	0.94	1.59	2.72	3.46	1.68	2.89	3.67
ORB 2	13.08	14.06	7.31	5.09	3.47	0.93	1.78	2.57	3.76	1.92	2.76	4.05
ORB 3	56.54	179.89	43.84	22.15	14.52	0.31	1.28	2.55	3.89	4.10	8.12	12.38
ORB 4	9.81	15.52	6.50	4.71	3.68	0.62	1.50	2.08	2.66	2.38	3.29	4.22
ORB 5	2.31	2.54	1.37	1.31	0.88	0.90	1.68	1.76	2.62	1.85	1.93	2.88

Table 5: Jobshop Scheduling using DFS.

For DFS, the parallel implementation exhibits good speedups in general. On the most difficult problems (ORB 3 and ORB 1), the speedups are about 3.89 and 3.46 for 4 processors. The results for a single worker may exhibit some serious slowdown due to the change in search strategy but the parallelism nicely recovers from these pathological behaviors, producing significant superlinear speedups.[4]

For LDS, the situation is reversed and the parallel implementation with a single worker is typically faster than the sequential version. The results systematically show superlinear speedups with respect to both the sequential and single-worker implementations. These speedups range from 5.43 to 6.31 when compared to the sequential constraint program.

Runtime						Speedup (seq)				Speedup (1w)		
	seq	1w	2w	3w	4w	1w	2w	3w	4w	2w	3w	4w
ORB 1	64.81	47.27	23.60	14.72	11.75	1.37	2.74	4.40	5.51	2.00	3.21	4.02
ORB 2	37.99	28.13	12.08	9.86	6.99	1.35	3.14	3.85	5.43	2.32	2.85	4.02
ORB 3	243.76	226.29	81.31	64.18	41.23	1.08	2.99	3.79	5.91	2.78	3.52	5.48
ORB 4	50.90	28.45	12.52	11.53	8.37	1.79	4.06	4.41	6.08	2.27	2.46	3.39
ORB 5	25.43	17.81	8.85	4.92	4.03	1.42	2.87	5.16	6.31	2.01	3.61	4.41

Table 6: Jobshop Scheduling using LDS.

Runtime						Speedup (seq)				Speedup (1w)		
	seq	1w	2w	3w	4w	1w	2w	3w	4w	2w	3w	4w
LA-24	1459.30	1487.89	723.65	503.69	444.47	0.98	2.01	2.89	3.28	2.05	2.95	3.35
LA-25	131.09	136.24	69.03	45.09	34.60	0.96	1.90	2.91	3.78	1.97	3.02	3.94

Table 7: Jobshop Scheduling: Proof of Optimality (DFS).

Runtime						Speedup (seq)				Speedup (1w)		
	seq	1w	2w	3w	4w	1w	2w	3w	4w	2w	3w	4w
LA-21	208.49	192.85	144.01	67.49	43.60	1.08	1.44	3.08	4.78	1.33	2.85	4.42
LA-22	36.86	40.02	21.26	13.78	13.47	0.92	1.73	2.68	2.73	1.88	2.90	2.97
LA-23	7.24	7.79	3.58	3.31	2.71	0.93	2.02	2.18	2.67	2.17	2.35	2.87
LA-24	3122.15	3153.86	1627.98	1091.63	704.29	0.98	1.92	2.86	4.43	1.94	2.89	4.47
LA-25	160.53	178.65	94.38	56.14	39.52	0.89	1.71	2.85	4.06	1.89	3.18	4.52

Table 8: Jobshop Scheduling: High-Quality Solutions using IBDS.

6.5.3 Proof of Optimality

To reduce the effect of the heuristic, Table 7 reports the time for the optimality proofs (using DFS) on two reasonably difficult benchmarks: LA-24 and LA-25. The speedups are 3.28 and 3.78 with respect to the sequential implementation, and 3.35 and 3.94 over the parallel implementation with a single worker.

6.5.4 High-Quality Solutions using Iterative Bounded Discrepancy Search

Table 8 looks at the opposite situation: It studies the benefits of parallelism for the search of high-quality solutions with Iterative Bounded Discrepancy Search (IBDS). In particular, the table reports the times to find the optimal solution to a series of benchmarks (LA-21 to LA-25) using an iterative bounded discrepancy search. The constraint program uses the bound discrepancy controller in Figure 10 iteratively with 3, 6, 9, ... allowed discrepancies until the optimal solution is found. This is significantly more efficient than depth-first search on these benchmarks. The parallel constraint program yields superlinear speedups in this setting for LA-21, LA-24, and LA-25 over both the sequential and single-worker versions.

6.5.5 Hybrid Optimization: Local Search and Constraint Programming

Table 9 reports experimental results on a hybrid algorithm consisting of two phases: The first phase is the multi-start version of the tabu-search algorithm of Dell'Amico and Trubian [7], while the second phase is a constraint program with a depth-first controller. The multi-start local search is executed from 16 different random starting schedules. The local search

[4] Once again, the problem originates from the fact that the first worker publishes problems in the pool. Those problems are solved with the search procedure with may now choose a different machine to rank first.

Runtime						Speedup (seq)				Speedup (1w)		
	seq	1w	2w	3w	4w	1w	2w	3w	4w	2w	3w	4w
LA-22	19.05	19.03	10.85	10.18	8.23	1.00	1.75	1.87	2.31	1.75	1.87	2.31
LA-23	18.55	18.38	10.19	9.59	7.72	1.01	1.82	1.93	2.40	1.80	1.92	2.38
LA-24	2402.48	1498.27	735.30	554.39	453.39	1.60	3.27	4.33	5.29	2.03	2.70	3.30
LA-25	151.21	153.45	91.59	55.25	43.45	0.98	1.65	2.73	3.48	1.67	2.77	3.53
LA-26	23.46	23.34	15.27	11.96	9.57	1.01	1.54	1.96	2.45	1.53	1.95	2.43

Table 9: Jobshop Scheduling: Hybridization of Local Search and Constraint Programming.

finds very high-quality solutions on these problems and some have easy optimality proofs, which reduces the potential for parallelism. However, the more difficult problems (LA-24 and LA-25) exhibit strong (possibly superlinear) speedups.

6.6 Distributed Results

The distributed implementation was evaluated on jobshop scheduling problems using instances LA21-26 [12] and either a DFS or LDS search strategy. The hardware is a network of AMD Athlon at 2Ghz (Athlon 3800+) with a 100 Mbit switched LAN. The heuristic in the constraint-based scheduler first selects the machine with the smallest local slack and breaks ties using the global slack. The heuristics also ranks the resources by selecting first those tasks with the earliest completion times. The constraint-based scheduler only uses the basic edge-finding algorithm.

Table 10 (top) depicts the times and speedups for the optimality proofs. This computation is interesting since it is not subject to superlinear speedups, e.g., speedups due to finding higher-quality solutions sooner in the search. Only LA-21, LA-24, and LA-25 are shown, the proof of optimality being trivial for the other problems. The speedups are excellent, ranging from 9.25 to almost 15 on 16 machines. It is also interesting to compare the speedups of the parallel and distributed implementations. On a multicore parallel machines with 4 workers, the speedups are 3.28 and 3.78 on LA-24 and LA-25, while they are 3.20 and 3.53 on the distributed implementation.

Table 10 (bottom) is particularly interesting. It describes the experimental results for finding the optimal solution and proving optimality. In sequential, this search takes a long time for LA-22 and LA-24 when DFS is used, so the table reports the LDS times to give a lower bound on the speedups. (The sequential times for LDS are given in italics). The results indicates speedups ranging from 61 to 42,090 for 16 machines: the transparent parallelization thus transforms a sequential implementation that takes hours or days into a distributed implementation that takes a few seconds or a couple of minutes. Readers may wonder why the speedup stalls from 8 to 16 workers on LA-23. Shipping the address space of the

Runtime (seconds)						Speedup (vs seq)			
	seq	4w	8w	12w	16w	4w	8w	12w	16w
LA-21-P	5171.32	1373.56	701.13	466.73	347.95	3.76	7.38	11.08	14.86
LA-24-P	2992.15	935.86	467.82	316.03	235.81	3.20	6.40	9.47	12.69
LA-25-P	269.58	76.28	44.47	34.70	29.15	3.53	6.06	7.77	9.25
LA-22	*1037*	18512.45	16.37	19.1	16.92	–	*63*	*54*	*61*
LA-23	464254	966.12	7.1	10.54	11.03	480	65388	44046	42090
LA-24	*42874*	1862.75	888.14	698.62	358.10	*23*	*48*	*61*	*119*
LA-25	23337	93.28	49.16	39.91	34.20	250	474	584	682

Table 10: Jobshop Scheduling: Proof of Optimality and Full Search for DFS. Italics indicate a comparison against sequential LDS (DFS being much too slow).

Runtime (seconds)						Speedup (vs seq)			
	seq	4w	8w	12w	16w	4w	8w	12w	16w
LA-22	1037.29	178.80	88.27	73.10	62.53	5.80	11.75	14.19	16.59
LA-23	219.02	50.19	27.81	25.28	24.30	4.36	7.88	8.66	9.01
LA-24	42873.71	6614.8	3369.65	3298.08	890.49	6.48	12.72	13.00	48.15
LA-25	3050.96	536.69	403.78	385.83	148.44	5.68	7.56	7.91	20.55

Table 11: Jobshop Scheduling: Full Search (LDS).

Comet virtual machine to **one** peer over a 100Mbit network takes about 0.5 second. The running times for LA-23 with 16 workers is therefore bounded from below by this initial communication (e.g., $16 \cdot 0.5$). To improve efficiency one must switch to a gigabit network or use a multicast IP protocol[5].

Table 11 shows the results for finding optimal solutions and proving optimality using LDS. As expected, sequential LDS performs better than DFS. For the most part, the results show steady improvements as the number of workers increases with significant super-linear speed-ups on LA-24. The speedups on LA-23 stall once again for reasons explained above, while LA-22 and LA-25 exhibit speedups of 16 and 20.

An interesting observation about these preliminary results is that DFS outperforms LDS in a distributed environment, although this is clearly not the case in sequential. The availability of parallel and distributed hardware, and the transparent parallelization of constraint programs, may alter the benefits and limitations of exploration strategies.

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 [23]. We review

[5] multicast involves invasive changes to network routing and is unreliable.

related work in constraint programming to emphasize the originality of our proposal.

Parallel constraint logic programming was pioneered by the CHIP/PEPSys system [25] and elaborated in its successors [17]. 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 [25]. *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 [19] and reports experimental results in [20], 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 [19]. 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.

Disolver [9], is a C++ constraint programming library that supports transparent distribution on top of MPI. Search heuristics can be defined by overriding variable and value selection methods of the provided `Solver` class. However, search strategies appear to be limited to DFS, optionally randomized with restarts. It is not clear from [9] how subproblems are implemented and how flexible the search language is since the system is available only for commercial use.

Oz/Mozart was a pioneering system in the separation of nondeterministic programs and exploration strategies, which are implemented as search engines [21]. Schulte [22] 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.

Acknowledgment This work was partially supported through NSF awards IIS-0642906, DMI-0600384 and ONR award N000140610607.

References

[1] Blumhofe, R.D., C.E. Leiserson. 1994. Scheduling Multithreaded Computations by Work Stealing. *Proc. of the 35th Annual Symposium on Foundations of Computer Science (FOCS '94)*. Santa Fe, New Mexico, 356–368.

[2] Carlier, J., E. Pinson. 1994. Adjustment of Heads and Tails for the Jobshop Problem. *European Journal of Operational Research* **78** 146–161.

[3] Chandra, R., L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon. 2000. *Parallel Programming in OpenMP*. Morgan Kaufmann. ISBN:1558606718.

[4] Choi, C.W., M Henz, K.B. Ng. 2001. Components for State Restoration in Tree Search. *Proceedings of 7th International Conference on the Principles and Practice of Constraint Programming (CP'01)*. Paphos, Cyprus, 240–255.

[5] Cilk. 2006. *Cilk Reference Manual v5.4.2.3 (rev 2867)*. Supercomputing Technologies Group MIT Laboratory for Computer Science, Cambridge, MA. URL `http://supertech.lcs.mit.edu/cilk`.

[6] Corp., Intel. 2007. Teraflops research chip. www.intel.com/research/platform/terascale/teraflops.htm.

[7] Dell'Amico, M., M. Trubian. 1993. Applying Tabu Search to the Job-Shop Scheduling Problem. *Annals of Operations Research* **41** 231–252.

[8] Gent, I.P., T. Walsh. 1999. CSPLib: A Benchmark Library for Constraints. *Fifth International Conference on the Principles and Practice of Constraint Programming (CP'99)*. Alexandria, Virginia, 480–481.

[9] Hamadi, Y. 2003. Disolver : A Distributed Constraint Solver. Tech. Rep. MSR-TR-2003-91, Microsoft Research.

[10] Harvey, W. D., M. L. Ginsberg. 1995. Limited Discrepancy Search. *Proceedings of the 14th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann.

[11] http://www.gecode.org/. 2005. Generic Constraint Development Environment.

[12] Lawrence., S. 1984. Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques (supplement). Tech. rep., GSIA, CMU, Pittsburgh, PA.

[13] Michel, L., A. See, P. Van Hentenryck. 2006. Distributed constraint-based local search. *Proceedings of the 12th International Conference on the Principles and Practice of Constraint Programming (CP-2006)*. Nantes, France, 344–358.

[14] Michel, L., A. See, P. Van Hentenryck. 2006. High-level nondeterministic abstractions in c++. *12th International Conference on Principles and Practice of Constraint Programming. (CP'06)*. Lecture Notes in Computer Science, Nantes, France.

[15] Michel, L., P. Van Hentenryck. 2004. A Decomposition-Based Implementation of Search Strategies. *ACM Transactions on Computational Logic* **5**.

[16] Michel, L., P. Van Hentenryck. 2005. Parallel local search in comet. *Proceedings of the 11th International Conference on the Principles and Practice of Constraint Programming (CP-2005)*. Sitges, Spain, 430–444.

[17] Mudambi, S., J. Schimpf. 1994. Parallel clp on heterogeneous networks. *ICLP-94*.

[18] Nuijten, W. 1994. Time and Resource Constrained Scheduling: A Constraint Satisfaction Approach. Ph.D. thesis, Eindhoven University of Technology.

[19] Perron, L. 1999. Search Procedures and Parallelism in Constraint Programming. *Proceedings of the Fifth International Conference on the Principles and Practice of Constraint Programming (CP'99)*. Springer Verlag, Alexandria, Virginia, 346–360.

[20] Perron, L. 2002. Practical Parallelism in Constraint Programming. Narendra Jussien, François Laburthe, eds., *Proceedings of the Fourth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'02)*. Le Croisic, France.

[21] Schulte, C. 1997. Programming Constraint Inference Engines. *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, vol. 1330. Springer-Verlag, Schloss Hagenberg, Linz, Austria, 519–533.

[22] Schulte, C. 2000. Parallel Search Made Simple. *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*. Singapore. URL `http://web.it.kth.se/ schulte/paper.html?id=Schulte:TRICS:2000`.

[23] Schulte, C., M. Carlsson. 2006. Finite Domain Constraint Programming Systems. Francesca Rossi, Peter van Beek, Toby Walsh, eds., *Handbook of Constraint Programming*, chap. 14. Foundations of Artificial Intelligence, Elsevier Science Publishers, Amsterdam, The Netherlands, 495–526. URL `http://web.it.kth.se/ schulte/paper.html?id=SchulteCarlsson:CPH:2006`.

[24] Torres, P., P. Lopez. 2000. On Not-First/Not-Last Conditions in Disjunctive Scheduling. *European Journal of Operational Research* **127** 332–343.

[25] Van Hentenryck, P. 1989. Parallel Constraint Satisfaction in Logic Programming: Preliminary Results of CHIP within PEPSys. *Sixth International Conference on Logic Programming*. Lisbon, Portugal.

[26] Van Hentenryck, P. 2002. Constraint and Integer Programming in OPL. *Informs Journal on Computing* **14** 345–372.

[27] Van Hentenryck, P., L. Michel. 2005. *Constraint-Based Local Search*. The MIT Press, Cambridge, Mass.

[28] Van Hentenryck, P., L. Michel. 2006. Nondeterministic control for hybrid search. *Constraints* **11** 353–373.

[29] Warren, D.H.D. 1987. The SRI Model for Or-Parallel Execution of Prolog. Abstract Design and Implementation Issues. The IEEE Computer Society Press, ed., *Proceedings - 1987 Symposium on Logic Programming*. IEEE, 46–53.