

Distributed Constraint-Based Local Search

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. Distributed computing is increasingly important at a time when the doubling of the number of transistors on a processor every 18 months no longer translates in a doubling of *speed* but instead a doubling of the number of cores. Unfortunately, it also places significant conceptual and implementation burden on programmers. This paper aims at addressing this challenge for constraint-based local search (CBLS), whose search procedures typically exhibit inherent parallelism stemming from multistart, restart, or population-based techniques whose benefits have been demonstrated both experimentally and theoretically. The paper presents abstractions that allows distributed CBLS programs to be close to their sequential and parallel counterparts, keeping the conceptual and implementation overhead of distributed computing minimal. A preliminary implementation in *COMET* exhibits significant speed-ups in constraint satisfaction and optimization applications. The implementation also scales well with the number of machines. Of particular interest is the observation that generic abstractions of CBLS and CP, such as models and solutions, and advanced control structures such as events and closures, play a fundamental role to keep the distance between sequential and distributed CBLS programs small. As a result, the abstractions directly apply to CP programs using multistarts or restarts procedures.

1 Introduction

Moore's law [13], i.e., the prediction that the number of transistors per square inch on integrated circuits would double every 18 months used to translate into a doubling of speed. While it marches on, these additional transistors are now devoted to doubling the number of *cores* and gave rise to commodity multiprocessors. As a result, parallel and distributed computing now offer reasonably cheap alternatives to speed up computationally intense applications. However, parallel and distributed computing also places significant conceptual and implementation burden on programmers. The computational model adds another dimension in conceptual complexity (i.e., the need to handle multiple threads of executions) and programming abstractions are often expressed at a lower level of abstraction than their sequential counterparts. This has slowed the use of distributed computing, even for applications that exhibit natural parallelism as is typically the case in constraint satisfaction and optimization.

The parallelism exhibited in constraint satisfaction and optimization is often coarse-grained, requires minimal synchronization and coordination, and may

originate from restart, multistart, and population techniques, whose benefits have been demonstrated both experimentally and theoretically (e.g., [10, 6, 16, 7, 9]). Indeed, task durations are far more uniform and predictable than those associated to search nodes produced by traditional CP solvers. Yet very few implementations actually exploit this inherent potential: it suffices to look over the experimental results published in constraint programming conferences to realize this. The main reason is the absence of high-level abstractions for distributed computing that makes distributed programs substantially different from their sequential counterparts, even for applications that should be naturally amenable to distributed implementations.

This paper originates as an attempt to address this challenge for constraint-based local search (CBLS) and constraint programming (CP) applications which use multistart, restart, or population-based techniques. It presents abstractions that allows distributed CBLS or CP programs to be close to their sequential and parallel counterparts, keeping the conceptual and implementation overhead of distributed computing minimal. The abstractions naturally generalize their parallel counterparts [12] to a distributed setting: they include distributed loops, interruptions, and model pools, as well as shared objects. The resulting distributed programs closely resemble their parallel counterparts which are themselves close to the sequential implementations.

A preliminary implementation of the abstractions in COMET (using, among others, sockets, forks, and TCP) exhibits significant speedups on constraint satisfaction (e.g., Golomb rulers) and optimization (e.g., graph coloring) applications when parallelizing effective sequential programs. The implementation is shown to scale well with the number of machines, even when the pool of machines is heterogeneous (e.g., the machines have different processor frequencies and cache sizes). Together with the simplicity of the resulting CBLS programs, these results indicate that the distributed abstractions offer significant benefits for practitioners at a time when the need for large-scale constraint satisfaction and optimization or fast response time is steadily increasing.

It is also important to emphasize that the abstractions result from the synergy between recent modeling abstractions from CBLS and CP, such as the concepts of models and solutions [11, 8], the novel distributed abstractions presented herein, and advanced control structures such as events and closures [19].

The rest of the paper is organized as follows. Section 2 presents the novel abstractions. Section 3 introduces two language extensions, processes and shared objects, that are fundamental in implementing the abstractions. Section 4 sketches the implementation. Section 5 discusses related work, while Section 6 reports the experimental results and concludes the paper.

2 Distributed Constraint-Based Local Search

This section reviews the distributed abstractions of COMET. The main theme is to show that the distance between sequential and distributed COMET programs is small, making distributed computing far more accessible for CBLS than existing

<pre> 1. ThreadPool tp(3); 2. SolutionPool S(); 3. parall<tp>(i in 1..nbStarts) { 4. WarehouseLocation location(); 5. location.state(); 6. S.add(location.search()); 7. } 8. tp.close(); 9. cout << S.getBest().getValue(); </pre>	<pre> 0. string[] macs = ["m1","m2","m3"]; 1. MachinePool tp(macs); 2. shared{SolutionPool} S(); 3. parall<tp>(i in 1..nbStarts) { 4. WarehouseLocation location(); 5. location.state(); 6. S.add(location.search()); 7. } 8. tp.close(); 9. cout << S.getBest().getValue(); </pre>
--	---

Fig. 1. From Parallel to Distributed Multistart Constraint-Based Local Search.

libraries such as PVM and MPI. To demonstrate this significant benefit, the paper contrasts the parallel and distributed applications in COMET, since the parallel abstractions designed for shared memory multi-processors were shown to allow a small distance between sequential and parallel code [12].

Parallel Iterations The main parallel and distributed abstraction of COMET is the concept of parallel loops. Figure 1 depicts how to move from a parallel to a distributed implementation of a multistart CBLS for warehouse location. The CBLS for warehouse location was described in [18]: it is organized here as a *model* providing methods to state its constraints and objectives, and to search for a (near-optimal) *solution*. The left part of the figure depicts the parallel implementation, while the right part of the figure exhibits the distributed version.

The parallel implementation declares a thread pool consisting of 3 threads (line 1) and a solution pool to collect the solutions of multiple runs (line 2). The parallel loop is shown in line 3: it is the equivalent of a for-loop but it uses the thread pool to dispatch the iterations of the body to the threads in the pool. The body creates the warehouse model, states its constraints and objectives, searches for a solution, and adds the solution to the solution pool. The various iterations are synchronized in line 8, which then closes the thread pool. The objective value of the best found solution is displayed in line 9.

The distributed implementation is almost identical to the parallel implementation. Instead of thread pools, it uses a machine pool specifying which machines to use for the parallel loop (line 0–1, where the names of the machines are “m1”, “m2”, and “m3”). The solution pool is now *shared*, meaning that processes on the different machines may access it in a synchronized and distributed fashion. The rest of the code is identical to the parallel code, although the loop iterations will now execute on machines “m1”, “m2”, and “m3”.

The simplicity of the implementation is partly due to the advanced control structures of COMET and partly to optimization concepts such as models and solutions. These concepts are recent innovations in CBLS [11] and CP [8] and are fundamental in that they allow for a clean separation between the specificities of the models and parallel and distributed abstractions. Note also that the warehouse model could be implemented as a CBLS algorithm or a randomized branch

```

1. Boolean found(false);           1. Boolean found(false);
2. parall<p>(i in 1..nbStarts) {    2. parever<p> {
3.   ProgressiveParty pp();        3.   ProgressiveParty pp();
4.   pp.state();                  4.   pp.state();
5.   Solution s = pp.search();     5.   Solution s = pp.search();
6.   found := (s.getValue() == 0); 6.   found := (s.getValue() == 0);
7. } until found;                 7. } until found;

```

Fig. 2. Distributed Interruptions of Constraint-Based Local Search.

```

                                0. string[] macs = ["m1","m2","m3"];
1. WarehouseLocationFactory f(); 1. WarehouseLocationFactory f();
2. ModelPool mp(3,f);            2. DistrModelPool mp(macs,f);
3. SolutionPool S();            3. shared{SolutionPool} S();
4. parall<mp>(i in 1..nbStarts)  4. parall<mp>(i in 1..nbStarts)
5.   S.add(mp.search());         5.   S.add(mp.search());
6. mp.close();                  6. mp.close();

```

Fig. 3. From Parallel to Distributed Model Pools.

and bound algorithm with a computation limit. The distributed abstractions are independent of the underlying optimization technology.

Interruptions In constraint satisfaction, the goal consists in finding a feasible solution. Random restarts or multiple random searches have been shown to be a fundamental ingredient of CBLS and CP algorithms [10, 6]. Such algorithms are inherently parallel but an implementation must stop as soon as a solution has been found. Figure 2 depicts part of the code for a parallel and a distributed multistart algorithm for the progressive party problem.

Consider first the distributed implementation depicted in the left part of the figure. Line 1 declares a Boolean that specifies whether a solution was found and the parallel loop now terminates as soon as the Boolean becomes true (line 6) or when the fixed number of iterations is exhausted. The code (line 1–6) executes with either a thread or a machine pool. In the distributed implementation, different machines execute the model and are interrupted as soon as a feasible solution is found on another machine. Observe once again how easy it is to move from a parallel to a distributed implementation: it suffices to replace a thread pool by a machine pool. Consider now the right part of Figure 2. Here the code uses the `parever` instruction which iterates its body until a solution is found. Once again, the code is identical for the parallel and distributed implementation.

It is important to emphasize that the constraint satisfaction was not modified at all to be amenable to a distributed implementation or to allow for interruptions. This clean separation of concerns is one of the main advantages of the distributed abstractions presented herein. They naturally leverage sequential models and automate tedious aspects of distributed computing. Once again, the progressive party implementation can be either a CBLS algorithm or a randomized constraint program with a computation limit.

```

1.  interface Model {
2.      void state();
3.      Solution search();
4.      Solution search(Solution s);
5.      Solution search(Solution s1,Solution s2);
6.      Solution search(Solution[] s);
7.  }
8.  interface ModelFactory { Model create(); }

```

Fig. 4. The Model and Model Factory Interfaces.

```

1.  found := false;
2.  while (!found) {
3.      Solution op[k in RP] = pop[k];
4.      parall<mp>(k in RP) {
5.          select(i in RP, j in RP: i != j) {
6.              pop[k] = mp.search(op[i],op[j]);
7.              if (pop[k].getValue() == 0)
8.                  found := true;
9.          } until found;
10. }

```

Fig. 5. The Core of the Distributed Implementation for Finding Golomb Rulers.

Model Pools One of the limitations of thread and machine pools is the necessity of creating and stating models multiple times. The concept of model pool was introduced in [12] to remedy this limitation by leveraging the concept of models and it naturally generalizes to a distributed setting. A model pool receives, as parameters, a factory to create models and the number of models to create for parallel execution. When a solution is required, the model pool retrieves, or waits for, an idle model and searches for a solution inside a new thread. Distributed model pools simply receive the names of the machines instead of the number of models. They also retrieve an idle model and search for a solution on one of the idle machines.

Figure 3 illustrates model pools on the warehouse location problem again. Consider first the left part of the figure that describes the parallel implementation. The COMET code first declares a factory to create the warehouse location models (line 1), a model pool (line 2), and a solution pool (line 3). The parallel loop simply asks the model pool for solutions (method `mp.search()`) and stores them in the solution pool. Consider now the right part of the figure: it is almost identical but uses distributed model pools.

For completeness, Figure 4 specifies the interface for models and model factories. A model provides methods to state its constraints and objectives and to search for solutions. The search for solutions may receive zero, one, or more solutions as starting points, allowing for a variety of search algorithms. A model factory simply creates models, which may or may not use different search procedures. Once again, the concept of models and solutions, fundamental abstractions in CBLS and CP, are critical in moving naturally from a sequential to a distributed implementation.

Finally, Figure 5 depicts the core of an evolutionary CBLS for finding Golomb rulers. The algorithm, used in our experimental results, integrates many of the abstractions just presented: parallel loops, interruptions, and model pools. The figure depicts the part of the COMET program searching for a Golomb whose length is smaller than a given number. In the full program, this core is embedded in an outer loop that searches for rulers of smaller and smaller lengths. The core of the algorithm maintain a population `pop` of (infeasible) rulers and generates new generations of the population until a solution is found (lines 4–9). To generate a new ruler, the algorithm selects two individuals in the population (line 5), crosses them, and applies a CBLS minimizing the number of violations to generate a new ruler (line 6). The distributed implementation uses a distributed model pool and generates the new population in parallel on different machines. Each such distributed computation is interrupted as soon as a feasible ruler is found, i.e., a ruler with no violations (lines 7–8). Observe once again the simplicity of the distributed implementation that closely resembles its sequential counterpart and is almost identical to the parallel implementation, since it only replaces model pools by distributed model pools.

3 Enabling Technology

To support the abstractions presented above, only two language extensions are required: processes and shared objects. This section briefly reviews them.

Processes COMET features a `process` construct to fork a new process on a specific machine. For instance, the code

```

1. Queens q(1024);
2. SolutionPool S();
3. process("bohr") {
4.   S.add(q.search());
5. }
```

forks a new COMET process on machine `bohr` to find a solution to the 1024-queens problem. When a COMET process is created, it is initialized with a copy of its parent’s runtime. The child process starts executing the body of the `process` instruction, while the parent continues in sequence. There is an implicit rendezvous when the parent is about to terminate. Observe that, since the child has its own copy of the runtime, operations performed in the child are only visible on its own runtime and do not affect the parent’s objects and data structures. In particular, in the above code, the solution to the queens problem is only available in the child, the parent’s model and its solution pool `S` being left unchanged.

Shared Annotations The second abstraction was already mentioned earlier. To allow distributed algorithms to communicate naturally, COMET features the concept of shared objects that are visible across processes. Consider the code

```

1. Queens q(1024);
2. shared{SolutionPool} S();
```

```

3. process("bohr") {
4.   S.add(q.search());
5. }

```

The solution pool S is now a shared object that lives in the runtime of the parent but is visible in the runtime of the child. As a result, when line 4 is executed, the solution is added into the parent's pool. The child in fact does not have a solution pool, simply a reference to the parent's pool since the object is shared. Shared objects allow processes to communicate naturally by (remote) method invocations very much like systems such as CORBA and MPI. The linguistic overhead in COMET is really minimal however, keeping the distance between sequential and distributed programs small.

It is important to connect shared objects and process creation. When a process is created, the parent's runtime is cloned, except for shared objects that are replaced by proxy objects with the same interface. These proxy objects encapsulate a remote reference to the original object and transform method invocations into remote method invocations using traditional serialization techniques.

Events on shared objects are also supported. A child process may subscribe to events of shared objects and be notified when these events occur. This functionality, which is highly convenient for implementing interruptions, allows this fundamental control abstraction of COMET [19] to be used transparently across threads and processes.

4 Implementation

This section sketches the implementation of the distributed abstractions which consists of three parts:

1. source to source transformations of the COMET programs to replace parallel loops by traditional loops, closures, and barriers;
2. machine and model pools which receive closures in input and creates processes to execute them on remote machines;
3. processes and shared objects which are now integral parts of the COMET runtime system.

4.1 Source to Source Transformation

The first step of the implementation consists of replacing parallel loops by sequential loops and barriers. This step is parameterized by the pool and thus identical in the parallel and distributed implementations. It is illustrated in Figure 6 for warehouse location. The left of the figure shows the original COMET program, while the right part depicts the transformed program. The transformed program declares a shared barrier (line 1) to synchronize the loop executions. The barrier is initialized to zero, incremented for each iteration (line 3), and decremented when an iteration is completed (line 8). The barrier is used in line

```

1.  parall<mp>(i in 1..nbStarts) {
2.    WarehouseLocation location();
3.    location.state();
4.    S.add(location.search());
5.  }
6.  mp.close();

1.  ZeroWait b(0);
2.  forall(i in 1..nbStarts) {
3.    b.incr();
4.    closure C {
5.      WarehouseLocation location();
6.      location.state();
7.      S.add(location.search());
8.      b.decr();
9.    }
10.   mp.submit(C);
11.  }
12.  b.wait();
13.  mp.close();

```

Fig. 6. Source to Source Transformation for Parallel Loops.

12 to ensure that the main thread continues in sequence only when all the iterations are completed. The parallel loop is replaced by a sequential loop (line 2), which creates a closure `C` which is then submitted to the pool. The thread executing the code thus simply forwards the closures, one for each iteration, to the pool where the parallel or distributed execution takes place.

Figure 7 generalizes the implementation to support interruptions. It shows the source to source transformation for the progressive party problem, the only differences being in line 5–8. The transformation creates an event-handler that calls method `terminate` on the pool whenever the Boolean `found` changes value (and becomes true), interrupting all the threads or processes in the pool. In addition, the pool now tests whether the Boolean is false before executing the body of the iteration. Once again, this transformation is valid for all parallel pools. This genericity is possible because all pools must implement the same `ParallelPool` interface depicted in Figure 8.

4.2 Machine and Distributed Model Pools

As mentioned earlier, the main thread only dispatches the closures, one for each iteration of the loop, to the parallel pool. It is thus the role of the parallel pools to execute these closures in parallel or in a distributed fashion. Figure 9 depicts the implementation of the machine pool.

Machine Pools The machine pool uses two producer/consumer buffers: one for the machines and one for the closures to execute (line 2). It uses a Boolean `cont` to determine if the pool should continue execution (line 2), and an object `interrupt` to handle interruption (line 3). The closures are produced by the execution of the parallel loop as shown in the source to source transformation. The resulting code (see Figure 6) calls method `submit` which produces a closure.

The core of the machine pool is in its constructor (lines 4–22). It creates the buffers (line 6–7), the interrupt handler (line 8), and produces all the machines in line 9. Observe that the machine buffer is shared, since upon termination processes must release their host and thus must access the buffer `macs` remotely.

```

1.  parall<mp>(i in 1..nbStarts) {
2.      ProgressiveParty pp();
3.      pp.state();
4.      Solution s = pp.search();
5.      found := (s.getValue() == 0);
6.  } until found;
7.  mp.close();

1.  ZeroWait b(0);
2.  forall(i in 1..nbStarts) {
3.      b.incr();
4.      closure C {
5.          when found@changes()
6.              mp.terminate();
7.          in {
8.              if (!found) {
9.                  ProgressiveParty pp();
10.                 pp.state();
11.                 Solution s = pp.search();
12.                 found:=(s.getValue()==0);
13.             }
14.         }
15.         b.decr();
16.     }
17.     mp.submit(C);
18. }
19. b.wait();
19. mp.close();

```

Fig. 7. Source to Source Transformation for Parallel Loops with Interruptions.

```

1.  interface ParallelPool {
2.      void submit(Closure c);
3.      void terminate();
4.      void close();
5.      int getSize();
6.  }

```

Fig. 8. ParallelPool interface

The distributed computing code lies in line 10–21. It creates a thread responsible for dispatching closures to the various machines as long as the machine pool must execute. The essence of the implementation is in lines 12–19 that are executed at each iteration. First, the implementation consumes a machine or waits until such a machine is available (i.e., the buffer `macs` contains an available host) (line 12). Once a machine is obtained, the implementation consumes a closure to execute or waits for a closure to become available (line 13). If the machine or the closure are `null`, then the model pool has terminated and execution completes. Otherwise, the implementation creates a process on the available machine, which executes the closure (line 17) and then returns the machine to the buffer. The closure execution may be interrupted, which happens if the closure calls method `terminate` (see line 6 in the right part of Figure 7). The `break` implementation was discussed in [12] and uses events with either exceptions or continuations. Finally, the `Interrupt` class is depicted in lines 27–31 and simply encapsulates an event. When method `terminate` on the machine pool is called, the event `raised` is notified and the closure call in line 17 is interrupted.

```

1. class MachinePool {
2.   StringBuffer macs;ClosureBuffer closures;Boolean cont;
3.   Interrupt interrupt;
4.   MachinePool(string[] mac) {
5.     cont = new Boolean(true);
6.     closures = new ClosureBuffer(mac.rng().sz());
7.     macs = new shared{StringBuffer}(mac.rng().sz());
8.     interrupt = new shared{Interrupt}();
9.     forall(i in mac.rng()) macs.produce(mac[i]);
10.    thread {
11.      while (cont) {
12.        string m = macs.consume();
13.        Closure v = closures.consume();
14.        if (m != null && v!=null)
15.          process(m) {
16.            break when interrupt@raised()
17.            call(v);
18.            macs.produce(m);
19.          }
20.        }
21.      }
22.    }
23.    void submit(Closure v) { closures.produce(v); }
24.    void close() {cont := false;macs.terminate();closures.terminate();}
25.    void terminate() { interrupt.raise(); close(); }
26.  }
27. class Interrupt {
28.   Event raised();
29.   Interrupt() {}
30.   void raise() { notify raised();}
31. }

```

Fig. 9. The Machine Pool Implementation.

It is interesting to trace the steps involved in executing the progressive party code in Figure 7. First, observe that the closure `v` in line 13 consists of the body of the loop and resides in the parent process. When the child is created, it inherits a copy of that closure and executes it locally. It thus creates the progressive party object (line 9), states the constraints (line 10), and searches for a solution on the remote machine (line 11). Second, if the search finds a solution (line 12), it assigns the Boolean to true. This Boolean also resides in the parent and was copied into the child. Since it is not shared, only the child copy is affected. The code in line 6 of the right-hand side Figure 7 executes and calls `terminate` on the machine pool to raise an exception caught in line 16 of the other processes interrupting their execution of line 17 in Figure 9. This is possible as `interrupt` is shared and all processes subscribe to its `raised` event.

Distributed Model Pools Distributed model pools are modeled after model pools. The implementation creates one process per machine and associates a unique model to it. The process then becomes a server, waiting for service requests.

4.3 The Runtime System

It remains to discuss how to implement the extensions to the runtime system.

Distributed Forks COMET processes have the same semantics as traditional `fork` system calls available in any modern operating system: they simply add the ability to spawn the child on a different host. The COMET implementation uses a small daemon process on all the machines allowed to host COMET processes. The daemon listens on a TCP port for process requests. When the COMET virtual machine creates a process, it contacts the daemon on the target machine. In response, the daemon forks itself and loads the COMET executable in the new child which uses the dynamic TCP port number for further exchanges. The parent then ships its runtime data structures (including the stack and the heap) to the child over the TCP connection, using traditional serialization and de-serialization. The runtime data structures are re-created at the exact same virtual memory addresses on the child process. The final step consists of replacing the shared objects by proxies. Each proxy then holds a reference to the network connection and the address of the parent process.

Shared Objects Method invocation on shared objects is completely transparent. It is the role of the proxy to contact the owner process, serialize the arguments, and de-serialize the results. Shared objects may be used as arguments and are replaced by a proxy. The remaining parameters must be serializable, which is the case for fundamental abstractions such as solutions. On the receiver side, the remote method invocation is de-serialized. Since it contains the actual address of the receiver, the implementation performs a standard method dispatch on the de-serialized arguments. Since this receiver is shared, it is also a monitor, automatically synchronizing remote and local invocations.

Remote Events Events can be handled with the same techniques. Since events are managed through a publish-subscribe model [19], a subscription on a remote object results in a message to the true receiver to notify the subscriber. When the event takes place, the remote object remotely publishes the event, inducing the subscriber to execute its closure locally.

5 Related Work

This section briefly reviews other initiatives in distributed computing.

Languages OZ is a concurrent language that supports parallel and distributed computing. In [17], it has been used to implement a distributed search engine that implements a protocol and a search node distribution strategy for distributed DFS. Like COMET, it argues in favor of a strong separation of search from concurrency and distribution. In contrast to COMET, search spaces are not distributable structures and the communication protocol relies on a combination of search path and recomputation to ship search nodes to workers.

OPENMP [3,2] is a preprocessor for parallel loops in C and Fortran. The parallel abstractions of COMET and OPENMP share the same motivations as both type of systems aim at making parallel computing widely accessible by reducing the distance between sequential and parallel code. However, as discussed in [12], the parallel abstractions of COMET are simpler and richer, primarily because of its advanced control abstractions. This paper goes one step further: it shows that the abstractions naturally generalize to distributed computing, opening a new realm of possibilities. FORTRESS [1] is a new language aimed at supporting high performance applications. To our knowledge, it is at the specification stage.

Libraries Libraries for distributed computing focuses on various forms of message passing. The actual implementations can be realized at different levels of abstraction: sockets or messaging. Sockets are very low-level and are best viewed as an implementation technology. MPI and PVM impose a significant burden for optimization software that must be explicitly reorganized to match the client-server architecture. Parallel Solver [14] is a domain-specific solution to parallelize the exploration of complete search tree but it focuses on SMP systems only.

Object Models DCOM and CORBA introduce an object model for distributed computing where remote method calls are performed transparently. However, both impose significant burden on programmers as discussed in [15]. Applications must be redesigned to fit a client-server model, object interfaces must be specified in a separate language (IDL) to generate proxies, and programmers are exposed to low-level threading and memory management issues. All these limitations are addressed by COMET's abstractions for distributed CBLs.

Distributed CSPs distributed CSPs are formalized and the first distributed asynchronous backtracking search algorithm is introduced in [20]. DiCSPs take a fundamentally different approach as the set of variables and constraints are themselves distributed and is more directly related to agent-based searching.

6 Experimental Results

The Benchmarks The benchmarks consist of a multistart version of the tabu-search algorithm for graph-coloring from [4] and the hybrid evolutionary algorithm for Golomb rulers mentioned earlier [5]. The coloring algorithm is an optimization application minimizing the number of colors. It was evaluated on the benchmarks R250.5 (250 vertices, 50% density and best coloring=65) and R250.1c (250 vertices, 90% density and best coloring=64). Note that these problems have 250 variables and thousands of constraints, which makes them interesting since distributed implementations must copy the entire address space for each process. The COMET program for Golomb rulers is particularly interesting. At each iteration, it generates a new population of rulers, unless it finds a feasible ruler in which case the computation terminates. There is significant variance in how fast a feasible ruler is found (within and across generations) and hence it

Names	Proc.	Cache	Freq.	bogoMIPS	T(R250.1c)	T(R250.5)
m1	P4	1M	3.0	5898	95.2	96.33
m2	P4	1M	2.8	5570	109.7	
m3	P4	512K	2.4	4771	194.3	150.59
m4	Xeon	512K	2.4	4784	202.6	
m5	P4	512K	2.4	4757	210.4	
m6	P4	512K	2.0	3932	248.6	

Fig. 10. Specifications of the Machines.

is interesting to assess the performance of the distributed implementation. The program was run until the optimum (of length 72) was found.

The Machines The benchmarks were executed on an heterogeneous pool of machines all running Debian Linux. The machines differ, not only in their clock frequencies but also in the type of processors and the size of their caches. The features of the machines are depicted in Figure 10 which specifies the processor type, the cache size, the clock frequency, the bogoMIPS speed estimate of Linux, and the time to execute R250.1c and R250.5 in deterministic mode. The boldfaced times for coloring on machine m3 are used as reference in Figure 11. As can be seen, there are significant speed differences between the machines. The results use prefixes of the worst-case ordering for COMET: m1, m2, . . . , m6; for instance, on four machines, the pool consists of machines m1, . . . , m4. Since the machines are increasingly slower and the sequential times are given on the fastest machine, the speed-ups are negatively affected by the heterogeneity. Nevertheless, they also show the flexibility of the abstractions.

B	N	m_s	μ_s	σ_s	μ_T	S_{m3-6}	S_{m3}	B	N	μ_T	S_{m1}	S_{m3}
R250.1c	1	64	64.50	0.50	193.16	1.11	1	R250.5	m1	100.36	0.95	1.50
	2	64	64.30	0.46	103.32	2.07	1.87		m1-2	52.09	1.84	2.89
	3	64	64.40	0.49	76.95	2.78	2.51		m1-3	42.33	2.27	3.55
	4	64	64.40	0.49	62.69	3.41	3.08		m1-4	38.33	2.51	3.92
							m1-5		37.47	2.57	4.01	
							m1-6		28.74	3.35	5.23	

Fig. 11. Results on Graph Coloring (Heterogeneous mix of machines).

The Graph Coloring Results Figure 11 depicts the results on graph coloring. Each line reports the average and deviation for 50 runs. The left part of the figure gives the results for the most homogeneous set of machines (m3-m6) on problem R250.1c. It reports the best coloring found (m_s), the average coloring (μ_s), and the standard deviation on the quality. It then reports the executing times in seconds (μ_T), and the speed-ups with respect to the average speed S_{m3-6} and with respect to the best machine in the experiments S_{m3} . Observe that, on the first three (roughly similar) machines, the speedups are about 2.78 and 2.51. With four machines, they are about 3.41 and to 3.08, which is still

N	S^*	μ_T	σ_T	S_{m3}	N	S^*	μ_T	σ_T	S_{m3}
m3	72	42.30	40.21						
1	72	66.51	56.22	0.64	3	72	19.85	19.06	2.13
2	72	31.30	23.24	1.35	4	72	11.02	6.95	3.84

Fig. 12. Results on Golomb Ruler (Homogeneous mix of machines).

excellent especially with the last machine being slower. The right part of the figure depicts the benchmarks for R250.5 using all the machines. It reports the average times in seconds, as well as the speedups with respect to $m1$ (the fastest machine) and $m3$ (about the average machine). Observe first the speedup of 1.84 on the fastest two machines (wrt $m1$) which indicates that the distributed implementation does not lose much compared to the parallel implementation in [12] despite having to copy the runtime data structures. Compared to the average machine $m3$, the results are quite impressive giving a 5.23 speedups with 6 machines. Even compared to the fast $m1$, the results remain convincing.

The Golomb Results Figure 12 depicts the same results on finding Golomb rulers using the homogeneous set of machines $m3 - 6$. The table reports the CPU time μ_T , its standard deviation σ_T , and the speedups with respect to the best machine in the set. Once again, the speedups are quite impressive. Moreover, the distributed implementation has a very interesting side-effect: it dramatically reduces the standard deviation on the execution times. This comes from the ability to interrupt the search, once a feasible solution has been found and, of course, the concurrent exploration from multiple startpoints.

7 Conclusion

The paper presented abstractions that allows distributed CBLS programs to be close to their sequential and parallel counterparts, keeping the conceptual and implementation overhead of distributed computing minimal. The new abstractions strongly rely on generic abstractions of CBLS and CP, such as models and solutions, and advanced control structures such as events and closures. A preliminary implementation in COMET exhibits significant speed-ups in constraint satisfaction and optimization applications. The implementation also seem to scale well with the number of machines, although more extensive eperimental evaluations are necessary. Overall, the simplicity of the abstractions and the resulting distributed CBLS, together with the excellent performance behavior of our implementation, indicates that distributed computing should become much more mainstream in years to come.

References

1. Allen, Chase, Luchangco, Maessen, Ryu, Steele, and Tobin-Hochstadt. The Fortress Language Specification, V0.866. Sun microsystems, Feb 2006.

2. R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000. ISBN:1558606718.
3. L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5:46–55, Jan-March 1998.
4. R. Dorne and J.K. Hao. *Tabu Search for Graph Coloring, T-Colorings and Set T-Colorings*, chapter Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization, pages 77–92. Kluwer Academic Publishers, 1998.
5. I. Dotu and P. Van Hentenryck. A Simple Hybrid Evolutionary Algorithm for Finding Golomb Rulers. In *Evolutionary Computation, 2005*, pages 2018–2023. IEEE, 2005.
6. C. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems. *Journal of Automated Reasoning*, 24(1/2):67–100, 2000.
7. G. Harm and P. Van Hentenryck. A MultiStart Variable Neighborhood Search for Uncapacitated Warehouse Location. In *Proceedings of the 6th Metaheuristics International Conference (MIC-2005)*, Vienna, Austria, August 2005.
8. Ilog Solver 6.2. Documentation. Ilog SA, Gentilly, France, 2006.
9. M. Laguna. *Handbook of Applied Optimization*, chapter Scatter Search, pages 183–193. Oxford University Press, 2002.
10. M. Luby, A. Sinclair, and Zuckerman. D. Optimal Speedup of Las Vegas Algorithms. *Inf. Process. Lett.*, 47(4):173–180, 1993.
11. L. Michel and P. Van Hentenryck. A Constraint-Based Architecture for Local Search. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 101–110, Seattle, November 2002.
12. L. Michel and P. Van Hentenryck. Parallel Local Search in Comet. In *Eleventh International Conference on Principles and Practice of Constraint Programming*, Stiges, Spain, 2005.
13. G.E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.
14. L. Perron. Search Procedures and Parallelism in Constraint Programming. In *Proceedings of the Fifth International Conference on the Principles and Practice of Constraint Programming*, pages 346–360, Alexandria, Virginia, Oct. 1999.
15. M. Philippsen and M. Zenger. JavaParty - Transparent Remote Objects in Java. *Concurrency: Practice & Experience*, 6:109–133, 1995.
16. G. Resende and R. Werneck. A Hybrid Multistart Heuristic for the Uncapacitated Facility Location Problem. Technical Report TD-5RELRR, AT&T Labs Research, 2003. (To appear in the European Journal on Operations Research).
17. Christian Schulte. Parallel search made simple. In *Proceedings of TRICS, a post-conference workshop of CP 2000*, Singapore, September 2000.
18. P. Van Hentenryck. *Constraint-Based Local Search*. The MIT Press, Cambridge, Mass., 2005.
19. P. Van Hentenryck and L. Michel. Control Abstractions for Local Search. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming*, pages 65–80, Cork, Ireland, 2003.
20. Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *Knowledge and Data Engineering*, 10(5):673–685, 1998.