

Parallel Local Search in Comet

Laurent Michel¹ and Pascal Van Hentenryck²

¹ University of Connecticut, Storrs, CT 06269-2155

² Brown University, Box 1910, Providence, RI 02912

Abstract. The availability of commodity multiprocessors offers significant opportunities for addressing the increasing computational requirements of optimization applications. To leverage these potential benefits, it is important however to make parallel processing easily accessible to a wide audience of optimization programmers. This paper addresses this challenge by proposing parallel programming abstractions that keep the distance between sequential and parallel local search algorithms as small as possible. The abstractions, that include parallel loops, interruptions, and thread pools, are compositional and cleanly separates the optimization program and the parallel instructions. They have been evaluated experimentally on a variety of applications, including facility location and coloring, for which they provide significant speedups.

1 Introduction

With the availability of commodity multiprocessors (e.g., dual Apple G5s) and the advent of multi-core chips (e.g., the 870 processor from AMD and the forthcoming Cell Architecture from Sony/IBM/Toshiba), parallel processing is becoming widely affordable. It also provides significant opportunities to meet the increasing computational requirements of optimization applications as the field moves towards large-scale, online, and stochastic optimization. Indeed, optimization applications often exhibit significant parallelism. For instance, multistart local search, hybrid local search, cooperating local search, and online stochastic optimization all involve multiple, largely independent, searches that only communicate loosely and asynchronously. Similarly, search in constraint programming can also be viewed as largely independent searches, communicating computation paths [3, 9] or subproblems [8].

Unfortunately, a major obstacle in exploiting these opportunities is the additional expertise required to write parallel algorithms, since optimization applications already demand mathematical sophistication, domain expertise, and programming abilities. It is thus important to leverage the promises of multiprocessors as transparently as possible, imposing as small a burden as possible on optimization modelers and programmers.

This paper describes an attempt to address this challenge. It proposes high-level abstractions for parallel programming in COMET that exploit multiprocessors in a transparent way. The parallel abstractions have a number of desirable properties. They are natural counterparts to sequential constructs, ensuring a

```

1.  synchronized class Count {
2.      int c;
3.      Count(int i) { c = i; }
4.      void incr(int n) { c += n; }
5.      int get() { return c; }
6.  }
7.      Count c(0);
8.      forall(i in 1..2) {
9.          thread
10.             forall(k in 1..200)
11.                 c.incr(i);
12.     }

```

Fig. 1. Illustrating Threads and Monitors in COMET.

small distance between the sequential and parallel versions of the same algorithm. The abstractions are general-purpose and widely applicable, yet they provide a natural vehicle to express a variety of parallel optimization algorithms.¹ The abstractions are compositional and cleanly separate the optimization models from the parallel code, allowing modelers/programmers to reuse their existing models when exploiting parallelism. Finally, their implementation leverages the advanced control abstractions of COMET such as events and first-order closures [10] and is primarily based on source-to-source transformations.

The rest of the paper specifies the abstractions, illustrates their potential uses on various applications and describes their implementations. Section 2 reviews the enabling technologies (i.e., threads and monitors) underlying the parallel abstractions. Section 3 is the core of the paper and describes the abstractions, their uses, and their implementations. Section 4 describes the experimental results.

2 Enabling Concurrent Technologies

Threads and monitors are the enabling technology for the parallel abstractions of COMET. Although these concurrent programming extensions resemble those of Java, the subtle differences between them contribute in making COMET’s abstractions natural, convenient, and easier to implement. In particular, threads exhibit a nice synergy with first-order closures, the foundation of many control abstractions of COMET [10].

Figure 1 illustrates both threads and monitors in COMET. The class `Count` (lines 1–6) encapsulates a counter. Due to the keyword `synchronized`, the class is a monitor: its method invocations execute in mutual exclusion. An instance of the class is declared in line 7. The core of the program is a loop creating two threads (lines 8–12). The body of each thread, executed upon the thread creation, increments the counter `c` 200 times, by 1 for the first thread (`i=1`) and by 2 for the second thread (`i=2`).

Threads have their own version of the COMET runtime, including a stack for function and method calls, and are implemented by native threads of the operating system. Threads are first-class objects in COMET but, unlike Java, they do not have to extend an existing class. Rather, upon creation, they execute a closure specified by their statement (e.g., lines 10–11 in Figure 1) and the current environment. This design decision, i.e., the ability to use a closure as the thread

¹ In that sense, they are orthogonal and complementary to parallel, black-box, implementations of solvers and search algorithms [9].

```

1. synchronized class Buffer {
2.     int[] buffer;
3.     int sz;
4.     int nb;
5.     Condition full;
6.     Condition empty;
7.     Buffer(int s);
8.     void produce(int s);
9.     int consume();
10. }
11. Buffer::Buffer(int s) {
12.     sz = s;
13.     nb = 0;
14.     buffer = new int[0..sz-1];
15. }
16. void Buffer::produce(int s) {
17.     if (nb == sz) full.wait();
18.     buffer[nb++] = s;
19.     empty.signal();
20. }
21. int Buffer::consume() {
22.     if (nb == 0) empty.wait();
23.     int v = buffer[--nb];
24.     full.signal();
25.     return v;
26. }
27. Buffer b(2);
28. thread consumer {
29.     for(int k=0;k<100;k++) {
30.         int c = b.consume();
31.         cout << "-" << c << flush;
32.     }
33. }
34. thread producer {
35.     for(int k=0;k<100;k++) {
36.         b.produce(k);
37.         cout << "+" << k << flush;
38.     }
39. }

```

Fig. 2. A Simple Producer/Consumer Pattern in COMET.

code, is fundamental in keeping the distance between sequential and parallel code small in COMET. Observe that, in Figure 1, the two threads naturally increment the counter by different amounts thanks to the use of closures.

Figure 2 shows a COMET program implementing a producer/consumer monitor. Class `Buffer` (lines 1–10) specifies a buffer that can store up to `sz` integers. Method `produce`s stores an integer in the buffer, while method `consume`s retrieves an integer from it. The monitor uses two *conditions*, `full` and `empty`, to synchronize the producers and consumers. When the buffer is full, the producers wait until there is room in the buffer (line 17). When the buffer is empty, the consumers wait until some integer is available (line 22). The producers and consumers also notify their respective conditions when they produce and consume integers (lines 19 and 24 respectively). As is traditional, when a thread waits on a condition, it releases the monitor and must re-acquire it upon reactivation. Observe that, unlike Java, COMET may use several conditions inside a monitor which, once again, simplifies several concurrent programming patterns. Here is the beginning of the output of this COMET program when executed on an Apple G5 with 2 processors: +0+1-1-0+2+3-3-2+4+5-5-4+6+7-7-6+8+9-9-8...

3 Parallel Abstractions

The parallel abstractions of COMET support various parallel loops, interruptions, as well as thread and model pools. This section describes the abstractions, illustrates their uses, and sketches their implementations. The abstractions are presented in increasing order of complexity and functionalities.

```

1. Buffer b(2);
2. pardo {
3.     forall(k in 0..99) {
4.         int c = b.consume();
5.         cout << "-" << c << flush;
6.     }
7. } | {
8.     forall(k in 0..99) {
9.         b.produce(k);
10.        cout << "+" << k << flush;
11.    }
12. }

```

Fig. 3. Revisiting the Simple Producer/Consumer Pattern in COMET.

3.1 Parallel Execution

The Abstractions COMET provides a `parall` construct, as the parallel counterpart to the sequential `forall` loop. Reconsider the counter example from Figure 1. It could be rewritten, and expanded, as follows:

```

1. Count c(0);
2. parall(i in 1..2)
3.     forall(k in 1..200)
4.         c.incr(i);
5. cout << "value: " << c.get() << endl;

```

Lines 2–4 feature a parallel loop in COMET: the loop body may be executed in parallel for all values of the loop parameter. Operationally the `parall` instruction 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. Hence the COMET code displays the correct value (600) of the counter in line 5, since the output instruction is guaranteed to execute only after the completion of the `parall` instruction (which was not the case for the program in Figure 1). The instructions have the same effect as the sequential code:

```

1. Count c(0);
2. forall(i in 1..2)
3.     forall(k in 1..200)
4.         c.incr(i);
5. cout << "value: " << c.get() << endl;

```

showing how easy it is to move from a sequential to a parallel implementation. Figure 3 revisits the producer/consumer pattern from Figure 2 and illustrates its implementation with the `pardo` construct of COMET. As is the case with the `parall` construct, the instruction following line 12 in the program (if any) only executes after completion of the `pardo` instruction.

Illustrations Many optimization algorithms apply a local search multiple times. This is the case for multistart or iterated local searches that apply a local search repeatedly from (typically) different starting solutions; hybrid evolutionary algorithms and scatter search that apply a local search on a population of solutions; and online stochastic optimization where a local search produces solutions to a variety of scenarios before making a decision. These algorithms exhibit inherent parallelism which an optimization system should be able to exploit naturally, without requiring significant expertise in parallel programming. Consider uncapacitated facility location for which state-of-the-art algorithms are based on multistart tabu search [11] and hybrid heuristics [1]. The fragment

```

1.  Solution sol[1..nbStarts];
2.  parall(i in 1..nbStarts) {
3.      WarehouseLocation location();
4.      location.state();
5.      sol[i] = location.search();
6.  }
7.  selectMax(i in 1..nbStarts)(sol[i].getValue())
8.      cout << "Solution at Cost: " << sol[i].getValue() << endl;

```

depicts the COMET implementation of a parallel multistart variable neighborhood search. Each execution of the body creates a facility location model (line 3), states its constraints (line 4), and searches for a high-quality solution from a randomly generated initial solution (line 5). The returned solution is stored in `sol[i]` (line 5 again). Once all executions are completed, the values of the best found solutions is displayed (lines 7–8). The parallel code enjoys some interesting properties. First, it provides a parallel counterpart to a natural sequential implementation: the `parall` instruction simply replaces the `forall` construct of COMET. Second, the modeling and search components of the application are independent from the parallel code: they have not been modified when moving from the sequential to the parallel implementation.

Some complex applications in scheduling and vehicle routing explore heterogeneous neighborhoods and may feature several distinct local search algorithms collaborating in finding a solution. For instance, the snippet

```
pardo exploreRNA() | exploreNB();
```

illustrates how the two neighborhoods of the jobshop algorithm of Dell’Amico and Trubian [5] can be explored in parallel. The exploration code, that is described in [10], uses a neighborhood selector that must now be a synchronized object. Once again, the effort of moving from a sequential to a parallel implementation is minimal.

Implementation The implementation of the `parall` construct uses a source-to-source transformation that creates a thread for each iteration and a barrier to join all the threads. For instance, the parallel COMET for facility location just described is transformed into the COMET code:

```

1.  Solution sol[1..nbStarts];
2.  Barrier joinPoint();
3.  forall(i in 1..nbStarts) {
4.      joinPoint.incr();
5.      thread {
6.          WarehouseLocation location();
7.          location.state();
8.          sol[i] = location.search();
9.          joinPoint.decr();
10.     }
11. }
12. joinPoint.wait();
13. selectMax(i in 1..nbStarts)(sol[i].getValue())
14.     cout << "Solution at Cost: " << sol[i].getValue() << endl;

```

During the transformation, the `parall` construct is replaced by a `forall` loop that creates a thread at each iteration (lines 5-10). As a consequence, $|Starts| + 1$ threads may be executing simultaneously: the master thread executing the `forall` instruction and the $|Starts|$ slave threads created during the loop. The master thread waits for the completion of all slaves after the loop (line 12) before proceeding to subsequent instructions. The synchronization is performed using a barrier declared in line 2. The monitor is incremented in line 4 to specify that a new thread is about to join the execution and is decremented in line 9, just before the thread completes its execution.

3.2 Thread Pools

The Abstraction 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. The concept of thread pool, i.e., a collection of cooperating threads, overcomes this limitation and allows COMET programs to map nicely onto the underlying architecture. The `parall` instruction can be parameterized by a thread pool which is then responsible for executing the loop iterations in parallel. The size of the thread pool can be determined (statically or dynamically) according to the number of available processors or any other appropriate criterion.

Figure 4 depicts the implementation of the multistart local search for facility location in terms of thread pools. The pool is declared in line 1, used in the `parall` instruction (line 3), and closed after the loop (line 8). 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.

Implementation The implementation of the `parall` instruction over parallel pools (and thus thread pools) is also based on a source-to-source transformation. However, it differs from the earlier implementation by not using threads.

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

```

1.  ThreadPool tp(4);
2.  Solution sol[1..nbStarts];
3.  parall<tp>(i in 1..nbStarts) {
4.    WarehouseLocation location();
5.    location.state();
6.    sol[i] = location.search();
7.  }
8.  tp.close();
9.  selectMax(i in 1..nbStarts)(sol[i].getValue())
10. cout << "Best Solution at Cost: " << sol[i].getValue() << endl;

```

Fig. 4. The Parall Instruction over Thread Pools for a Multistart Local Search.

```

1.  Solution sol[1..nbStarts];
2.  Barrier joinPoint();
3.  forall(i in 1..nbStarts) {
4.    joinPoint.incr();
5.    closure cl {
6.      WarehouseLocation location();
7.      location.state();
8.      sol[i] = location.search();
9.      joinPoint.decr();
10.   }
11.   tp.execute(cl);
12. }
13. joinPoint.wait();
14. selectMax(i in 1..nbStarts)(sol[i].getValue())
15. cout << "Solution at Cost: " << sol[i].getValue() << endl;

```

Fig. 5. The Implementation of the Parall Instruction over Thread Pools.

Instead it creates closures that are submitted to the pool for execution. Figure 5 depicts the result of the source-to-source transformation for the code presented in Figure 4. The main novelty is the creation, for each iteration of the parallel loop, of a closure (lines 5–10) which is submitted to the pool (line 11). The implementation of the thread pool is shown in Figure 6. Thread pools implement the interface `ParallelPool` and must support a method `execute` on closures. Its core is the constructor (lines 5–13). It first constructs a producer/consumer buffer of closures whose implementation is similar to the `COMET` class in Figure 2. It then creates `n` threads that consume and execute closures from the buffer. These closures are produced by the `parall` instruction for each iteration as shown in 5 and the `execute` method of the thread pool simply “produces” a closure for the buffer (line 14). When the thread pool is closed (line 15), all threads are terminated, since method `terminate` on the producer/consumer buffer wakes all threads waiting for a closure. It is worth emphasizing that the availability of closures as first-class objects in `COMET` [10] is critical in designing and implementing the abstractions.

```

1. class ThreadPool implements ParallelPool {
2.     ClosureBuffer buffer;
3.     bool closed;
4.     ThreadPool(int n) {
5.         closed = false;
6.         buffer = new ClosureBuffer(n);
7.         forall(i in 1..n)
8.             thread b
9.                 while (!closed) {
10.                    Closure v = buffer.consume();
11.                    if (v != null) call(v);
12.                }
13.     }
14.     void execute(Closure v) { buffer.produce(v); }
15.     void close() { closed = true; buffer.terminate(); }
16. }

```

Fig. 6. The Implementation of the Thread Pool.

3.3 Interruptions

The Abstraction Multistart local searches are also useful for constraint satisfaction where the goal is to find a feasible solution. Indeed, many local search algorithms use restarts to avoid being trapped in long unsuccessful runs. Contrary to optimization problems where all searches are potentially pertinent, a multistart local search for constraint satisfaction should terminate as soon as a feasible solution is found. Consider the code

```

1. Boolean found(false);
2. parall(i in Starts) {
3.     ProgressiveParty pp();
4.     pp.state();
5.     found := pp.search();
6. } until found;

```

which uses a multistart local search to solve the progressive party (see [7, 13] for descriptions of the tabu-search algorithm). The code declares a Boolean variable `found` to denote whether a solution has been found (line 1). This variable is then used in line 6 to specify that the local searches must terminate as soon as `found` becomes true. Each loop iteration creates an instance of the model, states the constraints, and searches for a feasible solution.

This COMET code features a complete separation between the model and the parallel instructions, although the operational behavior of the model is fundamentally affected. There is no need in COMET to enhance the model to implement interruptions, showing the compositionality and the modularity promoted by the parallel abstractions. Observe also that interruptions are a fundamental abstractions for many applications: they may be used to interrupt local searches after some internal or external event as is frequently the case in online optimization or when cooperating local searches produces new improving solutions.

<pre> 1. Boolean found(false); 2. Barrier joinPoint(); 3. forall(i in Starts:!found){ 4. joinPoint.incr(); 5. thread { 6. break { 7. ProgressiveParty pp(); 8. pp.state(); 9. if (pp.search()) 10. found := true; 10. } when found; 11. joinPoint.decr(); 12. } 13.} </pre>	\implies	<pre> 1. Boolean found(false); 2. Barrier joinPoint(); 3. forall(i in Starts:!found){ 4. joinPoint.incr(); 5. thread { 6a. try { 6b. whenever found@changes(){ 6c. throw new Stop(); 6d. } in { 7. ProgressiveParty pp(); 8. pp.state(); 9. found := found pp.search(); 10a. } 10b. } catch (Stop e) {} 11. joinPoint.decr(); 12. } 13.} </pre>
---	------------	---

Fig. 7. The Implementation of Early Termination for a Multistart Local Search.

Implementation The implementation of interruptions relies on events [10] and is once again based on a source-to-source transformation. The rewriting is best understood as a two-step process. The first step rewrites the `parall` instruction into a `forall` loop featuring a `break/when` construct to terminate the threads, while the second step rewrites this construct into an event and an exception. For instance, Figure 7 depicts the rewriting for the multistart local search for the progressive party problem described earlier: the left and right columns show the result of the first and second phases respectively. The main novelty in the left column is the `break/when` instruction in lines 6–10: the new construct encapsulates the search and terminates when `found` becomes true. The right column shows how the `break/when` instruction is rewritten in terms of an event (lines 6b–6d) which monitors whether the Boolean variable is updated (line 6b) and interrupts the search by throwing an exception from inside the event-handler (line 6c) when the Boolean becomes true.

It is important to highlight the operational behavior induced by this implementation. Whenever the Boolean becomes true, an event is published to all threads, each of which now interrupts its execution by throwing an exception in their event-handlers. The ability of a thread to post events caught in other threads is fundamental in implementing interruptions and providing the desired compositionality and separation of concerns of the abstractions. Observe also that the novel `whenever/in` construct that specifies the scope in which the event is active. This generalization ensures that notifications only reach relevant events, which is essentially when the parallel code is embedded in outermost loops (as in the Golomb ruler discussed later).

3.4 Parallel Repeat Loop

Once thread pools are available, some additional parallel abstractions become natural. Reconsider the progressive party problem where the number of restarts

is not chosen a priori. The multistart local search procedure can be implemented by the parallel counterpart to a `repeat` loop:

```

1. Boolean found(false);
2. ThreadPool tp(4);
3. parrepeat<tp> {
4.   ProgressiveParty pp();
5.   pp.state();
6.   found := found || pp.search();
7. } until found;
8. tp.close();

```

The `parrepeat` instruction uses the thread pool to execute its body in parallel until a feasible solution is found. Each time a thread completes a local search, it restarts a new search. Once a solution is obtained, all the threads in the pool terminate, at which stage the parallel repeat also terminates. It is important to emphasize the novelty of the `parrepeat` construct: the automatic vectorization performed by automatic vectorization in compilers (e.g., the recent GCC-4 compiler) only applies to `for` loops. It is the availability of interruptions (and thus of events and closures) and the application domain that allow the parallelizations of loops with no apriori bounds on the number of iterations.

Implementation The implementation of the `parrepeat` construct is based on a source-to-source transformation and a bounded semaphore (i.e., a synchronization object that blocks when it is about to become negative or to exceed its upper bound). For instance, Figure 8 shows the rewriting for the progressive party problem. The implementation creates a bounded semaphore (line 3) ensuring that at most 4 threads are searching at any one time. The semaphore is incremented before creating a closure (line 5) and decremented upon completion of a search (line 12). The master thread (executing the loop) blocks as soon as the semaphore reaches its upper bound before moving to the next iteration to produce a new closure. Whenever a solution is found by some thread, all threads in the pool are interrupted and the master thread exits the loop.

3.5 Model Pools

Thread pools remove the one-to-one mapping between a thread and a loop iteration, avoid the overhead of running many threads, and allow for novel parallel abstractions. However, they do not avoid the overhead induced by creating many instances of the same model and stating the model constraints. For instance, the multistart local searches presented so far always construct a new model for each iteration of the parallel loops, an overhead which may be avoided by sequential restarts. This overhead, if non-negligible, can be remedied by model pools which are now responsible for executing the loop iterations. Consider again the multistart local search for facility location. With model pools, it can be implemented as follows:

```

1. Boolean found(false);
2. ThreadPool tp(4);
3. BoundedSemaphore sem(4);
4. do {
5.     sem.incr();
6.     closure cl {
7.         break {
8.             ProgressiveParty pp();
9.             pp.state();
10.            found := found || pp.search();
11.        } when found;
12.        sem.decr();
13.    };
14.    tp.execute(cl);
15. } while (!found);
16. tp.close();

```

Fig. 8. The Implementation of Parallel Repeat Loops.

```

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

```

Fig. 9. The Interface of Parallel Models.

```

1. ParallelModel model[1..4] = new WarehouseLocation();
2. ModelPool mp(model);
3. Solution s[Starts];
4. parall<mp>(i in Restarts)
5.     s[i] = mp.search();
6. mp.close();

```

The COMET code creates an array of models (line 1), which are then used to define the model pool (line 2). The `parall` instruction is now parameterized by the model pool to implement the multistart local search. A particularly interesting feature of this COMET program is the way solutions are produced: the parallel loop requests the model pool to search for solutions, not individual models (line 5). This liberates programmers from keeping track of which models are now available and which are busy. As a consequence, the parallel abstractions promote elegant compositionality and separation of concerns, letting users focus on their (sequential) models, automating the synchronization aspects, and reusing the same models in sequential and parallel settings. Note also that the models in the pool may implement different search strategies as sometimes proposed in cooperating local search: it suffices to fill the `model` array with different models, showing the simplicity of parallelizing heterogeneous models.

To be included in a model pool, a model must implement the interface depicted in Figure 9. The interface contains methods for stating the model con-

straint, searching for a solution, searching for a solution from a given starting point, and searching for solutions using several existing solutions, as is typically the case in hybrid evolutionary algorithms and scatter search.

The problem of finding optimal, or near-optimal, Golomb rulers using hybrid evolutionary algorithms is an interesting illustration of model pools. The hybrid algorithm is organized as a series of searches, each of which finding a shorter ruler. More precisely, the basic step of the algorithm is to find a Golomb ruler whose length is smaller than l . This feasibility search is performed by an hybrid evolutionary algorithm applying a tabu search on solutions obtained by crossing existing solutions in the population. The initial population is simply a set of (infeasible) rulers of size smaller than l and the crossover operator combines two rulers by taking the first half of the first ruler and the second half of the second one. At each iteration, the hybrid algorithm has a population of n rulers and generates a new population of the same size by repeatedly choosing two rulers randomly, crossing them, and applying the tabu search. Each iteration is inherently parallel and can be implemented by the COMET code

```

1.  Solution o[k in Pop] = pop[k];
2.  parall<mp>(k in Pop) {
3.      select(i in Pop, j in Pop: i != j) {
4.          pop[k] = mp.search(o[i],o[j]);
5.          found := (pop[k].getValue() == 0);
6.      }
7.  } until found;

```

where `mp` is a model pool. The code uses a parallel loop, early termination, and model pools. The computation starts by storing the current population in `o`, making it the old population (line 1). The search for a feasible ruler uses a `parall` instruction over a model pool, terminating when `found` becomes true (lines 2-7). The model pool contains a number of Golomb ruler models that can be used simultaneously in the parallel loop. Each iteration selects two solutions in the old population, crosses them, searches for a solution using the model pool, and stores the resulting solution in `pop[k]`, i.e., the k^{th} element of the new population (line 4). If the ruler is feasible (i.e., its objective, which denotes the number of constraint violations, is zero), all threads are interrupted and the loop execution completes after the assignment in line 5. This code is included in an outermost loop, which will start a new search for a shorter ruler.

The resulting COMET code exhibits several desirable properties. First, the parallel code is almost identical to the sequential code. The only changes are, in fact, the parallel loop which replaces its sequential counterpart and the model pool which generalizes the single model of the sequential implementation. Second, the compositionality and separation of concerns promoted by the parallel abstractions ensure that the parallel implementation induces no changes to the basic model, i.e., the constraints, the tabu search, and the crossover operators. Only the generation of the new population is affected and it is precisely the code that is being parallelized. Observe that the COMET program has no explicit synchronization, thread management, or termination code.

```

1. class ModelPool implements ParallelPool {
2.     bool closed;
3.     ModelBuffer mbuf;
4.     ClosureBuffer cbuf;
5.     ModelPool(ParallelModel[] models){
6.         closed = false;
7.         mbuf = new ModelBuffer(models.size());
8.         cbuf = new ClosureBuffer(models.size());
9.         forall(i in 1..models.size()) {
10.            models[i].state();
11.            mbuf.produce(models[i]);
12.        }
13.        forall(i in 1..models.size())
14.            thread b
15.                while (!closed)
16.                    call(cbuf.consume());
17.    }
18.    void submit(Closure body) { bbuf.produce(body);}
19.    void close() { closed = false; mbuf.terminate(); cbuf.terminate(); }
20.    Solution search() {
21.        ParallelModel m = mbuf.consume();
22.        Solution s = m.search();
23.        mbuf.produce(m);
24.        return s;
25.    }
26.}

```

Fig. 10. The Implementation of Model Pools

Implementation Model pools use the same source-to-source transformation as thread pools, since both are implementations of parallel pools. The main difference is in the implementation of the pool itself which is depicted in Figure 10. In addition to the closure buffer, model pools also use a buffer to keep track of models available for execution. Initially, the pool states the constraints in all models and “produces” the models (lines 9–12). When a search instruction is executed (lines 20–24), the pool first “consumes” a model (line 21), applies the search on the so-obtained model (line 22), and “produces” the model back into the buffer (line 23). Note that the implementation decouples the threads and models: coupling the models and the threads is not correct when the body of the parallel loops also uses parallel instructions.

4 Experimental Results

Table 1 presents preliminary experimental results on a Apple G5 with two processors. The table reports the solution quality ($\min(S)$, $\max(S)$, $\text{avg}(S)$), the average running time in seconds, and how much of the potential speedup (in percentage) is achieved by the parallel implementation. Note that the parallel implementation is unlikely to approach 100% because the operating systems is also running on one of the processors. The tested program includes a multi-

Problem	$min(S)$	$avg(S)$	$max(S)$	$avg(T)$	$\%(S)$
Facility Location (//)	54526.00	54545.20	54557.00	63.97	
Facility Location (seq)	54526.00	54545.20	54557.00	49.92	44%
Coloring (5) (seq)	66.00	66.38	67.00	240.33	
Coloring (5) (//)	65.00	66.32	67.00	130.16	92%
Coloring (1c) (seq)	64.00	64.51	65.00	368.38	
Coloring (1c) (//)	64.00	64.34	65.00	209.16	86%
Golomb-11 (seq)	72.00	72.00	72.00	57.08	
Golomb-11 (//)	72.00	72.00	72.00	36.90	71%

Table 1. Experimental Results on a Dual Apple G5.

start variable-neighborhood search for uncapacitated facility location (probably the most effective algorithm for this problem), the tabu-search algorithm for graph-coloring from [6] running for a large number of iterations, and an hybrid evolutionary algorithm for finding Golomb rulers. The facility location lines covers 30 benchmarks from the class FPP17, which requires a multistart strategy to obtain high-quality results. Each benchmark is run 100 times, accounting for a total of 6,000 runs. The coloring algorithm was evaluated on the benchmarks R250.5.Co1 (250 vertices and 50% density) whose best-known solution is 65 and R250.1c.Co1 (250 vertices and 90% density) whose best-known solution is 64, accounting for 200 runs. The Golomb ruler program was run until the optimal solution (of length 72) was found, also accounting for 200 runs. The experimental results confirm the practicality of the extensions. On coloring, the parallel implementation produces 92% and 86% of the maximum speedups. On facility location, 44% of the potential speedup is realized. The smaller percentage is due partly due to differences in runtimes between several runs (decreasing in fact the maximum possible speedup) and the memory allocation necessary to maintain the best set of facilities to swap. Since both threads use the same memory pool and garbage collection, there is some contention for memory. The Golomb ruler produces 71% of the maximum speedup, which is quite impressive since there are considerable differences between runs and the execution also terminates as soon as the optimal solution is found. Overall these results are promising. Moreover, potential limitations have been isolated and can be remedied by using a more advanced memory system.

5 Related Work

The parallel abstractions of COMET share the same motivation as `openMP`, a pre-processor for parallel loops in C and Fortran [4, 2]. Both systems aim at making parallel computing widely accessible by reducing the distance between sequential and parallel code. `openMP` supports instructions that resemble the `parall` and `pardo` constructs of COMET. However, the parallel abstractions of COMET are simpler and richer, primarily because of its advanced control abstractions: first-order closures and events. Unlike `openMP`, there is no need in COMET to specify the role of variables in parallel loop. The role of variables in COMET is uniform with respect to events [10], nondeterminism [12], and parallelism. Unlike

openMP, COMET provides interruptions and parallel `repeat/while` loops which impose no a-priori bounds on the number of iterations. The concepts of parallel pools naturally support homogeneous and heterogeneous cooperating local searches and reduces the overhead that plagues simpler parallel loops. Finally, the parallel abstractions of COMET are modular and compositional, separates the optimization models from the parallel code, and imposes minimal requirements on the underlying implementation which remains simple and easily maintainable.

6 Conclusion

This paper proposed parallel programming abstractions to exploit the availability of commodity multiprocessors in COMET. The abstractions include parallel loops, interruptions, as well as thread and model pools. They address the need of making parallel constraint-based local search and hybrid algorithms as close as possible to their sequential counterparts. In particular, they are compositional, cleanly separates the parallel codes from the optimization models, and leverages the advanced control abstractions of COMET: events and first-order closures. Preliminary results on a variety of applications indicate that they can be implemented with minimal overhead. Future research will be devoted to a distributed implementation of COMET on networks of commodity computers.

References

1. R. Aiex, S. Binato, and M. Resende. Parallel GRASP with Path-Relinking for Jobshop Scheduling. *Parallel Computing*, 29(4):393–430, 2003.
2. R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000. ISBN:1558606718.
3. W.F. Clocksin and H. Alshawi. A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors. *New Generation Computing*, 5:361–376, 1988.
4. L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5:46–55, 1998.
5. M. Dell’Amico and M. Trubian. Applying Tabu Search to the Job-Shop Scheduling Problem. *Annals of Operations Research*, 41:231–252, 1993.
6. 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.
7. L. Michel and P. Van Hentenryck. A Constraint-Based Architecture for Local Search. In *OOPSLA02*, pages 101–110, Seattle, November 2002.
8. L. Michel and P. Van Hentenryck. A Decomposition-Based Implementation of Search Strategies. *ACM Transactions on Computational Logic*, 5(2), 2004.
9. L. Perron. Search Procedures and Parallelism in Constraint Programming. In *CP’99*, pages 346–360, Alexandria, Virginia, October 1999.
10. P. Van Hentenryck and L. Michel. Control Abstractions for Local Search. In *CP’03*, pages 65–80, Cork, Ireland, 2003.
11. P. Van Hentenryck and L. Michel. *Constraint-Based Local Search*. The MIT Press, Cambridge, Mass., 2005.
12. P. Van Hentenryck and L. Michel. Nondeterministic Control for Hybrid Search. In *CP-AI-OR’05*, Prague, May 2005.
13. P. Van Hentenryck, L. Michel, and L. Liu. Constraint-Based Combinators for Local Search. In *CP’04*, pages 47–61, Toronto, Canada, October 2004.