

Experimenting with X10 for Parallel Constraint-Based Local Search

Danny Munera¹, Daniel Diaz¹, and Salvador Abreu²

¹ University of Paris 1-Sorbonne, France

{Danny.Munera,Daniel.Diaz}@univ-paris1.fr

² Universidade de Évora and CENTRIA, Portugal
spa@di.uevora.pt

Abstract. In this study, we have investigated the adequacy of the PGAS parallel language X10 to implement a Constraint-Based Local Search solver. We decided to code in this language to benefit from the ease of use and architectural independence from parallel resources which it offers. We present the implementation strategy, in search of different sources of parallelism in the context of an implementation of the Adaptive Search algorithm. We extensively discuss the algorithm and its implementation. The performance evaluation on a representative set of benchmarks shows close to linear speed-ups, in all the problems treated.

1 Introduction

Constraint Programming has been successfully used to model and solve many real-life problems in diverse areas such as planning, resource allocation, scheduling and product line modeling [18, 19]. Classically constraint satisfaction problems (CSPs) may be solved exhaustively by complete methods which are able to find all solutions, and therefore determine whether any solutions exist. However efficient these solvers may be, a significant class of problems remains out of reach because of exponential growth of search space, which must be exhaustively explored. Another approach to solving CSPs entails giving up completeness and resorting to (meta-) heuristics which will guide the process of searching for solutions to the problem. Solvers in this class make choices which limit the part of the search space which actually gets visited, enough so to make problems tractable. For instance a complete solver for the *magic squares* benchmark will fail for problems larger than 15×15 whereas a local search method will easily solve a 100×100 problem instance within the lower resource bounds. On the other hand, a local search procedure may not be able to find a solution, even when one exists.

However, it is unquestionable that the more computational resources are available, the more complex the problems that may be solved. We would therefore like to be able to tap into the forms of augmented computational power which are actually available, as conveniently as feasible. This requires taming various forms of explicitly parallel architectures.

Present-day parallel computational resources include increasingly multi-core processors, General Purpose Graphic Processing Units (GPGPUs), computer clusters and grid computing platforms. Each of these forms requires a different programming model and the use of specific software tools, the combination of which makes software development even more difficult.

The foremost software platforms used for parallel programming include POSIX Threads [1] and OpenMP [17] for shared-memory multiprocessors and multicore CPUs, MPI [22] for distributed-memory clusters or CUDA [16] and OpenCL [13] for massively parallel architectures such as GPGPUs. This diversity is a challenge from the programming language design standpoint, and a few proposals have emerged that try to simultaneously address the multiplicity of parallel computational architectures.

Several modern language designs are built around the Partitioned Global Address Space (PGAS) memory model, as is the case with X10 [21], Unified Parallel C [9] or Chapel [6]. Many of these languages propose abstractions which capture the several forms in which multiprocessors can be organized. Other, less radical, approaches consist in supplying a library of inter-process communication which relies on and uses a PGAS model.

In our quest to find a scalable and architecture-independent implementation platform for our exploration of high-performance parallel constraint-based local search methods, we decided to experiment with one of the most promising new-generation languages, X10 [21].

The remainder of this article is organized as follows: Section 2 present some necessary background in Constraint-Based Local Search. Section 3 discusses the PGAS Model and presents X10 language. The sources of parallelism in the Adaptive Search algorithm and its X10 implementation are the object of Section 4. Section 5 describes the benchmarks used in our experiments. The results and a discussion of the performance evaluation may be found in Section 6. A short conclusion ends the paper.

2 Constraint-Based Local Search

It is possible to classify the method for constraint solving in two groups: *complete* or *incomplete* methods. Complete methods always find a solution of the problem, if any exists. There are two main groups of algorithms: search and inference. Firstly, a representative complete search method is depth-first backtracking, which incrementally builds a solution exploring as far as possible along each branch of the search space, such that no constraint is violated. If from some point no value allows to extend the current partial assignment, it backtracks and a previous assignment is reconsidered. Secondly, the inference methods use constraints to reduce the number of legal values for a variable, and propagate this new domain to reduce other domain variables, finding a solution.

Incomplete methods, like Local Search, try to found a solution using limited resources, however this type of methods does not guarantee neither to find one solution nor to detect an inconsistency problem (no possible solution). Local

search is suitable for optimization problems with a cost function which makes possible to evaluate the quality of a given configuration (assignment of variables to current values). It also needs a transition function which defines, for each configuration, a set of neighbors. The simplest Local Search algorithm start from a random configuration, explores the neighborhood and then selects a promising neighbor and moves there. This is a iteratively process that continues until a solution is found.

In this study, a Local Search method, the Adaptive Search algorithm is selected. The Adaptive Search [3, 4] is a generic, domain-independent constraint-based local search method. This meta-heuristic takes advantage of the CSP formulation and makes it possible to structure the problem in terms of variables and constraints and to analyze the current assignment of variables more precisely than an optimization of a global cost function e.g. the number of constraints that are not satisfied. Adaptive Search also includes an adaptive memory inspired in Tabu Search [11] in which each variable leading to a local minimum is marked and cannot be chosen for the next few iterations. A local minimum is a configuration for which none of the neighbors improve the current configuration. The input of the Adaptive Search algorithm is a CSP, for each constraint an error function is defined. This function is a heuristic value to represent the degree of satisfaction of a constraint and gives an indication on how much the constraint is violated.

Adaptive Search is based on iterative repair from the variables and constraint error information, trying to reduce the error in the worse variable. The basic idea is to calculate the error function for each constraint, and then combine for each variable the errors of all constraints in which it appears, thus projecting constraint errors on involved variables. Then, the algorithm chooses the variable with the maximum error as a “culprit” and selects it to modify later its value.

The purpose is to select the best neighbor move for the culprit variable, this is done by considering all possible changes in the value of this variable (neighbors) and selecting the lower value of the overall cost function. Finally, the algorithm also includes partial resets in order to escape stagnation around local minima; and it is possible to restart from scratch when the number of iterations becomes too large. Algorithm 1 presents a particular implementation of the Adaptive Search algorithm dedicated to permutation problems. In this case all N variables have the same initial domain of size N and are subject to an implicit *all-different* constraint.

3 PGAS model and X10

The current arrangement of tools to exploit parallelism in machines are strongly linked to the platform used. As it was said above, two broad programming models stand out in this matter: *distributed* and *shared memory* models. For large distributed memory systems, like clusters and grid computing, Message Passing Interface (MPI) [22] is a de-facto programming standard. The key idea in MPI is to decompose the computation over a collection of processes with private mem-

ory space. This processes can communicate with each other through message passing, generally over a communication network.

With the recent growth of many-core architectures, the shared memory approach have increased its popularity. This model decomposes the computation in multiple threads of execution sharing a common address space, communicating with each other by reading and writing shared variables. Actually, this is the model used by traditional programming tools like Fortran or C through libraries like *pthread*s [1] or OpenMP [17].

The PGAS model tries to combine the advantages of the two approaches mentioned so far. This model extends shared memory to a distributed memory setting. The execution model allows having multiple processes (like MPI), multiple threads in a process (like OpenMP), or a combination (see Figure 1). Ideally, the user would be allowed to decide how tasks get mapped to physical resources. X10 [21], Unified Parallel C [24] and Chapel [6] are examples of PGAS-enabled languages, but there exist also PGAS-based IPC libraries such as GPI [15], for use in traditional programming languages. For the experiments described herein, we used the X10 language.

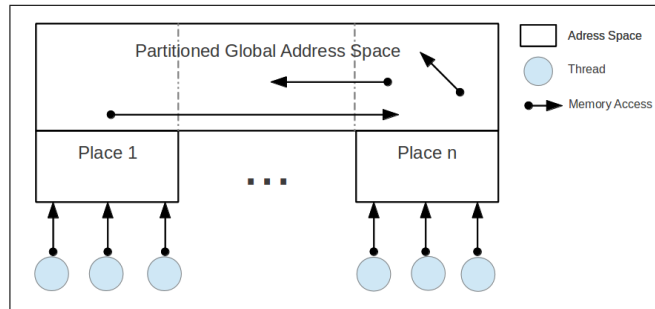


Fig. 1. PGAS Model

3.1 X10 in a Nutshell

X10 [21] is a general-purpose language developed by IBM, which provides a PGAS variation: Asynchronous PGAS (APGAS). APGAS extends the PGAS model making it flexible, even in non-HPC platforms [20]. Through this model X10 can support different levels of concurrency with simple language constructs.

There are two main abstractions in X10 model: *places* and *activities*. A *place* is the abstraction of a virtual shared-memory process, it has a coherent portion of the address space together with threads (*activities*) that operate on that memory. The X10 construct for creating a place in X10 is *at*, and is commonly used to create a place for each processing unit in the platform. An *activity* is the mechanism to abstract the single threads that perform computation within a place. Multiple activities may be active simultaneously in a place.

With these two components of X10 implement the main concepts of the PGAS model. However, the language includes other interesting tools with the goal of improving the abstraction level of the language. Synchronization is supported thanks to various operations such as *finish*, *atomic* and *clock*. The operation *finish* is used to wait the termination of a set of activities, it behaves like a traditional barrier. The constructs *atomic* ensures an exclusive access to a critical portion of code. Finally, the construct *clock* is the standard way to ensure the synchronization between activities or places. X10 supports the distributed array construct, which makes it possible to divide an array into sub-arrays which are mapped to available places. Doing this ensures a local access from each place to the related assigned sub-array. A detailed examination of X10, including tutorial, language specification and examples can be consulted at <http://x10-lang.org>.

4 X10 Adaptive Search Parallel Implementation

In order to get advantage of the parallelism it is necessary to identify the sources of parallelism of the algorithm. In [5], the authors survey the state-of-the-art of the main parallel meta-heuristic strategies and discuss general design and implementation principles. This study raises a number of important issues in the taxonomy of parallel sources that lies in meta-heuristics. They classify the decomposition of activities for parallel work in two main groups: *functional parallelism* and *data parallelism*.

On the one hand, in *functional parallelism* generally different tasks work in parallel on the same data, allocated in different compute instances. On the other hand, *data parallelism* refers to the methods in which the problem domain or the associated search space is decomposed. A particular solution methodology is used to address the problem on each of the resulting components of the search space. Based on this study, we explored the parallelism in the Adaptive Search method in both *functional parallelism* and *data parallelism* and this article mostly reports on the latter, because this is where we expect the most significant gains to show.

4.1 Adaptive Search X10 sequential implementation

The first stage of the X10 implementation was to develop a sequential algorithm, which is described as algorithm 1.

Figure 2 shows the class diagram of the basic X10 project. The class *ASPermutSolver* contains the Adaptive Search permutation specialized method implementation. This class inherits the basic functionality from a general implementation of the Adaptive Search solver (in class *AdaptiveSearchSolver*), which in turn inherits a very simple Local Search method implementation from the class *LocalSearchSolver*. This class is then specialized for different parallel approaches, which we experimented with.¹

¹ We experimented with two versions of Functional Parallism (FP1 and FP2) and a Random Walk version (RW.)

Algorithm 1 Adaptive Search Base Algorithm

Input: problem given in CSP format:

- set of variables $V = \{X_1, X_2 \dots\}$ with their domains
- set of constraints C_j with error functions
- function to project constraint errors on vars (positive) cost function to minimize
- T : Tabu tenure (number of iterations a variable is frozen on local minima)
- RL : number of frozen variables triggering a reset
- MI : maximal number of iterations before restart
- MR : maximal number of restarts

Output: a solution if the CSP is satisfied or a quasi-solution of minimal cost otherwise.

```
1: Restart  $\leftarrow$  0
2: repeat
3:   Restart  $\leftarrow$  Restart + 1
4:   Iteration  $\leftarrow$  0
5:   Compute a random assignment A of variables in V
6:   Opt_Sol  $\leftarrow$  A
7:   Opt_Cost  $\leftarrow$  cost(A)
8:   repeat
9:     Iteration  $\leftarrow$  Iteration + 1
10:    Compute errors constraints in C and project on relevant variables
11:    Select variable X with highest error: MaxV
12:                                      $\triangleright$  not marked Tabu
13:    Select the move with best cost from X: MinConflictV
14:    if no improvement move exists then
15:      mark X as Tabu for T iterations
16:      if number of variables marked Tabu  $\geq RL$  then
17:        randomly reset some variables in V
18:                                      $\triangleright$  and unmark those Tabu
19:      end if
20:    else
21:      swap(MaxV, MinConflictV),
22:                                      $\triangleright$  modifying the configuration A
23:      if cost(A) < Opt_Cost then
24:        Opt_Sol  $\leftarrow$  A
25:        Opt_Cost  $\leftarrow$  costs(A)
26:      end if
27:    end if
28:  until Opt_Cost = 0 (solution found) or Iteration  $\geq MI$ 
29: until Opt_Cost = 0 (solution found) or Restart  $\geq MR$ 
30: output(Opt_Sol, Opt_Cost)
```

Moreover, a simple CSP model is described in the class *CSPModel*, and specialized implementations of each CSP benchmark problem are contained in the classes *PartitModel*, *MagicSquareModel*, *AllIntervallModel* and *CostasModel*,

which have all data structures and methods to implement the error function of each problem.

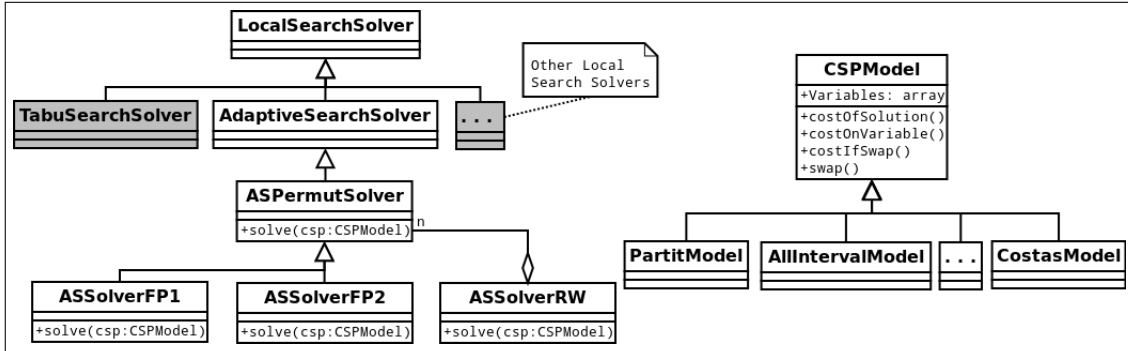


Fig. 2. X10 Class Diagram basic project

Listing 1.1 shows a simplified skeleton code of our X10 sequential implementation, based on Algorithm 1. The core of the Adaptive Search algorithm is implemented in the method `solve`. The `solver` method receives a `CSPModel` instance as a parameter. In line 8, the CSP variables of the model are initialized with a random permutation, in the next line the total cost of the current configuration is computed. The sentence `while` on line 10 begins the main loop of the algorithm. The `selectVarHighCost` function (Line 12) selects the variable with the maximum error and saves the result in `maxI` variable. The `selectVarMinConflict` function (Line 13) selects the best neighbor move from the high cost variable `maxI`, and saves the result index in `minJ` variable. Finally, if no local minimum is detected, the algorithm swaps the variables `maxI` and `minJ` (permutation problem) and recompute the total cost of the current configuration (Line 16). The solver function ends if the `totalCost` variable reaches the 0 value or if the algorithm reaches the maximum number of iterations.

Listing 1.1. Simplified AS X10 Sequential Implementation

```

1 class ASPermutSolver {
2   var totalCost: Int;
3   var maxI: Int;
4   var minJ: Int;
5
6   public def solve (csp: CSPModel): Int {
7     // (... local variables ...)
8     csp.initialize ();
9     totalCost = csp.costOfSolution ();
10    while (totalCost != 0) {
11      // (... restart code ...)
12      maxI = selectVarHighCost (csp);
13      minJ = selectVarMinConflict (csp);
  
```

```

14         // (... local min tabu list , reset code...)
15         csp.swapVariables (maxI, minJ);
16         totalCost = csp.costOfSolution ();
17     }
18     return totalCost;
19 }
20 }

```

4.2 Adaptive Search X10 Parallel Implementation

We first experimented with functional parallelism which consisted in executing the inner loop in parallel, but the results were not expressive, mostly because of thread handling overhead. We then implemented a data parallel variant, which turns out natural because the sequential Adaptive Search algorithm can be used as an isolated search instance. Furthermore, the search space is divided using different random start points (i.e configurations). This strategy is called Random Walks (RW) or Multi Search (MPSS, Multiple initial Points, Same search Strategies) [5] and has proven to be very efficient [7, 14]. The main point of our study is to explore the adequacy of a programming language based on a PGAS model: we will discuss the strengths and weaknesses of this language when applied to Independent Random Walks, without any further tuning of the parallel execution.

The implementation strategy is very different from the functional parallelism: the key of this algorithm is to have several independent and isolated instances of the Adaptive Search Solver applied to the problem model. Then it is necessary to initialize the problem variables with a random assignment of values for the variables and to distribute it to the available processing resources in the computer platform. Finally, when one instance reaches a solution, a termination detection communication strategy is used to finalize the remaining running instances. This simple parallel version has no inter-process communication, making it *Embarrassingly* or *Pleasantly Parallel*. The skeleton code of the algorithm is shown in the Listing 1.2.

Listing 1.2. Adaptive Search *data parallel* X10 implementation

```

1 public class ASSolverRW{
2     val solDist : DistArray[ASPermutSolver];
3     val cspDist : DistArray[CSPModel];
4     def this( ){
5         solDist=DistArray.make[ASPermutSolver](Dist.makeUnique());
6         cspDist=DistArray.make[CSPModel](Dist.makeUnique());
7     }
8     public def solve(){
9         val random = new Random();
10        finish for(p in Place.places()){
11            val seed = random.nextLong();
12            at(p) async {
13                cspDist( here.id ) = new CSPModel(seed);

```



```

14         solDist( here.id ) = new ASPermutSolver( seed );
15         cost = solDist( here.id ).solve( cspDist( here.id ) );
16         if ( cost==0 ){
17             for ( k in Place.places() )
18                 if ( here.id != k.id )
19                     at(k) async{
20                         solDist( here.id ).kill = true;
21                     }
22             }
23         }
24     }
25     return cost;
26 }

```

For this implementation the *ASSolverRW* class was created. This class has two global distributed arrays *solDist* and *cspDist* (lines 2 and 3). As explained in Section 3.1, the *DistArray* class creates an array which is spread across multiple X10 places. In this case, an instance of *ASPermutSolver* and *CSPModel* are spread over all the available places in the program. On line 10 a finish operation is executed over a for loop that goes through all the places in the program (*Place.places()*). Then, an activity is created in each place with the sentence *at(p) async* on line 12. Into the *async* block, a new instance of the solver (*new ASPermutSolver(seed)*) and the problem (*new CSPModel(seed)*) are created (lines 13 and 14) and a random seed is passed. In line 15, the solving process is executed and the returned cost is assigned to the *cost* variable. If this cost is equal to 0, the solver in a place has reached a valid solution, it is then necessary to send a termination signal to the remaining places (lines 16- 22). For this, every place (i.e. every solver), checks the value of a *kill* variable at each iteration. When it becomes equal to **true** the main loop of the solver is broken and the activity is finished. To set a *kill* remote variable from any X10 place it was necessary to create a new activity into each remaining place (sentence *at(k) async* in line 19) and into the *async* block to change the value of the *kill* variable. Line 18 with the sentence *if (here.id != k.id)* filters all the places that not are the winner place (*here*). Finally, the function returns the solution of the fastest place in line 25.

5 Benchmark description

In this study we used a set of benchmarks composed by four classical problems in constraint programming: the magic square problem, the number partitioning problem and the all-interval problem, all three taken from the CSPLib [10]; also we include the Costas Array Problem (CAP) introduced in [12], which is a very challenging real problem.

Magic Square Problem (MSP) The Magic Square Problem (**prob019** in CSPLib) consists of placing on a $N \times N$ square all the numbers in $\{1, 2, \dots, N^2\}$

such as the sum of the numbers in all rows, columns and the two diagonal are the same. It can therefore be modeled in CSP by considering N^2 variables with initial domains $\{1, 2, \dots, N^2\}$ together with linear equation constraints and a global *all-different* constraint stating that all variables should have a different value. The constant value that should be the sum of all rows, columns and the two diagonals can be easily computed to be $N(N^2 + 1)/2$.

All-Interval Problem (AIP) The All-Interval Problem (prob007 in CSPLib) consists of composing a sequence of N notes such that all are different and tonal intervals between consecutive notes are also distinct. This problem is equivalent to finding a permutation of the N first integers such that the absolute difference between two consecutive pairs of numbers are all different. This amounts to finding a permutation (X_1, \dots, X_N) of $(0, \dots, N-1)$ such that the list $(\text{abs}(X_1 - X_2), \text{abs}(X_2 - X_3), \dots, \text{abs}(X_{N-1} - X_N))$ is a permutation of $(1, \dots, N-1)$. A possible solution for $N = 8$ is $(3, 6, 0, 7, 2, 4, 5, 1)$ because all consecutive distances are different.

Number Partitioning Problem (NPP) The Number Partitioning Problem (prob049 in CSPLib) consists of finding a partition of numbers $\{1, \dots, N\}$ into two groups A and B of the same cardinality such that the sum of numbers in A is equal to the sum of numbers in B and the sum of squares of numbers in A is equal to the sum of squares of numbers in B. A solution for $N = 8$ is $A = \{1, 4, 6, 7\}$ and $B = \{2, 3, 5, 8\}$.

Costas Array Problem (CAP) The Costas Array Problem consists of filling an $N \times N$ grid with N marks such that there is exactly one mark per row and per column and the $N(N-1)/2$ vectors joining the marks are all different. It is convenient to see the Costas Array Problem as a permutation problem by considering an array of N variables (X_1, \dots, X_n) which forms a permutation of $\{1, 2, \dots, N\}$ subject to some *all-different* constraints (see [8] for a detailed modeling). This problem has many practical applications and currently it has a whole community active working around it (<http://www.costasarrays.org/>).

6 Performance Analysis

This section presents and discusses our experimental results of the X10 implementation of the Adaptive Search algorithm.

We present the experimental results of the Adaptive Search algorithm in its X10 data parallel implementation. We do not present results for functional parallelism because the tests we carried out show that the granularity of individual threads is too fine to yield any performance improvement.

The testing environment used in each running was a non-uniform memory access (NUMA) computer, with 2 Intel Xeon W5580 CPUs each one with 4

hyper-threaded cores running at 3.2GHz. This system has 12 GB of main memory.

At the software level, the X10 runtime system can be deployed in two different backends: Java backend and C++ backend; they differ in the native language used to implement the X10 program (Java or C++), also they present different trade-offs on different machines. Currently, the C++ backend seems relatively mature and faster, therefore, we have chosen it for this experimentation.

Regarding the stochastic nature of the Adaptive Search behavior, several executions of the same problem were done and the times averaged. We ran 100 samples for each experimental case in the benchmark.

In this presentation, all tables use the same format: the first column identifies the problem instance, the second column is the execution time of the problem in the sequential implementation, the next group of columns contains the corresponding speed-up obtained with a varying number of cores (activities or places), and the last column presents the execution time of the problem with the highest number of cores.

Magic Square Problem. Table 1 presents the data obtained solving several large instances of MSP. Both raw times (average of 100 runs) and relative speed-ups are reported. The results show quasi-linear speed-ups (which seem independent of the size of the problem).

Problem instance	time (s)	speed-up with k places				time (s)
	seq.	2	4	6	8	8 places
40	0.47	2.12	3.20	4.14	4.42	0.11
60	1.59	1.58	2.68	3.25	3.62	0.44
80	5.10	1.84	2.92	3.72	4.19	1.22
100	11.88	1.69	2.92	3.64	4.36	2.72

Table 1. Magic Square: data parallel (timings and speed-ups)

All-Interval Problem. Table 2 shows the average time of several instances of AIP together with the speed-ups acquired with different number of places. Here again the speed-ups are practically linear up to 5.38 with 8 places. Moreover, in this case the speed-up tends to increase with the size of the problem.

Number Partitioning Problem. Table 3 shows the results of NPP. From this data, it can be seen that the corresponding speed-up obtained is almost the ideal speed-up for each number of places used. Also, the speed-up increases linearly with the number of cores to reach a maximum of 7.64 with 8 places. Moreover, this speed-up seems to increase with the size of the problem.

Problem instance	time (s) seq.	speed-up with k places				time (s) 8 places
		2	4	6	8	
50	0.027	2.25	3.24	4.46	4.94	0.005
100	0.47	2.12	3.21	4.32	4.96	0.09
150	2.36	1.74	3.47	4.65	5.15	0.46
200	8.29	1.84	3.66	5.28	5.38	1.54

Table 2. All-interval: data parallel (timings and speed-ups)

Problem instance	time (s) seq.	speed-up with k places				time (s) 8 places
		2	4	6	8	
1400	1.07	1.50	2.60	4.49	5.00	0.21
1600	1.94	1.78	3.59	5.10	6.97	0.28
1800	2.30	1.62	3.48	3.98	5.49	0.42
2000	4.31	2.34	4.87	7.06	7.64	0.56

Table 3. Partition: data parallel (timings and speed-ups)

Costas Array Problem. Table 4 presents the average time for several instances of the Costas Array Problem together with the speed-up obtained when using different numbers of places. The data confirm the trends above observed. Note that the best speed-up (9.56) is super-linear and is obtained for the most difficult instance of CAP (size of 19): the execution time is drastically reduced from 103.98 seconds to only 10.87 seconds on 8 places.

Problem instance	time (s) seq.	speed-up with k places				time (s) 8 places
		2	4	6	8	
16	0.26	1.79	3.52	4.00	7.09	0.04
17	1.94	1.96	4.46	6.47	9.84	0.19
18	10.90	1.68	4.05	5.09	6.88	1.59
19	103.98	2.13	4.63	5.81	9.56	10.87

Table 4. Costas Array: data parallel (timings and speed-ups)

Figure 3 shows the speed-ups reached on the most difficult instance of each problem. It can be seen that the speed-up increases almost linearly with the number of places used in the X10 program.

The performance evaluation developed in this work shows that a parallel Local Search solver implemented in X10 has good performance² using a data parallel strategy. The study has identified constant behavior of the speed-up with relation to the size of the problem (for some problems the speed-up improves

² Within a bound of 4-5 w.r.t. our sequential C implementation.

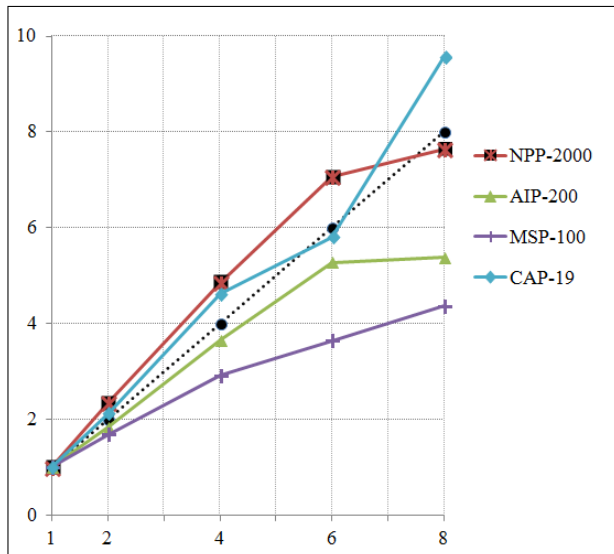


Fig. 3. Speed-ups for the most difficult instance of each problem

with the size of the problem). The resulting average runtime and the speed-ups obtained in the entire experimental test performed are as good as reported in the literature when using other IPC frameworks such as MPI [7, 8, 2] and seems to lie within the predictable bounds proposed by [23].

7 Conclusion and Future Work

We presented a parallel X10 implementation of an effective Local Search algorithm, Adaptive Search. We first experimented with functional parallelism, i.e. trying to divide the inner loop of the algorithm into various concurrent tasks. As expected, this yielded no speed-up, mainly because of the bookkeeping overhead (creation, scheduling and synchronization) that are too fine-grained.

We then proceeded with a data parallel implementation, in which the search space is decomposed in possible different random initial configurations of the problem and getting isolated solver instances to work on each point concurrently. We got a good level of performance for the X10 data-parallel implementation, reaching a maximum speed-up of 9.84 with 8 *places* for the Costas Array Problem. Linear (or close) speed-ups have been recorded in all problems we studied and they remain constant (or increasing) wrt the size of the problem.

X10 has proved a suitable platform to exploit parallelism in different ways for constraint-based local search solvers, ranging from single shared memory inter-process parallelism to more external distributed memory programming model. Additionally, the use of the X10 implicit communication mechanisms shows that X10 enables one to abstract the complexity of the parallelism with a very simple

model, e.g. the distributed arrays and the termination detection system in our data parallel implementation.

Future work will focus on the implementation of a cooperative Local Search parallel solver using data parallelism. The key idea is to take advantage of all communications tools available in this APGAS model, to exchange information between different solver instances in order to obtain a more efficient and scalable solver implementation. We also plan to test the behavior of a cooperative implementation, under different HPC architectures, such as the many-core Xeon PHI, GPGPU accelerators and grid computing platforms like Grid5000.

References

1. David Butenhof. *Programming With Posix Threads*. Addison-Wesley Professional, 1997.
2. Yves Caniou, Philippe Codognet, Daniel Diaz, and Salvador Abreu. Experiments in parallel constraint-based local search. In Peter Merz and Jin-Kao Hao, editors, *Evolutionary Computation in Combinatorial Optimization - 11th European Conference, EvoCOP 2011, Torino, Italy, April 27-29, 2011. Proceedings*, volume 6622 of *Lecture Notes in Computer Science*, pages 96–107. Springer, 2011.
3. Philippe Codognet and Daniel Diaz. Yet another local search method for constraint solving. In Kathleen Steinhöfel, editor, *Stochastic Algorithms: Foundations and Applications*, pages 342–344. Springer Berlin Heidelberg, London, 2001.
4. Philippe Codognet and Daniel Diaz. An Efficient Library for Solving CSP with Local Search. In *5th international Conference on Metaheuristics*, pages 1–6, Kyoto, Japan, 2003.
5. Teodor Gabriel Crainic and Michel Toulouse. Parallel Meta-Heuristics. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, number May, pages 497–541. Springer US, 2010.
6. Cray Inc. *Chapel Language Specification Version 0.91*. 2012.
7. Daniel Diaz, Salvador Abreu, and Philippe Codognet. Targeting the Cell Broadband Engine for constraint-based local search. *Concurrency and Computation: Practice and Experience (CCPE)*, 24(6):647–660, 2011.
8. Daniel Diaz, Florian Richoux, Yves Caniou, Philippe Codognet, and Salvador Abreu. Parallel local search for the costas array problem. In *PCO'12, Parallel Computing and Optimization*, Shanghai, China, 2012. IEEE.
9. Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *Wiley: UPC: Distributed Shared Memory Programming - Tarek El-*. Wiley, 2005.
10. I.P. Gent and T. Walsh. "CSPLib: a benchmark library for constraints. Technical report, 1999.
11. Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, July 1997.
12. Serdar Kadioglu and Meinolf Sellmann. Dialectic Search. In *Principles and Practice of Constraint Programming (CP)*, volume 5732, pages 486–500, 2009.
13. Khronos OpenCL Working Group. *OpenCL Specification*. 2008.
14. Rui Machado, Salvador Abreu, and Daniel Diaz. Parallel Local Search : Experiments with a PGAS-based programming model. In *12th International Colloquium on Implementation of Constraint and Logic Programming Systems*, pages 1–17, Budapest, Hungary, 2012.

15. Rui Machado and Carsten Lojewski. The Fraunhofer virtual machine: a communication library and runtime system based on the RDMA model. *Computer Science - R&D*, 23(3-4):125–132, 2009.
16. NVIDIA. CUDA C Programming Guide, 2013.
17. OpenMP. The OpenMP API specification for parallel programming.
18. Francesca Rossi, Peter Van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier Science, 2006.
19. Camille Salinesi, Raul Mazo, Olfa Djebbi, Daniel Diaz, and Alberto Lora-michiels. Constraints : the Core of Product Line Engineering. In *Conference on Research Challenges in Information Science (RCIS)*, number ii, pages 1–10, Guadeloupe, French West Indies, France, 2011.
20. Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. The Asynchronous Partitioned Global Address Space Model. In *The First Workshop on Advances in Message Passing*, pages 1–8, Toronto, Canada, 2010.
21. Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification - Version 2.3. Technical report, 2012.
22. Mark Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI : The Complete Reference*. The MIT Press, 1996.
23. Charlotte Truchet, Florian Richoux, and Philippe Codognet. Prediction of parallel speed-ups for las vegas algorithms. *CoRR*, abs/1212.4287, 2012.
24. UPC Consortium, editor. *UPC Language Specifications*. 2005.