HAL
archives-ouvertes.fr

Towards Parallel Constraint-Based Local Search with the X10 Language

Danny Munera, Daniel Diaz, Salvador Abreu

▶ To cite this version:

Danny Munera, Daniel Diaz, Salvador Abreu. Towards Parallel Constraint-Based Local Search with the X10 Language. 20th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2013), Sep 2013, Kiel, Germany. pp.168-182, 2013. <hal-00874633>

HAL Id: hal-00874633
https://hal-paris1.archives-ouvertes.fr/hal-00874633

Submitted on 18 Oct 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Towards Parallel Constraint-Based Local Search with the X10 Language

Danny Munera[1], Daniel Diaz[1], and Salvador Abreu[2]

[1] University of Paris 1-Sorbonne, France
Danny.Munera@malix.univ-paris1.fr, Daniel.Diaz@univ-paris1.fr
[2] Universidade de Évora and CENTRIA, Portugal
spa@di.uevora.pt

Abstract. In this study, we started to investigate how the Partitioned Global Address Space (PGAS) programming language X10 would suit the implementation of a Constraint-Based Local Search solver. We wanted to code in this language because we expect to gain from its ease of use and independence from specific parallel architectures. We present the implementation strategy, and search for different sources of parallelism. We discuss the algorithms, their implementations and present a performance evaluation on a representative set of benchmarks.

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 [16, 17]. 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 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 [15] for shared-memory multiprocessors and multicore CPUs, MPI [20] for distributed-memory clusters or CUDA [14] and OpenCL [10] 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 [19], Unified Parallel C [7] or Chapel [5]. 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 [19].

The remainder of this article is organized as follows: Section 2 discusses the PGAS Model and briefly introduces the X10 programming language. Section 3 introduces native X10 implementations exploiting different sources of parallelism of the Adaptive Search algorithm. Section 4 presents an evaluation of these implementations. A short conclusion ends the paper.

2 X10 and the Partitioned Global Address Space (PGAS) model

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) [20] is a de-facto programming standard. The key idea in MPI is to decompose the computation over a collection of processes with private memory space. These 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 has 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 *pthreads* [1] or OpenMP [15].

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 [19], Unified Parallel C [22] and Chapel [5] are examples of PGAS-enabled languages, but there exist also PGAS-based IPC libraries such as GPI [12], for use in traditional programming languages. For the experiments described herein, we used the X10 language.

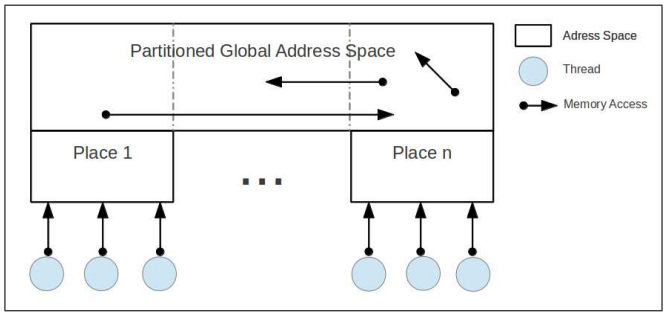

Fig. 1. PGAS Model

X10 [19] 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 [18]. Through this model X10 can support different levels of concurrency with simple language constructs.

There are two main abstractions in the 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.

X10 implements the major components of the PGAS model, by the use of places and activities. 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 for 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/.

3 Native X10 Implementations of Adaptive Search

In order to take advantage of the parallelism it is necessary to identify the sources of parallelism of the Adaptive Search algorithm. In [4], the authors survey the state-of-the-art of the main parallel meta-heuristic strategies and discuss general design and implementation principles. They classify the decomposition of activities for parallel work in two main groups: *functional parallelism* and *data parallelism* (also known as OR-parallelism and AND-parallelism in the Logic Programming community).

On the one hand, in *functional parallelism* different tasks run on multiple compute instances across the same or different datasets. 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. This article reports on our experiments concerning both kinds of parallelism applied to the Adaptive Search method.

3.1 Sequential Implementation

Our first experiment with AS in X10 was to develop a sequential implementation corresponding to a specialized version of the Adaptive Search for permutation problems [13][3].

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. As we will see below, we experimented with two versions of Functional Parallelism (FP1 and FP2) and a Data Parallelism version (called Random Walk, i.e. RW).

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.

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 *solve* method receives a *CSPModel* instance as parameter. On line 8, the CSP variables of the model are initialized with a random permutation. On the next line the total cost of the current configuration is computed. The *while* instruction on line 10 corresponds to the main loop of the algorithm. The *selectVarHighCost* function (Line 12) selects the

[3] In a permutation problem, all N variables have the same initial domain of size N and are subject to an implicit *all-different* constraint. The associated algorithm is reported in the appendix.

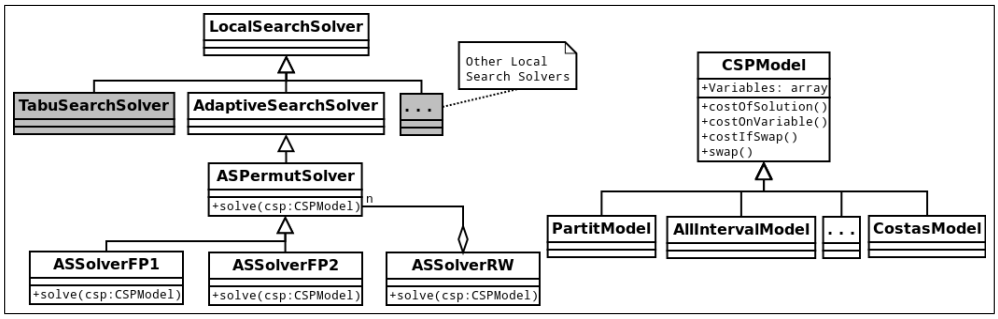

Fig. 2. X10 Class Diagram basic project

variable with the maximal error and saves the result in the *maxI* variable. The *selectVarMinConflict* function (Line 13) selects the best neighbor move from the highest cost variable *maxI*, and saves the result in the *minJ* variable. Finally, if no local minimum is detected, the algorithm swaps the variables *maxI* and *minJ* (permutation problem) and computes the total cost of the resulting new configuration (Line 16). The solver function ends if the *totalCost* variable equals 0 or when the maximum number of iterations is reached.

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 }
```

3.2 Functional Parallel Implementation

Functional parallelism is our first attempt to parallelize the Adaptive Search algorithm. The key aim for this implementation is to decompose the problem

into different tasks, each task working in parallel on the same data. To achieve this objective it is necessary to change the inner loop of the sequential Adaptive Search algorithm.

In this experiment, we decided to change the structure of the *selectVarHighCost* function, because therein lies the most costly activities performed in the inner loop. The most important task performed by this function is to go through the variable array of the CSP model to compute the cost of each variable (in order to select the variable with the highest cost). A X10 skeleton implementation of *selectVarHighCost* function is presented in Listing 1.2.

Listing 1.2. Function selVarHighCost in X10

```
1  public def selectVarHighCost( csp : CSPModel ) : Int {
2    . . . local variables . . .
3    // main loop: go through each variable in the CSP
4    for (i = 0; i < size; i++) {
5      . . . count marked variables . . .
6      cost = csp.costOnVariable (i);
7      . . . select the highest cost . . .
8    }
9    return maxI; // (index of the highest cost)
10 }
```

Since this function must process the entire variable vector at each iteration, it is then natural to try to parallelize this task. For problems with many variables (e.g. the magic square problem involves N^2 variables) the gain could be very interesting. We developed a *first approach* (called FP1), in which `n` single activities are created at each iteration. Each activity processes a portion of the variables array and performs the required computations. The X10 construct *async* was chosen to create individual *activities* sharing the global array. Listing 1.3 shows the X10 skeleton code for the *first approach* of the *functional parallelism* in the function *selectVarHighCost*.

Listing 1.3. First approach to *functional parallelism*

```
1  public def selectVarHighCost (csp : CSPModel) : Int {
2    // Initialization of Global variables
3    var partition : Int = csp.size/THNUM;
4    finish for(th in 1..THNUM){
5      async{
6        for (i = ((th−1)∗partition); i < th∗partition; i++){
7          . . . calculate individual cost of each variable . . .
8          . . . save variable with higher cost . . .
9        }
10     }
11   }
12   . . . terminate function: merge solutions . . .
13   return maxI; //(Index of the higher cost)
14 }
```

In this implementation the constant `THNUM` on line 4 represents the number of concurrent activities that are deployed by the program. On the same line,

the keyword *finish* ensures the termination of all spawned activities. Finally, the construct *async* on line 5 spawns independent individual tasks to cross over a portion of the variable array (sentence *for* on line 6). With this strategy we face up with a well known problem of functional parallelism: the overhead due to the management of fine-grained activities. As expected results are not good enough (see Section 4 for detailed results).

In order to limit the overhead due to activity creation, we implemented a *second approach* (called FP2). Here the n working activities are created at the very beginning of the solving process, just before the main loop of the algorithm. These activities are thus available for all subsequent iterations. However, it is necessary to develop a synchronization mechanism to assign tasks to the working activities and to wait for their termination. For this purpose we created two new classes: *ComputePlace* and *ActivityBarrier*. *ComputePlace* is a compute instance, which contains the functionality of the working activities. *ActivityBarrier* is a very simple barrier developed with X10 monitors (X10 concurrent package).

Listing 1.4 shows the X10 implementation of the *second approach*.

Listing 1.4. Second approach to *functional parallelism*

```
1  public class ASSolverFP1 extends ASPermutSolver{
2    val computeInst : Array[ComputePlace];
3    var startBarrier : ActivityBarrier;
4    var doneBarrier : ActivityBarrier;
5
6    public def solve(csp : CSPModel):Int{
7      for(var th : Int = 1; th <= THNUM ; th++)
8        computeInst(th)[4] = new ComputePlace(th , csp);
9
10     for(id in computeInst)
11       async computeInst(id).run();
12
13     while(total_cost!=0){
14       . . . restart code . . .
15       for(id in computeInst)
16         computeInst(id).activityToDo = SELECVARHIGHCOST;
17
18       startBarrier.wait(); // send start signal
19       // activities working...
20       doneBarrier.wait(); // work ready
21       maxI=terminateSelVarHighCost();
22       . . . local min tabu list, reset code . . .
23     }
24     // Finish activities
25     for(id in computeInst)
26       computeInst(id).activityToDo = FINISH;
27
28     startBarrier.wait();
29     doneBarrier.wait();
```

[4] Remark: in X10 the array notation is *table(index)* instead of *table[index]* as in C.

```
30        return totalCost;
31      }
32    }
```

This code begins with the definition of three global variables on lines 2-4: *computeInst*, *startBarrier* and *doneBarrier*; *computeInst* is an array of *ComputePlace* objects, one for each working activity desired. *startBarrier* and *doneBarrier* are *ActivityBarrier* instances created to signalize the starting and ending of the task in the compute place. On lines 7-11, before the main loop `THNUM` working activities are created and started over an independent X10 activity. When the algorithm needs to execute the *selectVarHighCost* functionality, the main activity assigns this task putting a specific value into the variable *activityToDo* in the corresponding instance of the *ComputePlace* class (lines 15 and 16), then the function *wait()* is executed over the barrier *startBarrier* to notify all working activities to start (line 18). Finally, the function *wait()* is executed over the barrier *doneBarrier* to wait the termination of the working activities (line 20). Then on line 21 the main activity can process the data with the function *terminateSelVarHighCost*. When the main loop ends, all the working activities are notified to end and the *solve* function returns (lines 25-30). Unfortunately, as we will see below, the improvement of this *second approach* is not important enough (and, in addition, it has its own overhead due to synchronization mechanisms).

3.3 Data Parallel Implementation

A straightforward implementation of data parallelism in the Adaptive Search algorithm is the multiple independent Random Walks (IRW) approach. The idea is to use isolated sequential Adaptive Search solver instances dividing the search space of the problem through different random starting points. This strategy is also known as Multi Search (MPSS, Multiple initial Points, Same search Strategies) [4] and has proven to be very efficient [6, 11].

The key of this implementation is to have several independent and isolated instances of the Adaptive Search Solver applied to the same problem model. The problem is distributed to the available processing resources in the computer platform. Each solver runs independently (starting with a random assignment of values). When one instance finds a solution it is necessary to stop all other running instances. This is achieved using a termination detection communication strategy. 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 1.5.

Listing 1.5. 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    }
27  }
```

For this implementation the *ASSolverRW* class was created. The algorithm has two global distributed arrays: *solDist* and *cspDist* (lines 2 and 3). As explained in Section 2, the *DistArray* class creates an array which is spread across multiple X10 places. In this case, an instance of *ASPermutSolver* and *CSPModel* are stored at each available place in the program. On lines 5 and 6 function *make* creates and initializes the ditributed vector in the region created by the function *Dist.makeUnique()* (*makeUnique* function creates a distribution over a region that maps every point in the region to a distinct place, and which maps some point in the region to every place). 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. On 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* on line 19) and into the *async* block to change the value of the *kill* variable. On line 18, the sentence *if (here.id != k.id)* filters all places which are not the winning one (*here*). Finally, the function returns the solution of the fastest place on line 25.

4 Performance Analysis

In this section, we present and discuss our experimental results of our X10 implementations of the Adaptive Search algorithm. The testing environment used 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 as well as a system based on 4 16-core AMD Opteron 6272 CPUs running at 2.1GHz.

We used a set of benchmarks composed of four classical problems in constraint programming: the magic square problem (MSP), the number partitioning problem (NPP) and the all-interval problem (AIP), all three taken from the CSPLib [8]; also we include the Costas Arrays Problem (CAP) introduced in [9], which is a very challenging real problem. The problems were all tested on significantly large instances. The interested reader may find more information on these benchmarks in [13].

It is worth noting, at the software level, that 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 more mature and faster for scientific computation. 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 report raw times in seconds (average of 100 runs) and relative speed-ups. These tables respect 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 (places), and the last column presents the execution time of the problem with the highest number of places.

4.1 Sequential Performance

Even if our first goal in using X10 is parallelism, it is interesting to compare the sequential X10 implementation with a reference implementation: our low-level and highly optimized C version initially used in [2, 3] and continuously improved since then. The X10 implementation appears to be 3 to 5 times slower than the C version: this is not a prohibitive price to pay, if one takes into account the possibilities promised by X10 for future experimentation.

A possible explanation of the difference between the performances of both implementations is probably the richness of the X10 language (OOP, architecture abstractions, communication abstractions, etc.). Also, maybe it is necessary to improve our X10 language skills good enough to get the best performance of this tool.

4.2 Functional Parallel Performance

Table 1 shows the results of the *first version* of the *functional parallelism* X10 implementation. Only two benchmarks (2 instances of MSP and CAP) are presented. Indeed, we did not investigate this approach any further since the results are clearly not good. Each problem instance was executed with a variable number of activities (`THNUM = 2, 4` and `8`). It is worth noting, that the environmental X10 variable *X10_NTHREADS* was passed to the program with an appropriate value to each execution. This variable controls the number of initial working threads per place in the X10 runtime system.

Problem instance	time (s) seq.	speed-up with k places 2	4	8	time (s) 8 places
`MSP-100`	11.98	0.86	0.95	0.77	15.49
`MSP-120`	24.17	1.04	0.97	0.98	24.65
`CAP-17`	1.56	0.43	0.28	0.24	6.53
`CAP-18`	12.84	0.51	0.45	0.22	57.16

Table 1. Functional Parallelism – first approach (timings and speed-ups)

As seen in Table 1, for all the treated cases the obtained speed-up is less than 1 (i.e. a slowdown factor), showing a deterioration of the execution time due to this parallel implementation. So, it is possible to conclude that no gain time is obtainable in this approach. To analyze this behavior it is important to return to the description of the Listing 1.3. As already noted, the parallel function *selVarHighCost* in this implementation are located into the main loop of the algorithm, so `THNUM` activities are created, scheduled and synchronized at each iteration in the program execution, being a very important source of overhead. The results we obtained suggest that this overhead is larger than the improvement obtained by the implementation of this parallel strategy.

Turning to the *second approach*, Table 2 shows the results obtained with this strategy. Equally, the number of activities spawn, in this case at the beginning, was varied from 2 to 8.

Problem instance	time (s) seq.	speed-up with k places 2	4	8	time (s) 8 places
`MSP-100`	11.98	1.15	0.80	0.86	13.87
`MSP-120`	24.17	1.23	0.94	0.63	38.34
`CAP-17`	1.56	0.56	0.30	0.25	6.35
`CAP-18`	12.84	0.74	0.39	0.27	46.84

Table 2. Functional Parallelism – second approach (timings and speed-ups)

Even if the results are slightly better, there is no noticeable speed-up. This is due to a new form of overhead due to the synchronization mechanism which is used in the inner loop of the algorithm to assign tasks and to wait for their termination (see Listing 1.4).

4.3 Data Parallel Performance

Table 3 and Figure 3 document the speedups we obtained when resorting to data parallelism. Observe that, for this particular set of runs, we used a different hardware platform, with more cores than for the other runs.

Problem instance	time (s) seq.	speed-up with k places				time (s) 32 places
		8	16	24	32	
`AIP-300`	56.7	4.7	7.1	9.9	10.0	5.6
`NPP-2300`	6.6	6.1	9.8	10.5	12.0	0.5
`MSP-200`	365	8.3	12.2	13.6	14.6	24.9
`CAP-20`	731	5.6	12.0	16.1	20.5	35.7

Table 3. Data Parallelism (timings and speed-ups)

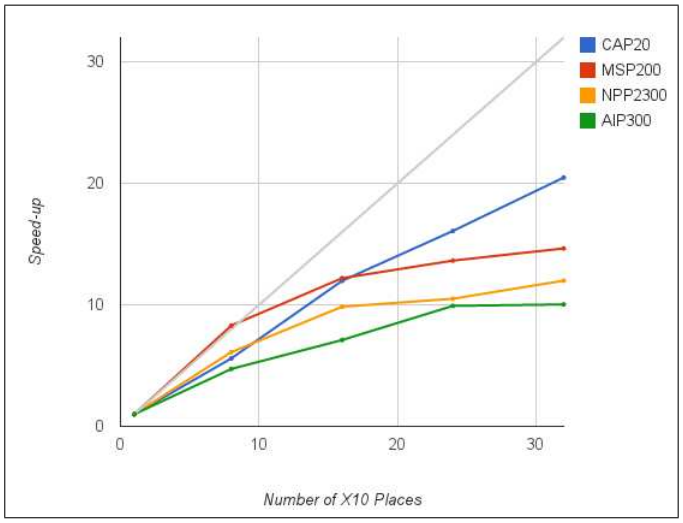

Fig. 3. Speed-ups for the most difficult instance of each problem

The performance of data parallel version is clearly above the performance of the functional parallel version. The resulting average runtime and the speed-ups obtained in the entire experimental test performed seems to lie within the

predictable bounds proposed by [21]. The Costas Arrays Problem displays remarkable performance with this strategy, e.g. the CAP reaches a speed-up of 20.5 with 32 places. It can be seen that the speed-up increases almost linearly with the number of used places. However, for other problems (e.g. MSP), the curve clearly tends to flat when the number of places increases.

5 Conclusion and Future Work

We presented different parallel X10 implementations of an effective Local Search algorithm, Adaptive Search in order to exploit various sources of parallelism. We first experimented two functional parallelism versions, i.e. trying to divide the inner loop of the algorithm into various concurrent tasks. This turns out to yield no speed-up at all, most likely because of the bookkeeping overhead (creation, scheduling and synchronization) that is incompatible with such a fine-grained level of parallelism.

We then proceeded with a data parallel implementation, in which the search space is decomposed into 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 with monotonously increasing speed-ups in all problems we studied, although they taper off after some point.

The main result we draw from this experiment, is that X10 has proved a suitable platform to exploit parallelism in different ways for constraint-based local search solvers. These entail experimenting with different forms of parallelism, ranging from single shared memory inter-process communication to a distributed memory programming model. Additionally, the use of the X10 implicit communication mechanisms allowed us to abstract away from the complexity of the parallel architecture with a very simple and consistent device: the *distributed arrays* and the *termination detection system* in our data parallel implementation.

Considering that straightforward forms of parallelism seem to get lower gains as we increase the number of cores, we want to look for ways of improving on this situation. Future work will focus on the implementation of a cooperative Local Search parallel solver based on data parallelism. The key idea is to take advantage of the many communications tools available in this APGAS model, to exchange information between different solver instances in order to obtain a more efficient and, most importantly, 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.

References

1. David Butenhof. *Programming With Posix Threads.* Addison-Wesley Professional, 1997.

2. 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.
3. 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.
4. 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.
5. Cray Inc. *Chapel Language Specification Version 0.91*. 2012.
6. Daniel Diaz, Salvador Abreu, and Philippe Codognet. Targeting the Cell Broadband Engine for constraint-based local search. *Concurrency and Computation: Practice and Experience (CCP&E)*, 24(6):647–660, 2011.
7. Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *Wiley: UPC: Distributed Shared Memory Programming - Tarek El-*. Wiley, 2005.
8. I.P. Gent and T. Walsh. "CSPLib: a benchmark library for constraints. Technical report, 1999.
9. Serdar Kadioglu and Meinolf Sellmann. Dialectic Search. In *Principles and Practice of Constraint Programming (CP)*, volume 5732, pages 486–500, 2009.
10. Khronos OpenCL Working Group. *OpenCL Specification*. 2008.
11. 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.
12. 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.
13. Danny Múnera, Daniel Diaz, and Salvador Abreu. Experimenting with X10 for Parallel Constraint-Based Local Search. In Ricardo Rocha and Christian Theil Have, editors, *Proceedings of the 13th International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS 2013)*, August 2013.
14. NVIDIA. CUDA C Programming Guide, 2013.
15. OpenMP. The OpenMP API specification for parallel programming.
16. Francesca Rossi, Peter Van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier Science, 2006.
17. 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.
18. 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.
19. Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification - Version 2.3. Technical report, 2012.
20. Mark Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI : The Complete Reference*. The MIT Press, 1996.
21. Charlotte Truchet, Florian Richoux, and Philippe Codognet. Prediction of parallel speed-ups for las vegas algorithms. 2013.
22. UPC Consortium, editor. *UPC Language Specifications*. 2005.

Algorithm 1 Adaptive Search Base Algorithm

Input: problem given in CSP format:

- set of variables $V = \{X_1, X_2 \cdots\}$ 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 ← 0
2: repeat
3:     Restart ← Restart + 1
4:     Iteration ← 0
5:     Compute a random assignment A of variables in V
6:     Opt_Sol ← A
7:     Opt_Cost ← cost(A)
8:     repeat
9:         Iteration ← Iteration + 1
10:        Compute errors constraints in C and project on relevant variables
11:        Select variable X with highest error: MaxV
12:                                                    ▷ 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 ≥ RL then
17:                randomly reset some variables in V
18:                                                ▷ and unmark those Tabu
19:            end if
20:        else
21:            swap(MaxV,MinConflictV),
22:                                          ▷ modifying the configuration A
23:            if cost(A) < Opt_Cost then
24:                Opt_Sol ← A
25:                Opt_Cost ← costs(A)
26:            end if
27:        end if
28:    until Opt_Cost = 0 (solution found) or Iteration ≥ MI
29: until Opt_Cost = 0 (solution found) or Restart ≥ MR
30: output(Opt_Sol, Opt_Cost)
```