POSL: A Parallel-Oriented Solver Language

Thesis for the Degree of
Doctor of Computer Science

Alejandro Reyes Amaro

Doctoral School STIM

Academic advisors:
Eric Monfroy[1], Florian Richoux[2]

[1]Department of Informatics
Faculty of Science
University of Nantes
France

[2]Department of Informatics
Faculty of Science
University of Nantes
France

Submitted: dd/mm/2016

Assessment committee:

Prof. (1)

Institution (1)

Prof. (2)

Institution (2)

Prof. (3)

Institution (3)

ISBN ??

Part I

Presentation

1

Prior works leading to POSL

In this chapter are presented Prior works leading to POSL*. In Section 1.1 we present a brief work where we applied the problem subdivision approach to solve the k-medoids problem in parallel, as a first attempt aiming for the right direction in order to find the proper approach. Finally we present in Section 1.2 a study applying the* ParamILS *tool in order to find the optimum parameter configuration to Adaptive Search solver.*

Contents

1.1 Domain Split

Usually, to solve some problem using parallelism, our first thought is either partitioning the problem into a set of sub-problems, or dividing its search space. In both cases, the idea is to solve a set of problems, all of them smaller and easer than the original one, and combining all the solution to obtain the solution of the original problem. In [83] a study of the impact of space-partitioning techniques on the performance of parallel local search algorithms to tackle the *k-medoids* clustering problem is presented. The authors use a parallel local search, in order to improve the scalability of the sequential algorithm, which is measured, in terms of the quality of the solution, with respect to the sequential algorithm. In this work two main techniques for domain partitioning are presented: *space-filling curves*, used to reduce any n-dimensional representation into a one-dimension space; and *k-Means* algorithm. We found that the used methods for domain partitioning do not take into account the number of clients associated to each new sub-domain. This results in an unbalanced distributions of workload phenomenon. For that reason the goal of this study was designing some ideas to tackle the same problem. [i]

The *k-medoids* problem aims to select a subset M of k points (the medoids) from a set of points S, such that the average distance from any point to its closest medoid is minimized. It finds a lot of applications in the industry, like in resource allocation, data mining, among others. It is quite similar to *k-means* problem, except that the set of medoids M is forced to be a subset of S.

We propose Algorithm 1, which represents the backbone of our idea. This algorithm takes a set of $\mathbb{R}^2$ *points* (representing the locations of the clients) and returns a partition of size K. Such a set of points is called a *domain* and the partition a *sub-domain*. At each intermediate step i, we have a set (list) of sub-domains. The algorithm takes the most populated one, splits it into two (or four, depending on the strategy) new sub-domains and includes them in

[i] This work falls within the framework of the Ulysses project between France and Ireland.

the list. The stop condition will depends on the fallowed approach (see below).

Algorithm 1: Domain_Split

input : U: Set of client locations

output : Q = $\{Q_i\}_{i=1...K}$: K subsets of U

```
1 A ← U
2 Q.Insert(A)
3 repeat
4 |   A ← Q.GetNext() /* It also removes the returned element */
5 |   [a1, a2] ← Split(A)
6 |   Q.Insert(a1, a2)
7 until <some condition>
```

First of all, we make clear some details of the algorithm exposed above:

- `Insert(...)` Inserts a set (or two) into the data structure.
- `GetNext()` Returns the next sub-set to be divided, tacking into account the *split strategy* (see below).
- `Split(...)` Returns two sub-domains as a subdivision of the given domain (parameter).

In the next sub-sections we answer two main question arising at this point:

a) How to split the each sub-domain? (<*Split*> function on line 5) It refers to, given a set of points (locations), how to decide which of them will be included into one sub-domain and which of them into the other.

b) How much to split each sub-domain? (<*some condition*> on line 3) It refers to decide when to stop splitting the domain.

1.1.1 Domain Splitting. General point of view

In order to split the domain, we can think in some approaches, taking into account the number of available cores and the number of metro-nodes we want to place in the system. In the article of A. Arbelaez and L. Quesada a domain split taking into account the number of available cores for parallel calculus is proposed. In our approach we intend to extend this idea keeping in mind also the number of metro-nodes to allocate. Following, we propose three variants to face the problem:

a) **one metro-node per core**: In this variant we can assign one metro-node to each core, and in this case, splitting the domain in K sub-domains (K is the number of

cores). It means that the algorithm will compute the best position for a metro-node in a current sub-domain.

- In this case we only have to replace the line 3 by something like: **for** $i \leftarrow 1$ **to** N **do** ..., where N is the number of metro-nodes.
- The ideal scenario here is when $N = K$ which is not probable at all. So we only should study the case when they are different
- In that case, we need to distribute efficiently the metro-nodes into the domain subdivisions, but here, one possible scenario arise: it can be happen that, depending on the followed domain-split strategy, we were trying to allocate a metro-node into an area with a few clients. This produce a very local point of view of the problem. That is the reason why we propose the following *second variant*.

b) **Incomplete partition**: To split a sub-domain if it can generate sub-sub-domains containing at least C clients. It means, for example, if there exist in list a sub-domain with 8 clients, and the number C is fixed in $C = 5$, this sub-domain can not be divided, because it will generate for sure a sub-sub-domain with less that 5 clients. In this case more than one metro-node would be allocated in some of those specific sub-domains, because we will have more metro-nodes than sub-domains in our model.

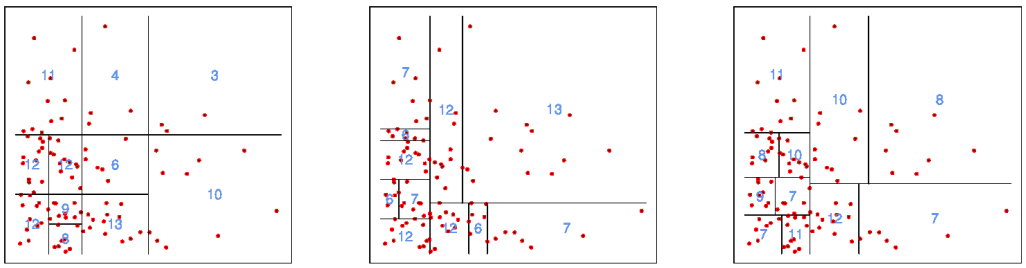
- Of course, using this variant we can find situations described in the first variant. In that case we should proceed consequently.

c) **Combination**: This variant is just a combination of the two previous variants. Then, we will be working with two parameters:

- $C \rightarrow$ Lower bound of clients for new sub-domains. It means that a sub-domain can be divided iff the new produced sub-domains will contains more than C clients.
- $M \rightarrow$ Lower bound of metro-nodes to be allocated in a sub-domain. In this case we will split the domain while it can be ensure that at least M metro-nodes will be assigned to each sub-domain.

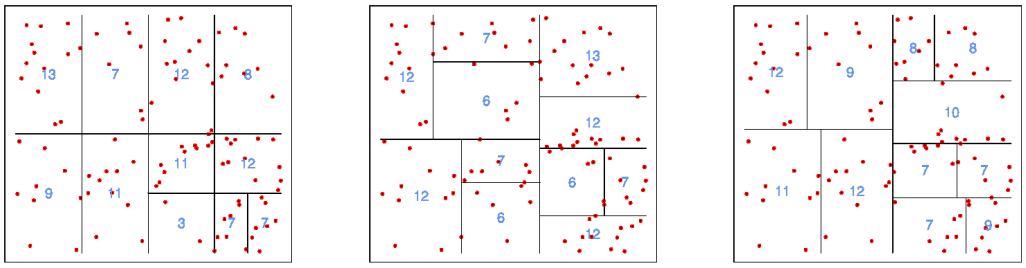
1.1.2 Split strategies

In this sub-section we first discuss three ideas to split the domain. They take a sub-domain and produce other two, dividing the space vertically or horizontally, depending on the shape of the current sub–domain. In other section we expose another strategy to follow, in which the subdivision of a sub-domain produces four sub-domains.



(a) Geometrical Split **(b)** Cardinal Split **(c)** Mean Split

Figure 1.1: Split domain of point normally distributed $\mathcal{N}(0, 0.35)$



(a) Geometrical Split **(b)** Cardinal Split **(c)** Mean Split

Figure 1.2: Split domain of point uniformly distributed

In all of them the number of clients in each sub-domain is taken into account, but in different ways. A common feature between them is that, at the moment of splitting a given sub-domain, it will be done in a perpendicular way (either to the x–axis or to the y–axis, depending on the characteristic of the sub-domain to be divided). The reference point to divide the sub-domain is called the *cut point* of the sub-domain.

a) **Geometrical Split**:

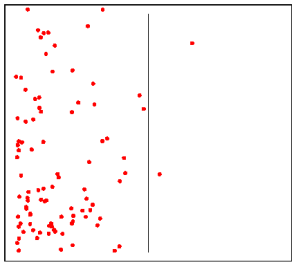


Figure 1.3: If we assume $C = 3$ then this set can't be divided

- Split criterion: *Geometrical* → Dividing the region in tow parts with the same area.
- Cut point: The middle point of the x–axis (or y–axis, depending on the shape)
- Result: Two equal regions, but with different amount of clients (see Figure 1.1a). In the next step, the next sub-domain to divide will be, from those already divided, the most populated one (the sub-domain with more clients).
- Problem: Maybe we need to divide a sub-domain because it has a lot of client, but in the other hand it generate a sub-domain with a few clients, as you can see in the Figure 1.3.
- Benefit: Very fast split. If we attack scenarios with point uniformly distributed, the behavior is pretty much desirable, as we can see in the Figure 1.2a.

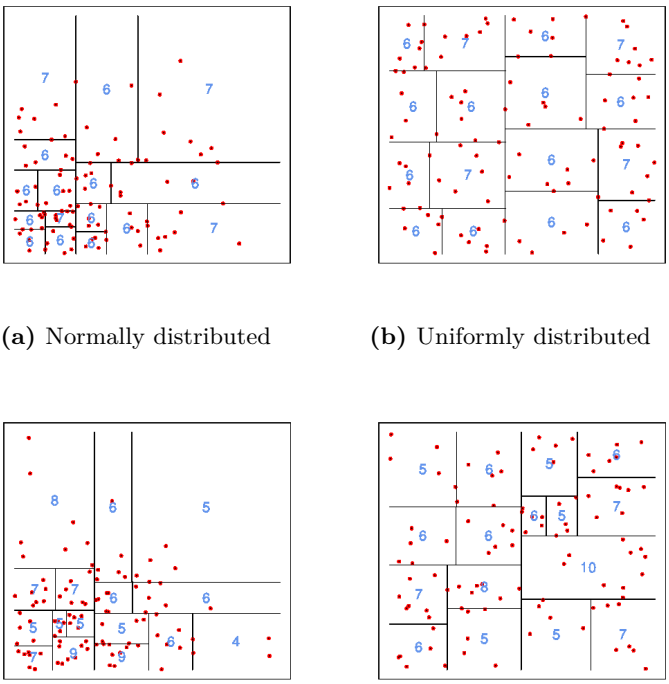
b) **Cardinal Split**

- Split criterion: The number of clients, i.e., a sub-domain is divided in such a way that in the resulting sub-sub-domains there will be the same number of clients.
- Cut point: The current sub-domain is ordered and the x–axis (or y–axis, depending on the shape) of the element (location of the client) right in the center is selected to be the *cut point*
- Result: Two regions with the same (± 1) amount of clients (see Figure 1.1a). In the next step, the next sub-domain to divide will be, from those already divided, the most populated one (the sub-domain with more clients).
- Problem: The subdivision process is a bit more costly: we need to group the clients on both sides of a *perfect pivot*[ii].
- Benefit: It guarantees the same cardinality in both new sub-sub-domains.

c) **Mean Split**

- Split criterion: This is a mid-point strategy between the two previous. The goal is to find a *cut point* to group the elements of the current sub-domain, but in a easy way (fast), for that reason we do not compute the exact middle point to produce two sub-sub-domains with the same cardinality, as in the previous approach, instead of that we work with his expected value: the arithmetic mean.
- Cut point: We compute the mean of the x–axis (or y–axis) of the elements (locations) of the current sub-domain, and it will be the *cut point*

[ii] Element of a set with the same number of elements lower and greater than him



(a) Normally distributed **(b)** Uniformly distributed

(c) Normally distributed **(d)** Uniformly distributed

Figure 1.4: Domain divided in $2^4 = 16$ sub-domains: (a) - (b) Cardinal; (c) - (d) Mean.

- Result: Two regions with not exact information about their sizes or their cardinality.
- Problem: *Idem.*
- Benefit: The *cut point* can be obtained by a $O(n)$ number of operations. As we can see in the preliminary results[iii] (Figures 1.4c and 1.4d), the behavior of this technique is near to the *cardinal split* technique, at least for normally and uniformly distributed sets of point.

1.1.3 Conclusion

In this section we present a theoretical and not validated work where we applied the *search space subdivision* approach to solve *k-medoids problem* in parallel. We have proposed some different strategies and we have showed graphically some characteristics of them.

[iii] The experiments are coded in R[119].

1.2 Tunning methods for local search algorithms

In this section we present our results applying ParamILS (version 2.3)[iv]. ParamILS (first introduced by Hutter, Hoos and St utzle in 2007), is a stochastic local search approach for automated algorithm configuration. The source is available on internet and includes some examples that you can run and see how the tool works. In addition, it brings a complete User Guide with a compact explanation about how to use it with a specific solver [109, 108]. In this study we used it to tune *Adaptive Search* solver[v].

The first step was building a *wrapper* in C++ language, in order to tune more than one problem with the same code. The goal of doing this is using the tool to tune the solver, i.e., finding the best parameter configuration for a specific problem, but also the best parameter configuration to solve any kind of benchmark (a general parameter configuration).

Following we present in Table 1.1 the parameter list that we worked with:

Table 1.1: Adaptive Search parameters

Parameter	Type	Description
-P	PERCENT	probability to select a local min (instead of staying on a plateau)
-f	NUMBER	freeze variables of local min for NUMBER swaps
-F	NUMBER	freeze variables swapped for NUMBER swaps
-l	LIMIT	reset some variables when LIMIT variable are frozen
-p	PERCENT	reset PERCENT of variables

In this section, we explain in details the implementation and the experimentation process.

1.2.1 Using ParamILS

To use the tool ParamILS, we have installed Ruby! 1.8.7 in our computer. We used a laptop Dell XPS 15(Intel Core i7-4702HQ 2.2 GHz, 16384 MB, Dual-channel DDR3L 1600 MHz) with Ubuntu 14.4. To run the tool, we needed to use the following command line:

```
>> ruby param_ils_2_3_run.rb -numRun 0 -scenariofile /.../<scenario_file> -validN 100
```

Where <scenario_file> is the name of the file where we have to put all the information that ParamILS needs to tune the solver (the *tuning scenario file*). We explain its content in the next section.

[iv] Open source program (project) in *Ruby*, available at http://cs.ubc.ca/labs/beta/Projects/ParamILS

[v] An implementation from Daniel Díaz available at https://sourceforge.net/projects/adaptivesearch/

1.2.2 Tuning scenario files

The *tuning scenario file* is a text file with all needed information to tune the solver using ParamILS. It includes where to find the solver binary file, the parameters domains, etc. In our case, the *tuning scenario file* looks like the following:

```
algo = ./QtWrapper_wrapper
execdir = /.../src
deterministic = 0
run_obj = runtime
overall_obj = mean
cutoff_time = 50.0
cutoff_length = max
tunerTimeout = 3600
paramfile = instances/all_intervals-params.txt
outdir = instances/all_intervals-paramils-out
instance_file = instances/.../all_intervals-lower-instances.txt
test_instance_file = instances/.../all_intervals-upper-instances.txt
```

We explain in details each line in this file:

- **`algo`** → An algorithm executable or a call to a wrapper script around an algorithm that aims the input/output format of *ParamILS* (the wrapper).
- **`execdir`** → Directory to execute **`algo`** from: "cd <`execdir`>; <`algo`>"
- **`run_obj`** → A scalar quantifying how "good" a single algorithm execution is, such as its required runtime.
- **`overall_obj`** → While **`run_obj`** defines the objective function for a single algorithm run, **`overall_obj`** defines how those single objectives are combined to reach a single scalar value to compare two parameter configurations. Implemented examples include **mean**, **median**, **q90** (the 90% quantile), **adj__mean** (a version of the mean accounting for unsuccessful runs: total runtime divided by number of successful runs), **mean1000** (another version of the mean accounting for unsuccessful runs: (total runtime of successful runs + 1000 x runtime of unsuccessful runs) divided by number of runs – this effectively maximizes the number of successful runs, breaking ties by the runtime of successful runs; it is the criterion used in most of Frank experiments), and **geomean** (geometric mean, primarily used in combination with **`run_obj`** = **`speedup`**. The

empirical statistic of the cost distribution (across multiple instances and seeds) to be minimized, such as the mean (of the single run objectives). [vi]

- **`cutoff_time`** → The time after which a single algorithm execution will be terminated unsuccessfully. This is an important parameter: if chosen too high, lots of time will be wasted with unsuccessful runs. If chosen too low the optimization is biased to perform well on easy instances only.
- **`tunerTimeout`** → The timeout of the tuner. Validation of the final best found parameter configuration starts after the timeout.
- **`paramfile`** → Specifies the file with the parameters of the algorithms.
- **`outdir`** → Specifies the directory *ParamILS* should write its results to.
- **`instance_file`** → Specifies the file with a list of training instances.
- **`test_instance_file`** → Specifies the file with a list of test instances.

Another important file that we have to compose properly is the *algorithm parameter file*, just following the instruction from [109] *–[...] each line lists one parameter, in curly parentheses the possible values considered, and in square parentheses the default value [...].* Our *algorithm parameter file* looks like follows:

```
P {20, 25, 30, 35, 40, 45, 50, 55, 60} [50]
f {0, 1, 2, 3} [1]
F {0, 1, 2, 3} [0]
l {0, 1, 2, 3} [1]
p {1, 2, 3, 5, 10, 20} [5]
```

In the current *Adaptive Search* implementation, the solver binary file and the problem instance are the same thing. It means that we only have to use the following command to solve the *All–intervals* problem of size K, for example:

```
>> ./all-intervals K
```

So, to use ParamILS we modified a little the code: now our solver takes the size parameter from a text file. That way, the instance file is a text file only containing a number.

The solver we want to tune using ParamILS (*Adaptive Search* in this case), must aims specific input/output rules. For that reason instead of modifying the current code of *Adaptive Search* implementation, we preferred to build a wrapper.

[vi] We use **mean** but maybe we can experiment with other values

1.2.3 Building the wrapper

The algorithm executable must follow the input/output criteria presented below:

Launch command:

```
>> <algo_exectuable> <instance_name> <instance-specific_information> ...
<cutoff_time> <cutoff_length> <seed> <params>
```

- <algo_exectuable> Solver
- <instance_name> In our case, a text file containing only the problem size
- <instance-specific_information> We don't use it
- <cutoff_time> Cut off time for each run of the solver (see above)
- <cutoff_length> We don't use it
- <seed> Random seed
- <params> Parameters and its values

Exmaple:

```
>> ./QtWrapper_320.txt "" 50.0 214483647 524453158 -p 5 -l 1 -f 1 -P 50 -F 0
```

Output:

```
>> <solved>, <runtime>, <runlength>, <best_sol>, <seed>
```

- <**solved**> `SAT` if the algorithm terminates successfully. `TIMESOUT` if the algorithm times out.
- <**runtime**> Runtime
- <**runlength**> -1 (as Frank Hutter recommended)
- <**best_sol**> -1 (as Frank Hutter recommended)
- <**cutoff_length**> We don't use it
- <**seed**> Used random seed

Exmaple:

```
>> SAT, 2.03435, -1, -1, 524453158
```

To build the wrapper we have followed a simple algorithm: launch two concurrent process. In the parent process we translate the input of the wrapper to the input of *Adaptive Search* solver. The solver is executed, and the runtime is measured. After that we post the output properly. In the child process a *sleep* command is executed for <`runtime`> seconds and after that, if the parent process has not finished yet, it is killed, posting a time-out message. See Algorithm 2 for more details.

Algorithm 2: Costas Wrapper

input : Pth_π: problem instance path, k: cut off time, s: random seed, θ: parameters configuration
output: $PiLS_{out}$: Output in a PARAMILS way

```
1  fork() /* Divide the execution in two processes                    */
2  if <in child process> then
3      t_0 ← clock_TIC()
4      strCall ← build_str(" ./AS_Wrapper %1 -s %2 %3", Pth_π, s, θ)
5      systemCall(strCall)
6      t_e ← clock_TOC()
7      killProcess(<parent process>)
8      t ← t_e- t_0
9      return paramilsOutput(SAT, t, s)
10 else
11     sleep(k)
12     killProcess(<child process>)
13     return paramilsOutput(TIMESOUT, k, s)
14 end
```

1.2.4 Using the wrapper

In this section we explain how to use our wrapper to be able to tune easily instances of *All-Interval Series* and *Costas Array* problems. The *All-Interval Series Problem*[vii] is the problem of finding a vector $s = (s_1, \ldots, s_n)$, given $n \in \mathbb{N}$, such that s is a permutation of the vector $(0, 1, \ldots, n-1)$ and the interval vector $v = (|S_2 - s_1|, |S_3 - s_2|, \ldots, |S_n - S_{n-1}|)$ (called an all-interval series of size n) is a permutation of the vector $(1, 2, \ldots, n-1)$. The *Costas Array Problem* consists in finding a Costas array, which is an $n \times n$ grid containing n marks such that there is exactly one mark per row and per column and the $n(n-1)/2$ vectors joining each couple of marks are all different (see below for more details about this problems).

[vii] CSPlib:007 (`http://www.csplib.org/Problems/prob007/`)

1.2.4.1 Factory call

The first step is to implement the class ICallFactory. Here, the string-binary-name for the command call is statically obtained. We present, as example, the class All_IntervalCallFactory:

```
// all_interval_call_factory.h
 class All_IntervalCallFactory:  public ICallFactory
{
    public:
       std::string BuildCall();
       std::string BuildDefaultCall();
};
```

```
// all_interval_call_factory.cpp
 #define ALGO_EXECUTABLE "./all-interval"
 #define DEFAULT_CALL "./all-interval _100.txt"

std::string All_IntervalCallFactory::BuildCall()
{
    return ALGO_EXECUTABLE;
}
std::string All_IntervalCallFactory::BuildDefaultCall()
{
    return DEFAULT_CALL;
}
```

All we have to do is to define our new macro **ALGO_EXECUTABLE** (**DEFAULT_CALL** is not being used)

1.2.4.2 Main method

Let's suppose now that we want to run an algorithm called *mySolver* that receives a file as parameter, called *my_instance_size.txt* (this is mandatory). We have to create (as we've explained before) the class My_SolverCallFactory and defining the macro as follows:

```
#define ALGO_EXECUTABLE "./mySolver"
```

Now, the main method would be exactly like this:

```
int main( int argc,  char* argv[])
{
   shared_ptr<ICallFactory> problem =
      make_shared<My_SolverCallFactory>();
   shared_ptr<TuningData> td =
      (make_shared<TuningData>(argc, argv, problem));

   shared_ptr<ADWrapper> w (make_shared<ADWrapper>());
   string output = w->tune(td);

   cout << output << endl;
    return 0;
}
```

1.2.5 Results

In this section we present the results of applying ParamILS to the resolution of *All-Interval Series* and *Costas Array* problems through *Adaptive Search*. In both cases, we need to chose a set of *training instances*, to train the tuner, and a set of *test instances*, used to obtain the parameter setting.

1.2.5.1 Tuning *Adaptive Search* for *All-Intervals Series Problem*

Study cases:

a) The *training instances set* is composed by instances of *All–Intervals* problems of order N with

$$N \in \{100, 110, 120, 130, 140, 150, 160, 170, 180\}$$

b) The *test instances set* is composed by instances of *All–Intervals* problems of order N with

$$N \in \{190, 200, 210, 220, 230, 240, 250, 260, 265\}$$

c) The time-out for each run is 50.0 seconds

d) The test quality is based on 100 runs

In a **First Experiment** we use the following *parameters domains*:

- **P** `{41, 46, 51, 56, 60, 66, 71, 76, 80}`
- **F, f, l** `{0, 1, 2, 3}`
- **p** `{5, 10, 15, 20, 25, 30, 35}`

Table 1.2 shows results using a time-out of 5.5 hours (20,000 seconds), and Table 1.3 shows results using a time-out of 1 hour. In the second case we where able to perform more runs, due to the available time, but in both cases the training qualities are not so different. However, we can se the difference int the test qualities, and conclude that results using 5 hours of time-out are more reliables.

Initial configuration					**Final best configuration**					**Training quality**	**Number of runs**	**Test quality**
F	P	f	l	p	F	P	f	l	p			
0	66	1	1	25	0	80	2	1	35	0.79666	1780	8.274
2	56	2	2	20	1	80	1	1	10	0.795	1637	5.508
0	41	0	0	5	1	80	3	0	15	0.789	1547	5.8478
3	80	3	3	35	1	80	2	0	10	0.880686	1258	6.15398

Table 1.2: *All-Intervals Series*: `tunerTimeout` = 20,000 seconds

Initial configuration					**Final best configuration**					**Training quality**	**Number of runs**	**Test quality**
F	P	f	l	p	F	P	f	l	p			
0	66	1	1	25	0	80	0	1	25	0.815	384	5.8191
0	66	1	1	25	1	80	1	1	35	0.737	452	6.267
0	66	1	1	25	1	56	0	1	35	1.03	371	9.056
0	66	1	1	25	0	76	0	1	20	0.814	385	4.915
0	66	1	1	25	0	80	3	1	20	0.76	469	5.417
2	56	2	2	20	0	41	0	1	10	0.919	239	18.364
2	56	2	2	20	0	56	1	1	20	0.819	407	5.409
2	56	2	2	20	1	80	1	1	35	0.772	457	5.43
2	56	2	2	20	1	80	0	1	10	0.858	504	5.566
2	56	2	2	20	0	80	1	1	10	0.7845	562	18.944
0	41	0	0	5	0	41	1	0	10	0.9749	367	5.97813
0	41	0	0	5	0	41	1	0	10	0.885	450	5.706
0	41	0	0	5	0	41	1	0	10	0.906	335	18.707
0	41	0	0	5	0	41	1	0	10	0.995	335	19.558
0	41	0	0	5	0	41	0	0	5	0.855	404	5.686
3	80	3	3	35	0	66	3	1	25	0.9118	230	26.585
3	80	3	3	35	0	80	1	1	10	0.732	310	7.875
3	80	3	3	35	0	80	0	1	20	0.816	303	7.2896
3	80	3	3	35	1	80	3	1	35	0.821	327	6.812
3	80	3	3	35	0	80	0	1	30	0.9203	443	5.401

Table 1.3: *All-Intervals Series*: `tunerTimeout` = 3,600 seconds

In a **Second Experiment** we decide to enlarge a bit more the parameters domains and use a time-out of 5 hours. The **Parameters domains** are the following:

- **P** `{10, 20, 30, 40, 50, 60, 70, 80, 90}`
- **F, f, l** `{0, 1, 2, 3, 4, 5, 6, 7, 8}`
- **p** `{10, 20, 30, 40, 50, 60, 70}`

Initial configuration					**Final best configuration**					**Training quality**	**Number of runs**	**Test quality**
F	P	f	l	p	F	P	f	l	p			
0	10	0	0	10	0	40	7	0	50	0.883188	936	6.3191
0	10	0	0	10	0	80	2	1	40	0.774659	1584	5.45674
0	10	0	0	10	0	40	2	0	10	0.96885	1104	6.82643
4	60	4	4	40	0	60	8	1	40	0.90358	1566	5.48127
4	50	4	4	40	0	80	5	1	20	0.78536	1662	11.5649
3	50	4	2	30	0	90	6	1	70	0.79440	1395	5.08108
0	90	0	0	10	1	90	6	1	10	0.859569	1379	5.4286
0	90	0	0	10	1	90	6	1	30	0.80738	1117	5.47126
8	90	8	8	60	0	80	5	1	10	0.834934	1384	5.5377
5	30	2	3	60	0	90	1	0	20	0.862013	1707	5.21837
3	20	2	4	60	0	80	6	1	10	0.805604	1630	5.4467
6	70	1	3	50	0	80	5	1	10	0.792600	1344	5.46558
6	40	1	3	30	1	80	7	0	20	0.822703	1977	5.41185

Table 1.4: *All-Intervals Series*: `tunerTimeout` = 18,000 seconds

The results presented in Table 1.4 show better results in terms of test quality with respect to Table 1.2. For that reason, in the **FINAL Experiment**, only the results obtained in those tables were took into account (also because they were obtained by using longer times-out). As it can be observed in those tables, *Adaptive Search* seems to show a good behavior if the parameters **F**, **P** and **l** are in the following sets: **F** $\in$ `{ 0, 1}`, **P** $\in$ `{ 80, 90}` and **l** $\in$ `{ 0, 1}`.

In that sense, a specific configuration was extracted from the results above, and 60 runs of *Adaptive Search* were performed solving *All–Intervals* ($N = 600$) benchmark:

- 30 using the default parameter configuration (`-F 0 -P 66 -f 1 -l 1 -p 25`)
- 30 with an optimal parameter configuration extracted from the Tables 1.2, 1.4 (`-F 0 -P 80 -f 6 -l 1 -p 10`)

Table 1.5 shows results by using the default parameter settings, and Table 1.6 shows the results by using the parameter configuration found by ParamILS, and it is clear that the default configuration shows better results than *ParamILS*'s one, in terms both of runtime mean and standard deviation Using the default parameter settings, *Adaptive Search* can obtains best results int terms of *mean* and *slowest run*. However, using the ParamILS found

parameter settings, it reached a *fastest* run two times faster than the one using the default parameter settings.

37.210	411.300	112.510	171.000	73.770
327.880	214.910	124.910	482.740	530.440
212.660	99.370	287.400	533.540	**18.410**
197.290	1016.950	110.230	566.480	**1362.010**
94.860	819.700	434.460	620.600	95.920
80.580	333.370	121.590	489.700	248.370
mean: 341.005333				
spread: 310.444635				

Table 1.5: *All-Intervals Series*: Default configuration runtimes (secs)

154.460	264.530	169.840	26.990	108.790
550.210	104.900	31.100	**9.870**	1242.900
678.760	475.570	201.200	622.410	297.960
526.930	375.620	293.380	598.850	350.270
540.290	252.940	673.630	423.030	589.210
32.080	254.640	**2034.020**	571.100	207.090
mean: 422.085667				
spread: 404.618226				

Table 1.6: *All-Intervals Series*: ParamILS configuration runtimes (secs)

1.2.5.2 Tuning *Adaptive Search* for *Costas Array Problem*

Study cases:

a) The *training instances set* is composed by instances of *Costas Array* problems of order N with $9 \leq N \leq 15$

b) The *test instances set* is composed by instances of *Costas Array* problems of order N with $14 \leq N \leq 19$

c) The cutoff for each run was 60.0 seconds

d) The test quality is based on 100 runs

The **First Experiments** with this benchmark was using the following parameter domains:

- **P** `{10, 20, 30, 40, 50, 60, 70, 80, 90}`
- **F, f, l** `{0, 1, 2, 3, 4, 5, 6, 7, 8}`

- **p** `{5, 10, 20, 30, 40, 50, 60, 70}`

Table 1.7 shows results selecting directly a time-out of 5 hours (18,000 seconds). In this case the training quality of the solutions is better, but do not observe any improvement in the test quality. We can see also how *Adaptive Search* seems to be not sensitive to parameters **F** and **p**, i.e. they don't change during the tuning process. On the other hand, the tuner seems to find some optimum values for the other parameters: **P** ∈ `{ 80, 90}`, **f** ∈ `{ 4, 5}` and **l** = `2`.

In that case also, an specific configuration was extracted from the results showed in Table 1.7, and 60 runs of *Adaptive Search* were performed solving *Costas Array* ($N = 20$) benchmark:

- 30 using the default parameter configuration (`-F 0 -P 50 -f 1 -l 0 -p 5`)
- 30 with an optimal parameter configuration extracted from the Table 1.7 (`-F 3 -P 90 -f 5 -l 2 -p 30`)

Initial configuration					**Final best configuration**					**Training quality**	**Number of runs**	**Test quality**
F	P	f	l	p	F	P	f	l	p			
0	10	0	0	5	2	90	2	2	5	0.0493699	957	5.8461
0	10	0	0	5	0	90	5	2	5	0.0509388	1783	6.52742
0	10	0	0	5	0	90	5	2	5	0.049901	1759	5.21828
3	40	4	4	30	3	90	5	2	30	0.053974	856	6.3539
4	50	3	5	20	4	90	5	2	20	0.0500355	2000	5.4047
4	60	5	3	50	4	60	5	3	50	0.0520575	2000	6.09106
8	90	8	8	70	8	80	4	2	70	0.052685	550	3.85682
8	90	8	8	70	8	80	4	2	70	0.054104	536	4.17855
8	90	8	8	70	8	80	4	2	70	0.0497819	1284	3.90945
3	10	1	6	60	3	90	5	2	60	0.054934	2000	6.81675
5	70	6	1	10	5	90	4	2	10	0.0499895	2000	4.07365
1	30	5	7	5	1	90	4	2	5	0.0525747	1237	2.70091
7	80	2	0	70	7	90	5	2	70	0.0502264	212	5.2637

Table 1.7: *Costas Array*: `tunerTimeout` = 18,000 seconds

Table 1.8 shows the results by using the default parameter configuration, and Table 1.9 shows the results by using the parameter configuration found by *ParamILS*. One more time, "in the mean", the default configuration outperforms ParamILS's.

1.2.6 Conclusion

The conclusion of this study is that the tunning process by hand in this case was more effective than using ParamILS. Results show that default parameters used in the current

452.980	91.420	31.510	**827.860**	96.670
635.030	295.830	272.360	151.040	170.660
183.550	161.340	91.240	426.470	62.020
138.090	236.030	**2.850**	187.240	21.510
165.370	90.440	195.580	15.390	229.720
170.840	174.210	30.520	6.570	115.880
mean: 191.007				
spread: 185.362				

Table 1.8: Default configuration runtimes (secs)

546.260	263.230	17.200	29.220	495.940
237.340	187.760	**7.810**	43.120	94.370
59.930	128.690	247.810	265.010	231.260
209.640	465.340	21.840	8.740	**1264.610**
57.700	122.890	450.610	229.580	174.540
414.080	402.250	91.150	677.190	58.640
mean: 250.125				
spread 263.539				

Table 1.9: ParamILS configuration runtimes (secs)

Adaptive Search implementation are more effective and consistent than those found by ParamILS for both benchmarks (*All-Interval Series* and *Costas Array* problems).

2

Bibliography

[1] Daniel Diaz, Florian Richoux, Philippe Codognet, Yves Caniou, and Salvador Abreu. Constraint-Based Local Search for the Costas Array Problem. In *Learning and Intelligent Optimization*, pages 378–383. Springer, 2012.

[2] Danny Munera, Daniel Diaz, Salvador Abreu, and Philippe Codognet. A Parametric Framework for Cooperative Parallel Local Search. In *Evolutionary Computation in Combinatorial Optimisation*, volume 8600 of *LNCS*, pages 13–24. Springer, 2014.

[3] Stephan Frank, Petra Hofstedt, and Pierre R. Mai. Meta-S: A Strategy-Oriented Meta-Solver Framework. In *Florida AI Research Society (FLAIRS) Conference*, pages 177–181, 2003.

[4] Alex S Fukunaga. Automated discovery of local search heuristics for satisfiability testing. *Evolutionary computation*, 16(1):31–61, 2008.

[5] Mahuna Akplogan, Jérome Dury, Simon de Givry, Gauthier Quesnel, Alexandre Joannon, Arnauld Reynaud, Jacques Eric Bergez, and Frédérick Garcia. A Weighted CSP approach for solving spatio-temporal planning problem in farming systems. In *11th Workshop on Preferences and Soft Constraints Soft 2011.*, Perugia, Italy, 2011.

[6] Louise K. Sibbesen. *Mathematical models and heuristic solutions for container positioning problems in port terminals.* Doctor of philosophy, Technical University of Danemark, 2008.

[7] Wolfgang Espelage and Egon Wanke. The combinatorial complexity of masterkeying. *Mathematical Methods of Operations Research*, 52(2):325–348, 2000.

[8] Barbara M Smith. Modelling for Constraint Programming. *Lecture Notes for the First International Summer School on Constraint Programming*, 2005.

[9] Ignasi Abío and Peter J Stuckey. Encoding Linear Constraints into SAT. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming*, pages 75–91. Springer, 2014.

[10] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-Carlo Tree Search: A New Framework for Game AI. *AIIDE*, pages 216–217, 2008.

[11] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–49, 2012.

[12] Christian Bessiere. Constraint Propagation. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, chapter 3, pages 29–84. Elsevier, 1st edition, 2006.

[13] Daniel Chazan and Willard Miranker. Chaotic relaxation. *Linear Algebra and its Applications*, 2(2):199–222, 1969.

[14] Patrick Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundations. In *ACM Symposium on Artificial Intelligence amd Programming Languages*, volume 12, pages 1–12, Rochester, NY, 1977.

[15] Krzysztof R. Apt. From Chaotic Iteration to Constraint Propagation. In *24th International Colloquium on Automata, Languages and Programming (ICALP'97)*, pages 36–55, 1997.

[16] Éric Monfroy and Jean-Hugues Réty. Chaotic Iteration for Distributed Constraint Propagation. In *ACM symposium on Applied computing SAC '99*, pages 19–24, 1999.

[17] Éric Monfroy. A coordination-based chaotic iteration algorithm for constraint propagation. In *Proceedings of The 15th ACM Symposium on Applied Computing, SAC 2000*, pages 262–269. ACM Press, 2000.

[18] Peter Zoeteweij. Coordination-based distributed constraint solving in DICE. In *Proceedings of the 18th ACM Symposium on Applied Computing (SAC 2003)*, pages 360–366, New York, 2003. ACM Press.

[19] Farhad Arbab. Coordination of Massively Concurrent Activities. Technical report, Amsterdam, 1995.

[20] Laurent Granvilliers and Éric Monfroy. Implementing Constraint Propagation by Composition of Reductions. In *Logic Programming*, pages 300–314. Springer Berlin Heidelberg, 2001.

[21] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates. The Iterator and Composite Patterns. Well-Managed Collections. In *Head First Design Patterns*, chapter 9, pages 315–384. O'Relliy, 1st edition, 2004.

[22] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates. The Observer Pattern. Keeping your Objects in the know. In *Head First Design Patterns*, chapter 2, pages 37–78. O'Relliy, 1st edition, 2004.

[23] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates. Introduction to Design Patterns. In *Head First Design Patterns*, chapter 1, pages 1–36. O'Relliy, 1st edition, 2004.

[24] Charles Prud'homme, Xavier Lorca, Rémi Douence, and Narendra Jussien. Propagation engine prototyping with a domain specific language. *Constraints*, 19(1):57–76, sep 2013.

[25] Ian P. Gent, Chris Jefferson, and Ian Miguel. Watched Literals for Constraint Propagation in Minion. *Lecture Notes in Computer Science*, 4204:182–197, 2006.

[26] Mikael Z. Lagerkvist and Christian Schulte. Advisors for Incremental Propagation. *Lecture Notes in Computer Science*, 4741:409–422, 2007.

[27] Narendra Jussien, Hadrien Prud'homme, Charles Cambazard, Guillaume Rochart, and François Laburthe. Choco: an Open Source Java Constraint Programming Library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08),*, Paris, France, 2008.

[28] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. MiniZinc: Towards A Standard CP Modelling Language. In *Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

[29] Ibrahim H Osman and Gilbert Laporte. Metaheuristics : A bibliography. *Annals of Operations research*, 63(5):511–623, 1996.

[30] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: overview and conceptual comparison. *ACM Computing Surveys (CSUR)*, 35(3):268–308, 2003.

[31] Ilhem Boussaïd, Julien Lepagnot, and Patrick Siarry. A survey on optimization metaheuristics. *Information Sciences*, 237:82–117, jul 2013.

[32] Alexander G. Nikolaev and Sheldon H. Jacobson. Simulated Annealing. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146, chapter 1, pages 1–39. Springer, 2nd edition, 2010.

[33] Aris Anagnostopoulos, Laurent Michel, Pascal Van Hentenryck, and Yannis Vergados. A simulated annealing approach to the travelling tournament problem. *Journal of Scheduling*, 2(9):177—-193, 2006.

[34] Michel Gendreau and Jean-Yves Potvin. Tabu Search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146, chapter 2, pages 41–59. Springer, 2nd edition, 2010.

[35] Iván Dotú and Pascal Van Hentenryck. Scheduling Social Tournaments Locally. *AI Commun*, 20(3):151—-162, 2007.

[36] Christos Voudouris, Edward P.K. Tsang, and Abdullah Alsheddy. Guided Local Search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146, chapter 11, pages 321–361. Springer, 2 edition, 2010.

[37] Patrick Mills and Edward Tsang. Guided local search for solving SAT and weighted MAX-SAT problems. *Journal of Automated Reasoning*, 24(1):205–223, 2000.

[38] Pierre Hansen, Nenad Mladenovie, Jack Brimberg, and Jose A. Moreno Perez. Variable neighborhood Search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146, chapter 3, pages 61–86. Springer, 2010.

[39] Noureddine Bouhmala, Karina Hjelmervik, and Kjell Ivar Overgaard. A generalized variable neighborhood search for combinatorial optimization problems. In *The 3rd International Conference on Variable Neighborhood Search (VNS'14)*, volume 47, pages 45–52. Elsevier, 2015.

[40] Edmund K. Burke, Jingpeng Li, and Rong Qu. A hybrid model of integer programming and variable neighbourhood search for highly-constrained nurse rostering problems. *European Journal of Operational Research*, 203(2):484–493, 2010.

[41] Thomas A. Feo and Mauricio G.C. Resende. Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization*, (6):109–134, 1995.

[42] Mauricio G.C Resende. Greedy randomized adaptive search procedures. In *Encyclopedia of optimization*, pages 1460–1469. Springer, 2009.

[43] Philippe Galinier and Jin-Kao Hao. A General Approach for Constraint Solving by Local Search. *Journal of Mathematical Modelling and Algorithms*, 3(1):73–88, 2004.

[44] Philippe Codognet and Daniel Diaz. Yet Another Local Search Method for Constraint Solving. In *Stochastic Algorithms: Foundations and Applications*, pages 73–90. Springer Verlag, 2001.

[45] Yves Caniou, Philippe Codognet, Florian Richoux, Daniel Diaz, and Salvador Abreu. Large-Scale Parallelism for Constraint-Based Local Search: The Costas Array Case Study. *Constraints*, 20(1):30–56, 2014.

[46] Danny Munera, Daniel Diaz, Salvador Abreu, Francesca Rossi, and Philippe Codognet. Solving Hard Stable Matching Problems via Local Search and Cooperative Parallelization. In *29th AAAI Conference on Artificial Intelligence*, Austin, TX, 2015.

[47] Kazuo Iwama, David Manlove, Shuichi Miyazaki, and Yasufumi Morita. Stable marriage with incomplete lists and ties. In *ICALP*, volume 99, pages 443–452. Springer, 1999.

[48] David Gale and Lloyd S. Shapley. College Admissions and the Stability of Marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[49] Laurent Michel and Pascal Van Hentenryck. A constraint-based architecture for local search. *ACM SIGPLAN Notices*, 37(11):83–100, 2002.

[50] Dynamic Decision Technologies Inc. *Dynadec. Comet Tutorial.* 2010.

[51] Laurent Michel and Pascal Van Hentenryck. The comet programming language and system. In *Principles and Practice of Constraint Programming*, pages 881–881. Springer Berlin Heidelberg, 2005.

[52] Jorge Maturana, Álvaro Fialho, Frédéric Saubion, Marc Schoenauer, Frédéric Lardeux, and Michèle Sebag. Adaptive Operator Selection and Management in Evolutionary Algorithms. In *Autonomous Search*, pages 161–189. Springer Berlin Heidelberg, 2012.

[53] Colin R. Reeves. Genetic Algorithms. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146, chapter 5, pages 109–139. Springer, 2010.

[54] Marco Dorigo and Thomas Stützle. Ant colony optimization: overview and recent advances. In *Handbook of Metaheuristics*, volume 146, chapter 8, pages 227–263. Springer, 2nd edition, 2010.

[55] Konstantin Chakhlevitch and Peter Cowling. Hyperheuristics : Recent Developments. In *Adaptive and multilevel metaheuristics*, pages 3–29. Springer, 2008.

[56] Patricia Ryser-Welch and Julian F. Miller. A Review of Hyper-Heuristic Frameworks. In *Proceedings of the Evo20 Workshop, AISB*, 2014.

[57] Kevin Leyton-Brown, Eugene Nudelman, and Galen Andrew. A portfolio approach to algorithm selection. In *IJCAI*, pages 1542–1543, 2003.

[58] Horst Samulowitz, Chandra Reddy, Ashish Sabharwal, and Meinolf Sellmann. Snappy: A simple algorithm portfolio. In *Theory and Applications of Satisfiability Testing - SAT 2013*, volume 7962 LNCS, pages 422–428. Springer, 2013.

[59] Alexander E.I. Brownlee, Jerry Swan, Ender Özcan, and Andrew J. Parkes. Hyperion 2. A toolkit for {meta- , hyper-} heuristic research. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO Comp '14, pages 1133–1140, Vancouver, BC, 2014. ACM.

[60] Enrique Urra, Daniel Cabrera-Paniagua, and Claudio Cubillos. Towards an Object-Oriented Pattern Proposal for Heuristic Structures of Diverse Abstraction Levels. *XXI Jornadas Chilenas de Computación 2013*, 2013.

[61] Laura Dioşan and Mihai Oltean. Evolutionary design of Evolutionary Algorithms. *Genetic Programming and Evolvable Machines*, 10(3):263–306, 2009.

[62] John N. Hooker. Toward Unification of Exact and Heuristic Optimization Methods. *International Transactions in Operational Research*, 22(1):19–48, 2015.

[63] El-Ghazali Talbi. Combining metaheuristics with mathematical programming, constraint programming and machine learning. *4or*, 11(2):101–150, 2013.

[64] Éric Monfroy, Frédéric Saubion, and Tony Lambert. Hybrid CSP Solving. In *Frontiers of Combining Systems*, pages 138–167. Springer Berlin Heidelberg, 2005.

[65] Éric Monfroy, Frédéric Saubion, and Tony Lambert. On Hybridization of Local Search and Constraint Propagation. In *Logic Programming*, pages 299–313. Springer Berlin Heidelberg, 2004.

[66] Jerry Swan and Nathan Burles. Templar - a framework for template-method hyper-heuristics. In *Genetic Programming*, volume 9025 of *LNCS*, pages 205–216. Springer International Publishing, 2015.

[67] Sébastien Cahon, Nordine Melab, and El-Ghazali Talbi. ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics. *Journal of Heuristics*, 10(3):357–380, 2004.

[68] Youssef Hamadi, Éric Monfroy, and Frédéric Saubion. An Introduction to Autonomous Search. In *Autonomous Search*, pages 1–11. Springer Berlin Heidelberg, 2012.

[69] Roberto Amadini and Peter J Stuckey. Sequential Time Splitting and Bounds Communication for a Portfolio of Optimization Solvers. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming*, volume 1, pages 108–124. Springer, 2014.

[70] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Features for Building CSP Portfolio Solvers. *arXiv:1308.0227*, 2013.

[71] Christophe Lecoutre. XML Representation of Constraint Networks. Format XCSP 2.1. *Constraint Networks: Techniques and Algorithms*, pages 541–545, 2009.

[72] Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. *Modeling and Programming with Gecode*. 2013.

[73] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Introduction to Parallel Computing. In *Introduction to Parallel Computing*, chapter 1, pages 1–9. Addison Wesley, 2nd edition, 2003.

[74] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 746–749, New York, 2007. ACM.

[75] Mark D. Hill and Michael R. Marty. Amdahl's Law in the multicore era. *IEEE Computer*, (7):33–38, 2008.

[76] Peter Sanders. Engineering Parallel Algorithms: The Multicore Transformation. *Ubiquity*, 2014(July):1–11, 2014.

[77] Javier Diaz, Camelia Muñoz-Caro, and Alfonso Niño. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1369–1386, 2012.

[78] Joel Falcou. Parallel programming with skeletons. *Computing in Science and Engineering*, 11(3):58–63, 2009.

[79] Ian P Gent, Chris Jefferson, Ian Miguel, Neil C A Moore, Peter Nightingale, Patrick Prosser, and Chris Unsworth. A Preliminary Review of Literature on Parallel Constraint Solving. In *Proceedings PMCS 2011 Workshop on Parallel Methods for Constraint Solving*, 2011.

[80] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Embarrassingly Parallel Search. In *Principles and Practice of Constraint Programming*, pages 596–610. Springer, 2013.

[81] Akihiro Kishimoto, Alex Fukunaga, and Adi Botea. Evaluation of a simple, scalable, parallel best-first search strategy. *Artificial Intelligence*, 195:222–248, 2013.

[82] Yuu Jinnai and Alex Fukunaga. Abstract Zobrist Hashing : An Efficient Work Distribution Method for Parallel Best-First Search. *30th AAAI Conference on Artificial Intelligence (AAAI-16)*.

[83] Alejandro Arbelaez and Luis Quesada. Parallelising the k-Medoids Clustering Problem Using Space-Partitioning. In *Sixth Annual Symposium on Combinatorial Search*, pages 20–28, 2013.

[84] Hue-Ling Chen and Ye-In Chang. Neighbor-finding based on space-filling curves. *Information Systems*, 30(3):205–226, may 2005.

[85] Pavel Berkhin. Survey Of Clustering Data Mining Techniques. Technical report, Accrue Software, Inc., 2002.

[86] Farhad Arbab and Éric Monfroy. Distributed Splitting of Constraint Satisfaction Problems. In *Coordination Languages and Models*, pages 115–132. Springer, 2000.

[87] Mark D. Hill. What is Scalability? *ACM SIGARCH Computer Architecture News*, 18:18–21, 1990.

[88] Danny Munera, Daniel Diaz, and Salvador Abreu. Solving the Quadratic Assignment Problem with Cooperative Parallel Extremal Optimization. In *Evolutionary Computation in Combinatorial Optimization*, pages 251–266. Springer, 2016.

[89] Stefan Boettcher and Allon Percus. Nature's way of optimizing. *Artificial Intelligence*, 119(1):275–286, 2000.

[90] Daisuke Ishii, Kazuki Yoshizoe, and Toyotaro Suzumura. Scalable Parallel Numerical CSP Solver. In *Principles and Practice of Constraint Programming*, pages 398–406, 2014.

[91] Charlotte Truchet, Alejandro Arbelaez, Florian Richoux, and Philippe Codognet. Estimating Parallel Runtimes for Randomized Algorithms in Constraint Solving. *Journal of Heuristics*, pages 1–36, 2015.

[92] Youssef Hamadi, Said Jaddour, and Lakhdar Sais. Control-Based Clause Sharing in Parallel SAT Solving. In *Autonomous Search*, pages 245–267. Springer Berlin Heidelberg, 2012.

[93] Youssef Hamadi, Cedric Piette, Said Jabbour, and Lakhdar Sais. Deterministic Parallel DPLL system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:127–132, 2011.

[94] Andre A. Cire, Sendar Kadioglu, and Meinolf Sellmann. Parallel Restarted Search. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 842–848, 2011.

[95] Long Guo, Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Diversification and Intensification in Parallel SAT Solving. *Principles and Practice of Constraint Programming*, pages 252–265, 2010.

[96] M Yasuhara, T Miyamoto, K Mori, S Kitamura, and Y Izui. Multi-Objective Embarrassingly Parallel Search. In *IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, pages 853–857, Singapore, 2015. IEEE.

[97] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Improvement of the Embarrassingly Parallel Search for Data Centers. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming*, pages 622–635, Lyon, 2014. Springer.

[98] Prakash R. Kotecha, Mani Bhushan, and Ravindra D. Gudi. Efficient optimization strategies with constraint programming. *AIChE Journal*, 56(2):387–404, 2010.

[99] Peter Zoeteweij and Farhad Arbab. A Component-Based Parallel Constraint Solver. In *Coordination Models and Languages*, pages 307–322. Springer, 2004.

[100] Akihiro Kishimoto, Alex Fukunaga, and Adi Botea. Scalable, Parallel Best-First Search for Optimal Sequential Planning. *In ICAPS-09*, pages 201–208, 2009.

[101] Claudia Schmegner and Michael I. Baron. Principles of optimal sequential planning. *Sequential Analysis*, 23(1):11–32, 2004.

[102] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Programming Using the Message-Passing Paradigm. In *Introduction to Parallel Computing*, chapter 6, pages 233–278. Addison Wesley, second edition, 2003.

[103] Brice Pajot and Éric Monfroy. Separating Search and Strategy in Solver Cooperations. In *Perspectives of System Informatics*, pages 401–414. Springer Berlin Heidelberg, 2003.

[104] Mauro Birattari, Mark Zlochin, and Marrco Dorigo. Towards a Theory of Practice in Metaheuristics Design. A machine learning perspective. *RAIRO-Theoretical Informatics and Applications*, 40(2):353–369, 2006.

[105] Agoston E Eiben and Selmar K Smit. Evolutionary algorithm parameters and methods to tune them. In *Autonomous Search*, pages 15–36. Springer Berlin Heidelberg, 2011.

[106] Maria-Cristina Riff and Elizabeth Montero. A new algorithm for reducing metaheuristic design effort. *IEEE Congress on Evolutionary Computation*, pages 3283–3290, jun 2013.

[107] Holger H. Hoos. Automated algorithm configuration and parameter tuning. In *Autonomous Search*, pages 37–71. Springer Berlin Heidelberg, 2012.

[108] Frank Hutter, Holger H Hoos, and Kevin Leyton-brown. ParamILS: An Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

[109] Frank Hutter. Updated Quick start guide for ParamILS, version 2.3. Technical report, Department of Computer Science University of British Columbia, Vancouver, Canada, 2008.

[110] Volker Nannen and Agoston E. Eiben. Relevance Estimation and Value Calibration of Evolutionary Algorithm Parameters. *IJCAI*, 7, 2007.

[111] S. K. Smit and A. E. Eiben. Beating the 'world champion' evolutionary algorithm via REVAC tuning. *IEEE Congress on Evolutionary Computation*, pages 1–8, jul 2010.

[112] E. Yeguas, M.V. Luzón, R. Pavón, R. Laza, G. Arroyo, and F. Díaz. Automatic parameter tuning for Evolutionary Algorithms using a Bayesian Case-Based Reasoning system. *Applied Soft Computing*, 18:185–195, may 2014.

[113] Agoston E. Eiben, Robert Hinterding, and Zbigniew Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124–141, 1999.

[114] Junhong Liu and Jouni Lampinen. A Fuzzy Adaptive Differential Evolution Algorithm. *Soft Computing*, 9(6):448–462, 2005.

[115] A Kai Qin, Vicky Ling Huang, and Ponnuthurai N Suganthan. Differential evolution algorithm with strategy adaptation for global numerical optimization. *IEEE Transactions on Evolutionary Computation*, 13(2):398–417, 2009.

[116] Vicky Ling Huang, Shuguang Z Zhao, Rammohan Mallipeddi, and Ponnuthurai N Suganthan. Multi-objective optimization using self-adaptive differential evolution algorithm. *IEEE Congress on Evolutionary Computation*, pages 190–194, 2009.

[117] Martin Drozdik, Hernan Aguirre, Youhei Akimoto, and Kiyoshi Tanaka. Comparison of Parameter Control Mechanisms in Multi-objective Differential Evolution. In *Learning and Intelligent Optimization*, pages 89–103. Springer, 2015.

[118] Jeff Clune, Sherri Goings, Erik D. Goodman, and William Punch. Investigations in Meta-GAs: Panaceas or Pipe Dreams? In *GECOO'05: Proceedings of the 2005 Workshop on Genetic an Evolutionary Computation*, pages 235–241, 2005.

[119] Emmanuel Paradis. R for Beginners. Technical report, Institut des Sciences de l'Evolution, Université Montpellier II, 2005.

[120] Scott Rickard. Open Problems in Costas Arrays. In *IMA International Conference on Mathematics in Signal Processing at The Royal Agricultural College*, Cirencester, UK., 2006.

[121] Frédéric Lardeux, Éric Monfroy, Broderick Crawford, and Ricardo Soto. Set Constraint Model and Automated Encoding into SAT: Application to the Social Golfer Problem. *Annals of Operations Research*, 235(1):423–452, 2014.

[122] Konstantinos Drakakis. A review of Costas arrays. *Journal of Applied Mathematics*, 2006:32 pages, 2006.

[123] Jordan Bell and Brett Stevens. A survey of known results and research areas for n-queens. *Discrete Mathematics*, 309(1):1–31, 2009.

[124] Rok Sosic and Jun Gu. Effcient Local Search with Conflict Minimization: A Case Study of the N-Queens Problem. *IEEE Transactions on Knowledge and Data Engineering*, 6:661–668, 1994.

[125] Stephen W. Soliday, Abdollah. Homaifar, and Gary L. Lebby. Genetic algorithm approach to the search for Golomb Rulers. In *International Conference on Genetic Algorithms*, volume 1, pages 528–535, Pittsburg, 1995.

[126] Alejandro Reyes-amaro, Éric Monfroy, and Florian Richoux. POSL: A Parallel-Oriented metaheuristic-based Solver Language. In *Recent developments of metaheuristics*, to appear. Springer.

[127] Alejandro Reyes-Amaro, Éric Monfroy, and Florian Richoux. A Parallel-Oriented Language for Modeling Constraint-Based Solvers. In *Proceedings of the 11th edition of the Metaheuristics International Conference (MIC 2015)*. Springer, 2015.