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 ??

Contents

Part I

Presentation

Part II

Study and evaluation of POSL

1

Experiments design and results

In this Chapter I expose all details about the evaluation process of POSL*, i.e., all experiments I perform. For each benchmark, I explain used strategies in the evaluation process and the used environments were the runs were performed (Curiosiphi server). I describe all the experiments and I expose a complete analysis of the obtained result.*

Contents

In this chapter I illustrate and analyze the versatility of POSL studying different ways to solve constraint problems based on local search meta-heuristics. I have chosen the *Social Golfers Problem*, the *N-Queens Problem*, the *Costas Array Problem* and the *Golomb Ruler Problem* as benchmarks since they are challenging yet differently structured problems. In this Chapter I present formally each benchmark, I explain the structure of POSL's solvers that I have generated for experiments and present a detailed analysis of obtained results.

The experiments[i] were performed on an Intel® Xeon™ E5-2680 v2, 10×4 cores, 2.80GHz. This server is called *Coriosiphi* and is located at *Laboratoire d'Informatique de Nantes Atlantique*, at the University of Nantes. Showed results are the means of 30 runs for each setup, presented in columns labeled **T**, corresponding to the run-time in seconds, and **It.** corresponding to the number of iterations; and their respective standard deviations (**T(sd)** and **It.(sd)**). In some tables, the column labeled **% success** indicates the percentage of solvers finding a solution before reaching a time–out (5 minutes).

The experiments in this Chapter are multi-walk runs. Parallel experiments use 40 cores for all problem instances. It is important to point out that POSL is not designed to obtain the best results in terms of performance, but to give the possibility of rapidly prototyping and studying different cooperative or non cooperative search strategies.

All benchmarks were coded using the POSL low-level framework in C++.

First results using POSL to solve constraint problems were published in [1] were we used POSL to solve the *Social Golfers Problem* and study some communication strategies. It was the first version of POSL, therefore it was able to solve only relatively easy instances. However, the efficacy of the communication was showed using this tool.

With the next and more optimized version of POSL, I decide to start to perform more detailed studies using the benchmark mentioned before and some others.

1.1 Solving the *Social Golfers Problem*

In this section I present the performed study using *Social Golfers Problem* (*SGP*) as a benchmark. The communication strategy analyzed in here consists in applying a mechanism of cost descending acceleration, exchanging the current configuration between two solvers with different characteristics. Final obtained results show that this communication strategy works pretty well for this problem.

[i] POSL source code is available on GitHub:https://github.com/alejandro-reyesamaro/POSL

1.1.1 Problem definition

The *Social Golfers Problem* (*SGP*) consists in scheduling $g \times p$ golfers into g groups of p players every week for w weeks, such that two players play in the same group at most once. An instance of this problem can be represented by the triple $g - p - w$. This problem, and other closely related problems, arise in many practical applications such as encoding, encryption, and covering problems [2]. Its structure is very attractive, because it is very similar to other problems, like *Kirkman's Schoolgirl Problem* and the *Steiner Triple System*.

The cost function for this benchmark was implemented making an efficient use of the stored information about the cost of the previews configuration. Using integers to work with bit-flags, a table to store the information about the partners of each player in each week can be filled in $O(p^2 \cdot g \cdot w)$. So, if a configuration has $n = (p \cdot g \cdot w)$ elements, this table can be filled in $O(p \cdot n)$. This table is filled from scratch only one time in the search process (I explain in the next section why). Then, every cost of a new configuration, is calculated based on this information and the performed changes between the new configuration and the stored one. This relative cost is calculated in $O(c \cdot g)$, where c is the number of performed changed in the new configuration with respect to the stored one.

1.1.2 Experiment design and results

Here, I present the abstract solver designed for this problem as well as concrete computation modules composing the different solvers I have tested:

1. Generation abstract module I:

   I_{BP}: Returns a random configuration s, respecting the structure of the problem, *i.e.*, the configuration is a set of w permutations of the vector $[1..n]$, where $n = g \times p$.

2. Neighborhood abstract modules V:

   V_{std}: Given a configuration, returns the neighborhood $\mathcal{V}(s)$ swapping players among groups.

   V_{BAS}: Given a configuration, returns the neighborhood $\mathcal{V}(s)$ swapping the most culprit player with other players from the same week. It is based on the *Adaptive Search* algorithm.

Algorithm 1: Simple solvers for *SGP*

```
abstract solver as_simple                                  // Itr → number of iterations
computation : I, V, S, A
begin
    [↻ (Itr < K1) I ↦ [↻ (Itr % K2) [V ↦ S ↦ A] ] ]
end
solver S_std implements as_simple
    computation : I_BP, V_std, S_best, A_AI
solver S_as implements as_simple
    computation : I_BP, V_BAS, S_best, A_AI
```

$V_{BP}(p)$: Given a configuration, returns the neighborhood $V(s)$ by swapping the culprit player chosen for all p randomly selected weeks with other players in the same week

3. Selection abstract modules S:

   S_{first}: Given a neighborhood, selects the first configuration $s' \in V(s)$ improving the current cost and returns it together with the current one, into a *decision–pair*

   S_{best}: Given a neighborhood, selects the best configuration $s' \in V(s)$ improving the current cost and returns it together with the current one, into a *decision–pair*

   S_{rand}: Given a neighborhood, selects randomly a configuration $s' \in V(s)$ and returns it together with the current one, into a *decision–pair*.

4. Acceptance abstract module A:

   A_{AI}: Given a decision–pair, returns the configuration marked as *"found"*

These concrete modules are very useful and can be reused to solve tournament-like problems like *Sports Tournament Scheduling*, *Kirkman's Schoolgirl* and the *Steiner Triple System* problems.

In a first stage of the experiments I use the operator-based language provided by POSL to build and test many different non communicating strategies. The goal is to select the best concrete modules to run tests performing communication. A very first experiment was performed to select the best neighborhood function to solve the problem, comparing a basic solver using V_{std}; a new solver using V_{BAS}; and a combination of V_{std} and V_{BAS} by applying the operator (ρ), already introduced in the previous chapter. Algorithms 1 and 2 present solvers for each case, respectively.

Solver	T	T(sd)	It.	It.(sd)
S_as	**1.06**	0.79	352	268
S_rho	41.53	26.00	147	72
S_std	87.90	41.96	146	58

Table 1.1: *Social Golfers*: Instance 10–10–3 in parallel

Algorithm 2: Solvers combining neighborhood functions using operator *RHO*

abstract solver *as_rho* // ITR → number of iterations
computation : I, V_1, V_2, S, A
begin
$[\circlearrowleft (\text{ITR} < K_1)\ I\ (\mapsto)\ [\circlearrowleft (\text{ITR}\ \%\ K_2)\ [[V_1\ (\rho)\ V_2]\ (\mapsto)\ S\ (\mapsto)\ A]\ ]\ ]$
end
solver S_rho **implements** as_rho
computation : $I_{BP}, V_{std}, V_{BAS}, S_{best}, A_{AI}$

Results in Table 1.1 are not surprising. The neighborhood module V_{BAS} is based on the *Adaptive Search* algorithm, which has shown very good results [3]. It selects the most culprit variable (i.e., a player), that is, the variable to most responsible for constraints violation. Then, it permutes this variable value with the value of each other variable, in all groups and all weeks. Each permutation gives a neighbor of the current configuration. V_{Std} uses no additional information, so it performs every possible swap between two players in different groups, every week. It means that this neighborhood is $g \times p$ times bigger than the previous one, with g the number of groups and p the number of players per group. It allows for more organized search because the set of neighbors is pseudo-deterministic, i.e., the construction criteria is always the same but the order of the configuration is random. On the other hand, *Adaptive Search* neighborhood function takes random decisions more frequently, and the order of the configurations is random as well. We also tested solvers with different combinations of these modules, using the (ρ) and the $(\cup)$ operators. The (ρ) operator executes its first or second parameter depending on a given probability ρ, and the $(\cup)$ operator returns the union of its parameters output. All these combinations spent more time searching the best configuration among the neighborhood, although with a lower number of iterations than V_{BAS}. The V_{BAS} neighborhood function being clearly faster, we have chosen it for our experiments, even if it shown a more spread standard deviation: 0.75 for AS versus 0.62 for Std, considering the ratio $\frac{T(sd)}{T}$.

***

With the selected neighborhood function, I have focused the experiment on choosing the best *selection* function. Solvers mentioned above were too slow to solve instances of the problem with more than three weeks: they were very often trapped into local minima. For that

Instance	Best improvement				First improvement			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
5–3–7	0.45	0.70	406	726	0.23	0.14	142	67
8–4–7	0.37	0.11	68	13	0.28	0.07	93	13
9–4–8	0.87	0.13	95	17	0.60	0.16	139	18

Table 1.2: *Social Golfers*: comparing selection functions in parallel

reason, another solver implementing the abstract solver described in Algorithm 3 have been created, using V_{BAS} and combining S_{best} and S_{rand}: it tries a number of times to improve the cost, and if it is not possible, it picks a random neighbor for the next iteration. We also compared the S_{first} and S_{best} selection modules. The computation module S_{best} selects the best configuration inside the neighborhood. It not only spent more time searching a better configuration, but also is more sensitive to become trapped into local minima. The second computation module S_{first} selects the first configuration inside the neighborhood improving the current cost. Using this module, solvers favor exploration over intensification and of course spend clearly less time computing the neighborhood.

Algorithm 3: Solver for *SGP* to scape from local minima

```
abstract solver as_eager                          // ITR → number of iterations
computation : I, V, S1, S2, A
begin
    [↺ (ITR < K1) I ↦ [↺ (ITR % K2) [V ↦ [S1 ?_{SCI<K3} S2] ↦ A] ] ]
end
solver SOLVER_best implements as_eager
    computation : I_BP, V_std, V_BAS, S_best, S_rand, A_AI
solver SOLVER_first implements as_eager
    computation : I_BP, V_std, V_BAS, S_first, S_rand, A_AI
```

Instance	T	T(sd)	It.	It.(sd)
5–3–7	1.25	1.05	2,907	2,414
8–4–7	0.60	0.33	338	171
9–4–8	1.04	0.72	346	193

Table 1.3: *Social Golfers*: a single sequential solver using first improvement

Tables 1.2 and 1.3 present results of this experiment, showing that an local exploration-oriented strategy is better for the *SGP*. If we compare results of Tables 1.2 1.3 with respect to the standard deviation, we can some gains in robustness with parallelism. The spread in the running times and iterations for the instance 5–3–7 is 24% lower (0.84 sequentially versus 0.60 in parallel), for 8–4–7 is 30% lower (0.55 sequentially versus 0.25 in parallel) and for 9–4–8 (the hardest one) is 43% lower (0.69 sequentially versus 0.26 in parallel), using the same ratio $\frac{T(sd)}{T}$).

***

The conclusion of the last experiment was that the best solver to solve *SGP* using POSL is the one using a neighborhood computation module based on *Adaptive Search* algorithm (V_{BAS}) and a selection computation module selecting the first configuration improving the cost. Using this solver as a base, the next step was to design a simple communication strategy were the shared information is the current configuration. Algorithms 4 and 5 show that the communication is performed while applying the acceptance criterion of the new configuration for the next iteration. Here, solvers receive a configuration from a sender solver, and match it with their current configuration. Then solvers select the configuration with the lowest global cost. Different communication strategies were designed, either executing a full connected solvers set, or a tuned combination of connected and unconnected solvers. Between connected solvers, two different connections operations were applied: connecting each sender solver with one receiver solver (one to one), or connecting each sender solver with all receiver solvers (one to N). The code for the different communication strategies are presented in Algorithms 6 to 11.

Algorithm 4: Communicating abstract solver for *SGP* (sender)

```
abstract solver as_eager_sender                    // ITR → number of iterations
computation : I, V, S1, S2, A                      // SCI → number of iterations with the same cost
begin
    [↻ (ITR < K1)  I ↦ [↻ (ITR % K2) [V ↦ [S1 ?SCI<K3 S2] ↦ (|A|)^o] ] ]
end
solver SOLVER_sender implements as_eager_sender
    computation : I_BP, V_BAS, S_first, S_rand, A_AI
```

Algorithm 5: Communicating abstract solver for *SGP* (receiver)

```
abstract solver as_eager_receiver                  // ITR → number of iterations
computation : I, V, S1, S2, A                      // SCI → number of iterations with the same cost
communication : C.M.
begin
    [↻ (ITR < K1)
        I ↦ [↻ (ITR % K2)  V ↦ [S1 ?SCI<K3 S2] ↦ [A m C.M.] ]
    ]
end
solver SOLVER_receiver implements as_eager_receiver
    computation : I_BP, V_BAS, S_first, S_rand, A_AI
    communication : CM_last
```

Algorithm 6: Communication strategy one to one 100%

```
[SOLVER_sender · A] → [SOLVER_receiver · C.M.] 20;
```

Algorithm 7: Communication strategy one to N 100%

$[\textsc{Solver}_{sender} \cdot A(20)] \ \overset{\rightsquigarrow}{\bigcirc} \ [\textsc{Solver}_{receiver} \cdot C.M.(20)]$;

Algorithm 8: Communication strategy one to one 50%

$[\textsc{Solver}_{sender} \cdot A] \ \overset{\rightarrow}{\bigcirc} \ [\textsc{Solver}_{receiver} \cdot C.M.]\ 10$;
$[\textsc{Solver}_{first}]\ 20$;

In Algorithm 5, the abstract communication module $C.M.$ was instantiated with the concrete communication module CM_{last}, which takes into account the last received configuration at the time of its execution.

Each time a POSL meta-solver is launched, many independent search solvers are executed. We call "good" configuration a configuration with the lowest cost within the current configuration neighborhood and with a cost strictly lesser than the current one. Once a good configuration is found in a sender solver, it is transmitted to the receiver one. At this moment, if the information is accepted, there are some solvers searching in the same subset of the search space, and the search process becomes more exploitation–oriented. This can be problematic if this process makes solvers converging too often towards local minima. In that case, we waste more than one solver trapped into a local minima: we waste all solvers that have been attracted to this part of the search space because of communications. I avoid this phenomenon through a simple (but effective) play: if a solver is not able to find a better configuration inside the neighborhood (executing S_{first}), it selects a random one at the next iteration (executing S_{rand}).

In all Algorithms in this section, three parameter can be found: 1. K_1: the maximum number of *restarts*, 2. K_2: the maximum number of iterations in each *restart*, and K_3: the maximum number of iterations with the same current cost. 3.

After the selection of the proper modules to study different communication strategies, I proceeded to tune these parameter. Only a few runs were necessaries to conclude that the mechanism of using the computation module S_{rand} to scape from local minima was enough. For that reason, since the solver never perform restarts, the parameter K_1 was irrelevant. So the reader can assume $K_1 = 1$ for every experiment.

With the certainty that solvers do not performs restarts during the search process, I select the same value for $K_2 = 5000$ in order to be able to use the same abstract solver for all instances.

Algorithm 9: Communication strategy one to N 50%

$[\textsc{Solver}_{sender} \cdot A(10)] \ \overset{\rightsquigarrow}{\bigcirc} \ [\textsc{Solver}_{receiver} \cdot C.M.(10)]$;
$[\textsc{Solver}_{first}]\ 20$;

Algorithm 10: Communication strategy one to one 25%

$[\text{SOLVER}_{sender} \cdot A] \ (\rightarrow) \ [\text{SOLVER}_{receiver} \cdot C.M.]\,5;$
$[\text{SOLVER}_{first}]\,30;$

Algorithm 11: Communication strategy one to N 25%

$[\text{SOLVER}_{sender} \cdot A(5)] \ (\rightsquigarrow) \ [\text{SOLVER}_{receiver} \cdot C.M.(5)]\,;$
$[\text{SOLVER}_{first}]\,30;$

Finally, in the tuning process of K_3, I notice only slightly differences between using the values 5, 10, and 15. So I decided to use $K_3 = 5$.

This communication strategy produces some gain in terms of runtime (Table 1.2 with respect to Tables 1.4, 1.5 and 1.6. Having many solvers searching in different places of the search space, the probability that one of them reaches a promising place is higher. Then, when a solver finds a good configuration, it can be communicated, and receiving the help of one or more solvers in order to find the solution. Using this strategy, the spread in the running times and iterations was reduced for the instance 9–4–8 (0.22 using communication one to one and 50% of communication solvers), but not for instances 5–3–7 and 8–4–7 (0.70 using communication one to N and 50% of communicating solvers, and 0.28 using communication one to one and 50% of communicating solvers, respectively).

Other two strategies were analyzed in the resolution of this problem, with no success, both based on the sub-division of the work by weeks, i.e., solvers trying to improve a configuration only working with one or some weeks. To this end two strategies were designed:

A **Circular strategy:** K solvers try to improve a configuration during a during a number of iteration, only working on one week. When no improvement is obtained, the current configuration is communicated to the next solver (circularly), which tries to do the same working on the next week (see Figure 1.1a).

This strategy does not show better results than previews strategies. The reason is because, although the communication in POSL is asynchronous, most of the times solvers were trapped waiting for a configuration coming from its neighbor solver.

B **Dichotomy strategy:** Solvers are divided by levels. Solvers in level 1, only work

Instance	Communication 1 to 1				Communication 1 to N			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
5–3–7	0.20	0.20	165	110	0.20	0.17	144	108
8–4–7	0.27	0.09	88	28	0.24	0.05	95	12
9–4–8	0.52	0.14	117	25	0.55	0.14	126	20

Table 1.4: *Social Golfers*: 100% of communicating solvers

Instance	Communication 1 to 1				Communication 1 to N			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
5–3–7	0.18	0.13	125	88	0.17	0.12	139	81
8–4–7	0.21	0.06	89	18	0.22	0.06	90	20
9–4–8	0.49	0.11	119	24	0.51	0.15	124	21

Table 1.5: *Social Golfers*: 50% of communicating solvers

Instance	Communication 1 to 1				Communication 1 to N			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
5–3–7	0.22	0.20	181	130	0.23	0.16	143	80
8–4–7	0.24	0.08	95	22	0.29	0.09	93	12
9–4–8	0.55	0.14	134	21	0.55	0.11	130	20

Table 1.6: *Social Golfers*: 25% of communicating solvers

on one week, solvers on level 2, only work on 2 consecutive weeks, and so on, until the solver that works on all (except the first one) weeks. Solvers in level 1 improve a configuration during some number of iteration, then this configuration is sent to the corresponding solver. A solver in level 2 do the same, but working on weeks k to $k+1$. It means that it receives configurations from the solver working on week k and from the solver working on week $k+1$, and sends its configuration to the corresponding solver working on weeks k to $k+3$; and so on. The solver in the last level works on all (except the first one) weeks and receive configuration from the solver working on weeks 2 to $w/2$ and from the solver working on weeks $w/2+1$ to w (see Figure1.1b). We tested this strategy with all possible levels.

The goal of this strategy was testing if focused searches rapidly communicated can help at the beginning of the search. However, The failure of this strategy is in the fact that most of the time the sent information arrives to late to the receiver solver.

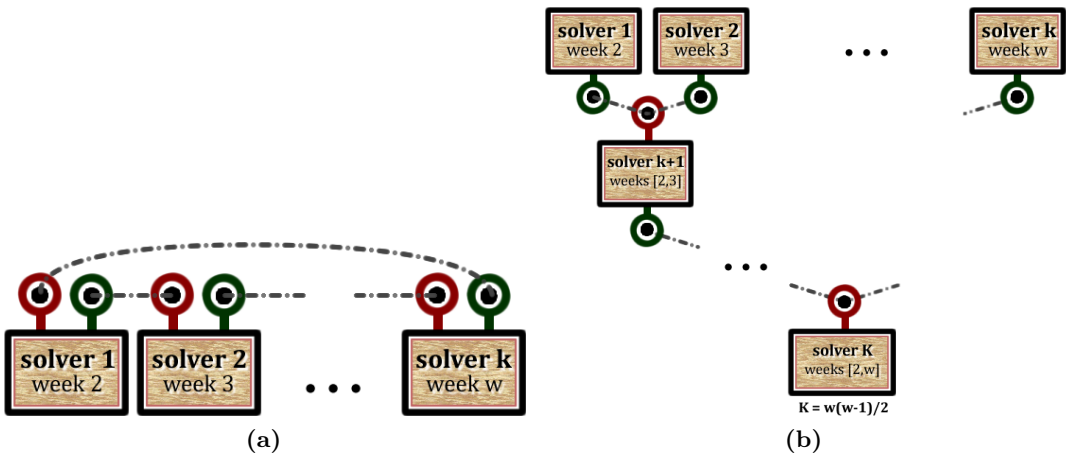

Figure 1.1: Unsuccessful communication strategies to solve *SGP*

***

One last experiment using this benchmark was a communication strategy, which applies a mechanism of cost descending acceleration, exchanging the current configuration between two solvers with different characteristics. Results show that this communication strategy works pretty well for this problem.

For this strategy, new solvers were built reusing same modules used for the communication strategies exposed before, and another different neighborhood computation module: $V_{BP}(p)$, which given a configuration, returns the neighborhood $V(s)$ by swapping the culprit player chosen for all p randomly selected weeks with other players in the same week. This new solver was called *partial solver*, and it descends quicker the cost of its current solution at the beginning because its neighborhood generates less values, but the convergence is slower and yet not sure. It was combined with the solver used for the communication strategies exposed before. It was called *full solver*, and converges in a stable way to the solution. So, the partial solver uses the same neighborhood function that the full solver, but parametrized in such a way that it builds neighbors only swapping players among two weeks.

The idea of the communication strategy is to communicate a configuration from the partial solver to the full solver. Here, the receiver solvers receive a configuration from a sender solver, and match it with their current configuration. Then it selects the configuration with the lowest global cost. This operation is coded using the *minimum* operator (m) in Algorithm 12. This way, the full solver continues the search from a more promising place into de search space. After some iterations, is the full solver who sends its configuration to the partial solver. The partial solver takes this received configuration and starts its search from it and finds quickly a much better configuration to send to the full solver again. To force the partial solver to take the received configurations over its own, we use the *not null* operator together with the communication module $C.M.$ (Algorithm 13). This process is repeated until a solution is found.

Figure 1.2 shows a full solver's run versus a partial solver's run. In this chart we can see that, at the beginning of the run, found configurations by the partial solver have costs significantly lower than those found by the full solver. At the 60-th millisecond the full solver current configuration has cost 123, and the partial solver's one, 76. So for example, the communication at this time, can accelerate the process significantly.

We also design different communication strategies, combining connected and unconnected solvers in different percentages, and applying two different communication operators: one to one and one to N.

This strategy produces some gain in terms of runtime as we can see in Tables 1.7 and 1.8 with respect to Table 1.2. It produces also more robust results in terms of runtime. The

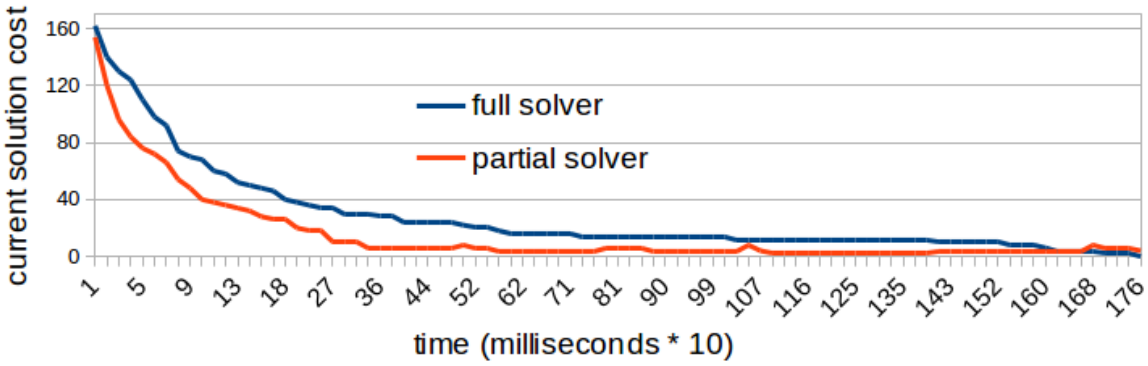

Figure 1.2: Partial solver vs. full solver (solving *Social Golfers Problem*)

Algorithm 12: Full solver for *Social Golfers Problem*

abstract solver *as_full*
computation : I, V, S_1, S_2, A
communication : $C.M.$
begin
$I \textcircled{\mapsto} [\circlearrowleft (\text{ITR} < K_1)\ V \textcircled{\mapsto} \left[S_1 \textcircled{?}_{\text{SCI}\%K_1} S_1\right] \textcircled{\mapsto} \left[C.M. \textcircled{m} (\!|A|\!)^d\right]\ ]$
end
solver SOLVER$_{full}$ **implements** as_full
computation : $I_{BP}, V_{BAS}, S_{first}, S_{rand}, A_{AI}$
communication : CM_{last}

Algorithm 13: Partial abstract solver for SGP

abstract solver *as_partial*
computation : I, V, S_1, S_2, A
communication : $C.M.$
begin
$I \textcircled{\mapsto} [\circlearrowleft (\text{ITR} < K_1)\ V \textcircled{\mapsto} \left[S_1 \textcircled{?}_{\text{SCI}\%K_1} S_2\right] \textcircled{\mapsto} \left[C.M. \textcircled{\vee} (\!|A|\!)^d\right]\ ]$
end
solver SOLVER$_{partial}$ **implements** as_partial
computation : $I_{BP}, V_{BP}(2), S_{first}, S_{rand}, A_{AI}$
communication : CM_{last}

Instance	Comm. one to N				(Comm. one to N)/2				(Comm. one to N)/4			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
5–3–7	0.14	0.08	102	53	0.14	0.07	97	73	0.12	0.08	175	162
8–4–7	0.30	0.13	101	24	0.22	0.06	92	29	0.22	0.06	88	45
9–4–8	0.55	0.15	125	20	0.53	0.14	107	20	0.40	0.14	101	70

Table 1.7: Full-partial communication strategy with communication one to N

Instance	Comm. one to one (100%)				Comm. one to one (50%)				Comm. one to one (25%)			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
5–3–7	0.10	0.05	98	75	**0.08**	0.04	139	122	0.11	0.05	190	142
8–4–7	**0.14**	0.05	100	64	0.22	0.06	119	74	0.21	0.5	101	64
9–4–8	0.37	0.14	86	65	**0.36**	0.12	144	92	0.45	0.11	150	96

Table 1.8: Full-partial communication strategy with communication one to N

spread of results in iterations show higher variances, because there are included also results of partial solvers, which performs many times more iterations that the full solvers. The percentage of the receiver solvers that were able to find the solution before the others did, was significant (see Appendix 3). That shows that the communication played an important role during the search, despite inter–process communication's overheads (reception, information interpretation, making decisions, etc).

The code for the communication strategy of 100% of communicating solvers is presented in Algorithm 14 and for 50% of communicating solvers in Algorithm 15.

Algorithm 14: Full-partial communication strategy 100% communication

$[\text{SOLVER}_{partial} \cdot A] \ (\rightarrow) \ [\text{SOLVER}_{full} \cdot C.M.]\ 20;$

$[\text{SOLVER}_{full} \cdot A] \ (\rightarrow) \ [\text{SOLVER}_{partial} \cdot C.M.]\ 20;$

Algorithm 15: Full-partial communication strategy 100% communication

$[\text{SOLVER}_{partial} \cdot A] \ (\rightarrow) \ [\text{SOLVER}_{full} \cdot C.M.]\ 10;$

$[\text{SOLVER}_{full} \cdot A] \ (\rightarrow) \ [\text{SOLVER}_{partial} \cdot C.M.]\ 10;$

$[\text{SOLVER}_{first}]\ 20;$

1.2 Solving the *N-Queens Problem*

In this section I present the performed study using *N-Queens Problem* (*NQP*) as a benchmark.

1.2.1 Problem definition

The *N-Queens Problem* (*NQP*) asks how to place N queens on a chess board so that none of them can hit any other in one move. This problem was introduced in 1848 by the chess player Max Bezzelas as the *8-queen problem*, and years latter it was generalized as *N-queen problem* by Franz Nauck. Since then many mathematicians, including Gauss, have worked on this problem. It finds a lot of applications, e.g., parallel memory storage schemes, traffic control, deadlock prevention, neural networks, constraint satisfaction problems, among others [4]. Some studies suggest that the number of solution grows exponentially with the number of queens (N), but local search methods have been shown very good results for this problem [5]. For that reason we tested some communication strategies using POSL, to solve a problem relatively easy to solve using non communication strategies.

The cost function for this benchmark was implemented in C++ based on the current implementation of *Adaptive Search*[ii].

1.2.2 Experiment design Nr. 1

To handle this problem, I reused some modules used for the *Social Golfers Problem*: the *Selection* and *Acceptance* modules. The new module is:

1. Neighborhood module:

   V_{AS}: Defines the neighborhood $V(s)$ swapping the variable which contributes the most to the cost with other.

Fors this problem I used a simple abstract solver showing good results with no communication, based on the idea introduced in the section 1.1, using the computation module S_{rand} to scape from local minima. The abstract solver is presented in Algorithm 16.

Algorithm 16: Abstract solver for *NQP*

abstract solver *as_eager* // ITR → number of iterations
computation : I, V, S_1, S_2, A // SCI → number of iterations with the same cost
begin
$I \ (\mapsto) \ [\circlearrowleft (\text{ITR} < K_1) \ V \ (\mapsto) \ [S_1 \ (?)_{\text{SCI}<K_2} \ S_2] \ (\mapsto) \ A \ ]$
end

Using solvers implementing this abstract solver we create communicating solvers to compare their performance with the non communicating strategies. The shared information is the

[ii] It is based on the code from Daniel Díaz available at https://sourceforge.net/projects/adaptivesearch/

current configuration. Algorithms 17 and 18 show that the communication is performed while selecting a new configuration for the next iteration. We design different communication strategies. Either I execute a full connected solvers set, or a tuned combination of connected and unconnected solvers. Between connected solvers, I have applied two different connections operations: connecting each sender solver with one receiver solver (*1 to 1*), or connecting each sender solver with all receiver solvers (*1 to N*).

Algorithm 17: Abstract solver for *NQP* (sender)

abstract solver *as_eager_sender* // ITR → number of iterations
computation : I, V, S_1, S_2, A // SCI → number of iterations with the same cost
begin
$I \ (\mapsto) \ [\circlearrowleft (\text{ITR} < K_1) \ V \ (\mapsto) \ \left[(\!|S_1|\!)^o \ (?)_{\text{SCI}<K_2} \ S_2 \right] \ (\mapsto) \ A \ ]$
end

Algorithm 18: Abstract solver for *NQP* (receiver)

abstract solver *as_eager_receiver* // ITR → number of iterations
computation : I, V, S_1, S_2, A // SCI → number of iterations with the same cost
communication : $C.M.$
begin
$I \ (\mapsto)$
$[\circlearrowleft (\text{ITR} < K_1)$
$V \ (\mapsto) \ \left[\left[S_1 \ (?)_{\text{ITR}\%K_2} \ \left[S_1 \ (m) \ C.M. \right] \right] \ (?)_{\text{SCI}<K_3} \ S_2 \right] \ (\mapsto) \ A$
$]$
end

1.2.3 Results analysis of experiment Nr. 1

I use directly the neighborhood module V_{AS} based on the *Adaptive Search* algorithm, and the selection module S_{First} which selects the first configuration inside the neighborhood improving the current cost, to create solvers, and studying communicating and non communicating strategies.

Instance	Sequential (1 core)				No Comm. (40 cores)			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
2000	6.20	0.12	947	21	6.15	0.20	952	20
3000	14.19	0.21	1,415	22	14.06	0.33	1,413	25
4000	25.63	0.36	1,900	28	25.46	0.51	1,898	34
5000	41.37	0.44	2,367	26	40.57	0.91	2,377	32
6000	60.42	0.52	2,837	31	60.10	0.70	2,849	43

Table 1.9: Results for *NQP* (no communication)

Instance	25% Comm.				50% Comm.				All Comm.			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
2000	6.05	0.25	934	36	6.01	0.19	920	41	5.92	0.17	885	49
3000	13.89	0.28	1,387	48	13.91	0.30	1,368	51	13.67	0.39	1,346	40
4000	25.26	0.63	1,868	43	25.14	0.50	1,855	50	25.11	0.39	1,834	58
5000	40.38	0.93	2,338	71	40.33	0.66	2,312	69	39.62	1.07	2,287	44
6000	59.28	1.34	2,794	78	58.97	1.19	2,775	67	58.97	1.38	2,729	78

Table 1.10: Results for *NQP* (40 cores, communication 1 to 1)

Instance	25% Comm.				50% Comm.				All Comm.			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
2000	6.07	0.15	925	41	5.98	0.19	915	41	6.01	0.19	887	57
3000	13.97	0.34	1,402	49	13.96	0.31	1,386	52	13.79	0.32	1,365	65
4000	25.30	0.57	1,867	52	25.29	0.42	1,851	66	25.17	0.47	1,838	65
5000	40.45	0.80	2,338	80	40.37	0.56	2,312	56	39.88	0.71	2,291	51
6000	59.77	1.50	2,824	49	59.53	0.98	2,773	69	59.16	1.37	2,773	57

Table 1.11: Results for *NQP* (40 cores, communication 1 to N)

POSL, as it can be seen in Tables 1.10 and 1.11, works very well without communication, for instances relatively big. This confirms once again the success of the computation module V_{AS} based on *Adaptive Search* algorithm to solve these kind of problems. That is the reason why the parallel approach does not outperforms significantly the sequential one, as we can see in Table 1.9. However, the communication improve the non communicating results in terms of runtime and iterations, but this improvement is not significant. In contrast to *SGP*, POSL does not get trapped so often into local minima during the resolution of *NQP*. For that reason, the shared information, once received and accepted by the receivers solvers, does not improves largely the current cost.

We can see the improvement with respect to the percentage of communicating solvers in Figure 1.3. The bigger the instance is, the more significant the observed improvement is. This phenomenon suggests that a deeper study and an efficient implementation can make the communication playing a more significant role in the solution process. For that reason, I decided to design another experiment to try to improve the results using communicating strategies using POSL.

1.2.4 Experiment design Nr. 2

The strategy of work sub-division proposed in the previews section with *Social Golfers Problem* seemed interesting to me and I did not want to give up on it. So I tried to apply it to *N-Queens Problem*.

In some experimental runs, I launched some *partial* solvers (i.e., solvers only performing permutations between variables into certain range), together with a *full* solver (i.e., a solver

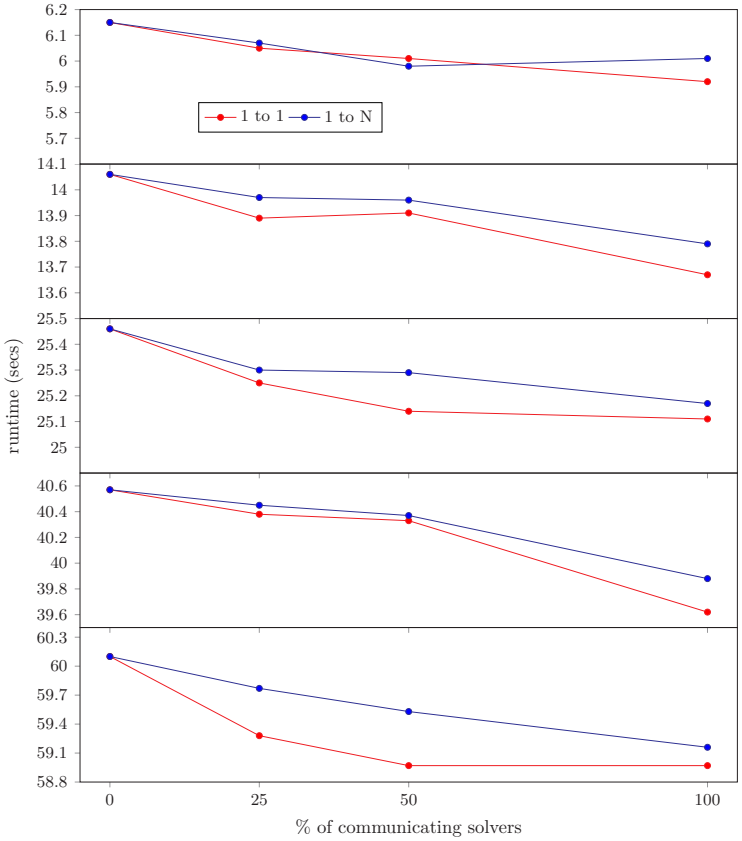

Figure 1.3: Runtime means of instances
2000-, 3000-, 4000-, 5000- and 6000-queens

working with the entire configuration). I used the instance *4000-queens* as test and I built the following solvers:

1. Solver S_1 only permuting the first 1000 variables
2. Solver S_2 only permuting the first 2000 variables
3. Solver S_3 only permuting the first 3000 variables
4. Solver S_4 a *full* solver.

Obviously, the first three solvers were not able to find a solution to the problem, but at the beginning of runs, it was possible to observe that these solvers were able to obtain configurations with costs considerably lower with respect to the *full* solver S_4. For that reason I put in practice the idea of connecting *partial* solvers together with *full* solvers. This way, the search process can be accelerated at the beginning.

Before designing the solution strategy (abstract solver) many experiments were launched to select: 1. The number of sub-divisions of the configuration, i.e., how many *partial* solvers works in different sections of the configuration. They are connected to the *full* solver. 2. The size of the section where the *partial* solvers work in.

After many runs of these experiments, it was decided to work with two *partial* sender solvers (implementing the abstract solver in Algorithm 19). In this algorithm a and b are parameters of the module $V_{[a,b]}$ used in the *partial* solvers. They represent the variables defining the range of the configuration where the *partial* solver works. They were chosen such that $b - a = \frac{n}{4}$ These solvers send their configurations to the *full* solver that implements the abstract solver in Algorithm 20.

Algorithm 19: Abstract solver for *NQP* (partial solver sender)

abstract solver *as_partial_sender* // ITR → number of iterations
computation : $I, V_{[a,b]}, S, A$ // SCI → number of iterations with the same cost
begin
$[\circlearrowleft (\text{ITR} < K_1)$
$I \ (\mapsto) \ [\circlearrowleft (\text{ITR} \ \% \ K_2 || \ \text{SCI} < K_3) \ \left[V_{[a,b]} \ (\mapsto) \ S \ (\mapsto) \ \left[(\!|A|\!)^o \ (?)_{\text{ITR}\%K_4} \ A\right]\right]]$
$]$
end

Algorithm 20: Abstract solver for *NQP* (full solver receiver)

abstract solver *as_full_receiver* // ITR → number of iterations
computation : I, V, S, A // SCI → number of iterations with the same cost
communication : $C.M.$
begin
$I \ (\mapsto)$
$[\circlearrowleft (\text{ITR} < K_1)$
$\left[V \ (\mapsto) \ S \ (\mapsto) \ \left[A \ (?)_{\text{ITR}\%K_2} \ \left[A \ (m) \ C.M.\right]\right]\right]$
$]$
end

1.2.5 Results analysis of experiment Nr. 2

Results in Table 1.12 show that this strategy is effective to solve the *N-Queens Problem* improving the runtimes already obtained in the previews experiment. In the resolution of this problem, the improvement rate of the current configuration cost is very slow (yet stable). The *partial* solvers work only on a section of the configuration, and for that reason, they are able to obtain configuration with costs considerably lower than the obtained by the *full* solver more quickly. This characteristic is taken into account: *partial* solvers send their obtained configurations to the *full* solvers. By doing this, the improvement rate of the current configuration can be accelerated at the beginning of the search.

Instance	T	T(sd)	It.	It.(sd)
2000	**5.11**	0.83	**841**	37
3000	**11.55**	1.96	**1,275**	67
4000	**21.27**	3.76	**1,656**	108
5000	**34.77**	4.99	**2,082**	108
6000	**51.72**	5.73	**2,501**	176

Table 1.12: Results for *NQP* (40 cores, communication partial-full solvers)

1.3 Solving the *Costas Array Problem*

In this section I present the performed study using *Costas Array Problem* (*CAP*) as a benchmark. This time, a simple communication strategy, in which the information to communicate between solvers is the current configuration was tested, showing good results.

1.3.1 Problem definition

The *Costas Array Problem* (*CAP*) 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. This is a very complex problem that finds useful application in some fields like sonar and radar engineering. It also presents an interesting characteristic: although the search space grows factorially, from order 17 the number of solutions drastically decreases [6].

The cost function for this benchmark was implemented in C++ based on the current implementation of *Adaptive Search*[iii].

1.3.2 Experiment design and results

To handle this problem, I have reused all modules used for solving the *N-Queens Problem*. First attempts to solve this problems were using the same strategies (abstract solvers) used to solve the *Social Golfers Problem* and *N-Queens Problem*, without success: POSL was not able to solve instances larger than $n = 8$ in a reasonable amount of time (seconds). After many unsuccessful attempts to find the rights parameter of *maximum number of restarts*, *maximum number of iterations*, and *maximum number of iterations with the same cost*, I

[iii] It is based on the code from Daniel Díaz available at https://sourceforge.net/projects/adaptivesearch/

decided to implement the mechanism used by Daniel Díaz in the current implementation of *Adaptive Search* to escape from local minima: I have added a *Reset* computation module R_{AS} based on the abstract computation module R.

The basic solver I use to solve this problem is presented in Algorithm 21, and I take it as a base to build all the different communication strategies. Basically, it is a classical local search iteration, where instead of performing restarts, it performs resets. After a deep analysis of this implementation and results of some runs, I decided to use $K_1 = 24000$ (maximum number of iterations) big enough to solve the chosen instance $n = 17$; and $K_2 = 3$ (the number of iteration before performing the next *reset*).

Algorithm 21: Reset-based abstract solver for *CAP*

```
abstract solver as_hard                                  // ITR → number of iterations
computation : I, R, V, S, A
begin
    I (↦) [↺ (ITR < K1)  R (↦) [↺ (ITR % K2) [V (↦) S (↦) A] ] ]
end
solver SOLVER_single implements as_hard
    computation : I_perm, R_AS, V_AS, S_first, A_AI
```

I present in Table 1.13 results of launching *solver sets* to solve each instance of *Costas Array Problem* 19 sequentially and in parallel without communication. Runtimes and iteration means showed in this confirm once again the success of the parallel approach.

STRATEGY	T	T(ds)	It.	It.(sd)	% success
Sequential (1 core)	132.73	80.32	2,332,088	1,424,757	40.00
Parallel (40 cores)	25.51	15.75	231,262	143,789	100.00

Table 1.13: *Costas Array* 19: no communication

***

In order to improve results, a simple communication strategy was applied: communicating the current configuration to other solvers. To do so, we insert a *sending output* operator to the

abstract solver in Algorithm 21. This results in the sender solver presented in Algorithm 22.

Algorithm 22: Sender solver for *CAP*

abstract solver *as_hard_sen*
computation *:* I, R, V, S, A
begin
$I \ (\mapsto) \ [\circlearrowleft (\text{ITR} < K_1) \ T \ (\mapsto) \ [\circlearrowleft (\text{ITR} \ \% \ K_2) \ \left[V \ (\mapsto) \ S \ (\mapsto) \ (\!|A|\!)^d\right] \ ] \ ]$
end
solver SOLVER$_{sender}$ **implements** as_hard_sen
computation : $I_{perm}, R_{AS}, V_{AS}, S_{first}, A_{AI}$

Studying some runs of POSL for solving *CAP*, it was observed that the cost of the current configuration of the first solver finding a solution. This cost describes an oscillatory descent due to the repeated resets. For that reason, it was decided to apply a simple communication strategy that shares the current configuration while applying the acceptance criterion: its goal is to accelerate the cost descent. To do so, a communication module using a *minimum* operator (m) together with the abstract computation module A was inserted, as shown in Algorithm 23.

One of the main purpose of this study is to explore different communication strategies. We have then implemented and tested different variations of the strategy exposed above by combining two communication operators (one to one and one to N) and different percentages of communicating solvers. For this problem, it was study also the behavior of the communication performed at two different moments: while applying the acceptance criteria (Algorithm 23),

	100% COMM				50% COMM			
STRATEGY	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
Str A: 1 to 1	11.60	9.17	84,159	68,958	16.78	13.43	148,222	121,688
Str A: 1 to N	**10.83**	8.72	79,551	63,785	13.03	13.46	106,826	120,894
Str B: 1 to 1	14.84	13.54	119,635	112,085	14.51	13.88	125,982	123,261
Str B: 1 to N	22.99	23.82	199,930	207,851	16.62	15.16	138,840	116,858

Table 1.14: *Costas Array* 19: with communication

and while performing a *reset* (Algorithm 24).

Algorithm 23: Receiver solver for *CAP* (variant A)

abstract solver *as_hard_receiver_a* // ITR → number of iterations
computation *: I, T, V, S, A*
communication *: C.M.*
begin
$I \ (\mapsto) \ [\circlearrowleft (\text{ITR} < K_1) \ T \ (\mapsto) \ [\circlearrowleft (\text{ITR} \ \% \ K_2) \ [V \ (\mapsto) \ S \ (\mapsto) \ [A \ (m) \ C.M.]] \ ] \ ]$
end
solver SOLVER$_{receiverA}$ **implements** as_hard_receiver_a
computation : $I_{perm}, R_{AS}, V_{AS}, S_{first}, A_{AI}$
communication: CM_{last}

Algorithm 24: Receiver solver for *CAP* (variant B)

abstract solver *as_hard_receiver_b* // ITR → number of iterations
computation *: I, R, V, S, A*
communication *: C.M.*
begin
$I \ (\mapsto) \ [\circlearrowleft (\text{ITR} < K_1) \ [R \ (m) \ C.M.] \ (\mapsto) \ [\circlearrowleft (\text{ITR} \ \% \ K_2) \ [V \ (\mapsto) \ S \ (\mapsto) \ A] \ ] \ ]$
end
solver SOLVER$_{receiverB}$ **implements** as_hard_receiver_b
computation : $I_{perm}, R_{AS}, V_{AS}, S_{first}, A_{AI}$
connection: CM_{last}

The instantiation for the receiver solvers instantiates the abstract communication module *C.M.* with the concrete communication module CM_{last}, which takes into account the last received configuration when it is running.

Table 1.14 shows that solver sets executing the strategy *A* (receiving the configuration at the time of applying the acceptance criteria) is more effective. The reason is that the others, interfere with the proper performance of the *reset*, that is a very important step in the algorithm. This step can be performed on three different ways:

1. Trying to shift left/right all sub-vectors starting or ending by the variable which contributes the most to the cost, and selecting the configuration with the lowest cost.

2. Trying to add a constant (circularly) to each element in the configuration.

3. Trying to shift left from the beginning to some culprit variable (i.e., a variable contributing to the cost).

Then, one of these 3 generated configuration has the sabe probability of being selected, to be the result of the *reset* step. In that sense, some different *resets* can be performed for the same configuration. Here is when the communication play its important role: receiver and sender solvers apply different *reset* in the same configuration, and results showed the efficacy of this communication strategy.

Analyzing the whole information obtained during the experiments, we can observe that the percentage of communicating solvers finding the solution thanks to the received information was high. That shows that the communicated information was very helpful during the search process. With the simplicity of the operator-based language provided by POSL, we were able to find a simple communication strategy to obtain better results than applying sequential and parallel independent multi-walk approaches. As expected, the best strategy was based on 100% of communication and a one to N communication, because this strategy allows to communicate a promising place inside the search space to a maximum of solvers, helping the decisive intensification process. Algorithm 25 shows the code of this communication strategy, where 20 is used as *syntactic sugar* to declare easily a list of 20 solvers of each type (20 senders and 20 receivers). Using the one to N operator $\circledcirc$ each sender solver sends information to every receiver solver.

Algorithm 25: Communication strategy one to N 100% for *CAP*

$[\textsc{Solver}_{sender} \cdot A(20)] \circledcirc [\textsc{Solver}_{receiverA} \cdot C.M.(20)]$;

Table 1.14 shows also high values of standard deviation. This is not surprising, due to the highly random nature of the neighborhood function and the selecting criterion, as well as the execution of many resets during the search process.

1.4 Solving the *Golomb Ruler Problem*

In this section the performed study using *Golomb Ruler Problem* (*GRP*) as a benchmark, is presented. Using this benchmark, a different communication strategy was tested: we communicate the current configuration in order to avoid its neighborhood, i.e., a *tabu* configuration.

1.4.1 Problem definition

The *Golomb Ruler Problem* (*GRP*) problem consists in finding an ordered vector of n distinct non-negative integers, called *marks*, $m_1 < \cdots < m_n$, such that all differences $m_i - m_j$ $(i > j)$ are all different. An instance of this problem is the pair (o, l) where o is the order of the problem, (i.e., the number of *marks*) and l is the length of the ruler (i.e., the last *mark*). We assume that the first *mark* is always 0. This problem has been applied to radio astronomy, x-ray crystallography, circuit layout and geographical mapping [7]. When I apply POSL to solve an instance of this problem sequentially, I can notice that it performs many *restarts* before finding a solution. For that reason I have chosen this problem to study a new communication strategy.

The cost function is implemented based on the storage of a counter for each measure present in the rule (configuration). I also store all distances where a variable is participating. This information is useful to compute the more culprit variable (the variable that interferes less in the represented measures), in case of the user wants to apply algorithms like *Adaptive Search*. This cost is calculated in $O\left(o^2 + l\right)$.

1.4.2 Experiment design and results

I use *Golomb Ruler Problem* instances to study a different communication strategy. This time I communicate the current configuration, to avoid its neighborhood, i.e., a *tabu* configuration. I have reused some modules used in the resolution of *Social Golfers* and *Costas Array* problems to design the solvers: the *Selection* and *Acceptance* modules. The new modules are:

1. I_{sort}: returns a random configuration s as an ordered vector of integers. The configuration is generated *far* from the set of *tabu* configurations arrived via solver-communication (in communicating strategies) (based on the *generation* abstract module I).

2. V_{sort}: given a configuration, returns the neighborhood $V\left(s\right)$ by changing one value while keeping the order, i.e., replacing the value s_i by all possible values $s'_i \in D_i$ satisfying $s_{i-1} < s'_i < s_{i+1}$ (based on the *neighborhood* abstract module V).

We also added an abstract module R for reset: it receives and returns a configuration. The concrete reset module used for this problem (R_{tabu}) inserts the input configuration into a *tabu* list inside the solver and returns the input configuration as-is. As Algorithm 27 shows, this module is executed just before performing a restart, so the solver was unable to find a

better configuration around the current one. Therefore, the current configuration is assumed to be a local minimum, and it is inserted in a tabu list.

Algorithm 26 and 27 present both solvers, using a tabu list and without using it. They were used to obtain results presented in Tables 1.15 and 1.16 to show that the approach explained above provides some gain in terms of runtime.

Algorithm 26: Solver without using tabu list, for *GRP*

```
abstract solver as_golomb_notabu                    // Itr → number of iterations
computation : I, V, S, A
begin
    [↺ (Itr < K1)  I (↦)  [↺ (Itr % K2) [V (↦) S (↦) A] ] ]
end
solver Solver_notabu implements as_notabu
    computation : I_sort, V_sort, S_first, A_AI
```

Algorithm 27: Solver using tabu list, for *GRP*

```
abstract solver as_golomb_tabu                      // Itr → number of iterations
computation : I, V, S, A
begin
    [↺ (Itr < K1)  I (↦)  [↺ (Itr % K2) [V (↦) S (↦) A] ] (↦) (|T|)^o ]
end
solver Solver_tabu implements as_tabu
    computation : I_sort, V_sort, S_first, A_AI
```

Instance	T	T(sd)	It.	It.(sd)	R	R(sd)	% success
8–34	0.79	0.66	13,306	11,154	66	55.74	100.00
10–55	66.44	49.56	451,419	336,858	301	224.56	80.00
11–72	160.34	96.11	431,623	272,910	143	90.91	26.67

Table 1.15: A single sequential solver without using tabu list for *GRP*

Instance	T	T(sd)	It.	It.(sd)	R	R(sd)	% success
8–34	0.66	0.63	10,745	10,259	53	51.35	100.00
10–55	67.89	50.02	446,913	328,912	297	219.30	88.00
11–72	117.49	85.62	382,617	275,747	127	91.85	30.00

Table 1.16: A single sequential solver using tabu list for *GRP*

The benefit of the parallel approach is also proved for the *Golomb Ruler Problem*, as we can see in Table 1.17. However, without communication, the improvement is not substantial (8% for 8–34, 7% for 10–55 and 5% for 11–72). The reason is because only one configuration is inserted in the tabu list after each restart. When we use one to one communication, after the restart k, the receiving solver has twice the number of configurations in the tabu list (one tabu configuration from itself and the received one after each restart).

Instance	T	T(sd)	It.	It.(sd)	R	R(sd)
8–34	0.43	0.37	349	334	1	1.64
10–55	4.92	4.68	20,504	19,742	13	13.07
11–72	85.02	67.22	155,251	121,928	51	40.64

Table 1.17: Parallel solvers using tabu list for *GRP*

***

The main goal of choosing this benchmark was to study a different communication strategy, since for solving this problem, POSL needs to perform some restarts. In this communication strategy, solvers do not communicate the current configuration to have more solvers searching in its neighborhood, but a configuration that we assume is a local minimum to be avoided. We consider that the current configuration is a local minimum since the solver (after a given number of iteration) is not able to find a better configuration in its neighborhood, so it will communicate this configuration just before performing the restart.

Algorithm 28 presents the sender solver and Algorithm 29 presents the receiver solver. Based on the connection operator used in the communication strategy, this solver might receives one or many configurations. These configurations are the input of the generation module (I), and this module inserts all received configurations into a *tabu* list, and then generates a new first configuration, far from all configurations in the *tabu* list.

Algorithm 28: Sender solver for *GRP*

abstract solver *as_golomb_sender* // ITR → number of iterations
computation : I, V, S, A, R
begin
$[\circlearrowleft (\text{ITR} < K_1)\ \ I \ (\mapsto)\ \ [\circlearrowleft (\text{ITR}\ \%\ K_2)\ \ [V \ (\mapsto)\ S \ (\mapsto)\ A]\ ]\ \ (\mapsto)\ (\!|T|\!)^o\ ]$
end
solver SOLVER$_{sender}$ **implements** as_golomb_sender
computation : $I_{sort}, V_{sort}, S_{first}, A_{AI}, R_{tabu}$

Algorithm 29: Receiver solver for *GRP*

abstract solver *as_golomb_receiver* // ITR → number of iterations
computation : I, V, S, A, R
connection : $C.M.$
begin
$[\circlearrowleft (\text{ITR} < K_1)\ \ [C.M. \ (\mapsto)\ I]\ \ (\mapsto)\ \ [\circlearrowleft (\text{ITR}\ \%\ K_2)\ \ [V \ (\mapsto)\ S \ (\mapsto)\ A]\ ]\ \ (\mapsto)\ (\!|T|\!)^o\ ]$
end
solver SOLVER$_{receiver}$ **implements** as_golomb_receiver
computation : $I_{sort}, V_{sort}, S_{first}, A_{AI}, R_{tabu}$
communication: CM_{last}

In this communication strategy there are some parameters to be tuned. The first ones are: 1. K_1, the number of restarts, and 2. K_2, the number of iterations by restart. Both are

instance dependent, so, after many experimental runs, I choose them as follows:

- *Golomb Ruler* 8–34: $K_1 = 300$ and $K_2 = 200$
- *Golomb Ruler* 10–55: $K_1 = 1000$ and $K_2 = 1500$
- *Golomb Ruler* 11-72: $K_1 = 1000$ and $K_2 = 3000$

The idea of this strategy (abstract solver) follows the following steps:

Step 1

The computation module generates an initial configuration tacking into account a set of configurations into a $tabu\ list$. The configuration arriving to this $tabu\ list$ come from the same solver (Step 3) or from outside (other solvers) depending on the strategy (non-communicating or communicating).

This module applies some other modules provided by POSL to solve the $Sub\text{-}Sum\ Problem$ in order to generates $valid$ configurations for $Golomb\ Ruler\ Problem$. A valid configuration s for $Golomb\ Ruler\ Problem$ is a configuration that fulfills the following constraints:

- $s = (a_1, \ldots, a_o)$ where $a_i < a_j, \forall i < j$, and
- all $d_i = a_{i+1} - a_i$ are all different, for all $d_i, i \in [1...o-1]$

The $Sub\text{-}sum\ Problem$ is defined as follows: Given a set E of integers, with $|E| = N$, finding a sub set e of n elements that sums exactly z. In that sense, a solution $S_{sub-sum} = \{s_1, \ldots, s_{o-1}\}$ of the $Sub\text{-}sum\ problem$ with $E = \left\{1, \ldots, l - \frac{(o-2)(o-1)}{2}\right\}$, $n = o - 1$ and $z = l$, can be traduced to a $valid$ configuration C_{grp} for $Golomb\ Ruler\ Problem$ as follows:

$$C_{grp} = \{c_1, c_1 + s_1, \ldots, c_{o-1} + s_{o-1}\}$$

where $c_1 = 0$.

In the selection module applied inside the module I, the selection step of the search process selects a configuration from the neighborhood far from the $tabu$ configurations, with respect to certain vectorial norm and an epsilon. In other words, a configuration C is selected if and only if:

1. the cost of the configuration C is lower than the current cost, and
2. $||C - C_t||_p > \varepsilon$, for all $tabu$ configuration C_t

where p and ε are parameters.

I experimented with 3 different values for p. Each value defines a different type of norm of a vector $x = \{x_1, \ldots, x_n\}$:

- $p = 1$: $||x||_1 = \sum_{i=0}^{n} |x_i|$
- $p = 2$: $||x||_2 = \sqrt{\sum_{i=0}^{n} |x_i|^2}$

- $p = \infty$: $||x||_{\infty} = \max(x)$

After many experimental runs with these values I choose $p = \infty$ and $\varepsilon = 4$ for the communication strategy study. I also made experiments trying to find the right size for the $tabu$ list and the conclusion was that the right sizes were 15 for non-communicating strategies and 40 for communicating strategies, taking into account that in the latter, I work with 20 receivers solvers.

Step 2

After generating the first configuration, the next step is to apply a local search to improve it. In this step I use the neighborhood computation module V, that creates neighborhood $\mathcal{V}(s)$ by changing one value while keeping the order in the configuration, and the other modules (selection and acceptance). The local search is executed a number K_2 of times, or until a solution is obtained.

Step 3

If no improvement is reached, the current configuration is classified as a *potential local minimum* and inserted into the $tabu$ list. Then, the process returns to the Step 1.

The POSL code of the communication strategy using the one to N operator is the following:

$$[S_{sender} \cdot T(20)] \ (\rightsquigarrow) \ [S_{receiver} \cdot C.M.(20)] \tag{1.1}$$

When we use communication one to one, after k restarts the receiver solver has $2k$ configurations in its tabu list: its own tabu configurations and the received ones. Table 1.18 shows that this strategy is not sufficient for some instances, but when we use communication one to N, the number of tabu configurations after k restarts in the receiver solver is considerably higher: $k(N+1)$: its own tabu configurations and the ones received from N sender solvers the receiver solver is connected with. Hence, these solvers can generate configurations far enough from many potentially local minima. This phenomenon is more visible when the problem order o increases. Table 1.19 shows that the improvement for the higher case (11-72) is about 29% w.r.t non communicating solvers (Table 1.17).

Instance	T	T(sd)	It.	It.(sd)	R	R(sd)
8–34	0.44	0.31	309	233	1	1.23
10–55	3.90	3.22	15,437	12,788	10	8.52
11–72	85.43	52.60	156,211	97,329	52	32.43

Table 1.18: *Golomb Ruler*: parallel, communication one to one

Instance	T	T(sd)	It.	It.(sd)	R	R(sd)
8–34	**0.43**	0.29	283	225	1	1.03
10–55	**3.16**	2.82	12,605	11,405	8	7.61
11–72	**60.35**	43.95	110,311	81,295	36	27.06

Table 1.19: *Golomb Ruler*: parallel, communication one to N

1.5 Summarizing

In this Chapter I have chosen various *Constraint Satisfaction Problems* as benchmarks to 1. evaluate the POSL behavior solving these kind of problems, and 2. to study different solution strategies, specially communication strategies.To this end, I have chosen benchmarks with different characteristics, to help me having a wide view of the POSL behavior.

In the solution process of *Social Golfers Problem*, it was that an exploitation-oriented communication strategy, in which the current configuration is communicated to ask other solvers for help to concentrate the effort in a more promising area, can provide some gain in terms of runtime. It was also presented results showing the success of a cost descending acceleration communication strategy, exchanging the current configuration between two solvers with different characteristics. Some other unsuccessful strategies were studied, showing that the sub-division of the effort by weeks, do not work well. Table 1.20 summarizes the obtained results solving *SGP*.

It was showed that simple communication strategies as they were applied to solve *Social Golfers Problem* does not improve enough the results without communication for the *N-Queens Problem*. However, a deep study of the POSL's behavior during the search process allows to design a communication strategy able to improve the results obtained using non-communicating strategies.

The *Costas Array Problem* is a very complicated constrained problem, and very sensitive to the methods to solve it. For that reason I used some ideas from already existent algorithms. However, thanks to some studies of different communication strategies, based on the configuration of the current communication at different times (places) in the algorithm, it was possible to find a communication strategy to improve the performance. Table 1.21 summarizes the obtained results solving *CAP*.

Instance	Sequential		Parallel		Cooperation	
	T	It.	T	It.	T	It.
5–3–7	1.25	2,907	0.23	142	**0.08**	139
8–4–7	0.60	338	0.28	93	**0.14**	100
9–4–8	1.04	346	0.60	139	**0.36**	144

Table 1.20: Summarizing results for *SGP*

STRATEGY	T	It.	% success
Sequential	132.73	2,332,088	40.00
Parallel	25.51	231,262	100.00
Cooperative Strategy	**10.83**	79,551	100.00

Table 1.21: Summarizing results for *CAP* 19

During the solution process of the *Golomb Ruler Problem*, POSL needs to perform many restarts. For that reason, this problem was chosen to study a different (and innovative up tu my knowledge) communication strategy, in which the communicated information is a potential local minimum to be avoided. This new communication strategy showed to be effective to solve these kind of problems. Table 1.22 summarizes the obtained results solving *GRP*.

In all cases, thanks to the operator-based language provided by POSL it was possible to test many different strategies (communicating and non-communicating) fast and easily. Whereas creating solvers implementing different solution strategies can be complex and tedious, POSL gives the possibility to make communicating and non-communicating solver prototypes and to evaluate them with few efforts. In this Chapter it was possible to show that a good selection and management of inter-solvers communication can largely help to the search process, working with complex constrained problems.

Instance	Sequential				Parallel			Cooperation		
	T	It.	R	% success	T	It.	R	T	It.	R
8–34	0.66	10,745	53	100.00	0.43	349	1	**0.43**	283	1
10–55	67.89	446,913	297	88.00	4.92	20,504	13	**3.16**	12,605	8
11–72	117.49	382,617	127	30.00	85.02	155,251	51	**60.35**	110,311	36

Table 1.22: Summarizing results for *GRP*

2

Bibliography

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

[2] 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.

[3] 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.

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

[5] 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.

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

[7] 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.

Part III

Appendix

3

Results of experiments with *Social Golfers Problem*

This Appendix, presents graphically a summary of individuals runs using Social Golfers Problem. Figures show a box-plot representation, indicating also, for each experiment, the percentage of communicating solvers finding the solution thanks to received information.

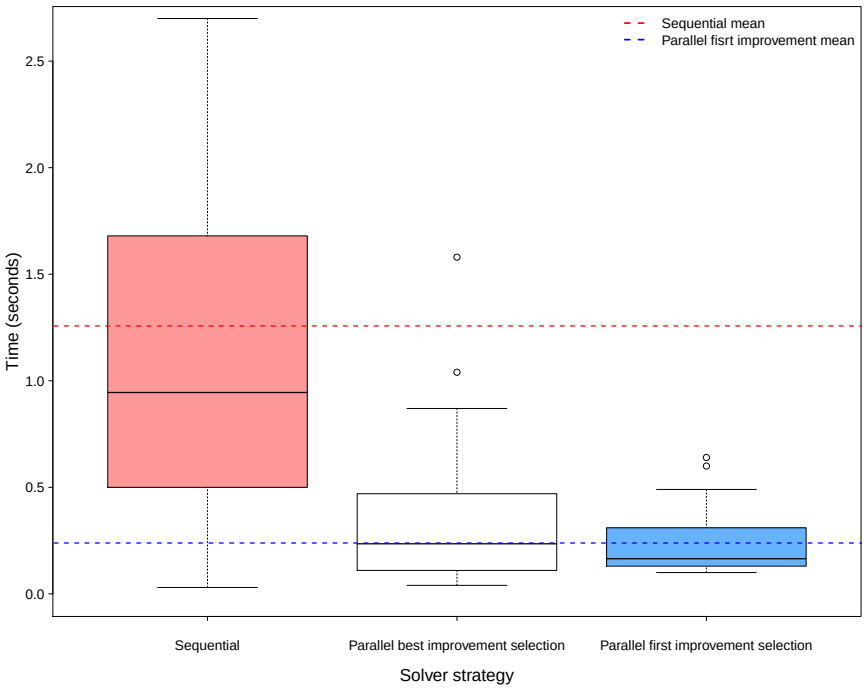

Figure 3.1: Comparison between sequential and parallel (best improvement and first improvement selections) runs to solve *SGP* 5-3-7 using POSL

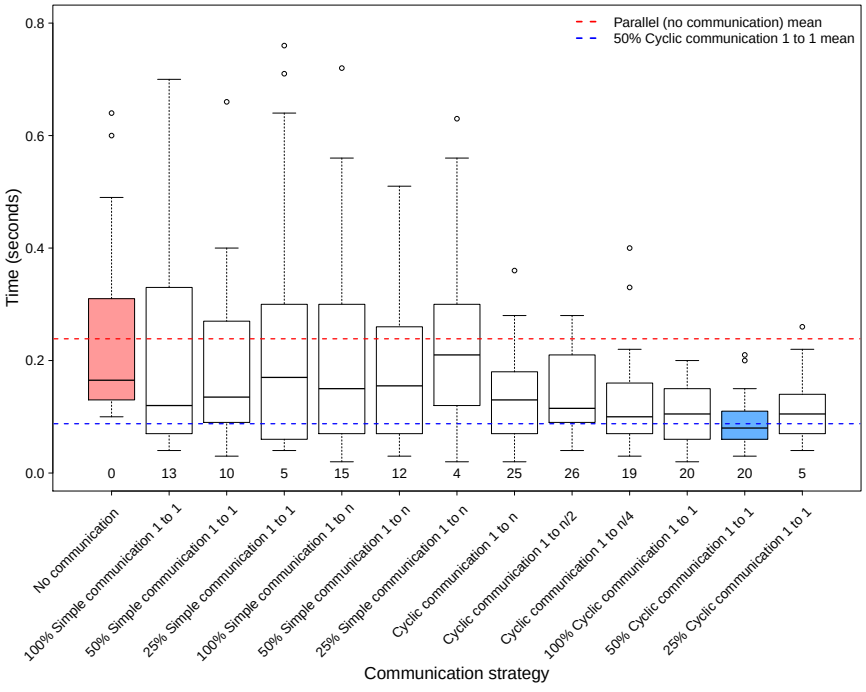

Figure 3.2: Different communication strategies to solve *SGP* 5-3-7 using POSL

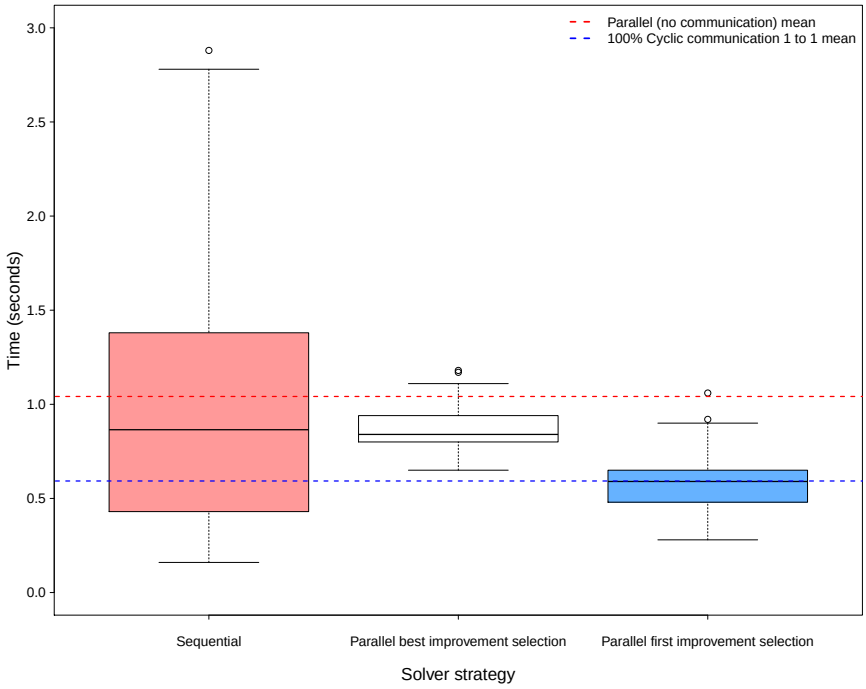

Figure 3.3: Comparison between sequential and parallel (best improvement and first improvement selections) runs to solve *SGP* 9-4-8 using POSL

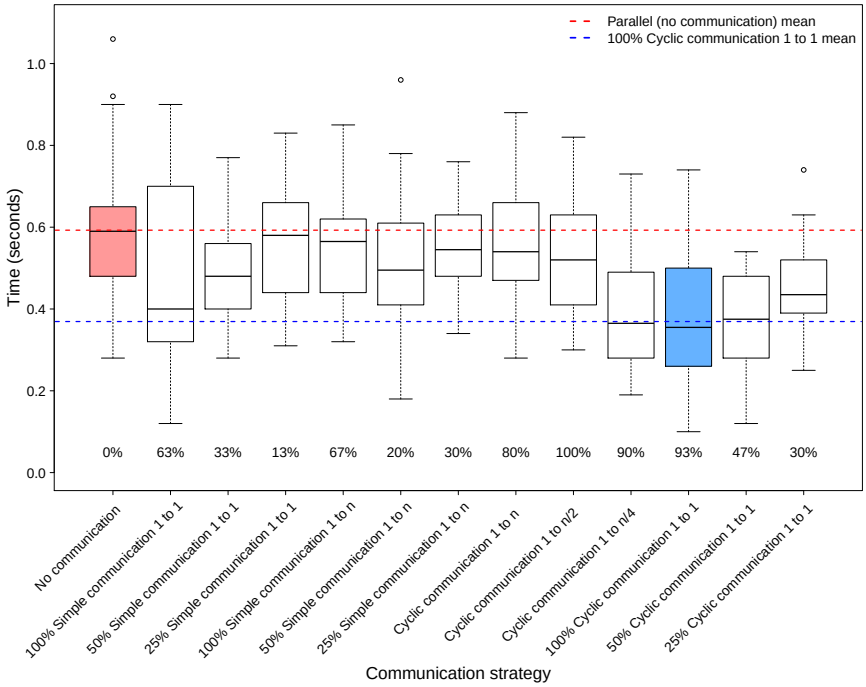

Figure 3.4: Different communication strategies to solve *SGP* 9-4-8 using POSL