2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

Dependent Walks in Parallel Local Search

Yves Caniou
JFLI, CNRS / Université de Lyon / University of Tokyo
Tokyo, Japan
Yves.Caniou@ens-lyon.fr

Philippe Codognet
JFLI, CNRS / UPMC / University of Tokyo
Tokyo, Japan
codognet@is.s.u-tokyo.ac.jp

***Abstract*—Following earlier work on *independent* multi-walk parallel local search, we present in this paper a framework for *dependent* multi-walk and its implementation. The new framework provides the possibility to communicate configurations between concurrent local search engines in order to better focus the overall search on promising configurations. An MPI-based implementation has been realized and its evaluation on various benchmarks is ongoing.**

***Keywords*-local search; metaheuristics; parallelism; communication; MPI; constraint solving;**

I. Introduction

In the last decade, the interest for the family of Local Search methods and Metaheuristics has been developing and has attracted much attention from both the Artificial Intelligence and the Operations Research communities for solving large combinatorial problems. Detailed descriptions of this very large family of methods can be found for instance in [1], [2]. Stochastic Local Search methods usually start from one random configuration (or a set of random configurations). They try to iteratively improve this configuration, little by little through small changes in the values of a few variables, hence the term "local search" as only new configurations that are neighbors of the current configuration are explored at each iteration. Simulated Annealing, Genetic Algorithms, Tabu Search, neighboring search, Swarm Optimization, Ant-Colony optimization, *etc.*, are all different kinds of metaheuristics based on random choices that can be applied to different sets of problems such as resource allocation, scheduling, packing, layout design, frequency allocation, in order to produce task-specific local search algorithms.

One way to improve the performance of Local Search Methods is to take advantage of the increasing availability of parallel computational resources. Parallel implementation of local search meta-heuristics have been studied since the early 90's, when multiprocessor machines started to become widely available, see [3]. One usually distinguishes between single-walk and multiple-walk methods. Single-walk methods consist in using parallelism inside a single search process, *e.g.,* , for parallelizing the exploration of the neighborhood, while multiple-walk methods (also called multi-start methods) consist in developing concurrent explorations of the search space, either independently or cooperatively with some communication between concurrent processes. A key point is that independent multiple-walk methods are the easiest to implement on parallel computers and can in theory lead to linear speed-up [3].

We already experimented with multi-walk local search in a massively parallel context up to a few hundreds and a few thousands of cores, see for instance [4], and achieved good parallel speedups but not linear, except for one complex problem [5]. We studied the use of local search methods to solve Constraint Satisfaction Problems (CSP), a powerful declarative programming paradigm which has been successfully used to tackle several complex problems, among which many combinatorial optimization ones. Using Local Search methods for solving problems formulated as a Constraint Satisfaction Problem amounts to the methods collectively designated as Constraint-Based Local Search, see for example [6] for a general introduction. On structured constraint-based problems such as (large instances of) Magic Square or All-Interval, independent multiple-walk parallelization does not yield linear speedups, reaching for instance a speedup factor of "only" 50-70 for 256 cores. However on the Costas Array Problem, the speedup can be linear, even up to 8000 cores [5]. On a more theoretical level, it can be shown that the parallel behavior depends on the *sequential runtime distribution* of the problem: for problems admitting an exponential distribution, the speedup can be linear, while if the runtime distribution is shifted-exponential or (shifted) lognormal, then there is a bound on the speedup (which will be the asymptotic limit when the number of cores goes to infinity), see [7] for a detailed analysis of these phenomena.

It is clear however that the classical multi-walk approach will not scale up to millions of cores, and thus a new paradigm has to be defined. In order to improve the independent multi-walk approach, a new paradigm that includes *cooperation* between walks has to be defined. Indeed, *Cooperative Search* methods add a communication mechanism to the independent walk strategy in order to exchange information between solver instances during the search process. However, developing an efficient cooperative method is a very complex task, see for instance the early experiments of [8], and many issues must solved.

This paper is organized as follows. Section 2 presents the general framework of communicating (sequential) local search engines and the notion of Elite Pools. Section 3 details a simplified implementation of the general framework based on the message-passing paradigm (MPI) and using adaptive search as basic search engine. Section 4 introduces different control parameters that can tune the ratio between communication between solvers and independent search computation. A brief conclusion ends the paper.

II. General Framework

In the general framework of parallel local search, we can consider a collection of search agents representing individual sequential local search solvers. The individual solvers can (and probably should) be heterogeneous: simulated annealing, tabu search, genetic algorithms, particle swarm optimization could be instances of such solvers. Moreover, several instances of the same metaheuristic method, with different parameter tunings, can (and probably should) co-exist in this parallel ecosystem. All these methods are instances of so-called "iterative improvement" methods: they start from one (or several) random configuration, that is, assignment of values to the problem variables, and try to assign iteratively better values in order to optimize some given *objective* (or *fitness*) function. Interestingly, all methods share the same basic object, the vector of values representing the configuration, even if they widely differ in the way to move from one configuration to another and to evaluate the fitness/objective function.

This means that the configuration currently examined by one solver can be a good information to exchange with another solver, if this configuration has some good properties, *e.g.,* it has a good value for the objective function. Then other solvers running concurrently can examine and develop this promising configuration with their own different heuristics and their own stochastic aspects.

We can thus define a general framework where search agents would share their best configurations in so-called *Elite Pools* and could in this way exchange information. Search agents do not communicate through posting/getting configurations to/from the Elite Pool and developing them (sequentially) for a given number of iterations. The management of an Elite Pool Manager can be seen as a complex adaptive system and a key issue is the control strategy which should enforce various issues such as:

- pool trimming, *i.e.,* which configurations to keep and to eliminate (keeping the best ones is obvious, but a certain diversity has also to be preserved),
- dynamic generation of configuration, *i.e.,* the creation of new configurations from the ones currently in the pool (in a genetic algorithm fashion, path-relinking manner, or any other policy),
- load balancing, *i.e.,* how to allocate computing power to agents depending on their performance,
- agent monitoring, *i.e.,* when to stop an agent (*e.g.,* because it cannot progress in the search) and replace it by a fresh new one.

Of course as we consider massively parallel machines with no or only very limited shared-memory (*e.g.,* within nodes composed of up to a few tens of cores only), we could not consider a single Elite Pool, but several distributed ones, communicating by exchanging some parts of their respective pools (best configurations), under certain policies reifying distributed decision making. Figure 1 depicts a simple example of this architecture.

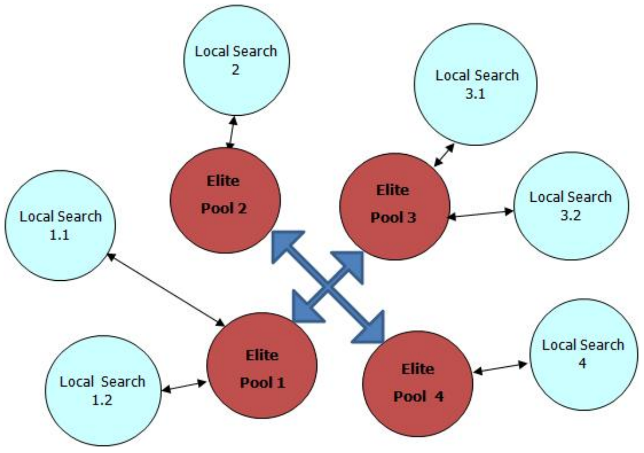

Figure 1. Elite Pools in the General Framework

As these control policies seem quite complex to define and depend on the problem to be solved, we consider that they can be based on a set of parameters defined for the management of each Elite Pool, and thus the global control strategy should be an emergent property evolved during the computation. Such parameters can be :

- the size of the Elite Pool,
- the number of iterations during which a solver engine works sequentially before communicating a new configuration and sharing it with other solver engines,
- a probabilistic criterion (*i.e.,* a probability rule) for an Elite Pool to accept, store and communicate a configuration received by a solver engine,
- a restart policy, that is, under certain conditions the use of a configuration in the Elite Pool instead of a random configuration when a restart is requested by the sequential solver engine.

III. Simplified Framework

Although this general framework presented in the above section is aimed at working with several different local search methods for each search engine, we decided to first design a simpler version where each solver belongs to the same family of local search method.

A. Constraint-based Local Search

In the continuation of the experiments presented in [4], [9], [5], we used for each basic (sequential) local search engine the Adaptive Search method, a generic, domain-independent constraint-based local search method proposed by [10], [11]. This metaheuristic takes advantage of the structure of the problem in terms of constraints and variables and can guide the search more precisely than a single global cost function to optimize, such as for instance the number of violated constraints. The algorithm also uses a short-term adaptive memory in the spirit of Tabu Search in order to prevent stagnation in local minima and loops.

An implementation of Adaptive Search (AS) has been developed in C language as a framework library and is available as a freeware at the URL:
http://cri-dist.univ-paris1.fr/diaz/adaptive/

We used this reference implementation for our experiments. The Adaptive Search method can be applied to a large class of constraints, *e.g.,* linear and non-linear arithmetic constraints, symbolic constraints, *etc.* and naturally copes with over-constrained problems. The input of the method is a Constraint Satisfaction Problem (CSP for short), which is defined as a triple (X;D;C), where X is a set of variables, D is a set of domains, *i.e.,* finite sets of possible values (one domain for each variable), and C a set of constraints restricting the values that the variables can simultaneously take.

Note that the Adaptive Search method is tackling constraint *satisfaction* problems as optimization problems, that is, it aims to minimize the global error (representing the violation of constraints) to value zero. Therefore finding a solution means that the execution actually reaches the bound (zero) of the objective function to minimize.

For each constraint, an *error function* needs to be defined; it gives, for each tuple of variable values, an indication of how much the constraint is violated. For example, the error function associated with an arithmetic constraint $|X - Y| < c$, for a given constant $c \geq 0$, can be $max(0, |X - Y| - c)$.

Adaptive Search relies on iterative repair, based on variable and constraint error information, seeking to reduce the error on the worst variable so far. The basic idea is to compute the error function for each constraint, then combine for each variable the errors of all constraints in which it appears, thereby projecting constraint errors onto the relevant variables. This combination of errors is problem-dependent, see [10] for details and examples, but it is usually a simple sum or a sum of absolute values, although it might also be a weighted sum if constraints are given different priorities. Finally, the variable with the highest error is designated as the "culprit" and its value is modified. In this second step, the well-known *min-conflict* heuristic is used to select the value in the variable domain which is the most promising, that is, the value for which the total error in the next configuration is minimal.

In order to prevent being trapped in local minima, the Adaptive Search method also includes a short-term memory mechanism to store configurations to avoid (variables can be marked Tabu and "frozen" for a number of iterations). It also integrates reset transitions to escape stagnation around local minima. A reset consists in assigning fresh random values to some variables (also randomly chosen). A reset is guided by the number of variables being marked Tabu. It is also possible to restart from scratch when the number of iterations becomes too large (this can be viewed as a reset of all variables but it is guided by the number of iterations).

The core ideas of adaptive search can be summarized as follow:

- to consider for each constraint a heuristic function that is able to compute an approximated degree of satisfaction of the goals (the current *error* on the constraint);
- to aggregate constraints on each variable and project the error on variables thus trying to repair the *worst* variable with the most promising value;
- to keep a short-term memory of bad configurations to avoid looping, *i.e.,* some sort of *tabu list*, together with a reset mechanism.

B. Communication between Sequential Solvers

Figure 1 presents a distributed framework whose communication layer can be instantiated with various paradigms such as GridRPC, peer-to-peer, *etc.*. For the sake of simplicity, we focus here on an implementation where there is a unique correspondance between an Elite Pool and a Search Engine, which can easily be deployed on testbeds composed of a cluster of workstations or a supercomputer. It is possible to parallelize and integrate a communication layer with a minimum impact on the Adaptive Search using the message passing interface in its core: Algorithm 1 presents how the code of Adaptive Search has been modified to its minimum in order to handle MPI communications.

Basically the MPI initialization takes care of:

1) distributing the correct seed to each search engine (MPI instantiation): We use a chaotic randomization based on runtime values obtained from the system and/or data given by the user.
2) memory allocation: Here takes place the pre-allocation of messages dispatched in lists that will be used to handle MPI buffers to send and receive configuration, and as non-blocking communications are invoved, wait for messages communications to complete.
3) communication initialization: We initiate a non-blocking receive to avoid some additional recopies of MPI internal buffers to user-space buffers.

Algorithm 2 is the Elite Pool manager algorithm. It highlights how MPI communications are managed once every C-block (a block of C iterations): this leads to control

Algorithm 1: Parallel Adaptive Search

```
Data: random seed or config.
MPI initialization;
Seq initialization;
repeat
    Manage MPI communications;
    Compute the errors on constraints;
    Combine the errors on each variable;
    Select the worst variable X (except if "tabu");
    Compute cost of config. with other values of X;
    if a better config. is found then
        Move to this config.;
    else Mark X tabu and move randomly;
    ;
until a sol. is found OR a max. nb. of iter. is reached;
Terminate MPI application and print outputs;
```

the granularity of the search, *i.e.,* the ratio of computations to be performed while messages can be sent between search engines. Indeed depending on the problem, a configuration can take a few bytes only (more or less the number of variables for the COSTAS ARRAY problem for example) to several kilobytes (the number of variables square for the MAGIC-SQUARE problem, for example 156KB for $n = 200$).

Algorithm 2: Managing MPI communications

```
Data: C: nb. of iter. of one block
      x: min. threshold of iter.
      X: max. threshold of cost
Input: cur_block: current C-block number
       nbiter: actual iter. number
if $nbiter > x$ AND $nbiter > cur\_block * C$ then
    cur_block++;
    Free previous sent messages;
    Check received messages;
    Forward aggregated messages if needed;
    Update best-block and best-ever messages;
    if $cost > X$ then
        Impact on search engine behavior;
        Send my message;
```

Managing MPI communications begins with an attempt to reuse the allocated buffer of the sent messages whose data have indeed completed to be sent. Then we check on the messages that have been received during the last C-block, messages are actually stored in MPI buffers. At this point, some aggregation of information can be performed in order to reduce messages to forward. The local Elite Pool is then updated with the new received information, and the search engine can decide to restart depending on a probability if the cost of some received messages is better than the local one: the configuration will be decided from the Elite Pool. If no restart is triggered, then the search engine send a message containing its configuration and associated information.

It is worth noticing that:

- The Elite Pool manager does not rely on any global communication pattern since they are non-blocking only in the recent MPI-3 standard, and thus not available in most of the implementations (still supporting at best the MPI-2-2 version).
- To perform a communication similar to a broadcast of some information performed by one search engine, the same message is sent to a subset of search engines which will forward the information, in a log(n) manner. All communications involved are non-blocking. The number of iterations for a C-block has thus to be taken with all the knowledge that messages have to reach their destinations which may take time depending on the underlying hardware.
- All Search Engines have a vision on their progress toward the solution that they are currently computing compared to information that they receive, are able to consider to restart from any configuration in its Elite Pool.
- In Algorithm 2, if the cost is under the threshold given by the user, a search engine still enters the MPI communication management procedure: in case the search engine decides to restart, it is possible to restart from the best or the neighborhood of the best received configuration instead of performing a random restart.

IV. CONTROLLING SEARCH ENGINE BEHAVIORS

The current implementation of the framework described above includes several ways to control the behavior of search engines and the trade-off between computation (*i.e.,* sequential local search) and communication between the solvers.

A. Communication

Depending on the problem (in particular the number of variables), the choice of using communication between solvers or not might greatly affect the performance. One can select the communication policy between solvers:

- pure independent walks,
- communication between solvers limited to only a given value (for example the cost, see [9] for a detailed presentation),
- communication among solvers of their current configuration.

B. Acceptance of Communication

An important control mechanism is to decide *when* a search engine who has received some configuration will discard its current configuration and start from one of the received configuration from the next iteration. An obvious criterion is that the received configuration has a better cost value than the current one, but one should be more careful than that, as switching every time to a configuration with a better cost value could lead to concentrate the solvers on the same area of the search space and possibly produce redundant work, or at least prevent diversification.

We thus decided to have a probabilistic *acceptance criterion* in order to have a search engine switching to a received configuration. This immediately brings forward the issue of which probability function (with its parameter) to use for acceptance. A first idea is to use a fixed probability threshold in order to introduce some *noise* in the acceptance of a communicated configuration. For instance a solver can decide that it will accept with probability 0.95 (*i.e.,* most of the time) or with probability 0.05 (*i.e.,* nearly never). We thus have a first acceptance policy based on a single parameter (constant probability). However, another observation is that the acceptance probability could be related to the cost values of the configurations. More precisely we would like to have a probability related to the difference or to the ratio of the current local cost and the one of the potentially new configuration. Indeed if a solver is currently working on a configuration of cost 1000001 and receives a (better) configuration of cost 1000000, it is probably useless to switch to it, as they are more or less equivalent (from the point of view of the cost). However if it received a configuration of cost 500, which is clearly better, then it should rather switch to it. Another observation is that the difference between a cost of 1000005 and another of 1000000 is not very significant, but between 15 and 10 is much more since a great number of iterations has been passed leading to these costs. Thus we decided to have a probability function related to the *ratio* of the costs of the current and the received configuration. As first candidate, we choose (for n=2, 4 and 6) the functions $e^{(-2/R-0.2)^n}$ where R is the ratio $cost_2/cost_1$, $cost_2$ being the cost of the received configuration and $cost_1$ the cost of the current configuration.

These functions are depicted in Figure 2 below.

As can be seen from this figure, if the ratio is less than 1 the probability of acceptance is zero for $e^{(-2/R-0.2)^6}$ and close to zero for $e^{(-2/R-0.2)^2}$ and $e^{(-2/R-0.2)^2}$. Another point to remark is that with $e^{(-2/R-0.2)^6}$ the probability of acceptance is 0.943 when the cost ratio is 2, while with $e^{(-2/R-0.2)^2}$ the probability of acceptance is only 0.539 when the cost ratio is 2.

Deciding which probability acceptance function is the best depends on the problem instance and more experimentation should be done to determine the interest of using each function.

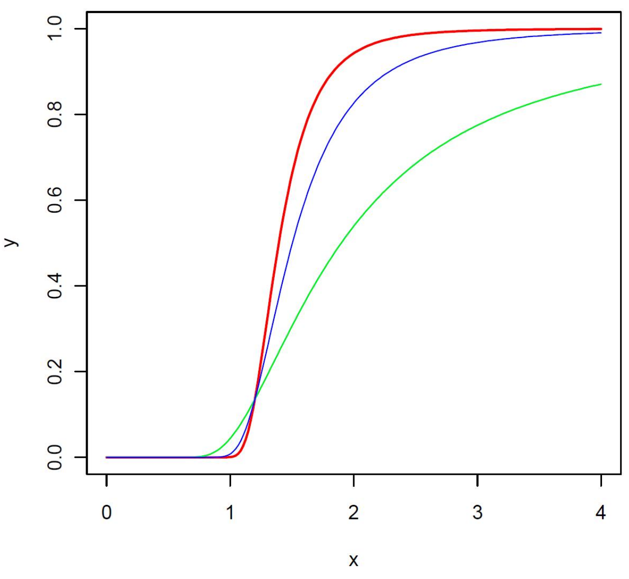

Figure 2. basic probability acceptance functions: $e^{(-2/R-0.2)^6}$ in red, $e^{(-2/R-0.2)^4}$ in blue and $e^{(-2/R-0.2)^2}$ in green

C. Sequential Behavior

As mentioned above, the *C-block* is the basic unit during which a solver will perform local search iterations without interacting with other search engines. The size of the C-block is defined as a number of iterations, and this size is common to all search engines of the same parallel run.

D. Beginning the Search

It is sometime useful to let a search engine start his computation in isolation of others for more than one C-block, *i.e.,* to have it not accepting communication and possible restarts for a certain time. This would let the search to *diversify* in a better way, and prevent early restarts which can be useless. Our framework allows to define a minimum number of iteration during which a solver is isolated from the others and do not take into account messages that may be sent by others.

E. During the Search

When a Search Engine decides to restart from a received configuration, it may be interesting to avoid any interaction with the others, giving thus the time to diversify. Still information is very important and thus non-blocking communications have to take place in order for the Elite Pool to update its current vision on the progress of its associated search engine, for later possible restart.

F. Finishing the Search

The behavior of the AS sequential engine on some combinatorial problems (for instance MAGIC-SQUARE), can be decomposed in two phases:

- in the beginning of the search, the solver improves the initial (random) configuration at each step, thus improves steadily the value of the cost function; we call this phase *deep search*.
- in the end of the search, when a solver is close to a solution, configurations with low cost values (possibly very low but non-zero) are explored and the solver is frequently stuck in local minima and performs resets; we call this phase *shallow search*.

Different problems exhibit different proportions of deep and shallow search. For instance with MAGIC-SQUARE the best run will generally exhibit a single deep search phase followed by a long phase of shallow search, while ALL-INTERVAL will rather exhibit series of deep searches and not much shallow search, see [9] for a detailed analysis.

It could be useless to perturb the behavior of a search engine during shallow search by making him restart from the configuration received from another solver, even if this configuration has a (slightly) better cost, because it might decrease diversification of the search. That is, all solvers might converge to the same part of the search space, which is not a desirable property in some cases.

Therefore we can define in our framework a cost value which will define the beginning of shallow search and thereafter prevent the solver to react to communications from other solvers as long as the cost value stays below this threshold. Of course if the shallow search is unsuccessful and the solver has to perform a reset or restart which will bring the cost of the configuration above the threshold of shallow search, then the solver will consider again the configurations communicated to him.

V. CONCLUSION

Following earlier work on *independent* multi-walk parallel local search, we presented in this paper a framework for *dependent* multi-walk and its implementation. It allows to communicate configurations between concurrent search engines and thus to focus the overall search on promising configurations. This should improve the performance of parallel local search for problems that do not exhibit a linear speedup. For instance, as shown in [5], the COSTAS ARRAY problem exhibits a linear speedup with independent multi-walks (even up to thousands of cores), while the MAGIC-SQUARE problem is sub-linear with independent multi-walks, efficiency being roughly 50% with 64 cores and 20% on 256 cores [4].

We are currently experimenting with this extended system on a series of benchmarks in order to compare the improvement of the dependent multi-walks versus the independent ones. We hope to have interesting results soon.

REFERENCES

[1] H. H. Hoos and T. Stützle, *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann / Elsevier, 2004.

[2] T. Gonzalez, Ed., *Handbook of Approximation Algorithms and Metaheuristics*. Chapman and Hall / CRC, 2007.

[3] M. Verhoeven and E. Aarts, "Parallel local search," *Journal of Heuristics*, vol. 1, no. 1, pp. 43–65, 1995.

[4] Y. Caniou, P. Codognet, D. Diaz, and S. Abreu, "Experiments in Parallel Constraint-Based Local Search," in *EvoCOP 2011, 11th European Conference on Evolutionary Computation in Combinatorial Optimisation*, ser. Lecture Notes in Computer Science, P. Merz and J.-K. Hao, Eds. Torino, Italy: Springer Verlag, 2011, pp. 96–107.

[5] D. Diaz, F. Richoux, Y. Caniou, P. Codognet, and S. Abreu, "Parallel Local Search for the Costas Array Problem," in *IEEE Workshop on new trends in Parallel Computing and Optimization (PCO'12), held in conjunction with IPDPS 2012*. Shanghai, China: IEEE, May 2012, pp. 1787–1796.

[6] P. Van Hentenryck and L. Michel, *Constraint-Based Local Search*. The MIT Press, 2005.

[7] C. Truchet, F. Richoux, and P. Codognet, "Prediction of parallel speed-ups for las vegas algorithms," in *ICPP'13, 43rd International Conference on Parallel Processing*. IEEE Press, Oct. 2013.

[8] T. G. Crainic, M. Gendreau, P. Hansen, and N. Mladenovic, "Cooperative parallel variable neighborhood search for the -median," *Journal of Heuristics*, vol. 10, no. 3, pp. 293–314, 2004.

[9] Y. Caniou and P. Codognet, "Communication in parallel algorithms for constraint-based local search," in *IEEE Workshop on new trends in Parallel Computing and Optimization (PCO'11), held in conjunction with IPDPS 2011*. Anchorage, USA: IEEE, May 2011, pp. 1961–1970.

[10] P. Codognet and D. Diaz, "Yet another local search method for constraint solving," in *proceedings of SAGA'01*. Springer Verlag, 2001, pp. 73–90.

[11] ——, "An efficient library for solving CSP with local search," in *MIC'03, 5th International Conference on Metaheuristics*, T. Ibaraki, Ed., 2003.