coPSSA - Constrained Parallel Stretched Simulated Annealing

José Rufino*†, Ana I. Pereira*‡ and Jan Pidanic§
*Polytechnic Institute of Bragança, Bragança, Portugal
†Laboratory of Instrumentation and Experimental Particle Physics, University of Minho, Braga, Portugal
‡Algoritmi R&D Centre, University of Minho, Braga, Portugal
§Faculty of Electrical Engineering and Informatics, University of Pardubice, Czech Republic
Email: rufino@ipb.pt, apereira@ipb.pt, Jan.Pidanic@upce.cz

***Abstract*—Parallel Stretched Simulated Annealing (PSSA) solves unconstrained multilocal programming optimization problems in distributed memory clusters, by applying the Stretched Simulated Annealing optimization method, in parallel, to multiple sub-domains of the original feasible region. This work presents coPSSA (constrained Parallel Stretched Simulated Annealing), an hybrid application that combines shared memory based parallelism with PSSA, in order to efficiently solve constrained multilocal programming problems. We devise and evaluate two different parallel strategies for the search of solutions to these problems. Evaluation results from a small set of test problems often reach superlinear speedup in the solution search time, thus proving the merit of the coPSSA parallelization approach.**

I. Introduction

Multilocal programming problems are numerical optimization problems that may be found in many theoretical fields, like reduction methods for solving semi-infinite programming problems [7], [16], or in more practical scenarios, like ride comfort optimization [2] and Chemical Engineering, specifically in the areas of process synthesis, design and control [3].

Multilocal programming problems may be *unconstrained* or *constrained*. In the latter case, they are subject to (s.t.) certain restrictions. More formally, a *constrained* multilocal programming problem may be defined by the formulation (1):

$$\begin{array}{ll} \max & f(x) \\ \text{s.t.} & h_k(x) = 0,\, k \in E \\ & g_j(x) \leq 0,\, j \in I \\ & -b_i \leq x_i \leq b_i,\, i = 1, \ldots, n \end{array} \tag{1}$$

where at least one of the n-dimensional functions $f, h_k, g_j : \mathbb{R}^n \to \mathbb{R}$ is nonlinear, and E and I are index sets of equality and inequality constraints, respectively. Since concavity is not assumed, the nonlinear optimization problem can have many global and local (non-global) maxima. Consider the *feasible region* (search space) defined by $\mathcal{R} = \{x \in \mathbb{R}^n : -b_i \leq x_i \leq b_i, i = 1, \ldots, n;\ h_k(x) = 0, k \in E;\ g_j(x) \leq 0, j \in I\}$. Thus, the purpose of the maximization problem (1) is to find all local maximizers, that is, all points $x^* \in \mathcal{R}$ such that the condition (2) holds for a vicinity $V_\epsilon(x^*)$ with a positive ray ϵ:

$$\forall x \in V_\epsilon(x^*) \cap \mathcal{R},\ f(x^*) \geq f(x). \tag{2}$$

It is also assumed that problem (1) has a finite number of isolated global and local maximizers. The existence of multi-solutions (local and global) makes this problem a great challenge that may be tackled with parallel solving techniques.

The most common methods for solving multilocal optimization problems are based on evolutionary algorithms, such as genetic [1] and particle swarm [12] algorithms. Additional contributions may be found in [6], [18], [20], [21]. Stretched Simulated Annealing (SSA) was also proposed [13], combining simulated annealing and a stretching function technique, to solve *unconstrained* multilocal programming problems.

In previous work [15], [17], Parallel Stretched Simulated Annealing (PSSA) was presented (and successively refined) as a parallel version of SSA, based on the decomposition of the initial search domain in several sub-domains to which SSA is independently applied by a set of processors. Several domain decomposition and work assignment approaches were explored, ranging from *homogeneous data decomposition* and *static work assignment*, to *heterogeneous data decomposition* and *dynamic work assignment*, leading to successively increasing levels of numerical efficiency (here, the main contribution of parallelization is not to decrease optima search times, but to increase the number of optima found in a bounded time).

This paper discusses the solving of *constrained* multilocal programming problems by coPSSA (*Constrained PSSA*), an hybrid application that implements a numerical penalty method, by combining shared memory parallelism with the current PSSA implementation. In coPSSA, the penalty method uses PSSA as an external software component, that must be invoked every time certain penalty parameters are changed. These parameters must be changed an unpredictable number of times, before convergence to the problem solution(s) is achieved. By splitting the parameters test space among several processors, it becomes possible to check, simultaneously, several convergence paths, thus accelerating the solving of the constrained problem. This translates in having several instances of PSSA executing in parallel (one per convergence path being tested), in a multi-master-slaves configuration. We evaluate two different strategies for the decomposition of the parameters test space (*alternation* and *slicing*). Both strategies are very effective, performance-wise, leading to superlinear speedups in search times for a small set of constrained test problems.

The rest of the paper is organized as follows: Section 2 revises our previous work on unconstrained optimization, covering the SSA method and the PSSA parallel approach; Section 3 presents the fundamentals of constrained optimization and introduces the coPSSA application; Section 4 presents the benchmarks scenario and the results of the evaluation; the last section concludes and points directions for future work.

II. Unconstrained Optimization

A. Stretched Simulated Annealing (SSA)

Stretched Simulated Annealing (SSA) is a multilocal programming method that solves *unconstrained* optimization problems. Generically, these problems may be described as:

$$\max_{x \in X} \varphi(x), \tag{3}$$

where $\varphi : \mathbb{R}^n \to \mathbb{R}$ is a given n-dimensional multimodal objective function (a function with many local optima[1]) and the feasible region is X, a compact set defined by $X = \{x \in \mathbb{R}^n : -b_i \leq x_i \leq b_i, i = 1, ..., n\}$. The SSA method solves a sequence of global optimization problems in order to compute the local solutions of the maximization problem (3). The objective function of each global optimization problem is obtained by applying a stretching function technique [11].

Let x^* be a solution of problem (3). The mathematical formulation of the global optimization problem is as follows:

$$\max_{x \in X} \Phi_l(x) \equiv \begin{cases} \hat{\phi}(x) & \text{if } x \in V_\varepsilon(x^*) \\ \varphi(x) & \text{otherwise} \end{cases} \tag{4}$$

where $V_\varepsilon(x^*)$ is the vicinity of solution x^* with a ray $\varepsilon > 0$.

The $\hat{\phi}(x)$ function is defined as

$$\hat{\phi}(x) = \bar{\phi}(x) - \frac{\delta_2[\text{sign}(\varphi(x^*) - \varphi(x)) + 1]}{2\tanh(\kappa(\bar{\phi}(x^*) - \bar{\phi}(x))} \tag{5}$$

where δ_1, δ_2 and κ are positive constants and $\bar{\phi}(x)$ is

$$\bar{\phi}(x) = \varphi(x) - \frac{\delta_1}{2}\|x - x^*\|[\text{sign}(\varphi(x^*) - \varphi(x)) + 1]. \tag{6}$$

To solve the global optimization problem (4) the Simulated Annealing (SA) method is used [5]. The Stretched Simulated Annealing algorithm stops when no new optimum is identified after r consecutive runs. For more details see [14], [15].

B. Parallel Stretched Simulated Annealing (PSSA)

The search for optima with SSA is an *embarrassingly parallel* problem. SSA searches for solutions in a given domain by applying a stochastic algorithm ι consecutive times. Augmenting ι increases the hit rate, but also the execution times. An alternative is to keep ι constant and to generate a partition of the initial search domain, consisting of disjoint sub-domains where SSA is applied independently, once there are no data dependencies involved. With a dedicated processor per sub-domain, each sub-domain will take no more time to search than the initial domain (in fact, the trend is to take less time, once the search scope is smaller). Thus, with as many processors as sub-domains, the numerical efficiency (number of optima found) may improve without degrading performance.

The parallelization strategy above, based on a Data Decomposition approach, is the one adopted by PSSA, in three variants that diverge on the way in which sub-domains are defined and then assigned to processors. In PSSA-HoS (Homogeneous decomposition, Static assignment), sub-domains are generated only once, have equal size and processors self assign the same number of sub-domains. PSSA-HoD (Homogeneous decomposition, Dynamic assignment) differs from PSSA-HoS in sub-domains being assigned to processors on-demand (thus possibly in varying number). Finally, in PSSA-HeD (Heterogeneous decomposition, Dynamic assignment), the initial homogeneous domain partition suffers a recursive adaptive refinement, leading to an unpredictable number of sub-domains, of variable size, dynamically generated and processed (on-demand), until certain stop criteria are met. Both PSSA-HoS and PSSA-HoD have equal numerical efficiency, but the PSSA-HoD variant is faster due to its workload auto-balancing. PSSA-HeD typically finds more optima, but is also the slowest, once it usually searches in many more sub-domains.

PSSA is written in C and runs on Linux, whether in shared-memory multi-core workstations or in distributed-memory clusters. This diversity of parallel execution environments is possible because PSSA is a parallel application built on the Message Passing paradigm, on top of an MPI [8] implementation. In this regard, PSSA follows the Single Program Multiple Data (SPMD) execution model and operates in a *master-slaves* configuration: *slave* MPI processes apply SSA in sub-domains; a *master* process does coordination and optima post-processing; with c CPU-cores reserved for PSSA, the MPI process mapping we adopted assigns one core to the *master* and the remaining $c - 1$ cores to the *slaves* (one per core).

III. Constrained Optimization

A. Penalty Method with the l_1 Penalty Function

In general, constrained optimization problems are harder to solve than unconstrained optimization problems, specially when the feasible region (problem domain) is not concave and is very small when compared with the whole search space.

There are three main classes of methods to solve constrained optimization problems [9], [22]: A) methods that use penalty functions, B) methods based on biasing feasible over infeasible solutions, and C) methods that rely on multi-objective optimization concepts. In this work constraints are handled using a class A method with the l_1 penalty function. The l_1 penalty function is a technique [10] defined by

$$P(x, \mu) = f(x) - \frac{1}{\mu}\left[\sum_{k \in E} |h_k(x)| + \sum_{j \in I} \lceil g_j(x) \rceil^+\right]$$

where μ is a positive penalty parameter that decreases to zero. A lower bound $\mu_{\min}$ is defined and the update is as follows:

$$\mu^{k+1} = \max\left\{\tau\mu^k, \mu_{\min}\right\} \tag{7}$$

where k represents the iteration, $\mu_{\min} \approx 0$ and $0 < \tau < 1$.

To solve the constrained optimization problem (1), the penalty method solves a sequence of unconstrained problems using the l_1 penalty function defined by

$$\max_{x \in X} P(x, \mu^k). \tag{8}$$

The problem (8) is solvable using the PSSA method. It is possible to prove that the sequence of solutions $\{x^*(\mu^k)\}$, from (8), will converge to the solution x^* of (1) [10], [14].

The penalty method stops when the successive solutions are similar, or a maximum number of iterations (K) is reached.

[1]See [4], [19] for plots of several reference functions of this kind.

B. Constrained PSSA (coPSSA)

As stated in the previous section, the penalty method will need to invoke the SSA method, possibly many times, until convergence is reached. In this work, we explore the coupling of the penalty method with PSSA, our parallel SSA implementation. When designing the interaction between the two, the decision was to keep PSSA as a self-contained application and to develop a separate application (coPSSA), that acts as a client of the PSSA application. Thus, from the perspective of the coPSSA application, PSSA is regarded as an external software component, with well-defined inputs and outputs, that is spawned when necessary. We believe that this modular design (in opposition to a single, monolithic application) will facilitate the integration of further optimization techniques.

Figure 1 is a representation of the coPSSA application and its interactions with the PSSA application.

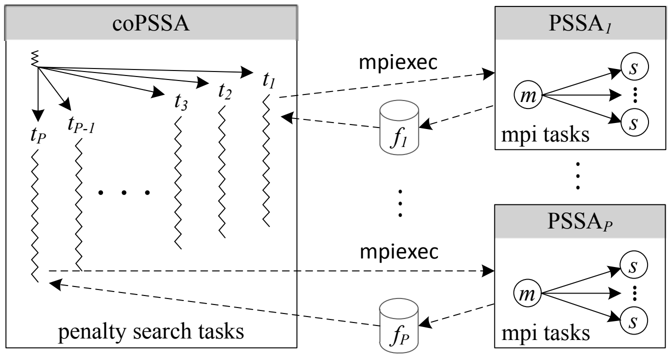

Fig. 1. coPSSA and its interactions with PSSA (t_p are coPSSA search tasks; m and s are PSSA tasks (master and slaves); f_P are result files from PSSA).

The coPSSA application is feed with several parameters: i) an identifier of the test problem $f(x)$, ii) the base initial value μ^0 for μ, iii) the number P of simultaneous search tasks, iv) the strategy to be used to decompose the μ test space, iv) a base MPI hostfile, and v) some parameters to forward to PSSA.

The coPSSA application is itself a parallel application, based on the conventional multi-process approach and System V IPC mechanisms[2], that forks P search tasks. Each one of these tasks will solve problem (8), a certain number of times, each time with a different value of μ^k. Overall, the test space for μ^k comprises K different values, defined by (7). These values are auto-assigned by the search tasks in two possible different ways: a) *alternation*; b) *slicing*. In both ways the test values are evenly assigned to the tasks, as next explained.

Let t_p be a search task, with $p = 1, 2, ..., P$; let μ^k be a test value, with $k = 1, 2, ..., K$; let $w = K \; div \; P$ be the uniform width (i.e., $K \, mod \, P = 0$)[3] of a test sub-space of μ^k.

With *alternation*, a task t_p will test μ^k if $p-1 = k \; mod \; P$. For instance, with $P = 4$ tasks, and $K = 100$ test values, task t_1 will test μ^1, $\mu^5, ...$; task t_2 will test μ^2, $\mu^6, ...$; and so on.

With *slicing*, a task t_p will test μ^k if $k \in \{k^p_{left}, ..., k^p_{right}\}$, where $k^p_{left} = (p-1) \times w + 1$ and $k^p_{right} = p \times w$. For instance, with $P = 4$ tasks, and $K = 100$ test values, task t_1 will test $\mu^1, \mu^2, ..., \mu^{25}$; task t_2 will test $\mu^{26}, \mu^{27}, ..., \mu^{50}$; and so on.

Prior to invoking PSSA, a coPSSA search task properly assembles the related command line string. This string includes the `mpiexec` MPI launcher (that will spawn several instances of the PSSA executable) and several parameters. Some of these replicate parameters initially supplied to the coPSSA application, that are just being forwarded to PSSA; these include the identifier of the test problem $f(x)$, the specific PSSA variant to execute (HoS, HoD or HeD) and the granularity g of the initial homogeneous partition of the search domain performed by PSSA. Other parameters are specific of each search task, whether fixed, like MPI related parameters, or variable during the task lifetime, like μ^k. The MPI parameters specify the number and location of the PSSA instances to execute, every time the search task spawns PSSA; this configuration is derived from the base MPI hostfile that is initially given to the coPSSA application; basically, the execution slots of the base MPI hostfile are evenly divided among the coPSSA search tasks, originating specific and disjoint hostfiles, one per search task. The crafted command string is then submitted to the `system` primitive. A coPSSA search task will then block while PSSA executes; when it finishes, its results are collected from a file.

The problems (multimodal objective functions) testable by PSSA are all implemented in a specific module which, until know, included only unconstrained problems. In order to support the work of this paper, it was necessary to augment that module with the implementation of functions $f(x)$ and $P(x, \mu)$, for each constrained problem tested. This was the only extension that was necessary to make, in the original PSSA code, in order to support the case study of this paper.

After each PSSA execution, its results are analyzed to verify if they fulfill certain convergence criteria. If they haven't converged, and the search task has not yet exhausted its μ^k values, the task will inspect a shared memory flag (visible by all search tasks, and protected by a semaphore) before moving on to its next μ^k; if the flag has been changed from its neutral initialization value, then it will hold the identifier (PID) of another task that has already converged, in which case the current search task aborts. If the results of the current iteration have converged, the shared memory flag will also be inspected; if the flag has not yet been modified, than the search task is the first one to converge, in which case changes the flag to its PID and stops the search; if the flag was already modified, some other task converged first, and so the task will abort.

IV. Evaluation

A. Setup

Evaluation took place in a commodity cluster of 8 worker nodes, with one Intel Core-i7 4790K 4.0GHz quad-core CPU per node, running Linux ROCKS version 6.1.1, with the Gnu C Compiler (GCC) version 4.4.7 and OpenMPI version 1.5.4.

In all tests, coPSSA was executed in a separate cluster host (its frontend), with the number of search tasks ranging from 1 to 8 (despite the frontend having also a quad-core CPU, we did not detect any overloading when executing 8 coPSSA search tasks). PSSA executions took place in the 8 worker nodes; these offer a total of 32 CPU-cores, fully specified in the base

[2]This classical approach is enough for a prototype level application. Alternatives like POSIX threads or OpenMP may be explored in the future.

[3]This assumption is only to ease this formal discussion. The coPSSA implementation handles properly non-uniform decomposition when $K \; mod \; P \neq 0$.

MPI hostfile supplied to coPSSA, that are used, four at a time, to service the PSSA execution requests of each coPSSA task; thus, each PSSA execution always consumed 4 cores, with 1 core for the master process, and 3 cores for slave processes. In order to fully exploit the 3 slave cores, the number of sub-domains processed by PSSA was defined to be no less (and as close as possible) than 3; for 2-dimensional problems, like the ones we tested, this is achieved with a granularity $g = 0.5$, that generates 4 sub-domains [15]. The PSSA variant used was always the HoD variant, once it is the fastest and uses a fixed number of sub-domains (4, in our evaluation scenario).

We have tested both strategies proposed (*alternation* and *slicing*) for the decomposition of the μ^k space. All tests shared the constrained parameters $K = 100$, $\mu^0 = 1.0$ and $\tau = 0.7$. The constrained problems selected for the tests were the 2-dimensional problems G8 and G11 from [4]. Specific (P)SSA numerical parameters were $r = 5$, $\sigma_1 = 1.5$, $\sigma_2 = 0.5$ and $\kappa = 0.05$.

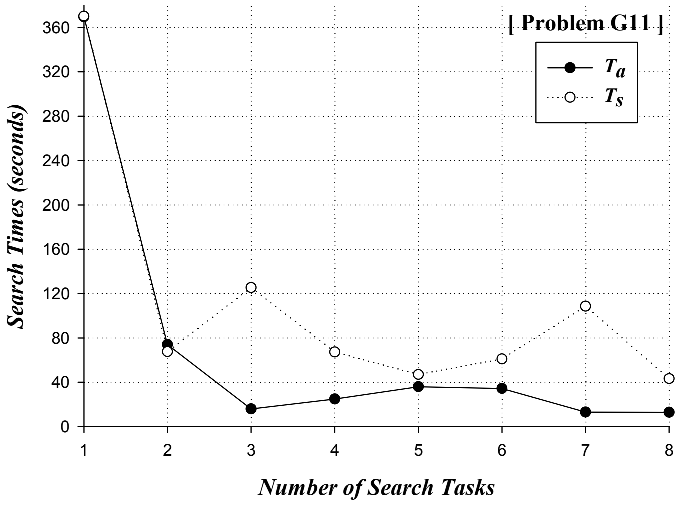

Fig. 3. Problem G11 - Search times with alternation (T_a) and slicing (T_s).

B. Results

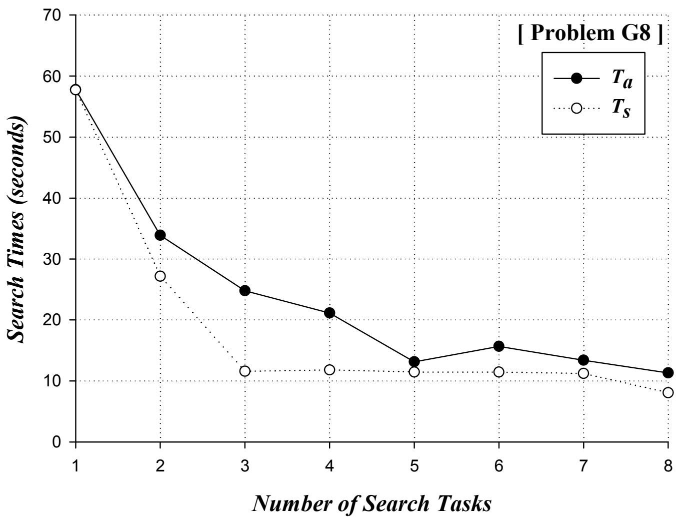

Fig. 2. Problem G8 - Search times with alternation (T_a) and slicing (T_s).

Figures 2 and 3 plot the search times with alternate (T_a) and sliced searching (T_s), for problems G8 and G11, respectively. Two important conclusions may readily be derived from these figures: i) in general, having more search tasks helps to decrease the solutions search times, for both problems, and for both search approaches; however, for problem G8, with slicing there's none (or very small) benefit in having more than 3 search tasks; moreover, for problem G11, search times may even increase in both approaches (smoothly with alternation, more dramatically with slicing), although well bellow the search time of a single task; ii) no approach (alternation vs slicing) is a clear winner; with problem G8, slicing provides the best search times; with problem G11 the fastest approach is alternation; this opposite behavior, coupled with unpredictable surges in search times when running more search tasks, shows how difficult it is to develop search techniques that perform reasonably well with a broad set of constrained problems.

Although we do not show the values of μ^k that lead to convergence, another interesting observation derived from our experiments is that those values may vary considerably, depending on the search approach (alternation vs slicing), and on the number of search tasks. This is easily explainable. Suppose that, with a single search task, which consecutively tests μ^1 to μ^{100}, the μ^k that leads to convergence is μ^3; it may happen that, with slicing and two tasks (the first one testing μ^1 to μ^{50}, and the second one testing μ^{51} to μ^{100}), the second task reaches convergence first, for instance, with μ^{52}; it may also happen that, with alternation and two tasks (the first one testing μ^i, where i is odd, and the second one testing μ^j, where j is even), the first task is now unable to converge with μ^3, simply because the convergence test, that also depends on the previous μ^k tested, considers μ^1 as the preceding μ^k, instead of μ^2 (which now belongs to the second search task).

Despite the nature of the search process, that may introduce some uncertainty on the search times, it is indisputable that the parallelization techniques used by coPSSA introduce important performance gains. This may be clearly recognized by the parallel search efficiency achieved with more than one search task, that is often of superlinear nature – see Figures 4 and 5.

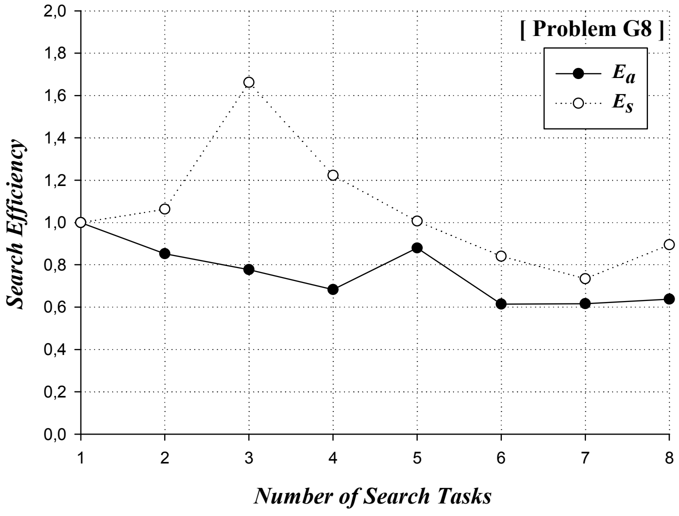

Fig. 4. Problem G8 - Search efficiency with alternat. (E_a) and slicing (E_s).

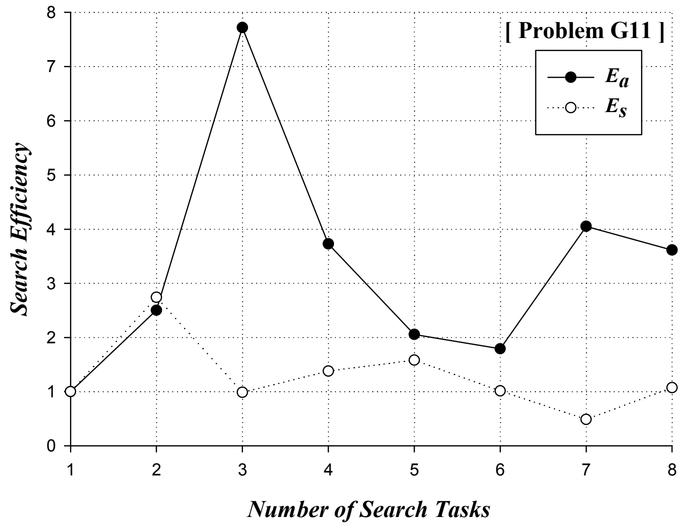

Fig. 5. Problem G11 - Search efficiency with alternat. (E_a) and slicing (E_s).

In Figures 4 and 5, E_a and E_s measure the parallel search efficiency with alternation and slicing, respectively. Generically, the efficiency E with P search tasks is $E_P = S_P/P$, where S_P is the speedup achieved with P search tasks; in turn, $S_P = T_1/T_P$, where T_1 is the search time with one task, and T_P is the search time with P tasks. Thus, E measures how well-utilized the search tasks are in solving the problem. A value $E > 1.0$ translates into a situation were efficiency is superlinear. We may observe that, in problem G8, efficiencies varies between ≈ 0.6 and ≈ 1.6, but are mostly bellow 1.0. However, in problem G11 superlinear efficiency is the norm (with a single exception, for sliced search with 7 search tasks).

Again, we note that performance gains may vary dramatically, for different constrained problems. Clearly, more tests are necessary, with a broader set of problems (though other results, not discussed here, seem to validate our approach).

V. Conclusion

In this paper we have presented coPSSA, an hybrid application that solves constrained optimization problems, by integrating a numerical penalty method with a parallel solver of unconstrained problems (PSSA). Two types of parallelism were explored: shared-memory parallelism, to split the test space of a penalty parameter, and distributed-memory parallelism, to test the possible convergence to the problem solution with each parameter value. In addition, two different strategies were compared (*alternation* and *slicing*), to assign the penalty parameter values by the search tasks. Although conducted with a limited set of numerical problems, our preliminary evaluation of coPSSA shows clear performance gains with regard to search times (including superlinear efficiencies). The same evaluation, however, showed that no search strategy (*alternation* or *slicing*) is always the best: depending on the problem properties, one may converge faster than the other.

In the future, we intend to refine this work, by solving more constrained problems (including problems with more than 2 dimensions) and by studying the effect of the variation of other parameters (like τ) on the time to converge to the solution. More efficient mechanisms (above the file system) to exchange results between PSSA and coPSSA will also be explored.

References

[1] Chelouah, R., Siarry, P.: A continuous genetic algorithm designed for the global optimization of multimodal functions, *Journal of Heuristics*, 6, 191–213 (2000).

[2] Eriksson, P., Arora, J.: A comparison of global optimization algorithms applied to a ride comfort optimization problem, *Structural and Multidisciplinary Optimization*, 24, 157–167 (2002).

[3] Floudas, C.: Recent advances in global optimization for process synthesis, design and control: enclosure of all solutions, *Computers and Chemical Engineering*, 963–973 (1999).

[4] Hedar, A.-R.: Global Optimization Test Problems, http://www-optima.amp.i.kyoto-u.ac.jp/member/student/hedar/Hedar_files/TestGO.htm .

[5] Ingber, L.: Very fast simulated re-annealing, *Mathematical and Computer Modelling*, 12, 967–973 (1989).

[6] Kiseleva, E. , Stepanchuk, T.: On the efficiency of a global non-differentiable optimization algorithm based on the method of optimal set partitioning, *Journal of Global Optimization*, 25, 209–235 (2003).

[7] León, T., Sanmatías, S., Vercher, H.: A multi-local optimization algorithm, *Top*, 6(1), 1–18 (1998).

[8] Message Passing Interface Forum - http://www.mpi-forum.org/.

[9] Michalewicz, Z.: A survey of constraint handling techniques in evolutionary computation methods. Proceedings of the 4th Annual Conference on Evolutionary Programming pp. 135-155 (1995)

[10] Nocedal, J., Wright, S.: Numerical Optimization, *Springer Series in Operations Research*, Springer (1999).

[11] Parsopoulos, K., Plagianakos, V., Magoulas, G., Vrahatis, M.: Objective function stretching to alleviate convergence to local minima, *Nonlinear Analysis*, 47, 3419–3424 (2001).

[12] Parsopoulos, K., Vrahatis, M.: Recent approaches to global optimization problems through particle swarm optimization, *Natural Computing*, 1, 235–306 (2002).

[13] Pereira, A. I., Fernandes, E. M. G. P.: Constrained Multi-global Optimization using a Penalty Stretched Simulated Annealing Framework, *Numerical Analysis and Applied Mathematics, AIP Conference Proceedings*, 1168, 1354–1357 (2009).

[14] Pereira, A. I., Ferreira, O., Pinho, S. P., Fernandes, E. M. G. P.: Multilocal Programming and Applications, *Handbook of Optimization*, Edited by I. Zelinka, V. Snasel and A. Abraham, Intelligent Systems series, Springer-Verlag, 157–186 (2013).

[15] Pereira, A.I., Rufino, J.: PSSA: A. Pereira, J. Rufino, Solving Multilocal Optimization Problems with a Recursive Parallel Search of the Feasible Region, ICCSA 2014, LNCS 8580, 2014, pp 154-168, Springer International Publishing, 2014.

[16] Price, C.J., Coope, I.D.: Numerical experiments in semi-infinite programming, Computational Optimization and Applications, 6, , 169-189 (1996).

[17] Ribeiro, T., Rufino, J., Pereira, A.I.: PSSA: Parallel Stretched Simulated Annealing, *Numerical Analysis and Applied Mathematics, AIP Conference Proceedings*, 1389, 783–786 (2011).

[18] Salhi, S., Queen, N.: A Hybrid Algorithm for Identifying Global and Local Minima When Optimizing Functions with Many Minima, *European Journal of Operations Research*, 155, 51–67 (2004).

[19] Surjanovic, S., Bingham, D.: Virtual Library of Simulation Experiments: Test Functions and Datasets, http://www.sfu.ca/~ssurjano .

[20] Tsoulos I., Lagaris, I.: Gradient-controlled, typical-distance clustering for global optimization, *www.optimization.org* (2004).

[21] Tu, W. , Mayne, R.: Studies of multi-start clustering for global optimization, *International Journal Numerical Methods in Engineering*, 53, 2239–2252 (2002).

[22] Wang, Y., Cai, Z., Zhou, Y., Fan, Z.: Constrained optimization based on hybrid evolutionary algorithm and adaptive constraint-handling technique. Struct. Multidiscip. Optim. 37, 395-413 (2008)