Optimizing a GPU-parallelized Ant Colony Metaheuristic by Parameter Tuning

Andrey Borisenko[1] and Sergei Gorlatch[2]

[1] Tambov State Technical University, Russia
borisenko@mail.gaps.tstu.ru
[2] University of Muenster, Germany,
gorlatch@uni-muenster.de

Abstract. We address the problem of accelerating the GPU-parallelized Ant Colony Optimization (ACO) metaheuristic used for an important class of optimization problems – design of multiproduct batch plants, with a particular use case of a *Chemical-Engineering System* (CES). We propose and implement a novel approach to ACO's parameter tuning, with the following advantages compared to previous work: we accelerate tuning by using GPU, and we do not require additional constructs like function mapping in fuzzy logic, algorithms for online-tuning, etc. We report our experimental results that confirm the efficiency of parameter tuning and the advantages of our approach.

Keywords: constraint satisfaction problem · ant colony optimization · tuning metaheuristics · parallel metaheuristics · GPU computing · multi-product batch plant design.

1 Motivation and Related Work

The Ant Colony Optimization (ACO) metaheuristic is a popular approach to solving optimization problems. It can be viewed as a multi-agent system in which agents (ants) interact with each other in order to reach a global goal [10]. ACO follows the idea of collective intelligence in colonies of ants: the ants cooperatively search for food and bring this food to their nest. While walking between food sources and the nest, ants deposit a chemical substance called *pheromone* on their path. The pheromone is used to find the shortest path from their nest to food. Parameters of ACO determine the probability with which ants follow the pheromone deposited by previous ants, and how fast the pheromone evaporates.

We apply ACO to an important class of real-world optimization problems – optimal design of multiproduct batch plants, with a particular use case of a *Chemical-Engineering System* (CES). Such a system is a set of equipment units (reactors, tanks, filters, dryers etc.) which manufacture products, and the problem is finding the optimal number of units at processing stages and their main sizes for the given input that includes: demand for each product of assortment, production horizon, accessible equipment set, etc. This problem is NP-hard, i.e., the time to solve a problem instance grows exponentially with the instance size.

Therefore, metaheuristics are often the only feasible way to obtain good-quality solutions at acceptable computational cost [11]. In our previous work [4], we develop a hybrid parallel algorithm consisting of two metaheuristics: 1) ACO finds an initial soluting of the Constraint Satisfaction Problem (CSP); 2) this initial solution is then optimized using Simulated Annealing (SA). The hybrid ACO+SA algorithm is parallelized for Graphics Processing Units (GPU) and successfully solves the design optimization problem for CES of size up to $12^{16} \approx 10^{17}$ variants; it demonstrates a significant time saving as compared to the traditional branch-and-bound optimization method.

In this paper, we aim at further acceleration of the GPU-parallelized ACO method, in order to apply it to even larger sizes of problems that arise in practice: already starting with the size 12^{16}, the run time of the original algorithm becomes prohibitively high despite the use of a highly parallel GPU. Our approach is to use the additional performance potential offered by the tunable parameters of the ACO algorithm.

A significant amount of work has been devoted to tuning ACO parameters [2] that has proven to be a hard problem [11, 24].

The approaches to parameters tuning can roughly be divided into offline versus online procedures. A tuning framework [25] is based on the sequential optimization of perturbed regression models. Paper [1] presents a methodology combining statistical and artificial intelligence methods in the fine-tuning of metaheuristics. Paper [21] uses a fuzzy system for parameter adaptation in the ACO metaheuristic. In [23], the problem of finding the parameters of a metaheuristic algorithm is formulated as a meta-optimization problem solved by an evolutionary metaheuristic. An enhanced ACO with dynamic mutation and ad hoc initialization for generating the initial ant solutions to improve the accuracy of fuzzy system design is proposed in [8]. Paper [7] explores a new fuzzy approach for diversity control in ACO. In [12], a parameter tuning methodology for metaheuristics based on the design of experiments is proposed. Paper [13] uses a Particle Swarm Optimization (PSO) algorithm to optimize the ACO parameters.

We propose and implement a novel approach to parameter tuning for an ACO algorithm that solves the CSP (Constraint Satisfaction Problem) part of our global problem. The main differences of the proposed approach to the previous work are as follows. By tuning for CSP, rather than for the optimization problem as in [17, 21], we can apply frequency analysis. For calculating how often values occur within a range of values, we do not need any specific and non-obvious information, like functions mapping in fuzzy logic, an algorithm for online-tuning, etc. [7, 8, 21]. The parallelization of the algorithm and the use of modern GPUs allow us to conduct a large number of computational experiments to accumulate statistical data in a short time. An advantage is also that the found optimal values of parameters for can be used for both parallel and sequential versions of the algorithm. Summarizing, the advantages of our approach as compared to previous work are two-fold: 1) we exploit the computation power of GPU in

the tuning process, and 2) we do not rely on any additional information like functions mapping in fuzzy logic, an algorithm for online-tuning, etc.

In the remainder of the paper, Section 2 outlines the CES optimization problem and its GPU-implementation. In Section 3, we analyze our ACO algorithm for CSP problem and the roles of its parameters, and we describe our novel methodology of ACO parameters tuning. In Section 4, we report our experimental results that confirm the advantages of our approach, and Section 5 concludes.

2 GPU-Algorithm for Designing Multi-Product Plants

Our application use case is designing a *Chemical-Engineering System* (CES) – a set of equipment (reactors, tanks, filters, dryers etc.) for manufacturing diverse products. Assuming that the number of units at every stage of CES is fixed, the problem can be formulated as follows (for a detailed formulation, see [3]). A CES consists of a sequence of I processing stages; i-th stage can be equipped with equipment units from a finite set X_i, with J_i being the number of equipment units variants in X_i. The goal is to find the optimal number of units at stages and their main sizes; the input data are: production horizon, demand for each product, available equipment, etc. Each system's variant Ω_e has to be in an operable condition (*compatibility constraint*) expressed by function S: $S(\Omega_e) = 0$. If T_{max} is the total available time horizon, then an operable variant of a CES must also satisfy a *processing time constraint*: $T(\Omega_e) \leq T_{max}$.

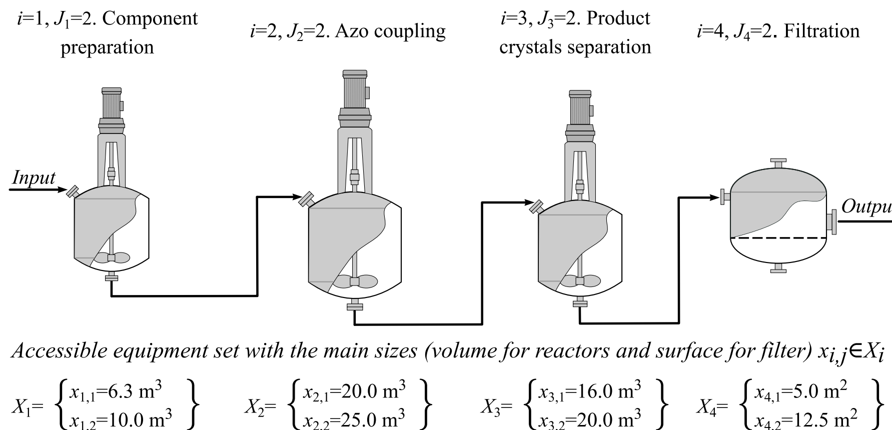


Fig. 1. Example: A Simple Chemical-Engineering System (CES)

Figure 1 shows an example CES consisting of 4 stages ($I = 4$), where each stage can be equipped with 2 devices ($J_1 = J_2 = J_3 = J_4 = 2$); the number of all possible system variants in this case is $2^4 = 16$. The optimization works on the search tree, in which each path from the root to one of the leaves in this tree corresponds to a candidate solution of the optimization problem.

In [3], we create a hybrid approach to optimizing CES; it combines two metaheuristics – *Ant Colony Optimization (ACO)* and *Simulated Annealing (SA)*. We parallelize and implement it on a CPU-GPU system using CUDA [19] and we show that it is preferable to the popular Branch-and-Bound (B&B) method [5]. In our approach, the solution of the optimization problem is divided into two stages: (1) construct a *feasible* (i.e., functionally operative CES variant) initial solution using ACO; (2) improve this feasible solution using SA. While for classical optimization problems, e.g., Traveling Salesman Problem (TSP), it is possible to use a random initial solution [17], in our case the random initialization is unacceptable, because the *compatibility* and the *processing time* constraints must be satisfied. Our search for a feasible solution in the first stage is a Constraint Satisfaction Problem (CSP) [26] which consists in finding an operable variant of a CES, where both the compatibility constraint and the processing time constraint are satisfied. For solving this problem, we use ACO.

In our parallel implementation on a GPU [3], the ACO kernel function searches for the first feasible solution using the Multiple Ant Colonies approach [9]: all colonies work as threads in parallel to solve a problem independently.

```
1  AntColonyOptimization()
2  { isFound = false; /* repeat while solution not found */
3   while(!isFound && iterCounter < maxIterNumber){
4    Initialize(); /* initialize pheromone value */
5    foreach(ant in colony){/* colony has M ants */
6     ConstructSolution(alpha, beta);}
7    if(isFound) return; /* if solution is found, then end */
8    PheromoneUpdate(); /* update pheromone */
9    EvaporatePheromone(rho);}
```

Listing 1. The pseudocode of ACO algorithm.

Listing 1 shows the pseudocode of our ACO algorithm for the CES optimization problem. This code is executed as kernel in a thread for each ant colony. The number of ants in the colony is the algorithm parameter which determines the trade-off between the number of iterations and the breadth of the search at each iteration: the larger the number of ants per iteration, the fewer iterations are needed in ACO [24]. The local iteration counter is used by each thread as a nonstop operation protection (line 3): if ants in this thread cannot find the solution after `maxIterNumber` iterations (which is in principle possible for stochastic algorithms), then the thread terminates.

Up to now, most improvement work for ACO has concentrated on the tour construction and pheromone update. But there is also a question how to decide the termination condition of ACO algorithms in practice [29]. The possible variants of termination condition include: (1) the algorithm has found a solution within a predefined distance from a lower bound on the optimal solution quality; (2) a maximum number of tour constructions or a maximum number of algorithm iterations has been reached; (3) a maximum CPU time has been spent;

(4) the algorithm shows stagnation behaviour [29]. These variants have shortcomings: e.g., we may not know the optimal solution, so (1) will lose the effect in the algorithm, while (2) and (3) are often not economical [29]. We use a combination of termination variants (1) and (2); they are good in our case, because for CSP, it is clear when constraints are fulfilled and when not. According to recommendations in [28, 29] and our previous work we use `maxIterCount`=100.

The first potential candidate ACO parameter for tuning in Listing 1 could be the size `M` of the ant colony. However, different sizes of colonies would adversely affect the GPU-algorithm, because of divergent branches and memory operations that cause uncoalesced accesses or bank conflicts [5, 6]. The NVIDIA Streaming Multiprocessors (SMs) only get one instruction at a time and all CUDA cores execute the same instruction. Threads within a *warp* (a group of 32 threads, that are used in hardware to coalesce memory access and instruction dispatch) must execute the same instruction at each cycle. The most common code construct that can cause thread divergence is branching in an *if-then-else* statement: it can hurt performance due to a lower utilization of the processing elements, which cannot be compensated for via increased amount of parallelism [14]. To reduce this divergence, we use one value of M for all threads.

As confirmed by numerous experiments in previous work [24, 18, 22] and our own work [4], a good approximation for the number of ants in a colony is $M = 100$, so we use this value as default in all our experiments described in this paper.

We now turn to other tunable parameters of ACO which are the subject in this work. Ants in Listing 1 all behave in a similar way: every ant moves from the top of the tree-structured search space to the bottom. Once the ant selects a node $r = n_{i,j}$ at tree level i, it can pick the next child node $s = n_{i+1,j}$. The tour of an ant ends in the leaves of the tree (level I); each path corresponds to a potential solution of the problem. The ant transition from node r to s is probabilistically biased by two values: pheromone trail τ_{rs} and heuristic information η_{rs} as follows: $p_{rs} = \tau_{rs}^{\alpha} \cdot \eta_{rs}^{\beta} / \sum_{k \in C_r} (\tau_{rk}^{\alpha} \cdot \eta_{rk}^{\beta})$, where C_r is the set of child nodes for r [10, 27], and k are indices of these nodes. The evaporation (line 9 in Listing 1) is performed at a constant rate ρ at the end of each iteration. It allows the ant colony to avoid an unlimited increase of the pheromone value and to "forget" poor choices made previously [24]. We implement this by the assignment: $\tau_{rs} = \rho \cdot \tau_{rs}$, where $\rho \in [0, 1]$ is the trail persistence parameter. In calculating heuristic information, we make a unit which satisfies the constraint for the beginning part of the CES and larger main size more preferable than a unit with the unsatisfied compatibility constraint and smaller main size.

We observe that parameters α and β influence the pheromone value and heuristic value, respectively. They control the relative importance of the pheromone trails and the heuristic information, as we explain in the following. We use the following rule for the pheromone update : $\tau_{rs} = \tau_{rs} + Q / \sum_{m=1}^{M} L_m$, where Q is some constant and L_m is the tour length of the m-th ant, M is the swarm size. The smaller is the value of L_m the larger is the value added to the previous pheromone value. We use L_m as a fitness value that indicates how close is a given solution to achieving the required goals.

3 ACO Parameter Tuning

For our target applications, we solve the constraint satisfaction problem (CSP), rather than the optimization problem as in previous work. A specific feature of our CSP is that only the existence of a valid solution is required. The quality criterion is the frequency of feasible solutions for particular parameters values. We use offline tuning in terms of [24] to configure the ACO parameters used to solve the CSP. Our objective function is the algorithm run time. Since ACO is a probability-based algorithm, its results are different if run multiple times on the same instance of a problem, with varying run time. So, in order to achieve reliable results, we run each instance multiple times and take the average value.

3.1 Choosing parameters for tuning

In the case of CES, we tune the following three parameters of ACO.

The *Information Elicitation Factor* α reflects the importance of the pheromone accumulation with regard to the ants' path selection. If α is large, the ants tend to choose the same path as the preceding ants, resulting in a stronger cooperation among the ants [16]. Although the convergence speed of ACO in this case increases, it is likely for the algorithm to fall into a locally optimal solution, i.e. large α reduces the global search ability. Conversely, if α is small, the convergence speed of the ACO is slowed down, although of the fact that the global search ability of the algorithm can be improved.

The *Expected Heuristic Factor* β represents the relative importance of the mutual ants' visibility, i.e., it reflects the importance of the heuristic information with regard to the ants' path selection. If the value is very large, the probability of a state transition is close to that of a greedy algorithm. If β is small, the heuristic information has virtually no effect on the path selection, which may lead ACO to fall into stagnation or a local optimum.

The third parameter, *Pheromone Evaporation Rate* $\rho \in [0, 1]$ regulates the degree of the decrease in pheromone level in trails. If ρ is high (near to 1) then pheromone values will persist longer, while low values of ρ (near to 0) allow forgetting quickly of previous choices and, hence, allow faster adaptation to changes [24]. In other words, smaller ρ reduces the global search ability of ACO, is while larger, ρ improves this ability but limits the convergence speed.

3.2 Our tuning method: the idea

For tuning parameters α, β and ρ of parallel ACO, we use a statistical analysis of the experimental data obtained as a result of computational experiments on the CPU-GPU system. The application code for a CPU-GPU systems consists of a sequential code (*host* code executed on the CPU) that invokes hundreds or thousands of parallel threads on the *device* (GPU), where threads execute the *kernel* code shown in Listing 1. If some thread finds a solution of CSP then all threads finish their work. With an increasing number of threads, the probability of finding a solution increases, and, therefore, the search time is typically reduced.

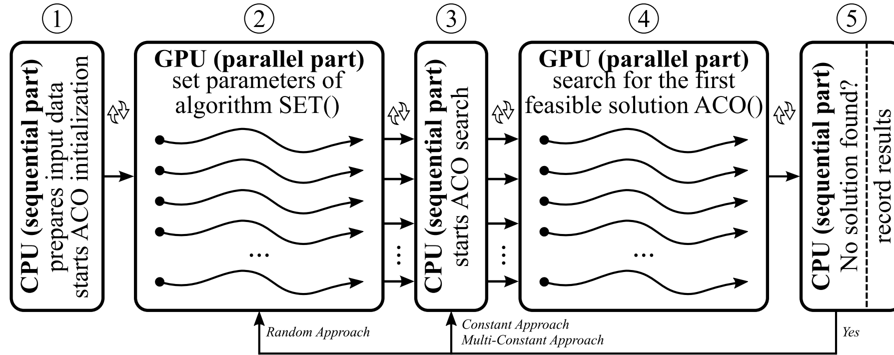
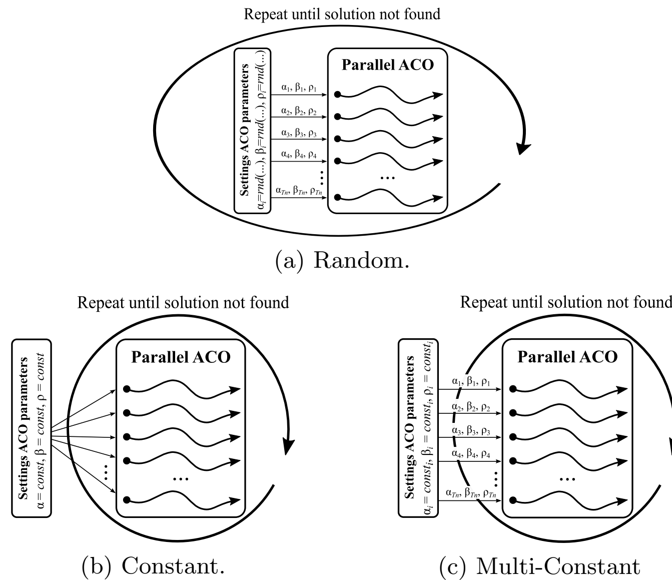


Fig. 2. General method of ACO parameter tuning.

Figure 2 shows the main steps of our tuning method, as follows: 1) CPU reads the input data (number of CES stages `I`, number of devices `J[I]`, production horizon `Tmax` etc.) and starts on the GPU the parameter initialization α, β and ρ; 2) GPU initializes ACO parameters by one of approaches described below; 3) GPU starts the kernel function of Listing 1; 4) the ACO kernel searches for the first feasible solution – the initial CES-variant; if some thread finds a solution then all threads finish their work; 5) a if solution is not found, repeat step 2 or 3 depending on the approach; otherwise CPU receives the obtained feasible solution and records results for further processing.



(a) Random.

(b) Constant.

(c) Multi-Constant

Fig. 3. Approaches for ACO-parameters tuning.

Figure 3 shows that within the general tuning method consisting of steps 1) – 5), we distinguish three particular approaches to parameter tuning *Random* 3a), *Constant Approach* 3b) and *Multi-Constant* 3c), as follows.

Random Approach In the Random Approach, algorithm parameters are initialized in step 2 by uniformly distributed random values from the intervals $[\alpha_a, \alpha_b]$, $[\beta_a, \beta_b]$, $[\rho_a, \rho_b]$. We set the bounds of these intervals as recommended in literature, e.g. [15]. We use high-performance, GPU-accelerated random number generator from NVIDIA's native cuRAND library (CUDA RAndom Number Generation) [20]. Function `curand_uniform()` returns a uniformly distributed value in the interval $(0.0, 1.0]$. We generate random numbers within a specified interval $(a, b]$ as follows: `rnd(a,b) = curand_uniform() * (b-a) + a`. So, the `SET()` function for the Random approach reads as in Listing 2.

```
1 __global__ void SET() {
2  ... /* obtaining thread identifier */
3  threadID = blockDim.x * blockIdx.x + threadIdx.x;
4  alpha[threadID] = rnd(alpha_a, alpha_b);
5  beta[threadID] = rnd(beta_a, beta_b);
6  rho[threadID] = rnd(rho_a, rho_b); ...}
```

Listing 2. Random Approach: the SET() pseudocode.

If a particular combination of parameter values produces a feasible solution then these values are saved for the further processing. This way, we obtain a set of triples of parameter values α, β and ρ, for which ACO finds feasible solutions. If a feasible solution in step 4 of the Random approach is not found after `maxIterNumber` iterations then the approach goes to step 2 of Figure 2.

Constant Approach The Constant approach, see Figure 3b) differs from the Random: all threads use the same values α, β and ρ for all threads (see Listing 3).

```
1 __global__ void SET() {
2  ... /* obtaining thread identifier */
3  threadID = blockDim.x * blockIdx.x + threadIdx.x;
4  alpha[threadID] = const_alpha;
5  beta[threadID] = const_beta;
6  rho[threadID] = const_rho; ...}
```

Listing 3. Constant approach: the SET() kernel pseudocode.

We obtain the starting values on the basis of a frequency analysis of the set of α, β and ρ obtained using, for example, the Random approach. In the following iterative process, we obtain a set of triples of parameter values α, β and ρ, for

which ACO finds feasible solutions. If a feasible solution is not found in step 2 after `maxIterNumber` iterations then we proceed to step 3 of Figure 2.

Multi-Constant Approach In the Multi-Constant Approach (see Figure 3c), all threads use different initial values of α, β and ρ for all threads.

```
1 __global__ void SET() {
2  ...  /* obtaining thread identifier */
3  threadID = blockDim.x * blockIdx.x + threadIdx.x;
4  alpha[threadID] = const_alpha[threadID];
5  beta[threadID] = const_beta[threadID];
6  rho[threadID] = const_rho[threadID]); ... }
```

Listing 4. Multi-constant approach: the SET() pseudocode.

In this case, parallel ACO algorithm with a Multi-Constant approach can be viewed as "learning" from the random parameter tuning, since only those triples of the parameters α, β and ρ are saved for which the solution was found. If the Multi-const approach does not find a feasible solution in `maxIterNumber` iterations then it goes to step 3.

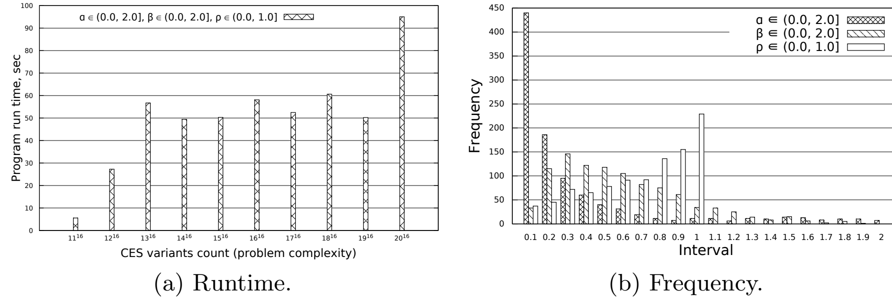
4 Experimental Evaluation

Our experiments are conducted on a hybrid system comprising: 1) a CPU: Intel Xeon Gold 5118, 12 cores with Hyper-Threading, 2.3 GHz with 192 GB RAM, and 2) a GPU: NVIDIA Tesla V100-SXM2-16GB with 80 multiprocessors, each with 64 CUDA cores (total 5 120 CUDA cores), GPU max clock rate 1.53 GHz, 16 GB of global memory. We use CentOS Linux release 7.5.1804, NVIDIA Driver version 410.72, CUDA version 10.0 and GNU C++ Compiler version 6.4.0. On the GPU we employ 5 120 threads as the number of CUDA cores for Tesla v100.

As our test case, we evaluate the use of ACO for designing a CES consisting of 16 processing stages with 11 to 20 variants of devices at every stage (in total from $11^{16} \approx 10^{17}$ up to $20^{16} \approx 10^{21}$ CES variants). Note that this size is significantly larger than was possible in our previous work [4] without parameter tuning.

In the experiments, for each size of the problem from 11^{16} to 20^{16} (total 10 series of experiments), the algorithm is launched 100 times. For each launch, the run time for finding a feasible solution is measured, the average run time is calculated, and the corresponding values of α, β and ρ are recorded.

Random approach For the first series of experiments, values of α and β are set using a random uniform distribution in range $(0, 2]$, and ρ in range $(0, 1]$, as recommended in [15]. After the entire series of experiments, we obtain $10 \cdot 100 = 1\,000$ triples of values α, β and ρ with the problem sizes for which solutions were found. The total run time spent by the GPU for the first series of experiments with the Random approach is $50\,575\,\text{sec} \approx 14\,\text{hr}$.

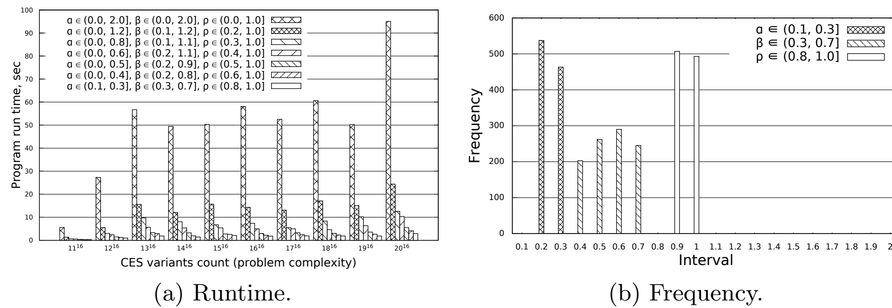


(a) Runtime. (b) Frequency.

Fig. 4. Random approach for ACO-parameters tuning. First iteration.

Figure 4a) shows the average run time of solving the optimization problem depending on the problem size. Figure 4b) shows the frequency of the found feasible solutions represented as histograms for different intervals/ranges of parameter values. We observe from the histograms that $\approx 90\%$ of the solutions are obtained when the ACO parameters are in the intervals: $\alpha \in (0.0, 1.2]$, $\beta \in (0.1, 1.2]$, $\rho \in (0.2, 1.0]$.

Our idea is to stepwise reduce the intervals for α, β, ρ by moving to values where feasible solutions are more frequent. Therefore, we repeat the procedure as above to obtain new 1 000 solutions with the reduced parameter ranges, and again analyze the frequency of solutions. We repeat this process (search for 1 000 solutions – frequency analysis – correction of ACO-parameters intervals) altogether 7 times. After these 7 repetitions, the parameter ranges narrow to a single point: $\alpha = 0.2$, $\beta = 0.5$, $\rho = 0.9$. We use it as the ACO parameter values for Constant Approach in Subsection 4.



(a) Runtime. (b) Frequency.

Fig. 5. Random Approach for ACO-parameters tuning.

Figure 5a) shows the change in the average run time of the algorithm for problems of various dimensions 11^{16} to 20^{16}. Figure 6 shows the run time for

the parameter triple $\alpha = 0.2$, $\beta = 0.5$, $\rho = 0.9$, together with other results for the Constant approach. Figure 5b) shows the frequency achieved after the final, 7th iteration of experiments for $\alpha \in (0.1, 0.3]$, $\beta \in (0.3, 0.7]$, $\rho \in (0.8, 1.0]$.

Summarizing the results achieved by the Random approach in our experiments, we can conlclude that, due to the parameter tuning, the average run time of the algorithm decreased by ≈ 29 times (from ≈ 50 sec for $\alpha \in (0.0, 2.0]$, $\beta \in (0.0, 2.0]$ and $\rho \in (0.0, 1.0]$ to ≈ 1.7 sec for $\alpha \in (0.1, 0.3]$, $\beta \in (0.3, 0.7]$, $\rho \in (0.8, 1.0]$). The total tuning time spent by the GPU for all seven iterations of experiments was 83 116 sec ≈ 23 hr. In the sequel, we use thus obtained values of α, β, ρ for problems of different size as initial values in the Constant and Multi-constant approach.

Constant approach In the Constant approach, each GPU thread uses the same triple of values α, β, ρ that was obtained by the Random approach.

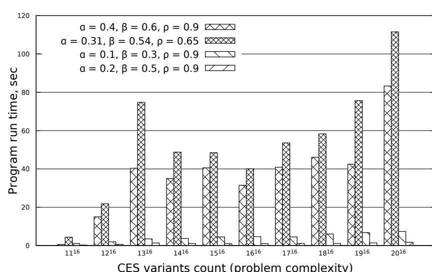


Fig. 6. Constant approach: Run time depending on the problem size.

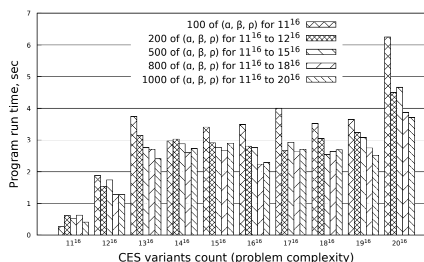


Fig. 7. Multi-constant Approach: Run time depending on the problem size.

Figure 6 shows the results. The first triple of parameter values $\alpha = 0.4$, $\beta = 0.6$, $\rho = 0.9$ are the same which we empirically used in our previous articles [3, 4] – we present them here for comparison.

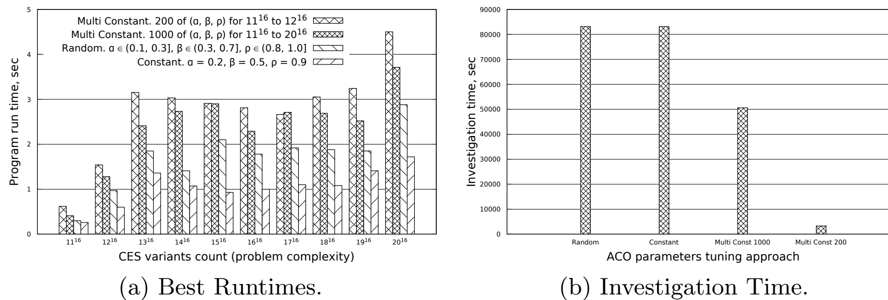
As we described in the previous subsection, after the first series of experiments with the Random approach we obtained first 1000 triples of α, β, ρ values. An interesting finding of our experiments is that, although it may seem intuitively apparent that the average values of the found parameter values would serve as good candidates for parameter values, this hypothesis was not confirmed. Indeed, the second triple in Figure 6 ($\alpha = 0.31$, $\beta = 0.54$, $\rho = 0.65$) is calculated as the average values of the parameter intervals obtained after the first iteration of the Random approach. We observe in the figure that these values seriously worsened the run time of the algorithm by ≈ 1.4 times.

The third triple $\alpha = 0.1$, $\beta = 0.3$, $\rho = 0.9$ in Figure 6 is set based on the analysis of the data frequency shown in Figure 4b) after the first iteration of the Random approach (we take the values that provide the highest frequency). This reduces the run time by ≈ 8.5 times. The best, fourth triple of values $\alpha = 0.2$, $\beta = 0.5$, $\rho = 0.9$ in Figure 6 is obtained after seven iterations of the Random approach. This triple reduces the run time of the algorithm by ≈ 35 times. The tuning time spent by the GPU is the same 23 hr as in the Random approach.

Multi-constant approach The Multi-constant approach differs from the previously discussed Constant approach in that each GPU thread uses its own triple of constants α, β, ρ that are obtained as a result of the first series of the Random approach. The approach yields a total of 1 000 triples of ACO-parameters for problems of various sizes from 11^{16} up to 20^{16}, for which feasible solutions are obtained. When running the program on the GPU (for our case on the Tesla v100 we use 5 120 threads, which corresponds to the number of CUDA cores for this GPU-model), the initialization of the algorithm parameters is performed cyclically. If we have the set of 100 triples of algorithm parameters, then on the GPU, after every 100 GPU threads, the values of algorithm parameters will be repeated, for set of 200 triples repetition values will be every 200 GPU threads, etc. For set of 1 000 triples of parameters, the values of the algorithm parameters will be repeated on every 1 000 GPU threads.

Figure 7 shows the results of using the Multi-const approach. The worst run time (especially for the maximum problem complexity 20^{16}) corresponds to the set of 100 triples, the best run time corresponds to the set of 1 000 triples. The average run time of the algorithm starting from the set of 200 triples differs from the runtime of the algorithm for the maximum set of 1 000 triples by only a factor of ≈ 1.16 (2.75 sec vs. 2.37 sec). The search time for the set of 1 000 triples is equal to the time of the first iteration of random approach with $\alpha \in (0, 2]$, $\beta \in (0, 2]$ and $\rho \in (0, 1]$ is 50 575 sec $\approx$ 14 hr, and the search time for the set of 200 triples with the same random approach for a problem complexity $11^{16} + 12^{16}$ is 3 282 sec $\approx$ 54 min, which is 15.4 times faster.

Comparison of tuning approaches Figure 8 compares the best run time results obtained by each of our three tuning approaches. For the Multi-constant approach, we compare also to the variant with 200 triples that provides still acceptable results achieved in a significantly shorter tuning time.



(a) Best Runtimes. (b) Investigation Time.

Fig. 8. Comparison of Approaches.

We observe in Figure 8a) that the fastest run time is achieved by using the Constant approach with constant parameter values $\alpha = 0.2$, $\beta = 0.5$, $\rho = 0.9$ (average execution time $\approx$ 1.1 sec). Figure 8b) shows that the tuning time for the most economical Multi-const approach with 200 samples ($\approx$ 54 min) is 25 times shorter, while the resulting run time of the optimization process is only $\approx$ 2.5 times slower.

5 Conclusion

Our contribution is a new set of three approaches to parameter tuning of the GPU-parallelized Ant Colonies Optimization (ACO) metaheuristic. The advantage of our approaches is that they work for different metaheuristics and different optimization problems. As a particular demonstration, this paper describes the use case when ACO is used for solving the Constraint Satisfaction Problem (CSP) in the process of optimizing the multiproduct batch plants design. Our three tuning approaches – Random, Const and Multi-Const – proceed by using a statistical analysis of solution frequences in particular intervals of parameter values. By stepwise narrowing these intervals, we arrive at the intervals or even single parameter values that provide good soltions in short time.

Using modern high-performance CPU-GPU systems, it is possible to conduct a large number of computational experiments (e.g., overnight), and to use their results for a statistical frequency analysis. We demonstrate that the user can choose between longer experiments with a very good quality of solutions and shorter experiments that still provide a acceptable level of quality. This shows that it is possible to use the parameters values obtained for problems of a small complexity for solving problems of a large complexity. It should be noted that despite the relatively short time of the algorithm (minutes) without tuning on high-performance equipment (Tesla v100), the values of the ACO parameters obtained as a result of our approach can be applied for a different equipment (e.g., Tesla k20s), as well as when implementing a sequential version of the algorithm on the CPU, since ACO parameter values are device-independent. ACO is a stochastic algorithm. On the one hand, the α, β, ρ parameters of the algorithm are not associated with a specific implementation, so they are architecture-independent. On the other hand, with an increase in the number of threads, the probability of finding a solution, and, consequently, the speed of the algorithm, increases. Therefore, improving the implementation for a particular target architecture allows to additionally increase the speed of finding the solution.

While the Const Approach achieves eventually the best performance, it requires the most investigation time due to multiple repetitions of Random Approach with narrowing of the parameter value intervals. MultiConst approach can significantly reduce the investigation time, but its results are applicable only for the parallel implementation of the algorithm. Our approach can be used for tuning other metaheuristic algorithms and for other applied problems based constraint satisfiability.

Acknowledgements

We are grateful to the anynymous reviewers for their very helpful comments, and to the Nvidia Corp. for the donated hardware used in our experiments. This work was supported by the DAAD (German Academic Exchange Service) and by the Ministry of Education and Science of the Russian Federation under the "Mikhail Lomonosov II"-Programme, and by the HPC2SE project of BMBF (Federal Ministry of Education and Research, Germany).

References

1. Barbosa, E., Senne, E.: Improving the Fine-Tuning of Metaheuristics: An Approach Combining Design of Experiments and Racing Algorithms. Journal of Optimization **2017**, 1–7 (Feb 2017). https://doi.org/10.1155/2017/8042436
2. Birattari, M.: Tuning Metaheuristics, Studies in Computational Intelligence, vol. 197. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00483-4
3. Borisenko, A., Gorlatch, S.: Parallelizing Metaheuristics for Optimal Design of Multiproduct Batch Plants on GPU. In: Lecture Notes in Computer Science, vol. 10421 LNCS, pp. 405–417. Springer Nature (Feb 2017). https://doi.org/10.1007/978-3-319-62932-2_39
4. Borisenko, A., Gorlatch, S.: Comparing GPU-parallelized metaheuristics to branch-and-bound for batch plants optimization. The Journal of Supercomputing pp. 1–13 (Jul 2018). https://doi.org/10.1007/s11227-018-2472-9
5. Borisenko, A., Haidl, M., Gorlatch, S.: A GPU parallelization of branch-and-bound for multiproduct batch plants optimization. The Journal of Supercomputing **73**(2), 639–651 (Feb 2017). https://doi.org/10.1007/s11227-016-1784-x
6. Burtscher, M., Nasre, R., Pingali, K.: A quantitative study of irregular programs on GPUs. In: 2012 IEEE International Symposium on Workload Characterization (IISWC). pp. 141–151. IEEE (Nov 2012). https://doi.org/10.1109/IISWC.2012.6402918, http://ieeexplore.ieee.org/document/6402918/
7. Castillo, O., Neyoy, H., Soria, J., Melin, P., Valdez, F.: A new approach for dynamic fuzzy logic parameter tuning in Ant Colony Optimization and its application in fuzzy control of a mobile robot. Applied Soft Computing **28**, 150–159 (Mar 2015). https://doi.org/10.1016/j.asoc.2014.12.002
8. Chen, C.C., Liu, Y.T.: Enhanced Ant Colony Optimization with Dynamic Mutation and Ad Hoc Initialization for Improving the Design of TSK-Type Fuzzy System. Computational Intelligence and Neuroscience **2018**, 1–15 (Jan 2018). https://doi.org/10.1155/2018/9485478
9. Delévacq, A., Delisle, P., Gravel, M., Krajecki, M.: Parallel Ant Colony Optimization on Graphics Processing Units. Journal of Parallel and Distributed Computing **73**(1), 52–61 (Jan 2013). https://doi.org/10.1016/j.jpdc.2012.01.003
10. Dorigo, M., Birattari, M.: Ant colony optimization. In: Encyclopedia of machine learning, pp. 36–39. Springer (2011). https://doi.org/10.1007/978-1-4899-7687-1_22
11. Dorigo, M., Stützle, T.: Ant Colony Optimization: Overview and Recent Advances. In: Handbook of metaheuristics, pp. 311–351. Springer (Sep 2018). https://doi.org/10.1007/978-3-319-91086-4_10
12. Fallahi, M., Amiri, S., Yaghini, M.: A parameter tuning methodology for metaheuristics based on design of experiments. International Journal of Engineering and Technology Sciences **2**(6), 497–521 (Dec 2014)
13. Gómez-Cabrero, D., Ranasinghe, D.N.: Fine-tuning the Ant Colony System Algorithm through Particle Swarm Optimization. arXiv preprint arXiv:1803.08353 (Mar 2018)
14. Han, T.D., Abdelrahman, T.S.: Reducing branch divergence in GPU programs. In: Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units - GPGPU-4. pp. 1–3. ACM, ACM Press, New York (Mar 2011). https://doi.org/10.1145/1964179.1964184

15. Khan, S., Bilal, M., Sharif, M., Sajid, M., Baig, R.: Solution of n-Queen problem using ACO. In: 2009 IEEE 13th International Multitopic Conference. pp. 1–5. IEEE (Dec 2009). https://doi.org/10.1109/INMIC.2009.5383157
16. Li, P., Zhu, H.: Parameter Selection for Ant Colony Algorithm Based on Bacterial Foraging Algorithm. Mathematical Problems in Engineering **2016**, 1–12 (2016). https://doi.org/10.1155/2016/6469721, https://www.hindawi.com/journals/mpe/2016/6469721/
17. Mahi, M., Baykan, Ö.K., Kodaz, H.: A new hybrid method based on Particle Swarm Optimization, Ant Colony Optimization and 3-Opt algorithms for Traveling Salesman Problem. Applied Soft Computing **30**, 484–490 (May 2015). https://doi.org/10.1016/j.asoc.2015.01.068
18. Maier, H.R., Simpson, A.R., Zecchin, A.C., Foong, W.K., Phang, K.Y., Seah, H.Y., Tan, C.L.: Ant colony optimization for design of water distribution systems. Journal of water resources planning and management **129**(3), 200–209 (2003)
19. NVIDIA Corporation: CUDA C programming guide 10.0 (Oct 2018), http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
20. NVIDIA Corporation: The NVIDIA CUDA random number generation library (cuRAND) (Dec 2018), https://developer.nvidia.com/curand
21. Olivas, F., Valdez, F., Castillo, O.: Dynamic parameter adaptation in Ant Colony Optimization using a fuzzy system for TSP problems. In: IFSA-EUSFLAT. pp. 765–770 (2015)
22. Simpson, A., Maier, H., Foong, W., Phang, K., Seah, H., Tan, C.: Selection of parameters for ant colony optimization applied to the optimal design of water distribution systems. In: Proc., int. congress on modeling and simulation, Canberra, Australia. pp. 1931–1936 (2001)
23. Skakov, E.S., Malysh, V.N.: Parameter meta-optimization of metaheuristics of solving specific NP-hard facility location problem. Journal of Physics: Conference Series **973**, 012063 (Mar 2018). https://doi.org/10.1088/1742-6596/973/1/012063
24. Stützle, T., López-Ibáñez, M., Pellegrini, P., Maur, M., Montes de Oca, M., Birattari, M., Dorigo, M.: Parameter Adaptation in Ant Colony Optimization, pp. 191–215. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21434-9_8
25. Trindade, Á.R., Campelo, F.: Tuning metaheuristics by sequential optimization of regression models. arXiv preprint arXiv:1809.03646 pp. 1–22 (Sep 2018)
26. Tsang, E.: Foundations of constraint satisfaction: the classic text. BoD–Books on Demand (May 2014)
27. Valadi, J., Siarry, P.: Applications of Metaheuristics in Process Engineering. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-06508-3, http://link.springer.com/10.1007/978-3-319-06508-3
28. Veluscek, M., Kalganova, T., Broomhead, P.: Improving ant colony optimization performance through prediction of best termination condition. In: 2015 IEEE International Conference on Industrial Technology (ICIT). pp. 2394–2402. IEEE, IEEE (Mar 2015). https://doi.org/10.1109/icit.2015.7125451
29. Zhang, Z., Feng, Z., Ren, Z.: Approximate termination condition analysis for ant colony optimization algorithm. In: 2010 8th World Congress on Intelligent Control and Automation. pp. 3211–3215. IEEE, IEEE (Jul 2010). https://doi.org/10.1109/wcica.2010.5554984