Efficient GPU-Parallelization of Batch Plants Design Using Metaheuristics with Parameter Tuning

Andrey Borisenko[a], Sergei Gorlatch[b]

[a] *Tambov State Technical University, Russia*
[b] *University of Muenster, Germany*

Abstract

We address a practice-relevant optimization problem: optimizing multi-product batch plants, with a real-world use case study – optimal design of chemical-engineering systems. Our contribution is a novel approach to parallelizing this optimization problem on GPU (Graphics Processsing Units) by combining two metaheuristics – Simulated Annealing (SA) and Ant Colony Optimization (ACO). We improve the implementation performance by tuning particular parameters of the ACO metaheuristic. Our tuning approach improves on the previous methods in two respects: 1) we do not have to rely on additional mechanisms like fuzzy logic or algorithms for online tuning; and 2) we use the high computation performance of GPU to speedup the tuning process. By parallelizing the tuning process on modern GPUs, we allow the user to experiment with large volumes of input data and find the optimal values of the ACO parameters in feasible time. Our experiments on NVIDIA GPU show the efficiency of our approach to parameter tuning for the ACO metaheuristic.

Keywords: metaheuristics, metaheuristics parameter tuning, Ant Colony Optimization tuning, combinatorial optimization, batch plant design, GPU computing
2000 MSC: 68R05; 68T20; 68U01; 90C27

1. Motivation and related Work

Optimal design of multiproduct batch plants is a practice-relevant optimization problem. In this paper, we conduct a case study of designing *Chemical-Engineering Systems* (CES), which consist of a set of equipment units (tanks, filters, reactors, etc.) manufacturing a variety of products. The problem is to determine, for the given input (production horizon, product demand, etc.), the number and main sizes of technological units to achieve the lowest possible capital and operating cost while meeting the design and production objectives.

Email addresses: `borisenko@mail.gaps.tstu.ru` (Andrey Borisenko), `gorlatch@wwu.de` (Sergei Gorlatch)

Combinatorial optimization [1] is, in practical cases, very time-consuming due to the "combinatorial explosion" – the number of combinations to be examined grows exponentially. Because of problem's high computational complexity, this paper addresses parallel implementation on high-performance systems comprising CPU and GPU (Graphics Processing Units). We achieve this two-fold: (1) we use metaheuristics that do not guarantee that the found solution is optimal, but solutions of good (and in practice acceptable) quality are obtained faster than when using precise methods, and (2) we further accelerate metaheuristics by means of tuning their parameters.

The performance of metaheuristics depends upon the suitable selection of parameters chosen by the user. Finding the most suitable values for the parameters (fine tuning) is a challenging problem [2]. The previous work on tuning metaheuristic parameters follows one of two directions: online or offline methods [3, 4, 5]. A general online parameter adaptation method based on estimating algorithm's performance and gradient search in the parameter domain is presented in [6]. A combination of statistical and AI mechanisms is proposed in [7], while [8] suggests using fuzzy logic for adapting parameters. [9] is a comprehensive survey of automatic parameter tuning methods for metaheuristics. Parameter tuning utilizes the design of experiments in [10] and swarm optimization in [11]; sequential optimization of perturbed regression models is used in the tuning framework of [12]. In [13], parameters of a metaheuristic are found by an evolutionary metaheuristic. An ad hoc initialization and dynamic mutation that together improve the accuracy of ACO are proposed in [14].

In this paper, we make two main contributions: (a) a GPU-parallelized approach to solving the plant optimization problem by combining two metaheuristics: ACO and SA, and (b) a new approach to parameter tuning in ACO for the CSP (Constraint Satisfaction Problem) part of the optimization problem. Since we tune parameters not for the optimization problem as in [15, 8] but rather for CSP, this allows us to use frequency analysis that is not dependent on any specific information, like fuzzy logic, algorithms for online-tuning, etc. [16, 14, 8]. Due to the use of modern GPUs on the parallelized algorithm, we can run many experiments in order to accumulate statistical data in a reasonable amount of time. Another positive feature is the possibility to use the tuned parameters for either parallel or sequential algorithm version.

The paper is organized as follows. Section 2 describes our CES optimization use case. In Section 3 we describe our use of metaheuristics for optimizing multiproduct batch plants, with a specific use case of CES. In our hybrid approach, we combine two metaheuristics: *Simulated Annealing* (SA) and *Ant Colony Optimization*. In Section 4, we further accelerate the GPU-parallelized ACO method by means of ACO's parameter tuning using statistical analysis. Finally, we summarize our contributions.

2. The Batch Plant Optimization: Problem Formulation

We provide here a rather intuitive description of the batch plant optimization problem for a particular use case of CES; a detailed formulation is given in [17].

A CES consists of equipment units organized in I processing stages: the i-th stage has set X_i of equipment units, while J_i is the amount of units' variants in X_i. We write $\Omega_e, e = \overline{1, E}$ for a CES variant, where $E = \prod_{i=1}^{I} J_i$ is the number of system variants.

In Figure 1 a simplified CES with 4 stages is shown, where each stage has two devices ($J_i = 2$); there are altogether $2^4 = 16$ variants of such a system.

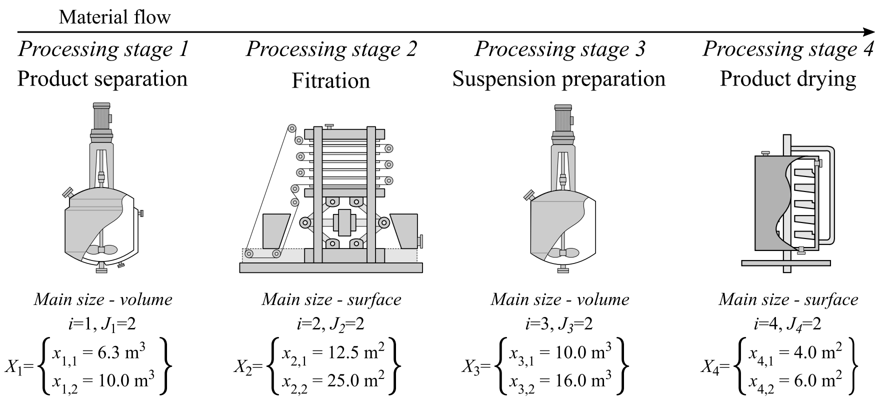

Material flow
Processing stage 1
Product separation
Processing stage 2
Fitration
Processing stage 3
Suspension preparation
Processing stage 4
Product drying
Main size - volume
i=1, J_1=2
$X_1=\left\{ x_{1,1} = 6.3 \text{ m}^3, x_{1,2} = 10.0 \text{ m}^3 \right\}$
Main size - surface
i=2, J_2=2
$X_2=\left\{ x_{2,1} = 12.5 \text{ m}^2, x_{2,2} = 25.0 \text{ m}^2 \right\}$
Main size - volume
i=3, J_3=2
$X_3=\left\{ x_{3,1} = 10.0 \text{ m}^3, x_{3,2} = 16.0 \text{ m}^3 \right\}$
Main size - surface
i=4, J_4=2
$X_4=\left\{ x_{4,1} = 4.0 \text{ m}^2, x_{4,2} = 6.0 \text{ m}^2 \right\}$

Figure 1: A Simple Chemical-Engineering System (CES): An Example

The optimization goal is – for the input data of the product demand, production horizon, and available equipment set – to compute the optimal units' number and their sizes at particular processing stages. Each system variant has to be operable and meet the processing time constraint.

Because of the "combinatorial explosion" – the number of combinations to be examined grows exponentially – even the fastest computers may require an intolerable time: e.g., if each process stage can be equipped with devices of 5 to 20 standard main sizes, then a CES consisting of 16 stages yields the total number of choices is 5^{16} up to 20^{16} (which is approximately 10^{11} to 10^{21}). This requires an acceleration which is our topic in the remainder of this paper.

3. A Parallel Metaheuristic Approach

Our approach to accelerating the solution of the CES optimization problem is to use metaheuristics and to develop a fast implementation of the metaheuristics using highly-parallel architectures, in particular Graphics Processing Units (GPU).

A heuristic algorithm for an optimization problem considers the most likely, rather then all possible states of the problem. Specific heuristics are usually problem-dependent. A *metaheuristic* is an algorithmic pattern for finding high-quality solutions for an optimization problem [3] by compromising between both global and local search. Randomization is used in metaheuristics in order to avoid local search and rather conduct search on the global scale, by efficiently exploring the space and reducing the size of the space. Experience shows that metaheuristic algorithms often converge to decent solutions in a reasonable amount of time, while the optimality of solutions is not guaranteed [18].

3.1. Combined Metaheuristic

In this section, we describe our approach combining two popular metaheuristics: *Simulated Annealing* (SA) and *Ant Colony Optimization* (ACO).

The ACO metaheuristic describes a multi-agent system: the agents called ants achieve a global goal [18] by interacting with each other. ACO is inspired by the life of an ant colony: when moving from the source of food to the joint nest, ants leave on their paths a substance called *pheromone*, which is employed for interaction between ants. Pheromone helps in finding the shortest path between a nest and food, because ants follow, with particular probability, the path labeled with pheromone.

The SA metaheuristic is very popular [18]: it finds an optimal solution in many application areas and generally finds solutions of good quality. SA employs random search in which it may accept non-ideal changes in order to avoid local optima and thus eventually converge to a good-quality global solution.

We combine in our approach ACO and SA because, before using SA, we need a feasible initial solution, for finding which we use ACO. Unlike classical optimization problems that use a random initial solution, our CES optimization problem cannot accept random initialization, because of the constraints that have to be satisfied. The search for an initial solution can be considered as a *Constraint Satisfaction Problem* (CSP) [19]. We solve this CSP by using the ACO metaheuristic that was shown to yield good-quality results for numerous practical cases of CSP [20].

3.2. The Ant Colony Optimization (ACO) Metaheuristic

In Listing 1, we demonstrate the basic structure of the ACO algorithm for solving our optimization problem. This code runs as an individual thread for every colony of ants. We parametrize the algorithm by the number of ants, as a compromise between the search breadth at each iteration and the iterations number: we need fewer iterations for a larger ants colony. In each thread, we use the local iteration counter as a non-stop operation protection (line 5): the thread terminates if it does not find a suitable solution after the maximum number of iterations, which is possible with randomized algorithms.

Each ant in Listing 1 moves in the tree-like space from top to bottom. After an ant selects a node at some tree level, it picks then a next child node at the same level. In the leaves, ant's movement terminates, so each path in the tree stands for a possible solution of the problem. The ant's transition from node r to s is impacted by two variables: heuristic information η_{rs} and pheromone trail τ_{rs}: $p_{rs} = \tau_{rs}^{\alpha} \cdot \eta_{rs}^{\beta} / \sum_{k \in C_r} (\tau_{rk}^{\alpha} \cdot \eta_{rk}^{\beta})$, where C_r are children of r [18], and k are children indices. Parameters α and β determine the pheromone and heuristic information, respectively.

In line 14 of Listing 1, we perform the pheromone evaporation in each iteration at a constant rate ρ, such that the pheromone value cannot become

```
AntColonyOptimization() {
 iterationCounter = 0; /* initialize iteration counter */
 Q = I + 1; /* initialize maximal fitness value */
 isFoundFlag = false;
 while(!isFoundFlag && iterationCounter < maxIterationNumber){
  /* repeat while solution not found */
  PheromoneInitialize();
  foreach(ant in colony){/* iteration on ants colony */
   Solution[ant] = ConstructSolution(alpha, beta);
   L[ant] = EvaluateSolutionFitness(Solution[ant]);}
  isFoundFlag = CheckSolutions(Solution, L, Q);
  if(isFoundFlag) return; /* stop if solution is found */
  UpdatePheromone(L, Q); /* update value of pheromone */
  EvaporatePheromone(rho); /* pheromone evaporation */
  iterationCounter++;}
```

Listing 1: ACO Algorithm: Pseudocode.

too high, and we also "forget" previous choices of poor quality. We code it as follows: $\tau_{rs} = \rho \cdot \tau_{rs}$, where $\rho \in [0, 1]$ is called a persistence parameter. Parameters α and β in our code impact both the pheromone and the heuristic value, and they determine their relative importance. Pheromone is updated as follows: $\tau_{rs} = \tau_{rs} + Q/\sum_{m=1}^{M} L_m$, with constant Q, while M is the swarm size and L_m - the path length for m-th ant: for a shorter path we update the pheromone stronger. Fitness function L_m specifies how near is a solution to the required goal. When computing heuristic values, a unit with a larger main size satisfying the constraint for the beginning part of the CES is preferred to a unit of smaller size or with unsatisfied compatibility constraint.

Listing 2 shows the calculation o fitness: we compute the stages number in the beginning of CES, i.e., for which devices at stages 1 to i (lines 5 – 6) the constraints are satisfied. The fitness value is 0 (minimal) if not a single constraint is satisfied; the value is $I + 1$ (maximal) if, i.e., ll constraint are satisfied, i.e., the problem is solved.

```
1  ...
2   L[m] = SatisfiedRestrictions(W[m]) + (T(W[m]) <= Tmax ? 1 : 0);
3  ...
4  int SatisfiedRestrictions(W) {counter = 0;
5   for (i = 1; i <= I; i++){ /* check constraints */
6    if(PartialSolutionRestrictions(W, i) == 0) counter++; }
7   return counter;}
```

Listing 2: Fitness Value Calculation in ACO: Pseudocode.

Here, `W` is the problem solution, function `SatisfiedRestrictions()` checks the *compatibility constraint* (CES has to be operable, i.e., it must satisfy the joint action conditions for all processing stages), function `T()` denotes the production rate, `T_max` is the total available production time (so-called horizon). The production rate has to be less or equal to the horizon (*processing time constraint*).

3.3. Simulated Annealing (SA)

The SA algorithm performs a search in the solution space and converge to the best solution by gradually restricting the space. An annealing parameter called "temperature" (in analogy to the process in nature) influences how the search proceeds, in particular a solution of lower quality can be accepted with some probability: this enables avoiding local optima.

Our SA implementation in Listing 3 is structured in two loops. The solution at stage i is `W[i]`; it is initialized by a solution `W_init` provided by ACO, as explained above.

```
1  SimulatedAnnealing() {
2   t = T_init; W = W_init;/* initialize temperature and guess */
3   while(t > T_final){/* loop until t don't reaches T_final */
4    for(l = 0; l < L_max; l++){/* repeat L_max times */
5     W_cand = Permutation(W);/* generate neighboring solution */
6     /* check compatibility and processing time constraint */
7     if (SolutionRestrictions(W_cand) == 0 && T(W_cand) <= T_max){
8      delta_Cost = SolutionCost(W_cand) - SolutionCost(W);
9      if (delta_Cost < 0){ /* if candidate solution is better */
10      W = W_cand; } /* accept it as new solution unconditionally */
11     else { /* accept new solution conditionally */
12      r = rand(0, 1); /* get a random number in the range (0, 1) */
13      p = exp(-delta_Cost / t); /* calculate probability */
14      if (p > r){ /* check acceptance of the candidate solution */
15       W = W_cand; }}}} /* accept the new solution */
16   t = sigma * t; }} /* lower the temperature t */
```

Listing 3: SA Algorithm: Pseudocode.

The outer loop of SA (line 3) decreases the probability of non-improving solutions in the inner loop (line 4); the latter loop is searching for a neighbouring solution. To generate a new candidate solution `W_cand`, we use special procedure `Permutation()` (Listing 4). Due to randomly generating neighbours and allowing with some probability a solution that worsens the objective function, our SA implementation avoids local optima.

```
Permutation(W) {
 rndStage = (int)rand(1, I); /* take random stage */
 W[rndStage] = (int)rand(1, J[rndStage]);/* take random unit */
 return W; }
```

Listing 4: SA Implementation: Generation of a New Candidate Solution.

At each iteration in Listing 4, procedure `Permutation()` computes a neighbouring solution. We randomly take some CES stage i for generating a new solution, and we also choose randomly the unit number j from set J_i. The iterations in SA run `L_max` times, where `L_max` is the equipment set size. We take a random unit in a random stage at each iteration, and we permute the feasible solution. At the end of each iteration, the temperature value `t` is decreased by means of a cooling procedure (line 3); we use the cheapest price of a unit for `T_final`. Effective cooling rule is essential for reducing the time needed by the algorithm. In (line 16) we apply the fast cooling rule $t = \sigma \cdot t$ [18]; we choose $0.8 \leq \sigma \leq 0.99$, following the recommendation in [21].

3.4. Parallel Implementation of the Combined Metaheuristic

Our combined (ACO+SA) parallel metaheuristic works on a system consisting of a CPU (host) and a GPU. The implementation code starts as the host code for the CPU, which calls multiple instances of the kernel code that run as parallel threads on the GPU.

There are 5 steps in our combined metaheuristic implementation, as follows:

1. The host takes the input data, prepares the SA and ACO parameters, and starts multiple instances of the ACO kernel on the GPU .
2. The GPU-Kernel for ACO finds an initial CES-variant that satisfies the problem's contraints. We employ the approach with multiple ant colonies that independently run on parallel threads, until one thread finds a feasible solution. The more threads we employ, the shorter is the time for finding a solution.
3. CPU takes the obtained initial solution, partitions it among the GPU threads as a start point for SA, and executes the SA kernel on the GPU.
4. The kernel for SA on the GPU looks in parallel for the optimal solution. Note that we aim not at reducing the iteration runtime, but rather at increasing the grade of parallelism in the execution of iterations: we increase the probability of converging to the global optimum, even when the initial solution is the same in all threads. The more threads we start, the higher is the chance for some thread to find an almost optimal solution.
5. CPU takes the solutions obtained by the SA GPU-threads, chooses the best of them and accepts it as the final solution of the problem.

On a parallel system comprising a CPU and a GPU, the CPU (host) takes the input data from a file, sends them to the GPU and starts there the ACO kernel function. To improve performance, CUDA kernels are launched asynchronously: after starting the kernel the control is immediately returned back to the CPU. We use `cudaDeviceSynchronize()` for synchronization: the host waits till the termination of the GPU kernel and obtains the result from it.

In our ACO program, threads simulate multiple ant colonies. At start, we assign small random values for pheromones. Threads are controlled by a local iteration counter and a global flag which is updated using `atomicAdd()` if the corresponding thread found a suitable solution. Each thread uses the local iteration counter as a non-stop operation protection: if thread's ants cannot find a solution after the specified number of iterations then the thread stops.

The implementation of SA consists of two loops. The outer loop decreases the temperature and thus reduces the probability of taking non-improving neighbouring solutions, search for which is performed in the inner loop. Note that our SA avoids local optimum in that it generates neighbors randomly and, with some probability, accepts solutions worsening the objective function; the probability of such acceptance decreases with lower temperature. Therefore, SA conducts a wide search at the beginning, and then it gradually restricts the solution space while converging to the optimal solution.

4. Parameter Tuning for the ACO Metaheuristic

The goal in this section is to improve the parallel ACO implementation by using the additional performance potential via tuning particular parameters of ACO. We employ offline parameter tuning according to the classification of [5]. The tuning task is to configure the ACO parameters, such that the CSP is solved as fast as possible. Our idea of improvement by tuning is to find the intervals of the parameter values in which the feasible solutions of the optimization problem (i.e., solutions that satisfy both compatibility and processing time constraints) are contained more frequently. As demonstrated in the next section, our tuning approach finds comparatively small intervals of parameter values, such that these intervals contain about 90 % of feasible solutions. By using these small intervals, we can significantly reduce the run time of the optimization algorithm.

4.1. The general tuning idea and the parameters for tuning

Figure 2 shows that our general tuning approach proceeds in five steps:

1. The host takes the input data and initializes the parameters on device.
2. The device initializes the parameters by one of the methods in Section 4.2.
3. On the device, we start the kernel shown in Listing 1.
4. The ACO kernel on device finds an initial solution for CES.
5. The host takes the computed results and records them for further processing; if the solution is not obtained, step 2 or 3 are repeated depending on which particular tuning approach from Section 4.2 is adopted.

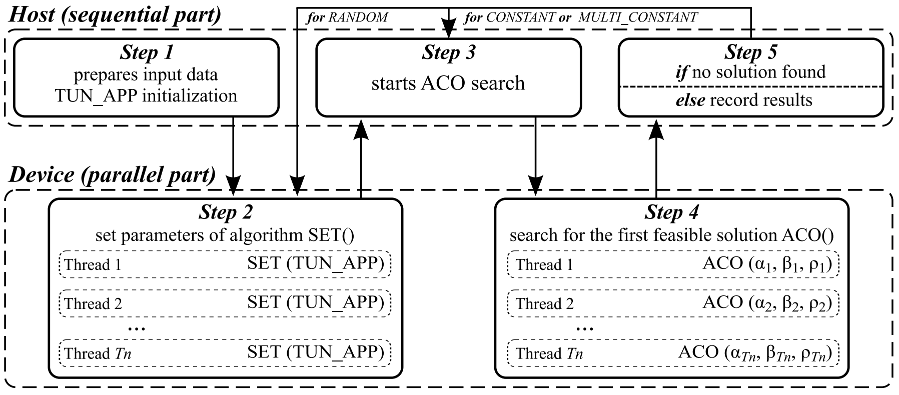

Host (sequential part)
for RANDOM
for CONSTANT or MULTI_CONSTANT
Step 1
prepares input data
TUN_APP initialization
Step 3
starts ACO search
Step 5
if no solution found
else record results
Device (parallel part)
Step 2
set parameters of algorithm SET()
Thread 1 SET (TUN_APP)
Thread 2 SET (TUN_APP)
...
Thread Tn SET (TUN_APP)
Step 4
search for the first feasible solution ACO()
Thread 1 ACO (α_1, β_1, ρ_1)
Thread 2 ACO (α_2, β_2, ρ_2)
...
Thread Tn ACO (α_{Tn}, β_{Tn}, ρ_{Tn})

Figure 2: General Method of ACO Parameter Tuning.

We tune the following parameters of ACO – α, β, and ρ – by means of a statistical analysis of the multiple experimental runs. In the following, we discuss our tuning parameters in detail.

Parameter α expresses how important is the accumulated pheromone for the path selection by ants: the larger is α, the more probable is that ants choose the same path as their predecessors on the path. On the one hand, the convergence of ACO increases in this case; on the other hand, the algorithm is more probable to trap into a local optimum.

Parameter β describes the mutual visibility of ants: how important is the heuristic information for selecting a particular path. If the value is small, it might bring ACO into stagnation or a local optimum, while a large value of β leads to the state transition probability similar to a greedy algorithm.

Parameter $\rho \in [0, 1]$ expresses the evaporation rate of the pheromone. Low value of ρ allow to quickly forget previous choices, while high values mean that the pheromone persists longer. Therefore, small ρ restricts the global search ability, while larger values improve it, but they slow down the convergence.

4.2. Three specific tuning approaches

We design and evaluate three specific approaches to parameter tuning: *Random*, *Constant* and *Multi-Constant*, as shown in Listing 5. The parameter `TUN_APP` of `SET()` stands for TUNing APProach.

```
__global__ void SET(TUN_APP) {
 ... /* obtaining thread identifier */
 threadID = blockDim.x * blockIdx.x + threadIdx.x;
 switch(TUN_APP){
  case RANDOM: /* parameters are initialized by random values */
   alpha[threadID] = rand(alpha_a, alpha_b);
   beta[threadID] = rand(beta_a, beta_b);
   rho[threadID] = rand(rho_a, rho_b);
  break;
  case CONSTANT: /* the same parameters for all threads */
   alpha[threadID] = constant_alpha;
   beta[threadID] = constant_beta;
   rho[threadID] = constant_rho;
  break;
  case MULTI_CONSTANT: /* different parameters for all threads */
   alpha[threadID] = constant_alpha[threadID];
   beta[threadID] = constant_beta[threadID];
   rho[threadID] = constant_rho[threadID]);
  break;
 } ...}
```

Listing 5: Approaches to parameter tuning: the SET() pseudocode.

Random Approach

In this approach, we initialize algorithm parameters by uniformly distributed randoms from intervals $[\alpha_a, \alpha_b]$, $[\beta_a, \beta_b]$, $[\rho_a, \rho_b]$, respectively. The bounds of the intervals are set as suggested in, e.g. [22]. We use the GPU-tailored random number generator available in the incuRAND library by NVIDIA [23]. The library function `curand_uniform()` computes uniformly distributed randoms in $(0.0, 1.0]$, so we use: `rand(a,b) = curand_uniform() * (b-a) + a`. The corresponding `SET()` function in the Random approach is shown in Listing 5 lines 6 – 8. If we find a combination of parameter values that yields a feasible solution then we save these values. Eventually, we find a set of tuples (α, β, ρ), for which ACO yields feasible solutions. If in step 4 of Random a feasible solution is not obtained after `maxIterationNumber` iterations then the approach moves to step 2 in Figure 2.

Constant Approach

In the Constant approach, all threads use same tuples of values α, β, ρ as in Listing 5, lines 11-13. The starting values are set using a frequency analysis of the tuple set from Random. In the iterations afterwards, we determine a set of tuples for which ACO computes feasible solutions. If no feasible solution can be in step 2 after `maxIterationNumber` iterations, we go to step 3 in Figure 2.

Multi-Constant Approach

This approach uses different initial values of tuples in the threads. In this approach, Listing 5, lines 16-18, we can view parallelized ACO as ”learning” on the base of random tuning: we save only the parameter tuples for we already found a solution. If no feasible solution is found after `maxIterationNumber` iterations, we go to step 3.

5. Experimental Results

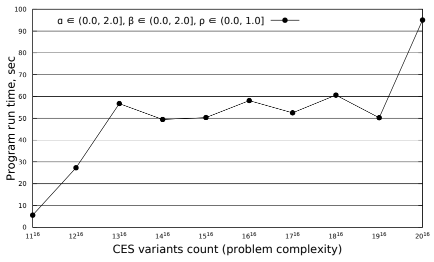
Our experimental setup consists of: 1) a CPU: Intel Xeon 2.3 GHz with 12 hyper-threaded cores, and 192 GB of RAM, 2) a GPU: NVIDIA V100 Tesla with 5 120 CUDA cores at 1.53 GHz, and 16 GB of global memory. We employ GNU C++ 6.4.0 and CUDA 10.0.

Our case study is to design a CES of 16 stages with 11 to 20 device variants per stage (i.e., from $11^{16} \approx 10^{17}$ to $20^{16} \approx 10^{21}$ variants). This size is significantly larger than was possible without parameter tuning [24] . We call the ACO algorithm 100 times for each problem size of 11^{16} to 20^{16} (i.e., 10 series of experiments). We measure the run time till finding a feasible solution, calculate the average and record it with the values α, β, ρ.

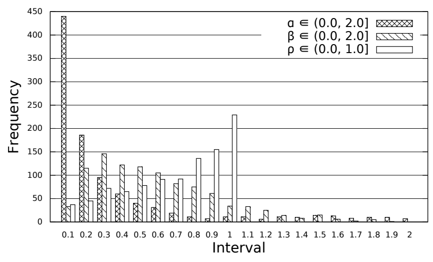
5.1. Random approach

In our first experiment, we assign α and β with random values of uniform distribution in $(0, 2]$, and ρ in $(0, 1]$, as suggested in [22]. In this series, we obtain $10 \cdot 100 = 1\,000$ tuples α, β, ρ with the sizes for which we found the solutions. The GPU run time for this Random series is approximately 14 hr.

Figure 3a) shows the average run time of ACO for different problem sizes, and Figure 3b) – the frequency of the found solutions, expressed as histograms for particular intervals of parameter values. We observe from the histograms that about 90% of solutions are obtained when the parameters belong to the intervals: $\alpha \in (0.0, 1.2]$, $\beta \in (0.1, 1.2]$, $\rho \in (0.2, 1.0]$.



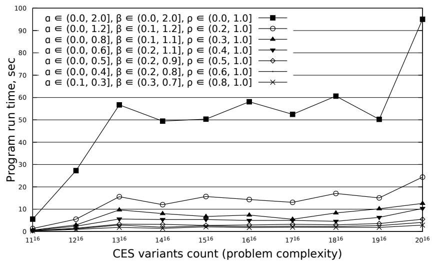
(a) Runtime.



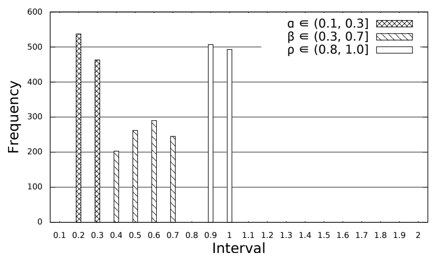
(b) Frequency.

Figure 3: Results of Random Parameters Tuning: First Iteration.

Our tuning approach reduces the intervals for α, β, ρ by selecting values with higher solutions frequency. For that, we iterate the runs and thus find new 1 000 solutions for the reduced parameter intervals, and analyze the solution frequency again. We repeat this process times, such that parameter intervals reduce to a single point: $\alpha = 0.2$, $\beta = 0.5$, $\rho = 0.9$, which we take as initial values for the Constant approach as described in Section 5.2.



(a) Runtime.



(b) Frequency.

Figure 4: ACO Random Parameters Tuning: Results.

In Figure 4a) , we observe the run time for problems of dimensions from 11^{16} to 20^{16}, and in Figure 4b) – the frequency after the final iteration of experiments for $\alpha \in (0.1, 0.3]$, $\beta \in (0.3, 0.7]$, $\rho \in (0.8, 1.0]$.

We conclude for Random approach that it decreases the run time of ACO on average by about 29 times (from about 50 sec for $\alpha \in (0.0, 2.0]$, $\beta \in (0.0, 2.0]$ and $\rho \in (0.0, 1.0]$ to about 1.7 sec for $\alpha \in (0.1, 0.3]$, $\beta \in (0.3, 0.7]$, $\rho \in (0.8, 1.0]$). The time for seven iterations of experiments is about 23 hr. In the following, we describe how the found parameter values are further used in the Constant and Multi-constant approaches.

5.2. The constant approach

In the constant approach, we use the same value for α, β, ρ obtained in the random approach, in every GPU thread.

Figure 5 shows our measurements, where the first tuple $\alpha = 0.4$, $\beta = 0.6$, $\rho = 0.9$ is taken for comparison from [24] where it was chosen empirically.

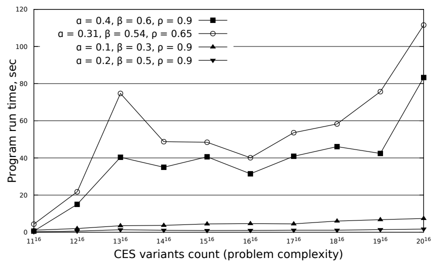


Figure 5: Constant approach: Measured run time vs. problem size.

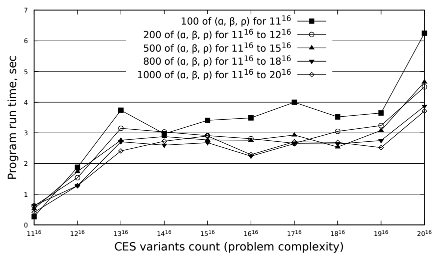


Figure 6: Multi-constant Approach: Measured run time vs. problem size.

An interesting observation is that, while it seems intuitively clear that by averaging the good parameter values we obtain good parameter candidates, this is not confirmed experimentally: the second tuple in Figure 5 ($\alpha = 0.31$, $\beta = 0.54$, $\rho = 0.65$) is the average of the parameter intervals found in the first iteration of Random, but these values increase the ACO run time by about 40%.

In Figure 5, the third tuple $\alpha = 0.1$, $\beta = 0.3$, $\rho = 0.9$ is obtained by analyzing the frequency data in Figure 3b) after Random's first iteration: we select values providing the highest frequency. This reduces the run time by a factor of about 8.5×. The fourth tuple $\alpha = 0.2$, $\beta = 0.5$, $\rho = 0.9$ is obtained after 7 iterations of Random: it shortens the run time of ACO by about 35×. The total GPU-time for tuning is the same 23 hrs, like in Random.

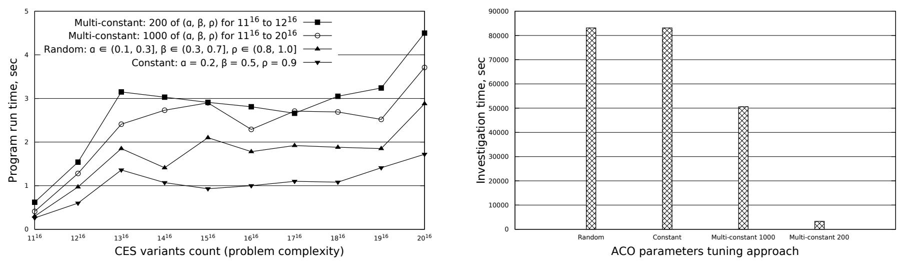
5.3. The multi-constant approach

In this approach, we set for each GPU-thread its own parameter values α, β, ρ which are found in the Random approach. The approach produces 1 000 parameter tuples for problem sizes of 11^{16} up to 20^{16}, for which we found feasible solutions. By running our program on the GPU we cyclically initialize the ACO parameters: e.g., with 100 tuples of parameters, and then after every 100 threads on the GPU, the values of parameters are repeated.

In Figure 6, we show the measurements for multi-constant approach. The shortest run time corresponds to the set of 1 000 tuples, and the longest – to the set of 100 tuples. The run time of ACO when beginning with 200 tuples differs from the run time for 1 000 tuples by only 16%. At the same time, the search time for 1 000 tuples is about 15.4 times longer than for 200 tuples.

5.4. Comparison of tuning approaches

In Figure 7, we draw a comparison of the best run times for ACO achieved by our different tuning approaches. We observe that the Multi-constant approach, compared to the version with 200 tuples, brings good solutions in a significantly shorter tuning time.



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

Figure 7: Comparison of Approaches.

Figure 7a) demonstrates that the lowest run time is obtained in the Constant approach with parameter values $\alpha = 0.2$, $\beta = 0.5$, $\rho = 0.9$ – run time about 1.1 sec. We observe in Figure 7b) that the tuning time of the Multi-constant approach with 200 samples (about 54 min) is 25 times shorter, whereas the resulting ACO run time is about 2.5 times shorter.

6. Conclusion

This paper makes the following contributions to the state of the art in the batch plant design optimization. First, we design and evaluate a parallel combined implementation of metaheuristic-based optimization. Second, we develop three tuning approaches for the parameters of the parallel Ant Colonies Optimization (ACO) metaheuristic. The advantages of our approach compared to related work are as follows: 1) using the high computing power of GPU for tuning, and 2) avoiding any additional information like algorithms for online-tuning, functions mapping in fuzzy logic, etc.

Acknowledgements

Thanks to the anonymous reviewers for their useful comments, and to NVIDIA for the provided hardware used in the experiments. Andrey Borisenko was partially supported by DAAD (German Academic Exchange Service) and the Russian Ministry of Education and Science within the "Mikhail Lomonosov II"-Program. Sergei Gorlatch was partially supported by the HPC2SE project of BMBF (German Ministry of Education and Research) at WWU Muenster.

References

[1] K. Hoffman, M. Padberg, Combinatorial and Integer Optimization, in: Encyclopedia of Operations Research and Management Science, Springer US, 2001, pp. 94–102. doi:10.1007/1-4020-0611-X_129.

[2] S. K. Joshi, J. C. Bansal, Parameter Tuning for Meta-Heuristics, Knowledge-Based Systems 189 (2020) 105094. doi:10.1016/j.knosys.2019.105094.

[3] M. Birattari, Tuning Metaheuristics: A Machine Learning Perspective, Springer Berlin Heidelberg, 2009. doi:10.1007978-3-642-00483-4.

[4] M. Dorigo, T. Stützle, Ant Colony Optimization: Overview and Recent Advances, in: Handbook of metaheuristics, Springer, 2018, pp. 311–351. doi:10.1007/978-3-319-91086-4_10.

[5] T. Stützle, M. López-Ibáñez, P. Pellegrini, M. Maur, M. Montes de Oca, M. Birattari, M. Dorigo, Autonomous Search, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, Ch. Parameter Adaptation in Ant Colony Optimization, pp. 191–215. doi:10.1007/978-3-642-21434-9_8.

[6] V. A. Tatsis, K. E. Parsopoulos, Dynamic Parameter Adaptation in Metaheuristics Using Gradient Approximation and Line Search, Applied Soft Computing 74 (2019) 368–384. doi:10.1016/j.asoc.2018.09.034.

[7] E. Barbosa, E. Senne, Improving the Fine-Tuning of Metaheuristics: An Approach Combining Design of Experiments and Racing Algorithms, Journal of Optimization 2017 (2017) 1–7. doi:10.1155/2017/8042436.

[8] F. Olivas, F. Valdez, O. Castillo, Dynamic Parameter Adaptation in Ant Colony Optimization Using a Fuzzy System for TSP Problems, in: IFSA-EUSFLAT, 2015, pp. 765–770.

[9] C. Huang, Y. Li, X. Yao, A Survey of Automatic Parameter Tuning Methods for Metaheuristics, IEEE Transactions on Evolutionary Computation 24 (2) (2020) 201–216. doi:10.1109/TEVC.2019.2921598.

[10] M. Fallahi, S. Amiri, M. Yaghini, A Parameter Tuning Methodology for Metaheuristics Based on Design of Experiments, International Journal of Engineering and Technology Sciences 2 (6) (2014) 497–521.

[11] D. Gómez-Cabrero, D. N. Ranasinghe, Fine-Tuning the Ant Colony System Algorithm through Particle Swarm Optimization, arXiv preprint arXiv:1803.08353 (Mar. 2018).

[12] Á. R. Trindade, F. Campelo, Tuning Metaheuristics by Sequential Optimization of Regression Models, arXiv preprint arXiv:1809.03646 (2018) 1–22.

[13] E. S. Skakov, V. N. Malysh, Parameter Meta-Optimization of Metaheuristics of Solving Specific NP-hard Facility Location Problem, Journal of Physics: Conference Series 973 (2018) 012063. doi:10.1088/1742-6596/973/1/012063.

[14] C.-C. Chen, Y.-T. Liu, 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 (2018) 1–15. doi:10.1155/2018/9485478.

[15] M. Mahi, Ö. K. Baykan, H. Kodaz, A New Hybrid Method Based on Particle Swarm Optimization, Ant Colony Optimization and 3-Opt Algorithms for Traveling Salesman Problem, Applied Soft Computing 30 (2015) 484–490. doi:10.1016/j.asoc.2015.01.068.

[16] O. Castillo, H. Neyoy, J. Soria, P. Melin, F. Valdez, 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 (2015) 150–159. doi:10.1016/j.asoc.2014.12.002.

[17] A. B. Borisenko, S. V. Karpushkin, Hierarchy of Processing Equipment Configuration Design Problems for Multiproduct Chemical Plants, Journal of Computer and Systems Sciences International 53 (3) (2014) 410–419. doi:10.1134/s1064230714030046.

[18] J. Valadi, P. Siarry, Applications of Metaheuristics in Process Engineering, Springer Science & Business Media Business Media, 2014. doi:10.1007/978-3-319-06508-3.

[19] F. Rossi, P. Van Beek, T. Walsh, Handbook of Constraint Programming, Elsevier, 2006.

[20] C. Solnon, Ant Colony Optimization and Constraint Programming, Wiley, Hoboken, NJ, 2013. doi:10.1002/9781118557563.

[21] E. Aarts, J. Korst, W. Michiels, Simulated Annealing, in: Search Methodologies, Springer Science & Business Media, 2014, pp. 265–285. doi:10.1007/978-1-4614-6940-7_10.

[22] S. Khan, M. Bilal, M. Sharif, M. Sajid, R. Baig, Solution of N-Queen Problem Using ACO, in: 2009 IEEE 13th International Multitopic Conference, IEEE, 2009, pp. 1–5. doi:10.1109/INMIC.2009.5383157.

[23] NVIDIA Corporation, The NVIDIA CUDA Random Number Generation Library (cuRAND) (Dec. 2018).
URL `https://developer.nvidia.com/curand`

[24] A. Borisenko, S. Gorlatch, Comparing GPU-parallelized Metaheuristics to Branch-and-Bound for Batch Plants Optimization, The Journal of Supercomputing (2018) 1–13doi:10.1007/s11227-018-2472-9.