Parallelizing Metaheuristics for Optimal Design of Multiproduct Batch Plants on GPU

Andrey Borisenko[1(✉)] and Sergei Gorlatch[2]

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

Abstract. We propose a metaheuristics-based approach to the optimal design of multi-product batch plants, with a particular application example of chemical-engineering systems. Our hybrid approach combines two metaheuristics: Ant Colony Optimization (ACO) and Simulated Annealing (SA). We develop a sequential implementation of the proposed method and we parallelize it on Graphics Processing Units (GPU) using the CUDA programming environment. We experimentally demonstrate that the results of our hybrid metaheuristic approach (ACO+SA) are very near to the global optimal solutions, but they are produced much faster than using the deterministic Branch-and-Bound approach.

Keywords: Hybrid metaheuristics · Ant Colony Optimization · Simulated Annealing · GPU computing · CUDA · Parallel metaheuristics · Combinatorial optimization · Multiproduct batch plant design

1 Motivation and Related Work

A *heuristic* for an optimization problem is an algorithm that explores not all possible states of the problem, but rather the most likely ones. Purely heuristics-based solutions may be inconsistent, therefore, *metaheuristics* are used that usually perform better than simple heuristics [14]. A metaheuristic is a generic algorithmic template that can find high-quality solutions of optimization problems [4] exploiting a trade-off of local search and global exploration. Metaheuristics find good-quality solutions for optimization problems in a reasonable amount of time, but there is no guarantee that the optimal solution is always reached [26].

In this paper, we consider a challenging area of optimisation – optimal design of multiproduct batch plants, e.g., in the chemical industry for producing pharmaceuticals, polymers, food etc. There has been an active research on efficiently solving such and similar problems. The classical n-queens problem was addressed using the Ant Colony Optimization (ACO) [17,24] and its combination with a Genetic Algorithm (GA) [2]. Paper [21] solves process engineering problems by the Differential Evolution (DE) algorithm and demonstrates its advantages over the exact optimization by Branch-and-Bound (B&B) and using a GA. In [13],

© Springer International Publishing AG 2017
V. Malyshkin (Ed.): PaCT 2017, LNCS 10421, pp. 405–417, 2017.
DOI: 10.1007/978-3-319-62932-2_39

a particle swarm algorithm and a GA are exploited for multiproduct batch plant design. Paper [10] develops a multiobjective GA which demonstrates high flexibility and adaptability for various engineering problems. The problem of the optimal design of batch plants with imprecise demands on product amounts is addressed in [3] by integrating an analytic hierarchy process strategy for the analysis of the GA Pareto-optimal solutions. Paper [19] uses ACO and SA to solve a stochastic facility layout problem in which product demands are normally distributed random variables.

In order to reduce the run time of metaheuristics-based approaches, their implementation on different parallel architectures has been studied. In particular, Graphics Processing Units (GPU) are widely used by employing the CUDA platform [20]. GPU were used for solving the classical TSP problem by simulated annealing [27] and by an ant system [8]. The problem of scheduling transit stop inspection and maintenance was studied by using Harmony Search and ACO [16], with alternative implementations on CPU and GPU.

Our contribution in this paper is two-fold: (1) we develop a novel, hybrid approach which combines two metaheuristics – Ant Colony Optimization (ACO) [11] and Simulated Annealing (SA) [18], and (2) we implement it on a CPU-GPU system using CUDA and we show that it is preferable to the Branch-and-Bound approach used in our previous work [7]. Section 2 describes the mathematical problem formulation, Sect. 3 – the methodology of our hybrid ACO+SA approach, and Sect. 4 – its parallelization. Section 5 reports our experimental results, and Sect. 6 concludes the paper.

2 Problem Formulation

Our application use case is optimizing a *Chemical-Engineering System* (CES) – a set of equipment (tanks, filters, dryers etc.) which manufacture some products. A CES consists of a sequence of I processing stages; i-th stage is equipped with equipment units from a finite set X_i, with J_i being the number of equipment units variants in X_i. All equipment unit variants of a CES are described as $X_i = \{x_{i,j}\}, i = \overline{1,I}, j = \overline{1,J_i}$, where $x_{i,j}$ is the main size j (working volume, working surface) of the unit suitable for processing stage i. A CES variant $\Omega_e, e = \overline{1,E}$ (where $E = \prod_{i=1}^{I} J_i$ is the number of all possible variants) is an ordered set of available equipment unit variants. The goal is finding the optimal number of units at processing stages and their sizes while the input data are: demand for each product of assortment, production horizon, available equipment set, etc. Each variant Ω_e of a system must be in an operable condition (*compatibility constraint*), i.e., it must satisfy the condition of a joint action for its processing stages expressed by function S: $S(\Omega_e) = 0$ if the compatibility constraint is satisfied. An operable variant of a CES must also satisfy a *processing time constraint*: $T(\Omega_e) \leq T_{max}$, where T_{max} is the total available time (horizon).

Thus, designing an optimal CES is formulated as follows [5,6]: find a variant $\Omega^* \in \Omega_e, e = \overline{1,E}$ of a CES, that minimizes the objective function – equipment costs $Cost(\Omega_e)$, and both compatibility and processing time constraint are satisfied:

$$\Omega^* = argmin\ Cost(\Omega_e), e = \overline{1, E} \tag{1}$$

$$\Omega_e = \{x_{1,j_1}, x_{2,j_2}, \ldots, x_{I,j_I} | j_i = \overline{1, J_i}, i = \overline{1, I}\}, e = \overline{1, E} \tag{2}$$

$$x_{i,j} \in X_i,\ i = \overline{1, I}, j = \overline{1, J_i} \tag{3}$$

$$S(\Omega_e) = 0, e = \overline{1, E} \tag{4}$$

$$T(\Omega_e) \leq T_{max}, e = \overline{1, E} \tag{5}$$

The search space can be represented as a tree of height I (Fig. 1). Each tree level corresponds to one processing stage of the CES, each edge corresponds to a selected equipment variant taken from the set of possible variants X_i at stage i. Each node $n_{i,k}$ at the tree layer $N_i = \{n_{i,1}, n_{i,2}, \ldots, n_{i,k}\}$, $i = \overline{1, I}, k = \overline{1, K_i}, K_i = \prod_{l=1}^{i}(J_l)$ corresponds to a variant of equipment units for stages 1 to i.

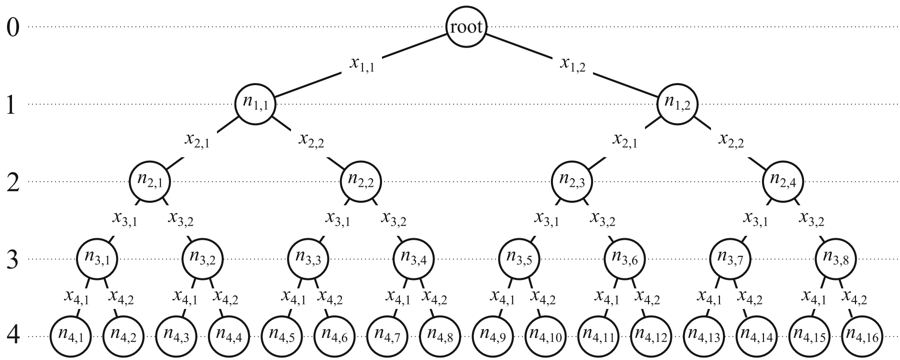

Fig. 1. The search tree for a CES with 4 stages.

Figure 1 shows an example CES consisting of 4 stages ($I = 4$), where each stage can be equipped with 2 devices ($J_1 = J_2 = J_3 = J_4 = 2$), i.e., the number of all possible system variants is $2^4 = 16$.

3 Hybrid Metaheuristic Approach

Our approach to the optimal design of multi-product batch plants is based on two metaheuristics: Simulated Annealing (SA) and Ant Colony Optimization (ACO). SA is widely used for solving optimization problems [18,26]; its key advantage is escaping from local optima by allowing hill-climbing moves to find a global optimum. SA can deal with arbitrary systems and objective functions; it often finds an optimal solution and generally finds a good-quality solution.

Before searching for a solution using SA, we need a feasible initial solution. For classical optimization problems, e.g., Traveling Salesman Problem (TSP), it is possible to use a random initial solution. However, for our problem described in Sect. 2, random initialization is unacceptable, because the compatibility (4)

and processing time (5) constraints must be satisfied. Our search for a feasible initial solution is a *Constraint Satisfaction Problem* (CSP) [23] without cost optimization, which consists in finding an operable variant of a CES, satisfying (4) and (5). For solving it, we use the Ant Colony Optimization (ACO) metaheuristic [15] that provides good-quality results in many applications, including CSP [24].

3.1 Ant Colony Optimization (ACO)

The Ant Colony Optimization (ACO) metaheuristic can be viewed as a multi-agent system in which agents (ants) interact with each other in order to reach a global goal [26]. It is inspired by the behaviour of ant colonies: while walking from food source to the nest and vice versa, ants deposit a chemical substance called *pheromone* on their path. Pheromone is used as a communication medium among ants and guides them to find the shortest path from the nest to food: ants follow, with some probability, the pheromone deposited by previous ants.

```
1  AntColonyOptimization(){
2   isFound = false; /* repeat while solution not found */
3   while(!isFound){
4      Initialize(); /* initialize pheromone value */
5      foreach(ant in swarm){/* for each ant in colony */
6       ConstructSolution(); }
7      if(isFound) return; /* if solution is found, then end */
8      PheromoneUpdate(); /* update pheromone */
9      EvaporatePheramone(); }
```

Listing 1. The pseudocode of ACO algorithm.

Listing 1 shows the pseudocode of the ACO algorithm for our problem. The number M of ants is the algorithm parameter which determines the trade-off between the number of iterations and the breadth of the search per iteration: the larger the number of ants per iteration, the fewer iterations are needed [25]. All ants behave in a similar way: each ant moves from the top of the tree in Fig. 1 to the bottom. Once an ant selected a node $r = n_{i,j}$ at level i, it can pick the next child node $s = n_{i+1,j}$. The tour of an ant ends at the last 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 [12,24,26]. The factors α and β influence the pheromone value and heuristic value respectively. These parameters control the relative importance of the pheromone trails and the heuristic information.

Our approach to calculating heuristic information is based on the fact that a CES with bigger units is usually more expensive, but it has a bigger batch

size of products and so produces faster than a CES with smaller units. This is favourable for satisfying the time constraint (5). Therefore, we make a unit which satisfies the compatibility constraint (4) for the beginning part of the CES and a larger basic size more preferable than a unit with the unsatisfied compatibility constraint and smaller basic size. We use the following rule for the pheromone update (line 8): $\tau_{rs} = \tau_{rs} + Q/\sum_{m=1}^{M} L_m$, 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. Listing 2 shows our approach to computing the fitness value `L[m]` (line 2). Function `NumberS()` (lines 4–8) counts the number of stages of the beginning part of the CES, composed of devices for stages 1 to i (lines 6–7), for which (4) is satisfied. We add 1 to `NumberS()` if constraint (5) is satisfied (line 2). Therefore, the minimal fitness value is 0 (no constraint is satisfied), the maximal value is $I + 1$ (all constraint are satisfied, the problem is solved). We use the maximal fitness value as constant Q: $Q = I + 1$. With time, the concentration of pheromone decreases due to evaporation. The evaporation (Listing 1 line 9) is performed at a constant rate after the completion of each iteration. It allows the ant colony to avoid an unlimited increase of the pheromone value and to forget poor choices made previously [25]. We implement this as follows: $\tau_{rs} = \tau_{rs} \cdot \rho$, where $\rho \in [0, 1]$ is the trail persistence parameter.

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

Listing 2. Pseudocode of the fitness value computing for ACO.

3.2 Simulated Annealing (SA)

The basic idea of SA is to use random search which accepts not only changes that improve the objective function, but also some changes that are not ideal, in order to escape local minima. A parameter t called *temperature* governs the search behaviour.

Listing 3 shows a pseudocode of our SA version that performs two loops: the inner loop (line 4) to search for a neighbouring solution, and the outer loop (line 3) to decrease the temperature in order to reduce the probability of accepting the non-improving neighbouring solutions in the inner loop. `W` is a vector of length `I`, each element `W[i]` specifying the device variant at each

```
1  SimulatedAnnealing(){
2    t = Tinit; W = Winit;/* initialize temperature and guess */
3    while(t > Tfinal) {/* loop until t don't reaches Tfinal */
4       for(l = 0; l < Lmax; l++) { /* repeat Lmax times */
5        Wcand = Perturb(W); /* construct neighbour solution */
6         /* check compatibility and processing time constraints */
7         if (S( Wcand ) == 0 && T( Wcand ) <= Tmax ){
8          deltaCost = Cost(Wcand) - Cost(W);
9          if (deltaCost < 0){ /* if new solution is better */
10          W = Wcand;} /* accept the new solution */
11         else {
12           r = rand(0, 1); /* generate a random number */
13           p = exp (-deltaCost / t); /* calculate probability */
14           if (p > r) {
15            W = Wcand; }}}} /* accept the new solution */
16     t = sigma * t; }} /* decrease the temperature value */
17 Perturb(W){
18   stage = (int) rand(1, I); /* select random stage */
19   W[stage] = (int) rand(1, J[stage]);/* select random unit */
20   return W; }
```

Listing 3. The pseudocode of SA algorithm.

stage of the problem solution (1)–(5). At each iteration of the inner loop, we generate a new candidate solution `Wcand` in the neighbourhood of the current feasible solution (line 5) using our procedure `Perturb()` (lines 17–20): we select a random stage (line 18) in the feasible solution, for which we select a random unit (line 19) from the equipment set accessible for this stage. Thus at each iteration we change only one unit in the feasible solution at a stage. We avoid getting trapped in a local optimum by randomly generating neighbours and accepting a solution that worsens the value of the objective function with certain probability [22] which depends on the change of the objective function $\Delta\mathcal{E}$ and parameter t: the acceptance probability p decreases over time as t decreases. Consequently, SA first performs a wide investigation of the solution space and then restricts the solution space gradually, converging to the best solution. We initialize `W` with an initial feasible solution `Winit` obtained as the result of ACO, and the temperature `t` with initial value `Tinit` (line 2). We choose `Tinit` as a difference between the cost of the most expensive and the cheapest CES variant, as recommended in [1]. The transition probability `p` (line 13) is determined by $p = \exp(-\Delta\mathcal{E}/(k_{\mathrm{B}} \cdot t))$, where k_{B} is the Boltzmann's constant, $\mathcal{E}$ is the change of the energy level [28]. We use $k_{\mathrm{B}} = 1$ and $\gamma = 1$ [28]. Thus, the probability becomes `p = exp (-deltaCost / t)` (line 13).

A finite-time implementation of SA is obtained by generating a sequence of homogeneous Markov chains of finite length L_{max} which depends on the size of the problem [1]. The iterations at a given value of `t` repeat `Lmax` times (line 4). We compute `Lmax` as the total size of equipment set, i.e., $L_{max} = \sum_{i=1}^{I} J_i$,

where J_i is the number of equipment units variants for stage i. By permuting the feasible solution, we select at each iteration a random unit (line 19) in one random stage (line 18). Temperature `t` is decreased at the end of each iteration using a cooling schedule defined by an initial temperature `Tinit`, a rule for reducing `t`, and a final temperature `Tfinal` which is fixed at a small value chosen as the smallest possible difference in cost between two neighboring solutions; in our case, we use for `Tfinal` the price of the cheapest unit. We use (line 16) the fast cooling rule $t = \sigma \cdot t$ [26], where $0.8 \leq \sigma \leq 0.99$ as recommended in [1].

4 Parallelization for GPU

Figure 2 illustrates our parallel implementation of the hybrid (ACO+SA) approach described in Sect. 3 on a system comprising a CPU and a GPU.

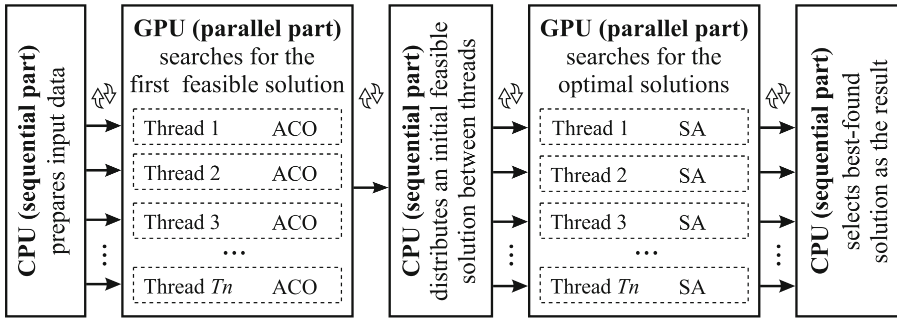

Fig. 2. The hybrid algorithm structure.

Application code consists of a sequential code (*host* code for CPU) that invokes parallel execution of hundreds or thousands of threads on the *device* (GPU), where all threads execute the same *kernel* code. The implementation consists of the following five steps (from left to right in Fig. 2):

1. CPU reads the input data (number of CES stages I, number of accessible equipment set J_i, production horizon T_{max} etc.) from a file, initializes the metaheuristics' parameters for ACO and SA, sends this data to GPU, and starts on the GPU the kernel function for ACO.
2. The ACO kernel on the GPU searches for the first feasible solution – the initial CES-variant, as described is Sect. 3.1. We use the Multiple Ant Colonies approach [9]: all colonies work as threads in parallel to solve the problem independently. If some thread finds a solution then all threads terminate. With an increasing number of threads, the probability of finding a solution increases, and therefore the search time is typically reduced.
3. CPU receives the obtained solution, distributes it between threads as an initial solution for SA, and starts the SA kernel function on the GPU.

4. The SA kernel on the GPU searches in each thread for the optimal solution with the initial solution found by ACO, i.e., we do not try to reduce the time of one iteration, but rather increase the number of iterations executed simultaneously. Each thread executes an independent instance of SA, thus, the chance of the algorithm to converge to the global optimum increases, even if all instances use the same initial solution. A larger number of threads does not reduce the run time of the algorithm, but rather increases the probability that some thread eventually finds a nearly optimal solution.
5. CPU receives the SA solutions obtained by the GPU threads and chooses the best among them – this is the final solution of our problem.

Host Code. The host starts its work by loading the input data from a file. The number of threads is a program launch parameter taken as a command-line argument. The host sends data to the GPU and starts the kernel `ACO()` that implements the ACO-algorithm. A CUDA kernel launch is asynchronous, i.e. it returns control to the CPU immediately after starting the kernel. Using `cudaDeviceSynchronize()`, the CPU waits until the GPU terminates and receives the results from it.

Kernel Code for ACO. Listing 4 shows our parallel implementation of ACO, where each thread simulates the work of one ant colony. For all threads, initially, all edges are assigned small random pheromone values from interval $[0, 1]$ (lines 4–5). The global flag `isFound` and the local iteration counter `iterCounter` are used to control threads. The flag is changed by a thread using `atomicAdd()` if this thread has found a feasible solution (line 21). The local iteration counter is used by each thread as a nonstop operation protection: if ants in this thread cannot find the solution after `maxIterNumber` iterations (which is possible for stochastic algorithms) then the thread terminates. After initialization, each ant `m` in swarm `M` generates a path (lines 9–17). Here, `Want` is a local two-dimensional array of length `M`, each element of which is a vector of length `I` specifying the device variant at each stage of the solution.

We do not discuss the kernel code of SA – it largely follows Listing 3.

5 Experimental Results

Our experiments are conducted on a heterogeneous system comprising: (1) a CPU: Intel Xeon E5-1620 v2, 4 cores with Hyper-Threading, 3.7 GHz with 16 GB RAM, and (2) a GPU: NVIDIA Tesla K20c with altogether 2496 CUDA cores and 5 GB of global memory. We use Ubuntu 16.04.2, NVIDIA Driver version 367.57, CUDA version 8.0 and GNU C++ Compiler version 5.4.0.

As our test case, we evaluate the design of a CES consisting of 16 processing stages with 2 to 12 variants of devices at every stage (total 2^{16} to 12^{16} CES variants). In our previous work [6,7], we used the Branch-and-Bound (B&B) algorithm to find the global optimal solution. Here we solve the same problem

```
1  __global__ void ACO(){ /* obtaining thread identifier */
2    threadID = blockDim.x * blockIdx.x + threadIdx.x;
3    if(threadID < numThreads){ /* pheromone initialization */
4     for (i = 1; i <= I; i++) {
5      for (j = 1; j <= J[i]; j++){tau[i][j] = curand(0, 1);}}
6     iterCounter = 0; /* while solution is not found */
7     while (isFound == 0 && iterCounter < maxIterNumber)){
8       /* generate path for each ant m in swarm M */
9       for(m = 1; m <= M && isFound == 0; m++){sum = 0.0;
10       for (i = 1; i <= I - 1; i++){
11        for (j = 1; j <= J[i]; j++){
12         eta[i][j] = (S(Want[m], i + 1) ? 1:0) + X[i][j];
13         sum += pow(tau[i][j], alpha) * pow(eta[i][j], beta);}
14        r = curand(0, 1); sump = 0.0;
15        for (j = 1; j <= J[i]; j++){
16         p = pow(tau[i][j],alpha) * pow(eta[i][j],beta) / sum;
17         sump += p; if(sump > r) {Want[m][i] = j; break; }}}}
18        /* calculate new pheromone values */
19        for(m = 1; m <= M && isFound == 0; m++){
20         L[m] = NumberS(Want[m],I) + (T(Want[m]) <= Tmax ? 1:0);
21         if (L[m] == Q) {atomicAdd(isFound, 1); bestAntId = m;}
22         for (i = 1; i <= I; i++) {
23          for (j = 1; j <= J[i]; j++) {dtau[i][j] = 0.0; }}
24         for (i = 1; i <= I; i++){
25          idx = Want[m][i]; dtau[i][idx] += Q / L[m]; }
26         /* pheromone update and evaporation */
27         for (i = 1; i <= I; i++){
28          for (j = 1; j <= J[i]; j++){
29           tau[i][j] = tau[i][j] * rho + dtau[i][j]; }}}
30           iterCounter++; }
31         /* save feasible solution and its thread identifier */
32         if(bestAntId != -1) {Wfirst[threadID] = Want[bestAntId];
33         threadIdx[threadID] = threadID; }}}
```

Listing 4. The kernel pseudocode for ACO.

on the same test system using our hybrid metaheuristic approach (ACO+SA), and we compare the results with the solution obtained by B&B. Since both SA and ACO are probability-based algorithms, their results will be different if run multiple times on the same instance of a problem; therefore, we run each instance for 100 times and we take the average of the measured values.

Figure 3 shows how the run time of the (ACO+SA) parallel program depends on the number of threads. We run our CUDA-based implementation with the number of threads from 100 to 2500 with step 100, for the CES example of 16 processing stages with 10 variants of units. We observe that the run time is decreasing with the increasing number of threads. While on 100 threads, the ACO takes 91% of the total run time, the portion of ACO decreases to only about

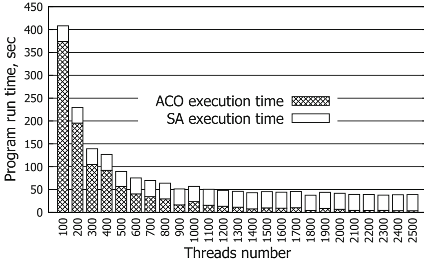

Fig. 3. Run time of (ACO+SA) depending on threads number.

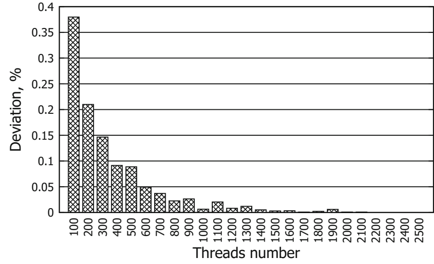

Fig. 4. Deviation of the found solution from global optimum.

10% on more than 2000 threads. This is because, with more threads, ACO finds a solution faster with a higher probability, whereas using more threads for SA can improve the quality of solution, but not the speed.

Figure 4 shows the deviation of the observed objective value (CES cost) by our hybrid algorithm, calculated as $|observed - expected|/expected \cdot 100\%$. As the expected value we use the global optimal value obtained by the B&B algorithm. On 100 threads, the deviation is about 0.38% and then it decreases to almost 0% for more than 2000 threads, so we achieve an almost optimal solution.

Figure 5 shows the deviation (vertical axis has a logarithmic scale) of solutions obtained by our hybrid (ACO+SA) algorithm (sequential on CPU, parallel on GPU using up to 2500 threads) from the global optimum obtained by sequential B&B. We observe that our parallel hybrid algorithm produces a nearly global optimal solution (for problem size 2^{16}–9^{16} the deviation is 0%, and for problem size 10^{16}–12^{16} the deviation is less than 0.01%). The deviation obtained by our sequential algorithm is less than 1% for small size problem 2^{16}, but it increases to about 14% for the problem size of 12^{16}. This is because the sequential implementation performs only a single run of the SA: for small problem sizes, the probability of finding a good solution is higher than for larger problem sizes.

Figure 6 shows the program run time (vertical axis has a logarithmic scale) of our hybrid approach vs. B&B depending on the problem size. The program

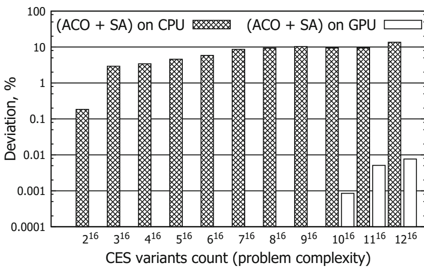

Fig. 5. Deviation (logarithmic scale) for different problem sizes.

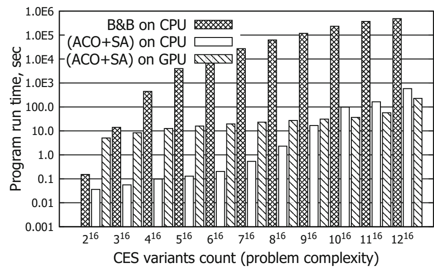

Fig. 6. Run time (logarithmic scale) depending on problem sizes.

run time of B&B increases exponentially: while for problem size 2^{16} the run time is less than 1 s, for size 12^{16} the run time of B&B becomes prohibitively long at about 134 h. The run time of the sequential implementation of our hybrid algorithm is less than 0.1 s for the smallest problem, and it increases to about 580 sec for our maximal problem size. The run time of the parallel implementation smoothly increases from 5 to 227 sec. The parallel implementation is slower than the sequential implementation for smaller problem sizes (2^{16} to 9^{16}), but for larger problem sizes (10^{16} to 12^{16}) it is faster than the sequential version by about 2.6–2.8 times. At first glance, the speedup of 2.7 times compared to the sequential case is small, but the quality of the solutions obtained by the parallel implementation is significantly higher: the deviation from the global optimum for the parallel implementation is less than 0.01% against more than 10% deviation for the sequential version. This good quality of solutions is achieved by the independent runs of SA: for a larger number of threads the probability that one of the threads finds a nearly optimal solution is higher, because running a parallel algorithm on 2500 threads is equivalent to the launch of sequential algorithm 2500 times and the choice of the best among the found solutions. The parameters of metaheuristic algorithms influence both the run time and the quality of solutions. Empirically we have found that $\alpha = 0.4$ and $\beta = 0.6$ with the colony size $M = 100$ are good values for our application: they were selected after numerous experiments. In our experiments, we use $\rho = 0.9$ and the cooling rule constant $\sigma = 0.9$, also chosen empirically for our problem.

6 Conclusion

Our contribution is the novel hybrid (ACO+SA) metaheuristic approach to solving the optimization problem for multiproduct batch plants design and its parallel implementation on a CPU-GPU platform. We have found out that increasing the number of threads accelerates finding the solution with ACO and increases the reliability and quality of the solutions obtained by SA. We compare our results with the global optimal solution obtained by the B&B method. Our experiments confirm that our parallel hybrid approach obtains good-quality solutions which are very near to the global optimal values obtained by a deterministic algorithm like B&B, but our approach finds the solution much faster.

Acknowledgement. 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, as well as by the German Research Agency (DFG) in the framework of the Cluster of Excellence CiM at the University of Muenster. We also thank the Nvidia Corp. for the donated hardware used in our experiments.

References

1. Aarts, E., Korst, J., Michiels, W.: Simulated annealing. In: Search Methodologies, pp. 265–285. Springer Science & Business Media, Heidelberg (2014)
2. Agarwal, K., Sinha, A., Hima Bindu, M.: A novel hybrid approach to N-Queen problem. In: Wyld, D., Zizka, J., Nagamalai, D. (eds.) Advances in Computer Science, Engineering & Applications. AISC, vol. 166, pp. 519–527. Springer, Heidelberg (2012). doi:10.1007/978-3-642-30157-5_52
3. Aguilar-Lasserre, A.A., Bautista, M.A.B., Ponsich, A., Huerta, M.A.G.: An AHP-based decision-making tool for the solution of multiproduct batch plant design problem under imprecise demand. Comput. Oper. Res. **36**(3), 711–736 (2009)
4. Birattari, M.: Tuning Metaheuristics: A Machine Learning Perspective. Springer, Heidelberg (2009)
5. Borisenko, A.B., Karpushkin, S.V.: Hierarchy of processing equipment configuration design problems for multiproduct chemical plants. J. Comput. Syst. Sci. Int. **53**(3), 410–419 (2014)
6. Borisenko, A., Haidl, M., Gorlatch, S.: A GPU parallelization of branch-and-bound for multiproduct batch plants optimization. J. Supercomput. **73**(2), 639–651 (2017)
7. Borisenko, A., Kegel, P., Gorlatch, S.: Optimal design of multi-product batch plants using a parallel branch-and-bound method. In: Malyshkin, V. (ed.) PaCT 2011. LNCS, vol. 6873, pp. 417–430. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23178-0_36
8. Dawson, L., Stewart, I.: Improving ant colony optimization performance on the GPU using CUDA. In: 2013 IEEE Congress on Evolutionary Computation, pp. 1901–1908. IEEE, June 2013
9. Delévacq, A., Delisle, P., Gravel, M., Krajecki, M.: Parallel ant colony optimization on graphics processing units. J. Parallel Distrib. Comput. **73**(1), 52–61 (2013)
10. Dietz, A., Azzaro-Pantel, C., Pibouleau, L., Domenech, S.: Strategies for multiobjective genetic algorithm development: Application to optimal batch plant design in process systems engineering. Comput. Ind. Eng. **54**(3), 539–569 (2008)
11. Dorigo, M., Blum, C.: Ant colony optimization theory: a survey. Theoret. Comput. Sci. **344**(2–3), 243–278 (2005)
12. Dorigo, M., Stützle, T.: Ant colony optimization: overview and recent advances. In: Gendreau, M., Potvin, J.-Y. (eds.) Handbook of Metaheuristics. International Series in Operations Research & Management Science, vol. 146, pp. 227–263. Springer, New York (2010). doi:10.1007/978-1-4419-1665-5_8
13. El Hamzaoui, Y., Bassam, A., Abatal, M., Rodríguez, J.A., Duarte-Villaseñor, M.A., Escobedo, L., Puga, S.A.: Flexibility in biopharmaceutical manufacturing using particle swarm algorithms and genetic algorithms. In: Schütze, O., Trujillo, L., Legrand, P., Maldonado, Y. (eds.) NEO 2015. SCI, vol. 663, pp. 149–171. Springer, Cham (2017). doi:10.1007/978-3-319-44003-3_7
14. Gandomi, A.H., Yang, X.S., Talatahari, S., Alavi, A.H.: Metaheuristic algorithms in modeling and optimization. In: Metaheuristic Applications in Structures and Infrastructures, pp. 1–24. Elsevier BV (2013)
15. Gonzalez-Pardo, A., Camacho, D.: A new CSP graph-based representation for ant colony optimization. In: 2013 IEEE Congress on Evolutionary Computation, pp. 689–696. Institute of Electrical and Electronics Engineers (IEEE), June 2013
16. Kallioras, N.A., Kepaptsoglou, K., Lagaros, N.D.: Transit stop inspection and maintenance scheduling: a GPU accelerated metaheuristics approach. Transp. Res. Part C Emerg. Technol. **55**, 246–260 (2015)

17. 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. Institute of Electrical and Electronics Engineers (IEEE), December 2009
18. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P., et al.: Optimization by simulated annealing. Science **220**(4598), 671–680 (1983)
19. Lee, T.S., Moslemipour, G., Ting, T.O., Rilling, D.: A novel hybrid ACO/SA approach to solve stochastic dynamic facility layout problem (SDFLP). In: Huang, D.-S., Gupta, P., Zhang, X., Premaratne, P. (eds.) ICIC 2012. CCIS, vol. 304, pp. 100–108. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31837-5_15
20. NVIDIA Corporation: CUDA C programming guide 8.0, September 2016. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
21. Ponsich, A., Coello, C.C.: Differential evolution performances for the solution of mixed-integer constrained process engineering problems. Appl. Soft Comput. **11**(1), 399–409 (2011)
22. Pourvaziri, H., Azimi, P.: A tuned-parameter hybrid algorithm for dynamic facility layout problem with budget constraint using GA and SAA. J. Optim. Ind. Eng. **7**(15), 65–75 (2014)
23. Rossi, F., Van Beek, P., Walsh, T.: Handbook of Constraint Programming. Elsevier, Amsterdam (2006)
24. Solnon, C.: Ant Colony Optimization and Constraint Programming. Wiley Inc., Hoboken (2010)
25. Stützle, T., López-Ibánez, M., Pellegrini, P., Maur, M., de Oca, M.M., Birattari, M., Dorigo, M.: Parameter adaptation in ant colony optimization. In: Hamadi, Y., Monfroy, E., Saubion, F. (eds.) Autonomous Search, pp. 191–215. Springer, Heidelberg (2011). doi:10.1007/978-3-642-21434-9_8
26. Valadi, J., Siarry, P.: Applications of Metaheuristics in Process Engineering. Springer Science & Business Media, Heidelberg (2014)
27. Wei, K.C., Wu, C.C., Yu, H.L.: Mapping the simulated annealing algorithm onto CUDA GPUs. In: 2015 10th International Conference on Intelligent Systems and Knowledge Engineering (ISKE), pp. 1–8, November 2015
28. Yang, X.S.: Nature-Inspired Metaheuristic Algorithms. Luniver Press, Bristol (2010)