Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200057

Parallelization Strategies for GPU-Based Ant Colony Optimization Applied to TSP

Breno Augusto DE MELO MENEZES [a,1], Luis Filipe DE ARAUJO PESSOA [a]
Herbert KUCHEN [a], and Fernando BUARQUE DE LIMA NETO [b]
[a] *University of Muenster, Germany*
[b] *University of Pernambuco, Brazil*

Abstract.
Graphics Processing Units (GPUs) have been widely used to speed up the execution of various meta-heuristics for solving hard optimization problems. In the case of Ant Colony Optimization (ACO), many implementations with very distinct parallelization strategies and speedups have been already proposed and evaluated on the Traveling Salesman Problem (TSP). On the one hand, a coarse-grained strategy applies the parallelization on the ant-level and is the most intuitive and common strategy found in the literature. On the other hand, a fine-grained strategy also parallelizes the internal work of each ant, creating a higher degree of parallelization. Although many parallel implementations of ACO exist, the influence of the algorithm parameters (e.g., the number of ants) and the problem configurations (e.g., the number of nodes in the graph) on the performance of coarse- and fine-grained parallelization strategies has not been investigated so far. Thus, this work performs a series of experiments and provides speedup analyses of two distinct ACO parallelization strategies compared to a sequential implementation for different TSP configurations and colony sizes. The results show that the considered factors can significantly impact the performance of parallelization strategies, particularly for larger problems. Furthermore, we provide a recommendation for the parallelization strategy and colony size to use for a given problem size and some insights for the development of other GPU-based meta-heuristics.

Keywords. Parallel Ant Colony Optimization, Graphic Processing Units, Parallelization Strategies

1. Introduction

Ant Colony Optimization (ACO) is a well-established algorithm to solve combinatorial optimization problems, such as the TSP, based on the behaviour of foraging ants [4]. Analogously to the ants behaviour in the natural environment, ants in ACO try to find the path (a combination of edges) with the lowest cost (e.g. shortest distance). At each iteration, each ant constructs a complete solution by a step-wise selection of the next edge to traverse based on its pheromone concentration: the higher the concentration, the higher is the probability of an edge of being chosen.

As the number of edges and vertices in the graph increases, the number of possible combinations explodes, thus requiring more calculations for each ant during the path cre-

[1] E-mail: breno.menezes@wiwi.uni-muenster.de

ation process. Since more ants are required to explore a large search space appropriately, the computation becomes even more costly. Hence, the parallelization of ACO is crucial for finding high-quality solutions to large problems in acceptable execution time.

The easy access to high-performance computing hardware, such as multi-core CPUs and GPUs, is promoting much effort towards the parallelization of metaheuristics by optimally splitting the workload among the several cores available, providing maximum occupancy as possible. The challenge is also to avoid overheads, mostly due to the communication between the different memory hierarchies [6], different processes, or different threads.

The different parallel formulations of ACO can be divided into groups according to the strategy applied and the degree of parallelization [8]. The first and most intuitive strategy is to parallelize the work of the different ants. The advantage is a simple, straightforward implementation, since the work of each ant just needs to be assigned to one of several concurrent threads. This approach already enables significant reductions of the execution time. It is often found in literature [2,3,5] and here referred as coarse-grained parallelization. The number of threads generated by the coarse-grained implementations is equal to the number of ants in the colony. This number is typically much lower than the value necessary to provide full occupation of a modern GPU with thousands of cores.

To overcome those drawbacks, fine-grained parallel strategies have been used in other works to explore better all computational power of a modern GPU [1,11,12]. When applied to ACO, this strategy parallelizes (in addition to the ants) the many probability calculations performed inside the ant's path creation process. In this approach, one ant is referred to as a block of threads and each probability calculation is performed in a single thread.

Considering both parallelization strategies mentioned so far, it has not been investigated how the problem configuration and the algorithm configuration affect the performance, and what are the benefits when compared to a sequential implementation. Therefore, in this work, both strategies are tested and evaluated in distinct scenarios. The goal is to state for which cases a fine-grained strategy is more beneficial than a coarse-grained parallelization.

This paper is organized as follow. Section 2 describes the ACO algorithm and the main features of the vanilla version. The parallelization strategies investigated in this work are described in Section 3. In Section 4, we describe our experimental comparison of the different parallelization approaches and the results obtained from them. Conclusions, insights and possible future work are presented in Section 5.

2. Ant Colony Optimization

Like other swarm-based metaheuristics, ACO is based on simple agents (i.e. the ants) that cooperate with each other to find improved solutions. The difference is that, in ACO, individuals do not represent a solution themselves. Instead, they are responsible for constructing solutions at every iteration of the algorithm over the influence of the environment (e.g. pheromones on each edge of the graph). Algorithm 1 presents the basic structure for ACO in its basic implementation for TSP.

The construction of a solution is performed step-by-step in a probabilistic manner. The solutions are built adding components to a partial solution, until a complete solution

Algorithm 1 Sequential ACO algorithm

```
Initialize graph
Initialize pheromone trails
while (Stop criterion is not met) do
  for each ant do
    Construction Solution
  end for
  Pheromone update
end while
```

is reached. In the case of the TSP problem, the probability of visiting a vertex is calculated considering the distance to this vertex and the amount of pheromone present on the edge leading to this vertex. It can be calculated as:

$$p_{i,j} = \frac{[\tau_{i,j}]^{\alpha}[\eta_{i,j}]^{\beta}}{\sum_{l \varepsilon N}[\tau_{i,l}]^{\alpha}[\eta_{i,l}]^{\beta}} \quad \forall j \varepsilon N \tag{1}$$

where $\tau_{i,j}$ is the amount of pheromone between vertices i and j, $\eta_{i,j} = 1/distance_{ij}$, α and β are parameters that determine, how much the pheromone quantity and the distance shall influence the probability, respectively. With all probabilities calculated, the ant is able to choose which vertex will be visited next. The process is repeated until a tour is complete.

The next phase is the pheromone update which can be divided into two parts, evaporation and deposit. Pheromone evaporation removes an equal fraction of pheromone of each edge in the graph (Equation 2). This process is important to decrease the relevance of edges that were visited in earlier stages of the search, thus favouring more exploration of the search space and avoiding getting trap in local optimum.

$$\tau_{i,j} := (1-\rho) * \tau_{i,j} \tag{2}$$

where, $\rho \in [0..1]$ defines the evaporation rate. Afterwards, the pheromone deposit take place. Now, each ant is responsible to deposit pheromone at each edge visited on the tour that was created during the latest tour construction phase. The amount of pheromone is proportional to the quality of the tour constructed by ant the ant that constructed the tour with the shortest path will deposit a higher amount of pheromone (Equation 3).

$$\tau_{i,j} := \tau_{i,j} + \Delta t \tag{3}$$

where Δt is calculated as $1/D_k$ and D_k is the length of the tour generated by ant k.

Over time, the amount of pheromone will increase on the edges that are often contributing to generate the shortest paths. The more pheromone they have, the more likely they will be chosen by ants in the subsequent tour construction phases. The ants will use these high-rated partial solutions to build new tours and possibly improved/shorter ones.

3. GPU-based Ant Colony Optimization

The conception of GPU-based algorithms can be a very complicated and error-prone task. The use of a framework, such as CUDA, becomes necessary to support the develop-

ment and execution of such algorithms. Supported by all Nvidia GPUs, CUDA provides several tools and a programming model for the development of highly parallel applications.

The computing power and aspects of a GPU and CUDA have notable differences compared to CPUs. A GPU is composed of a set of streaming multiprocessors (SM), each containing several cores. Cores inside an SM execute the same instruction simultaneously, following a SIMT schema (Single Instruction Multiple Threads).

GPUs also have different memory management. The main, largest and slowest memory space on the GPU is the global memory. Each SM has its own memory called shared memory. This memory is accessible to all threads running on this SM and it is not accessible for other SMs. Furthermore, each thread has a local memory which is only accessible by itself.

One of the advantages of using CUDA is the clear distinction of sequential and parallel parts of the code. The sequential parts are normally executed by the CPU, while the parallel parts, so-called CUDA kernels, are executed on the GPU. When launching a CUDA kernel, the programmer must specify how many CUDA blocks shall be used and how many threads each block will have. CUDA blocks are groups of threads that will be assigned to the same SM. Inside an SM, each block will be sub-divided into groups of e.g. 32 threads called warps. SMs execute one warp at a time until all threads have been processed.

Despite the bigger amount of cores present in GPUs, some limitations must be also taken into account. For example, when launching kernels, each block is assigned to an SM and blocks that are not allocated due to a lack of resources are queued until resources are available again. Limitations of an SM that determine this behavior are, for example, the maximum number of blocks, the maximum number of threads and the maximum size of data that can be loaded at once.

The implementations mentioned in this work consider all these factors. Algorithm 2 shows the basic template for the GPU-based ACO implementations used here, from the initialization and transfer of data to the GPU, to the parallel execution of the algorithm. The executions of the algorithms themselves run completely on the GPU side, avoiding overhead. [2]

Algorithm 2 Pseudo-code GPU-ACO

```
1: initialize ACO
2: initialize GPU
3: while stop criterion is not met do
4:    tour_construction_kernel <<< n_blocks, n_threads >>>;
5:    pheromone_update_kernel <<< 1, 1 >>>;
6: end while
7: copy results to host
8: clear GPU
```

[2] For simplicity, we here focus on the vanilla version of ACO as presented in Section 2 and do not take into account enhancements as in [9].

3.1. Coarse-grained ACO

The coarse-grained GPU-ACO has as its main idea the parallelization on the ant-level. To implement this strategy for the TSP, the CUDA kernel for the tour construction phase must execute the instructions regarding a single ant's work, as shown in Algorithm 3. Each ant is represented by a thread, which can be captured using CUDA's grid system (line 1). The *while* loop (line 2) is responsible for the tour construction, where in each step the next city to be visited is chosen. The calculation of the probabilities is executed sequentially, following the *for* loop in line 4. Afterwards, each ant decides on the next city using its probability calculations (line 7 and 8).[3]

Algorithm 3 Coarse-grained tour construction kernel

```
1:  ant_i = blockIdx.x * blockDim.x + threadIdx.x;
2:  while tour is not complete do
3:      city_i = get_current_city(ant_i)
4:      for each city_j do
5:          if city_j is a neighbor of city_i and city_j has not been visited yet then
6:              calc_probability(city_i, city_j);
7:          end if
8:      end for
9:      chose_next_city(ant_i);
10: end while
```

Typically, when launching a CUDA kernel, the number of threads per block is set to a multiple of 32, respecting the warp size as mentioned previously. In this implementation, it is not a problem to fit the colony size to a value that fits. Also, the number of blocks can be set to balance the threads among the SMs. For many typical colony sizes, this strategy makes use of a relatively small number of threads compared to what a modern GPU can handle. Thus, the coarse-grained parallelization is often not sufficient to exploit the computing power of a GPU and an additional parallelization approach is needed as presented in the next subsection.

Furthermore, the instructions executed inside the tour construction kernel make it not ideal for a GPU execution. The conditional statements used to check whether the cities are linked and whether the considered city has already been visited (line 5 of algorithm 3), create branches in the algorithm. As threads that belong to the same CUDA block must execute the same instruction at the same time (SIMT), branching makes some threads go idle while other threads execute a certain branch.

3.2. Fine-grained ACO

The fine-grained strategy for GPU-ACO aims at the *additional* parallelization of each ant's tour construction. In this strategy, each CUDA block represents an ant and the tour construction work is distributed among threads as shown in Algorithm 4. By doing so, the probability calculations performed at each step of the tour construction are now executed in parallel. Furthermore, control instructions are now assigned to a single thread (lines 9 and 14).

[3]The pheromone update could be executed in parallel as well. For simplicity, we refrain from doing so here.

Algorithm 4 fine grain parallel tour construction

```
1:  ant_i = blockIdx.x;
2:  city_j = threadIdx.x;
3:  while (tour_is_not_complete(ant_i)) do
4:     city_i = get_current_city(ant_i)
5:     d_eta = compute_inverted_distance(city_i, city_j);     // see line behind Eq. 1
6:     d_tau = compute_pheromone(city_i, city_j);             // see line behind Eq. 1
7:     Synchronize;
8:     if threadIdx.x == 0 then
9:        d_sum = compute_denominator(ant_i);                 // see denominator in Eq. 1
10:    end if
11:    Synchronize;
12:    calc_probability(city_i, city_j, d_eta, d_tau, d_sum); // see Eq. 1
13:    if threadIdx.x == 0 then
14:       chose_next_city(ant_i);
15:    end if
16:    Synchronize;
17: end while
```

This strategy benefits from the higher parallelization level. It naturally sets the quantity of CUDA blocks and threads in a way that better occupies the GPU, when compared to the coarse-grained parallelization, in particular for smaller colonies. A higher number of threads spread among several blocks are ideal for the GPU execution.

Furthermore, the instructions executed by each thread are simpler when compared to the coarse-grained implementation. In this case, only one thread per block is responsible for the instructions with conditional statements. In the coarse-grained implementation, all threads execute conditional statements, leading to the serialization of the execution.

The negative point in this strategy is the lack of control of the number of blocks and threads. As the number of threads per block is equal to the number of available edges in the graph, it is not necessarily a multiple of 32. Also, for bigger colony sizes, the number of blocks that can be allocated simultaneously is easily reached and the remaining blocks are then executed sequentially.

4. Experiments and Results

In order to run the different ACO approaches, an HPC cluster with GPUs was used. The sequential version of the algorithm used the normal partition of the cluster, in which the computing nodes are equipped with Intel Xeon Gold 6140 CPUs with 18 cores (36 threads) running at 2.3GHz (turbo boost frequency up to 3.7GHz) and 192GB RAM. Our GPU-ACO implementations used the partition that is equipped with K20 NVidia Tesla accelerators. Each of these GPUs has 2496 cores running at 706MHz, 5GB memory and they run CUDA 3.5.

The experiments performed in this work include several instances of the TSP problem taken from popular repositories [7,10]. The selected TSP instances have different number of vertices (→ Table 1) so that the behavior of considered ACO implementations are evaluated on various problem sizes.

Table 1. TSPLIB

Instance	dj38	qa194	d198	a280	lin318	pcb442	rat783	lu980	pr1002
# vertices	38	194	198	280	318	442	783	980	1002

Each implementation was tested using different colony sizes (1024, 2048, 4096 and 8192 ants) for all TSP instances. It is important to remark that, in this work, the fitness achieved is not relevant and the focus of the analysis is rather on the execution time. Since all three implementations are based on the same algorithm and just vary on the parallelization strategy, they achieve similar fitnesses when using the same colony size.

The execution time of the sequential implementation of ACO is displayed in Table 2. Table 3 displays the speedup achieved with both parallelization strategies. It is important to mention that the speedup was calculated using the execution time for the whole algorithm. In the case of the GPU execution, it includes not only the tour construction process but also the initialization of the GPU, data transfers from host to device and other calculations.

Table 2. Execution time in seconds - sequential implementation

Problem\Ants	1024	2048	4096	8192
dj38	2.29	4.58	9.16	18.33
qa194	41.24	82.48	164.96	329.93
d198	43.47	86.94	173.88	347.76
a280	87.17	174.33	348.66	697.33
lin318	114.82	229.64	459.28	918.57
pcb442	227.28	454.56	909.13	1818.25
rat783	806.01	1612.03	3224.06	6448.14
lu980	2271.36	4542.71	9085.42	18170.85
pr1002	1529.82	3059.65	6119.30	12238.60

Both parallelization strategies used in this work presented significant speedups. Only for the smallest TSP instance ($dj38$), the coarse-grained ACO was slower than the sequential implementation, and the speedup achieved by the fine-grained implementation was also not so significant. In this case, the amount of work to be processed does not

Table 3. Speedup rates compared to sequential ACO

	Coarse-grained ACO				Fine-grained ACO			
Problem\Ants	1024	2048	4096	8192	1024	2048	4096	8192
dj38	0.47	0.62	0.72	0.71	0.75	0.78	0.79	0.72
qa194	0.96	1.37	1.71	1.93	2.11	2.11	2.03	2.00
d198	0.99	1.41	1.77	2.01	2.13	2.10	2.04	2.01
a280	1.09	1.64	2.23	2.66	2.60	2.65	2.67	2.69
lin318	1.13	1.73	2.41	2.95	2.85	2.93	2.98	3.01
pcb442	1.21	1.96	2.90	3.75	3.08	3.32	3.44	3.51
rat783	1.18	2.09	3.42	4.74	3.20	3.65	3.96	3.92
lu980	2.95	5.18	8.45	11.72	7.97	9.08	9.79	9.67
pr1002	1.22	2.24	3.88	5.81	4.15	4.67	4.99	4.99

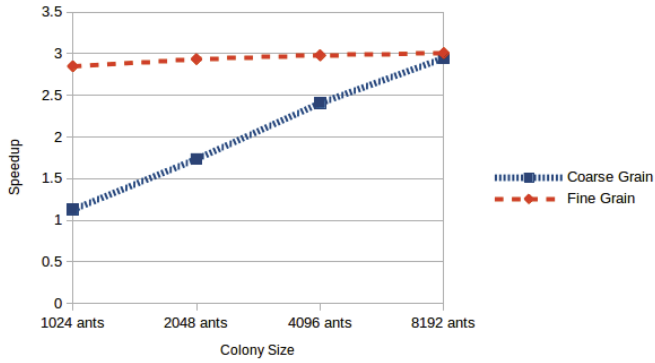

Figure 1. Speedup Comparison - lin318

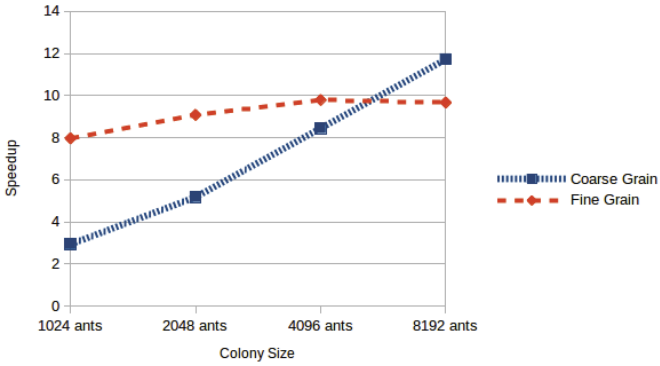

Figure 2. Speedup Comparison - lu980

justify the use of parallel strategies and the extra costs involved (i.e. starting the GPU, transferring data between devices).

Furthermore, in most of the cases, the fine-grained ACO enables a higher speedup when compared to the coarse-grained implementation, since the the later does not provide enough parallelism. The speedup comparison of the *lin*318 instance (Figure 1) shows that the fine-grained implementation achieved a higher speedup for all colony sizes, although the gap between the speedups decreases as the colony size increases.

For TSP instances with a higher number of cities (i.e. *rat*783, *lu*980, *pr*1002), the scenario changes and the coarse-grained implementation enables a higher speedup when using 8192 ants. For 8192 ants, there is now enough coarse-grained parallelism to fully exploit the hardware and, in contrast to the fine-grained implementation, there are no idle cores when combining the results obtained for different neighbor cities as in lines 9 and 14 of algorithm 4 (see Figure 2).

The results show that there is no universal best parallelization strategy between the ones evaluated in this work. The speedup correlates with the parameters of the problem, namely colony size and graph size. These parameters influence directly how the algorithm behaves on the GPU, how the data structures are created and how the multiprocessors are occupied by the program.

For the coarse-grained implementation, running ACO with a smaller colony size over problems with a low number of vertices is not enough to achieve a full occupation of the GPU. This fact explains why the speedup curve for the coarse-grained implementation is ascending with the colony size, while the fine-grained implementation is stable or decreasing. Using the coarse-grained strategy and independently of the TSP instance, a colony of 1024 ants, organized in 32 blocks of 32 threads, is mapped to the GPU cores at once. As the GPU is not fully occupied (the one used in our experiments has 2496 cores), all blocks can be loaded into the processors and execute simultaneously. The same happens for 2048 ants. For 4096 ants, where there are four times more threads, all blocks could still be resident on the streaming multiprocessors. Only with 8192 ants, the limit of resident blocks of the whole GPU is exceeded. In this case, 256 blocks are created, while the maximum number of resident blocks for the whole GPU is equal to 208. Hence not all blocks can be handled simultaneously.

Conversely, the fine-grained implementation makes better use of the high number of GPU cores even for the smallest colony sizes. It profits from creating a high number of blocks, one for each ant, and each block with several threads. Also, the threads processed during the tour construction have a smaller number of instructions and contain less conditional statements, which are serialized on the GPUs.

The fine-grained implementation is only penalized when applying a bigger colony size to a bigger graph. In this case, not only the number of blocks to be created is quite high, but also the GPU is not able to process the same quantity of blocks at the same time, since they need more resources. This scenario leads to an increase in the number of blocks that have to wait until resources are available on the GPU.

5. Conclusion

We have presented an analysis of two different parallelization strategies for the GPU-ACO algorithm applied to the TSP problem. One strategy parallelizes the ants work, while the other parallelizes the internal calculations. Both approaches were applied to distinct TSP instances in order to have a comparison of different scenarios and problem sizes. We also investigated how the colony size impacts the execution time of the algorithm according to the parallelization strategy chosen. The achieved speedup in relation to the sequential implementation was the measure used to compare both strategies.

Our experiments show that the coarse-grained implementation is already capable of generating a significant speedup when compared to the sequential version. The abstraction of parallelizing the work of each ant is quite intuitive and easy to implement, justifying its presence in many other works in literature. On the other hand, it is quite hard to generate enough parallelization (CUDA blocks and threads) to fully occupy a GPU using this coarse-grained strategy. Furthermore, the work performed inside each thread contains several conditional statements, creating branches in the execution of the algorithm, which impacts negatively the execution time.

With the fine-grained strategy, we were able to achieve a higher degree of parallelization, which presented a beneficial influence on the execution time of the algorithm. The additional parallelization of the internal work of each ant promoted a better distribution of processing among the GPU cores available. Furthermore, the work performed by each thread is much simpler than the threads generated by the coarse-grained strat-

egy. The fine-grained threads do not generate branching conditions, which is ideal for the GPU execution (SIMT). The speedup achieved by this strategy was higher than the coarse-grained strategy in most of the cases.

For experiments using a bigger instance of the TSP and a larger colony size, the fine-grained strategy generated so much parallelization that the GPU could not handle it in an optimal way. The number of blocks and threads was just bigger than the amount that could be handled simultaneously by the GPU, leading to a sequential execution of waiting blocks and threads. The overhead generated in these cases impacts the execution time negatively in a way that the coarse-grained was able to present shorter execution times. This fact magnifies the importance of knowing the hardware limitations and the characteristics of the problem being tackled in order to choose the proper parallelization strategy.

As a continuation of this work, we would like to extend the investigation towards other parallelization strategies and enhancements to the algorithm that can work in favour of parallelism. Also, other different parallel applications, such as other meta-heuristics, may benefit from our insights gained by the two-level parallelism. Furthermore, the knowledge acquired during this process will also be used to create a high-level parallelization framework that might be able to adapt itself aiming a better performance.

References

[1] José M. Cecilia, José M. García, Andy Nisbet, Martyn Amos, and Manuel Ujaldón. Enhancing data parallelism for ant colony optimization on GPUs. *Journal of Parallel and Distributed Computing*, 73(1):42–51, 2013.

[2] José M. Cecilia, Andy Nisbet, Martyn Amos, José M. García, and Manuel Ujaldón. Enhancing GPU parallelism in nature-inspired algorithms. *Journal of Supercomputing*, 63(3):773–789, 2013.

[3] Audrey Delévacq, Pierre Delisle, Marc Gravel, and Michaël Krajecki. Parallel ant colony optimization on graphics processing units. *Journal of Parallel and Distributed Computing*, 73(1):52–61, 2013.

[4] Marco Dorigo and Gianni Di Caro. Ant colony optimization: a new meta-heuristic, 1999.

[5] Huw Lloyd and Martyn Amos. Analysis of Independent Roule e Selection in Parallel Ant Colony Optimization. 2017.

[6] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[7] University of Waterloo. National traveling salesman problems. `http://www.math.uwaterloo.ca/tsp/world/countries.html`. Accessed: 14.03.2018.

[8] Martín Pedemonte, Sergio Nesmachnow, and Héctor Cancela. A survey on parallel ant colony optimization. 11:5181–5197, 2011.

[9] Akihiro Uchida, Yasuaki Ito, and Koji Nakano. Accelerating ant colony optimisation for the travelling salesman problem on the GPU. *International Journal of Parallel, Emergent and Distributed Systems*, 29(4):401–420, 2014.

[10] Heidelberg University. Discrete and combinatorial optimization. `https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/XML-TSPLIB/instances/`. Accessed: 14.03.2018.

[11] Fabian Wrede and Breno Augusto de Melo Menezes. High-level Parallel Implementation of Swarm Intelligence-based Optimization Algorithms with Algorithmic Skeletons. In *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing, ParCo 2017*, number September, pages 573—-582, Bologna, Italy, 2017. {IOS} Press.

[12] Fabian Wrede, Breno Augusto de Melo Menezes, and Herbert Kuchen. Fish School Search with Algorithmic Skeletons. In *10th International Symposium on High-Level Parallel Programming and Applications*, pages 1–19, 2017.