# Parallelization Strategies for GPU-Based Ant Colony Optimization Solving the Traveling Salesman Problem

Breno A. M. Menezes
and Herbert Kuchen
Universität Münster
Münster, Germany
breno.menezes, kuchen@uni-muenster.de

Hugo A. Amorim Neto
and Fernando B. de Lima Neto
Universidade de Pernambuco
Recife, Brazil
haan2, fbln@poli.upe.br

***Abstract*—Ant Colony Optimization (ACO) is a well known population-based algorithm used for solving combinatorial optimization problems, such as the Traveling Salesman Problem (TSP). The parallelization of ACO becomes necessary when tackling bigger instances of the TSP due to the high number of calculations performed. Many parallel approaches have been already proposed for ACO, in particular for contemporary high-performance hardware, such as GPUs. Typically, the ants are treated in parallel, since they are largely independent. In the case of the TSP, this concerns in particular the tour construction phase. Furthermore, strategies for parallelizing the pheromone deposit and evaporation phase were proposed. The achieved overall speedup hence depends on a combination of different parallelization strategies, where the impact of each strategy depends on the characteristics of the considered application problem and the hardware used. In the present paper, we aim to compare and analyze the performance of ACO implementations using distinct parallelization strategies when solving TSP instances of different magnitudes. At first, a comparison is made between a coarse-grain and a fine-grain parallel ACO. Furthermore the impact of the parallelization of the pheromone deposit process is also analyzed. The results show that there is no overall best parallelization strategy. Also, they highlight the importance of key-points that lead to a reduction of the execution time, such as the occupancy of the GPU and the work load shared among threads.**

## I. Introduction

The ant colony optimization algorithm was initially proposed by M. Dorigo [1] to solve combinatorial optimization problems, inspired by the behavior of an ant colony in its foraging behavior when searching for food.

ACO was initially applied to solve problems such as the Traveling Salesman Problem [2]. The TSP can be represented by a graph with vertices, edges that connect these vertices and a cost associated with each edge, representing the distance between the corresponding nodes. The goal is to find the shortest round tour visiting each node of the graph exactly once. When applying ACO to this problem, at each iteration, every single ant of the colony creates a complete round tour, also known in literature as Hamiltonian cycle. The fitness of the ant is determined by the sum of the costs of the edges that compose the round tour.

During the tour construction phase, each ant must decide which node to visit next. This decision is based on two factors, the distance and the pheromone quantity between the nodes. Pheromone is a representation of success and its quantity indicates whether an edge has been visited before and how successful were the paths that crossed this edge.

After the tour construction, each ant will deposit pheromone along the created path. Ants that achieved a better fitness deposit more pheromone when compared to other ants. Also, in each round, the pheromone evaporation is responsible for removing a certain quantity of pheromone in every single edge of the graph. This process guarantees that edges that were visited on tours with a good fitness have higher quantities of pheromone, elevating the chances of ants selecting those edges in the next iterations.

In the case of TSP, the number of possible combinations for a complete tour increases exponentially with the number of edges. In order to be able to explore more of these combinations, more ants are needed in the colony, elevating the computational costs of the algorithm. It is well known that ACO spends the largest part of its runtime for path construction [3].

Lately, with the easy access to high-performance hardware, such as GPUs and multi-core CPUs, many parallel versions of swarm intelligence algorithms have been proposed. For ACO this is not different since it is very suitable for parallelization. In particular, the tour construction process is independent for each ant. Therefore, many strategies have also been proposed for ACO aiming to speedup the algorithm [4], [5], [6], [7].

The straight forward strategy is to parallelize the work of each ant, which means the path construction and fitness evaluation steps. Furthermore, implementations can get more complex and increase the level of parallelization, for example, by performing the several probability calculations of each ant at one step of the tour construction simultaneously. These probability calculations are performed by each ant several times for each step during the path construction.

Another possibility is to parallelize the pheromone deposit process [8]. The difficulty is to avoid, for example, the creation

978-1-7281-2153-6/19/$31.00 

of race conditions, since there is the chance that the amount of pheromone in the same edge is being updated by different threads simultaneously. Creating locks manually could lead to serialization, which is not the ideal solution. On the other hand, the use of atomic operations could represent a solution for this problem. Atomic operations are available in the CUDA framework and they guarantee that an operation is not affected by other threads.

When creating a CUDA based parallel implementation of ACO, the programmer must take some factors into account. For example, the limitations of the hardware regarding the number of blocks and threads that can be created on a GPU. These values may vary according to the hardware. A parallelization strategy which works well on one hardware may behave worse on a different hardware.

The focus of the present paper is the investigation of the trade-off of different parallelization strategies for CUDA-based parallel ACO algorithms when applied to distinct instances of the TSP problem with different sizes. The goal is to determine, in which cases it is beneficial to use each strategy. Our experiments include 9 instances of the TSP problem with different sizes [9][10].

This paper is organized as follows. Section II describes the main features of the vanilla version of ACO. Section III includes an overview of parallelization approaches, GPUs, and the CUDA framework. Our experiments are described in Section V. Section VI lists and discusses related work. Our conclusions are presented in Section VII, where we also sketch future work.

## II. Ant Colony Optimization

The original ACO algorithm has two steps that are executed in every iteration, namely $tour\_construction$ and the $pheromone\_update$ as seen in **Algorithm 1**. During the $tour\_construction$ phase, each ant builds a round tour where, starting from one initial node, all nodes of the graph must be visited exactly once. After all ants have constructed a round tour, it is time to perform the $pheromone\_update$. Each ant will deposit pheromone on the edges visited according to the tour that was constructed in the previous step. The amount of pheromone to be deposited is based on how good the tour is compared to the other tours created in the same iteration. Also, there is the natural pheromone evaporation, decreasing a bit the quantity of pheromone present at every edge.

**Algorithm 1** Sequential ACO algorithm

```
1: initialize_ACO();
2: while (stop_criterion_is_not_met) do
3:     tour_construction();
4:     pheromone_update();
5: end while
```

In the $tour\_construction$ phase, each ant must decide which node to visit next. For this step, every node that could be visited is considered and a probability calculation is made. It takes into account the distance between the actual node and the possible next node and also the amount of pheromone that is present at the edge connecting these two nodes (see Equation 1).

**Algorithm 2** Tour Construction

```
1: for Each_Ant do
2:     while (Tour_is_not_complete) do
3:         calculate_probabilities_for_next_step();
4:         calculate_sum_of_probabilities();
5:         chose_next_city();
6:     end while
7: end for
```

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

where $\alpha$ and $\beta$ are the parameters that determine how much the pheromone quantity and the distance between the nodes influence the probability, respectively. $\tau_{i,j}$ is the pheromone at the edge from node $i$ to $j$, $\eta_{i,j}$ is $1/d_{i,j}$ , where $d_{i,j}$ is the distance from node $i$ to $j$. $N$ is the set of nodes that can follow node $i$. $p_{i,j}$ is the probability that the ant goes from node $i$ to $j$.

This way, the more pheromone is present on the route, and the smaller the distance, the bigger the chances of a node to be chosen for the next step. Nodes that have already been visited have no chance of being chosen. The selection is made by generating a random number inside the range of probabilities calculated before. After moving to the next node, each ant must re-calculate all the probabilities again for all nodes that were not visited yet and repeat the selection step until the tour is complete.

After all ants in the swarm have completed the $tour\_construction$, the next phase to be performed is the $pheromone\_update$. The objective in this phase is to deposit pheromone on the edges that were visited by the ants during the $tour\_construction$. The amount of pheromone is proportional to the fitness achieved by the tour when compared to the other ants' tours. Ants that achieve a shorter round tour than the others deposit more pheromone, turning that path more attractive in further path constructions. Also in the pheromone update phase, there is the evaporation of pheromone at all edges based in the parameter $\rho$ as seen in Equation 2:

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

where, $\rho \in [0..1]$ defines the evaporation rate and $\tau_{i,j}$ is the amount of pheromone between nodes $i$ and $j$. Then, each ant deposits pheromone on the routes they have visited, according to Equation 3:

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

where $\Delta t = 1/q_k$, and $q_k$ is the length of the round tour of ant $k$. This way, ants that generate shorter tours deposit more

pheromone at visited edges than the ones that generate longer tours.

It is important to control the pheromone deposit and evaporation. If the ants deposit too much pheromone, it will reduce diversity during path creation and it will be unlikely for ants to create different paths. Also, if the evaporation rate is too high, the influence of successful paths from previous iterations is reduced, representing a loss of success information, reducing the chances of creating better paths based on other paths previously created.

## III. Parallel Computing

### A. Graphic Processing Units

Graphic Processing Units (GPUs) are many-core accelerators suited for compute-intensive tasks. Their main feature is having a parallel throughput allowing the simultaneous execution of multiple threads (see Figure 1). Algorithms that run the *same instructions on many data elements* are perfectly suited for running on GPUs, which is the case for SI algorithms, such as ACO, where the same instructions are executed by each individual.

Modern GPUs can perform vector operations and floating-point arithmetic. More modern GPUs can also manipulate double-precision floating-point numbers. In order to develop programs for such devices, frameworks such as CUDA and OpenCL are commonly used.

CUDA [11] is a platform for parallel computing developed by NVIDIA especially for their GPUs. With CUDA it is possible to run the sequential parts of the algorithm on the CPU, which is optimized for single threaded execution, while the work-intensive, parallelizable parts of the code can be run on the GPU cores in parallel.

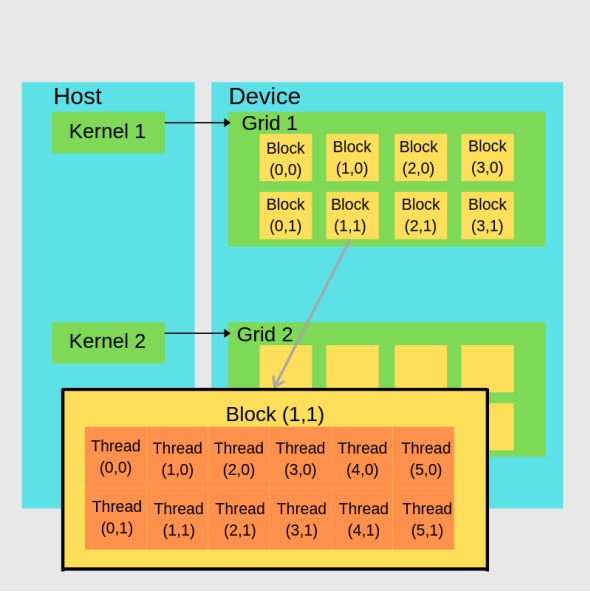


Fig. 1. CUDA Blocks Representation [12]

Parts of the code that are supposed to run in parallel should be implemented as so-called CUDA kernels. These kernels are invoked by the host, launching $n$ CUDA blocks, containing $m$ threads each, and should be distributed among the GPU's streaming multi-processors (SMs). Each of the streaming multi-processors contains a certain number of registers and CUDA cores. Using these resources, the SMs are responsible for handling CUDA blocks and executing thread warps, which are groups of threads that belong to a CUDA block.

When using CUDA, some attention in necessary to some specifications of the hardware. For example, the number of resident blocks and threads of the SMs. These values are referring to the amount of blocks and threads that can be loaded at the same time onto an SM. When these limits are not respected, overhead is generated because the blocks that were not loaded into a SM at first have to wait in a pipeline until there are enough free resources on an SM to execute them. A SM only loads a new CUDA block if there are enough resources for it. Therefore, it must first finish executing all threads from the block that is already loaded, in order to free the resources and then load a block that is in the waiting line.

One of the main concerns while developing using CUDA is the memory access. All the information needed by the threads must be previously transmitted from the main memory to the GPU memory. Frequent data exchange between these memories reduces the performance.

For the programmer, a CUDA program is typically structured as kernels that are executed by thread blocks. Each block has its own local memory which can be accessed more quickly than the global GPU memory. The local memory can only be reached by cores of the corresponding block. Thus, an important implementation decision when programming with CUDA concerns a reasonable distribution of data between local memory and global memory.

Furthermore, CUDA supports multi-GPU programming. It allows to further speedup the computation by increasing the number of used GPUs and hence cores. Also, this approach can be used if one GPU has not enough memory to handle the required data. In this case, there are two possible architectures: i) a single compute node contains (in addition to its CPU(s)) all the GPUs or ii) the GPUs are assigned to different compute nodes, each with different CPUs and memory.

When using multiple GPUs, the programmer has to worry about the communication and ensure that the data are transferred as required between different GPU memories or between main memory and certain GPU memories. The CUDA instruction $cudaSetDevice()$ opens a channel for data transfers. Also, the $cudaMemcpyAsync(...)$ instruction can be used to achieve asynchronous, simultaneous data transfers.

To conclude, the key aspect to improve parallel computing using CUDA is to optimize the data exchange between host CPU and GPUs and maximize the occupancy of the available cores.

## IV. Parallel GPU strategies for Ant Colony Optimization

When creating GPU algorithms, the CUDA platform is frequently used. With CUDA, it is possible to run the sequential

part of the code on the CPU and then allocate the GPU to run the computationally intensive parts of the algorithm. CUDA provides synchronization instructions that can be used to control the threads.

An obvious parallelization strategy of the TSP for ACO is to just let all ants construct their tours in parallel. This parallelization strategy, also known as *coarse-grained* parallel, takes advantage of the high amount of threads that can be created and distributed onto several CUDA-blocks. Global memory has to be used for the graph, since using local memory parts of it would not be reachable for non-local ants.

When just relying on the coarse-grained parallelization, the amount of work of each ant is still considerable, in particular for large graphs, since for every remaining node probability calculations must be executed by each ant. One option is to additionally parallelize the computations of the probabilities, since they are also independent of each other. By doing so, we increase the degree of parallelism to a so called *fine-grained* strategy. Each CUDA block represents one ant and its threads perform in parallel the internal calculations. A disadvantage of this approach is that the maximal number of blocks that can be created on a GPU is limited. Also, the occupancy of the GPU is not always optimal, since the thread numbers would vary along the tour construction phase steps. It occurs because, when developing CUDA applications, the programmer must also be aware of the warp size, which is the number of threads that are executed in parallel inside an SM. If e.g. the warp size is 32 and only 16 threads are actually used, it means that 16 threads are wasted. The optimal use of cores can hardly be guaranteed using a fine-grained parallelization for ACO, since at each step there is one node less to calculate the probability of being visited.

Most of the parallelization strategies for ACO have as the main focus the tour construction phase, mainly because it is the most demanding and time consuming phase, but also because during the pheromone deposit phase, one single structure is being updated, which is the pheromone matrix. This scenario must be handled with much care in order to avoid race conditions, where two ants would be updating the same pheromone edge at the same time. In modern GPUs, that support the most recent CUDA version, there is the possibility of using powerful features that could help avoiding race conditions. One of them is the use of atomic operations. They guarantee that a certain operation is performed by a single thread without interference of others. By using this feature, it is also possible to create a parallel pheromone update kernel. The down side of using such operations is that in the (unlikely) case that many threads are trying to update a value at the same time, some serialization may be observed.

The following sections give more details of the ACO implementations used in this work. The three implementations make use of the features and strategies mentioned above.

### A. Coarse-grained GPU-ACO approach for the TSP

The coarse-grained GPU-ACO was implemented aiming at the association of one ant with one thread distributed into several CUDA blocks. The pseudo-code for this implementation can be observed in **Algorithm 3**.

**Algorithm 3** Pseudo-code GPU-ACO of TSP

```
1: initialize_ACO
2: initialize_GPU
3: n_threads = warp_size;
4: n_blocks = n_ants/warp_size;
5: while (stop_criterion_is_not_met) do
6:     tour_construction_kernel
   <<< n_blocks, n_threads >>> ();
7:     synchronize;
8:     pheromone_update_kernel <<< 1, 1 >>> ();
9: end while
10: copy_Results_to_Host();
11: clear_GPU();
```

During the execution of CUDA-based algorithms, a major source of overhead is the exchange of information between the GPU and the CPU. Therefore, once the data structures are created (i.e. node coordinates matrix, tours matrix, random number generators, ...), everything is transferred to the GPU and processed there using CUDA kernels.

The main CUDA kernel in these implementations, the tour construction kernel, is responsible for the tour construction of each ant. The kernel is initialized in a way that each thread will do the work for a single ant. The indexation is done through CUDA's block ids and thread ids. Inside the kernel, there is a main loop containing the probabilities calculations and the selection of the next node (city). The loop ends when a complete tour is finished. A representation of the tour construction kernel can be seen in **(Algorithm 4)**;

**Algorithm 4** Coarse-grained parallel tour construction

```
1: ant_i = blockIdx.x * blockDim.x + threadIdx.x;
2: while (Tour_is_not_complete(ant_i)) do
3:     city_i = get_current_city(ant_i)
4:     for each_city do
5:         calc_probability(city_i, city_j);
6:     end for
7:     d_sum(ant_i)
8:     chose_next_city(ant_i);
9: end while
```

The main advantage of Algorithm 4 is its simplicity. Moreover, considerable speedups can be achieved compared to sequential implementations.

Modern GPUs are able to run many CUDA blocks with up to 1024 threads per block. Having in mind that the warp size is equal to 32, it means that, while executing a CUDA kernel, blocks are divided into smaller groups of 32 threads that run simultaneously. Therefore, it is wise to have a multiple of 32 as the number of threads per block. Also, for TSP instances with a higher number of nodes, the sequential calculations can be quite demanding. The fine-grained parallelization strategy can by applied to overcome this problem.

### B. Fine-grained GPU-ACO approach for the TSP

As mentioned, the fine-grained TSP implementation parallelizes not only the work of each ant during the tour construction phase, but also the (independent) internal probability calculations that will be used for the selection of the next node of the graph to be visited.

The fine-grained tour-construction kernel, represented in **Algorithm 5**, delivers the same results as the coarse-grained algorithm mentioned before. The main difference is regarding the indexation, since now each block represents an ant and each thread is used for calculating a probability. Also, synchronization points are necessary due to variables that require reduction.

**Algorithm 5** 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(city_i, city_j);
6:      d_tau(city_i, city_j);
7:      if threadIdx.x == 0 then
8:          d_sum(ant_i);
9:      end if
10:     calc_probability(d_eta, d_tau, d_sum);
11:     Synchronize;
12:     if threadIdx.x == 0 then
13:         chose_next_city(ant_i);
14:     end if
15: end while
```

Synchronization points are necessary during the route construction in the fine grain strategy. They guarantee, for example, that the sum of probabilities is computed only after all threads have finished calculating the probability values for their respective cities (lines 5 to 9). Only one thread of a block is responsible for calculating the sum of these results. Meanwhile, all other threads remain idle.

### C. Parallel pheromone update

After each ant has created a complete tour, it is time to perform the pheromone update. During this phase each visited edge receives an amount of pheromone which proportional to the fitness achieved by the ant's complete tour. Furthermore, every edge looses a bit of pheromone by a constant factor, which represents the pheromone evaporation.

Normally, this process is performed in a sequential manner, since race conditions could be created when running in parallel. Recent improvements on CUDA's tools, such as atomic operations, make possible to overcome this problem. The parallelization strategy for the pheromone update is detailed in the following.

The parallelization of the pheromone evaporation step is quite straightforward. Since, each operation is performed at a different edge, there are no race conditions and we can assign one thread as being responsible for updating a single edge in the matrix (**Algorithm 6**). In order to take maximum profit from the GPU, the threads are divided into many blocks.

**Algorithm 6** Parallel pheromone-evaporation kernel

```
1: edge = blockIdx.x * blockDim.x + threadIdx.x;
2: phero[edge] = (1 - RO) * phero[edge];
```

The parallel pheromone deposit is performed using CUDA's atomic operations (**Algorithm 7**).

**Algorithm 7** parallel pheromone update kernel

```
1: ant_k = blockIdx.x;
2: step_id = threadIdx.x;
3: city_i = route[step_id];
4: city_j = route[step_id + 1];
5: calc_delta_pheromone()
6: atomicAdd(pheromone_ij, deltapheromone)
```

Similarly to the fine-grained strategy used in the tour construction phase, the pheromone update kernel uses blocks to represent ants and each thread of this block is responsible for updating one edge visited in the tour. The positive side of this strategy is the high degree of parallelization, where all visited edges are updated at once.

On the other hand, the use of atomic operations can lead to a loss of performance in the (unlikely) case that many threads would be trying to update the same value at once. Note that not all GPUs support this operation and that without it the programmer has to use other (more expensive) means to ensure mutual exclusion.

## V. Experiments

The experiments performed to evaluate the mentioned parallelization strategies for ACO are detailed in this section. Our goal was to evaluate and compare the performance of each strategy in different scenarios. Therefore, we have chosen different TSP instances with different magnitudes (Table I [9][10]).

TABLE I
TSPLIB

| Instance | # Nodes | Optimal Tour (km) |
|---|---|---|
| dj38 | 38 | 6656 |
| lu980 | 980 | 11340 |
| qa194 | 194 | 9352 |
| a280 | 280 | 2579 |
| d198 | 198 | 15780 |
| lin318 | 318 | 42029 |
| pcb442 | 442 | 50778 |
| pr1002 | 1002 | 259045 |
| rat783 | 783 | 8806 |

These TSP instances have been chosen due to the variety of problem sizes they offer.

In order to run the algorithms, an HPC cluster was used. Each GPU node is equipped with a K20 NVIDIA Tesla accelerators with 2496 cores and CUDA 3.5 capability.

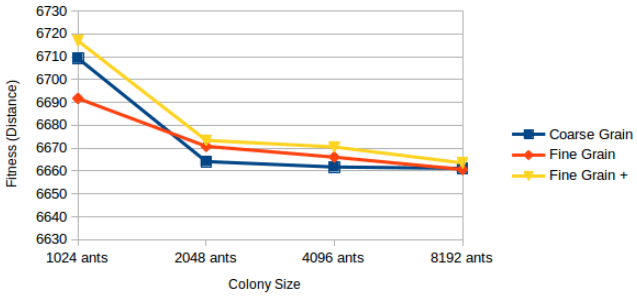


Fig. 2. Fitness Comparison - dj38

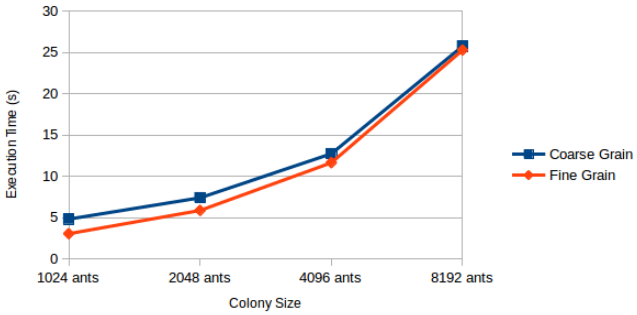


Fig. 3. Coarse Grain vs. Fine Grain - dj38

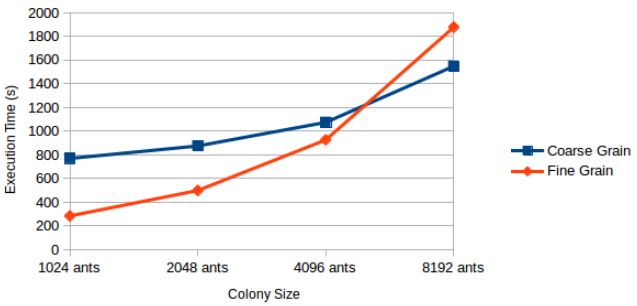


Fig. 4. Coarse Grain vs. Fine Grain - lu980

The 3 strategies were implemented and applied for each instance mentioned above. Also, in order to expose the implementations to different scenarios, our tests include also the use of different colony sizes (1024, 2048, 4096 and 8192 ants). Although the colony size is bigger than the necessary for some TSP instances, it allows us to have a better perspective regarding the limitations of the algorithm and hardware in evidence. For each experiment, 30 executions were performed. The execution times displayed in the results are the average regarding these 30 executions.

### A. *Results*

Our focus is on the relationship between execution time and parallelization strategy. For all TSP instances, the quality of the reached solutions are similar for the same colony sizes, regardless of the parallelization strategy (Figure 2). The execution times for each experiment are displayed in Table II.

Comparing the coarse-grained and the fine-grained parallelization strategies, our results show that there is not a single best strategy and the performance varies according to the set up of the experiment. More specifically, the performance depends on the number of nodes present in the TSP graph and the colony size.

Figures 3, 4 and 5 show the graphs comparing both implementations for three instances of the TSP with different numbers of nodes. For the dj38 instance (Figure 3), where the graph has only 38 nodes, the execution times for both strategies are very similar. The fine-grained implementation is 1.58x faster than the coarse-grained one when using 1024 ants. But the difference between both decreases when the colony grows, reaching 1.02x when using 8192 ants.

On the other hand, for bigger instances of the problem, such as the lu980 instance (980 nodes) (Figure 4) and pcb442 (442 nodes) (Figure 5), the fine-grained implementation is much faster when using 1024 ants (2.70x and 2.55x respectively). The difference between the execution times still decays as the number of ants in the colony is increased. The difference this time is that actually the coarse-grained implementation has a shorter execution time when the colony has 8192 ants.

The difference between both implementations compared so far can be attributed to how they distribute the CUDA blocks and threads among the GPU's SMs.

Considering the characteristics of the GPU used in this experiment, there are 2496 cores available distributed among 13 streaming multi-processors (SMs). As mentioned before, each streaming multi-processor has limitations regarding the number of blocks and threads that can be loaded at once. In this case, the limits are respectively, 16 blocks and 2048 threads per SM. This means that 16 blocks with 128 threads each could occupy a streaming multi-processor at the same time. The same would happen if there were 8 blocks with 256 threads each.

When a CUDA kernel is launched, the corresponding blocks are distributed among the SMs respecting the limitations mentioned above. If the amount of blocks is more than the SMs from the GPU can process at one time, the remaining unallocated blocks enter a pipeline. These blocks shall later be processed one after the other when resources become available on SMs.

With the coarse-grained implementation, for example, the block size was set to 32, which is exactly the warp size for the GPU, and the number of threads is equal to the number of ants in the colony. Using this setup, independently of the TSP instance, with a colony of 1024 ants, 32 blocks of 32 threads each would be distributed among the SMs at once. As the GPU is not fully occupied (it has 2496 cores), all blocks can be loaded into the SMs and execute simultaneously. The same happens for 2048 ants. For 4096 ants, where there are four times more threads, but all blocks could still be resident on the streaming multiprocessors. Only with 8192 ants, the limit

TABLE II
EXECUTION TIME IN SECONDS

| Problem\Ants | Coarse Grain | | | | Fine Grain | | | | Fine Grain + Parallel Pheromone Deposit | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1024 | 2048 | 4096 | 8192 | 1024 | 2048 | 4096 | 8192 | 1024 | 2048 | 4096 | 8192 |
| dj38 | 4.822 | 7.406 | 12.738 | 25.764 | 3.058 | 5.872 | 11.652 | 25.292 | 0.674 | 1.072 | 1.866 | 5.566 |
| lu980 | 770.712 | 876.46 | 1075.424 | 1550.1392 | 285.056 | 500.166 | 928.046 | 1879.528 | 167.692 | 317.312 | 619.804 | 1312.816 |
| qa194 | 43.024 | 60.314 | 96.658 | 170.598 | 19.504 | 39.012 | 81.064 | 164.78 | 4.61 | 11.43 | 27.804 | 60.32 |
| a280 | 80.238 | 106.062 | 156.41 | 262.5 | 33.466 | 65.778 | 130.316 | 259.216 | 10.496 | 24.214 | 51.474 | 106.462 |
| d198 | 43.586 | 61.398 | 97.938 | 173.24 | 20.444 | 41.324 | 85.198 | 173.138 | 5.152 | 12.892 | 30.52 | 65.876 |
| lin318 | 101.904 | 132.486 | 190.93 | 311.756 | 40.32 | 78.27 | 154.15 | 305.586 | 13.618 | 30.428 | 64.028 | 131.49 |
| pcb442 | 188.248 | 231.42 | 313.582 | 484.484 | 73.752 | 137.068 | 264.016 | 517.146 | 33.682 | 67.954 | 136.31 | 273.63 |
| pr1002 | 1249.585 | 1365.705 | 1576.94 | 2104.645 | 368.94 | 654.735 | 1226.17 | 2458.435 | 246.735 | 470.03 | 910.895 | 1887.57 |
| rat783 | 681.93 | 772.022 | 941.468 | 1358.762 | 251.825 | 441.409 | 812.323 | 1646.246 | 130.86 | 254.032 | 499.392 | 1054.184 |

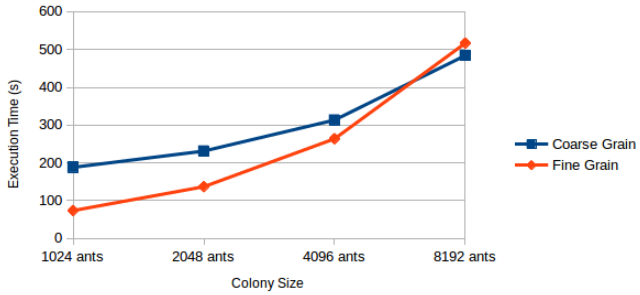


Fig. 5. Coarse Grain vs. Fine Grain - pcb442

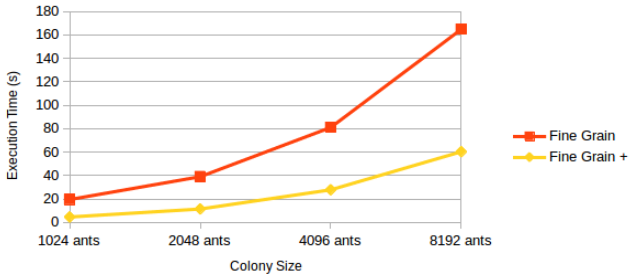


Fig. 6. Fine Grain vs. Fine Grain + Parallel Pheromone Update - qa194

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. This explains why the runtime increases significantly when the colony size changes from 4096 to 8192 ants.

Using the coarse-grained implementation, only in extreme cases the GPU is fully occupied and the blocks are competing for resources. Even so, the work load of a single thread can be quite high. In case of the TSP, the workload during the path construction it is directly related to the graph size.

The fine-grained implementation and the higher level of parallelism may lead to a shorter execution time, even though the number of blocks and threads created exceed the values of maximum resident blocks and threads specified for each GPU. The fine-grained implementation replaces the loop over all remaining cities by parallelism. Thus, it saves the overhead of the loop control. On the other hand, there is so much parallelism that it causes a significant overhead for block switching. As more ants exist in the colony, more overhead is generated. This fact explains a steeper slope of the resulting curve for the execution time. Eventually, it will pass the execution time of the coarse-grained approach as shown in e.g. Figure 4.

In addition, as the number of threads vary according to the number of cities that have to be considered, the number of threads per block is not controlled. This lack of control leads to the sub-utilization of the resources and loss of performance.

Analyzing further, Figure 6 shows the execution times for qa194 (194 cities) using the fine-grained strategy with and without parallel pheromone update. The results show a shorter execution time when using the parallel pheromone update and a progressive grow of the difference between the execution times as the ant colony grows. The same results are observed in all TSP instances tested. Since simultaneous attempts to update a value rarely occur, CUDA's atomic operations do not cause a significant sequentialization.

## VI. RELATED WORK

Many parallelization strategies have been proposed for GPU parallel ACO implementations. Cecilia et al. [4], [13], for example, focus on data parallelism and making use of memory hierarchy to improve and reduce the execution time of the algorithm. First they propose a better organization of data in order to ease the access and reduce branch divergences in the execution.

Lloyd and Amos [14] investigate the performance of distinct roulette selection mechanisms in parallel ant colony optimization. They state that the so called I-Roulette mechanism accelerates the algorithm.

Also aiming at GPU architectures, other parallelization strategies have been proposed by Uchida et al. [8]. Similarly to the work done by Cecilia et al. [4], they try to optimize the data access and also creating other data structures that would help reducing the execution time of the algorithm.

Other levels of parallelization are analyzed in the works of Fan [15] and Abdelkafi et al. [7]. They propose a multi-level parallel ACO-based algorithm. Their idea is to exploit

the resources offered by modern hardware to have not one but several colonies running in parallel. They profit from the high processing power hardware and the exchange of information between the colonies to generate better solutions.

The mentioned articles do not compare coarse-grained and fine-grained parallelization strategies.

## VII. Conclusions

This paper provides an analysis and comparison of different parallelization strategies for TSP algorithms based on Ant Colony Optimization using CUDA. Many parallelizations have been already proposed. Most of them focus on speeding up the tour construction phase of the algorithm.

The parallelization can be performed on different levels. A coarse-grained approach just treats different ants in parallel, since their computations are independent of each other. A fine-grained approach also parallelizes internal computations of an ant such as the selection of the next node to visit when solving a TSP. In addition to such embarrassingly parallel computations, also other phases of an algorithm can be handled in parallel, which require some communication or synchronization, such as the pheromone deposit phase. Tools provided by the CUDA framework, such as atomic operations, can be used here.

Our experimental results show that the key point to reduce the execution time of the algorithm is to be aware of the GPU capabilities and the problem being tackled. Based on these aspects, the programmer can choose the best parallelization strategy for a given problem. A coarse-grained approach may not provide enough parallelism to fully exploit the available hardware, while a fine-grained approach might introduce additional overhead. A two-level parallelization might also have difficulties to exhaustively use the available cores, in particular when the amount of parallelism on the second level varies over time. As a rule of thumb, a coarse grained implementation is preferable, if it provides enough parallelism. If not, a two-level parallelization might be better, if it does not introduce too much overhead.

In any case, the parallelization of all phases of an algorithm should be attempted. In case of the TSP, this means that in addition to the tour construction also the pheromone deposit phase and the evaporation phase should be parallelized, even if they require some synchronization or communication. In case of CUDA, atomic operations can help here. When considering the pheromone deposit phase of a TSP, the tours created by each ant at a certain iteration tend to be different. Thus, it is very unlikely that different ants try to update the same edge at the same time. Hence, a significant serialization of these updates due to atomic operations could not be observed in our experiments.

As a continuation of this work, we aim to extend the analysis and comparison to other parallelization strategies. We also want to consider other types of application problems that could be tackled with ACO or other swarm intelligent metaheuristics. Furthermore we want to incorporate the obtained insights into a high-level parallelization framework, which could be a tool used to facilitate the development of parallel swarm intelligent algorithms.

## References


[1] M. Dorigo and G. Di Caro, "Ant colony optimization: a new meta-heuristic," pp. 1470–1477, 1999. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=782657

[2] J. Lawler, I. Lenstra, M. Rinnooy, and K. Skadron, "The traveling salesman problem," *Queue*, vol. 6, no. 2, p. 463 pp, 1985.

[3] E.-G. Talbi, *Metaheuristics*. Hoboken, NJ, USA: John Wiley & Sons, Inc., jun 2009. [Online]. Available: http://doi.wiley.com/10.1002/9780470496916

[4] J. M. Cecilia, A. Nisbet, M. Amos, J. M. García, and M. Ujaldón, "Enhancing GPU parallelism in nature-inspired algorithms," *Journal of Supercomputing*, vol. 63, no. 3, pp. 773–789, 2013.

[5] H. Fingler, E. N. Cáceres, H. Mongelli, and S. W. Song, "A CUDA based solution to the Multidimensional Knapsack Problem using the ant colony optimization," *Procedia Computer Science*, vol. 29, no. 30, pp. 84–94, 2014.

[6] Y. Zhang and G.-D. Li, "Using Improved Parallel Ant Colony Optimization Based on Graphic Processing Unit-Acceleration to Solve Motif Finding Problem," *Journal of Computational and Theoretical Nanoscience*, vol. 11, no. 3, pp. 878–882, 2014. [Online]. Available: http://openurl.ingenta.com/content/xref?genre=article&issn=1546-1955&volume=11&issue=3&spage=878

[7] O. Abdelkafi, J. Lepagnot, and L. Idoumghar, "Multi-level Parallelization for Hybrid ACO," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, ser. Lecture Notes in Computer Science, P. Siarry, L. Idoumghar, and J. Lepagnot, Eds. Cham: Springer International Publishing, 2014, vol. 8472, pp. 60–67. [Online]. Available: http://link.springer.com/10.1007/978-3-319-12970-9 http://link.springer.com/10.1007/978-3-319-12970-9_7

[8] A. Uchida, Y. Ito, and K. Nakano, "Accelerating ant colony optimisation for the travelling salesman problem on the GPU," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 29, no. 4, pp. 401–420, 2014.

[9] U. of Waterloo. (2018) National traveling salesman problems. [Online]. Available: http://www.math.uwaterloo.ca/tsp/world/countries.html

[10] H. University. (2018) Tsplib. [Online]. Available: https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/XML-TSPLIB/instances/

[11] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[12] NVIDIA. (2018) Cuda c programming guide. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[13] J. M. Cecilia, J. M. García, A. Nisbet, M. Amos, and M. Ujaldón, "Enhancing data parallelism for ant colony optimization on GPUs," *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 42–51, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2012.01.002

[14] H. Lloyd and M. Amos, "Analysis of Independent Roule e Selection in Parallel Ant Colony Optimization," 2017.

[15] F. Li, "GACO: A GPU-based High Performance Parallel Multi-ant Colony Optimization Algorithm," *Journal of Information and Computational Science*, vol. 11, no. 6, pp. 1775–1784, 2014. [Online]. Available: http://www.joics.com/publishedpapers/2014_11_6_1775_1784.pdf