# A GPU Implementation of MOEA/D-ACO for the Multiobjective Traveling Salesman Problem

Murilo Zangari de Souza, Aurora Trinidad Ramirez Pozo
DInf - Federal University of Parana, CP: 19081, CEP 19031-970, Curitiba, Brazil
Email: {murilo.zangari, auroratrinidad}@gmail.com

***Abstract*—Several Ant Colony Optimization (ACO) algorithms have been proposed to solve multiobjective problems. MOEA/D-ACO is a multiobjective algorithm based on ACO and the decomposition approach and presents best results when compared with MOEA/D and BiCriterion on discrete optimization problems. This paper proposes a new parallel implementation of MOEA/D-ACO for execution on the Graphics Processing Unit (GPU) using NVIDIA CUDA in order to improve the efficiency, reaching high quality results in a reasonable execution time. Based on recent researches, both the solution construction and pheromone update phases are implemented using a data parallel approach. We use the multiobjective Traveling Salesman Problem (MTSP) as application. We report speedups of up to 11x from the sequential implementation. Moreover, the results point out that: the number of objectives and the number of subproblems affect directly the speedup.**

***Keywords*—Ant Colony Optimization (ACO), MOEA/D-ACO, Graphics Processing Units (GPU), multiobjective traveling salesman problem (MTSP).**

## I. Introduction

Ant Colony Optimization (ACO) is a population-based metaheuristic inspired by the pheromone trail laying and following behavior of some real ant species. ACO was originally designed for solving single-objective combinatorial optimization, and it was first applied to the Traveling Salesman Problem (TSP) by Dorigo et al. [1] [2]. ACO has proven to be one of the most successful algorithms for modeling discrete optimization problems. Due to notable results on these applications, ACO algorithms were soon extended to tackle problems with more than one objective (MOPs), called multiobjective ACO (MOACO) [3], most of these algorithms have different design choices and focus in terms of Pareto optimally, that is, they do not make a priori assumptions about the decisions maker's preferences.

MOEA/D [4] (Multiobjective Evolutionary Algorithm based on Decomposition) is a recent evolutionary algorithm for multiobjective optimization using the decomposition idea. MOEA/D outperforms to Multiobjective Genetic Local Search (MOGLS) and Non-dominated Sorting Genetic Algorithm II (NSGA-II). Extensions of MOEA/D have been applied for solving a number of MOPs. Recently in [5] an extension, called MOEA/D-ACO was proposed where each ant is responsible for solving one subproblem. The algorithm was applied on the multiobjective 0-1 knapsack problem (MOKP) and on the bi-objective traveling salesman problem (b-TSP) and achieved better results than MOEA/D and BicriterionAnt [6] respectively.

Several GPU (Graphics Processing Unit) parallelization approaches have been proposed for the ACO to solve single-objective problems [7] [8] [9] [10]. These researches improve the performance achieving high quality results in a reasonable execution time. Some studies proposed a CPU-based approach to parallel MOACOs [11] [12] applied on bi-TSP, and also a CPU-based to parallel MOEA/D [13] applied on continuous MOPs. These studies not achieved a high speedup due to target hardware. In [14] the author proposed a parallel MOEA on GPU applied on continuous MOPs and achieved speedups range from 5.62x to 10.75x. Parallel MOACO algorithm on GPU is a recent and open research field.

Moreover, ACO implementation mainly consists of two stages: *tour construction* and *pheromone update* (an optional local search stage may also be applied to improve the quality of the tours). The process of tour construction and pheromone update is applied iteratively until a termination condition is met (such as a set number of iterations). Both the tour construction and pheromone update stages can be performed independently for each ant and it makes ACO particularly suited to parallel on GPU - Graphics Processing Units [7]. However, the parallel implementation of multiobjective ACO has not been explored yet.

NVIDIA introduced CUDA (Compute Unified Device Architecture), a general purpose parallel computing platform and programming model for direct execution on GPU to solve many complex computational problems in a more efficient way than on a CPU. CUDA exposes the GPU's massively parallel architecture so parallel code can be written to execute faster than its optimized sequential counterpart [15].

This paper describes a parallel implementation of MOEA/D-ACO that is suitable for GPU computing environments using CUDA where both stages are parallelized: the *solution construction* and the *pheromone update*. The paper also presents results from the application on the multiobjective Traveling Salesman Problem (MTSP). The speedup results are compared to the sequential implementation. Moreover, many evaluations are made for different number of the objectives and number of subproblems.

The remainder of this paper is organized as follows. Section II introduces MOP, the Decomposition Approach, and the multiobjective traveling salesman problem; finally the section describes the MOEA/D-ACO proposed by [5]. Section III describes the parallel implementation of MOEA/D-ACO. Section IV reports the experimental results. And finally, the conclusion and future work are presented in Section VII.

## II. MOEA/D-ACO

This section reviews concepts of multiobjective optimization, the MTSP, Decomposition approach and finally the MOEA/D-ACO [5] algorithm.

### A. MOP definition

A general MOP can be state as follows [16]:

$$\textit{minimize (or maximize) } F(x) = (f_1(x), ..., f_m(x)) \quad \textit{subject to } x \in \Omega \tag{1}$$

where $x = [x_1, x_2, x_n]^T$ is the decision variable vector, $\Omega$ is the decision (variable) space, $F : \Omega \rightarrow R^m$ consist of *m* real-valued objective functions, and $R^m$ is called the *objective space*. The *attainable objective set* is defined as set $\{F(x)|x \in \Omega\}$.

Let the vectors $u, v \in R^m$, $u$ is said to *dominate* $v$ if and only if $u_i \leq v_i$ for every $i \in \{1, ..., m\}$ and $u_j < v_j$ for at least one index $j \in \{1, ..., m\}$[1]. Point $x^* \in \Omega$ is *Pareto optimal* if there is no point $x \in \Omega$ so that $F(x)$ dominates $F(x^*)$. $F(x^*)$ is then called a *Pareto optimal (objective) vector*. In other words, any improvement in the one objective of a Pareto optimal point must lead to deterioration to at least another objective. The set of all the Pareto optimal is called the *PS*, and the set of all the Pareto optimal objective vectors is called the *PF*.

### B. MTSP

Given a set of *n* cities and *m* symmetric matrices $(c^i_{k,l})_{n\times n}, i = 1, ..., m$, where $c^i_{k,l} > 0$ can be interpreted as the *i*th cost of traveling between cities *k* and *l*. The problem is as follows:

$$\textit{minimize } f_i(x) = \sum_{k=1}^{n-1} c^i_{x_k,x_{k+1}} + c^i_{x_1,x_n} \tag{2}$$

where *x* is a permutation of *n* cities, representing a tour that visits each city once.

### C. MOP Decomposition

As the algorithm MOEA/D-ACO use the decomposition approach, the *Weighted Sum Approach* [4] is described below. This is one of most commonly used approaches.

*Weighted Sum Approach:* Let $\lambda = (\lambda_1, ..., \lambda_m)^T$ be a weight vector, i.e., $\sum_{i=1}^{m} \lambda_i = 1$ and $\lambda_i \geq 0$ for all $i = 1, ..., m$. Then, the optimal solutions to the following single-optimization problems:

$$\textit{minimize (or maximize) } g^{ws}(x|\lambda) = \sum_{i=1}^{m} \lambda_i f_i(x) \quad \textit{subject to } x \in \Omega \tag{3}$$

is a Pareto optimal point to (1) if the *PF* of (1) is convex (or concave), where we use $g^{ws}(x|\lambda)$ to emphasize that $\lambda$ is a weight vector in this objective function, while *x* is the variable to be optimized. To generate a set of different Pareto optimal vectors, one can use different weight vectors $\lambda$ in the above scalar optimization problem [4].

[1]This definition of domination is for minimization. For maximization the inequalities should be reversed

### D. The algorithm MOEA/D-ACO

MOEA/D-ACO algorithm decomposes a MOP into *N* single-objective subproblems by choosing *N* weight vectors $\lambda^1, ..., \lambda^N$. Subproblem *i* is associated with weight vector $\lambda^i$ and its objective function is denoted as $g(x|\lambda^i)$. MOEA/D-ACO employs *N* ants for solving these single-objective subproblems. Ant *i* represents the subproblem *i*. The algorithm has two concepts:

- **Neighborhood:** $B(i)$ of ant *i* contains *T* ants, where *T* is the size of the neighborhood. The neighbors of ant *i* are *T* closest to $\lambda^i$ among all the *N* weight vectors;
- **Groups**: where *N* ants are divided into *K* groups by clustering their corresponding weight vectors. The ants in the same group share one pheromone matrix. Each group has a pheromone matrix, which contains learned information about the position of their Pareto, i.e., each group is intended to approximate a small range of the PF.

The algorithm maintains: (I) $\tau^1, ..., \tau^K$, where $\tau^j$ is the current pheromone matrix for group $j$, storing its learned knowledge about the sub-region of *PF* that it aims at approximating; (II) $\eta^1, ..., \eta^N$, where $\eta^i$ is the heuristic information matrix for subproblem *i*, which is predetermined before the construction solution starts; (III) *EP*, which is the external archive containing all the non-dominated solutions found so far. At each iteration the MOEA/D-ACO executes the following steps:

1) Generate *N* solutions;
2) Update *EP*;
3) Update the pheromone matrices according with the new solutions that were constructed by ants in group *j* and have just been added to *EP*;
4) Check the solutions on the neighborhood and updates the solutions if there is a solution that: 1) is better than its current solutions; 2) has not been used for updating other old solutions. This mechanism makes collaboration among different ant groups (sharing information).
5) The algorithm stops if a criterion is met.

Next, the main features of the algorithm applied on the MTSP are described [5].

### E. Initialization Setting

**Setting of *N* and** $\lambda^1, ..., \lambda^N$: First, the number of subproblems *N* is set, and each subproblem takes a weight vector from $\{\lambda^1, ..., \lambda^N\}$ which is controlled by parameter *H*. Each weight vector takes a value from

$$\{\frac{0}{H}, \frac{1}{H}, ..., \frac{H}{H}\} \tag{4}$$

.

And the number of weights vector is

$$N = C^{H+m-1}_{m-1} \tag{5}$$

Thus, a subproblem *i* has a weight vector $\lambda^i = (\lambda^i_1, ..., \lambda^i_m)^T$ that satisfies $\sum_{l=1}^{m} \lambda^i_l = 1$ and $\lambda^i_k \geq 0$ where *m* is the number of objectives.

**Setting of Group:** The number of groups is $K$ which is value parameter to be chosen. The number of subproblems at each group is equally distributed by clustering the weights in terms of Euclidean distance.

Each individual solution $x$ is a tour represented by permutation of the $n$ cities. The rest of parameters are domain dependent, for the MTSP:

**Pheromone matrices:** Each group $j$ has pheromone trail $\tau^j_{k,l}$ for a link between two different cities $k$ and $l$. All pheromone values are initialized to be the same value. The approach *Max-Min* [17] is used, so there are boundaries to maximum and the minimum value of $\tau$. All the $\tau^k_{k,l}$ is initialized to 1.

**Heuristic Information matrices**: Each ant $i$ has a heuristic information value $\eta^i_{k,l}$ for a link between cities $k$ and $l$. The heuristic information values are initialized as

$$\eta^i_{k,l} = \frac{1}{\sum_{j=1}^{m} \lambda^i_j c^j_{k,l}} \tag{6}$$

where $m$ is the number of objectives, $c^j_{k,l}$ is the cost between $k$ and $l$ with respect the objective $j$.

The ***EP*** is initialized empty.

## F. Solution Construction

Assume that ant $i$ is in group $j$, and its full-scale current solution $x^i = (x^i_1, ..., x^i_n)$. Ant $i$ constructs its new solution with the steps as follows:

1) First, the probability of choice a link is set. For $k, l = 1, ..., n$ set

$$\phi_{k,l} = [\tau^i_{k,l} \times In(x^i, (k,l))]^\alpha (\eta^i_{k,l})^\beta \tag{7}$$

where $\alpha$ and $\beta$ are control parameters. $\phi$ represents the attractiveness of the link between cities $k$ and $l$ to ant $i$. The indicator function $In(x^i, (k,l))$ is equal to 0 if link *(k,l)* is already in tour $x^i$ or 1 if otherwise.

2) Ant $i$ first randomly selects a city to start the tour. After, each city is chosen using the roulette wheel selection. Suppose that its current position is $k$ and it has not completed its tour. It is chosen city $l$ to visit from $C$ (cities not visited so far), according to the following probability by the roulette wheel selection:

$$\frac{\phi_{k,l}}{\sum_{s \in C} \phi_{s,l}} \tag{8}$$

3) If the ant has visited all the cities, return its tour.

## G. Pheromone Update

Let $\Pi$ be the set of all the new solutions that satisfy:

- were constructed by the ants in group $j$ in the current iteration;
- were just added to *EP*;
- contain the link between cities $k$ and $l$.

The pheromone trail value of link *(k,l)* for group $j$, is updated as follows:

$$\tau^j_{k,l} := \rho \tau^j_{k,l} + \sum_{x \in \Pi} \frac{1}{g(x|\lambda^i)} \tag{9}$$

where $\rho$ is the persistence rate of the pheromone trail. As mentioned, $\tau_{max}$ and $\tau_{min}$ are used to limit the range of the pheromone.

## III. Parallel Implementation of MOEA/D-ACO

This section briefly describes the CUDA architecture and after reports the decisions made to implement a parallel version of MOEA/D-ACO. The *solution construction* of $N$ subproblems and the *pheromone update* of $K$ groups are parallelized in this study.

CUDA allows developers to run blocks of code, known as *kernels*, directly on the GPU using a parallel programming interface and using familiar programming languages (such as C) [15]. CUDA parallel programming model has a hierarchy of thread groups called *grid, blocks (or thread blocks)* and *threads*. When a *kernel* function is invoked, it is executed $N$ times in parallel by $N$ different *CUDA threads*. A single grid is organized by multiple blocks, each of which has equal number of threads [15].

CUDA threads may access data from multiple memory spaces during their execution. All threads have access to the same *global memory*, which is implemented as an off-chip DRAM of the GPU, and has large capacity, say, 1.5-6 Gigabytes. All threads can access to the global memory, but its access latency is very long. The consecutive addresses of the global memory must be accessed in the same time. Thus, threads should perform coalesced access when they access to the global memory.

Each thread block has *shared memory* visible to all threads into a block and with the same lifetime as the thread block. The *shared memory* is an extremely fast on-chip memory with lower capacity, say, 16-48 Kbytes.

Each thread has its local (private) memory on-chip, called *register memory*. Registers are the fastest form of storage and each thread within a block has access to a set of fast local registers that are placed on-chip. Each thread can only access its own registers; moreover the number of registers is limited per block, so blocks with many threads will have fewer registers per thread.

The efficient usage of the memory types is a key for CUDA developers to accelerate applications using GPU [15] [18]. In Figure 1 is illustrated the coalesced and stride access of the *global memory*. When threads accesses to continuous locations in a row of a 2-dimensional *(horizontal access)*, the continuous locations in address space of the global memory are accessed in the same time *(coalesced access)*. From the structure of the global memory, the coalesced access maximizes the bandwidth of memory access. On the other hand, the stride access *(vertical access)* needs a lot of clock cycles [10].

## A. Preliminaries

Cecilia et al. [9] noted that the existing task-based approach of mapping one ant per thread is not suited to the GPU, because

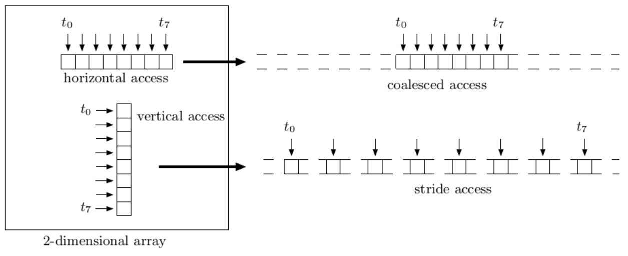

Fig. 1. Coalesced and stride access

with this approach (one ant per thread), each thread must store each ant's memory (e.g., list of visited cities, cities not visited yet) and this approach works only for small tours but quickly becomes problematic with larger tours, as there is limited shared memory and registers available for each block. The alternatives are to use fewer threads per block, which reduces GPU occupancy, or to use global memory to store each ant's memory, which dramatically reduces the kernels performance.

Based on this issue, the studies [7] [8] [9] adopted a novel data parallel approach that maps each ant to a block so as to better utilize the GPU architecture. All threads within the thread block then work in cooperation to perform a common task such as tour construction.

The performance on GPU can be drastically affected by the use of a costly math function like *powf()* see Eq.(7). Fortunately, there are analogous CUDA functions which map directly to the hardware level (like *__powf()*), although this comes at the expense of some loss of accuracy [9] [15]. Initial experiments showed that these CUDA math functions improve the execution time.

We base our parallelization strategy on the works [7] [8] [9] and adopt a data-parallel approach and mapping each ant to a thread block. The implementation consist of three CUDA parts (initialization, tour construction and pheromone update) and one CPU part (*EP* updated) for each iteration. We also consider the programming issue of the global memory (coalesced access). The details of the three GPU parts are described as follows.

### B. Initialization

The first stage of the algorithm allocates memory and the relevant data structures. Give *n* cities, the city-to-city distances are loaded in an $(n \times n)$ matrix for each objective (recall, $d_{k,l} = d_{l,k}$). The *N* ants are loaded to store each ant current tour and tour length. A kernel is invoked to calculate the distance between the cities and set the pheromone matrix initialization. The heuristic value and probability to be choice is executed during the tour construction, so we do not need to allocate a vector with size *N* (number of subproblems) to store the *N* heuristic values for each link *(k,l)*.

Also, the algorithm initializes the random seeds using the CURAND, which is a library that provides a pseudorandom number generator on the GPU by NVIDIA [19].

### C. Tour Construction

Each subproblem *i* is associated with a thread block. So the *N* ants (subproblems) construct their tour in parallel. The weight vectors are allocated on shared memory, because they are accessed few times. Each ant stores its structure (current tour, tour length, cities not visited, and its group) on the local memory (registers) of a thread. When an ant finishes its tour construction the ant is copied to the vector *N*-dimensional allocated on the global memory.

The matrix of cities $(n \times n)$ is stored on the global memory, however, the threads access only continuous location in a row of the matrix $(n \times n)$, i.e., when the current city of a tour is *k*, the threads access only the *k* row of the matrix *(k,l)*, where *l = (1,...,n)*. As we mentioned the coalesced access to the global memory is a key issue to accelerate the computation.

When all *N* ants end their tour construction, the *N-Dimensional* vector of ants is copied back to host. The *EP* is updated in host (sequential) because each new solution needs to be analyzed at time to update the *EP*. If no solutions in *EP* dominates the new one, then it is added to *EP*, and is removed from *EP* all dominated solutions. In parallel, it would be hard to guarantee correctness even using atomic functions.

### D. Pheromone Update

A new kernel is called to pheromone update. Now each block corresponds to a group and each thread corresponds to an ant according with $\Pi$ (see section II-G). The matrix $(n \times n)$ is allocated on the global memory. The first step of the update is the pheromone evaporation, which is trivial to parallelize as all pheromone trails are evaporated by a constant factor $\rho$. In the second step, the $\tau^{j}_{k,l}$ is updated according with its group, each ant in the same group deposits the same amount according with Equation 9, so we do not need to use atomic operations to guarantee correctness.

## IV. Experiments

In [5] the authors evaluated the MOEA/D-ACO for the MTSP against the BiciterionAnt [6] and achieve better results, the comparison was useful for understanding the benefit of the decomposition approach.

Initial studies show that our sequential implementation achieves the same results than the original MOEA/D-ACO. So in this section we attained the comparison between the sequential and the parallel implementation.

In the comparison we use 9 different combinations of instances from [20], using two, three and four objectives. The metric to evaluate the quality of the solutions is the *Hypervolume* [21]. To evaluate the execution time we show the speedup of the parallel implementation against the sequential counterpart.

The implementation was made using an NVIDIA GeForce GTX680 that contains 1536 CUDA cores and has a processor speed of 1058 MHz. It uses 32 threads per warp and up to 1024 threads per thread block with maximum shared memory size of 64 Kb. The CPU is an Intel i7-380QM and has 4 cores with support 8 threads with a clock speed of 3.60 GHz. Our implementation was written and compiled using the CUDA toolkit 5.0 for C with the Nsight Eclipse environment executed under Ubuntu 12.04.

The ACO parameters setting was the same used in [5]: $\alpha = 1, \beta = 2, \rho = 0.95$. The algorithms stop after 1000 generations. All the statistics are based on 10 independents runs. For the decomposition approach it was used only *Weighted sum*, because *Weighted sum* outperform the *Tcheby-cheff* approach as cited in [5]. The test instance with 500 cities and 4 objectives was infeasible due to CUDA memory constraints.

## A. Solution Quality

First, the quality of the solutions of the sequential and parallel versions is compared using hypervolume metric. In this experiment we set the number of subproblems to *N=300* and number of groups to *K=3*. The average and standard deviation of hypervolume for the 10 independent runs are summarized in Table I. The highest hypervolume value for each instance is highlighted in bold face. The Wilcoxon statistic test [22] was applied and showed that the results do not have significant difference with 0.99 level of confidence, i.e., the results show that to parallelize the MOEA/D-ACO does not decrease the solutions quality, i.e., the sequential and parallel algorithms achieve similar hypervolume.

TABLE I. AVERAGE HYPERVOLUME AND STANDARD DEVIATION OBTAINED

| Instance | Sequential | | Parallel | |
|---|---|---|---|---|
| | Average | Stand. Dev. | Average | Stand. Dev. |
| kroAB100 | 1,860E+10 | 2.439E+07 | **1.861E+10** | 1.938E+07 |
| euclidAB300 | 1.654E+11 | 4.820E+07 | **1.654E+11** | 6.680E+07 |
| euclidAB500 | **5.162E+11** | 7.569E+07 | 5.162E+11 | 6.367E+07 |
| kroABC100 | **2.618E+15** | 1.616E+12 | 2.596E+15 | 2.488E+12 |
| euclidABC300 | 5.461E+16 | 4.415E+13 | **5.446E+16** | 4.830E+13 |
| euclidABC500 | **3.148E+17** | 2.802E+14 | 3.129E+17 | 2.213E+14 |
| kroABCD100 | 2.156E+20 | 1.229E+19 | **2.551E+20** | 5.244E+17 |
| euclidABCD300 | 1.307E+22 | 1.218E+20 | **1.337E+22** | 9.224E+19 |

In Figure 2, the final approximation with the lowest Hypervolume value among the 10 runs obtained by the sequential and parallel implementation on the kroAB100 test instance is showed. The figure shows that neither algorithm established a domination relation.

## B. Execution time

Table II presents the average execution times for eight test instances and the speed up of the parallel against the sequential. Figure 3 presents the speedup attained of each test instance. The results with 300 subproblems and 3 groups show a speedup up to 8x faster than the sequential implementation.

Moreover, it is possible to observe in Figure 3, that the number of objectives affects directly the execution time. When the number of objectives increases (see test instances: *kroAB100*, *kroABC100* and *kroABCD100*), the speedup decreases, because the number of objectives affect the *EP* update phase which is executed in a sequential fashion. So, when the number of objectives increases, the number of non-dominated solutions at each generation increases, which makes the *EP* larger at each generation, consuming more time of the algorithm in the sequential part. For example, with *kroAB100*

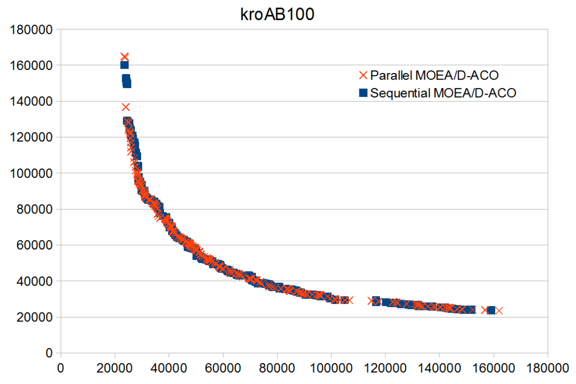

Fig. 2. Final approximation with the lowest Hypervolume value among the 10 runs obtained by the sequential and parallel MOEA/D-ACO on the kroAB100 test instance.

TABLE II. AVERAGE EXECUTION TIMES IN MILLISECONDS (MS)

| Instance | Sequential (CPU) | Parallel (GPU) | Speedup x |
|---|---|---|---|
| kroAB100 | 1.720E+05 | 2.014E+04 | 8.543 |
| euclidAB300 | 1.328E+06 | 1.691E+05 | 7.853 |
| euclidAB500 | 2.871E+06 | 4.590E+05 | 6.255 |
| kroABC100 | 4.915E+05 | 1.862E+05 | 2.640 |
| euclidABC300 | 1.617E+06 | 4.742E+05 | 3.411 |
| euclidABC500 | 3.726E+06 | 7.984E+05 | 4.667 |
| kroABCD100 | 9.099E+05 | 4.309E+05 | 2.111 |
| euclidABCD300 | 1.896E+06 | 5.821E+05 | 3.257 |

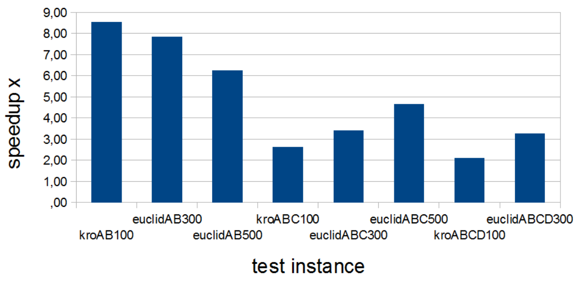

Fig. 3. Speedup of GPU parallel implementation against the sequential CPU implementation on the test instances.

test instance, one generation found 55 non-dominated solutions with final size of $|EP|$=195; with *kroABCD100* test instance, one generation found 290 non-dominated solutions and at end, more than 6000 solutions. The number of non-dominated solutions is an issue on many-objective optimization problem [23].

Another influence factor on the speedup is the number of subproblems *N*. Figure 4 shows the speedup with $N = \{150, 300, 450\}$ on the kroAB100 test instance. Each subproblem corresponds to a thread block in the parallel solution construction phase, so higher the number of subproblems higher number of thread blocks will be executing in parallel which increases the speedup. Moreover, with 450 subproblems the parallel algorithm achieves a speedup up to 11x faster than the sequential counterpart. However, because of the limited size of CUDA memory (shared memory and registers) a higher number of subproblems can be infeasible. The experiment with different values of groups was omitted because the results showed no influence on the speedup, because the number of groups is small when compared with the number of subproblems. Besides, the solution construction phase consumes more time (complexity) than the pheromone update phase.

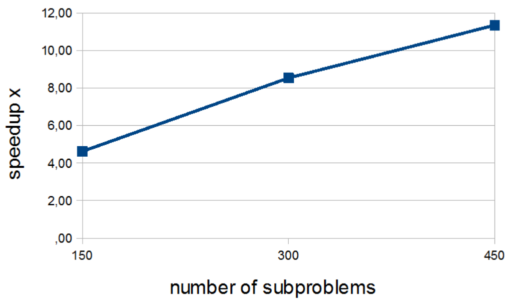

Fig. 4. Speedup of GPU parallel implementation against the sequential CPU implementation on kroAB100 test instance. The number of groups is fixed *K=3* and using three different numbers of subproblems $N = \{150, 300, 450\}$.

## V. Conclusion and Future Work

In this paper, we have proposed a new parallel implementation of the MOEA/D-ACO for the multiobjective traveling salesman problem. MOEA/D decomposes a MOP into a number of single objective subproblems and optimizes them simultaneously. The implementation is based on the data parallel approach and both construction solution and pheromone update stages are executed on GPU. Also, we have considered some programming issues of the GPU architecture such as the coalesced access of the global memory.

Our implementation is able to match the quality of solutions generated sequentially. However, the results show a speedup 11x faster than the sequential counterpart using a reasonable number of subproblems and groups. We observe some aspects that affect the execution time, concluding that: (1) A high number of objectives degrades the speedup because affects the *EP* update stage, which is implemented on host (CPU), i.e., a large |EP| decreases the performance; (3) A high number of subproblems increases the performance, because each subproblem executes its tour construction in parallel, but a high number can be an issue due to the memory limits.

Some aspects to consider for future work: (1) How to deal with the memory limits for work with largest test instances and a higher number of subproblems; (2) Improving the execution time using others parallelization approaches, e.g., a parallel version of the roulette wheel selection (not implemented due the architecture of the GPU available); (3) Parallelizing the *EP* update stage which is the main reason that we do not achieve highest speedup.

## References

[1] M. Dorigo and G. D. Caro, “The ant colony optimization meta-heuristic,” in *New Ideas in Optimization*. McGraw-Hill, 1999, pp. 11–32.

[2] M. Dorigo and T. Stutzle, *Ant colony optimization.* MIT Press, 2004.

[3] M. Lopez-Ibanez and T. Stutzlee, “The automatic design of multi-objective ant colony optimization algorithms,” *IEEE Trans. on Evol. Comp.*, vol. 16, no. 6, pp. 861–875, 2012.

[4] Q. Zhang and H. Li, “Moea/d: A multiobjective evolutionary algorithm based on decomposition.” *IEEE Trans. Evolutionary Computation*, vol. 11, no. 6, pp. 712–731, 2007.

[5] L. Ke, Q. Zhang, and R. Battiti, “Moea/d-aco: A multiobjective evolutionary algorithm using decomposition and ant colony.” *IEEE Trans. Cybern.*, vol. 43, no. 6, pp. 1845–1859, 2013.

[6] S. Iredi, D. Merkle, and M. Middendorf, “Bi-criterion optimization with multi colony ant algorithms.” in *EMO*, ser. Lec. Notes in Comp. Science, E. Zitzler, K. Deb, L. Thiele, C. A. C. Coello, and D. Corne, Eds., vol. 1993. Springer, 2001, pp. 359–372.

[7] L. Dawson and I. A. Stewart, “Improving ant colony optimization performance on the gpu using cuda.” in *IEEE Congress on Evol. Comp.*, 2013, pp. 1901–1908.

[8] A. Delevacq, P. Delisle, M. Gravel, and M. Krajecki, “Parallel ant colony optimization on graphics processing units.” *J. Parallel Distrib. Comput.*, vol. 73, no. 1, pp. 52–61, 2013.

[9] J. M. Cecilia, J. M. Garcia, and Nisbet, “Enhancing data parallelism for ant colony optimization on gpus.” *J. Parallel Distrib. Comput.*, vol. 73, no. 1, pp. 42–51, 2013.

[10] A. Uchida, Y. Ito, and K. Nakano, “An efficient gpu implementation of ant colony optimization for the traveling salesman problem.” in *Third International Conference on Networking and Computing*, 2012, pp. 94–102.

[11] A. M. Mora, P. Garcia-Sanchez, and P. A. Castillo, “Pareto-based multi-colony multi-objective ant colony optimization algorithms: an island model proposal.” in *Soft Computing*, ser. Lec. Notes in Comp. Science, vol. 17. Springer, 2013, pp. 1175–1207.

[12] A. M. Mora, J. J. M. Guervos, P. A. Castillo, M. G. Arenas, P. Garcia-Sanchez, J. L. J. Laredo, and G. Romero, “A study of parallel approaches in moacos for solving the bicriteria tsp.” in *IWANN (2)*, ser. Lecture Notes in Computer Science, vol. 6692. Springer, 2011, pp. 316–324.

[13] A. J. Nebro and J. J. Durillo, “A study of the parallelization of the multi-objective metaheuristic moea/d.” in *LION*, ser. Lecture Notes in Computer Science, C. Blum and R. Battiti, Eds., vol. 6073. Springer, 2010, pp. 303–317.

[14] M. L. Wong, “Parallel multi-objective evolutionary algorithms on graphics processing units,” *GECCO’09*, pp. 2515–2522, 2009.

[15] NVIDIA. (2014) Cuda c programing guide v5.5.

[16] C. C. Coello, G. Lamont, and D. van Veldhuizen, *Evolutionary Algorithms for Solving Multi-Objective Problems*, 2nd ed., ser. Genetic and Evolutionary Computation. Berlin, Heidelberg: Springer, 2007.

[17] T. Stutzle and H. Hoos, “Max-min antsystem.” *Fut. Gen. Comp. Syst.*, vol. 16, no. 8, 2000.

[18] NVIDIA. (2014) Cuda c best pratices guide v5.5.

[19] ——. (2014) Cuda toolkit guide v4.1 curand.

[20] [Online]. Available: http://eden.dei.uc.pt/ paquete/tsp/.

[21] J. Yan, C. Li, Z. Wang, L. Deng, and S. Demin, “Diversity metrics in multi-objective optimization: Review and perspective,” in *IEEE International Conference on Integration Technology. ICIT’07*, 2007, pp. 553–557.

[22] J. Derrac and S. Garcia, “A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms.” *Swarm and Evol. Comp.*, vol. 1, no. 1, pp. 3–18, 2011.

[23] H. Ishibuchi, N. Akedo, and Y. Nojima, “A study on the specification of a scalarizing function in moea/d for many-objective knapsack problems,” in *Learning and Intelligent Optimization*. Springer Link, 2013, pp. 231–246.