International Journal of Parallel Programming
https://doi.org/10.1007/s10766-022-00736-3

Parallelization of Swarm Intelligence Algorithms: Literature Review

Breno Augusto de Melo Menezes[1] · **Herbert Kuchen**[1] · **Fernando Buarque de Lima Neto**[2]

Received: 13 December 2021 / Accepted: 26 May 2022

Abstract

Swarm Intelligence (SI) algorithms are frequently applied to tackle complex optimization problems. SI is especially used when good solutions are requested for NP hard problems within a reasonable response time. And when such problems possess a very high dimensionality, a dynamic nature, or present intrinsic complex intertwined independent variables, computational costs for SI algorithms may still be too high. Therefore, new approaches and hardware support are needed to speed up processing. Nowadays, with the popularization of GPU and multi-core processing, parallel versions of SI algorithms can provide the required performance on those though problems. This paper aims to describe the state of the art of such approaches, to summarize the key points addressed, and also to identify the research gaps that could be addressed better. The scope of this review considers recent papers mainly focusing on parallel implementations of the most frequently used SI algorithms. The use of nested parallelism is of particular interest, since one level of parallelism is often not sufficient to exploit the computational power of contemporary parallel hardware. The sources were main scientific databases and filtered accordingly to the set requirements of this literature review.

Keywords Metaheuristics · Swarm Intelligence algorithms · Parallel computing · Cluster computingHigh-performance computing

✉ Breno Augusto de Melo Menezes
breno.menezes@wiwi.uni-muenster.de

Herbert Kuchen
kuchen@uni-muenster.de

Fernando Buarque de Lima Neto
fbln@ecomp.poli.br

[1] Department of Information Systems, University of Muenster, Leonardo-Campus 3, Muenster 48149, NRW, Germany

[2] ECOMP, University of Pernambuco, Rua Benfica, 455, Recife 50720-001, Pernambuco, Brazil

Published online: 10 August 2022

1 Introduction

Metaheuristics are often a primary choice used in hard optimization problems [1]. This kind of algorithm is used for solving complex problems when there is a need for solutions that are good enough and the execution time is not large. In such scenarios, the use of exact methods would need too much time to execute as there are many reasons for that.

Swarm Intelligence (SI) algorithms are important representatives of the population based metaheuristics. SI algorithms are inspired by the behavior of collections of individuals present in nature, such as a flock of birds, an ant colony, or even a fish school. The key principle in SI is the simulation of many particles that have a simple behavior but, together, these particles evoke the emergence of solutions through ad hoc topologies and communication strategies. This principle can be observed in the most diverse SI algorithms, such as the Particle Swarm Optimization (PSO)[2], Artificial Bee Colony (ABC)[3] and the Ant Colony Optimization (ACO)[4]. Despite the fact that each of these algorithms aims to solve different types of problems, they share many characteristics, e.g., movement operators, communication patterns and fitness evaluations.

The collective behavior is one of the main reasons why SI algorithms are able to achieve good results, but not without costs. For problems that have a high number of dimensions and fitness functions that are computationally expensive, the executions might also take a long time to solve even when SI algorithms are used. Parallel implementations of these algorithms become necessary in such cases. Nowadays, many parallelization possibilities arise with the easy access to high performance hardware such as multi-core CPUs, graphics processing units (GPU) and cluster architectures. Different approaches and strategies have been introduced, all of them aiming to boost SI algorithms and solve very complex problems still in a feasible amount of time.

Parallelization frameworks such as OpenMP [5], MPI [6] and CUDA [7] are often used by programmers to support the implementation of parallel SI algorithms for the most diverse hardware. But even so, the development of such applications can be complex and error prone. Besides the natural complexity of the algorithm, the programmers must be also aware of problems inherent to parallel programs (i.e. race conditions, deadlocks, data transfers and hardware limitations). The generation of high performance parallel code requires some expertise from programmers, even when using such frameworks.

In this complex scenario, high-level parallelization frameworks such as Muesli[8] and MALLBA[9] were created to facilitate the development of parallel applications. By using such tools, the programmer benefits from pre-defined operations (namely skeletons) that can be used to build up an implementation. These skeletons represent commonly used parallelization patterns such as map and reduce. Parallelization aspects are hidden from the programmer, who has to take care only of the algorithmic aspects.

The parallelization of SI algorithms is not an easy task. Although having many independent individuals is definitely a characteristic that makes SI algorithms suitable for parallelization, the communication steps require a great deal of attention since all particles have to stop their individual movements and start the communication coherently. Especially for parallel implementations, where this phase can cause high costs, for example, when communication happens between several devices. So, unnecessary communication and transfer of data can create an overhead that for sure penalizes the execution of the algorithm and consumes more time.

The parallelization of such algorithms is nowadays supported by the high computational power available in modern hardware, such as multi-core CPUs or GPUs. The high processing power enables the deployment of many parallel threads (or even thousands in the case of GPUs), while it typically does not make sense to employ more than a few hundreds of particles in an SI algorithm[10]. Thus, a simple approach mapping each particle to a thread will not be able to exploit the full power of the hardware. Rather, a *nested parallelization* is required in order to achieve this.

Considering all this, the goal of this paper is to inspect the applicable literature and summarize the trending approaches regarding parallelization of the selected prominent SI algorithms, e.g., PSO, ABC and ACO. Papers aiming at both, CPU and GPU implementations, will be considered in this work. By extracting information from the relevant papers from the last years, we want to identify the parallelization tools used and the parallelization approaches taken. The objective is that, after this study, we are able to guide future research to explore the gaps identified and further improve parallel approaches of swarm intelligence algorithms.

We assume the readers have some familiarity with hardware for parallel computing such as clusters [11] of multi-core processors [12] equipped with possibly several accelerators such as GPUs [13]. We also assume some familiarity with frameworks for parallel programming such as MPI [6] for clusters, OpenMP [5] for multi-cores processors as well as CUDA [7] and OpenCL [14] for GPUs. Message passing frameworks such as MPI allow the different computing nodes of a cluster to asynchronously run in parallel independent tasks which communicate via messages. Multi-core processors can run several threads in parallel, which communicate via a shared memory. Finally, GPUs allow to run typically thousands of threads in an SIMD (single instruction multiple data) style, where the same instruction is executed on different data stored in a separate GPU memory. In order to use a GPU, data have to be exchanged between main (CPU) memory and GPU memory.

This document is structured as follows. Section 3 introduces swarm intelligent algorithms. Section 4 presents the protocol used in the literature review. Discussion about the main findings of this review are presented in Sect. 5. Finally, the conclusion of this work, together with some outlook, is presented in Sect. 6.

2 Related Work

Y. Tan's work [10] is a survey over parallel implementations of Swarm Intelligence algorithms but only GPU-based ones. It also included other algorithms such as Genetic Algorithm (GA) and Differential Evolution (DE). While our focus is one the parallelization approach, Tan diverged some attention also to evaluation criteria, such as performance metrics.

Lalwani et al. presented a survey about parallel swarm optimization algorithms[15]. In the one hand, this survey goes over CPU- and GPU-based implementations, using well known frameworks such as OpenMP, MPI and R. On the other hand, it is limited to the PSO algorithm.

Kroemer et al. also presented a survey on parallel PSO implementations[16]. Their survey is limited to GPU implementations, while this work intends to identify aspects regarding the parallelization in different hardware.

3 Swarm Intelligence Algorithms

Before we come to the parallelization aspects, let us summarize the considered SI approaches, namely, PSO, ABC and ACO. Considering the size of the SI family, the selection of these approaches was based on specific characteristics that influence parallelization.

The first difference regarding these algorithms is the kind of problems they intend to solve. While PSO and ABC were originally designed to solve continuous optimization problems, ACO was designed to solve combinatorial optimization problems, such as the Traveling Salesmen (TSP)[17]. Figure 1 illustrates both types of problems. The main difference is that while a PSO particle may assume any position inside the continuous range established by the model, in this example the Ackley function, a TSP solution built by an ant in ACO has to include the exact nodes available in the graph to build a feasible solution to the problem.

Besides that, there are other aspects regarding the algorithms that can impact the process of parallelization. For example, while the particles' movements are guided by two opposing terms (namely cognitive and social) in PSO, bees have different roles that determine which function they should perform in ABC. Moreover, ACO differs from other algorithms the way it memorizes success. Instead of storing it in the ants themselves, pheromones are deposited in the environment which represent a body of shared memory from previous successes of the colony.

With the understanding of these three approaches and their differences, we cover aspects and parallelization scenarios that, conceptually, are also present in many other SI approaches. This allows us to widen our discussion that could also be extended for other SI approaches.

3.1 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is one of the most famous population-based meta-heuristics. It was first introduced by Kennedy and Eberhart back in 1995 [2]. The inspiration comes from the behavior of a flock of birds. PSO simulates all birds of a flock where each particle, representing a bird, is allowed to move inside the search space aiming to find the optimal location. For each particle, its coordinates in the search space determine a candidate solution for the considered optimization problem. PSO incorporates the notion of *speed* (as a metaphor of success) and each particle is moving in the search space at its own

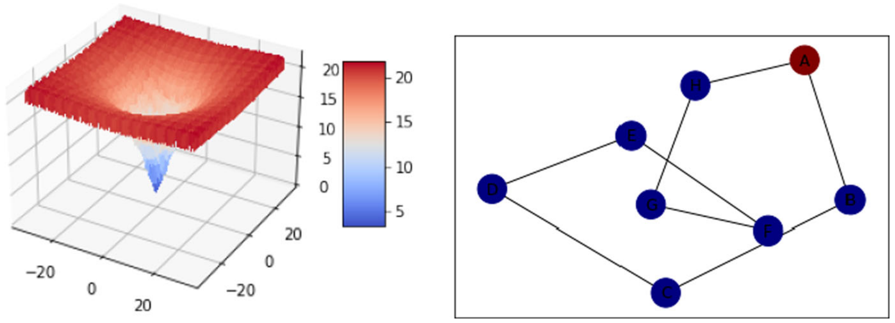

Fig. 1 continuous vs. discrete problems: Ackley function and a TSP instance

pace and direction. Updates in the particles' movements are performed considering two factors (see Eq. 1): *own success* (cognitive term) and *global success* (social term), expressed by the second and third summand in the equation, respectively.

$$v_{t+1} = \omega v_t + c_1 r_1 * (b_t - x_t) + c_2 r_2 * (g_t - x_t) \tag{1}$$

where ω is the inertia factor, c_1 and c_2 are weights of the cognitive and social term, respectively. Moreover, b_t, g_t and x_t are the local best position (found by the considered bird up to time t), the global best position (found by the whole flock up to time t) and the position of the considered bird at time t, respectively.. A wrong setup of ω, c_1 and c_2 might lead to an early convergence of the algorithm or make the particles get stuck in local minima.

After updating the speed, each particle will update its position according to Eq. 2, where v_{t+1} is the speed at time $t+1$, while x_t and x_{t+1} are the positions of the particle at time t and $t+1$, respectively.

$$x_{t+1} = x_t + v_{t+1} \tag{2}$$

After performing the speed and position updates, the fitness of each particle at its current position is calculated. The current local best fitness is then compared to the new value and possibly updated. Moreover, the current global best position will be updated, if a particle has obtained a better fitness than the best known one. After these updates, the mentioned steps are repeated, until a termination criterion is fulfilled. A fixed number of iterations is often used as termination criterion. In some cases, the stagnation of the fitness is used instead, which means that the iterations will stop when the swarm stops improving after a given number of unsuccessful attempts.

Determining the right swarm size and number of iterations is crucial for the success of PSO. As particles are initialized in random positions, having more of them means that diversity is increased and a larger area of the search space is covered. Also, the social factor, represented by the exchange of coordinates of the currently best solution, is important in PSO, since it directs the search into a promising direction. The more iterations an algorithm performs, the more steps are followed in this promising direction and the higher are the chances of finding an optimal (or at least very good) position. PSO can be applied to problems of different magnitudes and, just by adjusting these two parameters, it might be able to find a good solution.

3.2 Artificial Bee Colony

The artificial bee colony (ABC) algorithm is inspired by the behavior of honey bee colonies and their ability of finding food sources[3]. The algorithm is characterized by a division of labor and the tasks are divided among bees of three different types: employee, onlooker and scout bees. In ABC, the number of food sources must be determined and for each food source, there will be an employee bee assigned to it. Each employed bee will determine a food source in the surroundings of an old food source in their memory. These new spots are shared among the hive and onlooker bees select one of these food sources and search for a new better place in the surroundings of the selected source.

A food source that has not improved for a given number of iterations will be abandoned. The employed bee responsible for this food source becomes a scout bee and starts exploring the search space for a new food source. Scout bees move freely through the search space without any guidance.

Initially, each employee bee is randomly allocated in the search space and then evaluates the quality of the current food source/position. An employee bee is allowed to look for a better food source in the surroundings of the current established position. Once per iteration, they choose a random direction and evaluate the quality of the reached point. If the quality of the solution improves, it becomes the new food source, replacing the old one.

Onlooker bees are associated with employee bees. (Eq. 3) shows the probability P_i that an onlooker bee is associated with the food source, which is currently being explored by employee bee i. Food sources of higher quality are more probable to be chosen by onlooker bees. Each onlooker bee is allowed to search in the surrounding of its chosen food source. If it finds a better food source, it also replaces the respective food source it was assigned to.

$$P_i = \frac{F(\theta_i)}{\sum_{k=1}^{S} F(\theta_k)} \tag{3}$$

where $F(\theta_i)$ is the fitness of the food source i and S is the total number of food sources.

If a food source does not improve during a certain period, scout bees are activated and a new food source is assigned automatically. This behavior sets ABC apart from PSO. While in ABC a stagnation causes a switch from exploitation to exploration, PSO gradually changes its behavior from exploration to exploitation by continuously reducing the step size during the execution. Normally, PSO instances have bigger step sizes at the begging of the execution in order to explore the search space and, during the execution, this step size gets smaller allowing exploitation.

3.3 Ant Colony Optimization

The ant colony optimization (ACO) algorithm mimics the behavior of ants in their search for food. While searching, ants leave a trail of pheromone that will attract other ants. In this process, the shortest path found is the one with most pheromone, since ants will be passing it more often and pheromone on other trails is likely to evaporate. Proposed by Dorigo [4], the Ant System algorithm (AS), an implementation of ACO, was first employed to solve the shortest path problem.

When applied to problems such as the traveling salesperson problem (TSP), ACO repeatedly executes two steps. The first one is the tour construction, where each ant constructs a full round trip exploring the pheromone on the edges between the nodes of the considered graph. Starting at the initial node, each ant always determines its next step to a neighbor node in a probabilistic way. Edges with a higher amount of pheromone have higher chances of being chosen. Therefore, at each step, the probability of following a certain edge has to be re-calculated for all possible edges (Eq. 4).

$$P_{ij}^{k} = \frac{\tau_{ij}\eta_{ij}}{\sum_{l \in N} \tau_{il}\eta_{il}} \tag{4}$$

where, P_{ij}^{k} is the probability of ant k moving from node i to node j, τ_{ij} represents the amount of pheromone between i and j and η_{ij} is equal to $1/d_{ij}$, being d_{ij} the distance between i and j. τ_{il} and η_{il} are defined analogously for the edge between nodes i and l. N is the set of unreached neighbors of node i.

After each ant finishes constructing a complete round trip, it is time to update the pheromone. The edges on shorter trips receive more pheromone compared to those of longer ones. Also, the evaporation of pheromone must be considered, and edges that were not used will suffer a reduction of pheromone. After this phase, the iteration is concluded and, if the stopping criteria is not reached, the ants will start creating new paths again.

In contrast to PSO, the success in ACO is not represented by the particles' positions, but by pheromone levels on the edges. Thus, ACO is quite suitable for a parallelization, since the path construction is an independent task performed by each ant and the pheromone in the environment resembles shared memory. ACO also includes demanding tasks such as the calculation of probabilities for the next step and fitness calculations that can be parallelized.

4 Literature Review Protocol

This literature review includes recent approaches and concepts for the parallelization of prominent SI metaheuristics. In this section we present the protocol containing all steps and activities used to (1) collect the adequate papers that tackle the parallelization of swarm intelligence algorithms and (2) extract the relevant information.

With the main objective and research question defined, the next step was to define the keywords that should be used to build the search strings for each scientific base and also to assess the relevance of all gathered papers later on. For the parallelization theme, the keywords chosen are related to the main frameworks used nowadays for parallelization, namely, CUDA, OpenMP and MPI. The same rationale was used for the SI theme, where we focused on the prominent SI algorithms (i.e. PSO, ABC, FSS [18] and ACO). In addition to that, relevant terms such as *parallel* and *high-performance* were also used.

Besides keywords, selection and exclusion criteria were also defined in order to exclude papers that are not relevant for this work, such as papers that don't focus on the parallelization or papers that don't present concrete results.

The main scientific databases selected for this work were Scopus, Web of Science and Science Direct. Volume and relevance for Computer Science were the main criteria used in this selection. The keywords mentioned above were used to construct equivalent search strings for each scientific database.

The corresponding queries resulted in a list of initially more than 300 papers. After removing duplicates, we could start reading the papers, discard the ones that did not fulfill the inclusion criteria, and carefully extract the relevant information from the remaining ones. For this work, we've decided to include only full papers written in English. Papers to be included must also have some focus on the parallel implementation and concrete experimental results. Papers that did not meet these criteria were excluded from this work.

The remaining 87 papers were then organized in groups according to the SI algorithm used such that they could be compared better. The following subsections discuss these papers according to the group they belong to. The subsections give a first, coarse-grained overview of the approaches. A detailed discussion will follow in the next section.

4.1 Continuous Optimization Findings

4.1.1 PSO Findings

Among the papers that aimed to solve continuous optimization problems, PSO was the SI algorithm that appeared most frequently. From the 87 relevant papers found, 57 used classical PSO or an enhanced version of it. An overview of these works, including some information regarding the parallelization framework used and hardware environment, is displayed in Tables 1 and 2. For both tables, columns under *Hardware* label the maximum number of GPUs and CPUs used by each work. Furthermore, the number of nodes is also displayed so that works that aimed a distributed memory environment can be identified. The parallelization tools used by each work are listed under *Frameworks*. High level frameworks are explicitly named under the *HL* column, while approaches that used other approaches such as MATLAB were marked under the *Other* column. For clarity purposes, PSO works were divided among two tables. Table 1 lists works that did not use GPUs, while Table 2 lists works that use GPUs.

Tables 3 and 4 display details about the parallelization strategies used by the authors. For example, Garcia-Nieto et al. [19] implemented parallel swarms, with asynchronous communication between these swarms. They also used a topology to link the swarms such that they could exchange particles. Also, in the last column, the top speedup achieved from all experiments is displayed. This value can give an insight of what can be achieved when using a certain approach and tackling a specific problem. In contrast to this, the comparison between speedups from different works might not be realistic.

Table 1 PSO Hardware and Framework - CPUs

	Hardware			Frameworks				
Approach/Paper	GPUs	CPUs	Nodes	CUDA	OpenMP	MPI	HL	Other
PMSO [19]	–	8	8				MALLBA	
PSO [65]	–	6	6			x		
PSO [81]	–	1	1				MATLAB	
PPSO [59]	–	1	1					x
binaryPSO [82]	–	15	15					x
GA-PSO [60]	–	25	25				Hadoop	
PSO [61]	–	2	1				MATLAB	
AIU-PSO [64]	–	32	32		x	x		
PTVPSO-SVM [83]	–	11	11					x
PSOwVV [68]	–	8	4		x	x		
PSO [51]	–	1	1					x
pMOPSO [67]	–	56	7		x			
PSO [84]	–	1	1					x
PSO [85]	–	8	8				Hadoop	
MMPSO [49]	–	1	1					x
VEPSO [57]	–	256	64			x		
PSO [33]	–	10	10					x
H-MPSO-SA [39]	–	1	1					x
MOPSO [86]	–	1	1					x
CAPPSO[56]	–	>1	>1			x		
DEEPSO[52]	–	1	1		x			
MOPSO[41]	–	1	1		x	x		
PSO[54]	–	1	1				Fork/Join	
MO-TLPSO[87]	–	1	9			x		

The papers listed in these tables show that there are many possibilities to explore when talking about the parallelization of PSO. From a simple single swarm parallel implementation to a more complex multi-swarm multi-objective one. Regarding the hardware, approaches aiming at CPUs only are just as present as the ones using GPUs, bringing points to be discussed later on in this work.

4.1.2 Other algorithms

9 papers out of the 87 made use other algorithms for continuous optimization that were not PSO. The Artificial Bee Colony algorithm was found in 8 works ([20–25]) while the Fish School Search algorithm was used in [26]. Tables 5 and 6 display the information about these papers.

4.2 Discrete Optimization Findings

Regarding discrete optimization, ACO is the most known and used SI algorithm. The vanilla version of ACO and its variants (i.e. Ant System) were among the algorithms used in the works gathered and therefore they are listed together here. Tables 7 and 8 display the properties of the papers gathered in this review regarding ACO. In total, 21 papers describe the parallelization of the ACO algorithm.

Concerning Table 8, it is important to mention that the *probabilities* column refers to works that parallelize also the calculation of probabilities of taking a certain edge in the next movement, assigning one thread for each calculation, for example.

5 Discussion

In this section, we will discuss the main aspects extracted from the papers we have listed previously. Algorithms mainly used for continuous optimization problems will be addressed in one subsection and a second subsection will address discrete optimization problems. In order to structure the discussion, the different topics will be analyzed separately (as also done by Tan and Ying [10]).

5.1 Continuous Optimization

Most papers included in this literature review deal with continuous optimization problems. In these papers, algorithms such as Particle Swarm Optimization, Artificial Bee Colony and Fish School Search are used to solve a wide range of problems, from traditional benchmark functions to real world optimization problems. As these algorithms are aiming for the same type of problems, they share, for example, parallelization strategies and other aspects that will be explained in the following.

5.1.1 Parallelization of Basic Operations

The execution of SI algorithms in general includes movement operators and fitness function calculations. Normally, movement operators are composed of simple operations, which do not generate substantial computational costs and are hence not worthwhile to be parallelized. The *fitness functions*, in contrast, can get quite complex and their evaluations often require a big part of the computational time. For example, in [23], the authors state that, in a sequential implementation of the ABC algorithm, 85% of the execution time consists of fitness evaluations. Therefore, a speedup could be achieved just by parallelizing this step of the execution. Singh et al. use this strategy and state that it is possible to achieve a 10x speedup comparing their implementation (where only the fitness evaluation is performed in a CUDA kernel) to a fully sequential implementation [20, 27]. Unfortunately, the authors do not comment on the overhead generated by repeatedly transferring data between CPU and the GPU. Anyway, the achieved speedup is impressive.

5.1.2 Particle-level Parallelization

The straight-forward approach to implement SI metaheuristics in parallel is to employ a parallelization on the particle level, since the particles are rather independent in most of the algorithms. The authors of the large majority of the considered papers used this approach as displayed in Tables 3, 4 and 6. This strategy is easy to implement and it is capable of reducing considerably the execution time of the algorithm [25, 28–40].

When aiming for multicore CPUs, OpenMP provides tools for easily implementing SI algorithms that are parallelized on the *particle level*. These implementations also profit from features present in modern chips such as hyper-threading and other optimizations that allow the execution of several threads simultaneously by a single core. As SI algorithms are repeatedly iterating over their particles using loops, directives such as the *parallel-for* are can be applied, dividing the workload of those loops among all the cores available. Consequently, speedups are proportional to and limited by the number of available cores (up to hyper-threading).

This approach can be easily implemented and contribute to speedups for most of the applications. Even though, it must be taken into account that it might not have the best performance for more complex scenarios, for example when a large swarm size is needed or when the computations performed by each particle are too expensive. As each core processes its workload sequentially and there are no other resources available, performance improvements are limited to the number of CPU cores.

In order to exploit distributed memory architectures, this approach can be extended using MPI. Li et al. [41] presented in their work an MPI-OpenMP-based multi-objective PSO. Although the work is distributed among several nodes, all particles belong to the same swarm. Internal divisions, named *subspecies* are used to tackle the subproblems (as reducing communication).

The same parallelization can be applied when aiming a GPUs. With a high number of cores available, GPUs have an interesting architecture for running SI algorithms where the same instruction is applied in SIMD style to multiple data. A very simple implementation using this approach was presented by Dong et al. In their implementation, each particle was represented by one CUDA thread and, for their problem, just one CUDA block was enough to host the whole swarm [34]. A very good performance was achieved this way by using the memory available locally and the shared memory that belongs to the CUDA block.

Memory handling is normally a concern when developing for GPUs and using frameworks such as CUDA, especially when tackling bigger problems. Furthermore, there are other issues that the programmer has to consider in order to achieve high performance. One of them is regarding CUDA's architecture features such as the computation capability of the GPU targeted. This capability determines how many threads can be spawned simultaneously and what are the resource limitations regarding threads and blocks in use. Exceeding these limitations generates overhead.

Another issue when using CUDA to implement such algorithms is the memory hierarchy. In order to optimize the execution of the algorithm, it is important to make good use of the local memory available for each of the CUDA threads. The problem is that, although being really fast to access, the local memory available for CUDA threads is very small. If the computations of the particles are simple enough, they can benefit from it. Otherwise other approaches have to be used, such as the use of shared memory and global memory.

Shared memory is used specifically if the data stored is to be shared among the threads that belong to the same CUDA block. It is not as fast as the local memory, but it offers more space. In CUDA, the use of global memory is recommended to store data structures that are large and/or need to be kept during the entire execution of the program (e.g. the position array of all particles in PSO). It is important to note that going up in the memory hierarchy (to faster memory) means that more time will be needed to transfer data. Therefore some overhead must also be considered.

If there are as many particles as cores on the GPU, a particle-level parallelization can be very efficient. If there are less particles, some cores will be running idle during the execution. Unfortunately on modern GPUs, the parallelization on particle level of algorithms like PSO and FSS typically only allows to use a small fraction of all cores. Thus, a parallelization only on particle-level will be insufficient to fully exploit the hardware.

5.1.3 Fine-Grained Parallelization

With the possibility of spawning many more threads on modern GPUs, some authors used a parallelization on a lower level of the algorithm. Instead of (only) running particles in parallel, the internal operations of each particle run in parallel. In particular in highly dimensional optimization problems, it is worthwhile to consider *all dimensions of a particle in parallel*. This approach can be seen in [42–46].

A finer degree of parallelization as mentioned provides a better occupancy of the GPU cores as the number of threads spawned can be higher. Furthermore, this approach benefits from executing much shorter routines (so-called kernels) when compared to a coarse grained implementation which runs loops at the particle level. More threads and shorter routines are a good combination to deal with branching problems in CUDA. Note that branching of a computation on a GPU leads to a sequentialization of the branches due to the SIMD style of the computation.

The implementation of a fine grained parallel algorithm is not as simple as a parallelization on the particle level. Some knowledge about the internal calculations of a particle is necessary, so that the workload can be divided in smaller shares. In the ideal scenario, these smaller shares can be processed independently, exactly like it happens in the PSO's speed update phase (Eq. 1). During this step, the speed for each dimension is calculated independently. When this is not the case and threads must share information during a certain step, synchronization and data transfers become necessary. These operations are responsible for generating overhead and therefore must be used with care, aiming to minimize the time spent doing them. Furthermore, it is important to consider that not all operations can be parallelized in a finer degree. For example, some fitness functions are composed of just one simple term, which would not make sense to be parallelized. In such cases, the parallelization still occurs on the particle level.

Another problem with this approach is regarding the number of threads spawned and the limitations imposed by the hardware. In CUDA, the number of threads per block is limited, meaning that using this approach can be problematic if the problem has too many dimensions.

Even though, the use of the fine-grained parallelization approach generates better performance when using GPU environments compared to CPU environments. The high number of cores present in a GPU and the way it is organized make it perfect for the execution of simple instructions over a great quantity of data, which is exactly the case. On the other hand, the same approach aiming at a CPU environment would not profit so much because it is restrained by the limited number of cores present in such hardware. Even though modern CPUs have cores with more advanced features, GPUs compensate in numbers, being perfect for this kind of job.

5.1.4 Parallel and Cooperative Swarms

Until this point, all parallelization strategies mentioned lead to smaller execution times of the algorithms when compared to a sequential implementation of the considered metaheuristic. Moreover in both environments, GPUs and CPUs, the performance of the metaheuristic is equal or similar in terms of the quality of the computed solution of the considered optimization problem. However, the amount of computational power available on modern high performance computers allows to use even more complex and advanced parallelization approaches, aiming a satisfactory execution time but also a better solution at the end.

One way to profit from such powerful hardware is to run *multiple swarms* simultaneously. This kind of implementation can be seen in [24, 47, 47–56]. SI algorithms have a non-deterministic behavior, therefore each run typically produces different results. Each instance of the algorithm starts with its particles placed in different positions of the search space and this initial placement of particles is an important factor that has direct influence on the success of the algorithm. Running several instances of the same algorithm in parallel increases diversity and, at the same time, the chances of achieving a better fitness.

Distributed memory environments make it possible to run multiple instances of SI algorithms as separate swarms in parallel without increasing the run time, which would not be the case in a sequential approach. In this scenario, communication and cooperation strategies between the swarms have been introduced in order to improve the quality of the obtained solution. Such approaches were presented in [51, 57, 58] [19, 44, 48, 59–62]. These works also show that cooperative parallel swarms achieve better fitness results than parallel independent swarms due to the exchange of information.

There are many ways to exchange of information. The straightforward way is to share with all swarms the best global position found so far. This however reduces the diversity. A more complex approach is to introduce a topology to connect the swarms and determine how they share information with neighbor swarms in this topology. This helps to avoid that the swarms are guided to the same point and that a premature convergence of all swarms happens. It also enhances the diversity among the swarms, since the information shared is different among them.

Some authors propose to transmit just the location of the best position found so far by the sending swarm. Other approaches perform an exchange of particles among swarms. Both approaches increase the capability of finding better positions in the search space. But there are no comparisons that could help determine which approach leads to better solutions.

Another approach is to run different SI algorithms in parallel. Such an approach has been investigated in [63]. The algorithms need not belong to the same family. As each optimization algorithm has different operators and communication patters, such an approach can benefit from the strengths of each algorithm. Using this concept, more complex structures can be built. For example, in [63], the authors propose the idea of having multiple archipelagos. Each archipelago runs one algorithm; in this case PSO, a Genetic Algorithm (GA) and Simulated Annealing (SA). On an archipelago, each island runs an instance of the algorithm with different parameters. Islands communicate with each other using a given topology in an asynchronous way, and so do archipelagos.

It is important to choose suitable communication patterns when implementing such algorithms. In a distributed environment with several nodes, for example, data transfers have a huge impact on the execution time. An asynchronous communication strategy was proposed by Bourennani et al. in order to enable overlapping communications and computations [56]. In their proposed version of PSO, each particle uses its local best, the subswarm best and the best solution found so far among all subswarms to update its position. A master node is responsible for processing the global information. It works asynchronously, in a way that it does not impede the subswarms of keeping their processes running.

Another important aspect is regarding the efficiency of the algorithm and the occupancy of the processors. The programmer must keep in mind that different algorithms will require different computation times for one iteration. Therefore, the time spent for a synchronization is actually determined by the slowest algorithm. For this reason and also in order to reduce the communication overhead in general, it is a wise to exchange information not in every step, but every n steps with $n > 1$ [63]

Also, several authors have observed that asynchronous communication is recommendable. With asynchronous communication, it does not matter if the algorithms are in different iterations. The exchange of information between them is always beneficial [64].

5.1.5 Distributed Memory Environments

Cooperative Multi-swarm approaches are able to produce encouraging results. However, they require a considerable amount of hardware. The CUDA framework, for example, includes features that allow the development of applications that will run using more than one GPU. Or, even further, it is also possible to run an algorithm on several nodes of a cluster, each node having multiple GPUs [36].

Aiming at a CPU environment, two other low-level frameworks are often used (namely OpenMP and MPI). OpenMP consists of a set of compiler instructions that control how parallelism is added to the program running on a multi-core CPU with shared memory. MPI, on the other hand, is a message passing API used for parallel and distributed computing. It is typically used on clusters of computing nodes, which mostly consist of multi-core CPUs (and possibly GPUs). These two frameworks can be combined. Examples of parallel SI algorithms using these frameworks can be found in [26, 49, 56, 64–68].

The master-worker model is often used in combination with these frameworks. When aiming at a single node shared memory environment, the programmer can create n threads using OpenMP from a master thread. Worker threads are responsible for updating the values of the particles, while the master thread is responsible for determining the globally best solution so far and providing it to all worker threads.

Additionally, the programmer can add MPI instructions and adapt the implementation to a distributed memory environment, as it can be seen in [68]. Here, each node runs an MPI process such that the workload can be divided among the different nodes. The first approach is to divide the particles equally among the nodes. Additionally, OpenMP is used as mentioned above in order to take advantage of parallelism inside of each node. After executing the updates and calculating the new fitnesses, it is time to execute the movement steps. MPI offers the instruction *MPI_send* to send information from the worker nodes to the central processor and *MPI_Broadcast* for broadcasting global information from the central processor to the workers. In order to reduce the communication, each node calculates its local best and sends only its values, instead of sending all data of all particles it is responsible for.

The combination of master and workers enables a significant speedup compared to a sequential implementation. Nevertheless such a distributed approach also has disadvantages. For simple problems and a small number of particles, just using OpenMP is faster. This behavior is caused by the communication costs in a distributed environment. In some cases, even a sequential execution is faster than one using MPI. On the other hand, with a higher number of particles, a hybrid-approach is much faster than all other approaches. Therefore, in order to have an efficient execution, the programmer must be aware of the complexity and computational costs of the considered problem and the included steps.

Another point concerns the occupancy of the processors. Using frameworks such as MPI, the user is responsible for dividing the tasks among nodes and cores. Ideally, all nodes and cores are equally occupied. Achieving this is not always easy and it gets more complicated, if the execution environment has an heterogeneous structure.

5.1.6 Multi-core CPU vs. GPU

The number of possibilities for parallelizing swarm intelligence algorithms is indeed large. Besides the debate about which parallelization strategy to use, another current topic addresses the use of GPUs versus multi-core CPUs. Both have been widely used and shown to enable considerable speedups and high quality results [69].

Approaches using both types of hardware simultaneously have been also able to achieve impressive results. The use of OpenMP together with CUDA, for example, enables the concurrent processing on the CPU and GPU sides, increasing GPU utilization and reducing the overall execution time of the algorithm [63].

5.2 Discrete Optimization

Some algorithms of the SI family were tailored specifically to solve discrete problems, e.g. ACO. Despite the differences regarding the way these algorithms deal with the search space, many aspects are similar to the ones used in continuous search spaces such as PSO. For example, parallelization at the particle level is also commonly used for these algorithms. An exception can be found in the work of Cicireli et al.[70], where the processes were divided with respect to the search space rather than the particles, but the territory partitioning can lead to an unbalanced workload.

Ant Colony Optimization is an algorithm of the swarm intelligence family, initially created to solve combinatorial problems such as the Traveling Salesperson Problem. From the papers gathered, 20 explore this algorithm or a variant of it. Despite having a rather different way of operating when compared to the previously mentioned algorithms used for continuous optimization, such as PSO and ABC, ACO is also very suitable for a parallelization. The strategies for parallelizing such algorithms, together with pros and cons, will be discussed in the following.

5.2.1 Naive Parallelization

Path creation is the most demanding task in the ACO algorithm when solving problems such as TSP. In every step, an ant must calculate the probability of choosing each possible edge based on the distance and the amount of pheromone deposited on it. On the other hand, the path creation process of one ant is completely independent from that of the other ants. This property makes ACO suitable for parallelization in a very simple manner. As mentioned for continuous optimization techniques, naive parallelization is also possible for ACO. Using one thread per ant was investigated in [71–73]. For both, CPU and GPU, environments, this approach is quite straightforward and capable of achieving considerable speedups, especially when the dimensionality of the optimization problem grows.

5.2.2 Fine-Grained Parallelization

A more advanced strategy for parallelization can be found in [74–78]. Instead of having one thread per ant, the parallelization is performed in more refined way, in order to make better use of the available cores, if there are much more cores than ants. For example, Fingler et. al. [74] use one CUDA thread block to represent one ant and the threads of that block are used to execute internal calculations of the ant in parallel. This is possible, since many parts of the internal calculations are independent of each other, such as the calculation of probabilities during the path creation phase.

The path creation process is the most expensive step during the execution of ACO and therefore it receives most of the attention during the parallelization of the algorithm. However, others steps such as the pheromone deposit also have potential for a parallelization. Zhou et at. proposed a parallel approach for this step using CUDA's atomic operations. The use of atomic operations enables the possibility of initiating several updates concurrently without the problem of race conditions, in the case that two threads try to update the pheromone value of the same edge at the same time. By the atomic operations, conflicting updates are sequentialized. The results show how this parallel approach outperforms the purely sequential pheromone updates [77].

The improvement in the execution times caused by a fine-grained implementation was also observed by Tan and Ying [10] in their work on GPU implementations. As a drawback from this strategy, the authors mention the impossibility of having an efficient synchronization method on the hardware level, since CUDA allows it to happen only between threads that belong to the same CUDA block. A synchronization on the hardware level would require the division of the work into two different CUDA kernels. In such a scenario, threads would have to be spawned 2 times (one time for each kernel) and, therefore, overhead would be generated.

5.2.3 Parallel Ant Colonies

Similarly to the multi-swarm approaches of e.g. PSO, a parallel multi-colony approach can be used with ACO, as it was done by Li [79] and Fingler et al. [74]. Instead of having one single colony, multiple colonies were deployed and processed in parallel. Fingler et al.

state that it is very beneficial to have multiple instances of ACO running at the same time. The authors classify ACO as being an elitist algorithm, where only the most successful ants and the paths that they created are able to influence the behavior of the colony in the next iterations. Therefore, most of the work performed by other ants is basically discarded as they tend to follow the more successful paths. In order to avoid this problem, more possibilities can be generated by having multiple colonies running at the same time, expecting that they will have a bigger diversity of successful solutions.

Gao et al. proposed a multi-colony approach aiming at an environment with multiple GPUs [80]. The authors state that dividing the workload among available devices can be very beneficial to reduce the execution time of the algorithms. As each sub-colony runs independently on a GPU, an eventual synchronization between them is necessary in order to bring them to the same level regarding pheromone deposits. Synchronizing the pheromones between different GPUs can generate notable overhead and therefore the authors suggest that this step should take place sporadically.

Although ACO is aiming to solve discrete problems and the algorithm differs significantly from the ones used for continuous optimization (such as PSO and FSS), the parallelization strategies used in the gathered papers are similar for all of them. Independently of the platform or architecture, each of the mentioned strategies is able to reduce the execution time of the algorithms compared to a sequential execution. However, the actually achieved speedups have to be compared with care, since they depend on many factors, such as the quality of the implementations and the computational power of the hardware used. Nevertheless, all the described approaches give inspirations for own attempts to not only develop parallel implementations of SI metaheuristics but also for parallel implementations in other application areas, in particular in areas, where one level of parallelism is not sufficient to fully exploit the possibilities of the available hardware.

6 Conclusion

This document is the result of a literature review of the latest papers up to 2021 that approach the theme *parallelization of swarm intelligence algorithms*. The objective was to identify the most recent trends about this topic, including algorithms, parallelization frameworks and, most importantly, parallelization strategies. This literature review followed a systematic approach to gather the most relevant papers in the area available in the most relevant scientific databases. The research question, keywords, inclusion criteria and many other parameters helped to guide the paper selection and extraction of relevant information.

In the process, more than 300 papers were extracted from the three selected scientific databases. After the whole process including the removal of duplicates and checking the inclusion criteria, 85 papers remained. These papers were used for data extraction. They were also evaluated and their contributions were discussed. In particular, aspects regarding parallelization strategies of SI metaheuristics were analyzed and compared in order to give a clear view of the state of art and possible trends in the area.

The contribution of this document is not only a review that summarizes the latest papers about the considered topic. It can also serve as a reference when aiming for an own parallel implementation of an SI metaheuristic or other applications with similar characteristics, such as e.g. evolutionary algorithms and genetic algorithms. This is also more generally true for some other application areas which require more than one level of parallelism to be used in order to fully exploit the available hardware. Up to four levels of parallelism have been utilized. Besides the obvious parallel treatment of particles, there is also a parallelization of internal operations (such as the computation of probabilities in ACO), a fine-grained parallelization across the dimensions of the search space (e.g. when evaluating the fitness of a particle), and, most interestingly, a division into parallel collaborating swarms. Additionally, we have also raised some points that were not totally clarified by the authors of the selected papers and that can motivate further research, such as e.g. the use of hybrid swarms.

We can conclude that Swarm Intelligence metaheuristics are well suited for a parallelization. In all gathered papers, approaches were proposed, which enabled significant speedups. There were also variants of classical algorithms especially created for parallel environments. Also, several parallelization frameworks have been used; the most popular ones being CUDA, OpenMP and MPI, but there were also approaches using high-level approaches for parallel programming such as algorithmic skeletons.

The key point identified to improve the performance and reduce the execution time of SI algorithms was the wise use of resources. In order to solve problems in an efficient way, the programmer must be aware not only of the algorithm but also of the execution environment. A little surprising observation was that having more resources, such as a distributed memory system with several nodes, is not a guarantee for a faster execution. For smaller problems, the costs of communication can be higher than those of all the other steps, such that the use of such hardware is not justified. On the other hand, approaches with parallel collaborative swarms turn out to be an interesting alternative allowing to take advantage of such distributed environments. The improvements when compared to a single swarm can be huge, especially when considering hybrid swarms. From the latter, many new aspects emerge, regarding communication patterns, topology and so on.

Furthermore, we can state that the task of parallelization can be quite challenging for programmers due to the variety of approaches and different frameworks that can be used. Some experience is required in order to extract high performance from parallel applications and the hardware being used. Therefore, a further investigation on means that could be used for easing the implementation process would be of good use. For example, the combination of high-level parallelization frameworks and SI algorithms.

We conclude stating that this article provided a very decent perspective in the state of the art regarding the parallelization of swarm intelligence algorithms. Naturally, further research is needed to explore the gaps identified and so to create new methods that capitalize on the synergistic combination of SI and parallel HW platforms. More families of SI algorithms would also be interesting to broaden this study.

Funding Open Access funding enabled and organized by Projekt DEAL.

Appendix 1: Reviewed Papers

See Tables, 2, 3, 4, 5, 6, 7 and 8.

Table 2 PSO Hardware and Framework - GPUs

	Hardware			Frameworks				
Paper	GPUs	CPUs	Nodes	CUDA	OpenMP	MPI	HL	Other
PSO [69]	1	2	1	x		x		
GPSO [28]	1	1	1	x				
PSO [29]	1	1	1	x				
PSO [63]	1	1	1	x				
PSO [50]	1	1	1	x				
PSO [88]	1	1	1	x				
PSO [89]	1	1	1	x				
PSO-ANN [90]	1	1	1	x				
PSO [42]	1	1	1	x				x
CPSO [62]	7	1	1	x				
CCPSO [58]	1	2	1	x	x	x		
PSO [43]	1	1	1	x				
PSO [47]	2	1	1	x				
PPSO [91]	1	1	1	x				
PSO [44]	1	1	1	x				
p-MEMPSODE [66]	1	1	1		x			
GPUPSO [30]	1	1	1	x				
G-PCIPSO [48]	1	1	1	x				
PSO[31]	1	1	-	x				
PSO [32]	1	1	-	x				
GPU-PSO [45]	1	1	-	x				
PSO [92]	1	1	30	x				OpenCL[14]
DPSO [93]	1	1	–	x				
PSO [94]	1	1	–	x				OpenACC[95]
CO-PSO [34]	1	1	–	x				
PSO-QN [37]	1	1	1	x				
MOBPSO [35]	1	1	–	x				
DPSO [49]	1	1	–	x				
PSO [36]	1	1	32	x				
PSO [46]	1	1	–	x				
PSO [55]	1	1	–	x				
PSO [96]	1	1	–	x				
PSO [27]	1	1	–	x				

Table 3 PSO Parallelization Strategies - CPUs

	Parallelization			Synchronization		Communication			
Paper	Particles	Dimensions	Swarms	Sync	Async	Global	Local	Exchange	Speedup
PSO [65]	x			x		x			30x
PSO [69]	x			x		x			10x
PSO [81]	x			x		x			7.3x
PPSO [59]			x	x		x			NA
binary PSO [82]			x	x		x		x	NA
GA-PSO [60]			x	x		x		x	9.1x
PSO [61]			x	x		x		x	6.21x
AIU-PSO [64]	x				x	x			30x
PTVPSO-SVM [83]	x			x		x			10x
PSOwVV [68]	x			x		x			20x
PSO [51]	x		x	x			x		11.2x
pMOPSO [67]	x			x		x			16x
PSO [84]			x	x		x			3.53x
PSO [85]	x			x		x			2x
MMPSO [49]	x		x	x		x			7x
VEPSO [57]	x		x	x		x			10x
PSO [33]	x			x	x	x			NA
H-MPSO-SA [39]	x			x		x			NA
MOPSO [86]			x	x	x	x	x		NA
CAPPSO[56]			x		x	x			15.7x
DEEPSO[52]	x		x	x		x			NA
MOPSO[41]			x	x		x		x	3.47x
PSO[54]			x	x		x			NA
MO-TLPSO[87]			x	x		x			NA

Table 4 PSO Parallelization Strategies - GPUs

	Parallelization			Synchronization		Communication			
Paper	Particles	Dimensions	Swarms	Sync	Async	Global	Local	Exchange	Speedup
PSO [29]	x			x		x			200x
PSO [63]	x		x	x				x	346x
PSO [50]	x		x	x		x			4x
PSO [88]	x			x			x	x	NA
PSO [89]	x			x		x			12x
PSO-ANN [90]	x			x		x			327x
PSO [42]	x			x		x			6.25x
CPSO [62]	x	x	x	x		x	x		79.4x
CPPSO [58]			x	x		x	x		81.6x
PSO [43]	x	x			x	x			5x
PSO [47]	x			x		x			1000x
PPSO [91]	x			x		x			133x
PSO [44]	x		x	x		x	x		100x
p-MEMPSODE [66]	x			x		x			16x
GPUPSO [30]		x		x		x			136x
G-PCIPSO [48]	x		x	x		x			7.95x
PSO [31]	x			x		x			152x
PSO [32]	x			x		x			13.06x
GPU-PSO [45]		x	x	x		x			1.318x
PSO [92]	x		x	x		x			NA
DPSO [93]	x			x		x			87x
PSO [94]	x			x		x			11.5x
CO-PSO [34]	x			x		x	x		4.62x
PSO-QN [37]	x			x		x			8.9x

Table 4 continued

Paper	Parallelization			Synchronization		Communication			Speedup
	Particles	Dimensions	Swarms	Sync	Async	Global	Local	Exchange	
MOBPSO [35]	x			x		x			NA
DPSO [49]	x		x	x		x			13.76x
PSO [36]	x	x	x	x		x			>200x
PSO [46]	x	x		x		x			NA
PSO [55]	x			x		x			190 ×
PSO [96]		x		x		x			10×
PSO [27]	x			x		x			4.3×

Table 5 Continuous Optimization - Hardware and Framework

	Hardware			Frameworks				
Paper	GPUs	CPUs	Nodes	CUDA	OpenMP	MPI	HL	Other
CUBA [21]	1	1	1	x				
MOABC [23]	-	1	1		x			
ABC [20]	1	1	1					MATLAB
ABC [22]	1	1	1					MAS
BCO [24]	–	1	1			x		
ABC [25]	–	1	1					Matlab
FSS [26]	–	1	16		x	x	Muesli	
ABC [97]	–	1	1			x		
HSA [98]	1	1	1			x		OpenCL

Table 6 Continuous optimization - parallelization strategies

	Parallelization			Synchronization		Communication			
Paper	Particles	Dimensions	Swarms	Sync	Async	Global	Local	Exchange	Speedup
CUBA [21]			x	x			x		56x
MOABC [23]			x	x	x		x		40x
ABC [20]	x			x		x			10x
ABC [22]			x	x		x			2x
BCO [24]	x			x	x	x			10x
ABC [25]	x		x	x		x			2x
FSS [26]	x	x		x		x			2x
ABC [97]			x	x		x			1x
HSA [98]	x			x		x			NA

Table 7 ACO hardware and framework

	Hardware			Frameworks				
Paper	GPUs	CPUs	Nodes	CUDA	OpenMP	MPI	HL	Other
ACO [74]	1	1	1	x				
ACO [75]	1	1	1	x				
ACO [73]	1	1	1	x				
ACO [99]	4	1	1	x				
ACO [100]	-	1	1		x			
CACO [101]	1	1	1	x				
GACO [79]	1	1	1	x				
ACO [102]	1	1	1	x				

Table 7 continued

	Hardware			Frameworks				
Paper	GPUs	CPUs	Nodes	CUDA	OpenMP	MPI	HL	Other
RMACO [103]	–	1	1			x		
FGWC-ACO [104]	1	1	1	x				
ACO [71]	1	1	1	x				
ACO [76]	1	1	1	x				
ACO [105]	1	1	1	x				
ACO [106]	1	1	1	x				
ACO [72]	1	1	1	x				
ACO [107]	1	1	1	x				
ACO [78]	1	1	1	x				
ACO [77]	1	1	1					OpenCL
ACO [80]	2	8	1	x			MATLAB	
ACO [108]	1	1	1	x			Musket	
PMMAS [109]	–	1	10					Java

Table 8 ACO Parallelization Strategies

	Parallelization			Synchronization		
Paper	Ants	Probabilities	Swarms	Sync	Async	Speedup
ACO [74]	x		x	x		NA
ACO [75]	x			x		20x
ACO [73]		x		x		10x
ACO [99]	x			x		20x
ACO [100]	x	x		x		NA
CACO [101]	x	x		x		6x
GACO [79]	x		x	x		40.1x
ACO [102]	x	x		x		22.1x
RMACO [103]			x	x		NA
FGWC-ACO [104]	x			x		9.8x
ACO [71]	x			x		21x
ACO [76]	x	x		x		20x
ACO [105]	x			x		843x
ACO [106]	x	x		x		24.29x
ACO [72]	x	x		x		23.6x
ACO [107]	x			x		2.8x
ACO [78]	x			x		8x
ACO [77]	x			x		3.31x
ACO [80]	x		x	x		7.38x
ACO [108]	x			x		NA
PMMAS [109]	x			x		NA

References

1. Talbi, E.-G.: Metaheuristics: From design to implementation. Wiley Series on Parallel and Distributed Computing, vol. 74. John Wiley & Sons, Hoboken, NJ, USA (2009). https://doi.org/10.1002/9780470496916
2. Eberhart, R., Kennedy, J.: A new optimizer using particle swarm theory. In: Proceedings of the Sixth International Symposium on Micro Machine and Human Science, vol. 1, pp. 39–43. IEEE, (1995). https://doi.org/10.1109/MHS.1995.494215
3. Karaboga, D.: An idea based on Honey Bee Swarm for numerical optimization. Technical Report TR06, Erciyes University (TR06), 10 (2005) arXiv:arXiv:1011.1669v3. https://doi.org/citeulike-article-id:6592152
4. Dorigo, M., Birattari, M., Stutzle, T.: Ant colony optimization. IEEE Comput. Intell. Mag. **1**(4), 28–39 (2006). https://doi.org/10.1109/MCI.2006.329691
5. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming. Scientific and Engineering Computation. MIT Press, Cambridge, MA (2008)
6. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable parallel programming with the message-passing interface, 3rd edn. Scientific and Engineering Computation. MIT Press, Cambridge, MA (2014)
7. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue **6** (2), 40–53 (2008). https://doi.org/10.1145/1365490.1365500
8. Kuchen, H.: A skeleton library. In: Monien, B., Feldmann, R. (Eds.) Proceedings of the 8th International Euro-Par Conference on Parallel Processing. Lecture Notes in Computer Science, vol. 2400, pp. 620–629. Springer, (2002)
9. Alba, E., Almeida, F., Blesa, M., Cabeza, J., Cotta, C., Díaz, M., Dorta, I., Gabarró, J., León, C., Luna, J., Moreno, L., Pablos, C., Petit, J., Rojas, A., Xhafa, F.: MALLBA: A library of skeletons for combinatorial optimisation. In: Monien, B., Feldmann, R. (Eds.) Proceedings of the 8th International Euro-Par Conference on Parallel Processing. Lecture Notes in Computer Science, vol. 2400, pp. 927–932. Springer, Berlin, Heidelberg (2002)
10. Tan, Y.: A survey on GPU-Based implementation of swarm intelligence algorithms. GPU-Based Parallel Implement. Swarm Intell. Algorithms **46**(9), 1–236 (2016). https://doi.org/10.1016/c2015-0-02468-6
11. Shuka, R., Niemann, S., Brehm, J., Mueller-Schloer, C.: Towards an algorithm and communication cost model for the parallel particle swarm optimization. In: ARCS 2016; 29th International Conference on Architecture of Computing Systems, pp. 1–4 (2016)
12. Li, B., Chang, H., Song, S., Su, C., Meyer, T., Mooring, J., Cameron, K.W.: The power-performance tradeoffs of the intel xeon phi on hpc applications. In: 2014 IEEE International Parallel Distributed Processing Symposium Workshops, pp. 1448–1456 (2014). https://doi.org/10.1109/IPDPSW.2014.162
13. Li, A., Song, S.L., Chen, J., Li, J., Liu, X., Tallent, N.R., Barker, K.J.: Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. IEEE Trans. Parallel Distrib. Syst. **31**(1), 94–110 (2020). https://doi.org/10.1109/TPDS.2019.2928289
14. Stone, J.E., Gohara, D., Shi, G.: OpenCL: A parallel programming standard for heterogeneous computing systems. Comput. Sci. Eng. **12**(3), 66–72 (2010). https://doi.org/10.1109/MCSE.2010.69
15. Lalwani, S., Sharma, H., Satapathy, S.C., Deep, K., Bansal, J.C.: A survey on parallel particle swarm optimization algorithms. Arab. J. Sci. Eng. **44**(4), 2899–2923 (2019). https://doi.org/10.1007/s13369-018-03713-6
16. Krömer, P., Platoš, J., Snášel, V.: A brief survey of advances in particle swarm optimization on graphic processing units. 2013 World Congress on Nature and Biologically Inspired Computing, NaBIC 2013, 182–188 (2013). https://doi.org/10.1109/NaBIC.2013.6617859
17. Dorigo, M., Birattari, M.: Ant colony optimization (December) (2006). https://doi.org/10.1109/MCI.2006.329691
18. Bastos Filho, C.J.A., de Lima Neto, F.B., Lins, A.J.C.C., Nascimento, A.I.S., Lima, M.P.: A novel search algorithm based on fish school behavior. In: Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (SMC '08), pp. 2646–2651. IEEE, (2008). https://doi.org/10.1109/ICSMC.2008.4811695
19. García-Nieto, J., Alba, E.: Parallel multi-swarm optimizer for gene selection in DNA microarrays. Appl. Intell. **37**(2), 255–266 (2012). https://doi.org/10.1007/s10489-011-0325-9
20. Singh, A., Deep, K., Grover, P.: A novel approach to accelerate calibration process of a k-nearest neighbours classifier using GPU. J. Parallel Distrib. Comput. **104**, 114–129 (2017). https://doi.org/10.1016/j.jpdc.2017.01.003

21. Luo, G.H., Huang, S.K., Chang, Y.S., Yuan, S.M.: A parallel Bees Algorithm implementation on GPU. J. Syst. Architect. **60**(3), 271–279 (2014). https://doi.org/10.1016/j.sysarc.2013.09.007
22. Yang, L., Sun, X., Peng, L., Yao, X., Chi, T.: An Agent-Based Artificial Bee Colony (ABC) algorithm for hyperspectral image endmember extraction in parallel. IEEE J. Selected Topics Appl. Earth Observ. Remote Sens. **8**(10), 4657–4664 (2015). https://doi.org/10.1109/JSTARS.2015.2454518
23. Santander-Jiménez, S., Vega-Rodríguez, M.A.: On the design of shared memory approaches to parallelize a multiobjective bee-inspired proposal for phylogenetic reconstruction. Inf. Sci. **324**, 163–185 (2015). https://doi.org/10.1016/j.ins.2015.06.040
24. Davidovic, Tatjana, Jaksic, Tatjana, Dusan Ramljak, M.S., Teodorovic, D.: Parallelization strategies for bee colony optimization based on message passing communication protocol. Optimization (2013). https://doi.org/10.1080/02331934.2012.749258
25. Li, S., Li, W., Zhang, H., Wang, Z.: Research and implementation of parallel artificial bee colony algorithm based on ternary optical computer. Automatika **60**(4), 422–431 (2019). https://doi.org/10.1080/00051144.2019.1639118
26. Wrede, F., Menezes, B., Kuchen, H.: Fish school search with algorithmic skeletons. Int. J. Parallel Program. (2018). https://doi.org/10.1007/s10766-018-0564-z
27. Kwolek, B., Rymut, B.: Reconstruction of 3D human motion in real-time using particle swarm optimization with GPU-accelerated fitness function. J. Real-Time Image Proc. **17**(4), 821–838 (2020). https://doi.org/10.1007/s11554-018-0825-5
28. Hung, Y., Wang, W.: Accelerating parallel particle swarm optimization via GPU. Optimiz. Methods Softw. **27**(1), 33–51 (2012). https://doi.org/10.1080/10556788.2010.509435
29. Roberge, V., Tarbouchi, M.: Parallel particle swarm optimization on graphical processing unit for pose estimation. WSEAS Trans. Comput. **11**(6), 170–179 (2012)
30. Wu, Q., Xiong, F., Wang, F., Xiong, Y.: Parallel particle swarm optimization on a graphics processing unit with application to trajectory optimization. Eng. Optim. **48**(10), 1679–1692 (2016). https://doi.org/10.1080/0305215X.2016.1139862
31. Ibrahim, E.D.A.: Parallel implementation of particle swarm optimization variants using graphics processing unit platform. Int. J. Chem. Reactor Eng. **1**(1), 1–14 (2003). https://doi.org/10.1016/j.ijengsci.2010.03.001
32. Ouyang, A., Tang, Z., Zhou, X., Xu, Y., Pan, G., Li, K.: Parallel hybrid PSO with CUDA for lD heat conduction equation. Comput. Fluids **110**, 198–210 (2015). https://doi.org/10.1016/j.compfluid.2014.05.020
33. Ettouil, M., Zarrouk, R., Jemai, A., Bennour, I.: Study of runtime performance for Java-multithread PSO on multiCore machines. Int. J. Comput. Sci. Eng. **1**(1), 1 (2016). https://doi.org/10.1504/ijcse.2016.10015696
34. Dong, L., Quan Li, D., Bo Jiang, F.: A two-stage CO-PSO minimum structure inversion using CUDA for extracting IP information from MT data. J. Central South Univ. **25**(5), 1195–1212 (2018). https://doi.org/10.1007/s11771-018-3818-4
35. Elkhani, N., Muniyandi, R.C., Zhang, G.: Multi-objective binary PSO with kernel P system on GPU. Int. J. Comput. Commun. Control **13**(3), 323–336 (2018)
36. Djenouri, Y., Djenouri, D., Habbas, Z., Belhadi, A.: How to exploit high performance computing in population-based metaheuristics for solving association rule mining problem. Distrib. Parallel Databases **36**(2), 369–397 (2018). https://doi.org/10.1007/s10619-018-7218-4
37. Yan, S., Liu, Q., Li, J., Han, L.: Heterogeneous acceleration of Hybrid PSO-QN algorithm for neural network training. IEEE Access **7**, 161499–161509 (2019). https://doi.org/10.1109/ACCESS.2019.2951710
38. Da Costa, A.L.X., Silva, C.A.D., Torquato, M.F., Fernandes, M.A.C.: Parallel implementation of particle swarm optimization on FPGA. IEEE Trans. Circuits Syst. II Express Briefs **66**(11), 1875–1879 (2019). https://doi.org/10.1109/TCSII.2019.2895343
39. Zemzami, M., Elhami, N., Itmi, M., Hmina, N.: A modified particle swarm optimization algorithm linking dynamic neighborhood topology to parallel computation. Int. J. Adv Trends Comp. Sci. Eng. **8**(2), 112–118 (2019)
40. Yang, L., Sun, X., Li, Z.: An efficient framework for remote sensing parallel processing: Integrating the artificial bee colony algorithm and multiagent technology. Remote Sens. (2019). https://doi.org/10.3390/rs11020152
41. Li, J.Z., Chen, W.N., Zhang, J., Zhan, Z.H.: A parallel implementation of multiobjective particle swarm optimization algorithm based on decomposition. In: Proceedings - 2015 IEEE Symposium Series on Computational Intelligence, SSCI 2015, 1310–1317 (2015). https://doi.org/10.1109/SSCI.2015.187

42. Kalivarapu, V., Winer, E.: A study of graphics hardware accelerated particle swarm optimization with digital pheromones. Struct. Multidiscip. Optim. **51**(6), 1281–1304 (2015). https://doi.org/10.1007/s00158-014-1215-7
43. Qu, J., Liu, X., Sun, M., Qi, F.: GPU-Based parallel particle swarm optimization methods for graph drawing **2017** (2017)
44. Silva, E.H.M., Bastos Filho, C.J.A.: PSO efficient implementation on GPUs using low latency memory. IEEE Latin Am. Trans. **13**(5), 1619–1624 (2015). https://doi.org/10.1109/TLA.2015.7112023
45. Dali, N., Bouamama, S.: GPU-PSO: Parallel particle swarm optimization approaches on graphical processing unit for constraint reasoning: Case of Max-CSPs. Proc. Comp. Sci. **60**(1), 1070–1080 (2015). https://doi.org/10.1016/j.procs.2015.08.152
46. Zou, X., Wang, L., Tang, Y., Liu, Y., Zhan, S., Tao, F.: Parallel design of intelligent optimization algorithm based on FPGA. Int. J. Adv. Manuf. Technol. **94**(9–12), 3399–3412 (2018). https://doi.org/10.1007/s00170-017-1447-y
47. Kneusel, R.T.: Curve-fitting on graphics processors using particle swarm. Optimization **6891** (December), 37–41 (2016). https://doi.org/10.1080/18756891.2013.869901
48. Liu, Z.H., Li, X.H., Wu, L.H., Zhou, S.W., Liu, K.: GPU-accelerated parallel coevolutionary algorithm for parameters identification and temperature monitoring in permanent magnet synchronous machines. IEEE Trans. Industr. Inf. **11**(5), 1220–1230 (2015). https://doi.org/10.1109/TII.2015.2424073
49. Liao, S.-L., Liu, B.-X., Cheng, C.-T., Li, Z.-F., Wu, X.-Y.: Long-term generation scheduling of hydropower system using multi-core parallelization of particle swarm optimization. Water Resour. Manag. **31**(9), 2791–2807 (2017). https://doi.org/10.1007/s11269-017-1662-1
50. Cano, E.G., Rodríguez, K.: A parallel PSO algorithm for a watermarking application on a GPU. Comput. y Sistemas **17**(3), 381–390 (2013)
51. Peng, Y., Peng, A., Zhang, X., Zhou, H., Zhang, L., Wang, W., Zhang, Z.: Multi-Core Parallel Particle Swarm Optimization for the Operation of Inter-Basin Water Transfer-Supply Systems. Water Resour. Manage **31**(1), 27–41 (2017). https://doi.org/10.1007/s11269-016-1506-4
52. Yoshida, H., Fukuyama, Y.: Parallel multi-population differential evolutionary particle swarm optimization for voltage and reactive power control in electric power systems. In: 2017 56th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE), pp. 1240–1245. IEEE, (2017). https://doi.org/10.23919/SICE.2017.8105566
53. Luu, K., Noble, M., Gesret, A., Belayouni, N., Roux, P.F.: A parallel competitive Particle Swarm Optimization for non-linear first arrival traveltime tomography and uncertainty quantification. Comp. Geosci. **113**(2017), 81–93 (2018). https://doi.org/10.1016/j.cageo.2018.01.016
54. Niu, W.-J., Feng, Z.-K., Cheng, C.-T., Wu, X.-Y.: A parallel multi-objective particle swarm optimization for cascade hydropower reservoir operation in southwest China. Appl. Soft Comput. J. **70**, 562–575 (2018). https://doi.org/10.1016/j.asoc.2018.06.011
55. Keles, H.Y.: Embedding parts in shape grammars using a parallel particle swarm optimization method on graphics processing units. Artif. Intell. Eng. Design, Anal. Manuf: AIEDAM **32**(3), 256–268 (2018). https://doi.org/10.1017/S089006041700052X
56. Bourennani, F.: Cooperative asynchronous parallel particle swarm optimization for large dimensional problems. Int. J. Appl. Metaheuristic Comp. **10**(3), 19–38 (2019). https://doi.org/10.4018/IJAMC.2019070102
57. Omkar, S.N., Venkatesh, A., Mudigere, M.: MPI-based parallel synchronous vector evaluated particle swarm optimization for multi-objective design optimization of composite structures. Eng. Appl. Artif. Intell. **25**(8), 1611–1627 (2012). https://doi.org/10.1016/j.engappai.2012.05.019
58. Nedjah, N., Calazan, R.D.M., Mourelle, L.D.M., Wang, C.: Parallel Implementations of the Cooperative Particle Swarm Optimization on Many-core and Multi-core Architectures. Int. J. Parallel Program. **44**(6), 1173–1199 (2016). https://doi.org/10.1007/s10766-015-0368-3
59. Huang, H.-C.: FPGA-based parallel metaheuristic PSO algorithm and its application to global path planning for autonomous robot navigation. J. Intell. Robotic Syst. (2013). https://doi.org/10.1007/s10846-013-9884-9
60. Lee, W.-P., Hsiao, Y.-T., Hwang, W.-C.: Designing a parallel evolutionary algorithm for inferring gene networks on the cloud computing environment. BMC Syst. Biol. **8**(1), 5 (2014). https://doi.org/10.1186/1752-0509-8-5
61. Jalloul, M.K., Al-Alaoui, M.A.: A novel cooperative motion estimation algorithm based on particle swarm optimization and its multicore implementation. Signal Proc.: Image Commun. **39**, 121–140 (2015). https://doi.org/10.1016/j.image.2015.09.010

62. Nedjah, N., De Moraes Calazan, R., De Macedo Mourelle, L.: Particle, dimension and cooperation-oriented PSO parallelization strategies for efficient high-dimension problem optimizations on graphics processing units. Comput. J. **59**(6), 810–835 (2016). https://doi.org/10.1093/comjnl/bxu153
63. Roberge, V., Tarbouchi, M., Okou, F.: Collaborative parallel hybrid metaheuristics on graphics processing unit. Int. J. Comput. Intell. Appl. **14**(01), 1550002 (2015). https://doi.org/10.1142/S1469026815500029
64. Moraes, A.O.S., Mitre, J.F., Lage, P.L.C., Secchi, A.R.: A robust parallel algorithm of the particle swarm optimization method for large dimensional engineering problems. Appl. Math. Model. **39**(14), 4223–4241 (2015). https://doi.org/10.1016/j.apm.2014.12.034
65. Kamal, A., Mahroos, M., Sayed, A., Nassar, A.: Parallel particle swarm optimization for global multiple sequence alignment. Inf. Technol. J. **11**, 8–9981006 (2012)
66. Voglis, C., Hadjidoukas, P.E., Parsopoulos, K.E., Papageorgiou, D.G., Lagaris, I.E., Vrahatis, M.N.: P-MEMPSODE: Parallel and irregular memetic global optimization. Comput. Phys. Commun. **197**, 190–211 (2015). https://doi.org/10.1016/j.cpc.2015.07.011
67. Wu, Q., Cole, C., Spiryagin, M.: Parallel computing enables whole-trip train dynamics optimizations. J. Comput. Nonlinear Dyn. (2015). https://doi.org/10.1115/1.4032075
68. Thulasiram, R.K., Thulasiraman, P., Prasain, H., Jha, G.K.: Nature-inspired soft computing for financial option pricing using high-performance analytics. Concurrency Comput: Practice Exp. **28**(3), 707–728 (2016). https://doi.org/10.1002/cpe.3360
69. Roberge, V., Tarbouchi, M.: Comparison of parallel particle swarm optimizers for graphical processing units and multicore processors. Int. J. Comput. Intell. Appl. **12**(01), 1350006 (2013). https://doi.org/10.1142/S1469026813500065
70. Cicirelli, F., Forestiero, A., Giordano, A.: Transparent and efficient parallelization of swarm algorithms. ACM Trans. Autonom. Adapt. Syst. (TAAS) **11**(2), 1–26 (2016). https://doi.org/10.1145/2897373
71. Kallioras, N.A., Kepaptsoglou, K., Lagaros, N.D.: Transit stop inspection and maintenance scheduling: A GPU accelerated metaheuristics approach. Trans. Res. Part C: Emerg. Technol. **55**, 246–260 (2015). https://doi.org/10.1016/j.trc.2015.02.013
72. Delévacq, A., Delisle, P., Gravel, M., Krajecki, M.: Parallel ant colony optimization on graphics processing units. J. Parallel Distrib. Comp. **73**(1), 52–61 (2013). https://doi.org/10.1016/j.jpdc.2012.01.003
73. Zhang, Y., Li, G.-D.: Using improved parallel ant colony optimization based on graphic processing unit-acceleration to solve motif finding problem. J. Comput. Theor. Nanosci. **11**(3), 878–882 (2014). https://doi.org/10.1166/jctn.2014.3440
74. Fingler, H., Cáceres, E.N., Mongelli, H., Song, S.W.: A CUDA based solution to the Multidimensional Knapsack Problem using the ant colony optimization. Proc. Comp. Sci. **29**(30), 84–94 (2014). https://doi.org/10.1016/j.procs.2014.05.008
75. Cecilia, J.M., Nisbet, A., Amos, M., García, J.M., Ujaldón, M.: Enhancing GPU parallelism in nature-inspired algorithms. J. Supercomp. **63**(3), 773–789 (2013). https://doi.org/10.1007/s11227-012-0770-1
76. Cecilia, J.M., García, J.M., Nisbet, A., Amos, M., Ujaldón, M.: Enhancing data parallelism for ant colony optimization on GPUs. J. Parallel Distrib. Comput. **73**(1), 42–51 (2013). https://doi.org/10.1016/j.jpdc.2012.01.002
77. Zhou, Y., He, F., Hou, N., Qiu, Y.: Parallel ant colony optimization on multi-core SIMD CPUs. Futur. Gener. Comput. Syst. **79**, 473–487 (2018). https://doi.org/10.1016/j.future.2017.09.073
78. Cecilia, J.M., Llanes, A., Abellán, J.L., Gómez-Luna, J., Chang, L.-W., Hwu, W.-M.W.: High-throughput Ant Colony Optimization on graphics processing units. J. Parallel Distrib. Comput. **113**, 261–274 (2018). https://doi.org/10.1016/j.jpdc.2017.12.002
79. Li, F.: GACO: A GPU-based high performance parallel multi-ant colony optimization algorithm. J. Inform. Comput. Sci. **11**(6), 1775–1784 (2014)
80. Gao, J., Sun, Y., Zhang, B., Chen, Z., Gao, L., Zhang, W.: Multi-GPU based parallel design of the ant colony optimization algorithm for endmember extraction from hyperspectral images. Sensors (2019). https://doi.org/10.3390/s19030598
81. Roberge, V., Tarbouchi, M., Labonte, G.: Comparison of parallel genetic algorithm and particle swarm optimization for real-time UAV path planning. IEEE Trans. Industr. Inf. **9**(1), 132–141 (2013). https://doi.org/10.1109/TII.2012.2198665
82. Shimizu, Y., Sakaguchi, T., Miura, T.: Parallel computing for huge scale logistics optimization through binary PSO associated with topological comparison. J. Adv. Mech. Design, Syst., Manuf. **8**(1), 0005–0005 (2014). https://doi.org/10.1299/jamdsm.2014jamdsm0005

83. Chen, H.L., Yang, B., Wang, S.J., Wang, G., Liu, D.Y., Li, H.Z., Liu, W.B.: Towards an optimal support vector machine classifier using a parallel particle swarm optimization strategy. Appl. Math. Comput. **239**, 180–197 (2014). https://doi.org/10.1016/j.amc.2014.04.039
84. Ketabchi, H., Ataie-Ashtiani, B.: Assessment of a parallel evolutionary optimization approach for efficient management of coastal aquifers. Environm. Modell. Softw. **74**, 21–38 (2015). https://doi.org/10.1016/j.envsoft.2015.09.002
85. Xun, W., An, Y., Jie, R.: Application of Parallel Particle Swarm Optimize Support Vector Machine Model Based on Hadoop Framework in the Analysis of Railway Passenger Flow Data in China. Chem. Eng. Trans. **46**(2001), 367–372 (2015). https://doi.org/10.3303/CET1546062
86. de Campos, A., Pozo, A.T.R., Duarte, E.P.: Parallel multi-swarm PSO strategies for solving many objective optimization problems. J. Parallel Distrib. Comp. **126**, 13–33 (2019). https://doi.org/10.1016/j.jpdc.2018.11.008
87. Lalwani, S., Sharma, H.: Multi-objective three level parallel PSO algorithm for structural alignment of complex RNA sequences. Evol. Intel. **14**(3), 1251–1259 (2021). https://doi.org/10.1007/s12065-018-00198-y
88. Ugolotti, R., Nashed, Y.S.G., Mesejo, P., Ivekovic, S., Mussi, L., Cagnoni, S.: Particle swarm optimization and differential evolution for model-based object detection. Appl. Soft Comput. J. **13**(6), 3092–3150 (2013). https://doi.org/10.1016/j.asoc.2012.11.027
89. Rymut, B., Kwolek, B.: Real-time multiview human pose tracking using graphics processing unit-accelerated particle swarm optimization. Concurrency Comput: Practice Exp. **27**(6), 1551–1563 (2015). https://doi.org/10.1002/cpe.3329
90. Chen, F., Tian, Y.-B.: Modeling Resonant frequency of rectangular microstrip antenna using CUDA-based artificial neural network trained by particle swarm optimization algorithm. Appl. Comput. Electromag. Soci. J. **2**(12), 1025–1034 (2014)
91. Chang, Y.-L., Liu, J.-N., Chen, Y.-L., Chang, W.-Y., Hsieh, T.-J., Huang, B.: Hyperspectral band selection based on parallel particle swarm optimization and impurity function band prioritization schemes. J. Appl. Remote Sens. **8**(1), 084798 (2014). https://doi.org/10.1117/1.JRS.8.084798
92. Nagano, K., Collins, T., Chen, C.-A., Nakano, A.: Massively parallel inverse rendering using Multi-objective Particle Swarm Optimization. J. Visualization **20**(2), 195–204 (2017). https://doi.org/10.1007/s12650-016-0369-3
93. Phung, M.D., Quach, C.H., Dinh, T.H., Ha, Q.: Enhanced discrete particle swarm optimization path planning for UAV vision-based surface inspection. Automation. Constr. **81**, 25–33 (2017). https://doi.org/10.1016/j.autcon.2017.04.013
94. Wang, N., Huang, H.-C., Hsu, C.-R.: Parallel optimum design of foil bearing using particle swarm optimization method. Tribol. Trans. **56**(3), 453–460 (2013). https://doi.org/10.1080/10402004.2012.758334
95. OpenACC: OpenACC - More Science Less Programming. https://www.openacc.org/
96. Capozzoli, A., Curcio, C., Liseno, A.: Cuda-based particle swarm optimization in reflectarray antenna synthesis. Adv. Electromag. **9**(2), 66–74 (2020). https://doi.org/10.7716/aem.v9i2.1389
97. Dokeroglu, T., Pehlivan, S., Avenoglu, B.: Robust parallel hybrid artificial bee colony algorithms for the multi-dimensional numerical optimization. J. Supercomput. **76**(9), 7026–7046 (2020). https://doi.org/10.1007/s11227-019-03127-7
98. Perez-Cham, O.E., Puente, C., Soubervielle-Montalvo, C., Olague, G., Aguirre-Salado, C.A., Nuñez-Varela, A.S.: Parallelization of the honeybee search algorithm for object tracking. Appl. Sci. (2020). https://doi.org/10.3390/app10062122
99. Llanes, A., Cecilia, J.M., SÃnchez, A., GarcÃa, J.M., Amos, M., UjaldÃn, M: Dynamic load balancing on heterogeneous clusters for parallel ant colony optimization. Cluster Comput. (2016). https://doi.org/10.1007/s10586-016-0534-4
100. Thiruvady, D., Ernst, A.T., Singh, G.: Parallel ant colony optimization for resource constrained job scheduling. Ann. Oper. Res. **242**(2), 355–372 (2016). https://doi.org/10.1007/s10479-014-1577-7
101. Wang, C., Chen, Z.: Parallel ant colony optimisation algorithm for continuous domains on graphics processing unit. Int. J. Comput. Sci. Math. **4**(3), 231 (2013). https://doi.org/10.1504/IJCSM.2013.057252
102. Uchida, A., Ito, Y., Nakano, K.: Accelerating ant colony optimisation for the travelling salesman problem on the GPU. Int. J. Parallel Emergent Distrib. Syst. **29**(4), 401–420 (2014). https://doi.org/10.1080/17445760.2013.842568
103. Yang, Q., Fang, L., Duan, X.: RMACO :a randomly matched parallel ant colony optimization. World Wide Web **19**(6), 1009–1022 (2016). https://doi.org/10.1007/s11280-015-0369-6
104. Nurmala, N., Purwarianti, A.: Improvement of fuzzy geographically weighted clustering-ant colony optimization using context-based clustering. In: 2015 International Conference on Information

Technology Systems and Innovation, ICITSI 2015 - Proceedings 11(1), 21–37 (2016). https://doi.org/10.1109/ICITSI.2015.7437726

105. Cano, A., Olmo, J.L., Ventura, S.: Parallel multi-objective Ant Programming for classification using GPUs. J. Parallel Distrib. Comput. **73**(6), 713–728 (2013). https://doi.org/10.1016/j.jpdc.2013.01.017
106. Skinderowicz, R.: The GPU-based parallel Ant Colony System. J. Parallel Distrib. Comput. **98**, 48–60 (2016)
107. Borisenko, A., Gorlatch, S.: Comparing GPU-parallelized metaheuristics to branch-and-bound for batch plants optimization. J. Supercomput. **75**(12), 7921–7933 (2019). https://doi.org/10.1007/s11227-018-2472-9
108. de Melo Menezes, B.A., Herrmann, N., Kuchen, H., de Lima, Buarque, Neto, F.: High-level parallel ant colony optimization with algorithmic skeletons. Int. J. Parallel Program. **49**(6), 776–801 (2021). https://doi.org/10.1007/s10766-021-00714-1
109. Le, D.N., Nguyen, G.N., Garg, H., Huynh, Q.T., Bao, T.N., Tuan, N.N.: Optimizing bidders selection of multi-round procurement problem in software project management using parallel max-min ant system algorithm. Comput. Mater. Continua **66**(1), 993–1010 (2021)