We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,600
Open access books available

119,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

Chapter 2

Parallel Ant Colony Optimization: Algorithmic Models and Hardware Implementations

Pierre Delisle

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/54252

1. Introduction

The Ant Colony Optimization (ACO) metaheuristic [1] is a constructive population-based approach based on the social behavior of ants. As it is acknowledged as a powerful method to solve academic and industrial combinatorial optimization problems, a considerable amount of research is dedicated to improving its performance. Among the proposed solutions, we find the use of parallel computing to reduce computation time, improve solution quality or both.

Most parallel ACO implementations can be classified into two general approaches. The first one is the parallel execution of the ants construction phase in a single colony. Initiated by Bullnheimer *et al.* [2], it aims to accelerate computations by distributing ants to computing elements. The second one, introduced by Stützle [3], is the execution of multiple ant colonies. In this case, entire ant colonies are attributed to processors in order to speedup computations as well as to potentially improve solution quality by introducing cooperation schemes between colonies.

Recently, a more detailed classification was proposed by Pedemonte *et al.* [4]. It shows that most existing works are based on designing parallel ACO algorithms at a relatively high level of abstraction which may be suitable for conventional parallel computers. However, as research on parallel architectures is rapidly evolving, new types of hardware have recently become available for high performance computing. Among them, we find multicore processors and graphics processing units (GPU) which provide great computing power at an affordable cost but are more difficult to program. In fact, it is not clear that conventional high-level abstraction models are suitable for expressing parallelism in a way that is efficiently implementable and reproducible on these architectures. As academic and industrial combinatorial optimization problems always increase in size and complexity, the field of parallel metaheuristics has to follow this evolution of high performance computing.

INTECH
open science | open minds

© 2013 Delisle; licensee InTech. This is an open access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The main purpose of this chapter is to complement existing parallel ACO models with a computational design that relates more closely to high performance computing architectures. Emerging from several years of work by the authors on the parallelization of ACO in various computing environments including clusters, symmetric multiprocessors (SMP), multicore processors and graphics processing units (GPU) [5–10], it is based on the concepts of computing entities and memory structures. It provides a conceptual vision of parallel ACO that we believe more balanced between theory and practice. We revisit the existing literature and present various implementations from this viewpoint. Extensive experimental results are presented to validate the proposed approaches across a broad range of computing environments. Key algorithmic, technical and programming issues are also addressed in this context.

2. Literature review on Parallel Ant Colony Optimization

During the past 20 years, the ACO metaheuristic has improved significantly to become one of the most effective combinatorial optimization methods. For about a decade, following this trend, a number of parallelization techniques have been proposed to further enhance its search process. Works on traditional CPU-based parallel ACO can be classified into two general approaches: *parallel ants* and *multiple ant colonies*. These approaches are briefly explained in Sections 2.1 and 2.2. On the other hand, few authors have proposed parallel implementations dedicated to specific architectures. Section 2.3 is dedicated to these *hardware-oriented* approaches. In all cases, a survey of related works is also provided.

2.1. Parallel ants

Works related to the parallel ants approach, which aims to execute the ants tour construction phase on many processing elements, were initiated by Bullnheimer *et al.* [2]. They proposed two parallelization strategies for the Ant System on a message passing and distributed-memory architecture. The first one is a low-level and synchronous strategy that aims to accelerate computations by distributing ants to processors in a master-slave fashion. At each iteration, the master broadcasts the pheromone structure to slaves, which then compute their tours in parallel and send them back to the master. The time needed for these global communications and synchronizations implies a considerable overhead. The second strategy aims to reduce it by letting the algorithm perform a given number of iterations without exchanging information. The authors conclude that this partially asynchronous strategy is preferable due to the considerable reduction of the communication overhead.

The works of Talbi *et al.* [11], Randall and Lewis [12], Islam *et al.* [13], Craus and Rudeanu [14], Stützle [3] and Doerner *et al.* [15] are based on a similar parallelization approach and a distributed memory architecture. Delisle *et al.* [5, 6] implemented this scheme on shared-memory architectures like SMP computers and multi-core processors. They also compared performance between the two types of architectures [7].

2.2. Multiple ant colonies

The multiple ant colonies approach, also based on a message-passing and distributed memory architecture, aims to execute whole ant colonies on available processing elements.

http://dx.doi.org/10.5772/54252

It was introduced by Stützle [3] with the parallel execution of multiple independent copies of the same algorithm. Middendorf *et al.* [16] extended this approach by introducing four information exchange strategies between ant colonies: exchange of globally best solution, circular exchange of locally best solutions, migrants or locally best solutions plus migrants. It is shown that it can be advantageous for ant colonies to avoid communicating too much information and too often. Giving up on the idea of sharing whole pheromone information, they based their strategy on the trade of a single solution at each exchange step.

Chu *et al.* [17], Manfrin *et al.* [18], Ellabib *et al.* [19] and Alba *et al.* [20] have also proposed different information exchange strategies for the multiple ant colony approach. Many parameters are studied like the topology of the links between processors as well as the nature and frequency of information exchanges. These strategies are implemented using MPI on distributed memory architectures. On the other hand, Delisle *et al.* [8] adapted some of them on shared-memory architectures.

2.3. Hardware-oriented parallel ACO

Even though they mostly follow the parallel ants and multiple ant colonies approaches, hardware-oriented approaches are dedicated to specific and untraditional parallel architectures. Scheuermann *et al.* [21, 22] designed parallel implementations of ACO on Field Programmable Gate Arrays (FPGA). Considerable changes to the algorithmic structure of the metaheuristic were needed to take benefit of this particular architecture.

Few authors have tackled the problem of parallelizing ACO on GPU in the form of preliminary work. Catala *et al.* [23] propose an implementation of ACO to solve the Orienteering Problem. Instances of up to a few thousand nodes are solved by building solutions on GPU. Wang *et al.* [24] propose an implementation of the MMAS where the tour construction phase is executed on a GPU to solve a 30 city TSP. Similar implementations are reported by You [25], Zhu and Curry [26], Li *et al.* [27], Cecilia *et al.* [28] and Delévacq *et al.* [9] . Following these works, Delévacq *et al.* [10] have proposed various parallelization strategies for ACO on GPU as well as a comparative study to show the influence of various parameters on search efficiency.

Finally, concerning grid applications, Weis and Lewis [29] implemented an ACO algorithm on an ad-hoc grid for the design of a radio frequency antenna structure. Mocholi *et al.* [30] also proposed a medium grain master-slave algorithm to solve the Orienteering Problem.

In addition to a complete survey, Pedemonte *et al.* [4] proposed a taxonomy for Parallel ACO which is illustrated in Fig. 1. Although it provides a comprehensive view of the field, its relatively high level of abstraction does not capture some important features that are crucial for obtaining efficient implementations on modern high performance computing architectures.

The present work does not seek to replace this taxonomy but rather provides a conceptual view of parallel ACO that relates more closely to real parallel architectures. By bringing together the high-level concepts of parallel ACO and the lower-level parallel computing models, it aims to serve as a methodological framework for the design of efficient ACO implementations.

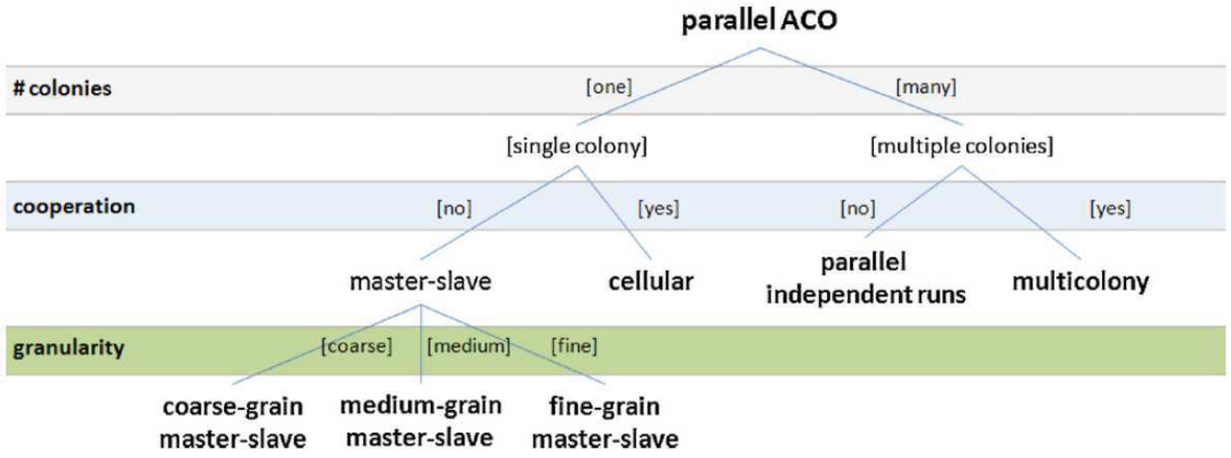

Figure 1. Taxonomy for parallel ACO [4].

3. A new architecture-oriented taxonomy for parallel ACO

The efficient implementation of a parallel metaheuristic in optimization software generally requires the consideration of the underlying architecture. Inspired by Talbi [31], we distinguish the following main parallel architectures: clusters/networks of workstations, symmetric multiprocessors / multicore processors, grids and graphics processing units.

Clusters and Networks of Workstations (COWs/NOWs) are distributed-memory architectures where each processor has its own memory (Fig. 2(a)). Information exchanges between processors require explicit message passing which implies programming efforts and communication costs. NOWs may be seen as an heterogeneous group of computers whereas COWs are homogeneous, unified computing devices.

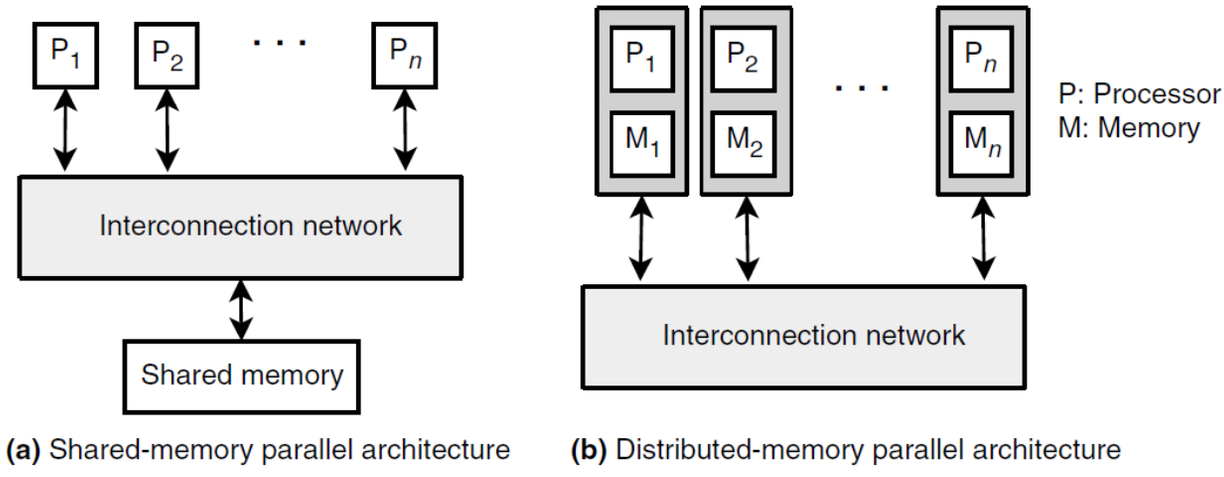

Figure 2. Shared-memory and distributed-memory parallel architectures [31].

Symmetric multiprocessors (SMPs) and multicore processors are shared-memory architectures where the processors are connected to a common memory (Fig. 2(b)). Information exchanges between processors are facilitated by the single address space but synchronizations still have to be managed. SMPs consist of many processors that are linked to a bus network and multicore processors contain many processors on a single chip.

http://dx.doi.org/10.5772/54252

Grids may be considered as pools of heterogeneous and dynamic computing resources geographically distributed across multiple administrative domains and owned by different organizations ([32]). These resources are usually high performance computing platforms connected with a dedicated high-speed network or workstations linked by a nondedicated network such as the Internet. In such volatile systems, security, fault tolerance and resource discovery are important issues to address. Fortunately, middleware usually frees the grid application programmer from much of these issues.

Finally, graphics processing units (GPUs) are devices that are used in computers to manipulate computer graphics. As GPU technology has evolved drastically in the last few years, it has been increasingly used to accelerate general-purpose scientific and engineering applications. As shown in Figure 3, the conventional NVIDIA GPU [33] includes many multiprocessors and processors which execute multiple coordinated threads. Several memories are distinguished on this special hardware, differing in size, latency and access type.

Figure 3. NVIDIA GPU architecture [33].

Considering the variety of architectures currently available in the world of high performance computing, the successful design and implementation of a parallel ACO algorithm on one platform or another may be a significant challenge. Moreover, most computers fall into many categories: a computational cluster may be composed of many distributed nodes which include multicore processors and GPUs. The challenge then becomes two fold: identifying a suitable combination of parallel strategies and implementing it on the target system. In order to make this process simpler, we propose a taxonomy for parallel ACO which takes implementation details into account. It distinguishes three criteria: the ACO granularity level, the "computational entity" associated to that level and the memory structure available at that level.

3.1. ACO granularity level

The decomposition of an ACO algorithm into tasks to be executed by different processors may be performed according to several granularities. One of the main goals of the parallelization process is to find an equitable compromise between the number of tasks and the cost associated to the management of these tasks. Based on the algorithmic structure of

ACO, the proposed classification distinguishes four granularity levels from coarsest to finest: colony, iteration, ant and solution element.

Parallelization at the **colony** level consists in defining the execution of a whole ACO algorithm as a task and assigning it to a processor. The multiple independent colonies and the multiple cooperating colonies approaches, as defined respectively by Stützle [3] and Middendorf *et al.* [16], may be associated to this level. A single colony is typically assigned to a processor but it is possible to assign many with some form of scheduling. At this level, the main factors to consider in the parallelization process are the homogeneity of the colonies as well as their interactions.

Depending on design choices, parallelization at the **iteration** level may be considered as a particular case of either the colony level or the ant level parallelizations. In fact, it may be seen as a hybrid between these two levels instead of a full level. The idea is then to share the iterations of the algorithm between available processors. A first way to implement this strategy is to divide the ants of a single colony into groups and to let each group evolve independently during the algorithm. A second way is to let these groups share their pheromone information after a given number of iterations in a way similar to the partially asynchronous implementation of Bullnheimer *et al.* [2]. At this level, the way the iterations are coordinated between groups will effect the global parallel performance.

Parallelization at the **ant** level implies the distribution of the tasks included in an iteration to available processors. It is mainly the ants construction phase but also operations associated to pheromone update and solution management. This level is related to the typical parallel ants strategy where one or many ants are assigned to each processing element. In that case, special care must be taken to ensure that pheromone updates and general management operations like the identification and update of the best ant do not significantly degrade the performance of the implementation.

Until a few years ago, parallelization at the ant level was generally the finest granularity considered for most optimization problems. However, the emergence of massively parallel architectures like the GPU have resulted in the need for finer approaches. At the **solution element** level, the main operations that are considered for parallelization are the state transition rule and solution evaluation. In the first case, one possible strategy is to evaluate several candidates in parallel to speedup the choice of the next move by an ant. In the second case, the evaluation of the objective function of a particular ant is decomposed among several processors.

The approach proposed in this section sought to determine a parallelization framework taking into account both the main ACO components and the multiple possible granularities. In the next section, it is augmented by considering the underlying computational architecture.

3.2. Computational entity

Nowadays, the typical high performance parallel computer is composed of a hierarchy of several different architectures. For example, it is common to find a computational cluster with multiple distributed SMP nodes, each one of them being composed of multicore processors and GPU cards. Moreover, this type of machine is often found in computational grids. In order to obtain the best possible performance on these platforms, an algorithm has

http://dx.doi.org/10.5772/54252

to be implemented according to at least a part of this hierarchy. The proposed classification distinguishes each level of this hierarchy from the parallel programming perspective. This translates into the definition of five computational entities: system, node, process, block and thread.

A **system** defines a parallel computer as a unified computational resource which may be a standard workstation or a cluster. A distinction is made between these single systems and grids which are considered multiple systems.

A **node** is a discernable part of a system to which tasks can be assigned. A system may then be composed of a single node which is the case of the standard workstation, or of multiple nodes which is the case of clusters.

A **process** is a computational entity that manages and executes sequential and parallel programs. As this concept refers to the typical process in operating systems, it can hold one or many threads which may be grouped together or not. When a process executes only sequential code, it is considered as the smallest indivisible entity of an implementation.

A **block** is an intermediate entity between process and thread. This notion comes from the field of GPU computing in which a block is composed of many threads. The standard processor may be seen as a particular case where a single block is executed. A sequential processor then holds one block and one thread whereas a multicore processor holds one block and several threads.

Finally, a **thread** is a sequential flow of instructions that is part of a block. It represents an indivisible entity and the smallest one in the model: it is always sequential and executes instructions on a processor at a given time. Therefore, even though in practice there may be more threads than processors (some threads will be executed while some others will be idle), in this model we consider that these threads may be merged into a smaller number of threads corresponding to the number of available processors.

Complementary to the notion of computational entity, we add the concept of memory that may be relevant to all five levels previously defined.

3.3. Memory

Memory is an important aspect of ACO algorithms. It serves as a container for pheromone information, problem data and various parameters. It also serves as a channel for information exchange in many parallel implementations. Therefore, as accessibility and access speed will have a significant impact on the feasibility and performance of the parallel implementation, three categories are distinguished: local, global and remote.

Local memory refers to a memory space that is directly accessible by the computational entities of a given level and fast in access time relatively to this particular level. For example, the shared memory of one multiprocessor of a GPU (see Figure 3) is considered as local memory for all the threads that are executed by a block on this multiprocessor. The registers of a processor could also be considered as local memory if they were managed directly, although it is usually not the case.

Global memory is a memory space that can also be accessed directly by the computational entities of a given level, but relatively slow in access time. For example, the device memory of

a GPU is considered as global memory for the threads of a given block. The shared memory of a SMP node is also considered as global memory for the processors or cores of that node.

Remote memory is a memory space that can not be directly accessed by the entities, but for which the information can be made available by an explicit operation between entities. Obviously, remote memory access is considered to be slower than global memory access. For example, the memory available to a processor located in a specific node of a cluster will be considered as remote for the processors on other nodes.

Table 1 summarizes the proposed taxonomy. According to it, designing a parallel ACO implementation implies to link a computational entity and a memory structure to each ACO granularity level. In the next section, two case studies, extracted from the author's previous works, are proposed and expressed according to this taxonomy. In each case, the parallelization strategy and experimental results are synthesized and discussed in order to illustrate various features of the classification.

ACO granularity	Computational entity	Memory
Colony	System	Local
Iteration	Node	Global
Ant	Process	Remote
Solution element	Block	
	Thread	

Table 1. Architecture-based taxonomy for parallel ACO.

4. Case studies

Two case studies are presented to illustrate how the proposed framework relates to real implementations. In order to cover the two main general parallelization strategies for ACO, both parallel ants and multicolony approaches are proposed. In the first case, SMP and muticore processors are considered as underlying architectures. In the second case, a GPU is used as a coprocessor of a sequential processor. This section is then concluded with a more general discussion about how this taxonomy applies to most other combinations of ACO algorithms and parallel architectures.

4.1. Multi-Colony parallel ACO on a SMP and multicore architecture

This approach deals with the management of multiple colonies which use a global shared memory to exchange information. The whole algorithm executes on a single system and a single node so there is no parallelism at these levels. The colonies are executed in parallel and spawn multiple parallel ants. Therefore, colonies are associated to processes and ants to threads. At the programming level, this can be implemented either with multiple operating system processes and multiple threads or with multiple nested threads. In this implementation, we choose the latter as the available SMP node supports nested threads with a shared memory available to all processors. Therefore, this implementation is defined as $COLONY^{global}_{process}$-$ITERATION^{global}_{process}$-$ANT^{global}_{thread}$. There is no additionnal parallelism at the solution element level so it is not specified here.

http://dx.doi.org/10.5772/54252

The proposed implementation is defined assuming a shared-memory model based on threads in which algorithm execution begins with a single thread called the master thread and executed sequentially. To execute a part of the algorithm in parallel, a parallel region is defined where many threads are created, each one of them executing that part of the algorithm concurrently. All threads have access to the whole shared memory, but we can define private data, which is data that will be accessible only by a single thread. Inside a parallel region, we can define a parallel loop, which is a loop where cycles are divided among existing threads in a work-sharing manner. To manage synchronizations between threads, some form of explicit control must be used. A barrier, as the name implies, is a point in the execution of the algorithm beyond which no thread may execute until all threads have reached that point. Also, a critical region is a part of a parallel region which can be executed only by one thread at a time. It is usually used to avoid concurrent writes to shared data. We can now describe the shared-memory parallelization strategy for ACO.

Two versions of the multicolony strategy are proposed which are related to the author's previous work ([6, 8]). The first one, related to parallel independent runs as defined by Stützle [3], implies multiple threads each executing their own copy of the sequential metaheuristic. For the second strategy, we let the colonies cooperate by using a common global best known solution in the shared memory. In both cases, ants are executed in parallel by many nested threads.

In the first implementation, search processes are independent. There are as many copies of data structures as there are colonies. In particular, even if they all reside in the shared memory, pheromone structures are private and exclusive to each thread. ACO parameters are also private, which means that they could be different even if it will not be experimented in this study. In a theoretical context, this kind of parallelization should imply minimal communication and synchronization overheads, hence maximal efficiency. However, this is not the case in a practical context. Even if the data structures are private, colonies need to simultaneously access them through common system resources. At this point, it is up to the computer system to efficiently manage this concurrency.

Parallelizing ACO in multiple search processes is quite simple: we only need to create a parallel region at the beginning of the sequential algorithm. This way, we can create as many threads as we have colonies. A memory location dedicated to store the global best solution known by all processors is reserved in the shared memory and is accessible by all threads. At the end of the parallel region, a critical section lets each thread verify if the best solution it has found qualifies for replacing the global best one and update the data structure accordingly. The best solution of the parallel independent runs can then be identified after the parallel region as the result of the parallel algorithm.

To illustrate the scheme of multiple interacting colonies in a shared-memory model, the simple case of a common best global solution located in the shared memory is implemented. This relates to the first strategy defined by Middendorf [16], that is, exchange of the globally best solution. The exchange rule of this strategy implies that in each information exchange step, the globally best known solution is broadcast to all colonies where it becomes the locally best solution. Information exchanges are performed at each given number of cycles.

In a shared-memory context, there is no such thing as an explicit broadcast communication step. It is replaced by the use of the global best solution as a dedicated structure in the shared memory. However, it is now used differently and more frequently. At each information

exchange step, each thread compare its local value of the best solution with the global best solution. If it has lower cost, it then becomes the new global best known solution. The use of a critical region lets threads do their comparison without risking concurrent writes to the data structure. At this point, the new global best known solution is used by all colonies for the upcoming pheromone update. Since all threads need to have done their comparisons for the new global best solution to be effectively known globally, a synchronization barrier needs to be placed before the pheromone update procedure.

Each colony executes its own ants in parallel by creating a nested group of threads with an additional parallel region. Ants are then distributed to the available processor cores and update the global shared pheromone structure of the colony. Therefore, these updates must be carried out within some form of critical zone to guarantee that unmanaged concurrent writes are avoided. Next subsection shows how these strategies translate into a real computing environment.

4.1.1. Experimental results

The proposed experimentations are based on the Ant Colony System (ACS) applied to the Travelling Salesman Problem ([34]). Both implementations have been experimented on ROMEO II in the Centre de Calcul de Champagne-Ardenne. ROMEO II is a parallel supercomputer of cluster type, consisting of 8 Novascale SMP nodes dedicated to computations. Each node includes 4 Intel Itanium II dual-core processors running at 1.6 GHz with 8MB of cache memory, for a total number of 8 cores, as well as from 16 GB to 128 GB of memory. Each execution is performed on a single node using from 1 to 8 cores. Application code is written in C++ with OpenMP directives for parallelization. The chosen TSP instances range in size from 783 cities to 13 509 cities. For a more detailed version of the experimental setup and results, the reader may consult Delisle *et al.* [8].

Table 2 provides the summary of the experimentations with 1 to 8 independent colonies, each colony residing on a separate core. For each problem and number of cores, the 4 columns provide respectively the speedup, the average tour length, the best tour length and the relative closeness of the average tour length to the optimal solution. For each execution, computed time comes from the last colony that finishes its search and tour length comes from the colony that found the best solution.

We first notice that this implementation is quite scalable. In fact, speedups are relatively close to the number of cores in all configurations. Obviously, there are still some system costs associated to the parallel execution in a shared memory environment, which tend to slightly grow as the number of processors/cores increases. Also, as each core performs the computations associated with a whole ant colony, workload is considerably large in the parallel region. The ratio between parallelism costs and total execution time per core is then greatly reduced.

Table 3 provides results obtained with multiple cooperating colonies. Every 10 iterations, the global best solution is used for the global pheromone update. For the remaining iterations, each colony uses its own best known solution to update its pheromone structure. We first note that the exchange strategy does not significantly hurt the execution time as speedups are still excellent with up to 8 processors. Still, when 4 and 8 processors are used, most efficiency measures are slightly inferior to the ones obtained with independent colonies. This was

http://dx.doi.org/10.5772/54252

Problem	Nb. of cores	Speedup	Avg. tour length	Best tour length	Closeness
rat783	1	-	8,824	8,810	99.80
	2	1.98	8,823	8,806	99.81
	4	3.69	8,820	8,815	99.84
	8	5.93	8,829	8,822	99.74
d2103	1	-	80,511	80,466	99.92
	2	1.97	80,573	80,466	99.85
	4	4.00	80,508	80,477	99.93
	8	6.92	80,501	80,463	99.94
pla7397	1	-	23,365,444	23,353,738	99.55
	2	1.99	23,352,192	23,332,663	99.61
	4	3.80	23,380,613	23,350,736	99.48
	8	7.80	23,425,288	23,396,612	99.29
usa13509	1	-	20,465,969	20,414,755	97.58
	2	1.89	20,376,567	20,250,719	98.03
	4	3.65	20,443,190	20,423,250	97.70
	8	7.30	20,441,068	20,410,519	97.71

Table 2. Multiple independent colonies: number of cores, speedup, average tour length, best tour length and relative closeness of the average tour length to the optimal solution.

Problem	Nb. of cores	Speedup	Avg. tour length	Best tour length	Closeness
rat783	1	-	8,824	8,810	99.80
	2	1.95	8,822	8,810	99.82
	4	3.69	8,819	8,815	99.86
	8	5.72	8,816	8,812	99.89
d2103	1	-	80,511	80,466	99.92
	2	1.95	80,475	80,450	99.97
	4	3.81	80,489	80,450	99.95
	8	6.85	80,484	80,454	99.96
pla7397	1	-	23,365,444	23,353,738	99.55
	2	2.00	23,348,946	23,322,729	99.62
	4	3.89	23,358,733	23,334,364	99.58
	8	7.75	23,356,251	23,350,596	99.59
usa13509	1	-	20,465,969	20,414,755	97.58
	2	2.02	20,456,702	20,392,284	97.63
	4	3.20	20,450,581	20,414,972	97.66
	8	5.55	20,434,287	20,375,145	97.74

Table 3. Multiple cooperating colonies - Global best exchange each 10 cycles: number of cores, speedup, average tour length, best tour length and relative closeness of the average tour length to the optimal solution.

expected as the information exchange steps imply a synchronization cost that grows with the number of colonies used.

Concerning solution quality, the reader may observe that in all cases, the average tour length obtained with multiple cooperating colonies is closer to the optimal solution than with independent colonies or sequential execution. In most cases, the minimum solution found is also better. It shows that the information exchange scheme, while simple, is benefical to solution quality. Overall, results show that a $COLONY^{g}_{process}$-$ITERATION^{global}_{process}$-$ANT^{global}_{thread}$ implementation can be efficiently implemented on a SMP and multi-core computer node containing up to 8 processors.

4.2. Parallel ants on Graphics Processing Units

This approach deals with the execution of a single ant colony on a GPU architecure as defined in the author's previous work ([10]). Ants are associated to blocks and solution elements are associated to threads. As it is shown below, ants may communicate with the relatively slow device memory of the GPU and solution elements may do so with the faster, shared memory of a multiprocessor. As the ACO is not parallelized at the colony and iteration levels, their execution remain sequential and memory structure is not specified. This implementation is then defined as $COLONY^{-}_{process}$-$ITERATION^{-}_{process}$-ANT^{global}_{block}-$SOLUTION\_ELEMENT^{local}_{thread}$. Before providing more details about this implementation, a brief description of the underlying GPU architecture and computational model are given.

As it may be seen in Figure 3, the conventional NVIDIA GPU [33] includes many *Streaming Multiprocessors* (SM), each one of them being composed of *Streaming Processors* (SP). Several memories are distinguished on this special hardware, differing in size, latency and access type (read-only or read/write). *Device memory* is relatively large in size but slow in access time. The *global* and *local* memory spaces are specific regions of the device memory that can be accessed in read and write modes. Data structures of a computer program to be executed on GPU must be created on the CPU and transferred on global memory which is accessible to all SPs of the GPU. On the other hand, local memory stores automatic data structures that consume more registers than available.

Each SM employs an architecture model called *SIMT* (*Single Instruction, Multiple Thread*) which allows the execution of many coordinated threads in a data-parallel fashion. It is composed of a *constant memory cache*, a *texture memory cache*, a *shared memory* and *registers*. Constant and texture caches are linked to the constant and texture memories that are physically located in the device memory. Consequently, they are accessible in read-only mode by the SPs and faster in access time than the rest of the device memory. The constant memory is very limited in size whereas texture memory size can be adjusted in order to occupy the available device memory. All SPs can read and write in their local shared memory, which is fast in access time but small in size. It is divided into memory banks of 32-bits words that can be accessed simultaneously. This implies that parallel requests for memory addresses that fall into the same memory bank cause the serialization of accesses [33]. Registers are the fastest memories available on a GPU but involve the use of slow local memory when too many are used. Moreover, accesses may be delayed due to register read-after-write dependencies and register memory bank conflicts.

http://dx.doi.org/10.5772/54252

GPUs are programmable through different Application Programming Interfaces like CUDA, OpenCL or DirectX. However, as current general-purpose APIs are still closely tied to specific GPU models, we choose CUDA to fully exploit the available state-of-the-art NVIDIA Fermi architecture. In the CUDA programming model [33], the GPU works as a SIMT co-processor of a conventional CPU. It is based on the concept of kernels, which are functions (written in C) executed in parallel by a given number of CUDA threads. These threads are grouped together into *blocks* that are distributed on the GPU SMs to be executed independently of each other. However, the number of blocks that an SM can process at the same time (*active blocks*) is restricted and depends on the quantity of registers and shared memory used by the threads of each block. Threads within a block can cooperate by sharing data through the shared memory and by synchronizing their execution to coordinate memory accesses. In a block, the system groups threads (typically 32) into *warps* which are executed simultaneously on successive clock cycles. The number of threads per block must be a multiple of its size to maximize efficiency. Much of the global memory latency can then be hidden by the thread scheduler if there are sufficient independent arithmetic instructions that can be issued while waiting for the global memory access to complete. Consequently, the more active blocks there are per SM, and also active warps, the more the latency can be hidden.

It is important to note that in the context of GPU execution, flow control instructions (if, switch, do, for, while) can affect the efficiency of an algorithm. In fact, depending on the provided data, these instructions may force threads of a same warp to diverge, in other words, to take different paths in the program. In that case, execution paths must be serialized, increasing the total number of instructions executed by this warp.

In the parallel ants general strategy, ants of a single colony are distributed to processing elements in order to execute tour constructions in parallel. On a conventional CPU architecture, the concept of processing element is usually associated to a single-core processor or to one of the cores of a multi-core processor. On a GPU architecture, the main choices are to associate this concept either to an SP or to an SM. As this case study is concerned with the latter, each ant is associated to a CUDA block and runs its tour construction phase in parallel on a specific SM of the GPU. A dedicated thread of a given block is then in charge of managing the tour construction of an ant, but an additional level of parallelism, the solution element level, may be exploited in the computation of the state transition rule. In fact, an ant evaluates several candidates before selecting the one to add to its current solution. As these evaluations can be done in parallel, they are assigned to the remaining threads of the block.

A simple implementation would then imply keeping ant's private data structures in the global memory. However, as only one ant is assigned to a block and so to an SM, taking advantage of the shared-memory is possible. Data needed to compute the ant state transition rule is then stored in this memory that is faster and accessible by all threads that participate in the computation. Most remaining issues encountered in the GPU implementation of the parallel ants general strategy are related to memory management. More particularly, data transfers between CPU and GPU as well as global memory accesses require considerable time. As it was mentioned before, these accesses may be reduced by storing the related data structures in shared memory. However, in the case of ACO, the three central data structures are the pheromone matrix, the penalty matrix (typically the transition cost between all pairs of solution elements) and the candidates lists, which are needed by all ants of the colony while being too large (typically ranging from $O(n)$ to $O(n^2)$ in size) to fit in shared memory. They are then kept in global memory. On the other hand, as they are not modified during

the tour construction phase, it is possible to take benefit of the texture cache to reduce their access times.

4.2.1. Experimental results

The proposed GPU strategy is implemented into an MMAS algorithm ([35]) and experimented on various TSPs with sizes varying from 51 to 2103 cities. Minimums and averages are computed from 25 trials for problems with less than 1000 cities and from 10 trials for larger instances. An effort is made to keep the algorithm and parameters as close as possible to the original MMAS. Following the guidelines of Barr and Hickman [36] and Alba [37], the *relative speedup* metric is computed on *mean execution times* to evaluate the performance of the proposed implementation. Speedups are calculated by dividing the sequential CPU time with the parallel time, which is obtained with the same CPU and the GPU acting as a co-processor.

Experiments were made on one GPU of an NVIDIA Fermi C2050 server available at the Centre de Calcul de Champagne-Ardenne. It contains 14 SMs, 32 SPs per SM, 48 KB of shared memory per SM and a warp size of 32. The CPU code runs on one core of a 4-core Xeon E5640 CPUs running at 2.67 Ghz and 24 GB of DDR3 memory. Application code was written in the "C for CUDA V3.1" programming environment.

The implementation uses a number of blocks equal to the number of ants, each one of them being composed of a number of threads equal to the size of candidate lists, in that case 20. Also, the number of iterations is set with the intent of globally keeping the same global number of tour constructions for each experiment. For more details on the experimental setup, the reader may consult Delévacq *et al.* ([10]).

A first step in our experiments is to compare solution quality obtained by sequential and parallel versions of the algorithm. Table 4 presents average tour length, best tour length and closeness to the optimal solution for each problem. The reader may note the similarity between the results obtained by our sequential implementation and the ones provided by the authors of the original MMAS ([35]), as well as their significant closeness to optimal solutions.

A second step is to evaluate and compare the reduction of execution time that is obtained with the GPU parallelization strategy. Table 4 shows the speedups obtained for each problem. The reader may notice that speedups are ranging from 6.84 to 19.47. This shows that distributing ants to blocks and sharing the computation of the state transition rule between several threads of a block is efficient. Also, speedup generally increases with problem size, indicating the good scalabilty of the strategy. However, a slight decrease is encountered with the 2103 cities problem. In that case, the large workload and data structures imply memory access latencies and bank conflicts costs that grow faster than the benefits of parallelizing available work. Associated to the combined effect of the increasing number of blocks required to perform computations and a limited number of active blocks per SM, performance gains become less significative. Overall, results show that a $COLONY^{-}_{process}$-$ITERATION^{-}_{process}$-ANT^{global}_{block}-$SOLUTION\_ELEMENT^{local}_{thread}$ implementation can be efficiently implemented on a state-of-the-art GPU.

Problem		Speedup	Stützle and Hoos	Avg. tour length	Best tour length	Closeness
eil51	Sequential	-	427.80	427.32	426	99.69
	Parallel	6.84	-	427.20	426	99.72
kroA100	Sequential	-	21,336.90	21,314.36	21,282	99.85
	Parallel	8.12	-	21,317.32	21,282	99.83
d198	Sequential	-	15,952.30	15,973.84	15,913	98.77
	Parallel	11.13	-	15,961.64	15,851	98.85
lin318	Sequential	-	42,346.60	42,341.72	42,107	99.26
	Parallel	11.03	-	42,325.32	42,147	99.29
rat783	Sequential	-	-	9,042.44	8,923	97.32
	Parallel	15.58	-	9,002.32	8,899	97.77
fl1577	Sequential	-	-	24,490.30	24,201	89.83
	Parallel	19.47	-	24,287.80	23,938	90.84
d2103	Sequential	-	-	82,754.30	82,378	97.14
	Parallel	17.64	-	82,756.00	82,547	97.13

Table 4. GPU implementation: speedup, average tour length from Stützle and Hoos original MMAS implementation [35], average tour length, best tour length and relative closeness of the average tour length to the optimal solution.

5. Conclusion

The main objective of this chapter was to provide a new algorithmic model to formalize the implementation of Ant Colony Optimization on high performance computing platforms. The proposed taxonomy managed to capture important features related to both the algorithmic structure of ACO and the architecture of parallel computers. Case studies were also presented in order to illustrate how this classification translates into real applications. Finally, with its synthesized literature review and experimental study, this chapter served as an overview of current works on parallel ACO.

Still, as it is the case in the field of parallel metaheuristics in general, much can still be done for the effective use of state-of-the-art parallel computing platforms. For example, maximal exploitation of computing resources often requires algorithmic configurations that do not let ACO perform an effective exploration and exploitation of the search space. On the other hand, parallel performance is strongly influenced by the combined effects of parameters related to the metaheuristic, the hardware technical architecture and the granularity of the parallelization. As it becomes clear that the future of computers no longer relies on increasing the performance on a single computing core but on using many of them in a hybrid system, it becomes desirable to adapt optimization tools for parallel execution on many kinds of architectures. We believe that the global acceptance of parallel computing in optimization systems requires algorithms and software that are not only effective, but also usable by a wide range of academicians and practitioners.

Acknowledgements

This work is supported by the Agence Nationale de la Recherche (ANR) under grant no. ANR-2010-COSI-003-03 and by the Centre de Calcul de Champagne-Ardenne ROMEO which provides the computational resources used for experiments.

Author details

Pierre Delisle

CReSTIC, Université de Reims Champagne-Ardenne, Reims, France

6. References

[1] M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT Press/Bradford Books, 2004.

[2] B. Bullnheimer, G. Kotsis, and C. Strauss. Parallelization strategies for the ant system. In R. De Leone, A. Murli, P. Pardalos, and G. Toraldo, editors, *High Performance Algorithms and Software in Nonlinear Optimization*, volume 24 of *Applied Optimization*, pages 87–100. Kluwer, Dordrecht, 1997.

[3] T. Stützle. Parallelisation strategies for ant colony optimization. In A.E. Eiben, T. Bäck, H.-P. Schwefel, and M. Schoenauer, editors, *Proceedings of the Fifth International Conference on Parallel Problem Solving from Nature (PPSN V)*, volume 1498, pages 722–731. Springer-Verlag, New York, 1998.

[4] M. Pedemonte, S. Nesmachnow, and H. Cancela. A survey on parallel ant colony optimization. *Applied Soft Computing*, 11:5181–5197, 2011.

[5] P. Delisle, M. Krajecki, M. Gravel, and C. Gagné. Parallel implementation of an ant colony optimization metaheuristic with openmp. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 3rd European Workshop on OpenMP (EWOMP'01)*, pages 8–12, Barcelona, Spain, 2001.

[6] P. Delisle, M. Gravel, M. Krajecki, C. Gagné, and W. L. Price. A shared memory parallel implementation of ant colony optimization. In *Proceedings of the 6th Metaheuristics International Conference (MIC'2005)*, pages 257–264, Vienna, Autria, 2005.

[7] P. Delisle, M. Gravel, M. Krajecki, C. Gagné, and W. L. Price. Comparing parallelization of an aco: Message passing vs. shared-memory. In M.J. Blesa, C. Blum, A. Roli, and M. Sampels, editors, *Proceedings of the 2nd International Conference on Hybrid Metaheuristics*, volume 3636 of *Lecture Notes in Computer Science*, pages 1–11. Springer-Verlag Berlin Heidelberg, 2005.

[8] P. Delisle, M. Gravel, and M. Krajecki. Multi-colony parallel ant colony optimization on smp and multi-core computers. In *Proceedings of the World Congress on Nature and Biologically Inspired Computing (NaBIC 2009)*, pages 318–323. IEEE, 2009.

[9] A. Delévacq, P. Delisle, M. Gravel, and M. Krajecki. Parallel ant colony optimization on graphics processing units. In H. R. Arabnia, S. C. Chiu, G. A. Gravvanis, M. Ito, K. Joe, H. Nishikawa, and A. M. G. Solo, editors, *Proceedings of the 16th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'10)*, pages 196–202. CSREA Press, 2010.

[10] A. Delévacq, P. Delisle, M. Gravel, and M. Krajecki. Parallel ant colony optimization on graphics processing units. *Journal of Parallel and Distributed Computing*, page doi : 10.1016/j.jpdc.2012.01.003, 2012.

[11] E. Talbi, O. Roux, C. Fonlupt, and D. Robillard. Parallel ant colonies for the quadratic assignment problem. *Future Generation Computer Systems*, 17(4):441–449, 2001.

[12] M. Randall and A. Lewis. A parallel implementation of ant colony optimization. *Journal of Parallel and Distributed Computing*, 62(9):1421–1432, 2002.

[13] M. T. Islam, P. Thulasiraman, and R. K. Thulasiram. A parallel ant colony optimization algorithm for all-pair routing in manets. In *Proceedings of the 17th international Symposium on Parallel and Distributed Processing*. IEEE Computer Society, 2003.

[14] M. Craus and L. Rudeanu. Parallel framework for ant-like algorithms. In *Proceedings of the Third International Symposium on Parallel and Distributed Computing (ISPDC/HeteroPar'04)*, pages 36–41, 2004.

[15] K. Doerner, R. Hartl, S. Benker, and M. Lucka. Parallel cooperative savings based ant colony optimization - multiple search and decomposition approaches. *Parallel Processing Letters*, 16(3):351–370, 2006.

[16] M. Middendorf, F. Reischle, and H. Schmeck. Multi colony ant algorithms. *Journal of Heuristics*, 8(3):305–320, 2002.

[17] D. Chu and A. Y. Zomaya. Parallel ant colony optimization for 3d protein structure prediction using the hp lattice model. In N. Nedjah, L. de Macedo, and E. Alba, editors, *Parallel Evolutionary Computations*, volume 22 of *Studies in Computational Intelligence*, chapter 9, pages 177–198. Springer, 2006.

[18] M. Manfrin, M. Birattari, T. Stützle, and M. Dorigo. Parallel ant colony optimization for the traveling salesman problem. In *Proceedings of the 5th International Workshop on Ant Colony Optimization and Swarm Intelligence*, volume 4150 of *Lecture Notes in Computer Science*, pages 224–234, 2006.

[19] I. Ellabib, P. Calamai, and O. Basir. Exchange strategies for multiple ant colony system. *Information Sciences*, 177(5):1248–1264, 2007.

[20] E. Alba, G. Leguizamon, and G. Ordonez. Two models of parallel aco algorithms for the minimum tardy task problem. *International Journal of High Performance Systems Architecture*, 1(1):50–59, 2007.

[21] B. Scheuermann, K. So, M. Guntsch, M. Middendorf, O. Diessel, H. ElGindy, and H. Schmeck. Fpga implementation of population-based ant colony optimization. *Applied Soft Computing*, 4:303–322, 2004.

[22] B. Scheuermann, S. Janson, and M. Middendorf. Hardware-oriented ant colony optimization. *Journal of Systems Architecture*, 53:386–402, 2007.

[23] A. Catala, J. Jaen, and J. Mocholi. Strategies for accelerating ant colony optimization algorithms on graphical processing units. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 492–500. IEEE Press, 2007.

[24] J. Wang, J. Dong, and C. Zhang. Implementation of ant colony algorithm based on gpu. In E. Banissi, M. Sarfraz, J. Zhang, A. Ursyn, W. C. Jeng, M. W. Bannatyne, J. J. Zhang, L. H. San, and M. L. Huang, editors, *Proceedings of the Sixth International Conference on*

Computer Graphics, Imaging and Visualization: New Advances and Trends, pages 50–53. IEEE Computer Society, 2009.

[25] Y. You. Parallel ant system for traveling salesman problem on gpus. In *Proceedings of GECCO 2009 - Genetic and Evolutionary Computation*, pages 1–2, 2009.

[26] W. Zhu and J. Curry. Parallel ant colony for nonlinear function optimization with graphics hardware acceleration. In *Proceedings of the 2009 IEEE international conference on Systems, Man and Cybernetics*, pages 1803–1808. IEEE Press, 2009.

[27] J. Li, X. Hu, Z. Pang, and K. Qian. A parallel ant colony optimization algorithm based on fine-grained model with gpu-acceleration. *International Journal of Innovative Computing, Information and Control*, 5(11(A)):3707–3716, 2009.

[28] J. M. Cecilia, J. M. Garcia, A. Nisbet, M. Amos, and M. Ujaldon.

[29] G. Weis and A. Lewis. Using xmpp for ad-hoc grid computing - an application example using parallel ant colony optimisation. In *Proceedings of the International Symposium on Parallel and Distributed Processing*, pages 1–4, 2009.

[30] J. Mocholí, J. Martínez, and J. Canós. A grid ant colony algorithm for the orienteering problem. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 942–949. IEEE Press, 2005.

[31] E. Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009.

[32] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[33] *CUDA : Computer Unified Device Architecture Programming Guide 3.1*, 2010.

[34] M. Dorigo and L. M. Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

[35] T. Stützle and H. Hoos. Max-min ant system. *Future Generation Computer Systems*, 16(8):889–914, 2000.

[36] R. S. Barr and B. L. Hickman. Reporting computational experiments with parallel algorithms : Issues, measures and experts' opinions. *ORSA Journal on Computing*, 5(1):2–18, 1993.

[37] E. Alba. Parallel evolutionary algorithms can achieve super-linear performance. *Information Processing Letters*, 82(1):7–13, 2002.