Journal of Supercomputing manuscript No.
(will be inserted by the editor)

Reenginering the Ant Colony Optimization for CMP Architectures

José M. Cecilia · José M. García

Received: date / Accepted: date

Abstract The Ant Colony Optimization (ACO) is inspired by the behavior of real ants and, as a bioinspired method; its underlying computation is massively parallel by definition. This paper shows re-engineering strategies to migrate the ACO algorithm applied to the Traveling Salesman Problem (TSP) to modern Intel-based multi-and-many-core architectures in a step-by-step methodology. The paper provides detailed guidelines on how to optimize the algorithm for the intra-node (thread and vector) parallelization, showing the performance scalability along with the number of cores on different Intel architectures, reporting up to 5.5x speed-up factor between the Intel Xeon Phi Knights Landing (KNL) and Intel Xeon v2. Moreover, parallel efficiency is provided for all targeted architectures, finding that core load imbalance, memory bandwidth limitations, and NUMA effects on data placement are some of the key factors limiting performance. Finally, a distributed implementation is also presented, reaching up to 2.96x speed-up factor when running the code on 3 nodes over the single-node counterpart version. In the latter case, the parallel efficiency is affected by the synchronization frequency, which also affects the quality of the solution found by the distributed implementation.

Keywords Parallel and Distributed ACO · CMP code redesign · Intel Xeon Phi · Performance Evaluation · Ant Colony Optimization · TSP

José M. Cecilia
Universidad Católica San Antonio de Murcia
Campus de los Jerónimos s/n. 30107 Guadalupe
Tel.: +34-968-278587
E-mail: jmcecilia@ucam.edu

José M. García
Facultad de Informática. Universidad de Murcia
Campus de Espinardo s/n. - 30080 Murcia (SPAIN)
Tel.: +34-868-884819
E-mail: jmgarcia@um.es

1 Introduction

Many real-world problems, such as route scheduling, goods dispatching, protein folding, etc., may benefit from the use of computers to find their solution. They result in NP-hard optimization problems, which are easy to define but really difficult to solve, from a computational point of view.

Metaheuristics have emerged as a novel solution to provide approximate solutions in a reduced time frame [1] for these types of challenging problems. Among them, bioinspired metaheuristics (i.e., those methods inspired by natural procedures) are gaining special interest within the research community. Examples of bioinspired methods include ant colony optimization (ACO), particle swarm optimization (PSO), or genetic algorithms (GAs) [32]. Although they offer very good solutions for optimization problems, they still require too much execution time to generate optimal solutions. However, they are inherently parallel by their own definition and they are therefore well-suited for parallelization on the current massively parallel architectures.

Of particular interest to us is the ACO algorithm [2] which is inspired in the ants foraging process. ACO uses ants as artificial agents to explore a graph where a complete trajectory is a solution to a given problem. All the solutions found by the ants are evaluated depending on their quality. Thus, ants can deposit "pheromone" in the paths of each solution, according to its quality. ACO is based on two main stages: *tour construction* and *pheromone update*. In the first stage, each ant builds a path choosing the next action to perform among those that have not yet been taken. Then, the pheromone update is performed which consists of two phases: *pheromone evaporation* to gradually forget unpromising solutions and *pheromone deposit* to reinforce high quality solutions.

The first problem solved by ACO was the Traveling Salesman Problem (TSP) [2]. TSP is computationally expensive (i.e. $O(n^3)$ at each iteration, where n is the size of the problem) but massively parallel by its definition. Therefore, the research community has provided several ways to optimize the ACO algorithm for TSP (ACO-TSP) in High Performance Computing (HPC) architectures. The first attempts were carried out on NVIDIA GPUs using CUDA [4–7], and more recently on the first generation of Intel Xeon Phi (Knights Corner) [8–10]. Intel architectures offer several advantages compared to Nvidia architectures. These include code portability across all high-end architectures, which allow you to scale across a large number of cores and nodes. Besides, Intel architectures can be programmed using a popular high level programming language such as OpenMP, which shows a smaller learning curve compared to CUDA.

In this work, we provide detailed guidelines on how to redesign the ACO-TSP algorithm to leverage multi-and-many-core Intel-based architectures. A systematic approach is performed by starting the optimization process in a single node, fine-tuning both thread and vectorization parallelism. An additional level of parallelism is then introduced by providing a distributed implementation of ACO-TSP for multiple nodes. Our work focuses only on the tour construction stage, as this phase takes 95% of the runtime of the sequential version of ACO-TSP, and it is common to all ACO variants.

The main novelties of this paper include the following:

– We provide the re-engineering process to migrate ACO-TSP to Intel architectures in a step-by-step methodology, giving detailed guidelines on how to optimize the algorithm for the intra-node (thread and vector) parallelization.
– We also introduce a distributed implementation for a different number of computing nodes, distributing the iterations among all the nodes. To avoid the degradation of the solution quality, our implementation relies on a synchronization mechanism to exchange and combine pheromone matrices.
– Evaluation results are carried out by targeting Intel Xeon processors, ranging from few cores (Xeon E5-2650 v2 and E5-2698 v4) to high-end many-core (7120P Knights Corner and 7250 Knights Landing) processors. In terms of performance, the Intel Xeon Phi KNL (68 cores) version is the fastest architecture in our study, defeating by all other architectures by a wide margin (up to 5.5x speed-up factor). In addition, the distributed implementation running on 3 nodes reaches a speedup factor of up to 2.96x compared to its counterpart version executed in 1-node with only a final synchronization.
– Finally, parallel efficiency is analyzed by running the largest benchmark (up to 7397 cities) in all the architectures under study, which have different number of cores. For parallel efficiency on a single platform, the main problems detected are the core load imbalance, memory bandwidth limitations and the effects of NUMA on data placement. In the case of parallel efficiency between nodes, the main problem is related to the frequency of synchronization between nodes that also affects the quality of the solution found by the distributed implementation.

The rest of the paper is organized as follows. Section 2 briefly introduces the ACO metaheuristic and its application for the TSP problem, Intel Xeon Phi main features and the code modernization process. Extensive guidelines on porting ACO-TSP to CMP architectures are given in Section 3. Our experimental results are presented in Section 4. Section 5 gives an overview of the related work on implementing ACO-TSP in GPUs and CMP architectures. Finally, we summarize our findings and give some suggestions for future work in Section 6.

2 Background

2.1 ACO Foundations

In this paper, we use ACO for solving the Traveling Salesman Problem (TSP). The Traveling Salesman Problem (TSP) [11] involves finding the shortest (or *cheapest*) round-trip route that visits each city exactly once. The symmetric

TSP on n cities may be represented as a complete weighted graph, G, of n nodes, with each weighted edge, $e_{i,j}$, representing the inter-city distance $d_{i,j} = d_{j,i}$ between cities i and j. The general structure of the ACO, when it is applied to a combinatorial optimization problem such as the TSP, is based on performing a number of iterations until some end criteria is met. Each iteration is composed of two main stages: *tour construction* and *pheromone update*. The tour construction stage is the same for all ACO variants. At the start of this stage, each ant is placed on a randomly chosen initial city. Then, each ant makes use of a probabilistic rule in order to choose its next city to visit, until it builds a complete tour. The probability for ant k, currently placed at city i, of selecting city j is specified in Equation (1).

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in N_i^k} [\tau_{ij}]^\alpha [\eta_{ij}]^\beta}, \quad if\ j \in N_i^k, \tag{1}$$

τ_{ij} is the amount of pheromone associated with edge (i, j), $\eta_{ij} = 1/d_{ij}$ (d_{ij} is the length of edge (i, j)) is the reciprocal of the distance value computed *a priori*, α and β are two input parameters, which determine the influence of the pheromone trail and the heuristic information, and N_i^k is the set of cities that have not been visited yet by ant k, currently placed at city i. The probability of choosing a city outside this set is 0, thus preventing an ant from visiting a city more than once. According to this probabilistic rule, the probability of selecting an edge (i, j) increases with the amount of pheromone on that edge (τ_{ij}) and the distance information value (η_{ij}).

Once the probabilities have been computed, a selection function is used for choosing the next city to visit taking into account these probabilities. Roulette Wheel (RW) selection was first suggested as the default selection function [2], but other selection procedures have been proposed in the literature, which are better suited for parallelization [4,5]. In this paper, we use I-Roulette [4], a selection mechanism that is fully parallel [10,13]. Moreover, this function accelerates convergence to a solution without affecting the quality of the solutions achieved [12]. In I-Roulette, the probability of visiting each city is multiplied by a random number between 0 and 1, obtaining a weight for each city. The city with the highest weight is selected as the next city to visit.

Once ants have finished their tours, pheromone update is performed by applying of two phases: *pheromone evaporation*, in order to gradually evaporate the pheromone trails to avoid stalling in a local optimum and *pheromone deposit* to reinforce good quality solutions. Firstly, pheromone trails are evaporated by means of lowering the pheromone value on all edges by a constant factor (ρ). Then, each ant deposits pheromone on each edge from its tour proportionally to the tour's quality. The pheromone update process previously described is used in Ant System [15], the first ACO algorithm proposed, but each ACO variant introduces some changes to this stage.

2.2 Intel many-core (Xeon Phi) Features

The Intel Xeon Phi is based on the Many Integrated Core (MIC) architecture [16]. The first generation of this many-core architecture, also known as Knight's Corner (KNC), was launched in 2012, and the second generation, code-named Knight's Landing (KNL), in 2016. They have a high number of cores (57-61 for KNC and 64-72 for KNL) and four hardware threads per core, allowing the use of up to 244 (KNC) and 288 (KNL) threads. Each core is provided with one (in KNC) or two (in KNL) vector processing units (VPU), which can operate on 512-bit wide registers. KNC cores have in-order instruction execution, while KNL has out-of-order cores. They both run at a low base clock frequency (less than 1.3 GHz for KNC and 1.5 for KNL). Despite having single cores, Intel's multi-core architectures have a better performance/power ratio than Intel's multi-core processors. However, it is necessary to make use of both thread parallelism and vectorization to exploit all the hardware capabilities of these architectures.

Xeon Phi KNC and KNL are programmed using C, C++ or Fortran, with OpenMP extensions [17] for thread parallelism, MPI extensions [18] for distributed memory parallelism, and compiler directives and hints for vectorization. Moreover, the same code is able to run on Xeon Phi[1] and Xeon.

Two additional new features included in the Xeon Phi KNL are (1) its high bandwidth memory (HBM) and (2) its clustering modes. KNL has up to 16 GiB of MCDRAM based on-package HBM, which can be used either as a last-level cache or as addressable memory. It also provides different cache organization modes, called clustering modes [31]. In contrast to the Xeon Phi KNC, which has a single socket, the Xeon Phi KNL can be configured as sub-NUMA (i.e., *Non-Uniform Memory Access*) cluster modes (SNC-2 or SNC-4) that divide the chip into two or four clusters, having these clusters as NUMA nodes. NUMA architectures have a different memory latency depending on the NUMA node accessing the data, and may also vary depending on the consistency state of the accessed data.

2.3 Code Reengineering Process

Code Reengineering or *Code modernization*[2] is a process that consists on redesigning and tuning applications to take advantage of all hardware resources they are running on in order to achieve peak performance for a particular platform. As an initial step, the programmer must perform some scalar or core optimizations, such as choosing the correct numerical accuracy (single or double for floats, etc.), and avoiding type conversions and repetitive calculations, to name just a few. Typically, the reengineering process includes the following stages:

[1] In the case of Xeon Phi KNC, the code needs to be recompiled with the `-mmic` option.

[2] As it is named by Intel [19,20].

1. *Thread parallelism*: It divides the work among different threads that may share information via shared memory. The programmer should profile the thread scaling, making sure they are mapped to different cores (thread affinity) to avoid thread synchronization or inefficient memory utilization.
2. *Vectorization*: It identifies parts of the code in which the same instructions are performed on different data. The programmer shall use compiler directives to ease vectorization and data layout optimizations.
3. *Distributed memory parallelism (cluster computing)*: It distributes the work among different computing nodes via some message passage system (e.g., MPI).

It is worth highlighting that all the optimizations are orthogonal to the *memory traffic optimization*, which must always be taken into account at all stages.

3 ACO-TSP re-engineering for multi and many-core Architectures

This Section shows the *code reengineering* process applied to the ACO-TSP algorithm. The code obtained as a result of these optimizations is adapted to both, Intel Xeon and Xeon Phi processors. Firstly, some *scalar optimizations* are made to enhance the performance of each parallel task. This is particularly important on the Intel Xeon Phi KNC architecture, as its cores are quite simple compared to the cores available on the Intel multi-core architecture. For the ACO-TSP, we replaced *pow()* by *powf()*), avoiding *static and runtime type conversion*, and also avoiding repetitive computations using results previously calculated for the numerator of Equation (1).

3.1 Thread Parallelism

The tour construction stage is inherently parallel as ants run in parallel building their own solutions. Thus, we map ants to threads using the OpenMP *pragma* `#pragma omp parallel for` (see Algorithm 1). The computation of the numerator of Equation (1), performed before the ants build their solutions, is also parallelized using the same *pragma*.

3.2 Vectorization

Vectorization is particularly well-suited to fully leverage the current high performance processors, which are equipped with width vector units. There are several ways to develop vectorized codes, beyond the use of low level instructions, such as assembly language or *intrinsics*, that may introduce portability issues between different architectures. All of them rely on the compiler's capabilities to automatically vectorize some loops (enabled with the `-O2` optimization level or higher). However, programmers have to facilitate the compiler's task by rearranging and adding some hints into the source code.

Algorithm 1 Parallel tour construction

```
1: #pragma omp parallel for
2: for $a = 1$ to $m$ do
3:     {Place ant on initial city}
4:     $initial\_city \leftarrow choose\_initial\_city()$
5:     $tour[a][1] \leftarrow initial\_city$
6:     $visited[a][initial\_city] \leftarrow$ true
7:     {Construct tour}
8:     for $step = 2$ to $n$ do
9:         $current\_city = tour[a][step-1]$
10:        $tour[a][step] = choose\_next(a, current\_city, thread\_id)$
11:    end for
12:    $tour[a][n+1] \leftarrow tour[a][1]$
13:    $tour\_length[a] \leftarrow compute\_tour\_length(tour[a])$
14: end for
```

Within the tour construction stage, over 99% of the time is spent on the selection function (*choose_next()* in the Algorithm 1). Thus, we focus our efforts on vectorizing this procedure taking the following actions to ease the vectorization process[3].

– *Data alignment*: Use *_mm_malloc(size, 64)* instead of *malloc()* for data alignment on the heap (to a multiple of 64 bytes).
– *Align padding*: Pad the inner dimension of multi-dimensional arrays to guarantee alignment for each row of the matrix. This is required to avoid misalignment between rows, as the memory of a structure is allocated using a single call to *_mm_malloc()*.
– *Data alignment hints*: Give the compiler data alignment hints to prevent it from implementing runtime checks for alignment. Concretely, we use `__assume_aligned(ptr, 64)` for pointers. These clues are provided in the region of the code where the data structure is used within a loop.
– *Pointer disambiguation*: Use `#pragma ivdep` before a loop for telling the compiler to ignore vector dependencies, avoiding loop multiversioning.
– *Data structure changes*: Precisely, as ants are mapped to threads, and each thread (simulating an ant) generates n random numbers in a vectorized way, the seed for generating random numbers needs to be replicated to a matrix of *seeds*, having a row for each thread and as many columns as the number of cities (n).

Algorithm 2 shows our vectorized implementation of I-Roulette (see Section 2.1). This is actually the selection procedure used in this work. The *choice_info* matrix stores the probabilities of choosing each city without taking into account whether the cities are visited or not. This latter information is stored in the *visited* matrix. A position (a, i) in this matrix has the value 0 if the ant a has already visited city i, or the value 1, otherwise. In this way, the weights associated with already visited cities have the value 0. From the 17.0 Intel compiler version, the loop (lines 3-9) is fully vectorized by the compiler. Note the use of the `pragma ivdep` before the loop for pointer disambiguation.

[3] Notice that we have used the Intel C++ compiler in this work.

Algorithm 2 Vectorized I-Roulette

Input: Ant identifier (a), current city ($current\_city$), thread identifier ($thread\_id$).
Output: Selected city.

```
1: $city \leftarrow -1$
2: $max\_weight \leftarrow -1$
   #pragma ivdep
3: for $i = 1$ to $n$ do
4:     $w \leftarrow choice\_info[current\_city][i] * visited[a][i] * random01(seeds[thread\_id][i])$
5:     if $w > max\_weight$ then
6:         $city \leftarrow i$
7:         $max\_weight \leftarrow w$
8:     end if
9: end for
10: return $city$
```

3.3 Distributed-Memory Parallelism

This Section shows the parallelization strategy for the ACO-TSP in multiple nodes to enhance its computation. There are several ways for distributing the work among the computing nodes (e.g., an island-model which divides the colony into smaller subcolonies, divide the number of ants, etc.). Our approach is basically to replicate the parallel ACO implementation developed for a single node to all the computing nodes. An MPI process is created in each node for each copy and then, the thread and vector parallelism is enabled in each node. Every copy has the same meta-parameters, the same number of ants, and they face the same problem (with the total number of cities), but each node have its own copy of the pheromone matrix. In this way, each copy of the ACO implementation can explore the entire search space.

The number of iterations of each ACO-TSP instance is obtained by dividing the total number of iterations among all the computing nodes[4]. In this way, a lineal speedup close to the number of distributed nodes could be achieved, considering negligible the time for the replication and distribution of the copies. However, the downside of this approach is that the solution quality could be degraded in each ACO-TSP replicated solution. This is due to the total number of iterations is split among the nodes, and the actual number of iterations on each node might not be enough for ensuring the convergence to a solution.

Our proposal overcomes this problem by letting the different ACO-TSP instances exchange and combine their pheromone matrices among them at a certain frequency. This is what we call synchronization point. Then, each copy runs a given number of iterations ($num\_it\_bt\_synch$) between each synchronization point, being the number of synchronizations (num_synchs) a design parameter of the distributed implementation. There are some possibilities to implement the synchronization mechanism. In this paper, we have used a cen-

[4] If the total number of iterations is not a multiple of the number of nodes, the number of iterations per node is increased by one unit to ensure the addition of the iterations carried out by all the nodes is greater than or equals to the total number of iterations.

tralized one, that is, a master copy is in charged to receive, combine and send each pheromone matrix, but distributed mechanisms are also possible. Note that the number of synchronizations is a relevant parameter that affects both the speedup and the solution quality of the distributed implementation.

The synchronization mechanism comprises three steps:

1. The master ACO-TSP instance receives the pheromone matrix from other instances.
2. The master instance generates a new pheromone matrix as a result of the other pheromone matrices received from the other instances. There are several alternatives for combining the matrices; i.e. mean, maximum, minimum, etc. In particular, we compute the resulting matrix as the mean of the individual pheromone matrices.
3. The master instance sends the new pheromone matrix to the other instances. Each instance uses this updated matrix as its pheromone matrix for the following iterations.

Let's give an example to make clearer our proposal. Let's assume that the parallel ACO-TSP, running in a single node, uses 100 ants (the original colony) and 1000 iterations to converge to a optimal solution. Then, following our approach, we replicate four times (assuming there are 4 nodes in the cluster) the original colony (100 ants in each node but with its own pheromone matrix), and each node performs 250 iterations. With only one synchronization performed at the end of the execution, we would obtain the best speedup (almost 4 in this example), but the worst solution quality. Setting the number of synchronizations to 5, each copy will run 50 iterations in isolation and then a synchronization point will be reach to exchange and update the pheromone matrix. In this case, the speedup obtained will be worse than before, but the quality of solution found would be better. The objective is to choose a value for the number of synchronizations (synchronization frequency) that gives a good speedup for the distributed implementation while improving the quality of the solution found over the single-node implementation.

Algorithm 3 shows the structure for the distributed ACO metaheuristic.

Algorithm 3 Distributed ACO metaheuristic

```
1: $Initialization()$
2: **for** $s = 1$ **to** $num\_synchs$ **do**
3:     **for** $i = 1$ **to** $num\_it\_bt\_synch$ **do**
4:         $TourConstruction()$
5:         $PheromoneUpdate()$
6:     **end for**
7:     $SynchronizePheromoneMatrix()$
8: **end for**
9: $GatherSolutions()$
```

First, in addition to allocating and initializing the data structures (distance matrix, pheromone matrix, ant colony, etc.), the ACO-TSP master instance sends the required parameters (number of iterations, number of cities, number of ants, α, β, ρ, initial amount of pheromone on all edges of the first pheromone matrix, number of synchronizations, etc.) and the structures (distance matrix) for executing the different iterations. Next, blocks of *num_it_bt_synch* iterations (consisting of building tours and updating pheromone) are run within the nested loop. Note that, we have exploited thread parallelism and vectorization in the single node implementation. At the end of each block of iterations, pheromone matrices are synchronized as described previously. At a final step, the master instance receives the best solution (quality of the solution and its associated tour) from each of the other processes, and computes the best of the received solutions.

4 Evaluation

This section answers three different questions: a) How good are the performance results (runtime and acceleration) of our parallel implementation for each of the architectures? b) How much is the parallel efficiency in each of the architectures? and, c) How good is the distributed implementation running on multiple nodes? In the following sub-sections we give a detailed explanation of these issues.

4.1 Test Bed

Hardware Platform: The evaluation platform is equipped with an Ivy Bridge EP Intel Xeon E5-2650 v2 CPU (16 cores), a Broadwell Intel Xeon E5-2698 v4 CPU (40 cores), three Intel Xeon Phi 7120P Knights Corner coprocessors (61 cores) and an Intel Xeon Phi 7250 Knights Landing processor (68 cores), whose main features are shown in Table 1. In our results, we refer to the Intel Xeon E5-2650 v2 chip as Xeon v2, to the Intel Xeon E5-2698 v4 as Xeon v4, to the Intel Xeon Phi 7120P as Xeon Phi KNC and to the Intel Xeon Phi 7250 as Xeon Phi KNL. Note that, Xeon Phi KNL has an additional MCDRAM-based on-package high bandwidth memory.

Table 1 Hardware features.

	Xeon v2	Xeon v4	Xeon Phi KNC	Xeon Phi KNL
Sockets	2	2	1	1
Clock Frequency	2.6 GHz	2.2 GHz	1.238 GHz	1.4 GHz
Cores/socket	8 out-of-order	20 out-of-order	61 in-order	68 out-of-order
Threads/core	2	2	4	4
VPU Width	256 bits	256 bits	512 bits	512 bits
Peak Performance	665.6 GFLOPs SP	1408 GFLOPs SP	2020 GFLOPs SP	3046.4 GFLOPs SP
Peak Memory Bandwidth	59.7 GB/s	76.8 GB/s	352 GB/s	76.8 GB/s
L1d-cache size/core	32 KB	32 KB	32 KB	32 KB
L2-cache size/core	256 KB	256 KB	512 KB	512 KB
L2-cache size (total)	4 MB	10 MB	30.5 MB	34 MB
L3-cache	20 MB	50 MB	—	—
MCDRAM size	—	—	—	16 GB
MCDRAM Peak Bandwidth	—	—	—	400 GB/s

The system runs on a Linux CentOS 7.2 with kernel 3.10.0, and Intel MPSS 3.7.2. Codes are built using Intel C++ compiler (version 17.0.6 for Xeon Phi KNC, and 18.0.1 for the other platforms) with the optimization level `-O3`. In addition, the compiler option `-mmic` is used when the code is compiled for Xeon Phi KNC.

Software Application and Experimental Methodology: Our baseline implementation is based on Stützle's implementation [14] of ACO. Particularly, we use the Ant System algorithm with I-Roulette as selection procedure (see Section 2.1). We do not use nearest neighbor lists nor local search. Regarding the random number generator, the function proposed by Stützle [14] is included in our code.

The implementations are tested using a set of instances from the TSPLIB benchmark library [21]: *lin318*, *rat783*, *pr1002*, *rl1889*, *pr2392*, *fl3795*, *rl5934* and *pla7397*[5]. We set ACO parameters as recommended in [2]; i.e. $m = n$, where m is the number of ants and n is the number of cities [6], $\alpha = 1$, $\beta = 5$ and $\rho = 0.5$.

The performance evaluation is based on single-precision numbers. Each experiment is repeated 10 times, and the values shown are the averages over these 10 independent runs, where each run is composed of 100 iterations. The standard deviation is not provided, as it is negligible. The number of threads per core is empirically selected to get the highest performance on each architecture: 2 on Intel Xeon v2 and v4, 4 on Xeon Phi KNC, and 3 on Xeon Phi KNL. As for the CPU affinity scheduling parameter, evaluations performed on Intel Xeon Phi KNC and KNL are carried out with *balanced* affinity, while on *compact* is set on Intel Xeon v2 and v4, as they provide the best results in terms of performance.

The Intel Xeon Phi KNC is used in native mode, that is, the application is run completely on Xeon Phi KNC as an independent node. On Xeon Phi KNL, the MCDRAM is set to flat mode and execute the application in two configurations: 1) on DDR4 (without using the MCDRAM) and 2) on MCDRAM as the main memory (this memory is exposed as an independent NUMA node, so we use *numactl* for placing the memory of the application on this node). Moreover, the clustering mode of Intel Xeon Phi KNL is set to *SNC-4*, a sub-NUMA cluster mode that partitions the chip into four quadrants exposing these quadrants as NUMA nodes (see Section 2.2).

4.2 Single Node Evaluation

Our first experiment evaluates the execution time for the tour construction stage on each architecture when the optimizations detailed in Sections 3.1

[5] The name of the instance includes an acronym of the problem and the number of cities, that is, the problem size.

[6] Although setting m=n might be a good choice for a sequential implementation, for a parallel implementation other choices could be better. This is proposed as future work.

and 3.2 are applied. Table 2 shows the wall-clock time of this stage for a single iteration averaged over 100 iterations. For Intel Xeon Phi KNL, the application is executed on MCDRAM.

Table 2 Execution time (miliseconds) for the tour construction stage on different architectures.

Instance	Wall-clock time (ms)			
	Xeon v2 (16 cores)	Xeon v4 (40 cores)	Phi KNC (61 cores)	Phi KNL (68 cores)
lin318	2	1	2	1
rat783	26	8	18	11
pr1002	50	15	30	20
rl1889	336	101	170	186
pr2392	680	200	330	330
fl3795	4,230	850	1,640	1,220
rl5934	20,800	9,210	6,900	3,660
pla7397	40,900	20,230	16,050	7,100

Table 2 shows that Intel Xeon v4 outperforms the other architectures when it runs instances of up to 3795 cities. However, both KNC and KNL Intel Xeon Phi outperform Intel Xeon v4, achieving a speedup factor between 1.3x and 2.5x respectively for the largest instances (i.e. 5934 and 7397 cities). Finally, the Intel Xeon Phi KNL obtains the best execution time with a speedup factor of up to 5.7x against Xeon v2 (for the largest problem size). Note that, Intel Xeon processors (v2 and v4) have high-end cores, but Intel Xeon Phi processors have more but simpler cores, each of them doubling Xeon's VPU width (256 bits), so both, Intel Xeon Phi (KNC and KNL) are benefit from vectorization strategies. Then, we conclude here that our implementation shows a good behaviour for all the different architectures, even if they have a different number of cores.

4.3 Parallel Efficiency

This section analyzes the parallel efficiency for each architecture. Parallel efficiency is defined as the ratio of speedup (sequential execution time divided by parallel execution time) to the number of cores. Parallel efficiency is an interesting measurement on every architecture as it gives the point of diminishing returns, i.e. the optimal number of cores to use in the parallel implementation for a fixed problem size.

Figures 1 and 2 show the parallel efficiency obtained on each architecture[7]. The affinity has been set to `compact` in all the architectures to fully leverage all the cores.

Figure 1 shows that Xeon v2 obtains good parallel efficiency for small and medium-sized problem instances (around 80%), but it decreases for larger

[7] We have omitted the curve for lin318 to ease the visualization of every plot.

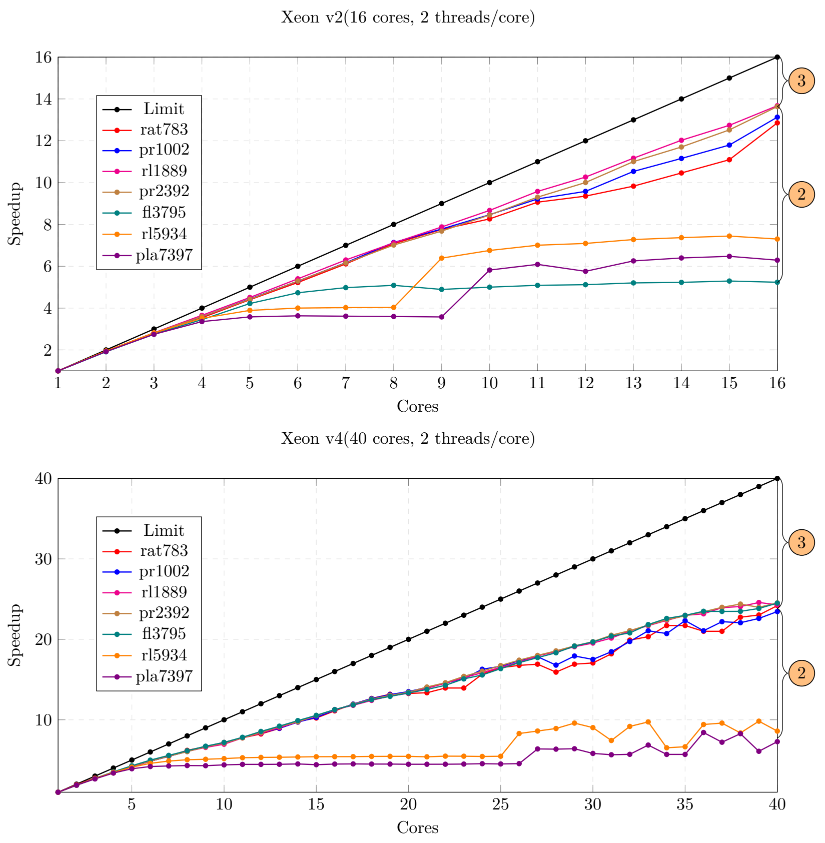

Fig. 1 Parallel efficiency of tour construction on Xeon multicore architectures.

problem sizes (around 40%). The Xeon v4 shows worse scalability, ranging from 62% to only 20% for large problem sizes. Moreover, there are flat regions in both plots (for large problem sizes), meaning that from a certain number of cores, the speed (or runtime) is not improved regardless of the number of cores actually used by the architecture, so the decreasing return point has been reached.

Figure 2 shows the Xeon Phi KNC achieves the best parallel efficiency, ranging from near 100% for small problems to 70% for larger problem sizes, although it drops to 33% for the largest size. As for Xeon Phi KNL with DDR4 memory, the parallel efficiency ratio ranges from 41% to 10%, while using high bandwidth memory (MCDRAM) helps to obtain a better ratio, varying between 44% to 26%. Again, there are some flat regions in these plots, especially for large-sized problem instances. We have identified three problems that could explain these results: 1) core load unbalance, 2) memory bandwidth limitations and 3) effects of NUMA on data placement. In the following sections, we show other experiments and give some ways or clues to mitigate these causes and improve efficiency in parallel.

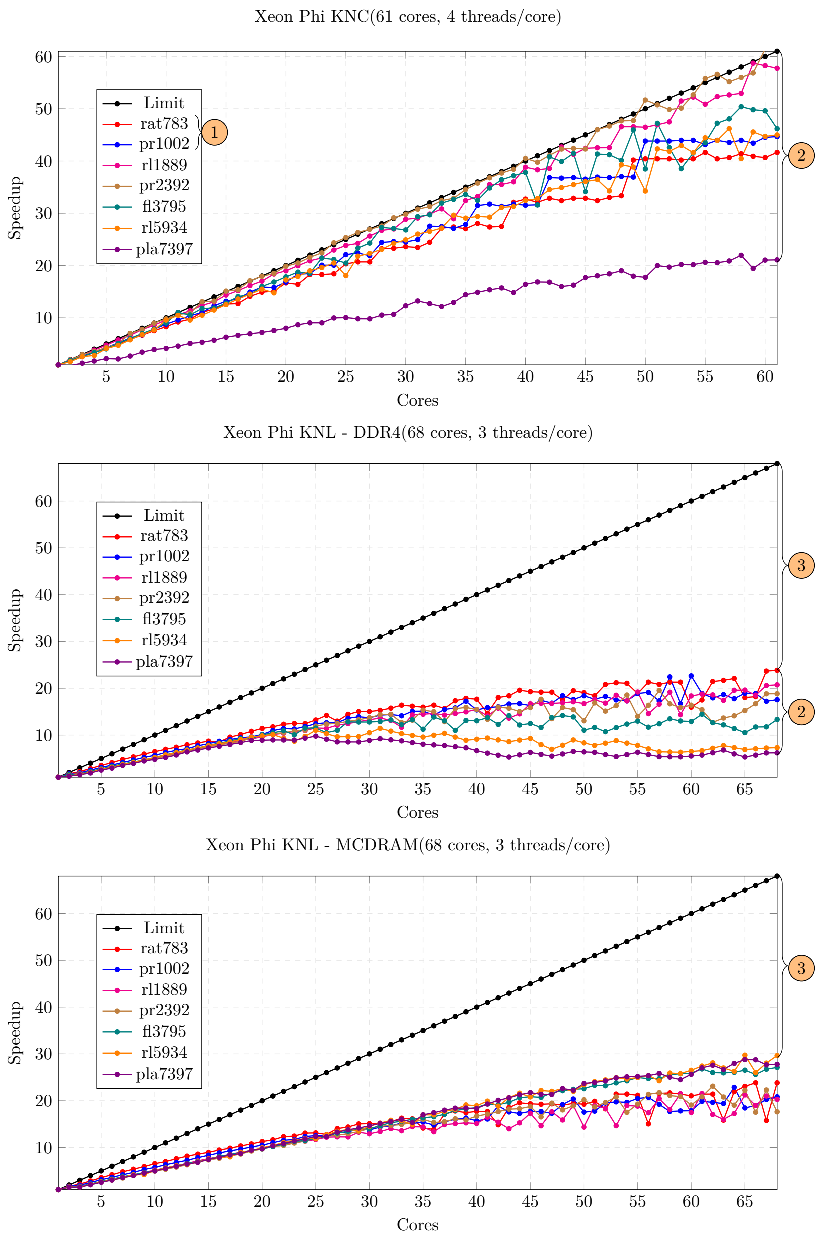

Fig. 2 Parallel efficiency of tour construction on Xeon Phi KNC and KNL architectures.

4.3.1 Core load imbalance

Uneven load among cores affects the parallel efficiency of our implementation. Core load imbalance is directly proportional to the number of cores (number of processing elements), and inversely proportional to the amount of work (problem size), assuming that the work is evenly distributed among the processing elements. Therefore, load imbalance has a lower impact on Xeon v2 and v4 than on both Xeon Phi KNC and KNL. Besides, for all architectures, the core load imbalance has less impact in larger instances, as the ratio between the maximum load of a thread and the average load of all threads is lower. Figure 2 shows some examples of this issue for the Xeon Phi KNC architecture. For the problem instances lin318, rat783 and pr1002 (indicated with the number 1 in this Figure), there are flat regions in which the speedup remains constant as the number of cores increases, and then suddenly rises to the theoretical limit before a new flat region begins.

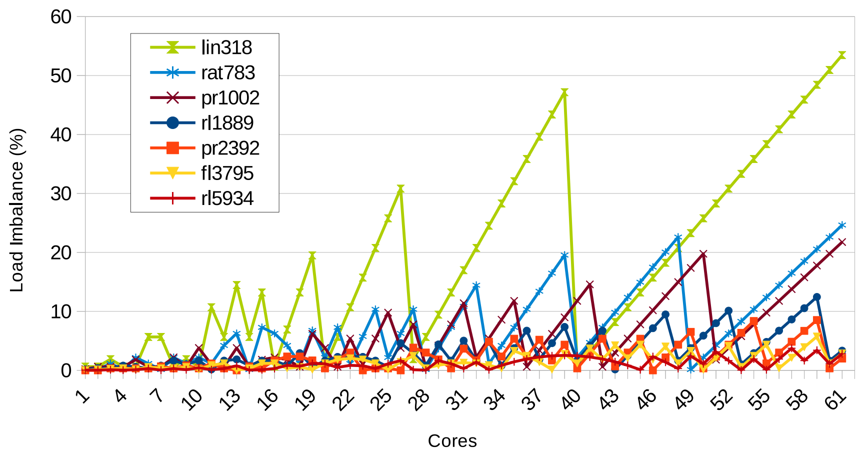

Fig. 3 Load imbalance on Xeon Phi KNC.

To confirm our assumptions, the Figure 3 shows the percentage of load imbalance vs. the number of cores for different TSP instances on the Intel Xeon Phi KNC architecture. The load imbalance is greater than or equal to 20% for smaller cases, but it falls below 10% for larger problems (more than 2,392 cities), regardless of the number of running cores.

We try to reduce the effects of core load imbalance by using the `dynamic` and `guided` scheduling policies included in OpenMP. However, these scheduling policies did not improve performance as the ants (i.e. threads) take the same amount of time to complete. Therefore, load imbalance is a factor to be taken into account for the ACO-TSP parallel implementation, as it could damage the parallel efficiency depending on the actual number of cores of the architecture and the size of the chosen problem.

4.3.2 Memory bandwidth limitations

Memory bandwidth affects parallel efficiency, especially in memory-bounded algorithms. For the ACO-TSP implementation, memory bandwidth is an issue to increase the parallel efficiency especially for larger instances. This problem affects our CPU context (multi- and many-core architectures) more deeply than in GPUs due to their different memory bandwidth capabilities. Therefore, we have carefully evaluated this issue.

Firstly, the static memory usage of the application was evaluated depending on the problem size and the number of threads[8] (see Table 3). Then, the table 4 shows the place of the memory hierarchy in which each instance fits according to their static memory usage.

Table 3 Static memory usage (MiB) depending on the problem size (n) and the total number of threads: 32 for Xeon v2, 80 for Xeon v4, 244 for Xeon Phi KNC and 272 for Xeon Phi KNL.

		Number of threads			
Instance	**n**	32	80	244	272
lin318	318	2.36	2.42	2.61	2.65
rat783	783	14.14	14.28	14.77	14.85
pr1002	1002	23.11	23.30	23.92	24.03
rl1889	1889	81.92	82.27	83.45	83.65
pr2392	2392	131.28	131.72	133.21	133.47
fl3795	3795	330.14	330.84	333.21	333.62
rl5934	5934	806.74	807.83	811.54	812.17
pla7397	7397	1257.49	1258.37	1261.67	1262.38

These tables shows that the parallel efficiency is worse in those instances that do not fit in the caches of each architecture. On Xeon v2, the parallel efficiency starts decreasing from 3795 cities onwards; on Xeon v4, from 5934 (see Figure 1); on Xeon Phi KNC, from 5934; and on Xeon Phi KNL (DDR4), from 3795 (see Figure 2). This issue is shown in these figures with the number 2.

Table 4 Place of the memory hierarchy in which each instance fits.

Memory	Xeon v2	Xeon v4	Phi KNC	Phi KNL
L2	lin318	lin318	lin318 - pr1002	lin318 - pr1002
L3	rat783	rat783 - pr1002	—	—
Main Memory	pr1002 - pla7397	rl1889 - pla7397	rl1889 - pla7397	rl1889 - pla7397

Larger instances that do not fit in the caches are continuously accessing the main memory, and the memory bandwidth of each architecture is not able to cope with the dynamic memory bandwidth the application requires.

[8] Although the actual number of threads has a little effect, it slightly affects the size of several memory structures.

Previous statement is confirmed for the Xeon Phi KNC, which has a high bandwidth memory (GDDR-5 with 352 GB/s) and it is the least affected by this issue. Again, this is confirmed when the application was run on MCDRAM of the Xeon Phi KNL. Comparing the parallel efficiency charts for Xeon Phi KNL on DDR4 and MCDRAM[9], the bandwidth problems are quite alleviate when the application is run on the KNL's MCDRAM. It seems that when the application requires a higher bandwidth (because of a greater problem size) than the one indicated by the architecture, the performance decreases. Therefore, memory bandwidth is another important factor for the ACO-TSP parallel implementation as it could damage the parallel efficiency depending on the actual problem size. To cope with this issue, other versions of the ACO algorithm could be used with smaller memory footprints.

4.3.3 NUMA effects

The NUMA effects are the third problem that prevents parallel efficiency from reaching the theoretical linear limit. In NUMA architectures, the location of data plays a fundamental role, since the memory access time depends on the memory location in relation to the. Under NUMA, a core accesses its own local memory (memory attached to its socket) faster than non-local memory (local memory to another socket(s) on the chip). With the exception of the Xeon Phi KNC architecture (it has a single socket), all other test-bed platforms (Xeon v2, Xeon v4 and Xeon Phi KNL) are NUMA architectures. The NUMA effects appear on architectures with more than one socket, as Xeon v2 and v4 (with two sockets), and Xeon Phi KNL (four sockets). The Xeon Phi KNL is an advanced architecture that can be configured either as non-NUMA or as different NUMA modes. As already mentioned, the experiments are performed using the Xeon Phi KNL as SNC-4 mode (four NUMA sockets).

In our ACO-TSP parallel implementation, we identified that the cause of the performance loss is because each thread writes a portion of the `choice_info` matrix when it calculates the selection probabilities in parallel, and then all threads access this matrix in parallel for reading, which implies many accesses to other NUMA nodes that may have the dirty copies. Two main features of NUMA effects are observed: (1) They occur equally with the problem instances that fit either into the caches or the main memory, and b) they are greater with the actual number of cores in the architecture. We have tagged Figures 1 and 2 with the number 3 to indicate the effects of this problem. Therefore, NUMA effects is the last important factor affecting parallel efficiency for the ACO-TSP parallel implementation. To address this problem, the application requires fine-tuning and an exhaustive analysis of the memory traffic and data placement.

[9] These memories have quite different bandwidths (see Table 1), being MCDRAM the one with the highest bandwidth (400 GB/s).

4.4 Multiple Node Evaluation

Our second experimental scenario refers to the distributed implementation presented in Section 3.3. This experiment uses 1000 iterations and three Intel Xeon Phi KNC (in native mode) as (homogeneous) nodes. The first part of the experiment consists of the speedup obtained by the ACO-TSP code when it is executed on 3 nodes compared to 1-node counterpart version. For 3 nodes, only a final final synchronization is performed, that is, 1000 iterations are divided among the three nodes (334 iterations per node) and, the master node eventually gathers the best global solution. The table 5 shows the execution time and the speedup factor, varying the number of threads per core (2, 3 and 4 threads per core correspond to 122, 183 and 244 threads per process, respectively). The table 5 shows the speedup factors achieved are between 2.78 and 2.96x, i.e. a parallel efficiency is in the range of 92 and 98%. Thus, our distributed implementation scales well along with the number of nodes, showing good results for all the combinations of threads per core, being slightly better the ones obtained for 2 threads per core.

Table 5 Execution time (s) and speedup for distributed ACO on 1 and 3 nodes. Only one synchronization is carried out at the end of the computation

Instance	Threads per core								
	2			3			4		
	Time (s)		Speedup	Time (s)		Speedup	Time (s)		Speedup
	1 node	3 nodes		1 node	3 nodes		1 node	3 nodes	
lin318	7.87	2.66	2.96	7.52	2.63	2.86	7.92	2.74	2.89
pr1002	123.34	42.66	2.89	111.82	42.71	2.62	110.20	38.58	2.86
pr2392	957.25	331.52	2.89	811.35	279.99	2.90	759.34	272.74	2.78

As mentioned in the Section 3.3, there is a trade-off between obtaining good performance and scalability in the distributed implementation, and the quality of the solution found. Although a detailed study of the quality of solution is left for future work, it is clear that the solution quality is improved by increasing the number of synchronizations done. Next, we analyze the relation between the number of synchronizations and the speedup factor achieved by the distributed implementation. This analysis attempts to show the margin for increasing the number of synchronizations without losing the performance benefit of running the code on multiple nodes.

Figure 4 shows the speedup factor obtained by varying the number of synchronizations for different problem sizes. As previously mentioned, the number of iterations is fixed at 1000, which are equally distributed into the three nodes, meaning each node performs 334 iterations. Therefore, the experimental results shown in Figure 4 are obtained ranging from 1 to 300 synchronizations. Note that it makes no sense to carry out more synchronizations than iterations per process.

As expected, the speedup factor (and, consequently, the efficiency) decreases as the number of synchronizations increases. However, if the problem size (number of cities) increases, the reduction of the speedup is less significant.

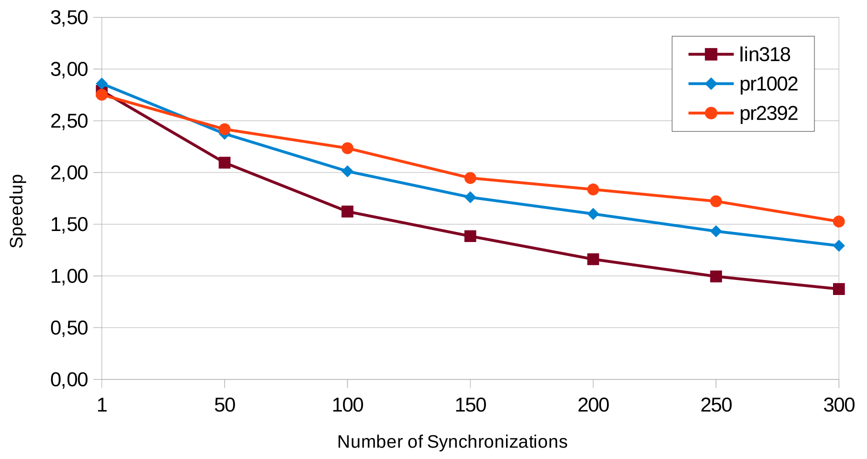

Fig. 4 Speedup factor for the distributed ACO on three nodes when the number of synchronizations is varied.

This is because the tour construction stage is $O(n^3)$, while a synchronization is $O(p \cdot n^2)$, where n is the problem size and p is the number of processes or nodes. Let's us remind the reader that $m = n$ ants choose $n - 1$ cities, being each of this selections $O(n)$. Besides, a synchronization consists of 3 steps. In the first and third steps, $p - 1$ processes (or nodes) send and receive n^2 data (the pheromone matrix) each, so they are $O(p \cdot n^2)$. However, as in our case p is set to 3, we have $O(n^2)$ for the communications. In step 2, the computation of the mean of the pheromone matrices is also $O(p \cdot n^2)$.

Moreover, the tour construction is performed at each iteration, while a synchronization is performed only every *num_it_bt_synch* iterations. The larger the problem size, the higher the percentage of time spent on tour construction and the lower the percentage spent on synchronization. Note that for the smallest instance (318 cities), there is only performance loss (compared to the code running on a node) when the number of synchronizations increases above 250. For larger instances, we still benefit from running the code on multiple nodes, even when the communication takes place in almost every iteration.

5 Related Work

The Ant Colony Optimization is computationally expensive and, therefore, it has been object of different parallelization studies. After noticing that data parallelism and vectorization are essential for obtaining high performance for this metaheuristic on GPUs [4, 22, 5] and Intel Xeon Phi [8], respectively, several strategies and implementations were proposed following this approach. Firstly, two main alternative selection functions (for the tour construction) were introduced for exploiting parallelism on GPUs: I-Roulette (*Independent*

Roulette) [4] and DS-Roulette (*Double-Spin Roulette*) [5], both of them obtaining good performance against the CPU sequential counterpart versions. A data-parallel version of Roulette Wheel selection was also proposed [4], but still preserving an important sequential portion. Recently, Cecilia et al. [7] have shown how to parallelize this latter sequential part using two parallel patterns (*prefix-scan* and *stencil*), achieving a speedup factor of up to 6x compared to I-Roulette. Finally, Skinderowicz [23] presents a parallel implementation for the Ant Colony System algorithm, as an alternative to the ACO and the MAX-MIN Ant System.

Regarding ACO on CMP Architectures, several parallel and vectorized implementations of Roulette Wheel, I-Roulette and DS-Roulette have been proposed targeting Intel multi-core processors and the first generation of the Intel Xeon Phi (KNC) coprocessor [9,8,10,24]. Tirado et al. [9] presented a variant of Roulette Wheel, called UV-Roulette (*Unique random Value Roulette*), which generates the same random number for all the ants. Zhou et al. [24] introduced a new selection approach named Vector-based Roulette Wheel to properly exploit SIMD units. Montesinos and García [10] showed how to completely parallelize and vectorize I-Roulette for Intel architectures using strip-mining. Recently, Peake at. al [25] proposed a vectorized candidate set selection for ACO Systems which use candidate sets. [24] has already noted that bandwidth limitation affects the scalability (and performance) of the ACO implementation.

Concerning distributed ACO, Stützle [26] introduced the first and most simple strategy, in which parallel independent runs of the algorithm were executed, and the best solution of the runs was taken as the final solution. This parallelization scheme, with no communication overhead, was followed in [27, 6] targeting GPU-based clusters. Approaches based on the island-model, in which the nodes (colonies) exchange information after every certain number of iterations, have also been proposed [28,29]. Other work [30] divides the problem into subcomponents (subgraphs), assigning each of them to a different process. In order to build a complete solution, ants are migrated from one process to another. Finally, Llanes et al. [6] presented an MPI implementation of ACO for GPU-based heterogeneous clusters. They dynamically estimated the execution time of the slowest GPU and, based on this measurement, the other GPUs performed a deeper search (running additional iterations) or reduce its power consumption (by decreasing the clock rate).

6 Conclusions and Future Work

Ant Colony Optimization (ACO) and other bioinspired metaheuristics emerged as a novel solution to provide approximate solutions in a reduced time framework for many real-world problems. When accelerators came to the HPC world, the research community started to migrate the ACO algorithm to NVIDIA GPUs using CUDA. Only recently there have appeared some works that migrate ACO to Intel Xeon Phi. This paper gives detailed guidelines of the reengineering process of migrating ACO-TSP to Intel architectures, discussing how to optimize ACO for the intra-node (thread and vector) parallelization. Our single-node implementation is portable and works perfectly in four different Intel architectures. It also scales well with the number of cores, ranging from an Intel Xeon E5-2650 v2 (16 cores) multi-core to an Intel Xeon Phi 7250 KNL (68 cores) many-core processor, obtaining an speedup factor of up to 5.7X between these two computing architectures. Moreover, the paper discusses an ACO-TSP distributed implementation, using the MPI library. We have tested it over three Intel Xeon Phi 7120P KNC processors obtaining a speedup factor of up to 2.96x compared to its 1-node counterpart version (with only one final synchronization). Finally, the paper studies the parallel efficiency of the ACO-TSP implementation in both a single and multiple nodes. We found that core load imbalance, memory bandwidth limitations, and NUMA effects on data placement are the key players in the parallel efficiency in a single node, while frequency among synchronizations is the main issue in the multiple node environment.

As the new Xeon scalable architectures are NUMA, our goal is to study how our implementation is affected by different memory placement policies. We also intend to implement other Ant Systems versions with less memory footprints to study the memory bandwidth limitation. The other line we are interested in is for distributed implementation the synchronization frequency, taking into account not only the speedup factor but also the quality of the solution. In addition, we are developing and comparing other distributed implementations, allowing the number of ants to be different from the number of cities, aimed at a heterogeneous environment.

Acknowledgments

This work was partially supported by the Fundación Séneca, Agencia de Ciencia y Tecnología de la Región de Murcia under Project 20813/PI/18, and by Spanish Ministry of Science, Innovation and Universities as well as European Commission FEDER funds under grants TIN2015-66972-C5-3-R, RTI2018-098156-B-C53, TIN2016-78799-P (AEI/FEDER, UE) and RTC-2017-6389-5. We acknowledge the excellent work done by Victor Montesinos while he was doing a research internship supported by the University of Murcia.

References

1. Yang, X. S.: Nature-inspired metaheuristic algorithms. Luniver press, 2010.
2. Dorigo, M., Stützle, T.: Ant Colony Optimization. A Bradford Book, The MIT Press, USA, 2004.
3. Crainic, T.G. and Toulouse, M.: Parallel Strategies for Meta-heuristics. State-of-the-Art Handbook in Metaheuristics, Kluwer Academic Publishers, pp. 475-513, 2003.
4. 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) (2013) 42-51.

5. Dawson, L., Stewart, I.: Improving Ant Colony Optimization performance on the GPU using CUDA. IEEE Conf. on Evolutionary Computation, pp. 1901-1908, 2013.
6. Llanes, A., Cecilia, J.M., Sánchez, A., García, J. M., Amos, M., Ujaldón, M.: Dynamic load balancing on heterogeneous clusters for parallel ant colony optimization. Cluster Computing, 19 (1) (2016) 1-11.
7. Cecilia, J.M., Llanes, A., Abellán, J. L., Gómez-Luna, J., Chang, L., Hwu, W. W.: High-throughput Ant Colony Optimization on graphics processing units. J. Parallel Distrib. Comput., 113 (2018) 261-274.
8. Lloyd, H., Amos, M.: A Highly Parallelized and Vectorized Implementation of Max-Min Ant System on Intel Xeon Phi. IEEE Computational Intelligence, 2016.
9. Tirado, F., Barrientos, R. J., González, P., Mora, M.: Efficient exploitation of the Xeon Phi architecture for the Ant Colony Optimization (ACO) metaheuristic. The Journal of Supercomputing, pp. 1-18, 2017.
10. Montesinos, V., García, J. M.: Vectorization Strategies for Ant Colony Optimization on Intel Architectures. Parallel Computing is Everywhere, IOS Press, pp. 400-409, 2018.
11. Lawler, E., Lenstra, J., Kan, A., Shmoys, D.: The Traveling Salesman Problem. Wiley, New York, 1987.
12. Lloyd, H and Amos, M.: Analysis of Independent Roulette Selection in Parallel Ant Colony Optimization. Genetic and Evolutionary Computation Conference, ACM, pp. 19-26, 2017.
13. Montesinos, V.: Performance Analysis of Ant Colony Optimization on Intel Architectures. Master's Thesis, University of Murcia (Spain), June 2018.
14. Stützle, T.: ACOTSP v1.03. Last accessed 2018-02-15. [Online]. URL: iridia.ulb.ac.be/∼mdorigo/ACO/downloads/ACOTSP-1.03.tgz
15. Dorigo, M.: Optimization, Learning and Natural Algorithms. PhD Thesis, Politecnico di Milano, Italy, 1992.
16. Duran, A., Klemm, M.: The Intel Many Integrated Core Architecture. Int. Conf. on High Performance Computing and Simulation (HPCS), pp. 365-366, 2012.
17. The OpenMP API specification for parallel programming. [Online]. URL:`https://www.openmp.org` [Last accessed: 14 June 2018].
18. The Message Passing Interface (MPI) standard. [Online]. URL:`http://www.mcs.anl.gov/research/projects/mpi/` [Last accessed: 15 June 2018].
19. Intel Developer Zone. [Online]. URL: `https://software.intel.com/en-us/modern-code` [Last accessed 2018-10-02].
20. Pearce, M. What is Code Modernization? Intel Developer Zone. [Online]. URL: `http://software.intel.com/en-us/articles/what-is-code-modernization` [Last accessed 2018-02-15].
21. Reinelt, G.: TSPLIB - A traveling salesman problem library. ORSA Journal on Computing, 3 (1991) 376-384.
22. Delévacq, A., Delisle, P., Gravel, M., and Krajecki, M.: Parallel Ant Colony Optimization on Graphics Processing Units. J. Parallel Distrib. Comput., 73 (1) (2013) 52-61.
23. Skinderowicz, R. The GPU-based parallel ant colony system J. Parallel Distrib. Comput., 98 (2016) 48-60.
24. Zhou, Y., He, F., Hou, N., and Qiu, Y.: Parallel ant colony optimization on multi-core SIMD CPUs. Future Generation Computer Systems, 79 (2018) 473-487.
25. Peake, J., Amos, M., Yiapanis, P., and Lloyd, H.: Vectorized Candidate Set Selection for Parallel Ant Colony Optimization. Genetic and Evolutionary Computation Conference, ACM, pp. 1300-1306, 2018.
26. Stützle, T.: Parallelization Strategies for Ant Colony Optimization. Parallel Problem Solving from Nature (PPSN V), pp. 722-731, Springer, Berlin, 1998.
27. Abdelkafi, O., Lepagnot, J., and Idoumghar, L.: Multi-level Parallelization for Hybrid ACO. International Conference on Swarm Intelligence Based Optimization, 2014.
28. Michel, R. and Middendorf, M.: An island model based ant system with lookahead for the shortest supersequence problem. Parallel Problem Solving from Nature (PPSN V), pp. 692-701, Springer, Berlin, 1998.
29. Chen, L., Sun, H. and Wang, S.: Parallel implementation of ant colony optimization on MPP. International Conference on Machine Learning and Cybernetics, 2008.
30. Lin, Y., Cai, H., Xiao, J, Zhang, J.: Pseudo Parallel Ant Colony Optimization for Continuous Functions. International Conference on Natural Computation, 2007.

31. Vladimirov, A. and Asai, R.: Clustering modes in Knights Landing Processors: Developer's Guide. Colfax International, May 2016. [Online]. URL: `https://colfaxresearch.com/knl-numa/` [Last accessed: 16 June 2018].
32. M. Akila ; P. Anusha ; M. Sindhu ; Krishnasamy T. Selvan: Examination of PSO, GA-PSO and ACO algorithms for the design optimization of printed antennas. IEEE Applied Electromagnetics Conference (AEMC), 2017.