J Supercomput (2017) 73:5053–5070
DOI 10.1007/s11227-017-2124-5

Efficient exploitation of the Xeon Phi architecture for the Ant Colony Optimization (ACO) metaheuristic

Felipe Tirado[1] · **Ricardo J. Barrientos**[1] · **Paulo González**[1] · **Marco Mora**[1]

Published online: 16 August 2017

Abstract In recent years, the use of compute-intensive coprocessors has been widely studied in the field of Parallel Computing to accelerate sequential processes through a Graphic Processing Unit (GPU). Intel has recently released a GPU-type coprocessor, the Intel Xeon Phi. It is composed up to 72 cores connected by a bidirectional ring network with a Vector Process Unit (VPU) on large vector registers. In this work, we present novel parallel algorithms of the well-known Ant Colony Optimization (ACO) on the recent many-core platform Intel Xeon Phi coprocessor. ACO is a popular metaheuristic algorithm applied to a wide range of NP-hard problems. To show the efficiency of our approaches, we test our algorithms solving the Traveling Salesman Problem. Our results confirm the potential of our proposed algorithms which led to distinct improvements of performance over previous state-of-the-art approaches in GPU. We implement and compare a set of algorithms to deal with the different steps of ACO. The matrices calculation in the proposed algorithms efficiently exploit the VPU and cache in Xeon Phi. We also show a novel implementation of the roulette wheel selection algorithm, named as UV-Roulette (*u*nique random *v*alue roulette). We compare our results in Xeon Phi to state-of-the-art GPU methods, achieving higher

✉ Ricardo J. Barrientos
ricardo.j.barrientos@gmail.com; rbarrientos@ucm.cl

Felipe Tirado
ftirado@ucm.cl

Paulo González
pgonzalez@ucm.cl

Marco Mora
marcomoracofre@gmail.com

[1] Department of Computer Science, Universidad Católica del Maule, Talca, Chile

performance with large size problems. We also exposed the difficulties and key hardware performance factors to deal with the ACO algorithm on a Xeon Phi coprocessor.

Keywords ACO · Metaheuristic · Xeon Phi · Parallel computing · Coprocessors

1 Introduction

The Ant Colony Optimization (ACO) [1] algorithm is based on the behavior of real ants. It is a population-based search algorithm applied to a wide scope of problems [2,3], such as NP-hard problems.

We deal in this work with a variant of ACO named Ant System (AS) [4], which consists of two stages: *tour construction* and *pheromone update*. These two stages are iteratively applied until a predefined number of iterations is achieved. The first stage is used to decide which is the next city to be visited by an ant. The second stage is used to communicate ants updating the pheromone value of each edge between cities.

In recent years, the use of coprocessors has been applied to the field of Parallel Computing to accelerate compute-intensive sequential algorithms. Up to now, the NVIDIA GPU (Graphics Processing Units) [5,6] has been the most popular coprocessor. It is implemented on a *many-core architecture* (massive multi-core architecture), i.e., it consists of a huge quantity of cores compared to a conventional multi-core architecture.

Recently, Intel has released a coprocessor called Xeon Phi ([7]), based on the Intel MIC architecture [8], which consists of 61–72 cores, achieving a performance of up to 1208 GFLOP/s (billions of floating-point operations per second) with double-precision floating point.

In the present paper, we show novel parallel algorithms of all the stages of the Ant Colony Optimization (ACO) problem using an Intel Xeon Phi coprocessor. To show the efficiency of our approaches, we test our algorithms solving the *Travelling Salesman Problem* (TSP) [9], where the objective is to find the shortest path around a set of cities. We also compare our results to previous state-of-the-art works in GPU, achieving a better performance in the solution for TSP with a high number of cities. To the best of our knowledge, this publication is an early work using a Xeon Phi coprocessor with the ACO algorithm.

The organization of the paper is as follows. Section 2 gives details about the ACO algorithm, some background on the Xeon Phi architecture, and also information about related work. In Sect. 3 we describe our algorithms using a Xeon Phi to solve the ACO algorithm. Section 4 shows the experimental results between our proposals, and also comparison measures against previous GPU-based state-of-the-art algorithms. Finally, Sect. 5 shows the main conclusions of the present work.

2 Background knowledge and related work

In the following sections, we give a description of the ACO algorithm (Sect. 2.1). We also describe the Xeon Phi coprocessor, its architecture (Sect. 2.3), and related work (Sect. 2.4).

2.1 Ant Colony Optimization

Ant Colony Optimization (ACO) is a metaheuristic for combinatorial optimization problems, formalized by Dorigo and Caro [10] and Dorigo et al. [11]. A metaheuristic is a set of algorithmic concepts that can be used to define heuristic methods applicable to a wide set of different problems. This means that a metaheuristic is a general-purpose algorithmic framework that can be applied to different optimization problems with relatively few modifications.

ACO takes inspiration from the foraging behavior of some ant species. These ants deposit pheromone on the ground in order to mark some favorable path that should be followed by other members of the colony.

The Ant System (AS) [4,12] is the first ACO algorithm proposed in the literature. Its main characteristic is that, at each iteration, the pheromone values are updated by all the m ants that have built a solution in the iteration itself. The pheromone $\tau_{i,j}$, associated with the edge joining cities i and j, is updated as follows:

$$\tau_{i,j} \leftarrow (1-\rho)\tau_{i,j} + \sum_{k=1}^{m} \Delta\tau_{i,j}^{k}, \tag{1}$$

where ρ is the evaporation rate, m is the number of ants, and $\Delta\tau_{i,j}^{k}$ is the quantity of pheromone deposited on edge (i, j) by ant k:

$$\Delta\tau_{i,j}^{k} = \begin{cases} Q/L^{k}, & \text{if ant } k \text{ used edge } (i, j) \text{ in its tour,} \\ 0, & \text{otherwise,} \end{cases} \tag{2}$$

where Q is a constant, and L_k is the length of the tour constructed by ant k.

In the construction of a solution, ants select the following city to be visited through a stochastic mechanism. When ant k is in city i and has so far constructed the partial solution s^p, the probability of going to city j is given by:

$$p_{i,j}^{k} = \begin{cases} \frac{[\tau_{i,j}]^{\alpha}[\eta_{i,j}]^{\beta}}{\sum_{l \in N_i^k}[\tau_{i,l}]^{\alpha}[\eta_{i,l}]^{\beta}}, & \text{if } c_{i,j} \in N(s^p), \\ 0, & \text{otherwise,} \end{cases} \tag{3}$$

where $N(s^p)$ is the set of feasible components; that is, edges (i, l) where l is a city not yet visited by the ant k. The parameters α and β control the relative importance of the pheromone versus the heuristic information $\eta_{i,j}$, which is given by:

$$\eta_{i,j} = \frac{1}{d_{i,j}} \tag{4}$$

where $d_{i,j}$ is the distance between cities i and j.

2.2 ACO for Traveling Salesman Problem (TSP)

The Traveling Salesman Problem (TSP) [9] aims to find the shortest path between a set of cities, visiting each city just once. The latter can be represented as a complete graph G of n nodes, where each edge $e_{i,j}$ represents the distance between cities i and j. TSP is a NP-hard problem; therefore, we aim to achieve as good a solution as possible within a practicable time.

ACO in TSP consists of two main stages: *tour construction* and *pheromone update*. Initially, the ants are randomly placed in different cities. In each step of the tour construction stage, the ants apply the random proportional rule to choose which city to visit next.

Dorigo and Stützle in [1] recommend the values of $\alpha = 1$ and $2 \leq \beta \leq 5$ for the Eq. 3, and also one ant for each city. In the experimental results, we adopted these parameter values. The probability $p^k_{i,j}$ (in Eq. 3) favors edges with smaller distances. The sequential ACO algorithm for TSP is shown in Algorithm 1.

Algorithm 1 Overview of the sequential ACO algorithm for the TSP.

```
1: data_initialization();
2: while (!convergence()) do
3:     tour_construction();
4:     pheromone_update();
5: end while
```

To avoid visiting a city more than once, each ant k manages a matrix called `tabu_list` with the cities already visited. A matrix called `choice_info` (see [1]) is used to store the numerator of Eq. 3. A value `choice_info`[$current\_city$][j] of a city j which ant k has not yet visited is associated with a part on a roulette wheel which is proportional to the choice weight. Then, the wheel is spun to choose which city to visit next.

After each ant builds its tour, the pheromone levels of the edges must be updated. The pheromone level of the edges are first evaporated according to the user-defined evaporation rate ρ (Eq. 1). We used $\rho = 0.5$ as recommended by Dorigo and Stützle [1]. Then, each pheromone level $\tau_{i,j}$ is updated, but over time the edges which are few selected will be discarded by evaporation. Then, each ant k deposits an amount of pheromone on its tour τ^k and that pheromone level $\tau_{i,j}$ becomes the Eq. 1.

2.3 Intel Xeon Phi coprocessor

Currently, there are two generations of the Intel Xeon Phi coprocessor [8,13], which are KNC (Intel Knights Corner) and KNL (Intel Knights Landing), where the former is composed from 57 to 61 cores, and the latter from 64 to 72 cores. The cores are connected by a high performance on-die bidirectionalinterconnect. The

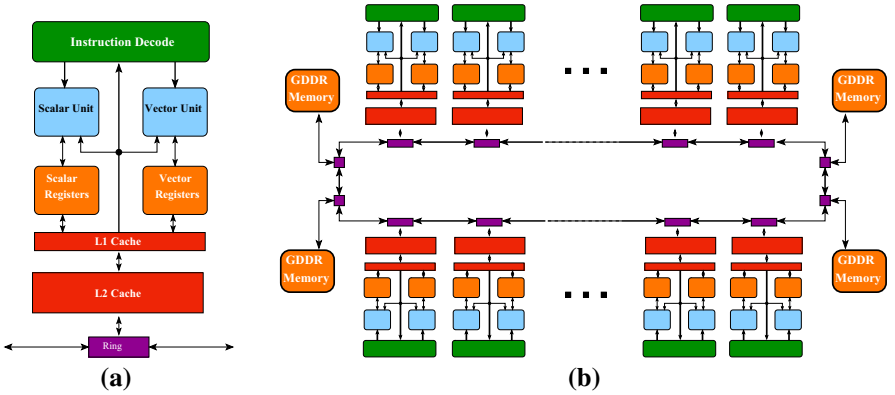

Fig. 1 Diagram of MIC architecture. **a** Intel Xeon Phi coprocessor core and **b** interconnection of the Xeon Phi coprocessor cores

coprocessor runs a Linux operating system and supports all main Intel development tools like C/C++, Fortran, MPI and OpenMP. The coprocessor is connected to an Intel Xeon processor (the host) via the PCI Express (PICe) bus. The most important properties of the MIC architecture are shown in Fig. 1a. They are:

Core The processor core (scalar unit) is an in-order architecture (based on the Intel Pentium processor family), which fetches and decodes instructions from four hardware threads. New vector instructions provided by the Intel Xeon Phi coprocessor instruction set uses a dedicated 512-bit wide vector floating-point unit (VPU) provided for each core. A core executes two instructions per cycle, one on the U-pipe and the other on the V-pipe.

Vector Processing Unit (VPU) The VPU includes the EMU (Extended Math Unit), and it executes 16 single-precision floating point, 16 32-bit integer operations or 8 double-precision floating point operations per cycle. Each operation can be a fused multiply-add giving 32 single-precision or 16 double-precision floating-point operations per cycle.

L1 Cache It has a 32 KB L1 instruction cache and 32 KB L1 data cache. It is 8-way associativity, with a cache line-size of 64 byte. It has a load-to-use latency of 1 cycle, which means that an integer value loaded from the cache can be used in the next clock by an integer instruction.

L2 Cache Each core contributes 512 KB of L2 to the total global shared L2 cache storage. If no cores share any data or code, then the effective total L2 size of the chip is up to 31 MB.

The Fig. 1b shows an overview diagram of the architecture, where it is shown how cores are tied together via fully coherent caches into a bidirectional ring of 115 GB/s. This figure also shows a GDDR5 memory with 16 memory channels, which reaches up to 5.5 GB/s.

2.4 Related works

To the best of our knowledge, recent related works on ACO using a GPU are [14–16]. Due to the recent release of the Intel Xeon Phi, the only related work for ACO using this coprocessor is [17,18]. We describe these works below.

[14] maps ants to CUDA Blocks so as to better utilize the GPU architecture. The latter avoids task-based parallelism, because the ants are not assigned to threads, but to thread blocks exploiting data-based parallelism. They calculate the probability of visiting a city without branching the warp by using a selection method known as I-Roulette defined in the same work. I-Roulette achieves 2.36× compared to the classic roulette wheel. They also provide an approach to the pheromone update stage on GPU, achieving up to 21× when both stages are performed in GPU.

Dawson and Stewart [15] also concludes that a task-based approach is more suitable for GPU because it takes advantage of the GPU memory hierarchy. The authors implement the MMA algorithm to solve TSP using CUDA-based GPU approaches. After a thread block constructs a new tour, 3 edges are removed from the tour rearranging them in such a way that the resulting tour has a shorter length than the initial tour. The latter must be done with data stored in the device memory, decreasing the performance. The authors show a speed-up up to 20× against sequential implementation.

Delevacq et al. [16] shows a solution for TSP using ACO in GPU. The authors implement parallel approaches of both stages of ACO, and a roulette wheel selection method based on I-Roulette called DS-Roulette. The results show a 82× speed-up over the sequential counterpart.

The approaches based on a Xeon Phi coprocessor are [17,18]. The former shows a very simple approach of ACO for TSP using a Xeon Phi, and giving very few details about the implementation, it achieves a speed-up up to 6.2× with its Xeon Phi approach over the sequential counterpart. The latter is a preliminary study of this work, where we showed an initial version of the *Probability MIC v2* algorithm comparing it with the sequential counterpart.

3 Tour construction in Xeon Phi

This section shows our proposed algorithms for ACO using an Intel Xeon Phi coprocessor. We focus on the main bottleneck of the ACO algorithm which is the tour construction stage (around 85% of the execution time).

In Sect. 3.1 we describe our approach to calculate the `choice_info` matrix considering Xeon Phi vectorization restrictions. In Sect. 3.2 we describe our methods to calculate the probability matrix which uses the `choice_info` matrix of the previous section. In Sect. 3.3 we compare a set of different algorithms for the roulette wheel selection procedure.

3.1 Calculating the `choice_info` matrix

As described in Sect. 2.1, the `choice_info` matrix stores the numerator of Eq. 3. It is a matrix of size $n \times n$, where n is the number of cities. We implemented two meth-

ods to obtain the `choice_info` matrix, named as *choice_info v1* and *choice_info v2*.

`choice_info`, we need to access to the i-th rows of the matrices τ and η. To try to increase cache hits, for both methods we distribute (in a circular manner) the rows of τ and η among the cores (line 7, Algorithm 2). In the experiments we always execute 4 threads per core, which is the maximum supported by the Xeon Phi. Because of this, 4 threads executed in the same core access to a different row of τ and η, accessing the elements of the row in a circular manner again (line 8), and writing in `choice_info` in the same manner (line 10).

Algorithm 2 Algorithm for both versions of *choice_info* calculation.

```
choice_info(float **τ, float **η)
1: {Let ID_thread be the current thread ID and num_cores the total number of cores}
2: {Let TxCore be the number of threads per core}
3: {Let num_cities be the number of cities}
4:
5: #pragma offload
6: #pragma omp parallel
7:     for(i = ID_thread/TxCore; i < num_cities; i += num_cores)
8:         for(j = ID_thread%TxCore; j < num_cities; j += TxCore)
9:             /* In choice_info v2 the pow function is changed for consecutive multiplications*/
10:            choice_info[i][j] = pow(τ_ij, α) * pow(η_ij, β);
```

Both algorithms, *choice_info v1* and *choice_info v2*, are similar, but the difference between them is that the *choice_info v2* avoids using the `pow()` function. Instead of `pow()`, *choice_info v2* performs consecutive multiplications, trying to take advantage of the multiplication operation, which consumes just one clock cycle. This also allows us to observe the impact of these kinds of optimizations on the Xeon Phi in a similar way that [14] applied it in GPU.

3.2 Calculating the probability matrix

To obtain the probability matrix, the `choice_info` matrix (calculated in the previous section) is required. The probability matrix is used to find the probability of each ant visiting a city, where the probability is defined by Eq. 3. The i-th ant stores its probability in the i-th row of this matrix, and its probability of visiting the j-th city is stored in the j-th column. This means that the cell (i, j) stores the probability of the ant i visiting the city j.

We implemented two different approaches to calculate the probability matrix named as *Probability MIC v1* and *Probability MIC v2*, which are described below.

3.2.1 Probability MIC v1

This method (illustrated by Fig. 2a) distributes the ants among the cores (lines 8 and 16, Algorithm 3), which means that a set of four threads, executed in the same core, calculate the probability of an ant visiting each city (line 21). These threads access in a circular manner (line 12) to the elements of the `choice_info` matrix, calculating

the denominator of the Eq. 3 (line 13). This denominator is used after a barrier (line 21) for the probability calculation (line 21). This matrix probabilities is also calculated accessing in a circular manner to the elements of `choice_info` (line 20). This algorithm was implemented trying to maximize the hits cache.

Algorithm 3 Algorithm for *Probability MIC v1*.

```
Probability_MIC_v1(float **choice_info)

1: {Let ID_thread be the current thread ID and num_threads the total number of threads}
2: {Let num_cities be the number of cities}
3: {Let num_ants be the number of ants}
4:
5: #pragma offload
6: #pragma omp parallel
7: {
8:       for(i = ID_thread/4; i < num_ants; i += (num_threads/4))
9:       {
10:            #pragma vector aligned
11:            #pragma ivdep
12:            for(j = ID_thread%4; j < num_cities; j += (num_threads/4))
13:                 denominator[i] += choice_info[i][j];
14:      }
15:      #pragma omp barrier
16:      for(i = ID_thread/4; i < num_ants; i += (num_threads/4))
17:      {
18:            #pragma vector aligned
19:            #pragma ivdep
20:            for(j = ID_thread%4; j < num_cities; j += (num_threads/4))
21:                 probability_matrix[i][j] = choice_info[i][j]/denominator[i];
22:      }
23: }
```

3.2.2 Probability MIC v2

This version (illustrated by Fig. 2b) calculates the probability matrix mapping each ant to a different thread. Instead *Probability MIC v1*, trying to exploit cache access, this method tries to exploit the Vector Process Unit (VPU) in the Xeon Phi. This is because each thread is accessing to 16 consecutive float numbers, which is the length of register (512-bit) of the VPU.

This method is shown in the Algorithm 4, where each thread calculates the probability of a different ant (line 7, Algorithm 4). Each ant visits all the cities (line 15), using different directives to indicate the alignment of the data to be accessed. In this algorithm, a barrier is not required, because each ant is able to calculate its own denominator (of Eq. 3) in line 12.

3.3 Roulette wheel selection in Xeon Phi

We proposed and compared different methods of dealing with the *roulette wheel selection* procedure in Xeon Phi. We compared our methods to the state-of-the-art *I-Roulette* [14], which is a recent roulette wheel selection method for ACO in GPU. We describe them below.

Algorithm 4 Algorithm for *Probability MIC v2*.

```
Probability_MIC_v2(float **choice_info)
```

```
1: {Let ID_thread be the current thread ID and num_threads the total number of threads}
2: {Let num_cities be the number of cities}
3: {Let num_ants be the number of ants}
4:
5: #pragma offload
6: #pragma omp parallel
7:     for(i = ID_thread; i < num_ants; i += num_threads)
8:     {
9:         #pragma vector aligned
10:        #pragma ivdep
11:        for(j = ID_thread; j < num_cities; j++)
12:            denominator[i] += choice_info[i][j];
13:        #pragma vector aligned
14:        #pragma ivdep
15:        for(j = ID_thread; j < num_cities; j++)
16:            probability_matrix[i][j] = choice_info[i][j]/denominator[i];
17:    }
```

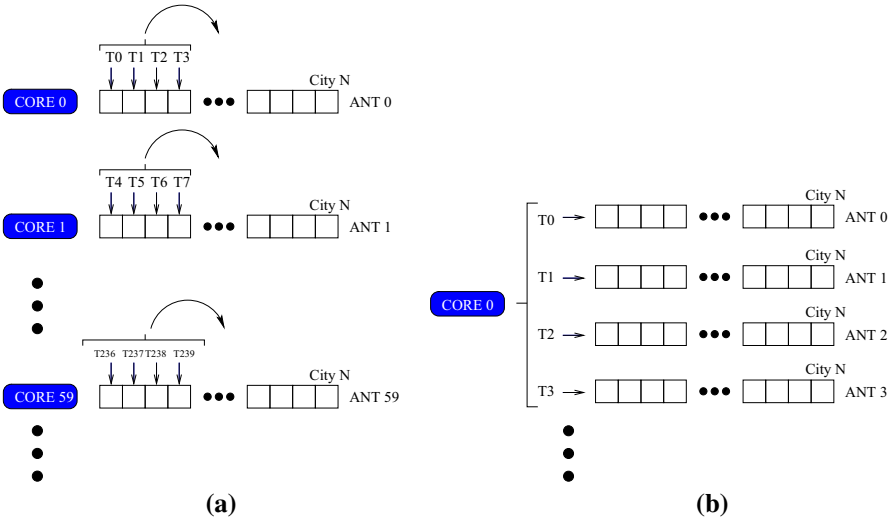

Fig. 2 Illustration of MIC v1 and MIC v2 methods accessing the probability matrix. **a** Probability MIC v1 method and **b** probability MIC v2 method

3.3.1 I-Roulette

We adapted the *I-Roulette* [14] method (originally implemented for GPU) to Xeon Phi. This method creates three vectors per ant of length equal to the number of cities. The first is composed by one random value per city. The second vector has the probability of visiting each city. The third is the *tabu list*, which indicates the previously visited cities. Each coordinate of the three vectors is multiplied, and the results are stored in a fourth vector which is reduced to obtain the highest of them.

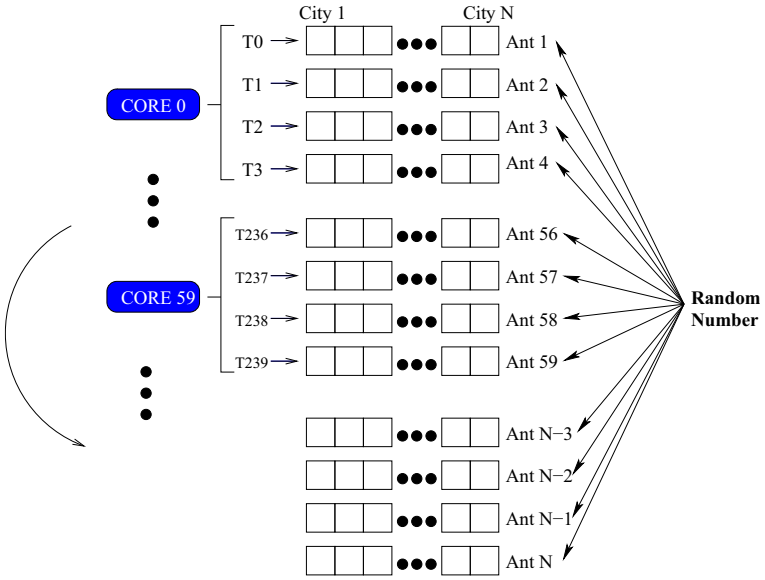

Fig. 3 Illustration of UV-Roulette method

In Xeon Phi we distribute (in a circular manner) the ants among the threads, and each thread multiplies its own three vectors. This distribution allows the VPU to be used in the Xeon Phi.

3.3.2 Sequential-extension

In the sequential method, we generate one random value per ant, and this is used in the roulette wheel selection procedure to obtain the city to visit next. We adapted it to the Xeon Phi by distributing the ants among the threads, and each thread generates one random value. Each thread executes the sequential algorithm using its own random value.

3.3.3 UV-Roulette

Based on the previous method and observing the cost of creating a random number in a core of the Xeon Phi, we reuse the same random number for all the ants (see Fig. 3). For this, we used a shared variable with the unique random value (*UV-Roulette*) generated by the first thread (of the first core).

4 Experimental results

The details of the coprocessor and the host machine used in the multi-core algorithms are described in Table 1.

Table 1 Platforms description

(a) Intel Xeon Phi details	
Coprocessor	Intel Xeon Phi 71200P
Cores	61 cores of 1.24 GHz
Memory	16 GB of memory (bandwidth 352 GB/s)
Cache L1	3.7 MB L1 (64 KB L1 per core)
Cache L2	30 MB L2 (512 KB L2 per core)
Compiler	icc version 15.0.2, flags: −03
(b) Platform for sequential and multi-core versions	
Processor	2×Intel Xeon E5-2620v3, 12-cores (in total)
	15 MB Cache, 2.40 GHz, Haswell
Memory	32 GB
Operative System	GNU Debian System Linux
	kernel 3.10.0–123.9.3.el7.×86_64
Compiler	icc version 15.0.2, flags: −03

The data is always firstly mapped to 1-dimensional arrays to transfer them to the coprocessor properly, aligned to 512-bit with the function `posix_memalign()`. We use the offload mode. Also, we use `#pragma vector aligned` and `#pragma ivdep` to ensure the use of the VPU (Vector Processing Unit).

We used as benchmarks complete graphs from the well-known TSPLIB library [19]. We used single-precision floats in all the algorithms. We always used the number of ants equal to the number of cities, and the parameters $\alpha = 1$, $\beta = 2$ and $\rho = 0.5$ which are the value recommendations in [1]. We used 256 iterations, which is a sufficient value to converge to a solution according to previous works ([14,15]).

This paper is focused on the optimization in Xeon Phi of the *tour construction* stage, which is the most costly stage ([1]). For the lighter stage, *pheromone update*, we used the sequential algorithm.

4.1 Experiments on `choice_info` matrix

Our first experiment was the `choice_info` matrix calculation in Xeon Phi, illustrated in Fig. 4a where we show our two approaches. The main difference between the two approaches is the use of the `pow()` function. *choice_info v2* avoids its use extending the multiplication operations. In this figure, we observe how costly this operation is for a Xeon Phi coprocessor, and how important it is to avoid using it.

4.2 Experiments on probability matrix

Figure 4b shows the speed-up of our two methods (*Probability MIC v1* and *Probability MIC v2*) to calculate the probability matrix. The calculation of this matrix is the most costly procedure in the *tour construction* stage of the ACO algorithm. With the

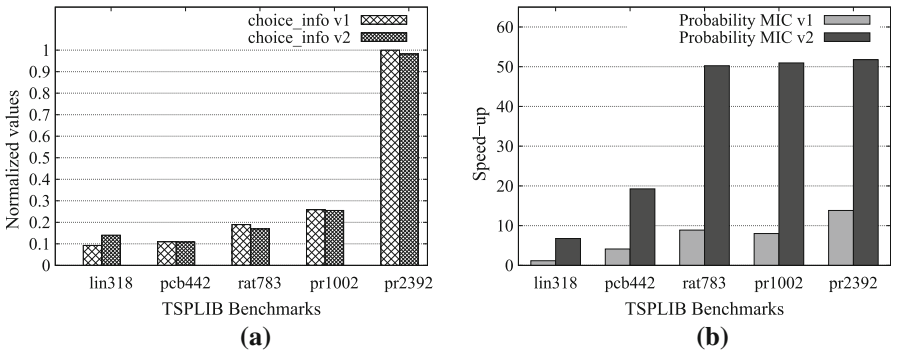

Fig. 4 Speed-up over the sequential counterparts. **a** Normalized running time for `choice_info` matrix calculation in Xeon Phi and **b** probability matrix calculation in Xeon Phi

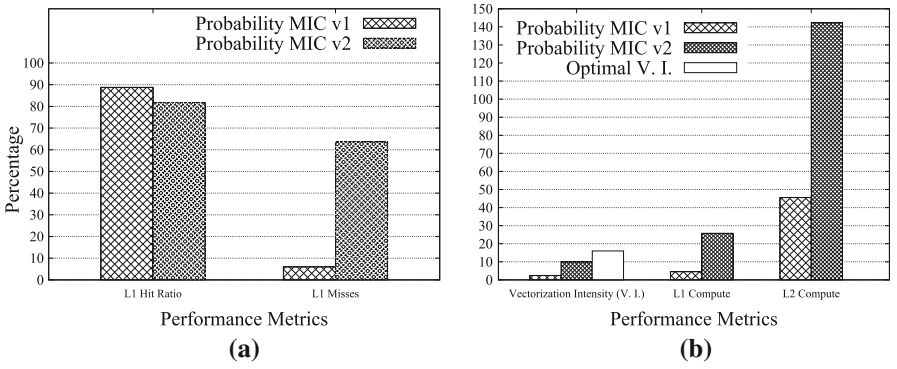

Fig. 5 Measures of different performance metrics on the probability matrix methods. **a** L1 cache measures and **b** measures of the Vector Process Unit (VPU)

Probability MIC v2 method, we achieve more than 51× speed-up over the sequential probability matrix calculation. This performance was achieved because the *Probability MIC v2* method allows the use of vectors aligned to 512-bit, efficiently using the VPU in Xeon Phi.

Figure 5a shows two metrics of the cache usage for both probability matrix methods. *L1 Misses* shows the number of cache misses that occurred in the application. *L1 Hit Ratio* shows the sum of the standard hits and hits to an in-flight prefetch. Hits to an in-flight prefetch occur when the datum was not found in the cache, and it was a match for a cache line already being retrieved for the same cache level by a prefetch. These types of hits have a longer latency than a standard hit, but shorter than a miss. In this figure, we observe that the *Probability MIC v1* takes advantage of the use of L1 cache, but in this case, the efficient use of the VPU (Vector Unit Process) is more relevant as we can see in Fig. 5b.

Figure 5b shows some metrics of the VPU management. The metric *Vectorization Intensity* is a ratio between the total number of data elements processed by vector instructions and the total number of vector instructions. If the ratio approaches 16 (because we used single-precision float numbers), then there is a good chance that

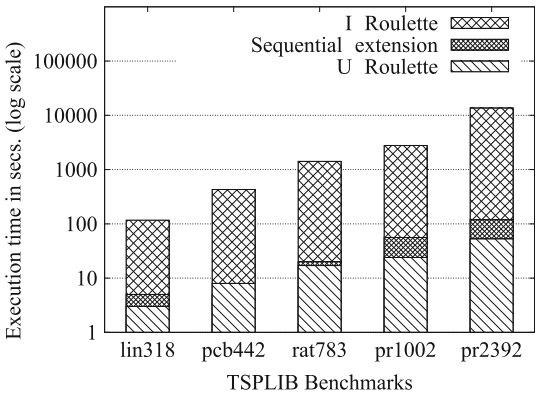

Fig. 6 Roulette wheel selection methods. Speed-up calculated over the sequential algorithm

the loop is well vectorized. *L1 Compute* measures the computational density of an application, or how many computations it is performing on average for each piece of data loaded. *L2 Compute* shows the average number of vectorized operations that occur for each L2 access. The applications that are able to block data for the L1 cache, or reduce data access in general, will have a higher number for *L2 Compute* and the *Probability MIC v2* method takes advantage of this metric and also of the other two. *Probability MIC v2* accesses a row by a different thread, and it was more efficient in filling the VPU registers executing vector instructions.

4.3 Experiments on *roulette wheel selection* in Xeon Phi

Figure 6 shows the three approaches described in Sect. 3.3. The lowest performance is achieved by the Xeon Phi adaptation of *I-Roulette*, showing that although it is suitable for GPU, it is not suitable for the Xeon Phi. For this adaptation, critical regions and synchronization barriers were required (Sect. 3.3.1).

On the other hand, the *sequential-extension* and *UV-Roulette* methods were designed to increase the cache hits of each core in the Xeon Phi. They avoid critical regions and synchronizations barriers. Besides, the *UV-Roulette* method generates an unique random value which is used for all the ants in parallel, decreasing the execution time.

4.4 Experiments with all the stages of the ACO algorithm in Xeon Phi

Figure 7 shows the performance for all the stages of the ACO algorithm, using our methods that achieved the highest performance in the previous sections: *choice_info v2*, *Probability MIC v2* and *UV-Roulette*. Figure 7b shows a speed-up of up to 42× for 2392 cities. The higher the quantity of cities, the higher the speed-up performance, which is because the bad scalability of the sequential algorithm due to its complexity ($O(n^3)$, where n is the number of cities) [20], as we can see in Fig. 7a.

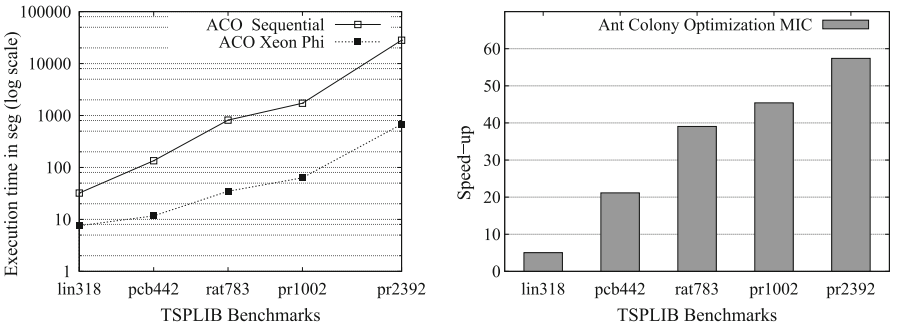

Fig. 7 Speed-up obtained over the sequential counterparts, using *choice_info v2*, *Probability MIC v2* and *UV-Roulette*. **a** Running time of the complete Xeon Phi ACO algorithm and **b** speed-up of the complete Xeon Phi ACO algorithm

Table 2 Specifications of the Xeon Phi and GPUs used in the experiments

	Geforce GTX 580	Xeon Phi 7120p	Tesla C2050
Cores	16 MP, 32 cores per MP, total: 512 cores	61 cores of 1.24 GHz (support 4 threads per core)	14 MP, 32 cores per MP, total: 448 cores
Memory	1536 GB	16 GB	3 GB
Peak single-precision floating-point performance	1.581 Tflops	2.416 Tflops	1.03 Tflops
Peak double-precision floating-point performance	–	1.208 Tflops	515 Gflops
Memory bandwidth	192.4 GB/s	352 GB/s	144 GB/s

4.5 Experimental results between Xeon Phi and state-of-the-art GPU algorithms

In this section we compare our methods on Xeon Phi to the state-of-the-art GPU algorithm presented in [14,15], which are (to the best of our knowledge) the most efficient solution for the ACO algorithm in GPU. Regarding [14], its method with the data-based approach in was used in the experiments, because it was the best alternative shown by the authors.

The measures for the state-of-the-art GPU versions were performed on the same graphics card where they were presented ([14,15]). We used a model of Xeon Phi of the 7000 series for the experiments with our algorithms. The coprocessor details are shown in Table 2.

Figure 8 shows the running time between the GPU-based [14,15] approaches and our Xeon Phi version using the methods *choice_info_v2* (Sect. 3.1), *Probability MIC v2* (Sect. 3.2.2) and *UV-Roulette* (Sect. 3.3.3). Also, Table 3 shows the running time in seconds of our Xeon Phi algorithm using 256 iterations.

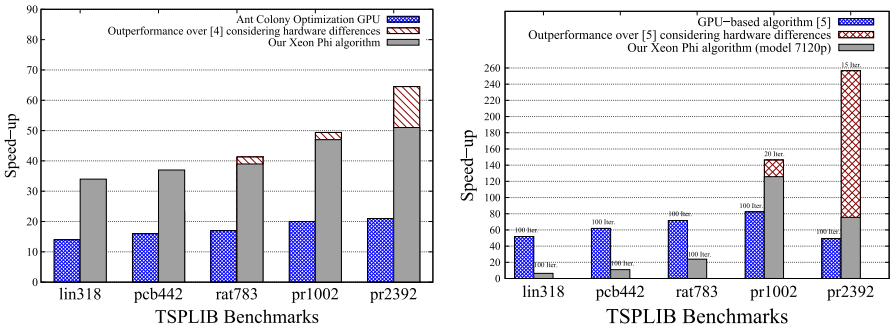

Fig. 8 Comparison between our Xeon Phi algorithm with state-of-the-art GPU algorithms. The outperformance is the speed-up over the expected one according to the GFLOPS difference of the coprocessors (Table 2). **a** Outperformance of our Xeon Phi algorithm over [14], using 256 iterations and **b** outperformance of our Xeon Phi algorithm over [15], using the number of iterations needed to achieve the same tour length

Table 3 Percentage time of the different stages of the ACO algorithm using the highest performance versions of our Xeon Phi algorithms with 256 iterations

Benchmark	Complete ACO algorithm (s)	Percentage time used by choice_info v2 (%)	Percentage time used by UV-Roulette (%)	Percentage time used by Probability MIC v2 (%)
lin318	4.76	5.9	44.6	47.8
pcb442	10.64	3.2	48.1	49.5
rat783	32.92	1.5	49.4	50.4
pr1002	58.12	1.8	44.1	55.1
pr2392	603.91	1.1	16.8	81.9

We expected an advantage of 2.34× and 1.53× for the Xeon Phi 7120p over the GPU Tesla C2050 and GTX 580, respectively, because the specifications (Table 2) for the peak in single-precision floating point. Our Xeon Phi algorithm achieves an improvement from *rat783* onwards over the paper [14] as we can observe in Fig. 8a, where the improvement is for benchmarks with high quantity of cities, form 783 onwards. Regarding the comparison with the paper [15], the improvement of our algorithm is from the benchmark *pr1002* onwards as we can see in Fig. 8b. In this figure, we used the required quantity of iterations to achieve the same quality in the tour length of the solution. That is the reason of using different quantity of iterations in this figure. It is not possible a direct comparison because the lack of information in [15] about the quality of the solution and the required iterations to achieve it. Both GPU-based algorithms achieve a better performance with small quantity of cities because the efficient use of the GPU shared memory. But, this memory is small, and when the quantity of cities grows, these GPU-based algorithms decrease their performance, which is declared by the authors in the corresponding articles.

Table 4 Tour lengths obtained with 100 iterations

Instance	Optimal	CPU	GPU-based [15]	Our Xeon Phi algorithm	Improvement of our Xeon Phi algorithm over [15] (%)
lin318	42,029	46,651	44,495	46,771	−5.1
pcb442	50,778	62,255	56,639	57,597	−1.7
rat783	8806	10,896	9390	10,492	−11.7
pr1002	259,045	333,262	341,080	304,917	10.6
pr2392	378,032	511,977	537,127	450,303	16.2

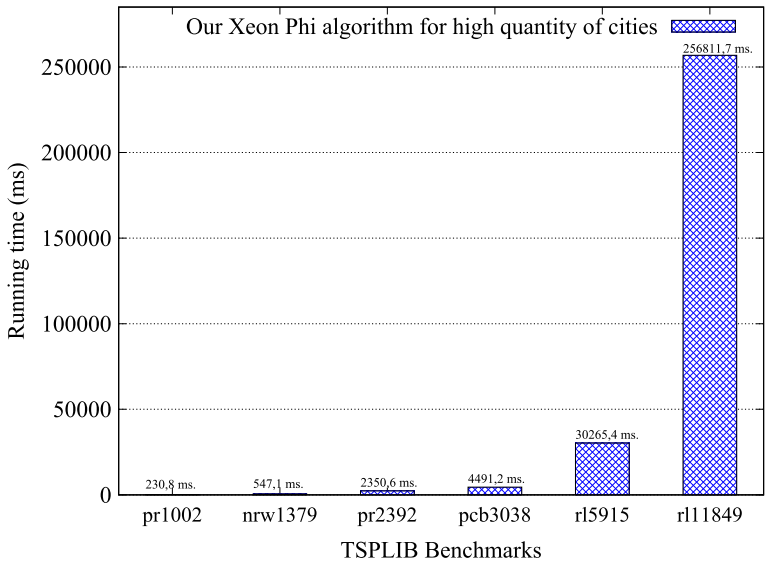

Fig. 9 Our Xeon Phi algorithm using benchmarks with high quantity of cities

The authors in [15] conclude that their GPU-based algorithm decreases in quality when the number of cities is increased. This behavior can be observed in the Table 4 where we achieve an improvement up to 16,2% over the GPU-based algorithm [15].

From the Table 4 and the Fig. 8b, we observe that our Xeon Phi algorithm achieve a better performance in quality solution (tour length) for a single iteration of ACO, over the article [15] (which achieves the highest performance in running time using a GPU). Our algorithm achieve a better quality solution in tour length because the GPU optimized mathematic functions lose precision.

Our Xeon Phi algorithm is able of processing benchmarks with a quantity of cities up to 11,849 (Fig. 9), which is a quantity considerably higher than the state-of-the-art GPU-based approaches are able to process.

5 Conclusions

In this work, we present novel parallel algorithms to implement the Ant Colony Optimization (ACO) using an Intel Xeon Phi coprocessor. The results confirm that our proposed algorithms outperform previous state-of-the-art GPU approaches.

We implemented and compared different methods of the main step *tour construction stage*, which is the most costly stage of the ACO algorithm. More specifically, we showed different methods to deal with the `choice_info` and `probability` matrices calculation. We also showed different *roulette wheel selection* algorithms for ACO in Xeon Phi, where using a unique random value was the most suitable method. In this work, we exposed some key factors in the Xeon Phi architecture to optimize its performance for this metaheuristic.

Our ACO algorithm in Xeon Phi is competitive against the best GPU-based state-of-the-art algorithm from the (TSPLIB) benchmark with 1002 cities onwards. Our Xeon Phi algorithm is able of processing benchmarks with a quantity of cities up to 11,849, which is a quantity considerably higher than the state-of-the-art GPU-based approaches. Thus, we have empirically proved that an efficient exploitation of cache and VPU in Xeon Phi considerably improve the performance in this coprocessor for the ACO problem.

To the best of our knowledge, this is one of the earliest work for ACO in a Xeon Phi coprocessor, which shows that a proper use of its architecture allows a performance similar to GPU approaches.

Acknowledgements This research was supported by the Project of the Universidad Católica del Maule (Chile) "Plan de Desarrollo Anual Facultad de Ingeniería. Convenio de Desempeño". This work was partially supported by the Project FONDEF IDeA en 2 Etapas ID15i10142, "Estimación del Contenido de Aceite en Olivas en base a Tecnologías no Destructivas" (Olive Oil Content Estimation based on non Destructive Technologies), Scientific and Technological Development Support Fund (FONDEF), Government of Chile. Powered@NLHPC: This research was partially supported by the supercomputing infrastructure of the NLHPC (ECM-02).

References

1. Dorigo M, Stützle T (2004) Ant colony optimization. MIT Press, Cambridge
2. Dorigo M, Birattari M, Stützle T (2006) Ant colony optimization. IEEE Comput Intell Mag 1(4):28–39. doi:10.1109/MCI.2006.329691
3. Dorigo M, Blum C (2005) Ant colony optimization theory: a survey. Theor Comput Sci 344(2–3):243–278. doi:10.1016/j.tcs.2005.05.020. http://www.sciencedirect.com/science/article/pii/S0304397505003798
4. Dorigo M (1992) Optimization, learning and natural algorithms. Ph.D. thesis, Politecnico di Milano, Italy
5. NVIDIA GPU Computing. http://www.nvidia.com/object/what-is-gpu-computing.html
6. Hwu W (2012) Programming massively parallel processors, second edition: a hands-on approach. Morgan Kaufmann, Burlington
7. Jeffers J, Reinders J (2013) Intel Xeon Phi coprocessor high performance programming. Elsevier, Philadelphia. ISBN:9780124104143
8. Wang E, Zhang Q, Shen B, Zhang G, Lu X, Wu Q, Wang Y (2014) High-performance computing on the Intel Xeon Phi(TM): how to fully exploit mic architectures. Springer, Berlin
9. Lawler EL, Lenstra JK, Kan AR, Shmoys DB (1985) The traveling salesman problem: a guided tour of combinatorial optimization, vol 3. Wiley, New York

10. Dorigo M, Di Caro G (1999) New ideas in optimization. Chap. The ant colony optimization meta-heuristic. McGraw-Hill Ltd., Maidenhead, pp 11–32
11. Dorigo M, Di Caro G, Gambardella LM (1999) Ant algorithms for discrete optimization. Artif Life 5(2):137–172. doi:10.1162/106454699568728
12. Dorigo M, Maniezzo V, Colorni A (1996) Ant system: optimization by a colony of cooperating agents. Trans Syst Man Cybern Part B 26(1):29–41. doi:10.1109/3477.484436
13. PRACE (2017) (Partnership for advanced computing in Europe). Best Practice Guide Intel Xeon Phi v2.0. http://www.prace-ri.eu/best-practice-guides/
14. Cecilia JM, García JM, Nisbet A, Amos M, Ujaldon M (2013) Enhancing data parallelism for ant colony optimization on GPUs. J Parallel Distrib Comput 73(1):42–51. doi:10.1016/j.jpdc.2012.01.002
15. Dawson L, Stewart IA (2013) Improving ant colony optimization performance on the GPU using CUDA. In: Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2013, Cancun, Mexico, June 20–23 pp. 1901–1908. IEEE (2013). doi:10.1109/CEC.2013.6557791
16. Delevacq A, Delisle P, Gravel M, Krajecki M (2013) Parallel ant colony optimization on graphics processing units. J. Parallel Distrib. Comput. 73(1):52–61. doi:10.1016/j.jpdc.2012.01.003
17. Sato M, Tsutsui S, Fujimoto N, Sato Y, Namiki M (2014) First results of performance comparisons on many-core processors in solving QAP with ACO: kepler GPU versus xeon PHI. In: Genetic and Evolutionary Computation Conference, GECCO '14, Vancouver, BC, Canada, July 12-16, 2014, Companion Material Proceedings, pp. 1477–1478. ACM. doi:10.1145/2598394.2602274. http://dl.acm.org/citation.cfm?id=2598394
18. Tirado F, Urrutia A, Barrientos R.J (2015) Using a coprocessor to solve the ant colony optimization algorithm. In: 34th International Conference of the Chilean Computer Science Society (SCCC), pp. 1–6. doi:10.1109/SCCC.2015.7416584
19. TSPLIB Library. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/
20. Dorigo M, Maniezzo V, Colorni A (1996) Ant system: optimization by a colony of cooperating agents. IEEE Trans Syst Man Cybern Part B 26(1):29–41. doi:10.1109/3477.484436