Vectorized Candidate Set Selection for Parallel Ant Colony Optimization

Joshua Peake
Centre for Advanced Computational Science
Manchester Metropolitan University
Manchester, United Kingdom
J.Peake@mmu.ac.uk

Martyn Amos
Centre for Advanced Computational Science
Manchester Metropolitan University
Manchester, United Kingdom
M.Amos@mmu.ac.uk

Paraskevas Yiapanis
Centre for Advanced Computational Science
Manchester Metropolitan University
Manchester, United Kingdom
P.Yiapanis@mmu.ac.uk

Huw Lloyd
Centre for Advanced Computational Science
Manchester Metropolitan University
Manchester, United Kingdom
Huw.Lloyd@mmu.ac.uk

ABSTRACT

Ant Colony Optimization (ACO) is a well-established nature-inspired heuristic, and parallel versions of the algorithm now exist to take advantage of emerging high-performance computing processors. However, careful attention must be paid to parallel components of such implementations if the full benefit of these platforms is to be obtained. One such component of the ACO algorithm is *next node selection*, which presents unique challenges in a parallel setting. In this paper, we present a new node selection method for ACO, Vectorized Candidate Set Selection (VCSS), which achieves significant speedup over existing selection methods on a test set of Traveling Salesman Problem instances.

ACM Reference Format:
Joshua Peake, Martyn Amos, Paraskevas Yiapanis, and Huw Lloyd. 2018. Vectorized Candidate Set Selection for Parallel Ant Colony Optimization. In *GECCO '18 Companion: Genetic and Evolutionary Computation Conference Companion, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3205651.3208274

1 INTRODUCTION

Ant Colony Optimization (ACO) [10, 13] is a population-based optimization technique inspired by the foraging behavior of ants, and it has been successfully applied in a wide variety of domains [28]. The fundamental principle of the algorithm is that agents representing ants traverse some structure (such as a graph), constructing a solution to the given problem and laying virtual "pheromone trails" as they proceed. The amount of pheromone deposited is proportionate to the "quality" of the solution; as path choices made by individual ants are informed by pheromone concentrations, this leads the population to converge on high-quality solutions [25]. In this paper, we focus on the $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System variant of the algorithm [27], which allows only the "best performing" ant to deposit pheromone (and also restricts the range of pheromone concentrations, in order to prevent stagnation).

Because of the inherently distributed nature of ACO (whereby ants work independently of each other, guided only by a shared global pheromone network), the ACO algorithm presents significant opportunities in terms of its implementation on high-performance parallel hardware [2–4, 6, 8, 14, 21]. In terms of this paper, we are specifically interested in performance improvements that are made possible by the *vector processing* capabilities of chips such as the Intel® Xeon Phi [22, 30], which have instructions that operate on one-dimensional *arrays* of data (vectors), rather than on single data items. In the case of Xeon Phi, these *Single Instruction Multiple Data (SIMD)* instructions operate simultaneously on 16 floating point registers.

A significant bottleneck can arise in ACO-based algorithms when ants are required to select their next "move" [19]. Such algorithms generally (but not exclusively) work on graph-based representations of problems, where ants traverse edges, moving from vertex to vertex (as in the Traveling Salesman Problem (TSP) [11, 26]). Because the number of possible "next moves" can be extremely large, many algorithms use *candidate sets* (or candidate lists) to restrict movement to a select subset of vertices [15], and this has been successfully used in the Ant Colony System variant of ACO [12].

More recently, the use of *nearest neighbour* candidate lists has shown significant promise in solving large instances of the TSP using ACO [4, 7]. This refinement is based on the assumption that good solutions to the TSP avoid large "jumps", and that they can generally be found by making only relatively *local* transitions from vertex to vertex. Although candidate lists are now a standard component of parallel ACO-based algorithms [4, 7], previous implementations of this feature have failed to take advantage of the vector processing capabilities of processors such as the Xeon Phi. In this paper, we show how a modified representation of the nearest neighbour list can fully utilize vector processing, yielding significant performance improvements. Moreover, the speedup obtained increases as the problem size grows, suggesting that our method

GECCO '18 Companion, July 15–19, 2018, Kyoto, Japan

ACM ISBN 978-1-4503-5764-7/18/07…$15.00
https://doi.org/10.1145/3205651.3208274

will be a required component of future ACO-based algorithms for large-scale instances of similar problems.

The rest of the paper is organized as follows: in Section 2 we discuss relevant earlier work, before presenting our algorithm in Section 3. We give and discuss our experimental results in Section 4, before concluding in Section 5 with a brief consideration of possible future work in this area.

2 BACKGROUND AND RELATED WORK

Early work on parallelizing the basic ACO algorithm used multiple independent runs of the sequential algorithm, with the best result from all runs being selected [24]. Later work [16] investigated its implementation on Graphics Processing Units (GPUs), which offered significant acceleration of the basic algorithm [1, 2, 4, 7, 9]. We now focus on key developments that contribute to the work described in the current paper.

The *independent-roulette* technique (*iRoulette*) was introduced by Cecilia *et al.* [4] as a parallel alternative to the traditional roulette-wheel selection method commonly used in sequential ACO algorithms. Roulette wheel selection is used whenever an ant must choose the next edge to traverse (and, thus, the next city to visit), with the probability of an edge being selected being proportional to its pheromone concentration. Although this is straightforward in the sequential algorithm, control flow and synchronization issues mean that it is more challenging in a parallel setting (Dawson and Stewart subsequently proposed an alternative *double-spin roulette* (DSRoulette) technique [8]). For an in-depth analysis of the properties of *iRoulette*, see [18].

With the availability of the Xeon Phi came new variants of *iRoulette*, due to the potential for vectorization offered by the its Vector Processing Unit (VPU). The algorithm described in [17] (which we refer to as *vRoulette-1*) is one example of this; the basic principles remain the same, but this variant makes use of intrinsic instructions available on the Xeon Phi to vectorize the *iRoulette* process, which yields improved performance over the original method (as well as over a vectorized version of the DSRoulette algorithm). We also note the existence of another vectorized version of *iRoulette*, *UVRoulette*, due to Tirado, *et al.* [31]. Along with their *vRoulette-1* method, Lloyd and Amos [17] utilized nearest neighbour information in their scheme for selecting cities. However, in this implementation, the candidate lists were used only to improve solution *quality*, and did not yield any speedup. In this paper, we describe an amended version of the algorithm described in [17], which replaces *vRoulette-1* with a properly vectorized nearest neighbour list (which we call *Vectorized Candidate Set Selection* (*VCSS*). As we will demonstrate, *VCSS* brings significant performance benefits in terms of execution time, especially with larger problem instances.

In the rest of this Section, we give a brief description of the base $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System (MMAS), and more details of the *iRoulette* method (both of which provide a foundation for our own contributions described in this paper).

2.1 MAX-MIN Ant System

ACO is an iterative algorithm, in which each step comprises two main phases: *Tour Construction* and *Pheromone Update*. During the tour construction phase, ants construct tours of the graph, making probabilistic choices based on heuristic weights derived from the lengths and pheromone values associated with edges in the complete graph. Each of the m ants starts on a different randomly-selected vertex of the graph. At each stage in the construction of a tour, the probability of ant k on vertex i choosing to move to vertex j is given by:

$$p_{i,j}^{k} = \begin{cases} \frac{[\tau_{i,j}]^{\alpha}[\eta_{i,j}]^{\beta}}{\sum_{i \in N_i^k}[\tau_{i,j}]^{\alpha}[\eta_{i,j}]^{\beta}} & i \in N_i^k \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

Here, $\eta_{i,j} = 1/d_{i,j}$ where $d_{i,j}$ is the length of edge (i, j), $\tau_{i,j}$ is the pheromone value for edge (i, j) and N_i^k is what we call the *feasible region* for vertex i. One constraint inherent to TSP problems is that cities can only be visited once. This is enforced in the ACO algorithm through the *tabu list*, which keeps track of every vertex visited during the current tour. During tour construction, vertices on the tabu list are ignored when making the choices. Thus, the *feasible region*, N_i is the set of all vertices *not* in the tabu list which are adjacent to vertex i.

During the pheromone update phase, ants deposit pheromone on the edges traversed in their tours in an amount that is proportional to the objective quality of the tour (measured in terms of its length). In $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System, only one ant (the iteration best or best-so-far ant) deposits pheromone. This makes $\mathcal{MM}$AS well-suited to a parallel implementation, since we do not require concurrent write access to the pheromone matrix in order to allow multiple agents to deposit pheromone in parallel.

The amount of pheromone deposited is given by

$$\tau_{i,j} = \tau_{i,j} + \Delta\tau_{i,j} \forall (i, j) \in L \quad (2)$$

where L is the set of edges in the complete graph and $\Delta\tau_{i,j}$ is the amount of pheromone deposited on edge (i, j), given by

$$\Delta\tau_{i,j} = \begin{cases} 1/C & \text{if edge}(i, j) \in T \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

where T is the set of edges in the iteration-best or best-so far tour, and C is the total length of this tour. Once pheromone has been distributed, the next step is *pheromone evaporation*, using the rule

$$\tau_{i,j} = (1 - \rho)\tau_{i,j} \forall (i, j) \in L \quad (4)$$

where $\rho \in [0, 1]$ controls the evaporation rate.

In $\mathcal{MM}$AS pheromone values in are "clamped" between two limits, τ_{min} and τ_{max}, which are defined by

$$\tau_{\max} = \frac{1}{pC_{\text{best}}}; \tau_{\min} = \tau_{\max}\frac{2(1-a)}{a(n_{\text{neighbours}} + 1)} \quad (5)$$

where $n_{\text{neighbours}}$ is the number of nearest neighbours and $a = \exp(\log(0.05)/n)$. These limits are used to prevent solution *stagnation*, where one edge dominates others in its vicinity, due to it having a significantly higher pheromone concentration than the others (such edges will effectively become "locked in" to solutions, leading to a rapid loss of diversity). This restriction exists in conjunction with the standard pheromone *evaporation* operation, which is used at every iteration to allow concentrations to gradually decay (and thus allow the algorithm to "forget" sub-optimal solutions, over time).

2.2 Independent Roulette

In serial implementations of ACO, the probabilistic selection of edges (according to Equation 1) is performed using the *Roulette Wheel* algorithm. Various approaches have been used to parallelize this algorithm: here we concentrate on *independent-roulette (iRoulette)* [4], which forms a component of our *VCSS* algorithm described in Section 3.4.

In *iRoulette*, a choice between N weights, $W_i, i \in [1, N]$ is made by multiplying the edge weights by uniform random deviates $R_i, i \in [1, N]$ with $R_i \in [0, 1]$. The selected edge is then

$$i_{\text{sel}} = \underset{i \in [1,N]}{\text{argmax}}\, W_i R_i .$$

In this scheme, the probability of selecting an edge is no longer proportional to the weights, although higher-weighted edges are more likely to be selected than those with lower weights. A detailed analysis of *iRoulette* [18] shows that this produces greedier selection than standard Roulette Wheel selection, but the effect on solution quality is small. Furthermore, for $\mathcal{MM}$AS using this scheme demonstrably speeds up convergence to a solution.

A vectorized version of *iRoulette*, *vRoulette-1* [17], forms part of a Xeon Phi implementation of $\mathcal{MM}$AS and is the inspiration for the work presented here. In this original algorithm, the generation of random deviates and weight multiplication is vectorized across 16 lanes, with a final reduction over the vector producing the maximum.

3 VECTORIZED CANDIDATE SET SELECTION

The key contribution in this work is a vectorized algorithm (and associated data structure) to accelerate vertex selection using candidate sets (nearest neighbour lists). The selection procedure is modified (compared to previous versions) so that *only vertices in the nearest neighbour list for the current vertex are considered.* Only in cases where all of these are *tabu* will the remainder of the feasible region be considered. Typically, the nearest neighbour maximum list length is set to ~20; for large instances (with thousands of vertices) this can speed up the selection process by an order of magnitude or more.

We base our algorithm implementation on the Xeon Phi code described in [17]. The code has been ported to use the *AVX512* vector instruction set of the *Knight's Landing* architecture (but it can be ported to any multi-core SIMD architecture). The Intel® Xeon Phi range of processors are designed for use in high-performance computing, containing between 57-72 cores depending on the processor. These cores provide 4 threads (concurrent processes) each, which enables a high level of parallelization and vectorization. The latest generation of Xeon Phi, Knight's Landing, has several benefits over the previous generation, Knight's Corner, including a higher number of processor, and support for AVX-512 Single Instruction Multiple Data (SIMD) instructions, which allows for highly efficient vectorization of the ACO algorithm.

3.1 Instance Preprocessing

A potential performance problem is caused by the distribution of nearest neighbours in the problem instance. The relative proximity of vertices in space is not necessarily correlated with the vertex indices (that is, two vertices that are spatially adjacent might have indices that are widely separated, and vice versa; see Figure 1). If the indices of nearest neighbours tend to be close together, the nearest neighbour list can be kept short. Conversely, in the worst case, the nearest neighbour list will contain the full set of N_p entries. In order to keep the size of the nearest neighbour list relatively low, we pre-process the problem instance before constructing the nearest neighbour list, by sorting the vertices into greedy tour order.

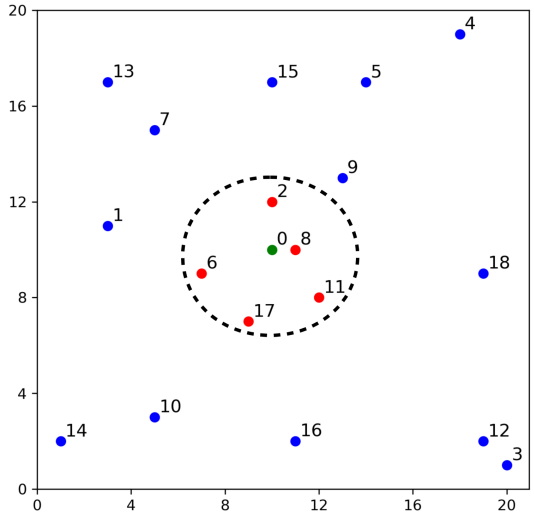

Figure 1: Sample TSP graph, with the current city (labelled 0) in the center, and five nearest neighbour cities highlighted in the dashed containing region.

3.2 Nearest neighbour List Construction

During the *setup* phase of the algorithm, the distance matrix (an $n \times n$ matrix encoding the edge lengths of the complete TSP graph) is used to calculate a vectorized nearest neighbour list data structure. This is performed as follows:

Let the number of nearest neighbours be N_{nn}, and the width of a SIMD vector (in floats) be p. We then let

$$N_p = \lceil N_{nn}/p \rceil,$$

the maximum number of SIMD vectors required to store one line of the nearest neighbour data structure. The data structure then comprises a list of up to N_p *Nearestneighbour* objects (one per vertex) with each *Nearestneighbour* entry containing an integer index `ivec` and a p-wide bitmap `mask`. To add a vertex j to the nearest neighbour list, we first ensure that there exists an entry with `ivec` $= \lfloor j/p \rfloor$, and set the bit in `mask` corresponding to j mod p.

The data structure for a vertex is filled as follows: first, the other vertices are sorted by distance, and the first N_{nn} of these are processed. For each of these vertices, `ivec` is calculated. If a *Nearestneighbour* entry already exists for this value of `ivec`, the appropriate bit is set in `mask`. If not, a new *Nearestneighbour* is added to the end of the list. The data structure is illustrated in Figure 2, for a vector width of 16.

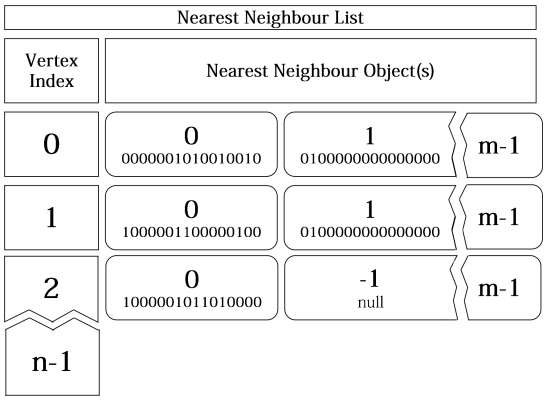

Figure 2: Nearest neighbour data structure, with each vertex having an associated array of nearest neighbour objects containing a vector index `ivec` and a bit mask. A sentinel value of `ivec` = −1 is used to indicate the end a line in the data structure. n is the number of vertices and n_{16} is the maximum number of entries for a vertex (for 16-wide vectors)

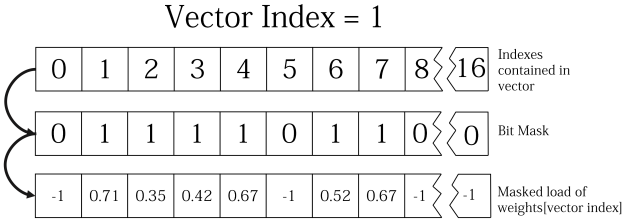

Figure 3: Masked load process using the nearest neighbour list to retrieve the weights of nearest neighbour vertices.

3.3 Tour Construction

We use OpenMP to assign each ant's tour construction process to a single thread. As no updates are made to any of the values used by the ants until the end of an iteration, and ants only write to their own local memory, no synchronization is required. We randomly select the starting vertex for each tour. We then repeatedly call the edge selection function to determine the ant's path around the graph. Each ant maintains its own *tabu* list, which keeps track of visited vertices.

In our experiments, described below, we evaluate two vectorized edge selection functions: *vRoulette-1* (which examines *every* vertex in the feasible region) and *Vectorized Candidate Set Selection* (*VCSS*), our new vectorized procedure, which uses nearest neighbour information. We now describe *VCSS* in more detail.

3.4 Vectorized Candidate Set Selection (VCSS)

Here, we propose a new selection procedure, *Vectorized Candidate Set Selection*, based on *iRoulette* [4], in which *iRoulette* selection is performed on a candidate set drawn from the nearest neighbour list data structure. If this *fails* to produce a selection (which will only happen when all the nearest neighbours have already been visited in the tour), the *vRoulette-1* procedure is used to select a vertex from the remaining feasible region.

VCSS takes the tabu list, an array of weights and the nearest neighbour list for the current vertex and then proceeds as follows (assuming a vector width p): first, we initialize p-wide vectors representing the running maximum weight and corresponding vertex indices. The algorithm then iterates over the nearest neighbour list. For each *Nearestneighbour* object, we load the edge weights for the corresponding vertices as a single p-wide vector. We use the bitmask in the *Nearestneighbour* object to mask this weight vector such that only the vertices in the nearest neighbour list remain (illustrated in Figure 3).

We also construct a vector of consecutive integers, corresponding to the vertex indices in the vector. We multiply the weight vector by a p-wide vector of random deviates (produced using a simple linear congruential generator). We then compare the modified weight vector, on an element-wise basis (in a single instruction), with the running maximum. This produces a bit mask which is used to update both the running maximum and the corresponding index vector. A reduction is then performed over the vector lanes to produce the maximum weight and corresponding index (in our implementation, this reduction is performed in $\log_2 p$ steps using bit-swizzling instructions). If no edge has been selected, the *vRoulette-1* algorithm is used by default on the complete set of weights.

The algorithm is formally described in Algorithm 1. Here, *Random()* is a function which returns a p-wide vector of uniform deviates, and *ApplyMask(mask, a, b)* is a function which returns a vector filled with values from a in positions where the corresponding value of *mask* is set, and values from b where the *mask* value is not set.

Algorithm 1: Pseudo-code for Vectorized Candidate Set Selection.

Input : Edge Weight array $\mathbf{W}_{0\ldots N-1}$, Tabu Mask array $\mathbf{T}_{0\ldots n-1}$, Maximum number of nearest neighbours N_p, nearest neighbour list $L_{0\ldots N_p-1}$
Output: Selected Edge
// Variables in bold are p-vectors, superscripts indicate vector lanes
$\mathbf{W}_{max} = (0\ldots0)$;
$\mathbf{I}_{max} = (0\ldots0)$;
for $i = 0$ **to** $N_p - 1$ **do**
  if $L[i]$.ivec $\neq -1$ **then**
    $\mathbf{R} = Random()$;
    $\mathbf{V} = L[i]$.mask;
    $\mathbf{I} = (pL[i]$.ivec$\ldots pL[i]$.ivec $+ p - 1)$;
    $\mathbf{w} = ApplyMask(V_i, \mathbf{W}_i \times \mathbf{R}, (-1\ldots-1))$;
    $\mathbf{w} = ApplyMask(T_i, (-1\ldots1), \mathbf{w})$;
    $max\_mask = \mathbf{w} > \mathbf{W}_{max}$;
    $\mathbf{I}_{max} = ApplyMask(max\_mask, \mathbf{w}, \mathbf{W}_{max})$;
  end
end
//Reduction
j = argmax($\mathbf{W}_{max}$);
return $\mathbf{I}^j_{max}$;

3.5 Pheromone Update

The pheromone update process is split into four phases: Deposit, evaporation, clamping and edge probability calculation. The deposit phase is carried out by a single thread, with little potential for vectorization. The remaining phases are carried out in a pair of nested loops, with the outer loop being parallelized with OpenMP. The inner loop iterates over the pheromone matrix 16 values at a time, using vector instructions to perform the evaporation, clamping and probability calculations.

4 EXPERIMENTAL RESULTS

In what follows, we measure the performance three variants of ACO: the first is CPU reference code, the second uses only *vRoulette-1*, and the third uses our new *VCSS* method. Experiments were carried out on a machine with an Intel® Xeon Phi 7210 processor with 64 cores and 4 threads per core (for a total of 256 threads), running at a base speed of 1.3GHz. The code was compiled with the Intel® C++ compiler (`icc`) at -O3 optimization level. The code was run under Linux, with timings obtained using the *gettimeofday()* function. The CPU reference code used is ACOTSP version 1.03 [23], compiled with `gcc` (with optimization -O3) and run on a Linux machine containing an 8-core Intel® Xeon E5-2650 v2 at a base speed of 2.6GHz.

4.1 ACO Parameters and Problem Instances

We use 32 nearest neighbours for our experiments, which is both in line with Dorigo and Stützle's recommended list size [29], and a convenient power of two for the purposes of data alignment. The values of the $\mathcal{MM}$AS parameters used are as follows: $\alpha = 1, \beta = 2, \rho = 0.02$. In all cases, the number of ants is set to 256 (so that all available threads are used when assigning ants to threads).

The problem instances used in our experiments are taken from the TSPLIB library[20], and include all instances solved in [17] and [32]. We also include larger instances, in order to investigate the performance of the algorithm as the problem size increases. The instances used are: `lin318`, `pcb442`, `rat783`, `pr1002`, `fl1577`, `pr2392`, `fl3795`, `rl5934`, `pla7397`, and `rl11849`. For each instance, we performed 50 runs of 1024 iterations.

4.2 Execution Time

We measure execution time on a per-iteration basis. Results are shown in Figure 4, which (log) plots the mean time per iteration over all instances for *VCSS* and *vRoulette-1* on the Xeon Phi, and ACOTSP on CPU, and Table 1, which gives the numerical values, along with the speedup.

While *vRoulette-1* and *VCSS* have similar execution times on the smaller instances, as the instance size grows, the execution time for *VCSS* grows more slowly than that of *vRoulette-1*. For the largest instance, `rl11849`, *VCSS* is faster than *vRoulette-1* by an order of magnitude, and is faster than the reference code by two orders of magnitude. On the other hand, *vRoulette-1*, while performing two orders of magnitude faster than the reference code on smaller instances, shows a declining speed-up compared to the CPU as the instance size grows.

Table 1: Execution time per iteration in milliseconds, and speedup relative to CPU.

	CPU	*vRoulette-1*		*VCSS*	
Instance	t/ms	t/ms	Speedup	t/ms	Speedup
`lin318`	18.1	0.73	24.8	0.52	34.8
`pcb442`	29.2	1.37	21.3	0.74	39.5
`rat783`	68.3	5.65	12.1	1.37	49.9
`pr1002`	94.1	9.01	10.4	1.85	50.9
`fl1577`	176.7	20.9	8.45	3.5	50.1
`pr2392`	371.0	46.4	8.00	7.02	52.9
`fl3795`	785.7	143.3	5.48	13.5	58.2
`rl5934`	2088.9	426.8	4.90	27.9	74.9
`pla7397`	3388.3	724.3	4.68	43.04	78.7
`rl11849`	10578.8	1975.0	5.36	97.47	108.5

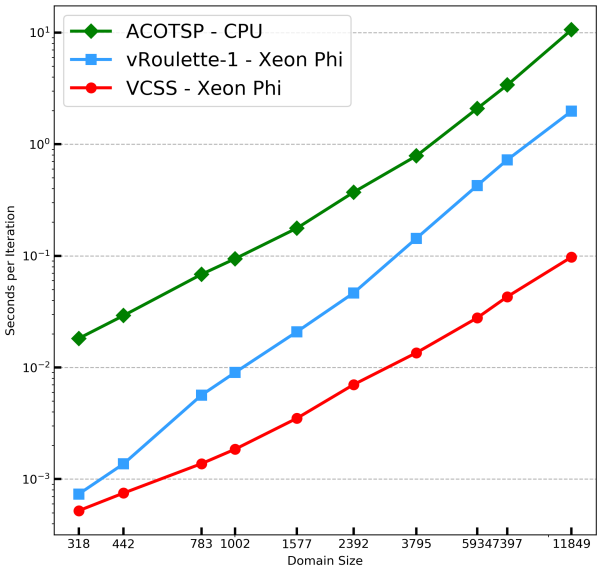

Figure 4: Execution times for *ACOTSP*, *vRoulette-1* and *VCSS*.

4.3 Solution Quality

In Figure 5 we show box plots of solution quality of each algorithm (where solution quality is measured as the ratio of the length of the best tour found to the known optimum for the problem instance). We would expect differences between the solution quality obtained with the CPU code and the two Xeon Phi variants due to the modified selection probabilities in the *iRoulette* scheme compared with those in the roulette wheel selection used by ACOTSP. It is already known that *iRoulette* can affect the solution quality on individual instances, although its average behavior does not significantly affect the quality of solution [18]. There is some variation between the solution qualities obtained using *vRoulette-1* and *VCSS*. It should be noted that this experiment used a relatively small sample of instances, with 50 runs per instance. In order to measure effects on solution quality, a larger sample of instances (with one run per instance) would be better. Additionally, for the larger instances, the number of iterations (1024) is relatively small and the solutions may

not be converged. However, the focus of this paper is the efficiency measured in terms of time per iteration: in order to investigate any effects on solution quality, more extensive experiments would be required. Given that VCSS is formally equivalent to the nearest-neighbour list algorithms already widely studied in serial ACO, we would not expect to see large systematic effects on the solution quality arising from the use of *VCSS*, although this will be a topic for further investigation.

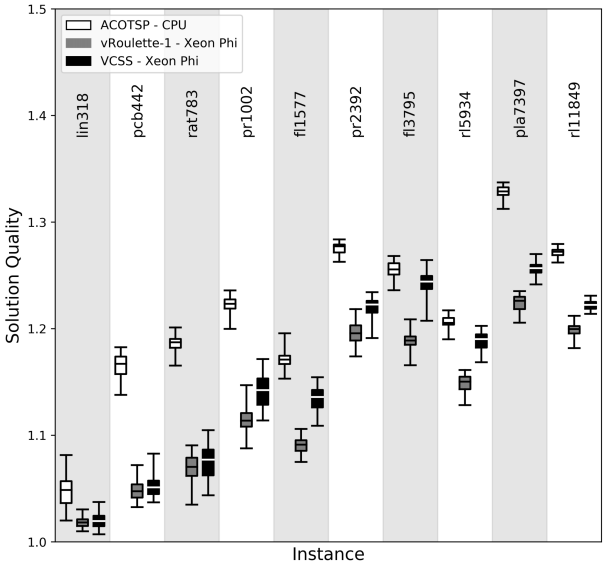

Figure 5: Solution quality for *ACOTSP*, *vRoulette-1* and *VCSS*.

4.4 Discussion

We have demonstrated that significant speedups are obtained using our *VCSS* scheme. The speedup over *vRoulette-1* grows as the instance size increases. Without the nearest neighbour list, the tour construction process has time complexity $O(n^2)$ (since at each of n vertices, $n-1$ vertices are included in the selection process). The nearest neighbour list reduces this complexity to $O(n)$ (since the workload per vertex is constant, determined only by the size of the nearest neighbour list). The speedup, relative to the CPU code, also increases with the instance size. This is more difficult to explain, since the CPU code also uses a fixed size nearest neighbour list, and the execution time should therefore scale in the same way. However, considering the number of 16-wide vectors which are processed in making a selection with *vRoulette-1*, we see that - for instances up to around 500 vertices - this is less than or equal to the size of the nearest neighbour list. The increase in speed up is, therefore, due to the nearest neighbour list conferring minimal benefit with smaller instances. In this case, we would expect the speedup to eventually level off.

5 CONCLUSION

In this paper, we presented *VCSS*, a novel nearest neighbour vectorization technique and selection method for ant colony optimization. This method is an order of magnitude faster than the previous best-performing algorithm on the Xeon Phi platform, *vRoulette-1*, and two orders of magnitude faster than the reference CPU code.

While we have shown that the performance of *VCSS* improves with increasing instance size, there is a limit to how far this can be pursued. One of the inherent limitations of the general ACO algorithm is its $O(n^2)$ memory complexity, due to the need to store a square pheromone matrix. Around 500MB of memory is required for our largest instance, and to move to the next order of magnitude (a 100,000 city instance) we would require around 37GB. This limitation must be overcome before the speed gains obtained using the latest parallel and vector ACO techniques can be fully exploited on larger instances. The PartialACO [5] method is a promising development in overcoming this barrier.

Our solution may also benefit further from the inclusion of *local search*, which is often used to accelerate convergence [13]. While local search *may* be parallelized, there are currently no vectorized algorithms which can utilize the full power of many-core SIMD hardware. The possibility of using local search with parallel ACO requires further investigation.

There are a number of areas for further investigation in terms of the details of our *VCSS* algorithm. Firstly, the fall-back to *vRoulette-1* when all nearest neighbours are tabu introduces a potential load-balance issue (although, in practice, this happens very rarely). The work carried out by each thread will differ, depending on how many times *vRoulette-1* is used, and all threads must wait for the slowest to complete. This could be alleviated either by using a faster fall-back algorithm, or by organizing the workload differently, with ants sending work to a pool of threads, rather than decomposing the work strictly by ant.

Secondly, the distribution of the lengths of the nearest neighbour lists (in terms of the number of vectors required to store the nearest neighbours) is another potential source of load imbalance. The greedy tour scheme used here to reorder the vertices may be improved upon. It is possible, for example, that a distribution of list lengths with a larger mean, but smaller variance, could give shorter execution times, due to improved load balancing. A full analysis of the relationship between this distribution and the load balance is another area for future work.

REFERENCES

[1] Alberto Cano, Juan Luis Olmo, and Sebastián Ventura. Parallel multi-objective ant programming for classification using GPUs. *Journal of Parallel and Distributed Computing*, 73(6):713 – 728, 2013.
[2] José M Cecilia, José M García, Manuel Ujaldón, Andy Nisbet, and Martyn Amos. Parallelization strategies for Ant Colony Optimisation on GPUs. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 339–346, May 2011.
[3] José M Cecilia, Andy Nisbet, Martyn Amos, José M García, and Manuel Ujaldón. Enhancing GPU parallelism in nature-inspired algorithms. *The Journal of Supercomputing*, 63(3):773–789, 2013.
[4] José M Cecilia, José M García, Andy Nisbet, Martyn Amos, and Manuel Ujaldón. Enhancing data parallelism for Ant Colony Optimization on GPUs. *Journal of Parallel and Distributed Computing*, 73(1):42–51, 2013.
[5] Darren M Chitty. Applying ACO to large scale TSP instances. In *UK Workshop on Computational Intelligence*, pages 104–118. Springer, 2017.
[6] Laurence Dawson. *Generic Techniques in General Purpose GPU Programming with Applications to Ant Colony and Image Processing Algorithms.* PhD thesis, Durham

University, UK., 2015.
[7] Laurence Dawson and Iain A Stewart. Candidate set parallelization strategies for Ant Colony Optimization on the GPU. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 216–225. Springer, 2013.
[8] Laurence Dawson and Iain A Stewart. Improving Ant Colony Optimization performance on the GPU using CUDA. In *2013 IEEE Congress on Evolutionary Computation*, pages 1901–1908, June 2013.
[9] Audrey Delévacq, Pierre Delisle, Marc Gravel, and Michaël Krajecki. Parallel ant colony optimization on graphics processing units. *Journal of Parallel and Distributed Computing*, 73(1):52–61, 2013.
[10] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, 2006.
[11] Marco Dorigo and Luca Maria Gambardella. Ant colonies for the Travelling Salesman Problem. *BioSystems*, 43(2):73–81, 1997.
[12] Marco Dorigo and Luca Maria Gambardella. Ant colony system: a cooperative learning approach to the Traveling Salesman Problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, Apr 1997.
[13] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. Bradford Company, Scituate, MA, USA, 2004.
[14] Jie Fu, Lin Lei, and Guohua Zhou. A parallel Ant Colony Optimization algorithm with GPU-acceleration based on All-In-Roulette selection. In *Advanced Computational Intelligence (IWACI), 2010 Third International Workshop on*, pages 260–264, 2010.
[15] Fred Glover. Tabu search – Part I. *ORSA Journal on computing*, 1(3):190–206, 1989.
[16] Wang Jiening, Dong Jiankang, and Zhang Chunfeng. Implementation of Ant Colony Algorithm based on GPU. In *CGIV '09: Proceedings of the 2009 Sixth International Conference on Computer Graphics, Imaging and Visualization*, pages 50–53, Washington, DC, USA, 2009. IEEE Computer Society.
[17] Huw Lloyd and Martyn Amos. A highly parallelized and vectorized implementation of Max-Min Ant System on Intel Xeon Phi. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–6, Dec 2016.
[18] Huw Lloyd and Martyn Amos. Analysis of independent roulette selection in parallel Ant Colony Optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, pages 19–26, New York, NY, USA, 2017. ACM.
[19] Marcus Randall and James Montgomery. Candidate set strategies for Ant Colony Optimisation. In *International Workshop on Ant Algorithms*, pages 243–249. Springer, 2002.
[20] Gerhard Reinelt. TSPLIB - a Traveling Salesman Problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.
[21] Rafał Skinderowicz. The GPU-based parallel Ant Colony System. *Journal of Parallel and Distributed Computing*, 98(Supplement C):48 – 60, 2016.
[22] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights Landing: Second-generation Intel® Xeon Phi product. *IEEE Micro*, 36(2):34–46, 2016.
[23] Thomas Stützle. ACOTSP. Available at http://iridia.ulb.ac.be/~mdorigo/ACO/downloads/ACOTSP-1.03.tgz (2005/06/12).
[24] Thomas Stützle. Parallelization strategies for Ant Colony Optimization. In *PPSN V: Proceedings of the 5th International Conference on Parallel Problem Solving from Nature*, pages 722–731, London, UK, 1998. Springer-Verlag.
[25] Thomas Stutzle and Marco Dorigo. A short convergence proof for a class of ant colony optimization algorithms. *IEEE Transactions on Evolutionary Computation*, 6(4):358–365, 2002.
[26] Thomas Stützle and Holger Hoos. MAX-MIN ant system and local search for the Traveling Salesman Problem. In *Evolutionary Computation, 1997., IEEE International Conference on*, pages 309–314, Apr 1997.
[27] Thomas Stützle and Holger H Hoos. Max–min ant system. *Future Generation Computer Systems*, 16(8):889–914, 2000.
[28] Thomas Stützle, Manuel López-Ibáñez, and Marco Dorigo. A concise overview of applications of Ant Colony Optimization. In James J. Cochran, Louis A. Cox, Pinar Keskinocak, Jeffrey P. Kharoufeh, and J. Cole Smith, editors, *Wiley Encyclopedia of Operations Research and Management Science*. John Wiley & Sons, Inc., 2010.
[29] Thomas Stützle and Marco Dorigo. Ant Colony Optimization. 01 2004.
[30] Xinmin Tian, Hideki Saito, Serguei V Preis, Eric N Garcia, Sergey S Kozhukhov, Matt Masten, Aleksei G Cherkasov, and Nikolay Panchenko. Practical SIMD vectorization techniques for Intel® Xeon Phi coprocessors. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1149–1158. IEEE, 2013.
[31] Felipe Tirado, Ricardo J. Barrientos, Paulo González, and Marco Mora. Efficient exploitation of the Xeon Phi architecture for the Ant Colony Optimization (ACO) metaheuristic. *The Journal of Supercomputing*, 73(11):5053–5070, Nov 2017.
[32] Felipe Tirado, Angelica Urrutia, and Ricardo J. Barrientos. Using a coprocessor to solve the Ant Colony Optimization algorithm. In *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–6, Nov 2015.