A Highly Parallelized and Vectorized Implementation of Max-Min Ant System on Intel® Xeon Phi™.

Huw Lloyd
Informatics Research Centre
Manchester Metropolitan University
Manchester, M1 5GD, UK
Email: huw.lloyd@mmu.ac.uk

Martyn Amos
Informatics Research Centre
Manchester Metropolitan University
Manchester, M1 5GD, UK
Email: m.amos@mmu.ac.uk

***Abstract*—The increasing trend in processor design towards many-core architectures with wide vector processing units is largely motivated by the fact that single core performance has hit a 'power wall', meaning that performance gains are currently achievable only through increasingly parallel and vectorized execution models. Consequently, applications can only exploit the full performance of modern processors if they achieve high parallel and vector efficiencies. In this paper, we illustrate how this might be achieved for the well-established Ant Colony Optimization metaheuristic. We describe a highly parallel and vectorized variant of the Max-Min Ant System algorithm applied to the Traveling Salesman Problem, and present two novel vectorized algorithms for selecting cities during the tour construction phase. We present experimental results from an implementation on the Intel® Xeon Phi™ platform, which show that very high parallel and vector efficiencies are achieved, and significant speedups are obtained compared to both the reference serial implementation and the previous best Xeon Phi implementation available in the literature.**

I. Introduction

In this paper, we consider the implementation of a well-known metaheuristic method on a new parallel computing platform. Ant Colony optimization (ACO) [1], [2], [3] is a well-established population-based optimization algorithm inspired by the foraging behavior of ants. It has been applied to a wide range of NP-hard problems such as the Traveling Salesman Problem (TSP), in which the aim is to find the shortest Hamiltonian circuit of a graph. Applied to the TSP, a population of simulated ants (agents) construct tours of the graph, choosing vertices probabilistically based on a combination of heuristic weights and a simulated pheromone trail. After constructing their tours, ants deposit pheromone on the visited edges in proportion to the quality of the tour (which encourages the following ants to select those edges with higher probability, leading to positive reinforcement of higher-quality tours). In order to remove unproductive paths from the search, pheromone is gradually evaporated from the trail. Several variants of the basic algorithm have been proposed (see [3] for a review). In this paper we concentrate on the Max-Min Ant System algorithm (MMAS) [4], which has emerged as the best overall performing variant for the Travelling Salesman Problem [3].

The main motivation for this paper lies in advances in processor chip technology. In recent years, the use of coprocessors with many-core architectures for accelerating compute-intensive workloads has received widepread attention; Graphics Processing Units (GPUs) are widely used, especially NVIDIA® hardware programmed using the proprietary CUDA API [5]. The GPU has provided a valuable platform for nature-inspired algorithms [6], [7] and other metaheuristics [8], [9], [10], and several authors have specifically used it to study ACO [11], [12].

The latest generation of NVIDIA hardware, *Tesla*, offers up to 4992 CUDA cores [13], but fine-grained parallelism is required in order to extract the full performance from this class of hardware. A recent entry into the coprocessor market is Intel's Xeon Phi coprocessor, based on the Many Integrated Core (MIC) [14] architecture.

The current generation of Xeon Phi (at the time of writing) is known as *Knight's Corner*, and has up to 61 cores: one of these is reserved for the operating system (Busybox Linux) whereas the other 60 are available to applications. Each core runs four hardware threads, giving up to 240 threads in total. The cores run at a relatively low clock speed (1.2GHz), and each core has a vector processing unit (VPU) which operates on 512-byte wide registers. In order to best exploit the hardware, it is therefore essential that applications make full use of the fine-grained data parallelism offered by the VPU, in addition to filling all 240 threads with work. Using all four threads on a core offers the best chance of hiding latency.

In this paper we present an implementation of the MMAS algorithm on the *Knight's Corner* architecture. In this way, we demonstrate how a fundamental metaheuristic method may be optimized for this important new parallel computing architecture. We present two vectorized algorithms for selecting vertices during the tour construction phase, and show that the code makes efficient use of the VPU and the hardware threads.

The remainder of the paper is organized as follows: in section II we summarize related work in implementing ACO

on MIC and GPUs. Section III describes the MMAS algorithm and the details of our implementation. Section IV presents the results of experiments conducted with the code. We summarize our findings in section V.

II. Related Work

Fu, *et al.* [15] implemented the MMAS algorithm on GPU, and introduced the *'All-in Roulette'* algorithm for selecting the next vertex in a tour. Several authors subsequently explored data parallel approaches to the Ant System (AS) variant of ACO on GPU, [11], [12], [16], [17]. Cecilia, *et al.* [11] introduced the *Independent Roulette (IRoulette)* algorithm for selecting vertices in tour construction as an alternative to the traditional roulette wheel approach. In this method, the weights associated with the edges are independently multiplied by random numbers in parallel, and a parallel reduction is carried out to find the largest product. As in *All-in-roulette*, the probabilities are not proportional to the weights, but higher weighted edges are more likely than lower weighted edges, and the scheme can be implemented efficiently on GPU. Around the same time, Dawson and Stewart [12] introduced the *Double Spin Roulette (DS Roulette)* scheme, in which two roulette wheels selections are used; one which selects a block of threads (and its associated vertices) and a second which selects within the chosen block. DS Roulette preserves the proportionality between the edge weights and selection probabilities.

Candidate sets are an important optimization in ACO implementations [3]. By only considering the 'most likely' vertices when constructing a tour, runtime can be reduced considerably, and better solutions may even be obtained in some cases [18]. Several authors use nearest-neighbor lists to accelerate their GPU implementations of the AS algorithm [19], [16].

To the best of our knowledge, there exist only two published studies of Ant Colony Optimization on the Xeon Phi platform. Sato, *et al.* [20] used ACO to solve the Quadratic Assignment Problem on Xeon Phi and Graphics Processing Unit (GPU). They conclude that the Xeon Phi gives poor performance for this application compared to GPU; however it seems their implementation made no use of the SIMD capability of Xeon Phi, which is essential for good performance. Tirado, *et al.* [21] used the Ant System algorithm to solve the Travelling Salesman Problem on Xeon Phi. Their implementation obtained significant speedups over the single-threaded CPU. Methods were tested for constructing the tours, which differed in their memory access patterns, and for assessing whether the weights were calculated using the *pow()* function, or multiplied out explicitly. Best performance was obtained when the memory access was cache-friendly, and without using *pow()*.

III. MMAS on Xeon Phi

A. MMAS Algorithm

An iteration of the MMAS algorithm comprises two stages: *tour construction* and *pheromone update*. The ant system contains m ants. At the beginning of the tour construction stage, each ant is placed randomly on one of the n vertices of the graph. At each subsequent step in the construction of a tour, ants randomly select the next vertex to visit; the probability of ant k, currently placed on vertex i, choosing vertex j is given by

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

where $\eta_{i,j} = 1/d_{i,j}$, $d_{i,j}$ is the length of the edge connecting vertices i and j and $\tau_{i,j}$ is the amount of pheromone associated with edge i, j. N_i^k is the feasible region for ant k on vertex i – this is simply the set of vertices not yet visited on the current tour, and is maintained in practice using the *tabu list*, a list of the vertices already visited by a given ant. The two parameters α and β are fixed at the beginning of a run and control the relative importance of edge cost and pheromone in determining the probabilities.

When all ants have completed their tours, the pheromone values associated with each edge of the graph are updated. Firstly, the pheromone values are *evaporated* according to the rule

$$\tau_{i,j} \leftarrow (1-\rho)\tau_{i,j} \forall (i,j) \in L \tag{2}$$

where $\rho \in [0,1]$ is a parameter which controls the rate of evaporation and L is the set of edges in the complete graph. Finally, pheromone is deposited on all edges which form part of either the iteration-best or best-so-far tour. The pheromone is updated using

$$\tau_{i,j} \leftarrow \tau_{i,j} + \Delta\tau_{i,j} \forall (i,j) \in L \tag{3}$$

where $\Delta\tau_{i,k}$ is the amount of pheromone deposited on edge (i,j), given by

$$\Delta\tau_{i,k} = \begin{cases} 1/C & \text{if edge}(i,j) \in T \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

where T is the set of edges in the iteration-best or best-so-far tour, and C is the total cost of tour T – this is simply the sum of the edge lengths, $\sum_{i,j \in T} d_{i,j}$.

In the MMAS algorithm, the pheromone values are clamped between limits $\tau_{\min}$ and $\tau_{\max}$. These limits are given by

$$\tau_{\max} = \frac{1}{\rho C_{\text{best}}}; \tau_{\min} = \tau_{\max} \frac{2(1-a)}{a(n_{\text{neighbors}}+1)} \tag{5}$$

where $a = \exp(\log(0.05)/n)$ and $n_{\text{neighbors}}$ is the length of the nearest-neighbor list (see section III-D).

We now describe how each of these stages were implemented on the Xeon Phi.

B. Tour Construction

We use OpenMP to distribute the work of the tour construction phase by assigning ants to threads. This can be done without any synchronization, as the tour construction phase requires only read access to the shared pheromone trail and edge length data, and write access is required only for memory which is private to each ant. We use all of the 240 hardware

threads on the Xeon Phi to maximize latency-hiding. Each ant maintains a *tabu list* during the tour construction phase, which tracks the vertices already visited in the current tour. We store the tabu list as an array of 16-bit masks.

To construct an ant's tour, a random vertex is selected, and the bit corresponding to that vertex is set in the tabu list. We then repeatedly call an edge selection function, which takes as input (a) the tabu list and (b) an array of weights for all edges which include the current vertex. The edge weights are stored in a two dimensional array of 32-bit floats. Each row of the array is aligned on a 64-byte boundary (which allows efficient loading into AVX-512 registers) – the pointer passed to the edge selection function is then simply the beginning of the ith row of the weights array, where i is the current vertex. Having selected the next vertex, the tour length and tabu list are updated, and the process repeats until the tour is complete.

We implemented two distinct edge selection functions, which we call *Vectorized Roulette-1* and *Vectorized Roulette-2* (*vRoulette-1* and *vRoulette-2*). In both cases, the functions were vectorized by hand using the AVX-512 compiler intrinsics.

1) vRoulette-1: vRoulette-1 is a vectorized version of the *IRoulette* algorithm ([11]). In this algorithm, each edge weight is multiplied by a uniform random deviate, and the selected edge is the edge not in the tabu list with the highest product of weight and random number. This algorithm does not select edges with probabilities proportional to the weights; however, edges with higher weights are more likely to be selected than edges with lower weights. Our vectorized implementation loops through the weights 16 at a time, maintaining 16-wide vectors of the highest-per-lane products and of the corresponding indices, before performing a serial reduction to find the selected edge. The algorithm is given in Algorithm 1 for p-wide vectors. Here, `Random()` is a function which returns a vector of p random deviates, and `MaskedSave( mask, a, b )` is a function which returns a vector in which elements are taken from `a` if the corresponding bit in `mask` is set, otherwise from `b`. Masks are generated as the output of vectorized Boolean operations.

2) vRoulette-2: vRoulette-2 chooses edges with probabilities which are proportional to the weights. The algorithm uses repeated binary trials, choosing with probability proportional to the weights. The winner of a trial accumulates the weight assigned to the loser. It is straightforward to show that this produces a final result in which the probabilities are proportional to the weights. A similar method (with two trials) is used in the *DS-Roulette* algorithm [12]. Again, the weights and indices are tracked in 16-wide vectors, and 16 trials are conducted simultaneously using the AVX-512 vector instructions. A final serial phase performs a binary tree of trials to decide the selected edge. The algorithm is given in Algorithm 2.

ALGORITHM 1: vRoulette-1

Input: Edge weight p-vector array $\mathbf{W}_{0\ldots N-1}$, Tabu mask array $T_{0\ldots N-1}$
Output: Selected edge
// Variables in bold are p-vectors, superscripts indicate vector lanes

$\mathbf{W}_{\max} \leftarrow (0\ldots0)$;
$\mathbf{I}_{\max} \leftarrow (0\ldots0)$;
$\mathbf{I} \leftarrow (0\ldots p-1)$;
for $i \leftarrow 0$ **to** $N-1$ **do**
  $\mathbf{R} \leftarrow$ `Random()`;
  $\mathbf{w} \leftarrow$ `MaskedSave(`T_i, $(-1\ldots-1)$, $\mathbf{W}_i \times \mathbf{R}$`)`;
  max_mask $\leftarrow \mathbf{w} > \mathbf{W}_{\max}$;
  $\mathbf{W}_{\max} \leftarrow$ `MaskedSave(` *max_mask*, $\mathbf{w}$, $\mathbf{W}_{\max}$`)`;
  $\mathbf{I}_{\max} \leftarrow$ `MaskedSave(` *max_mask*, $\mathbf{I}$, $\mathbf{I}_{\max}$`)`;
  $\mathbf{I} \leftarrow \mathbf{I} + (p\ldots p)$;
end
// Serial Reduction
j = argmax($\mathbf{W}_{\max}$);
return $\mathbf{I}^{j}_{\max}$;

C. Pheromone Update

In the Max-Min Ant System algorithm, only one ant (the iteration-best, or best-so-far ant) deposits to the pheromone trail. This process is carried out using a single thread, and there is little scope for vectorization.

After the pheromone is deposited, the code performs three tasks: evaporation of the pheromone trail, application of the pheromone minimum and maximum limits, and calculation of edge weights for the next round of tour construction. These three operations are computed in a pair of nested loops which iterate over the pheromone matrix. We parallelize the outer loop with OpenMP over 240 threads, leaving an inner loop which vectorizes trivially. The edge weight calculation is accelerated by hard-coding the powers α and β using multiplication (rather than calling the *pow()* function) and by using a precomputed matrix of inverse edge lengths (saving a divide). Note that, as neither of the edge selection schemes require the weights to be normalized, we do not need to compute the denominator in equation 1; we store a two-dimensional array of edge weights

$$W_{i,j} = [\tau_{i,j}]^{\alpha}[\eta_{i,j}]^{\beta} \tag{6}$$

D. Nearest-neighbor Lists

The reference implementation of MMAS [22] uses a list of nearest neighbors at each vertex to accelerate the calculation. When constructing a tour, the next edge is chosen from the nearest neighbor list, unless all nearest neighbors have already been visited, in which case the shortest of the remaining edges is chosen. As well as optimizing the serial version of the algorithm, this technique may also improve the solution quality [18]. In order to obtain the potential solution quality benefits of the nearest neighbor list, we emulate this effect in our vectorized implementation.

At the beginning of the calculation, we compute the nearest neighbor lists, and then construct a two-dimensional array of

ALGORITHM 2: vRoulette-2

Input: Edge weight p-vector array $\mathbf{W}_{0\ldots N-1}$, Tabu mask array $T_{0\ldots N-1}$
Output: Selected edge
// Variables in bold are p-vectors, superscripts indicate vector lanes

$\mathbf{W}_{\text{acc}} \leftarrow (0\ldots0)$;
$\mathbf{I}_{\text{win}} \leftarrow (0\ldots0)$;
$\mathbf{I} \leftarrow (0\ldots p-1)$;
for $i \leftarrow 0$ **to** $N-1$ **do**
  $\mathbf{R} \leftarrow$ `Random()`;
  $\mathbf{w} \leftarrow$ `MaskedSave(`T_i, $(0\ldots0)$, $\mathbf{W}_i$`)`;
  $\mathbf{W}_{\text{acc}} \leftarrow \mathbf{W}_{\text{acc}} + w$;
  win_mask $\leftarrow (\mathbf{W}_{\text{acc}} \times \mathbf{R}) < \mathbf{w}$;
  $\mathbf{I}_{\text{win}} \leftarrow$ `MaskedSave(` *win_mask*, $\mathbf{I}$, $\mathbf{I}_{\text{win}}$`)`;
  $\mathbf{I} \leftarrow \mathbf{I} + (p\ldots p)$;
end
// Serial Phase
numTrials $\leftarrow p/2$;
$\mathbf{R} \leftarrow$ `Random()`;
iRandom $\leftarrow 0$;
for $i \leftarrow 0$ **to** $\log_2(p) - 1$ **do**
  for $j \leftarrow 0$ **to** $numTrials - 1$ **do**
    $\mathbf{W}^{2j}_{\text{acc}} \leftarrow \mathbf{W}^{2j}_{\text{acc}} + \mathbf{W}^{2j+1}_{\text{acc}}$;
    if $\mathbf{W}^{2j}_{\text{acc}} \times \mathbf{R}^{iRandom} < \mathbf{W}^{2j+1}_{\text{acc}}$ **then**
      $\mathbf{I}^{2j}_{\text{win}} \leftarrow \mathbf{I}^{2j+1}_{\text{win}}$;
    end
    $\mathbf{I}^{j}_{\text{win}} \leftarrow \mathbf{I}^{2j}_{\text{win}}$;
    $\mathbf{W}^{j}_{\text{acc}} \leftarrow \mathbf{W}^{2j}_{\text{acc}}$;
    iRandom $\leftarrow$ *iRandom* $+1$;
  end
  numTrials $\leftarrow$ *numTrials* $/2$;
end
return $\mathbf{I}^{0}_{\text{win}}$;

floating point values $f_{i,j}$ where $f_{i,j} = 1$ if edge (i, j) is in the nearest neighbor list for vertex i and 0 otherwise. Then, when computing the weight for edge (i, j), we modify the weight such that

$$W'_{i,j} = W_{i,j}\,(Af_{i,j} + 1) \tag{7}$$

where A is some constant $\gg 1$ (we use 1000). In this way, edges *not* in the nearest neighbor list have a very low probability of selection unless all the nearest neighbor edges are already used in the tour. This correction is folded in to the weight calculation, and also vectorizes trivially.

IV. Experimental Results

We carried out experiments on the *Iden* cluster at the UK STFC Hartree Centre, using compute nodes equipped with Xeon Phi 5110P accelerators. The code was compiled using the Intel C++ compiler (`icc` version 16.0.0) with the optimization level set to *-O3*. The code was run in native mode on the Xeon Phi, and all timings were obtained by instrumenting the code with calls to the standard `gettimeofday` function which provides a microsecond timer. The reference (baseline) sequential implementation is the *ACOTSP* code [22], which was compiled using `icc` with *-O3*. The reference code was run on one of the Intel Xeon E5-2697 v2 2.7 GHz processors of the host node. We used a nearest neighbour list of 20 for the reference runs. Timings for the reference code were averaged over five runs of ~ 500 iterations for each problem instance.

A. MMAS Parameters and Problem Instances

We used $\alpha = 1$, $\beta = 2$ and $\rho = 0.5$. In all cases, the number of ants m was set equal to n, the number of vertices in the problem instance. The pheromone trail is updated by the iteration-best ant. We ran experiments using five values for the size of the nearest neighbor list – 20, 40, 70, 100, and n (which is equivalent to no nearest-neighbor list). We used the same five TSP instances from the TSPLIB [23] library as solved in[21] (`lin318, pcb442, rat783, pr1002, pr2392`), in order to allow for a direct comparison of execution times. For each problem instance and nearest-neighbor list size, we ran an ensemble of 16 runs, with 512 iterations per run.

We now present comparative results in terms of both execution time and solution quality.

B. Execution Time

Figure 1 shows the average execution time per iteration for the two Xeon Phi implementations, compared to both the reference implementation on CPU, and times taken from [21]. The times from [21] were taken from their Figure 5a. Note that [21]'s results are comparable with the CPU implementation here. This is most likely due to the fact that the reference implementation uses the nearest-neighbour list optimization, which speeds up the calculation considerably. Also, [21] used the Ant System (AS) algorithm, which requires more work in the pheromone update stage than MMAS (as well as producing poorer quality solutions). vRoulette-1 and vRoulette-2 are very similar in execution time, with vRoulette-1 being slightly faster, and both are an order of magnitude faster than [21]'s implementation and the reference code with the nearest-neighbour list optimization.

C. Solution Quality

We define solution quality as the ratio of the length of the best tour found to the known optimum for the problem instance. Figure 2 shows the average solution quality from the ensembles of runs. The solution quality using vRoulette-2 is strongly dependent on the size of nearest-neighbor list – with no nearest-neighbor list the solution is significantly worse, especially on the larger instances. There is a much weaker dependence on the nearest-neighbor list when using vRoulette-1. In addition, the solution quality is consistently better with vRoulette-1 than with vRoulette-2, for all instances and all nearest-neighbor list sizes.

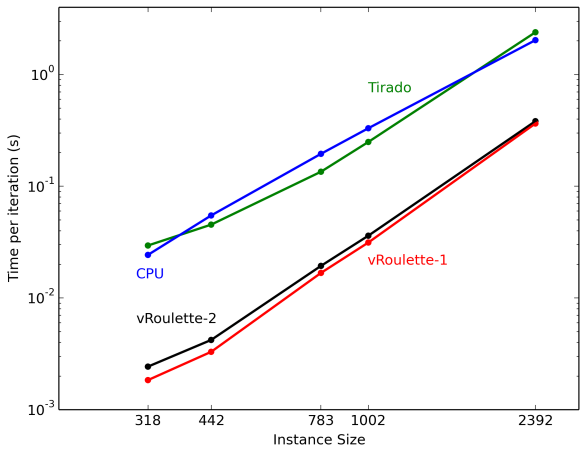

Fig. 1. Execution time per iteration for CPU and Xeon Phi implementations.

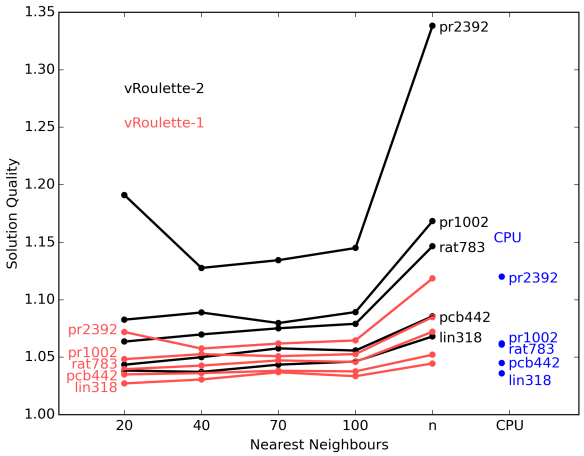

Fig. 2. Average solution quality after 512 iterations as functions of nearest-neighbor list size. Average solution quality from the CPU reference runs (with 20 nearest neighbours) are shown for comparison.

D. Xeon Phi Diagnositcs

Intel VTune Amplfier XE 2016 was used to gather performance diagnostics on runs of the code using vRoulette-1 and vRoulette-2. Table I shows data obtained from the profiling runs. The Level 1 Cache Hit Ratio gives the percentage of memory accesses which did not result in a Level 1 Cache miss ('in-flight' hits are not counted as misses in this diagnostic). Typically, the code achieves around 95%. Vectorization intensity is the ratio of data elements processed to vector instructions issued. For single precision float vectors, the upper limit to this measure is 16. Our code typically achieves > 10 on this measure. We derived the thread usage from the CPU usage per thread, as the mean CPU per thread divided by the maximum. Although this measure does not directly map to parallel efficiency (for example, in the perverse case of only one thread active at a time, with all threads consuming equal time, this measure would give 100% whereas the parallel efficiency would be very small), in this case it is a good guide to the efficiency as the timelines show very little idle time in the threads. Typically, we achieve over 90% on this measure. For comparison, [21] report L1 hit ratio of $\sim 80\%$ and vectorization intensity of ~ 10 for their best-performing method. The results show that our method is cache-friendly, and efficiently parallelized and vectorized.

TABLE I
XEON PHI PERFORMANCE DIAGNOSTICS - LEVEL 1 CACHE HIT RATIO (L1HR), VECTORIZATION INTENSITY (VI), THREAD USAGE (TU) AND SPEEDUP (COMPARED TO CPU REFERENCE CODE)

Algorithm	Instance	L1HR (%)	VI	TU (%)	Speedup
vRoulette-1	lin318	97.9	10.49	94.2	13.2
	pcb442	96.3	10.96	92.4	16.6
	rat783	94.6	11.27	94.6	11.6
	pr1002	94.3	11.37	93.9	10.5
	pr2392	92.5	11.56	87.8	5.6
vRoulette-2	lin318	98.5	9.09	93.2	10.0
	pcb442	97.5	10.07	90.0	13.0
	rat783	95.4	10.91	94.5	10.1
	pr1002	94.9	11.33	95.4	9.2
	pr2392	92.1	12.21	88.7	5.3

E. Discussion

vRoulette-1 and vRoulette-2 show very similar execution times, although vRoulette-1 is slightly faster on the smaller problems. This is possibly due to vRoulette-1 requiring less work in the serial phase; the main loops are comparable, so the effect of the serial part of the algorithm will be more pronounced when the trip count of the main loop is lower. The solution quality, and the sensitivity to nearest-neighbor list size, seems to be considerably better with vRoulette-1. However, this may be down to convergence speed, and it is possible that given more iterations, the vRoulette-2 code may converge to similar, or better, solutions although we note that the relatively high value of ρ used here should give rapid convergence (slightly better solutions may be obtained with lower ρ and more iterations [24]). Further work is required to investigate this effect fully, especially in the context of varying the MMAS parameters. Finally, we note that our implementation does not include a local search phase, which is commonly added to accelerate convergence in ACO [3]. Local search is added as an extra step in the tour construction phase, and hence could be parallelized in a similar manner to the remainder of the tour construction (i. e. by assigning ants to threads) although it is unlikely to benefit from vectorization. However, we note that in the ACOTSP code, local search typically adds less than 10% to the run time per iteration. Investigating the addition of local search to the Xeon Phi implementation is an area for future work.

V. Conclusion

In this paper, we have described an efficient implementation of the MMAS algorithm on the Xeon Phi platform. The key contributions are two novel vectorized procedures for selecting vertices in the tour construction phase; vRoulette-1, which is based on the IRoulette method of [11], and vRoulette-2, which is based on roulette wheel selection. In addition, we present a novel implementation of nearest-neighbour lists in a data-parallel ACO code. Experimental results show that our code makes efficient use of the parallel and vector capabilities of the hardware, and is an order of magnitude faster than both the previous best implementation in the literature and the reference CPU implementation. There is some evidence that vRoulette-1 gives better quality solutions, and is less sensitive to the size of the nearest-neighbor list. Further work is required to investigate this effect in more detail. The algorithm may be suitable for other many-core, SIMD architectures; in future work we will investigate its use on other platforms, in particular the next generation of Xeon Phi hardware (*Knight's Landing*). The use of the nearest neighbor list to accelerate the calculation, as in [16], and the implementation of local search are also areas for future work.

Acknowledgments

We acknowledge use of Hartree Centre resources in this work. The STFC Hartree Centre is a research collaboration in association with IBM providing High Performance Computing platforms funded by the UK's investment in e-Infrastructure. The Centre aims to develop and demonstrate next generation software, optimised to take advantage of the move towards exa-scale computing. HL thanks Stephen Pickles of the STFC Scientific Computing Department for useful discussions during the development of this work.

References

[1] M. Dorigo, "Optimization, learning and natural algorithms," Ph.D. dissertation, Politecnico di Milano, Italy, 1992.

[2] M. Dorigo and L. Gambardella, "Ant Colony System: a cooperative learning approach to the Traveling Salesman Problem," *Evolutionary Computation, IEEE Transactions on*, vol. 1, no. 1, pp. 53–66, 1997.

[3] M. Dorigo and T. Stützle, *Ant Colony Optimization*. Scituate, MA, USA: Bradford Company, 2004.

[4] T. Stützle and H. H. Hoos, "MAX-MIN ant system," *Future Gener. Comput. Syst.*, vol. 16, no. 9, pp. 889–914, Jun. 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=348599.348603

[5] NVIDIA. What is GPU computing? Last accessed 2016-11-7. [Online]. Available: http://www.nvidia.com/object/what-is-gpu-computing.html

[6] X. Cui, J. S. Charles, and T. Potok, "GPU enhanced parallel computing for large scale data clustering," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1736–1741, 2013.

[7] P. Pospichal, J. Jaros, and J. Schwarz, "Parallel genetic algorithm on the CUDA architecture," in *European Conference on the Applications of Evolutionary Computation*. Springer, 2010, pp. 442–451.

[8] E. Alba, G. Luque, and S. Nesmachnow, "Parallel metaheuristics: recent advances and new trends," *International Transactions in Operational Research*, vol. 20, no. 1, pp. 1–48, 2013.

[9] P. Krömer, J. Platoš, and V. Snášel, "Nature-inspired meta-heuristics on modern GPUs: state of the art and brief survey of selected algorithms," *International Journal of Parallel Programming*, vol. 42, no. 5, pp. 681–709, 2014.

[10] T. Van Luong, N. Melab, and E.-G. Talbi, "GPU computing for parallel local search metaheuristic algorithms," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 173–185, 2013.

[11] J. M. Cecilia, J. M. García, A. Nisbet, M. Amos, and M. Ujaldón, "Enhancing data parallelism for Ant Colony Optimization on GPUs," *J. Parallel Distrib. Comput.*, vol. 73, no. 1, pp. 42–51, 2013.

[12] L. Dawson and I. Stewart, "Improving Ant Colony Optimization performance on the GPU using CUDA," in *2013 IEEE Conference on Evolutionary Computation*, L. G. de la Fraga, Ed., vol. 1, Cancun, Mexico, 2013, pp. 1901–1908.

[13] NVIDIA. NVIDIA tesla GPU accelerators. Last accessed 2016-11-7. [Online]. Available: http://international.download.nvidia.com/pdf/kepler/TeslaK80-datasheet.pdf

[14] A. Duran and M. Klemm, "The intel many integrated core architecture," in *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, 2012, pp. 365–366.

[15] J. Fu, L. Lei, and G. 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*, 2010, pp. 260–264.

[16] L. Dawson and I. A. Stewart, *Candidate Set Parallelization Strategies for Ant Colony Optimization on the GPU*. Cham: Springer International Publishing, 2013, pp. 216–225. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-03859-9_18

[17] A. Uchida, Y. Ito, and K. Nakano, "An efficient GPU implementation of Ant Colony Optimization for the Traveling Salesman Problem," in *Networking and Computing (ICNC), 2012 Third International Conference on*, 2012, pp. 94–102.

[18] M. Dorigo and L. M. Gambardella, "Ant colonies for the Travelling Salesman Problem," *Biosystems*, vol. 43, no. 2, pp. 73 – 81, 1997. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0303264797017085

[19] J. M. Cecilia, J. M. Garcia, M. Ujaldon, A. Nisbet, and M. Amos, "Parallelization strategies for ant colony optimisation on gpus," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, May 2011, pp. 339–346.

[20] M. Sato, S. Tsutsui, N. Fujimoto, Y. Sato, and M. Namiki, "First results of performance comparisons on many-core processors in solving QAP with ACO: Kepler GPU versus xeon PHI," in *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO Comp '14. New York, NY, USA: ACM, 2014, pp. 1477–1478. [Online]. Available: http://doi.acm.org/10.1145/2598394.2602274

[21] F. Tirado, A. Urrutia, and R. J. Barrientos, "Using a coprocessor to solve the ant colony optimization algorithm," in *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*, Nov 2015, pp. 1–6.

[22] T. Stützle. ACOTSP v1.03. Last accessed 2016-11-7. [Online]. Available: http://iridia.ulb.ac.be/ mdorigo/ACO/downloads/ACOTSP-1.03.tgz

[23] G. Reinhelt. TSPLIB, url=http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/, note=Last accessed 2016-11-7.

[24] T. Stützle, M. López-Ibáñez, P. Pellegrini, M. Maur, M. Montes de Oca, M. Birattari, and M. Dorigo, *Parameter Adaptation in Ant Colony Optimization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 191–215. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21434-9_8