Future Generation Computer Systems 79 (2018) 473–487

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Parallel ant colony optimization on multi-core SIMD CPUs

Yi Zhou[a,b], Fazhi He[a,c,*], Neng Hou[a], Yimin Qiu[b]

[a] State Key Laboratory of Software Engineering, School of Computer Science, Wuhan University, Wuhan 430072, China
[b] School of Information Science and Engineering, Wuhan University of Science and Technology, Wuhan 430081, China
[c] State Key Laboratory of Digital Manufacturing Equipment and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China

HIGHLIGHTS

- A new parallel ACO model for multicore-SIMD CPU architecture is proposed.
- A new fitness proportionate selection approach for vector-level parallelism.
- A maximum speedup factor of 57.8x compared to the standard CPU sequential version.
- Our approach shows a promising performance on multicore-SIMD CPUs.

ARTICLE INFO

Article history:
Received 24 October 2016
Received in revised form 6 June 2017
Accepted 30 September 2017
Available online 14 October 2017

Keywords:
Ant colony optimization
SIMD
Multi-core CPU
TSP
Performance evaluation

ABSTRACT

Ant colony optimization (ACO) is a population-based metaheuristic for solving hard combinatorial optimization problems. Many studies are dedicated to accelerating ACO by parallel hardware, especially by graphics processing units (GPUs). However, due to the irregular (random) pattern of ACO algorithms in data access and control flow, the performance of GPU-based approaches is constrained by hardware limitations. CPU-based SIMD computing for ACOs is rarely investigated in previous literatures, and how well multicore-SIMD CPU-based parallel ACOs could perform remains unknown. In this paper, we present and evaluate a model of vector parallel ACO for multi-core SIMD CPU architecture. In the proposed model, each ant is mapped with a CPU core and the tour construction of each ant is accelerated by vector instructions. Furthermore, based on the model, we propose a new fitness proportionate selection approach named Vector-based Roulette Wheel (VRW) in the tour construction stage. In this approach, the fitness values are grouped into SIMD lanes and the prefix sum is computed in vector-parallel mode. The proposed algorithm is tested on standard TSP instances ranging from 198 to 4461 cities and shows a speedup factor of 57.8x compared to the single-threaded CPU counterpart. More significantly, we compare our approach with high performance GPU-based ACOs, and the results demonstrate the strong potential of CPU-based parallel ACOs.

1. Introduction

Solving optimization problems in the real world is complex and time consuming for central processing units (CPUs), especially for large-scale problems [1]. The modeling of these problems is also problem-dependent. Metaheuristics are efficient methods to obtain satisfactory resolution (an approximate optimum) in a reasonable time [2] and to provide a generic algorithm framework that can be applied in various problems with few modifications [3]. Ant Colony Optimization (ACO) is a population-based metaheuristic inspired by the social behavior of ants [4]. Artificial ants construct solutions independently and communicate with each other by a stigmergy mechanism [5]. This process is executed iteratively until a termination criterion is reached. The most significant feature of ACO is positive feedback that benefits from the pheromone left by ants that can guide the solution construction process. This metaheuristic has been used to solve NP-hard problems, such as the traveling salesman problem (TSP) and the quadratic assignment problem (QAP) [6,7].

In recent years, new types of hardware that deliver massive amounts of parallel processing power, such as Cell/BE, FPGA, and graphics processing units (GPUs), have become available. Among these types of hardware, GPUs are one of the most remarkable accelerators that could provide great computing power at a low cost. GPU-accelerated parallel ACO algorithms have become a topic

[*] Corresponding author at: State Key Laboratory of Software Engineering, School of Computer Science, Wuhan University, Wuhan 430072, China.
E-mail address: fzhe@whu.edu.cn (F. He).

https://doi.org/10.1016/j.future.2017.09.073

of considerable interest [8–11]. The computational efficiency of the ACO algorithms is significantly improved by the use of GPUs.

However, existing GPU-based parallel ACOs face at least two problems. Firstly, the GPU-based ACOs show moderate speedups against CPU-based sequential ACOs when considering that the theoretical peak performance of GPUs is considerably higher than that of a single CPU thread. Secondly, the GPU-based ACOs are highly depended on the on-chip memory which is a very limited resource [9,10], thus the problem size to be solved is constrained.

Recent CPU architectures have significant modifications leading to high-performance computing capabilities. First, due to instruction level parallelism (ILP) walls and frequency walls, the performance of CPUs is enhanced by the multi-core scheme. Second, modern CPU architecture increases stress on single instruction multiple data (SIMD) vector instruction extensions [12]. Regular algorithms have taken advantage of current CPU architecture, such as sorting [13], stencil computation [14] and matrix multiplication [15].

Unfortunately, little attention has been paid to the modeling of parallel ACO to be adapted to CPU-based SIMD architecture. Although the existing GPU-based ACO model may be theoretically applicable on the multicore-SIMD CPU architecture, the performance is hardly guaranteed for different characteristics of the parallel architectures [16], computational features of the ACO algorithm [17] and discrepancies of the compiler optimizations [18]. The optimized implementation of multicore-SIMD CPU version ACO is still a technique challenge.

The oral communication [19] in META'14 proposed an approach of vectorization for ACO algorithm on the GPU architecture. But whether it is suitable for the multicore-SIMD CPU architecture is unknown and how well CPU-based parallel ACOs could perform when compared with GPU-based methods remains unclear.

This paper presents an idea to accelerate a fully developed parallel ACO for the TSP on multi-core SIMD CPUs. In our algorithm, critical stages, tour construction and pheromone update are parallelized and optimized. Our major contributions are as follows:

1. To the best of our knowledge, this is the first parallel ACO algorithm exploiting both task-level and vector-level parallelism on multicore-SIMD CPUs.
2. We present a new model of vector parallel ACO on multicore-SIMD CPU, which extends the general task parallel model widely being used.
3. In the tour construction stage, we design a new fitness proportionate selection approach named Vector-base Roulette Wheel (VRW) to improve vector-level parallelism on multicore-SIMD CPUs.
4. We evaluate our algorithm with the standard TSPLIB problems ranging from 198 to 4461 cities and obtain a maximum speedup factor of 57.8x compared to the standard CPU sequential version.
5. More importantly, we compare our algorithm with previous high performance GPU-based data parallel ACOs that are already proposed in literature [8–11]. The results indicate the strong potential of multicore-SIMD CPU-based parallel ACOs.

The remainder of this paper is organized as follows. We first briefly introduce the ACO for the TSP. Related studies are discussed in Section 2. Then, we present our approaches of parallel ACO on multicore-SIMD CPUs in Section 3. Our experimental methodology is outlined in Section 4, and we describe the performance evaluation of our algorithm in this section. Finally, we summarize our findings and conclude with suggestions for future work.

2. Background

2.1. Ant colony optimization for the TSP

The TSP is an NP-hard problem in combinatorial optimization and is important in operations research and theoretical computer science. The objective of the TSP is to find a minimum-weight Hamilton cycle in a complete weighted directed graph $G = (V, A, d)$, where $V = 1, 2, \dots, n$ is a set of vertices (cities), $A = \{(i, j)|(i, j)\epsilon V \times V\}$ is the set of arcs (paths), and $d : A \rightarrow \mathbb{N}$ is a function assigning a weight or distance (positive integer) d_{ij} to every arc (i, j).

The TSP is solved by Dorigo et al. [20] using the ACO, which uses many artificial ants performing parallel searches on a graph. Each ant moves independently on the graph until it travels to all of the cities on the graph. This process is typically called the tour construction phase, and then, the process provides a solution. To obtain a better solution, each ant strengthens the pheromones on its path to guide other ants. The ants stochastically select the next city to visit based on heuristic information obtained from inter-city distances and the net pheromone trail. However, a process of pheromone evaporation is also applied to avoid falling into a local optimum solution.

Algorithm 1 presents the sequential code framework for the ACO. First, all data structures for the TSP problem, including visited city list and city distance, are initialized. Next, in the tour construction stage, m ants travel to n cities sequentially. Then, each tour could be improved by a local search process. In the pheromone update stage, each ant deposits pheromone on n paths of its travel separately one by one. These stages are performed iteratively until the termination criterion is reached.

Algorithm 1 Sequential AS Pseudo-code for the TSP

```
1: InitializeData();
2: repeat
3:   TourConstruction();
4:   LocalSearch(); // Optional
5:   PheromoneUpdate();
6: until Termination_criterion()
7: end
```

The Max–Min Ant System (MMAS) [21] is commonly recognized as one of the best performing ACO algorithms at the present time. The main mechanisms and memory structures of MMAS are derived from AS. There are two key improvements over AS In MMAS. (1) To strengthen exploitation of the search history, MMAS only allows one ant (global best or iteration best) to update the pheromone trails after each iteration; (2) to avoid premature convergence of the search, a pheromone strength control mechanism is applied to MMAS in which each pheromone trail is limited between τ_{min} and τ_{max}. For better readability, we refer the reader to [20,21] containing detailed information on this subject.

2.2. Multi-core SIMD CPU and OpenCL Programming Model

For the purpose of understanding our work, a brief description of multi-core SIMD CPU architecture and its parallel programming model is required. CPUs with x86 architecture support a set of SIMD instructions called Streaming SIMD Extensions (SSEs) in Pentium III by Intel. With these instructions, the system can perform the operations on the vector data elements in many ways, including operating on many elements simultaneously. This flexibility lets vector designs use slow but wide execution units to achieve high performance at low power [12]. In 2011, Intel released Sandy Bridge micro-architecture that supports a new set of instructions known as Advanced Vector eXtensions (AVXs) [22], an enhanced version of the SSE instructions. In the AVX instruction set, the width of the SIMD registers is extended from 128-bit to 256-bit. Hence, an AVX instruction set is able to process twice the amount of data that the SSE instruction set is able to process.

Open Computing Language (OpenCL) is an open standard for general-purpose parallel programming across CPUs, GPUs and other processors, providing software developers with portable and efficient access to the power of these heterogeneous processing platforms. In OpenCL, multi-core CPUs and GPUs are defined in a unified model, and they are all considered as devices. Fig. 1 illustrates that an OpenCL device is divided into one or more compute units (CUs), which are further divided into one or more processing elements (PEs). Computations on a device occur within the processing elements. The global memory and constant memory together represent the off-chip memory of compute devices. Generally, a CPU in OpenCL architecture could also be called a host. The OpenCL application submits commands from the host to execute computations on the PEs within a device.

At a high level, multi-core SIMD CPUs share similarities with GPUs. Therefore, CPUs and GPUs share the same terms in OpenCL. For efficient comprehension of our work and the related GPU-based parallel ACOs, we summarize these terms in Table 1.

2.3. Related works

Parallelism is as an effective method for metaheuristics to reduce the exploration time and improve the quality of the solutions provided [24,25]. Moreover, ACOs are based on swarm intelligence theory [26], that is, a collection of identical ants work collaboratively to search for an optimal solution. ACOs are also typical bioinspired algorithms for that they are based on the natural process of ant foraging. Because such behaviors of ants are well suited to parallel or distributed processing, many studies are dedicated to improving the performance by parallel computing technology [27]. In terms of efficiency, Bullnheimer et al. [28] accelerate the ACO algorithm by distributing ants to processing elements while they conform to the original algorithm. In terms of improving quality, Stützle [29] has researched the strategy of using parallel independent runs. In Stützle's work, multiple ant colonies could be executing on processors and communicate by the timely exchange of good solutions or pheromone trials. These approaches are often implemented on processors with shared memory and distributed through a message passing interface (MPI).

From the architecture point of view, we review literatures on parallel ACOs by two types of parallel processor: multi-core and many-core.

2.3.1. Multi-core CPU-based parallel ACOs

Bui et al. [30] present a parallel ant-based optimization (a variant of ACO) algorithm on an eight-core CPU using shared memory strategies. Their approach is up to six times faster than the sequential approach for the Max-Clique problem. Tsutsui and Fujimoto [31] propose a parallel ACO and evaluate this parallel ACO on a four-core CPU. They compare three parallel ACO models and conclude that the asynchronous parallel model is the most promising model, with speedups ranging from 2.7 to 5.1. Li et al. [32] implement a parallel ACO on a dual-core CPU using a Thread Building Block (TBB). They achieve a speedup of 1.72 in solving TSP instances with city numbers up to 500. With similar ideas, Gao et al. [33] apply two multi-threading implementations to solve the Target Assignment Problem with ACO. OpenMP and TBB are selected in their experimental analysis. They obtain nearly linear speedups with 100 targets and prove that the approach using OpenMP outperforms the approach using TBB. Zhang et al. [34] use the Parallel Pattern Library to implement an Ant System algorithm. Their experiments are performed on a quad-core CPU and demonstrate that the overall efficiency of the algorithm is improved.

More recently, Kugu et al. [35] aim to present the performance increase of parallel ACOs on multi-core CPUs. They implement the Ant Colony System and the Rank-Based Ant System using the Java thread programming approach. They report speedups ranging from 6.11 to 13.65 in an 8-thread configuration. Zhang et al. [36] propose a parallel ant system using OpenMP to make full use of the computing power of a multi-core CPU. Their algorithm is based on the multiple ant colony approach for solving the TSP. Their experimental results show that both the efficiency and solution quality are improved by their approach.

This type of parallel ACOs is based on the task parallel ACO model (TPAM). The main concept of the TPAM is mapping each ant to a thread. Because of the inherent parallel nature of the ACO, threads could execute concurrently in the tour construction. This concurrent execution could be implemented on a multi-core CPU by using multi-threading technology. However, the performance of this approach is constrained by the number of CPU cores, which is typically 4 or 8, resulting in the speedup values that do not typically surpass 10.

2.3.2. Many-core GPU-based parallel ACOs

The architecture of GPUs is considered a typical variant of SIMD architecture in a recent classification [12]. The existing parallel approaches on GPUs are also related to our work. Because GPUs could deliver higher peak computational throughput than multi-core CPUs and are inexpensive, researchers are more interested in parallelizing the ACO algorithm on GPUs than on multi-core CPUs.

Researching GPU-based Parallel ACOs started as the emergence of the programmable shader. Catala et al. [37] and Wang et al. [38] research the GPU implementation of ACO using vertex processors and fragment processors. The ecosystem of CUDA has been growing rapidly since 2009 with the development of general-purpose computing on graphics processing units (GPGPUs) and with the ease of programming and the need for computing power. Since then, most ACO algorithms have been developed with CUDA [39–42].

More systematic studies on GPU-based ACO have been published since 2013. Delévacq et al. [9] propose effective parallelization strategies for the ACO metaheuristic on a GPU. They associate each ant with a thread-block, and parallelism is exploited through the computation of the state transition rule. They report speedups as high as 19.47 with a solution quality similar to the solution quality of the original sequential implementation. Their results also show low speedups of up to only 5.84 by implementing TPAM on GPU. Cecilia et al. [10,43] extend the taxonomy *hardware-parallel ACOs* proposed by Pedemonte et al. [27]. In their paper, they first present the concept of the task-based parallel model and the data-based parallel model for ACOs and implement them using CUDA. Their data parallel approach is based on the idea that a thread-block represents a queen ant, and each thread within a block is associated with cities. They provide experimental results to prove that the data-based parallel model is better fitted to GPUs than the task-based parallel model.

Also more addressing implementation strategies for higher efficiency, Uchida et al. [8] implement ACO for the TSP using CUDA with the consideration of many programming issues in the GPU architecture, including coalesced access of global memory and shared memory bank conflicts. In addition, these authors present a method called Stochastic Trial to avoid the prefix sum calculation as much as possible. Dawson et al. [11] extend the data parallel approach of Cecilia et al. [10] and Delévacq et al. [9]. They propose a new parallel implementation of roulette wheel selection called Double-Spin Roulette that significantly reduces the running time of tour construction.

Recently, Llanes et al. [44] present a hybrid parallel ACO on heterogeneous clusters. This approach mainly accelerates ACO by

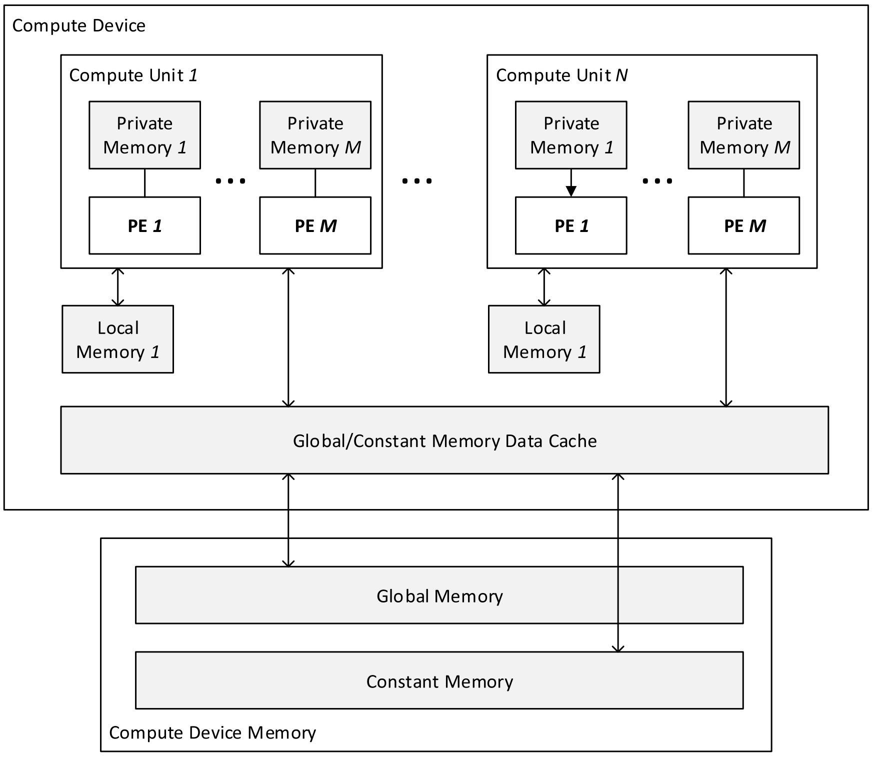

Fig. 1. OpenCL compute device architecture [23].

Table 1
Conversion from terms used in OpenCL to official NVIDIA/CUDA and Intel Haswell.

OpenCL term	Intel Haswell term	NVIDIA/CUDA term
CU	Logical core/Vector unit	Streaming multi-processor (SM)
PE	Vector lane	Streaming processor (SP)
Global memory	L3 cache/host memory	GPU memory
Local memory	L1/L2 cache	Shared memory
Private memory	Registers	Local memory
Work-group	Work-group	Thread block
Work-item	Work-item	CUDA thread

GPUs, and CPUs are used as performance monitoring via OpenMP threads. Skinderowicz [45] propose a parallel Ant Colony System (ACS) for GPUs, which achieves a speedup up to 24.29x.

This type of parallel ACOs is based on the data parallel ACO model (DPAM). The main concept of the DPAM is to associate each ant with a group of threads with shared memory, for example, a CUDA thread block. Furthermore, by assigning each thread to one or more cities, threads in a block could compute the state transition rule cooperatively, thus enhancing data-level parallelism. Additionally, because the on-chip shared memory is able to store the data structures of an ant, the latency of uncoalesced memory access manner in tour construction is decreased.

2.4. Problems and motivations

The previous works intensively investigate parallel strategies of implementation of ACOs on GPUs and demonstrate the more promising efficiency on GPUs than on multi-core CPUs. However, there are still several major problems as follow.

1. Problem 1. The theoretical peak performance of a GPU is several hundred times higher than that of a CPU of similar price and/or similar level. However, the GPU-based ACO speedups are only reach about 30x [8–10]. This demonstrates that the hardware utilization ratio of GPU-based ACO is low. The reason mainly comes from the unregular (random) and divergence nature of ACO algorithm. Therefore, multi-core SIMD CPU is a potential solution of the problem, which is worthy of exploration.
2. Problem 2. The existing CPU-based parallel ACOs is commonly accelerated by multi-core in literatures [31,35,36] with task-parallel model, and the CPU vector units are ignored. Therefore, the vector-parallel approach of ACO on SIMD CPU should be investigated. Whether the vector-parallel approach is suitable for the multicore-SIMD CPU architecture is unknown and how well CPU-based parallel ACOs could perform when compared with GPU-based methods remains unclear.
3. Problem 3. Furthermore, the performance of GPU-based parallel ACOs is limited by shared-memory resources [9], while CPU-based parallel ACOs have no such limitation and could potentially deal with larger scale TSP problems.

In order to address these problems, the motivations of our research on CPU-based ACO are as follow.

1. Aiming at problem 1, although we have done a previous work on the optimization of GPU-based data-parallel ACO [46], the speedups are still below 50x with our efforts. Therefore, a new parallel ACO model should be investigated, which can exploit both task-parallelism and data-parallelism. And the model should also be suitable for multi-core SIMD CPU to achieve better hardware utilization.

2. Aiming at problem 2, a new parallel method should be designed for the most time-consuming part of the ACO algorithm, that is the roulette-wheel selection process. We propose an approach of vector-level parallelism for the problem.
3. Aiming at problem 3, the utilization of CPU memory architecture is an additional benefit of the new parallel ACO model.

3. Parallel ACOs on multi-core SIMD CPUs

In this section, we review the previous parallel designs for the ACO algorithm and then present our approaches orienting multicore SIMD CPUs. For tour construction, based on our analysis on two previous GPU-based parallel approaches, we present our CPU-based SIMD algorithm. For the pheromone update, we introduce our implementation.

3.1. Our proposed parallel ACO approaches

The DPAM is suitable for GPU architecture, but we should reconsider it for a CPU-based SIMD. There are two issues of adapting DPAM to multicore-SIMD CPUs. The first issue is the data synchronization overhead. In the data parallel approach, because the ant data are shared between threads via shared memory, these threads communicate through barrier synchronization points, resulting in cross-module round trips that degrade performance [47]. Another issue is that the explicit use of local memory introduces a moderate cost [47] because for CPU, all memory objects are cached by hardware. In this section, we present several different multicore-SIMD CPU designs for AS and MMAS, as applied to the TSP.

3.1.1. The vector parallel ACO models

We extend the TPAM to exploit vector-level parallelism in two schemes named as Vectorized Task Parallel ACO Model (VTPAM) and Vector Enhanced Task Parallel ACO Model (VETPAM) (Fig. 2). The models are described as follows:

VTPAM (Fig. 2(a)): In this design, v ants (depending on the vector instruction width) are grouped to run together. This group of working ants perform the *vectorized task* (VT). Each VT is associated with a CU to exploit task level parallelism. Because each task in a VT performs the same scalar instructions, they could be packed into vector instructions, which exploit data-level parallelism on the CPU-based SIMD architecture. With vector instructions, the data of v ants could be processed simultaneously. Nevertheless, due to the fact that the ACO algorithm diverges in the RW selection process, the vectorization module of the compiler has to execute both code path of an if-statement (or loop-statement) by masks (predicates). Thus, the utilization of SIMD units could be reduced dramatically because more instructions are generated.

VETPAM (Fig. 2(b)): In this model, each ant is mapped to a thread, which is the same strategy as in TPAM. We use vectors to accelerate the tour construction stage. The ant data is divided into vector lanes to enhance vector-level parallelism. And the diverged code is still executing sequentially to avoid masking-overhead.

In addition, the task-based coding style is more developer-friendly than the data-based coding style, because the major logical flow of the program is close to the sequential version. The vector-based approach could have three advantages over TPAM. First, we could exploit the parallelism of vector registers and vector function units. Second, since the ant travel is guided by a stochastic process, the memory access pattern is unpredictable. Thus, the CPU cache is a better fit for this type of random memory access. Third, the reuse of data is transparently managed by hardware. Because the cores in a multicore CPU share the L3 cache, the running ants could benefit from hitting data in the cache. To ensure the effectiveness of these advantages, we present the performance evaluation of the proposed models on a multicore-SIMD CPU in Section 4.

3.1.2. System overview

A multicore-SIMD CPU based ACO is illustrated in Fig. 3, which divides different operations in the ACO algorithm into several separate kernels for encapsulation. The host allocates memory region for ACO structures and pass the pointers of these structures to the device through kernel parameters, so data transfer is not needed between them. Function of the kernels are described as follow:

Kernel(RNG): generate random numbers between 0 and 1 for the roulette wheel selection procedure. This is a common functional kernel for data preparation.

Kernel(CI): the *choiceinfo* matrix stores heuristic information, which is calculated by the heuristic value times pheromone for each edge ($[\tau_{i,j}]^{\alpha}[\eta_{i,j}]^{\beta}$). We associate each *choiceinfo* entry with a work-item to execute on a process element of an SIMD CPU.

Kernel(TC): this kernel contains an iterative process, in which each ant stochastically move to the next city according to the state transition rule until all the cities are visited. Each ant is mapped to a work-item or a work-group (coarse-grain or fine-grain strategy). This kernel requires random numbers generated in RNG kernel.

Kernel(LS): performs local search for each ant's tour to enhance exploitation. This kernel may require a lot of computation time, however, is an important part of the state-of-the-art ACO algorithm which augment solution quality largely. This kernel is optional in ACO.

Kernel(PU): performs pheromone evaporation and pheromone deposit. We use atomic operation to solve the conflicts when several ants visit a same edge.

Kernel(DC): reset the tours of ants for the next iteration.

The kernels execute iteratively until termination criterion is reached. The typical stop conditions are max execution time, max iterations and convergence criterion. In this work, we focus on the computational features of the ACO, so we choose max iterations for the convenience of performance comparison with existing parallel ACOs.

3.1.3. Memory management strategies

In this section, we propose our memory management strategies for CPU-based SIMD architecture. Additionally, the methods for the GPU are given for comparison. There are two types of data structures in the ACO. The first type is the city-related data, which contain the distance matrix and pheromone matrix at $n * n$ in size, where n is the number of cities. The data size increases dramatically with an increasing number of cities. It is suitable to write on global memory because of the limitation on the local memory size. The second type of data is the ant data. Each ant has its own private data to record the path and visitations or temporarily save the probability array. These data have a size of $m * n$, where m is the number of ants.

Table 2 summarizes the ACO data distribution on GPU and CPU memory. On GPUs, an ant data structure that is used frequently in the tour construction could be saved on local memory to reduce the overhead on data accessing to global memory. Typically, the ant data structure is a one-dimensional array of size n, such as the tabu list, city indices, city visitations and transfer probabilities. On CPUs, all of the data structures except temporary variables could be stored in the global memory. The accessing of these data structures could be cached by hardware. Thus, we use the structure of arrays (SOA) pattern, which is typically more cache-friendly than the array of structures (AOS) [47].

On GPUs, the host initializes all of the data structures and then copies them to the global memory. These data will be read to local memory as a user-managed cache. On CPUs, because vector units share the same memory space with scalar units, they could use the host memory pointer directly.

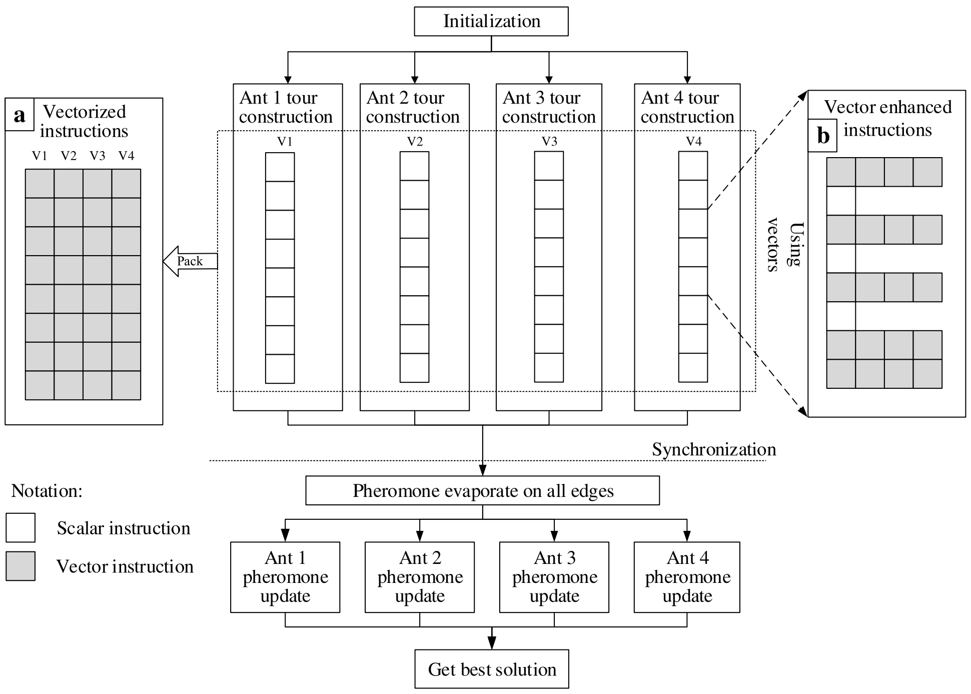

Fig. 2. The task-based parallel ACO model and its extensions: VTPAM (a) and VETPAM (b).

Table 2
Data distribution on GPU and CPU memory.

ACO data structure	GPU memory	CPU memory	Data size
City distance	Global	Global	$n*n$
Pheromone information	Global	Global	$n*n$
Parameters	Constant	Constant	–
Tabu list	Global	Global	$m*n$
City visitation	Global	Global	$m*n$
Tabu list (per ant)	Local	Not need	n
City visitation (per ant)	Local	Not need	n
Probability array (per ant)	Local	Not need	n
Temporary variable	Private	Private	–

3.2. The tour construction stage

3.2.1. The basic tour construction kernel

Algorithm 2 shows the pseudo code of the tour construction kernel based on the TPAM. At the start Algorithm 2, the ant identity i is set to the global work-item identity and later used to access data of the ant i. The main process of Algorithm 2 (from lines 3 to 25) is the same as the original sequential algorithm.

In this kernel, each ant step is divided into two processes. First, each ant loads the heuristic information for visiting city j from *current_city* (see Algorithm 2, lines 4–13). The *choice_info* matrix is calculated in kernel CI, as explained before. Second, base on the heuristic information, each ant chooses next city to move to using RW selection (see Algorithm 2, lines 15–24).

3.2.2. Compaction-based tour construction

The idea of this approach is to bypass the check of the visited cities in the tour construction. Thus, the number of cities involved for RW selection is compacted. We use a city index array to replace the tabu list and city visitation array in the basic tour construction algorithm.

Fig. 4 depicts our design. The city index array is initialized to an ordered sequence of city identities, which is used to access the city related heuristic information (choice_info in Algorithm 2). To be clearly, we show several steps in an ant's tour construction (Fig. 4).

Algorithm 2 The pseudo code of the basic task parallel tour construction kernel.

```
1: for each $i := 0 \to m-1$ parallel do
2:   ant[i].visited := {false};
3:   for $step := 1 \to n-1$ do
4:     current_city:=ant[i].tabu[step-1];
5:     sum_prob:=0;
6:     for $j := 0 \to n-1$ do
7:       if ant[i].visited[j] then
8:         ant[i].prob[j]:=0
9:       else
10:        probability:=choice_info[current_city][j];
11:        ant[i].prob[j]:=probability;
12:        sum_prob+=probability;
13:      end if
14:    end for
       {Roulette Wheel Selection Process}
15:    rand:=random(0, sum_prob);
16:    probability:=0;
17:    for $k := 0 \to n-1$ do
18:      probability+=ant[i].prob[k];
19:      if $probability >= rand$ then
20:        break;
21:      end if
22:    end for
23:    ant[i].tabu[step]:=k;
24:    ant[i].visited[k]:=true;
25:  end for
26: end for
27: end
```

The private variable *cursor* represents an index of the candidate city for selection, and increase by one after the ant moves to the next city. In the first step, all n cities are collected for selection computation in RW. Then, the selected city index is swapped with the candidate city index. In step 2, the RW process only needs $n-1$

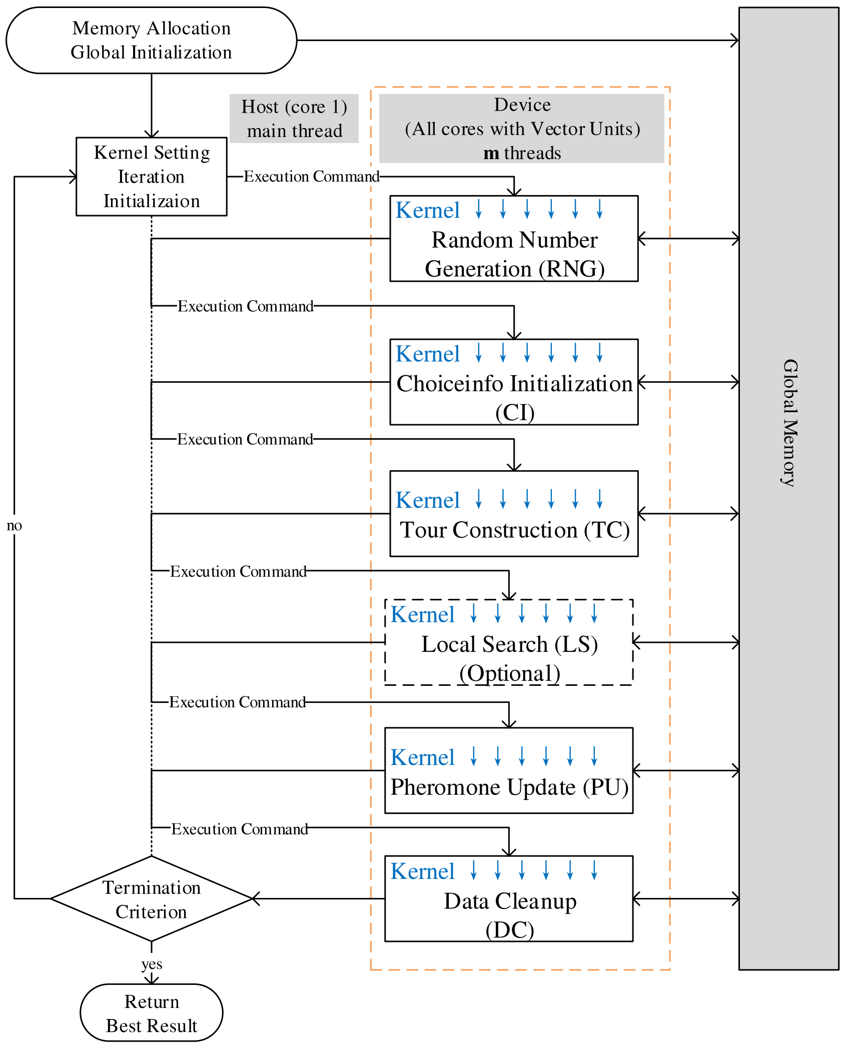

Fig. 3. Flow chart of a multicore-SIMD CPU based ACO.

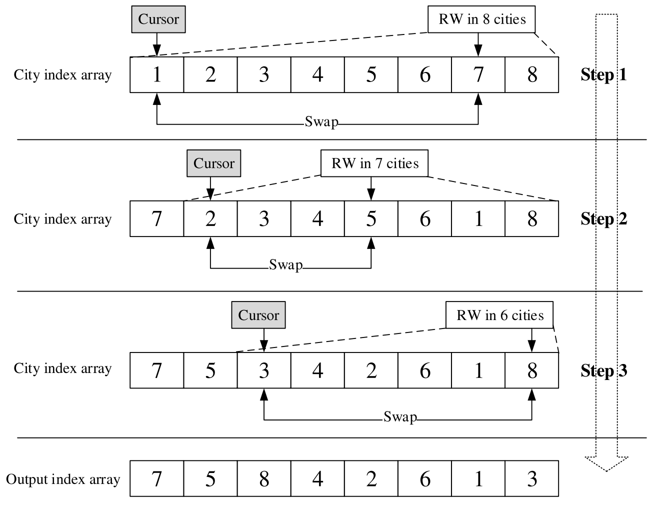

Fig. 4. Compaction-based tour construction.

cities for computation, because the first element of the city index array is a visited city. In step 3, the number of cities for computation also decrease by one as illustrated in Fig. 4. Consequently, the total number of cities processed in RW is $\Sigma_{i=1}^{n} i = \frac{n(n+1)}{2}$, which is approximately half of the basic version.

3.2.3. *Vector-based roulette wheel process*

In ACO algorithms, the roulette wheel selection is a key process for parallelization. This process is generally recognized as the most time consuming part [8,10,11]. It is trivial to implement on a single-threaded machine. However, efficiently parallelizing RW on many core machines requires a considerable effort, for the reason that RW is highly stochastic and contains divergence in control flow. Several literatures have proposed parallel RW approaches on GPU implementation, but it is still rarely investigated on multicore-SIMD CPUs.

Our approach is based on the following definitions:

Definition 1. The addition scan operation takes an input array a of n elements

$[a_0, a_1, \ldots, a_{n-1}]$,

and outputs

$[s_0, s_1, \ldots, s_{n-1}]$,

where $s_i = \sum_{j=0}^{i} a_j$. The scan operator defined here is an inclusive scan.

Definition 2. The partial scan takes a sequence of element from the array a whose index range is from x to y where $x \geq 0$ and $x < y < n$. It outputs an array $s_{(x,y)}$ in which $s_{(x,y,i)} = \sum_{j=x}^{i} a_j$, where $x \leq i \leq y$.

Corollary 1. *The partial scanned array could be processed to scanned array by*

$$s_{(x,y,i)} + s_{x-1} = \sum_{j=x}^{i} a_j + \sum_{j=0}^{x-1} a_j = \sum_{j=0}^{i} a_j = s_i$$

Based on the scan operator, the standard RW selection on the array a can be described as follows. Given r a random floating point value between 0 and s_{n-1}. Then, we can locate an output element using r, where $s_k \geq r$ and $s_{k-1} < r$ (if $k > 0$). The scan-base RW implementation may perform better than the original implementation in Algorithm 2 for that there is no floating point computation in the locating step. We make a validation in Section 4.

We design a new RW approach named Vector-base Roulette Wheel (VRW) to exploit vector-level parallelism on multicore-SIMD CPU. This process is demonstrated in Fig. 5. For the convenience of understanding our approach, the probability array is shown in a 2D view with a width of eight. We divide the probability array into eight lanes. In each lane, the probability data is then scanned sequentially to produce a partial scanned array. The computation of the eight lanes could be combined to vector instructions and executes in streams. More efficiently, the sum of each lane is kept in an accumulative vector reside in register file to improve memory locality.

After the vectorized partial scan phase, the accumulative vector is scanned sequentially. Now, each element in the accumulative vector represents the total sum of its previous lanes including itself. By multiplying the last element of the cumulative vector with a uniformly distributed floating point value between 0.0 and 1.0, we get a random value r. Then, the VRW locates a city index in two phases. In the first phase, we could directly select the lane number *slane* by r using the elements in the accumulative vector. In the second phase, we could locate the city index in the selected lane. Since the probability values in the selected lane is only partial scanned when $slane > 1$, we could modify the elements of the lane to scanned values according to Corollary 1 by adding the value stored in the lane $slane - 1$. Finally, we could search the probability array starting from index *slane*, and in a stride size of eight until the city index is located.

3.2.4. *Adapting our approaches to max–min ant system*

Since MMAS has been made great improvements over AS, it is important to adapt our approaches to MMAS. As we have mentioned above, the main structure of the MMAS algorithm is almost the same as the AS algorithm. The pheromone update strategy could be modified effortless to extend AS to MMAS. Therefore, we focus on the strategies for extending our approach to MMAS in the tour construction stage.

MMAS uses a heuristic approach named nearest neighbor in the tour construction stage, to speedup sequential computation and improve solution quality. For each city, a candidate list is initialized with a fixed size (cl) of its nearest neighbors sorted in ascending order. An ant chooses a city whose index is selected exclusively from the candidate list. Until all candidate cities are visited, the ant is able to take a city outside the list. The candidate list is similar to the city index array in the compaction-based approach. In both of these structures, city index is explicitly defined so that the accumulative vector cannot directly loads probability values in sequence. Therefore, we first gather a number of probability items by the city indexes in the candidate list, and pack them into a vector (Fig. 6). Then, the vector could be processed by the VRW.

A competitive parallel ACO has to be efficient both in execution speed and in solution quality. So integrating a local search process is required. According to the local search code provide by Dorigo and Stützle [4], we implement a 3-opt local search in OpenCL to augment solution quality of our algorithm. Our parallel implementation of the local search part is based on TPAM for the reason that the sequential speedup mechanisms *fixed radius search* and *don't look bit* [48] are rather complex in control flow.

3.3. *The pheromone update stage*

Algorithm 3 presents the pheromone update kernel. The global synchronization method is needed to avoid data writing conflicts because there is no mechanism for synchronization between work-groups [23]. In practice, this kernel is decomposed into two kernels that are called sequentially by the host.

In the first kernel, every path evaporates pheromones independently, as configured by the evaporation rate. This process could be fully parallelized by creating threads with the same number of paths. Each thread reads the pheromone value of its corresponding path to private memory to calculate the new pheromone value and then writes it back to global memory. In the second kernel, each ant deposits pheromone along their traveled paths. Unlike the prior step, two or more ants could walk along the same path. Global memory writing conflicts occur in this situation. However, CPUs support atomic operations, such as the atomic add function, to guarantee that the pheromone value could be updated by several ants in parallel. The atomic add function for float is not directly supported in OpenCL. Thus, we implemented the atomic float add function using atomic_cmpxchg() (compare and exchange). We refer to the method described in [49], which is the CUDA version.

Algorithm 3 Pheromone update kernel

```
1: for each $i := 0 \to n * n - 1$ parallel do
2:   Pheromone evaporate
3: end for
4: Global synchronization
5: for each $j := 0 \to m - 1$ parallel do
6:   for each $k := 0 \to n - 1$ parallel do
7:     Ant $j$ deposit pheromone on its path $k$ using atomic add operation
8:   end for
9: end for
10: end
```

4. Performance evaluation

We have experimented with our algorithm implemented using C++ and OpenCL on a PC with an Intel Core(TM) i7-5820k that has 6 cores and 16 GB of DDR4-2133 memory (dual channel). We use a high-end graphic device, a NVIDIA GTX780 with 3 GB of video memory and 2304 CUDA cores for comparison. Table 3 provides the detailed hardware specifications.

Our experiment uses a standard set of benchmark instances from the TSPLIB library [50]. We use single-precision float-point numbers for the pheromone information and city distances. On the algorithmic level, conforming to the experimental principles adopted by Dorigo et al. [20], our key parameters are configured as follows: $m = n$ (n being the number of cities), $\alpha = 1$, $\beta = 2$, and $\rho = 0.5$. On the device level, the work-group size is set to 16, which is a multiple of vector lanes as recommended in [47]. The number of total work-items is calculated by $\lceil \frac{m}{16} \rceil * 16$. The target instruction set architecture of the Intel OpenCL kernel builder is set to AVX2.

For the AS algorithm, Each TSP instance is tested in a single iteration and averaged over 100 independent runs, which is the

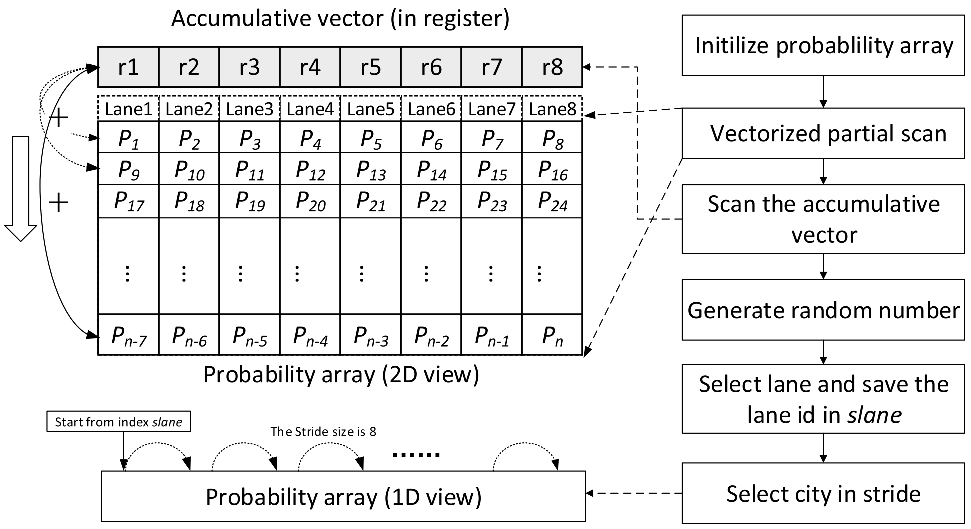
Accumulative vector (in register)
r1 r2 r3 r4 r5 r6 r7 r8
Lane1 Lane2 Lane3 Lane4 Lane5 Lane6 Lane7 Lane8
P1 ... Pn
Probability array (2D view)
Start from index slane
The Stride size is 8
Probability array (1D view)
Initilize probablility array
Vectorized partial scan
Scan the accumulative vector
Generate random number
Select lane and save the lane id in slane
Select city in stride

Fig. 5. Vector parallelism approach on the roulette wheel selection process.

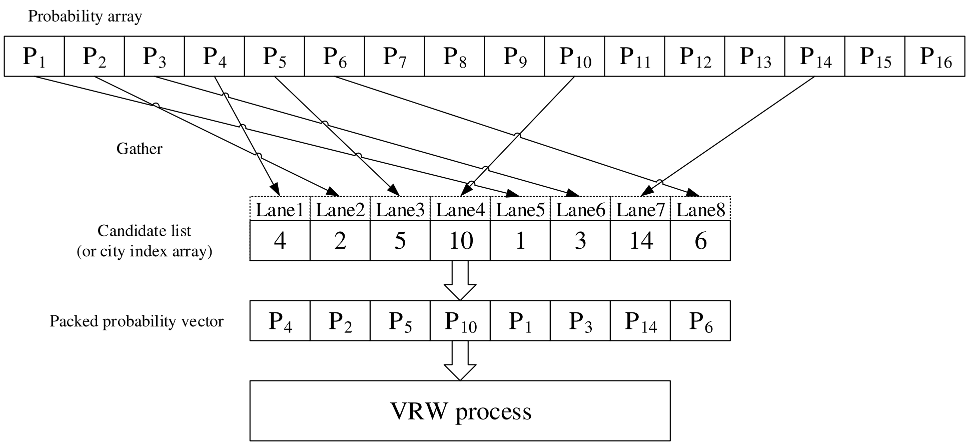
Probability array
P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11 P12 P13 P14 P15 P16
Gather
Candidate list (or city index array)
Lane1 Lane2 Lane3 Lane4 Lane5 Lane6 Lane7 Lane8
4 2 5 10 1 3 14 6
Packed probability vector
P4 P2 P5 P10 P1 P3 P14 P6
VRW process

Fig. 6. The VRW extends to MMAS in the tour construction stage.

Table 3
Hardware specifications of our test platform.

Property	CPU	GPU
Manufacturer	Intel	NVIDIA
Model	Core i7-5820 k	GeForce GTX780
Codename/architecture	Haswell-E	Kepler
Clock frequency	3.3 GHz	863 MHz
Cores (SMs)/Threads	6/12	12/2304
Vector lanes/warp size	8	32
L1 Cache size	32 KB per core	64 KB per SM
L2 Cache size	256 KB per core	1.5 MB
L3 Cache size	15 MB	N.A.
DRAM memory	16 GB	3 GB
Memory bandwidth	34 GB/s	288.4 GB/s
Max GFLOP/s(single)	316.8	3977
Max TDP	140 W	250 W

same as Cecilia et al. [10] in performance side. The sequential baseline algorithm is provided by Stüzle in [4]. We focus on the efficiency of the parallel ACO algorithm for multicore-SIMD CPU architecture. However, to ensure that our speedups are convincing, solution quality comparisons among the results obtained by the sequential, GPU-based and multicore-SIMD CPU-based algorithms are also provided.

For the MMAS algorithm, we comply with a different guideline which is explained in further detail below.

4.1. Evaluation of the TPAM-CPU algorithms in tour construction on multicore-SIMD CPU

The baseline code is the naive tour construction kernel without optimizations based on Algorithm 2. Five additional code versions are presented to show the impact of each strategy applied on the overall performances of the TPAM-CPU algorithm (Fig. 7). The speedups are calculated by dividing the naive kernel time with the optimized kernel time.

Code version 1 is based on the original ACO algorithm. Simply using the scan operation would reduce about half of the time required for the TC kernel execution (version 2). On the contrary, version 3 encounters a dramatic performance loss. This kernel presents the already described worse utilization of SIMD units. We could deduce that the branches in the code flow (for example, 'if' statements or different loop exit conditions) have a great impact on the performance of the algorithm. As we have expected, the compaction strategy (version 4) is almost twice as fast as version 2. After we add vectorization support by the VRW method in version 5, the speedup ratio have a little bit growth. The best efficiency is achieved in version 6 which expands version 2. This presents the vector units are fully utilized for acceleration. Note that the VRW applied to the TPAM have significant better performance gain than it applied to the TPAM-COMP. The major reason is that the gather operation in Fig. 6 is actually compiled to a manual packing process

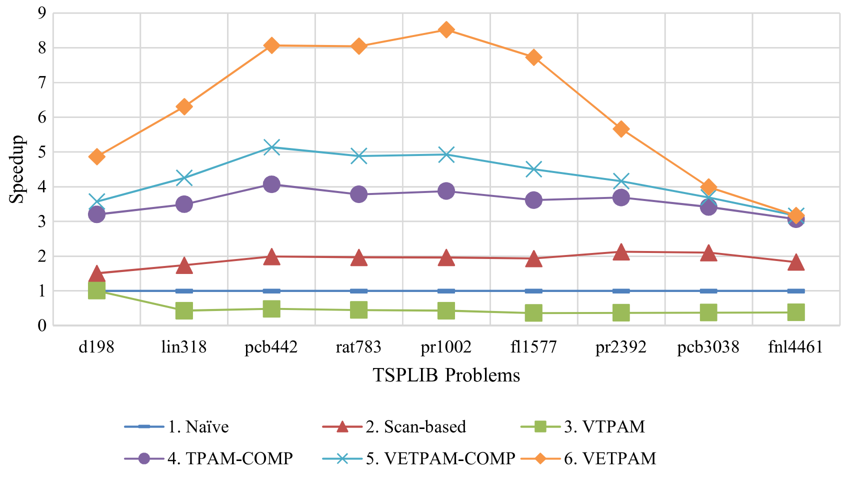

Fig. 7. Speedups of optimizations on naive tour construction kernel.

in which eight elements are loaded to the vector sequentially on the CPU. Nevertheless, in parallel machines, the gather operation is a standard method which is usually supported in instruction-level (VGATHERDPS instruction in AVX2, for example). So potentially, there would be a performance gain when the compiler is able to identify the codes of gather operation in the future.

4.2. *Evaluation of the pheromone update kernel*

In this section, we discuss performance issues for the pheromone update kernel on multicore-SIMD CPU. For the portability of OpenCL, our ACO kernels could also be built to run on the GPU. So we add GPU execution results to compare.

Fig. 8 demonstrate that in small problems, parallel workloads are too light to efficiently use the SIMD or GPU resources. As the problem size increases, the kernel launch latency is better amortized over parallel computation. Therefore, the parallel algorithms significantly outperform the sequential algorithm in large problems.

The kernel without atomic operation is the worst parallel strategy. This is because that the kernel is execute serially to avoid race conditions in writing pheromone values of each ant and the kernel should launch $O(n)$ times to process for all ants. With higher memory bandwidth and a lot more hardware threads, the GPU version perform better than the SIMD CPU.

4.3. *Impact of CU number and memory bandwidth on performance*

To demonstrate the scalability of our algorithm, we evaluate it for different numbers of CUs. In our benchmark platform, the CPU has 6 physical cores and 12 logical cores that are enhanced by the hyper threading (HT) technology; thus, the CPU contains 12 equivalent logical CUs. We use the *subdevice* [23] function in OpenCL to control the active CUs for computing, and each CU represent a CPU core. Fig. 9 shows that our system performance increases nearly linearly with the number of active CUs because more CUs could enhance the parallelism so that more data could be processed concurrently.

Note that the speedup start decreasing from *pr*2392. This behavior can be elucidated by the following reason: the problem dataset is too large to fit in last level cache (LLC), so that the thread stalls to wait for data retrieved from global memory (DRAM). In our algorithm, two ant data structures are used frequently: the *prob* and the *visited*. The total data size is $8n^2 = n * n * 4 * 2$ (bytes). Thus, in the *pr*1002, the data size is 7.66 MB. Because its size is smaller than the L3 cache (15 MB), accessing these data structures could mainly benefit from the cache because in problems where n is less than or equal to 1002, the memory bandwidth has only a slight impact on the performance. As the problem size rises, the size of the ant data structures with space complexity $O(n^2)$ increases dramatically. This result in LLC misses more frequently, and the data missed are forced to read from the host memory to the L3 cache. Therefore, the system performance is bounded by the host memory bandwidth in large-scale TSPs. An analysis of the LLC miss in solving each TSP confirms this fact (see Table 4) by using Intel VTune Amplifier XE 2015. The LLC miss start increasing dramatically from *pr*2392. This demonstrates that the L3 cache hit rate decreases dramatically when the ant data structures could not fully fit in L3 cache.

Our major conclusion here is twofold. The performance of our algorithm scales up with the number of CPU cores, and the memory bandwidth is a primary limiting factor when solving large-scale TSPs on multicore-SIMD CPUs.

4.4. *Multicore-SIMD CPU versus GPU*

In this section, we investigate the performance of our algorithm on GPU and compare it with the multicore-SIMD CPU version. we use TPAM for comparison because it could run on GPU without any code modification. Fig. 10 shows that the CPU version is obviously better than the GPU version. This demonstrates that TPAM is better fit for CPU architecture because that the architecture could handle diverged ACO codes efficiently for its data caching and flow control devoted design.

4.5. *Comparisons with high performance GPU-based ACO algorithms*

4.5.1. *Comparisons with high performance AS algorithms*

To ensure the computational efficiency of our algorithm, we compare against the first data parallel GPU implementation of AS (DPAM-GPU-AS) provided by Cecilia et al. [10]. We rebuild the original DPAM-GPU-AS in our hardware platform to execute on GTX780, thus, to ensure our comparison is fair. In Table 5, the results show our algorithm is up to 3.31x faster than the DPAM-GPU-AS.

We also compare our algorithm with two improved GPU-based ACO algorithms [8,11]. Because there are no source codes provided from both of them, we could not reproduce results of these algorithms in our hardware platform for direct comparison. Therefore, we refer to the average execution time reported in [8,11]. To be fair, a modified version of the VETPAM-CPU algorithm (VETPAM-CPU-AS-MOD) is used for comparison with Dawson's [11]. In

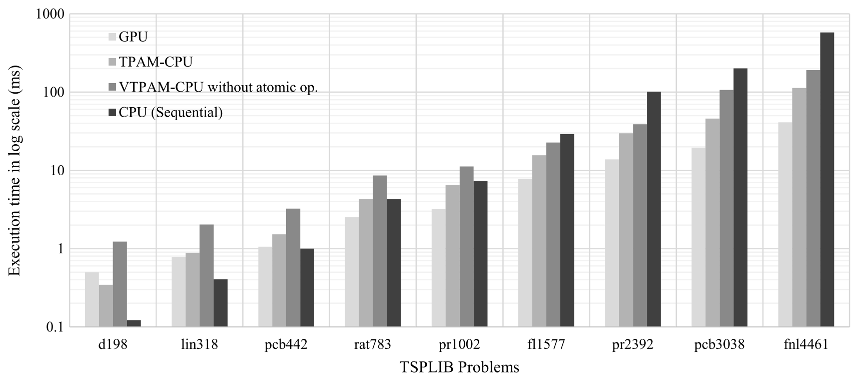

Fig. 8. Execution time comparison of pheromone update kernels.

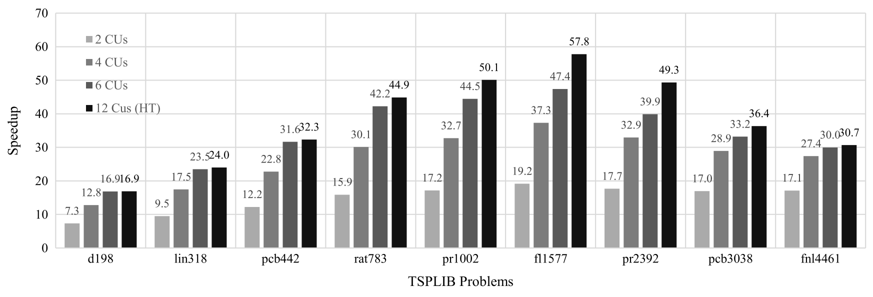

Fig. 9. Speedups on different CU number configurations.

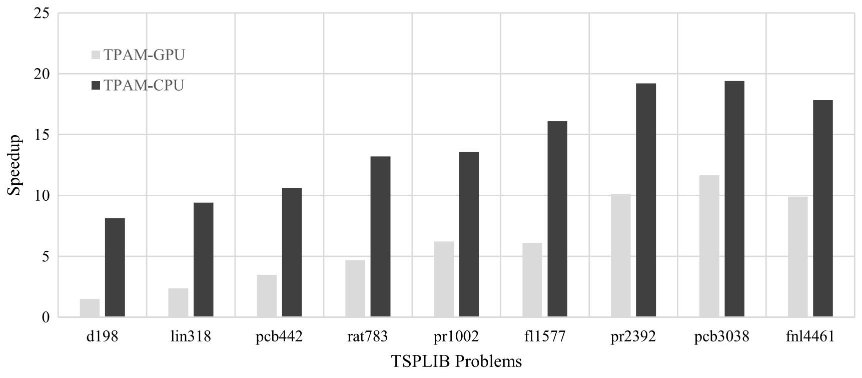

Fig. 10. Speedup comparison of GPU and multicore-SIMD CPU for the VTPAM algorithm.

Table 4
LLC miss analysis of the VETPAM algorithm on the CPU.

TSP problems	d198	lin318	pcb442	rat783	pr1002	fl1577	pr2392	pcb3038	fnl4461
LLC miss	0.012	0.010	0.015	0.011	0.012	0.013	0.228	0.367	0.344

Table 5
Overall execution times (ms) and speedups of our algorithm against the original DPAM-GPU-AS.

Algorithms	TSPLIB problems								
	d198	lin318	pcb442	rat783	pr1002	fl1577	pr2392	pcb3038	fnl4461
DPAM-GPU-AS	2.22	7.51	18.09	86.53	149.85	586.40	2026.47	NA[a]	NA
VETPAM-CPU-AS	1.21	3.42	7.11	29.06	52.70	177.24	858.23	2557.59	10483.20
Speedups	1.82x	2.20x	2.55x	2.98x	2.84x	3.31x	2.36x	NA	NA

[a] NA means "not available" due to on-chip shared memory constraints.

VETPAM-CPU-AS-MOD, the pheromone update stage is adopted from MMAS, which reduces atomic operations largely. The results demonstrate that our proposed VTPAM-CPU algorithm is over two times faster than the current best DPAM-GPU-AS algorithms in the best cases (Table 6).

Table 5 also indicates that the scalability of the VETPAM-CPU is better than the scalability of the DPAM-GPU. Due to the limitation of shared memory size on each CU, previous DPAM-GPU algorithms usually test TSPs up to 2392 cities. Using strategies in DPAM-GPU, for example only part of ant data is placed on local memory and the

Table 6
Overall execution times (ms) and speedups of our algorithm against improved GPU-based ACO algorithms.

Algorithms	TSPLIB problems							
	d198	a280	lin318	pcb442	rat783	pr1002	nrw1379	pr2392
Uchida et al. [8]	2.64	5.06	8.97	11.54	56.73	87.06	–	2084.78
VETPAM-CPU-AS	1.21	2.38	3.42	7.11	29.06	52.70	124.91	858.23
	(2.17x)	(2.12x)	(2.63x)	(1.62x)	(1.95x)	(1.65x)	–	(2.43x)
Dawson and Stewart [11]	1.16	2.68	3.39	7.79	42.7	85.11	323	1979.31
VETPAM-CPU-AS-MOD	0.87	1.75	2.57	5.66	25.25	47.17	117.73	822.54
	(1.33x)	(1.53x)	(1.32x)	(1.38x)	(1.69x)	(1.80x)	(2.74x)	(2.41x)

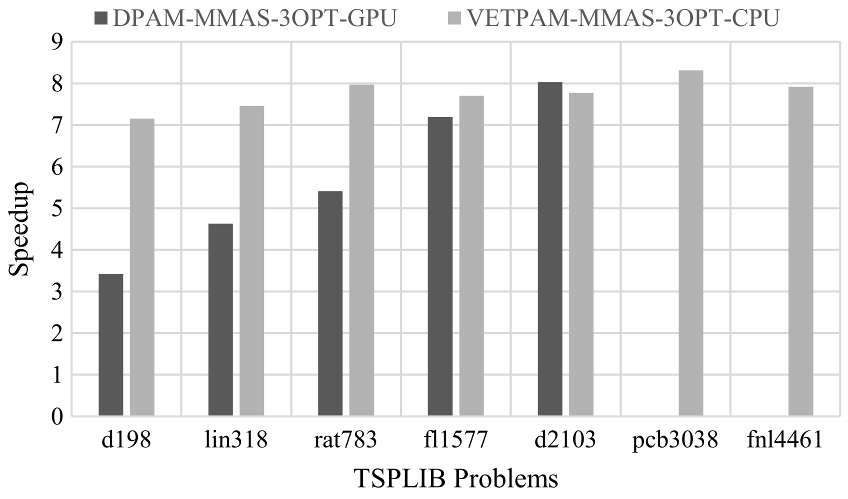

Fig. 11. Speedup comparison of parallel MMAS with local search (3-opt) algorithms.

rest is loaded from the global memory on demand, may solve this limiting issue. However, an increased coding effort is required to manage the GPU memory. Our VETPAM-CPU algorithm could solve TSPs up to 4461 cities, and the speedups are still kept as high as 30x (Fig. 9). Furthermore, the VETPAM-CPU could potentially solve TSPs larger than 4461 cities efficiently without modification to the code. This is because it directly accesses the global memory, and the complex memory management in algorithm level is hidden by the CPU cache hierarchy.

4.5.2. *Comparisons of high performance MMAS algorithms*

Following the criterions of Delévacq et al. [9], we obtain the result of testing 7 TSP instances in a fixed number of 2048 iterations and averaged over 25 tries. The candidate list size of MMAS and local search are set at 20 and 40 respectively. We use the same number of ants (28) as [9], which is different from 25 in [51]. Since we do not have Delévacqs source code, we pick some of the speedup results from their paper [9]. Note that, the GPU platform of Delévacq is NVIDIA Tesla C2050 and the sequential version of their algorithm was run on Intel Xeon E5640, where both of them were up-to-date hardware architectures at that time. Fig. 11 shows that our VETPAM-CPU algorithm could achieve similar speedups compared with the GPU-based counterpart under the same algorithm configuration.

As aforementioned, we present a new vector-enhanced algorithm for ACO on multicore-SIMD CPUs and new design of roulette wheel selection approach in the tour construction stage. We believe our ideas could bring acceleration to other population-based metaheuristic algorithms on multicore-SIMD CPUs.

4.6. *Solution quality validation*

4.6.1. *VTPAM-AS algorithm*

Following the guidelines of Cecilia et al. [10], we obtain the result of testing all algorithms a fixed number of 1000 iterations and averaged over 10 tries. Fig. 12 shows the quality comparison for the solutions.

The results indicate that the solution quality of the algorithm that we present is similar to the solution quality of the sequential code. The solutions we obtain are not optimal because of the limitation of the original Ant System algorithm as explained by Stützle and Hoos [51], and this can be improved by adding a local search process to each ant.

4.6.2. *VTPAM-MMAS with local search algorithm*

We compare our solution quality with the state-of-the-art MMAS algorithms in literature [9,51]. Table 7 provides average and minimum results for each approach. We also give results of two large TSP instances for that CPU-based algorithm is not limited by local memory size. Our solution quality is similar to the previous algorithms. The results are near optimum (sometimes optimum) except two large datasets. The reason is that the number of iterations (2048) is not enough for that problem size.

The comparison results of the solution quality present that our methods are efficient both in execution speed and in solution quality.

5. Conclusions and future work

In this paper, we aim to provide efficient parallel ACO models for CPU-based SIMD architecture. After improving the general task-parallel approach, we investigate both sequential and vector-parallel optimization strategies. We have demonstrated that simple vectorization strategy does not fit well on SIMD CPUs. In order to fully exploit vector-level parallelism, we propose an alternative approach based on vectors. In this approach, our algorithm distinguishes branch codes and data-intensive codes, and then the codes are executed on scalar units and vector units respectively. We use a large range of TSP instances varying from 198 to 4461 cities to

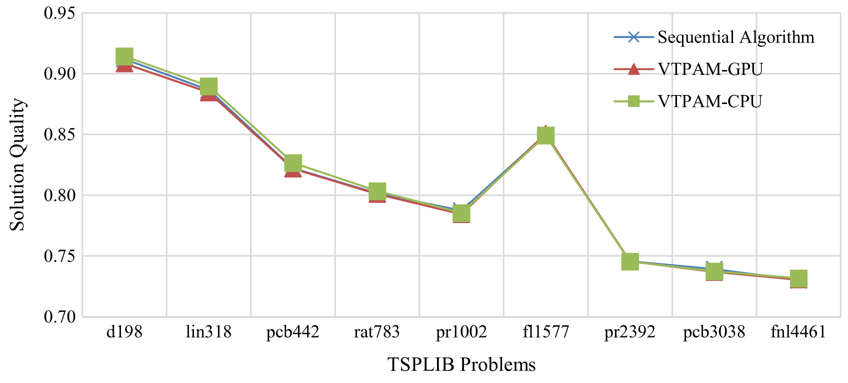

Fig. 12. Solution quality comparison results of AS algorithms.

Table 7
Solution quality comparison results of MMAS with local search (3-opt) algorithms. Minimum and average values are given.

Algorithms	TSPLIB problems						
	d198	lin318	rat783	fl1577	d2103	pcb3038	fnl4461
Stützle and Hoos [21]	15780	42029	8806	22261	–	–	–
	15780.2	42061.7	8816.08	22271.8	–	–	–
Delévacq et al. [9]	15780	42029	8806	22262	80522	–	–
	15780.04	42036.04	8828.68	22286.1	80682.4	–	–
Sequential algorithm	15780	42029	8806	22281	80463	137905	183010
	15780	42038.12	8808	22303.76	80511.96	138097	183289.96
VTPAM-CPU-MMAS-3OPT	15780	42029	8806	22274	80451	137949	183145
	15780	42033.56	8810	22304.72	80513.64	138111.52	183322.72
Optimum	15780	42029	8806	22249	80450	137694	182566

evaluate our algorithm, and we achieve speedups over sequential version as high as 57.8 in the AS and 8.3 in the MMAS with 3-opt local search, and meanwhile keep similar solution quality.

Furthermore, to demonstrate the validity and significance of our contributions, we compare our approach with state-of-art high performance GPU-base parallel ACOs. Our benchmark results demonstrate that our SIMD CPU-based ACO algorithm achieves a competitive performance compared with GPU-based ACO algorithms. The solution quality of our approach is evaluated to guarantee that the competition is convincing when compared with GPU-based approaches.

More interesting, the proposed method can be generalized. VRW approach can be applied to other metaheuristics including stochastic selection process such as genetic algorithms. And the vector-enhanced task parallel model also can be applies to other population-based metaheuristics for their similarities in the behaviors.

In the future, we should investigate the vectorization approaches in the context of MIC architectures, which have larger SIMD registers such as Intel Xeon Phi Coprocessors. And the quantitative performance analysis and algorithmic optimization for parallel ACO algorithms should be conducted on different parallel architectures. We should also apply our approaches to parallelize other algorithms which we have implemented the sequential versions, such as collaborative computing [52–54] and Hausdorff distance computing [55,56]. Finally, we should also try to extend the idea to the areas of graphics/image/video process [57–64].

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments. And thank Dr. Cecilia for providing the CUDA source code in [10] which is a great benchmark for comparison. The authors would also like to thank AMAX China Suzhou branch for their support of accessing computational resources. This paper is supported by the National Science Foundation of China (Grant No. 61472289) and National Key Research and Development Project (2016YFC0106305).

References

[1] T. Van Luong, N. Melab, E.-G. Talbi, GPU computing for parallel local search metaheuristic algorithms, IEEE Trans. Comput. 62 (1) (2013) 173–185. http://dx.doi.org/10.1109/TC.2011.206.
[2] C. Blum, A. Roli, Metaheuristics in combinatorial optimization: overview and conceptual comparison, ACM Comput. Surv. 35 (2003) 268–308. http://dx.doi.org/10.1145/937503.937505.
[3] Metaheuristics network website http://www.metaheuristics.net/index.php?main=1. (Accessed 2 December 2014).
[4] M. Dorigo, T. Stützle, Ant Colony Optimization, MIT Press, Cambridge, Mass, 2004.
[5] M. Dorigo, E. Bonabeau, G. Theraulaz, Ant algorithms and stigmergy, Future Gener. Comput. Syst. 16 (2000) 851–871. http://dx.doi.org/10.1016/S0167-739X(00)00042-X.
[6] D. Shmoys, J. Lenstra, A. Kan, E. Lawler, The traveling salesman problem, in: Wiley Interscience Series in Discrete Mathematics, John Wiley & Sons, 1985.
[7] Y. Zhou, F. He, Y. Qiu, Optimization of parallel iterated local search algorithms on graphics processing unit, J. Supercomput. 72 (6) (2016) 2394–2416. http://dx.doi.org/10.1007/s11227-016-1738-3.
[8] A. Uchida, Y. Ito, 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, IEEE, 2012, pp. 94–102. http://dx.doi.org/10.1109/ICNC.2012.22.
[9] A. Delévacq, P. Delisle, M. Gravel, M. Krajecki, Parallel ant colony optimization on graphics processing units, J. Parallel Distrib. Comput. 73 (1) (2013) 52–61. http://dx.doi.org/10.1007/978-3-642-34413-8_30.
[10] J.M. Cecilia, J.M. García, A. Nisbet, M. Amos, M. Ujaldón, Enhancing Data parallelism for ant colony optimization on GPUs, J. Parallel Distrib. Comput. 73 (1) (2013) 42–51. http://dx.doi.org/10.1016/j.jpdc.2012.01.002.
[11] L. Dawson, I. Stewart, Improving ant colony optimization performance on the GPU using CUDA, in: Evolutionary Computation (CEC), 2013 IEEE Congress on, 2013, pp. 1901–1908. http://dx.doi.org/10.1109/CEC.2013.6557791.
[12] J.L. Hennessy, D.A. Patterson, Computer Architecture - A Quantitative Approach Fifth Edition, Morgan Kaufmann, Waltham, MA, 2012.
[13] J. Chhugani, A.D. Nguyen, V.W. Lee, W. Macy, M. Hagog, Y. kuang Chen, A. Baransi, S. Kumar, P. Dubey, Efficient implementation of sorting on multi-core SIMD CPU architecture, Proc. Vldb Endowment 1 (2008) 1313–1324. http://dx.doi.org/10.14778/1454159.1454171.
[14] N. Sedaghati, R. Thomas, L.-N. Pouchet, R. Teodorescu, P. Sadayappan, StVEC: A vector instruction extension for high performance stencil computation, in: International Conference on Parallel Architectures and Compilation Techniques, 2011, pp. 276–287. http://dx.doi.org/10.1109/PACT.2011.59.

[15] J. Kurzak, W. Alvaro, J. Dongarra, Optimizing matrix multiplication for a short-vector SIMD architecture - cell processor, Parallel Comput. 35 (2009) 138–150. http://dx.doi.org/10.1016/j.parco.2008.12.010.

[16] F. Pratas, P. Trancoso, L. Sousa, A. Stamatakis, G. Shi, V. Kindratenko, Fine-grain parallelism using multi-core, cell/BE, and GPU systems, Parallel Comput. 38 (8) (2012) 365–390. http://dx.doi.org/10.1016/j.parco.2011.08.002. Application accelerators in HPC.

[17] J. Shen, J. Fang, H. Sips, A.L. Varbanescu, An application-centric evaluation of OpenCL on multi-core CPUs, Parallel Comput. 39 (12) (2013) 834–850. http://dx.doi.org/10.1016/j.parco.2013.08.009. Programming models, systems software and tools for high-end computing.

[18] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, J. Dongarra, From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming, Parallel Comput. 38 (8) (2012) 391–407.

[19] J.M. Cecilia, A. Llanes, L. Chang, J.M. Garca, I. Navarro, W. Hwu, V-aco: A vectorization approach for high-performance ant colony optimization, in: The 5th International Conference on Metaheuristics and Nature Inspired Computing, META2014, Oral Communication, 2014, Marrakech (Morocco). http://meta2014.sciencesconf.org/42493/document.

[20] M. Dorigo, V. Maniezzo, A. Colorni, Ant system: Optimization by a colony of cooperating agents, IEEE Trans. Syst. Man Cybern. B 26 (1) (1996) 29–41. http://dx.doi.org/10.1109/3477.484436.

[21] T. Stützle, H.H. Hoos, MAX-MIN ant system, Future Gener. Comput. Syst. 16 (2000) 889–914. http://dx.doi.org/10.1016/S0167-739X(00)00043-1.

[22] Intel, Intel Architecture Instruction Set Extensions Programming Reference, Intel, 2014.

[23] Khronos OpenCL Working Group, The OpenCL Specification Version 1.2, Khronos Group, 2011.

[24] X. Yan, F. He, Y. Chen, A novel hardware/software partitioning method based on position disturbed particle swarm optimization with invasive weed optimization, J. Comput. Sci. Tech. 32 (2) (2017) 340–355. http://dx.doi.org/10.1007/s11390-017-1714-2.

[25] X. Yan, F. He, N. Hou, H. Ai, An efficient particle swarm optimization for large scale hardware/software co-design system, Int. J. Cooperative Inf. Syst. (2017). http://dx.doi.org/10.1142/S0218843017410015.

[26] E. Bonabeau, M. Dorigo, G. Theraulaz, Swarm Intelligence: From Natural to Artificial Systems, Oxford University Press, New York, 1999.

[27] M. Pedemonte, S. Nesmachnow, H. Cancela, A survey on parallel ant colony optimization, Appl. Soft Comput. 11 (2011) 5181–5197. http://dx.doi.org/10.1016/j.asoc.2011.05.042.

[28] B. Bullnheimer, G. Kotsis, C. Strauß, in: R. De Leone, A. Murli, P. Pardalos, G. Toraldo (Eds.), High Performance Algorithms and Software in Nonlinear Optimization, in: Applied Optimization, vol. 24, Springer US, 1998, pp. 87–100. http://dx.doi.org/10.1007/978-1-4613-3279-4_6.

[29] T. Stützle, Parallelization strategies for ant colony optimization, in: A. Eiben, T. Bäck, M. Schoenauer, H.-P. Schwefel (Eds.), Parallel Problem Solving from Nature PPSN V, in: Lecture Notes in Computer Science, vol. 1498, Springer Berlin Heidelberg, 1998, pp. 722–731. http://dx.doi.org/10.1007/BFb0056914.

[30] T.N. Bui, T. Nguyen, J.R. Rizzo Jr., (2009) Parallel shared memory strategies for ant-based optimization algorithms, in: Genetic and Evolutionary Computation Conference, 2009, pp. 1–8. http://dx.doi.org/10.1145/1569901.1569903.

[31] S. Tsutsui, N. Fujimoto, Parallel ant colony optimization algorithm on a multi-core processor, in: Swarm Intelligence, in: Lecture Notes in Computer Science, vol. 6234, Springer Berlin Heidelberg, 2010, pp. 488–495. http://dx.doi.org/10.1007/978-3-642-15461-4_48.

[32] N. Li, D. Gao, G. Gong, Z. Chen, Realization of parallel ant colony algorithm based on TBB multi-core platform, in: International Forum on Information Technology and Applications, 2010. http://dx.doi.org/10.1109/IFITA.2010.143.

[33] D. Gao, G. Gong, L. Han, N. Li, Application of multi-core parallel ant colony optimization in target assignment problem, in: International Conference on Computer Application and System Modeling, 2010. http://dx.doi.org/10.1109/ICCASM.2010.5620672.

[34] Y. Zhang, H. Wang, Y. Zhang, D. Liu, Y. Chen, An improved ant system algorithm based on PPL, in: Information Engineering and Computer Science (ICIECS), 2010 2nd International Conference on, 2010, pp. 1–4. http://dx.doi.org/10.1109/ICIECS.2010.5677707.

[35] E. Kugu, O. Sahingoz, ACO algorithms with multi-core implementation, in: Application of Information and Communication Technologies (AICT), 2013 7th International Conference on, 2013, pp.1 –5. http://dx.doi.org/t10.1109/ICAICT.2013.6722749.

[36] Y. Zhang, H.L. Wang, X. Li, Y.H. Zhang, H. Wang, Parallel ant system based on OpenMP, Adv. Mater. Res. 765 (2013) 658–661. http://dx.doi.org/10.4028/www.scientific.net/AMR.765-767.658.

[37] A. Catala, J. Jaen, J.A. Mocholi, Strategies for accelerating ant colony optimization algorithms on graphical processing units, in: Evolutionary Computation, 2007, CEC 2007, IEEE Congress on, IEEE, 2007, pp. 492–500. http://dx.doi.org/10.1109/CEC.2007.4424511.

[38] J. Wang, J. Dong, C. Zhang, Implementation of ant colony algorithm based on GPU, in: Computer Graphics, Imaging and Visualization, 2009, CGIV '09, Sixth International Conference on, IEEE, 2009, pp. 50–53. http://dx.doi.org/10.1109/CGIV.2009.20.

[39] W. Zhu, J. Curry, Parallel ant colony for nonlinear function optimization with graphics hardware acceleration, in: Systems, Man and Cybernetics, 2009, SMC 2009, IEEE International Conference on, IEEE, 2009, pp. 1803–1808. http://dx.doi.org/10.1109/ICSMC.2009.5346870.

[40] J. Li, X. Hu, Z. Pang, K. Qian, A parallel ant colony optimization algorithm based on fine-grained model with GPU-acceleration, Int. J. Innovative Comput. Inf. Control 5 (11) (2009) 3707–3716.

[41] H. Bai, D. OuYang, X. Li, L. He, H. Yu, MAX-MIN ant system on GPU with CUDA, in: Innovative Computing, Information and Control (ICICIC), 2009 Fourth International Conference on, IEEE, 2009, pp. 801–804. http://dx.doi.org/10.1109/ICICIC.2009.255.

[42] J. Fu, L. Lei, 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, IEEE, 2010, pp. 260–264. http://dx.doi.org/10.1109/IWACI.2010.5585115.

[43] J.M. Cecilia, A. Nisbet, M. Amos, J.M. García, M. Ujaldón, Enhancing GPU parallelism in nature-inspired algorithms, J. Supercomput. 63 (3) (2013) 773–789. http://dx.doi.org/10.1007/s11227-012-0770-1.

[44] A. Llanes, J.M. Cecilia, A. Sánchez, J.M. García, M. Amos, M. Ujaldón, Dynamic load balancing on heterogeneous clusters for parallel ant colony optimization, Cluster Comput. 19 (1) (2016) 1–11. http://dx.doi.org/10.1007/s10586-016-0534-4.

[45] R. Skinderowicz, The GPU-based parallel ant colony system, J. Parallel Distrib. Comput. 98 (2016) 48–60. http://dx.doi.org/10.1016/j.jpdc.2016.04.014.

[46] Y. Zhou, F. He, Y. Qiu, Dynamic strategy based parallel ant colony optimization on GPUs for TSPs, Sci. China Inf. Sci. 60 (6) (2017) 068102. http://dx.doi.org/10.1007/s11432-015-0594-2.

[47] Intel, Writing Optimal OpenCL Code with Intel OpenCL SDK, Intel, 2011.

[48] H.H. Hoos, T. Stützle, Stochastic Local Search: Foundations and Applications, in: The Morgan Kaufmann Series in Artificial Intelligence, Morgan Kaufmann Publishers, San Francisco, CA, 2005.

[49] Nvidia, NVIDIA CUDA C Programming Guide 4.0, Nvidia, 2011.

[50] G. Reinelt, TSPLIB - A Traveling Salesman Problem Library, INFORMS J. Comput. 3 (4) (1991) 376–384. http://dx.doi.org/10.1287/ijoc.3.4.376.

[51] T. Stützle, H.H. Hoos, MAX-MIN ant system, Future Gener. Comput. Syst. 16 (2000) 889–914. http://dx.doi.org/10.1016/S0167-739X(00)00043-1.

[52] X. Lv, F. He, W. Cai, Y. Cheng, A string-wise CRDT algorithm for smart and large-scale collaborative editing systems, Adv. Eng. Inf. 32 (3) (2017) 397–409. http://dx.doi.org/10.1016/j.aei.2016.10.005.

[53] Y. Wu, F. He, D. Zhang, X. Li, Service-oriented feature-based data exchange for cloud-based design and manufacturing, IEEE Trans. Sev. Comput. (2015). http://dx.doi.org/10.1109/TSC.2015.2501981.

[54] Y. Wu, F. He, S. Han, Collaborative CAD Synchronization based on a symmetric and consistent modeling procedure, Symmetry 9 (4) (2017) 59. http://dx.doi.org/10.3390/sym9040059.

[55] Y. Chen, F. He, Y. Wu, N. Hou, A local start search algorithm to compute exact hausdorff distance for arbitrary point sets, Pattern Recognit. 67 (2017) 139–148. http://dx.doi.org/10.1016/j.patcog.2017.02.013.

[56] D. Zhang, F. He, S. Han, Z. Lu, Y. Wu, Y. Chen, An efficient approach to directly compute the exact Hausdorff distance for 3D point sets, Integrated Comput.-Aided Eng. 24 (3) (2017) 261–277. http://dx.doi.org/10.3233/ICA-170544.

[57] K. Li, F. He, X. Chen, Real-time object tracking via compressive feature selection, Front. Comput. Sci. 10 (4) (2016) 689–701. http://dx.doi.org/10.1007/s11704-016-5106-5.

[58] K. LI, F. He, H. Yu, X. Chen, A correlative classifiers approach based on particle filter and sample set for tracking occluded target, Appl. Math. J. Chin. Univ. 32 (3) (2017) 294–312. http://dx.doi.org/10.1007/s11766-017-3466-8.

[59] J. Sun, F. He, Y. Chen, X. Chen, A multiple template approach for robust tracking of fast motion target, Appl. Math. J. Chin. Univ. 31 (2) (2016) 177–197. http://dx.doi.org/10.1007/s11766-016-3378-z.

[60] Y. Cheng, F. He, Y. Wu, D. Zhang, Meta-operation conflict resolution for human–human interaction in collaborative feature-based CAD systems, Cluster Comput. 19 (1) (2016) 237–253. http://dx.doi.org/10.1007/s10586-016-0538-0.

[61] B. Ni, F. He, Y. Pan, Z. Yuan, Using shapes correlation for active contour segmentation of uterine fibroid ultrasound images in computer-aided therapy, Appl. Math. J .Chin. Univ. 31 (1) (2016) 37–52. http://dx.doi.org/10.1007/s11766-016-3340-0.

[62] K. Li, F. He, H. Yu, X. Chen, A parallel and robust object tracking approach synthesizing adaptive bayesian learning and improved incrementalsubspace learning, Front. Comput. Sci. http://dx.doi.org/10.1007/s11704-018-6442-4.

[63] K. Li, F. He, H. Yu, Robust visual tracking based on convolutional features with illumination and occlusion handling, J. Comput. Sci. Tech. (2017). http://dx.doi.org/10.1007/s11390-017-1764-5.

[64] D. Zhang, F. He, S. Han, X. Li, Quantitative optimization of interoperability during feature-based data exchange, Integrated Comput.-Aided Eng. 23 (1) (2016) 31–50. http://dx.doi.org/10.3233/ICA-150499.

Yi Zhou received his B.S. (2005) and M.S. (2008) degrees from Wuhan University of Science and Technology. He received his Ph.D. degree (2016) from Wuhan University. He was a software engineer at WISDRI Engineering and Research Incorporation Limited. He is currently a lecturer at School of Information Science and Engineering, Wuhan University of Science and Technology. His research interest includes metaheuristics, nature-inspired computing and parallel computing.

Fazhi He received Ph.D. degree from Wuhan University of Technology. He was post-doctor researcher in The State Key Laboratory of CAD & CG at Zhejiang University, a visiting researcher in Korea Advanced Institute of Science & Technology and a visiting faculty member in the University of North Carolina at Chapel Hill. Now he is a professor in State Key Laboratory of Software Engineering, School of Computer, Wuhan University. His research interests are Computer Graphics, Computer-Aided Design, Image Processing and Computer Supported Cooperative Work.

Neng Hou is currently a Ph.D. candidate at the School of Computer Science in Wuhan University. His research interests include Hardware/Software Co-design, intelligent optimization algorithms, and GPU computing.

Yimin Qiu received her B.S. (2002) and M.S. (2009) degrees from Wuhan University of Science and Technology. She received her Ph.D. degree (2014) from Wuhan University. She is currently a senior engineer at School of Information Science and Engineering, Wuhan University of Science and Technology. Her research interest includes image processing, computer vision and high-performance computing.