


RESEARCH

Open Access



# An efficient parallel genetic algorithm solution for vehicle routing problem in cloud implementation of the intelligent transportation systems

Mahdi Abbasi<sup>1\*</sup> , Milad Rafiee<sup>1</sup>, Mohammad R. Khosravi<sup>2,3</sup>, Alireza Jolfaei<sup>4</sup>, Varun G. Menon<sup>5</sup> and Javad Mokhtari Koushyar<sup>1</sup>

## Abstract

A novel parallelization method of genetic algorithm (GA) solution of the Traveling Salesman Problem (TSP) is presented. The proposed method can considerably accelerate the solution of the equivalent TSP of many complex vehicle routing problems (VRPs) in the cloud implementation of intelligent transportation systems. The solution provides routing information besides all the services required by the autonomous vehicles in vehicular clouds. GA is considered as an important class of evolutionary algorithms that can solve optimization problems in growing intelligent transport systems. But, to meet time criteria in time-constrained problems of intelligent transportation systems like routing and controlling the autonomous vehicles, a highly parallelizable GA is needed. The proposed method parallelizes the GA by designing three concurrent kernels, each of which running some dependent effective operators of GA. It can be straightforwardly adapted to run on many-core and multi-core processors. To best use the valuable resources of such processors in parallel execution of the GA, threads that run any of the triple kernels are synchronized by a low-cost switching mechanism. The proposed method was experimented for parallelizing a GA-based solution of TSP over multi-core and many-core systems. The results confirm the efficiency of the proposed method for parallelizing GAs on many-core as well as on multi-core systems.

**Keywords:** Vehicle routing, Cloud computing, Genetic algorithm, Transportation systems, Parallel

## Introduction

Vehicle routing problems have been the focus of extensive research over the past 60 years, driven by their economic importance and their theoretical interest. The diversity of applications has motivated the study of an innumerable problem variants with different attributes in transportation literature [1, 2].

Using efficient vehicle routes provides a direct competitive advantage to transportation companies, which usually operate with limited profitability margins. Moreover, the fact that these problems share a simple yet rich structure, generalizing the traveling salesman problem, has helped to elevate the VRP family into one of the

main testbed for studies in optimization and heuristics [3, 4]. Recently, integration of the traffic management systems (TMSs) with cloud computing paradigm has brought the extensibility and scalability for these systems [5]. Recently, many researches seek VRP solutions which can be easily exploited on vehicular cloud computing (VCC) platforms [6–8].

This research paper proposes an optimally parallelizable Genetic Algorithm solution to TSP problem in VRPs. Approximate solutions have been widely used in science and engineering problems. Genetic Algorithm is a class of evolutionary algorithms which is used for finding approximate solutions for search and optimization problems.

A challenging issue in solving problems using GA is the considerable iterations of the genetic algorithm. In such cases, with the increase in the number of generations, the

\* Correspondence: [abbasi@basu.ac.ir](mailto:abbasi@basu.ac.ir)

<sup>1</sup>Department of Computer Engineering, Engineering Faculty, Bu-Ali Sina University, Hamedan, Iran

Full list of author information is available at the end of the article

populations and their required crossovers and mutations will increase. These, in turn, significantly increase the time complexity of the deployed algorithm. This computational disadvantage of GA is considered as a major problem in deploying GA-based solution to time-constrained problems like vehicle routing and controlling navigation of autonomous vehicles in intelligent transportation systems [9].

Computational bottlenecks of the GA are fitness, mutation, crossover and selection functions. A common solution to this challenge is to migrate the computation of these functions to parallel machines [10, 11]. Therefore, regarding the wide use of GA in a variety of problems such as optimization [12], image processing [13] artificial neural networks training [14] and rule-based systems [15] many researchers try to investigate parallelization methods for GAs on multi-core systems as well as many-core systems.

This paper presents a method for parallelizing the main operators of the genetic algorithm. The proposed parallelism is based on the structure of multi-core Central Processing Units (CPUs) and many-core Graphics Processing Units (GPUs) and tries to compare the power of the two processors in parallelizing genetic algorithms. Our methodology for evaluating the performance of these two different types of processors is based on the results of executing parallel GAs which solve a well-known complex problem in computer science, i.e. TSP problem. TSP is an NP-complete problem with high complexity. The main idea of TSP is finding the shortest route among a set of cities, provided that each city is visited only once, and at the end the city of origin should be visited again.

Specifically, the contributions of the paper are as follows:

- 1- Usually, in comparisons between multi-core CPU and many-core GPU, the CPU is illustrated weaker than GPU in certain cases. In this article, by comparing the performance of Threading Building Blocks (TBB) and Compute Unified Device Architecture (CUDA) platforms in solving TSP problem with a parallel GA, we show that by optimally exploiting the parallel resources of a multi-core CPU, a higher level of performance could be obtained.
- 2- In our multi-core parallelization, all working threads should be synchronized. But, in GPU-based parallelization only threads of one block could be synchronized with each other. Hence, in order to best utilize the resources of GPU in parallelization of GA, we use three distinct kernels which let us synchronize threads in different blocks of GPU. This setting makes it possible to synchronize all of

the working threads but, meanwhile, imposes the cost of switching among kernels. We investigate this cost in our parallel kernels.

- 3- The investigated TBB-based and CUDA-based methods are evaluated on different data sets, according to the variations of GA operators including length of chromosomes, number of generations, and size of populations.

The rest of the paper is structured as follows. In the next section, a brief overview of the related works is presented. Section three reviews the structure of CUDA and TBB platforms. In addition, the method of solving TSP with genetic algorithm is explained next. The proposed approach to parallel implementation of a GA problem is discussed in the fourth section. Experiments and their corresponding analyses are extensively explored in the following section. Section six concludes the paper and suggests some future directions.

### Related work

In this section, we review the related works in parallel implementation of GA algorithms for solving TSP. For this purpose, first we review the studies that have parallelized their algorithm on multi-core CPUs or many-core GPUs. Next, we review the works that have compared the performance of different parallel GA-based TSP solutions.

J. Zhu et al. [16] combine TBB and MPI platforms to parallelize a GA-based solution of TSP. In their parallelization model which is named Island Model, the original population is divided into a series of subpopulations called islands or demes. Such demes could evolve in parallel but are mostly isolated from each other. Interaction between demes is done by an operator known as migration. Each island is a process which is connected to other ones via MPI ring topology. After running five generations of each process, it sends 5% of its best population to another process. Initially, each process individually generates the first population randomly, which increases the probability that the population will be uniform in the process. According to the results obtained from implementation on different datasets in 100 generations, the acceleration rate achieved on four processing cores in 144 and 1889 cities is 2.1 and 2.55, respectively.

Studies conducted by Fujimoto et al. [17] and Chen et al. [18] can be considered as preliminary studies which introduce two different methods for Parallelizing GA computations on CUDA platform. The former exploits all threads of a CUDA block for performing the computations of each GA chromosome and the latter exploits only one thread for this purpose. Both of these methods use only one CUDA block for parallelization. Moreover, due to the need for synchronizing threads in

both of these methods, none of them can fully utilize the computational resources of the GPU.

Sánchez et al. [19] consider this issue and propose two parallel models for GAs on GPU, namely, PGAIM and PGAEI which stand for a Parallel Genetic Algorithm with Islands Model and a Parallel Genetic Algorithm with Elite Island, respectively. In both models, each individual is represented by a thread, and each island is represented by a CUDA block. Their results show that PGAEI model outperforms PGAIM model. PGAEI does not let any migration between islands and, instead, creates an elite island with the best individuals from each island. Despite its better results, PGAEI model is unable to produce the intended solution in a shorter time as compared to PGAIM model.

The work of Kang et al. [20] proposes a parallel solution to genetic TSP. In their method, a single gene corresponds to a tour visiting all the cities. For large problem spaces, they propose an improved crossover that maintains a diversity of genes. However, by increasing the problem size, the size and number of genes increases and, consequently, the necessary computation tasks increase. Although they propose a parallel implementation for their proposed GA which can be executed on GPU-like many-core machines, experimental results show the inefficiency of this method in achieving a high level of parallelism on GPUs. Other researches like [21–24] have obscurely implemented parallel kernels for GA-based solutions of TSP.

Up to now, few studies have been carried out to quantitatively compare the efficiency of parallel kernels on multi-core machines with that of GPU-like many-core machines.

The most prominent example is the study conducted by Saxena et al. [25] which compares the efficiency of Open Multi-Processing (OpenMP) and CUDA via execution of parallel GA-based optimization kernels on multi-core CPUs and many-core GPUs. Unfortunately, this study does not provide a common experimental setting for all parallel kernels. As a result, the effect of different parameters of GA on the performance of both parallel kernels which run over OpenMP and CUDA platforms has not been assessed. For example, they have provided plots which represent the effect of the number of cities on the processing time of OpenMP kernel while providing plots which demonstrate the effect of the size of initial population on the processing time of CUDA kernel. Such superficial results cannot be used for any judgment and comparison as to the efficiency of those parallelization platforms.

These studies clearly indicate that research in the field of GA parallelization has tended only to focus on obscure design of parallel programs. None of the presented studies have investigated the efficient parallelization of

GA operators on multi-core platforms like TBB and many-core platforms like CUDA. Specifically, the parallel processing models of the threads and different approaches of synchronization of threads in different blocks of GPU have not been exactly studied in any of them.

In this paper, we investigate the effect of different parameters of GA-based solutions of TSP on the performance of parallel kernel on both multi-core systems as well as many-core systems. For this purpose, we provide parallel kernels for multi-core systems using TBB and for many-core GPUs using CUDA. The results of running this kernels with identical GA settings are used to provide an in-depth comparison of the efficiency of the kernels as well as to assess the effect of each GA parameter on the overall efficiency of each kernel.

## Background

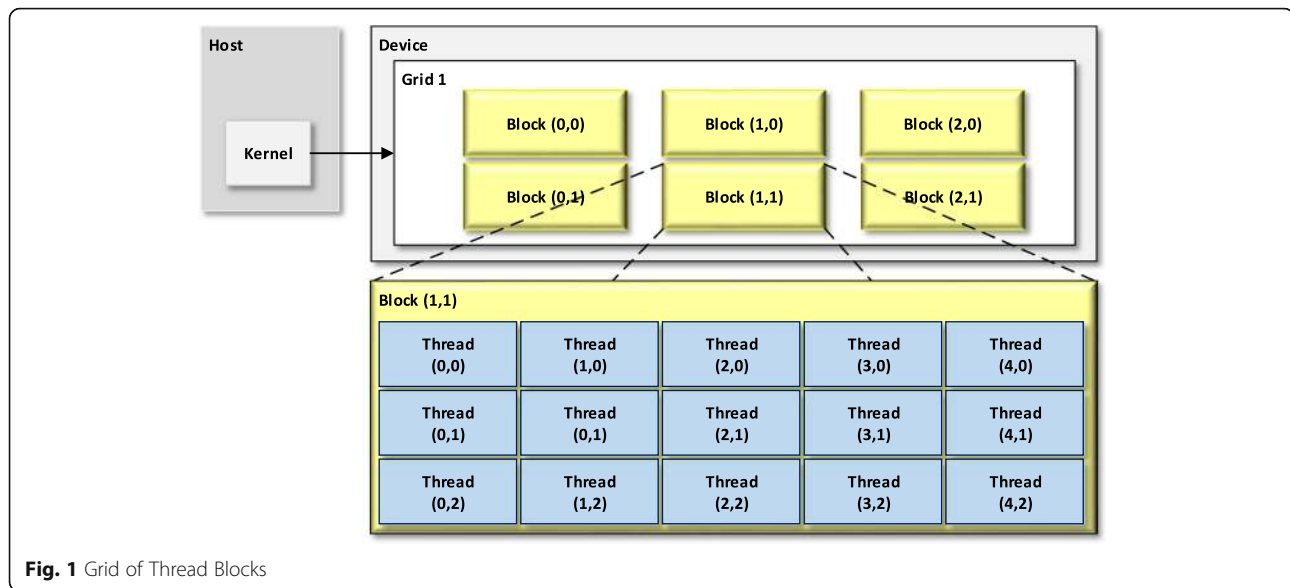
This section gives a brief overview of the related concepts including the architecture of GPU in the CUDA platform and the main ingredients of TBB architecture.

### CUDA

Graphics processing unit is a tool assigned to display graphic images in workstations, game consoles, or personal computers. Due to its high processing power in non-graphic applications, a new branch has been developed in computer science called GPGPU (General-Purpose computing on Graphics Processing Unit). Nvidia's graphics cards include one of the Femi, Tesla, and Kepler structures. Each structure has specific characteristics such as the maximum number of threads in blocks, the size of shared memory, etc. However, in order to facilitate programming, the programming interface has been developed. The programming of GPU is of course a difficult task. Therefore, Nvidia company presented a software platform called CUDA in 2006 to implement non-graphic computation on the graphics processor [26–28].

CUDA provides features for developers to use the hardware capabilities of Nvidia graphics cards in non-graphic programs and to increase the implementation rate of sophisticated algorithms using GPU capabilities. CUDA supports the main factors involved in the computation from two different points of view: host and device. Host runs the main program while device is a help with the processing. A typical scenario is that CPU is considered as the host and GPU is considered as an aid to the processor [29].

Any program that is written in CUDA may consist of several kernels. Each kernel is executed by a grid which consists of several blocks. Each block consists of multiple threads. These threads are responsible for implementing the program. In Fig. 1, the concepts of thread, block and grid are depicted [30].

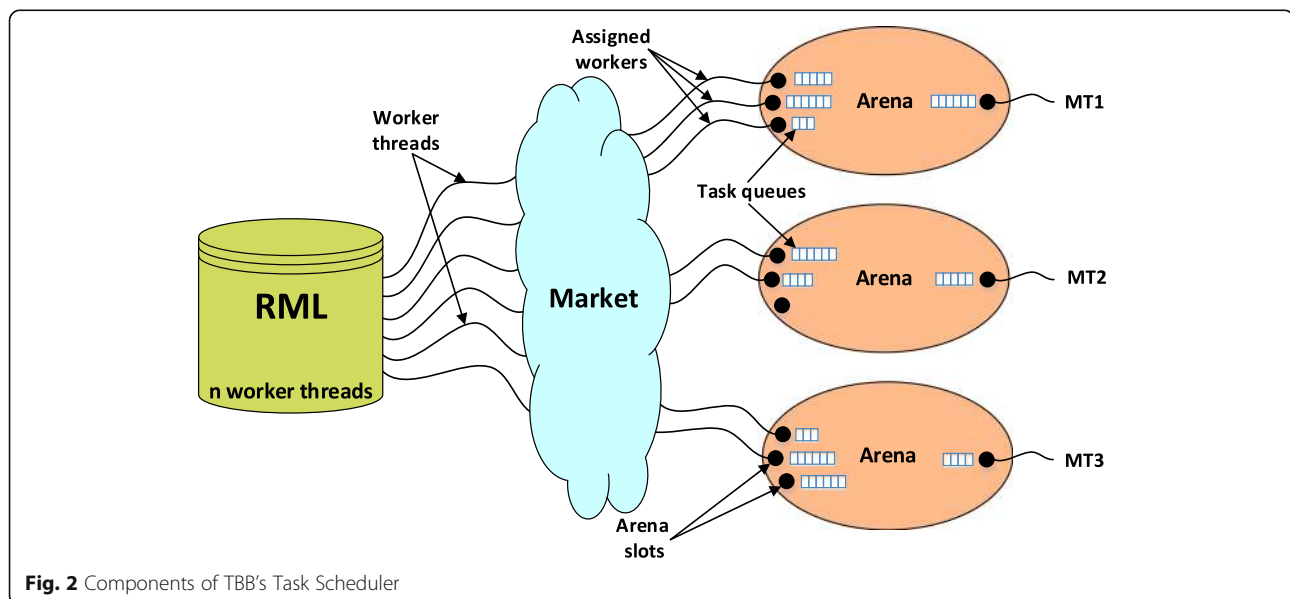


### TBB

TBB was first introduced by Intel in 2006 as a parallel programming library. Figure 2 illustrates the general structure of TBB and how it works to create threads and balance their workloads. This library allows parallelization to be interpreted both explicitly and implicitly. In explicit mode, spawning provides the programmer with full control over the work of each task. The implicit state can be achieved using patterns such as `parallel_for` or `parallel_reduce` that accelerate code-writing. Tasks created explicitly or implicitly are added to the queue of thread tasks in an abstract space called Threads Arena. These tasks are carried out by the master thread or by other workers through

a mechanism called theft. In what follows, we shall have a look at this concept [31].

TBB's master thread represented by MT in this figure is a software thread that instantiates the `TBB::task_scheduler_init` object. All threads that are created by MT and are used to complete MT's task are referred to as worker threads. The resource management layer (RML) is the host of a pool of worker threads. The role of the Market is to distribute the workloads of master threads as well as to assign workers to the arenas of master threads for carrying out the allocated workloads. The number of available worker threads is always one less than the maximum of `tbb::task_scheduler_init` argument and the total number of logical cores on the system



CPU. The next structure is the arena of each MT that encapsulates all the tasks and resources available (worker threads) to execute a master thread. A number of slots are assigned to each arena that represent the number of worker threads which are needed to complete the parallel tasks of the MT. If the total number of slots needed for all master threads exceeds the number of workers in the RML pool, allocation of slots to the arena of MTs will be tailored to the needs. When the task of the master thread ends, the threads produced at the time of creating each arena are either destroyed or assigned by RML to active arenas.

Each working thread, when run in an arena, executes a scheduling procedure called `wait_for_all()` which consists of three nested loops. The inner loop executes the current task by calling the `execute()` method. Upon completion of this task, if no further task is called, the program exits the inner loop. In the middle loop, the `get_task()` method attempts to dequeue local tasks in a Last-In-First-Out (LIFO) order. If successful, the inner loop will be called again. Otherwise, the thread exits the middle loop and the outer loop activates the stealing mechanism by calling the `receive_or_steal_task()` method. This method searches for all tasks on this level. The search includes sending tasks through the task-thread dependency mechanism, reloading tasks without uploading priority, or reloading tasks left by other workers. If the search does not return a task to run, this method steals from a victim thread that is randomly selected at the current location. If the failures of a worker thread to steal exceed a certain threshold (a default value of 100) and the arena of the MT is empty, the failed worker is released and returned to the pool of RML. The details of how tasks are stolen can be found in many sources such as [31, 32].

## The proposed approach

### Parallel GA for TSP

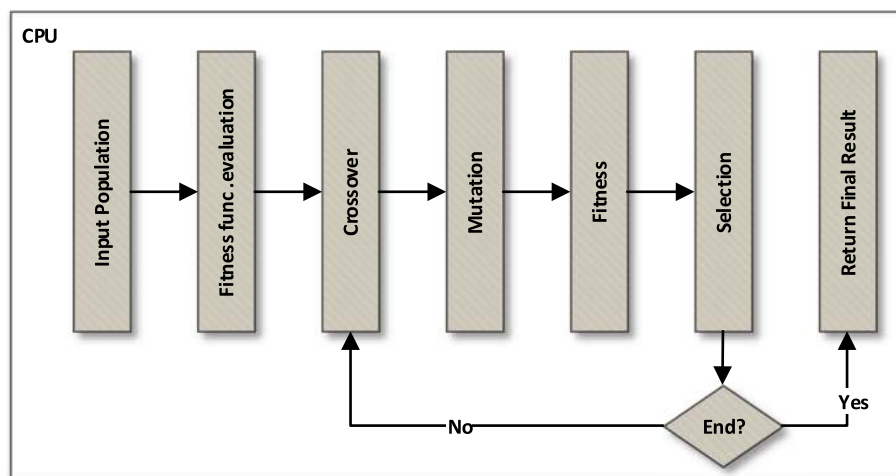
The general method of solving TSP with genetic algorithm, which is discussed in this section, is presented in a flowchart in Fig. 3. It has a lot of applications in different areas of engineering such as timing, routing, etc. [33].

**Population:** each chromosome consists of a fixed number of genes. In this case, each gene is a city and every permutation of cities can be considered as a chromosome.

**Fitness function of the initial population:** for each chromosome, the function gives a non-negative integer which indicates competence and individual ability of each chromosome. To calculate the fitness of each chromosome in TSP, we consider the matrix of coordinates of cities. With respect to the coordinates matrix, the distance between the cities of a chromosome is obtained from the following equation [34]:

$$f(x) = \left( \sum_{i=2}^n \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2} \right) + \sqrt{(x_1 - x_n)^2 + (y_1 - y_n)^2} \quad (1)$$

**Crossover:** this operator is responsible for mating process (exchange of information between paired chromosomes) and also the convergence speed of genetic algorithm. It usually acts with high probability, i.e. 0.6 to 0.9. This value is called crossover rate and is denoted by  $P_c$ . In this case, a parent and a random position between the parent's genes are considered. Then all the genes which were at the right side of the position of the parent



**Fig. 3** Sequential approaches for running Genetic Algorithm



chromosome will be moved so that the new chromosome is obtained.

**Mutation:** it is another operator which is responsible for the new information. This operator with the low probability of 0.01 accidentally changes one of the resulted genes. The overall probability of mutation on a chromosome is called mutation rate which is denoted by  $P_m$ . In this paper, two genes of a chromosome are randomly displaced.

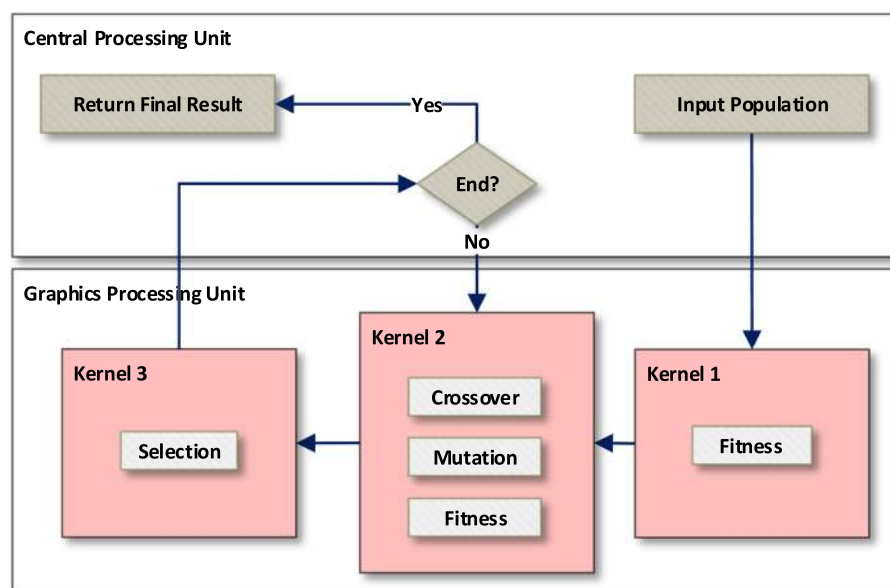
**Calculation of the fitness of new generation:** the fitness function of the population is calculated using crossover and mutation operators just as the calculation of initial population.

**Selection:** there are different methods for selecting the best chromosome and its transfer to the next generation. In parallel implementation, it is better to use tournament selection. In this method, two chromosomes are randomly selected from the population. Then a random number between zero and one is chosen as  $r$ . if  $r < k$  ( $k$  is a parameter for the case of 0.8) then a fitted person or the one which is less adapted is selected as the parents. These two are then returned to the initial population and again participated in the selection process. Finally, the selected chromosomes are recognized as the next generation and are sent to the next round of algorithm implementation [10]. The statistical analysis of the methods of selection in genetic algorithms, as well as other operators, is already studied in many researches including [35]. This issue has a significant effect in designing efficient GAs [36].

The general structure of the sequential and parallel genetic algorithm is illustrated in Fig. 4. In all of the

proposed parallel kernels, the initial population is same. The aim of the proposed kernels for parallel GA is to compare the parallelization power of CUDA and TBB. In the sequential method, all the operations of the genetic algorithm are performed sequentially. However, in the parallel method, the important operations of the GA are implemented in parallel, which reduces the runtime of the genetic algorithm. These operators are fitness, crossover, mutation and selection. Figure 4 shows the flowchart of the proposed method for parallelizing GA. According to the structure of GPUs, only the threads of one block can be synchronized. The synchronization is very important in the genetic algorithm where some operators need to execute in sequence. The most important advantage of our method over recent methods like [18] is proposing a trick for solving the problem of synchronizing threads and using the threads of more than one block. For this purpose, we organize three different kernels, each of which corresponds to a distinct function of GA. These kernels would be replicated on different CUDA blocks simultaneously. By switching between kernels, all threads coincide. The main challenge in switching among kernels is to minimize the related cost. Given that, the result of the computations of each kernel are stored in the global memory of GPU, and no data would be exchanged at each switching step between the host and the device. Therefore, the time required for switching among kernels is very small. The operation of each kernel is described below.

In the proposed parallelism, first the primary population fitness is calculated in parallel by the first kernel. In this kernel, each thread is responsible for calculating the



**Fig. 4** Parallel approaches for running Genetic Algorithm

fitness of a chromosome. Then, in order to create a new generation, the operators of crossover, mutation, and fitness functions are implemented in parallel by the second kernel. In this kernel, each thread is responsible for creating a new child using different operators of the GA. The next operator, which is the selection, is also executed in parallel by the third kernel and selects the best chromosomes of current generation to be passed to the next generation. In this kernel, each thread is responsible for choosing a chromosome. In the end, the condition of the end of the generation is checked. If the condition of the completion of the generation is established, this operation stops and the best answer is returned from the populations as a result. Otherwise, the second and third kernel that perform the creation of a new generation as well as the selections will run to the end of the pre-defined number of generations. In the next section, we examine the performance of parallel methods on TSP.

### Integration to vehicular cloud

Vehicles and sensors in a local area produce the vehicle content. These contents are processed and used by neighbourhood vehicles. Vehicles can access these resources by means of vehicular cloud computing (VCC) [6, 37]. VCC lets vehicles to compute, process, store, and communicate with each other. The corresponding resources provide the feasibility of management of traffic and safety of the roads. Indeed, VCC opens new

possibilities for traffic management via efficient vehicle routing [8].

Our proposed routing system is exploitable on VCCs. In this case, the systems can offer a new direction for the development of traffic management systems. Note that, the integration of VCC with other commercial and roadside clouds can expand the capabilities of VCC. The different types of cloud generated by vehicles and distinctions between them depend on the underlying network infrastructure and the nature of merging between the clouds. Here, we only focus on the overall structure of the VCC platform which is best suited for our method and skip issues and challenges that it brings [7, 8, 37].

Figure 5 shows the proposed VCC structure for implementing the proposed method for routing in smart vehicular networks. In the bottom of the diagram, a vehicular ad-hoc network (VANET) infrastructure is presented. The cloud shows the connections of the participating vehicles. One vehicle acts as a cloud leader or cloud controller. The cloud controller also links to the Internet for additional services. Through the internet, the leader communicates with traffic management officials which runs the proposed algorithm on received vehicular positional information thus providing required routing information (information or event - related instructions) about optimum routes for the entire road network of a vehicles over the city.

Each vehicle have a collection of resources and knows the main service provider which can provide routing information. The abstraction of the vehicle, as shown in

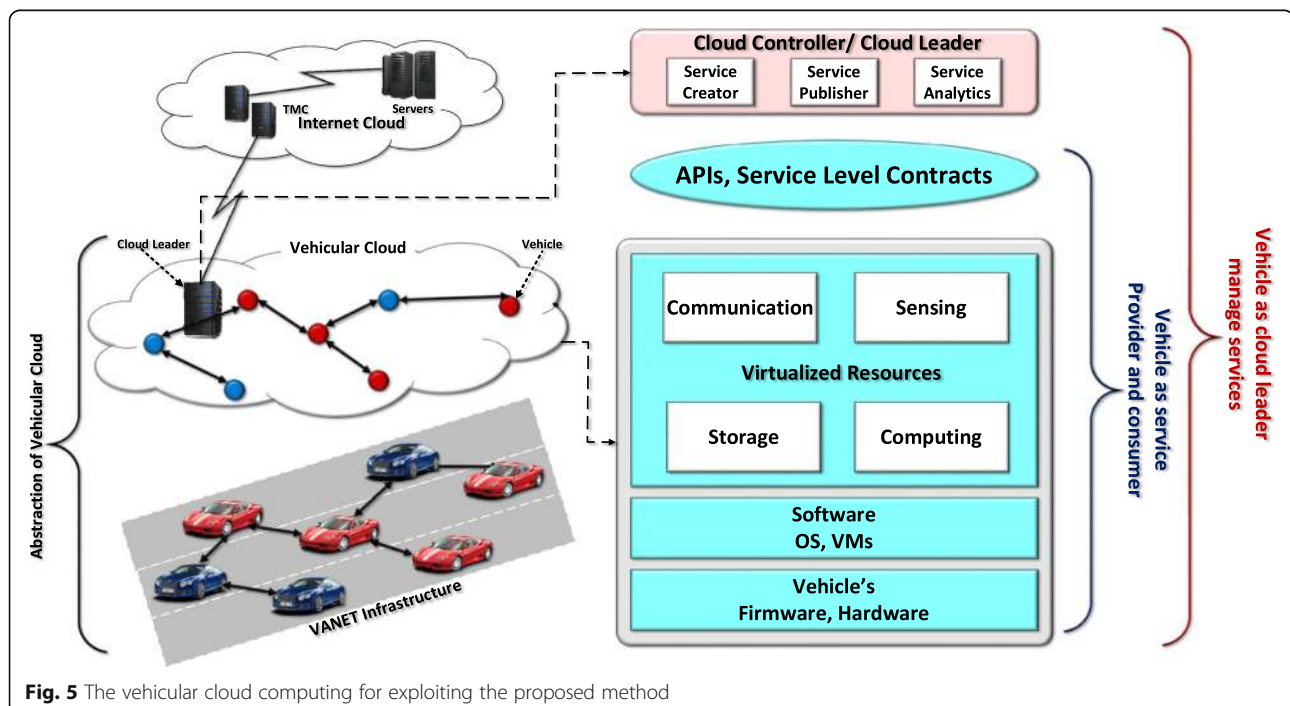


Fig. 5, is shown in the right side of the diagram. Each vehicle has an operating system and hardware at the primary level controlled by a software that can be run on virtual machines. This system provides collects positional data of vehicle and communicates. The resources of the vehicle are actually made available to the cloud according to the service level agreements with cloud leader. The cloud leader initiates, publishes and analyzes the services for the vehicles by continually evaluating and monitoring the virtual resources of the contributing vehicles.

### Implementation and performance evaluation

In this section, first the characteristics of the used CPU and GPU as well as the values of the different parameters of the GA are described. Next, the performance of the proposed parallelization method is investigated based on different metrics.

#### Experimental setup and methodology

The experiments were conducted on an Intel (R) Core i5-7600 3.50GHz personal computer with 8 GB of main memory that was equipped with NVIDIA GeForce GTX 1060 graphics card. This GPU has 1280 cores, and its base and boost clocks are 1506 MHz, and 1708 MHz, respectively. The frameworks used in this implementation are based on C++ CUDA 8.0 (V8.0.61) for many-core GPU and TBB version 2018 for multi-core CPU. The number of threads in the CUDA platform was set equal to the number of chromosomes while the number of threads on multi-core CPUs was set equal to the number of cores. Therefore, in the TBB-based parallel code, the number of cores is impressive. To investigate this effect, we run the TBB-based parallel code on dual-core and quad-core CPUs.

Crossover operator probability is 0.8 and mutation probability is 0.02. The probability of selection operator, providing that the operator may choose a sample with less fitness, is equal to 0.8. The crossover operator is a single-parent and single-point one and the mutation operator are of the movement type.

The standard data of BCL380, RBU737, PBD984 derived from VLSI data were used to implement TSP. The datasets consisted of 380, 737 and 984 geographical regions of cities [38]. TSP was solved with 1024 to 16,384 populations by applying 100, 200 to 3000 generations on each. The number of offsprings produced in the crossover was between 10%–50% of the size of the initial population. In the next section, we review the results which were acquired from the experiments that were repeated 10 times.

#### Performance evaluation

To evaluate the proposed methods and compare their performances, we reviewed the influence of parameters

such as population size, number of generations, number of crossover-mutation and chromosomes size on the performance of each method.

Evaluation criteria are the implementation time of the GA in the proposed methods and accelerating the parallel versions of the serial method. First, we investigate the cost of switching between the proposed kernels over CUDA platform in different scenarios. Table 1 shows the time required for solving TSP with 380 cities using the proposed set of tertiary kernels with different sizes of population and different numbers of generations. For each case, the time required for switching among kernels as well as their execution time is reported. Table 2 shows the switching and execution time of the kernels in 100 generations, with different numbers of cities and different sizes of population. In Tables 1 and 2, the number of offsprings is 40% and 30% of the population, respectively. As explained in Section 4, since there is no data transmission between kernels and generations (from GPU to CPU and vice versa), increasing the number of generations has not any effect on the switching time. But by increasing the number of cities and the size of population, switching time increases due to the transfer of the initial population from CPU to GPU. However, the switching time is negligible compared with the kernel running time. The total computation time for any CUDA kernel is actually the sum of the switching time and the time of kernel computation.

The plots in Fig. 6 shows the effect of increasing the size of the initial population and the number of generations on the execution time of the serial method, the parallel method on CUDA platform, and the parallel methods formed by exploiting TBB on two/four CPU cores. In this experiment, the size of the new population generated from the crossover operation is 40% of the initial population. By increasing the number of generations, the running time of TSP has increased in all methods.

A remarkable point in this experiment is the impact of the size of population on the running time of CUDA-based parallel code as compared to TBB-based parallel codes running on dual-core and quad-core CPUs. As shown in Fig. 6-a, the execution time of CUDA-based parallel TSP code is worse than TBB-based parallel code on dual-core and quad-core CPUs. In this experiment, all resources of CPUs have been used while in the GPU only up to 1024 threads have been used. Increasing the size of populations increases the level of parallelism and consequently the performance of GPU code is improved. For example, in the case of having a population of size 4096, CUDA-based parallel TSP has been better than TBB-based TSP. The degree of this superiority reaches a maximum when the size of population hits 16,384. In this state, the GPU resources has been well used.



**Table 1** Switching and kernel computation time (ms) in CUDA- 380 cities (BCL380)

Population		Generation							
		100	200	400	800	1000	2000	2500	3000
1024	Switch	0.73822	0.75217	0.76018	0.75258	0.74694	0.75838	0.75547	0.75114
	Kernel	2183.76	4352.68	8687.09	17,355.7	21,697.6	43,378.4	54,213.4	65,068.2
2048	Switch	1.28195	1.47194	1.35182	1.26267	1.2215	1.37934	1.28483	1.28211
	Kernel	2268.34	4520.98	9024.91	18,036.1	22,541.3	45,064	56,323.4	67,587.5
3072	Switch	1.80883	1.80882	1.94954	1.81701	1.83845	2.31515	1.8183	1.80926
	Kernel	2359.75	4700.82	9390.72	18,762.1	23,439.8	46,859.3	58,596.2	70,285.9
4096	Switch	2.33739	2.33973	2.33702	2.52005	2.35171	2.3325	2.3419	2.28941
	Kernel	2362.82	4705.47	9399.08	18,779.1	23,461.8	46,914.9	58,661.3	70,357.7
5120	Switch	2.90762	2.92509	2.86219	2.91802	2.91189	2.97024	3.03762	2.88035
	Kernel	2384.06	4750.92	9484.7	18,944.5	23,687.8	47,371.8	59,201.4	71,052.9
6144	Switch	3.73773	3.4692	3.58517	3.48848	3.40894	3.45525	3.5112	3.60488
	Kernel	2399.68	4779.72	9544.65	19,071.8	23,829.4	47,643.4	59,560	71,438.4
8192	Switch	4.68266	4.48123	4.52781	4.87566	4.90762	4.45749	5.53869	4.46005
	Kernel	2443.86	4876.11	9721.46	19,423.6	24,309.7	48,604.7	60,656.3	72,888.9
10,240	Switch	5.55746	5.61781	5.79314	5.61149	5.62642	5.53069	5.70728	5.61958
	Kernel	2239.25	4434.78	8829.64	17,603.5	21,985.7	43,953.9	54,938.4	66,007.3
16,384	Switch	11.3822	9.32661	8.84053	8.85627	9.85898	9.05573	9.3819	8.99462
	Kernel	2434.82	4814.55	9593.75	19,134	23,902.9	48,031.8	60,033	72,071.8

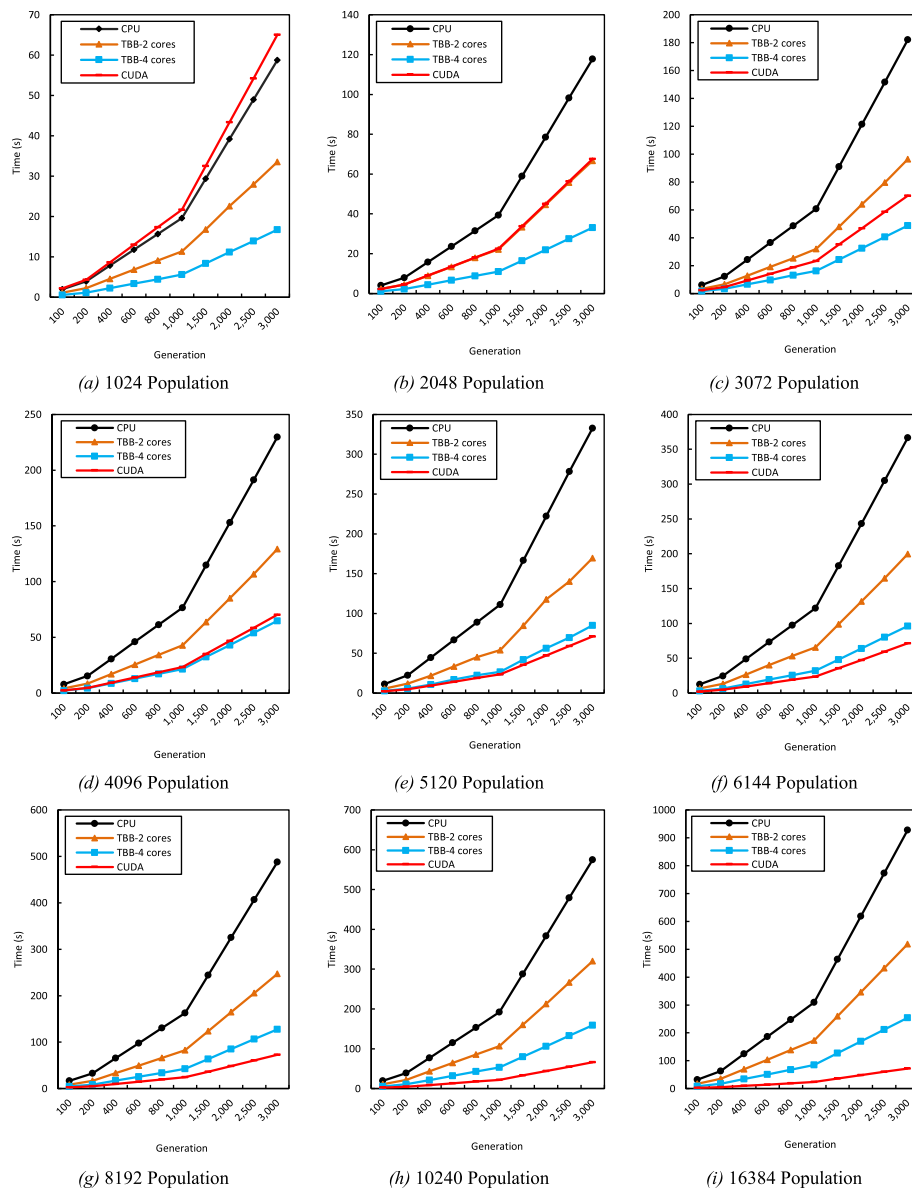
Figure 7 shows the speedup of three parallelization methods, namely TBB-based TSP kernel on dual-core and Quad-core CPUs and CUDA-based GPU kernel with respect to the sequential code in different sizes of population and offsprings created by crossover. As a common setting in this experiment, the maximum number of generations of the GA is set to 100. In all cases, the speedup of TBB on four cores is more than the speedup of TBB on two cores. In addition, the overall

rate of the speedup has not changed with the size of population and the number of offsprings. This is while in the CUDA-based TSP kernel the speedup has changed with both of these parameters. The reason is that both parameters affect the utilization of the resources of GPU.

The effect of the number of chromosomes on the execution time of parallel TSP codes is illustrated in the plots in Fig. 8. In the corresponding experiment, the number of

**Table 2** Switching and kernel computation time (ms) in CUDA- 100 generations

Cities	Population							
	Switch	Kernel	Switch	Kernel	Switch	Kernel	Switch	Kernel
	<b>1024</b>		<b>2048</b>		<b>4096</b>		<b>6144</b>	
380 (BCL380)	0.68951	2175.47	1.1975	2211.08	2.20462	2359.25	3.24037	2366.61
737 (RBU737)	1.15285	4060.45	2.1389	4114.11	3.83376	4367.86	5.60917	4377.8
984 (PBD984)	1.57603	6200.72	3.11869	6301.57	6.60059	6714.43	8.52061	6727.66
	<b>7168</b>		<b>8192</b>		<b>9126</b>		<b>10,240</b>	
380 (BCL380)	3.67074	2138.17	4.96326	2400.57	4.65838	2179.79	5.00744	2198.41
737 (RBU737)	6.61571	4410.09	7.89123	4447.35	8.11848	4503.38	9.41981	4528.84
984 (PBD984)	10.4104	6786.87	11.0019	6829.77	14.361	6917.51	13.9765	6953.94
	<b>16,384</b>		<b>20,480</b>		<b>32,768</b>		<b>65,536</b>	
380 (BCL380)	8.41643	2280.52	10.2355	2632.1	16.026	4905.64	32.1677	9212.24
737 (RBU737)	16.22	4694.13	194,048	4917.1	30.0229	9179.92	58.7788	17,204.9
984 (PBD984)	24.4927	7204.22	28.8571	7572.09	44.0464	14,145.5	91.1372	26,544.6



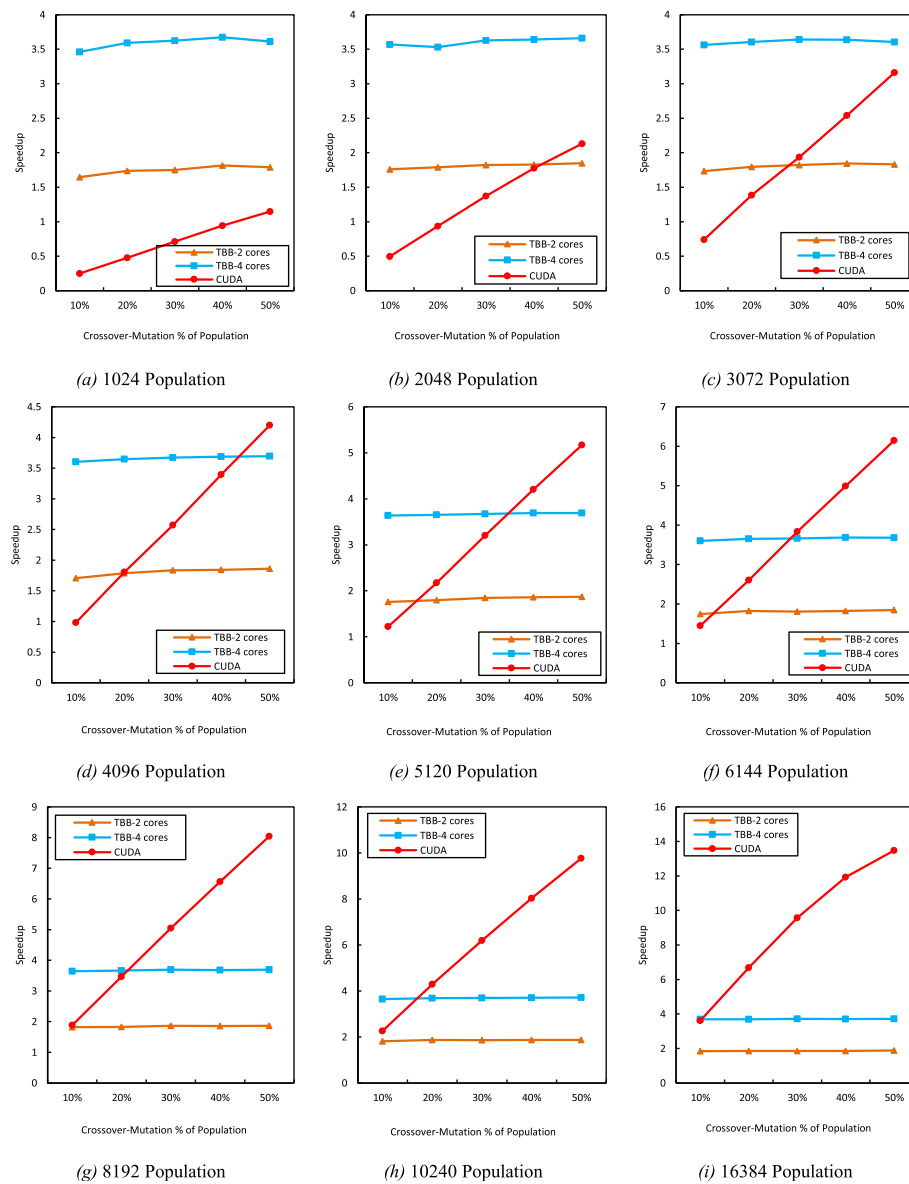
**Fig. 6** The running time of the algorithm over different numbers of generations and different sizes of population. Each subplot is labeled with the corresponding population size

generations and the number of offsprings resulting from each crossover process are kept constant while the size of the population is allowed to be varied. For this experiment, three different datasets with 380, 737 and 984 cities have been used. The size of offsprings is set to be 40% of the initial population during 100 generations.

As shown in Fig. 8, TBB-based code on four cores in the population of less than or equal to 4096 has the least running time and the highest speedup as compared to other methods. But when the population is more than 4096, CUDA has the best performance compared to other methods. This state hits when CUDA computes the GA solution of TSP with 20,480 population; in this

case, each thread calculates a chromosome. As the population grows, each thread should examine more than one chromosome, which increases the running time. This is illustrated in the speedup plots of Fig. 8.

According to the GPU specifications used in this study, the maximum number of threads to run simultaneously is 20,480 threads. Thus, when the initial population exceeds 20,480, the computational load per thread increases, which reduces the acceleration. Another remarkable point in this experiment is the ineffectiveness of changes in chromosome length (number of cities) in the overall running time of the three parallel methods provided by the GA. The maximum speedup values obtained in experimentation of



**Fig. 7** The speedup of the parallel methods with respect to different ratios of offspring on different populations in a TSP with 380 cities. Each subplot is labeled with the corresponding population size

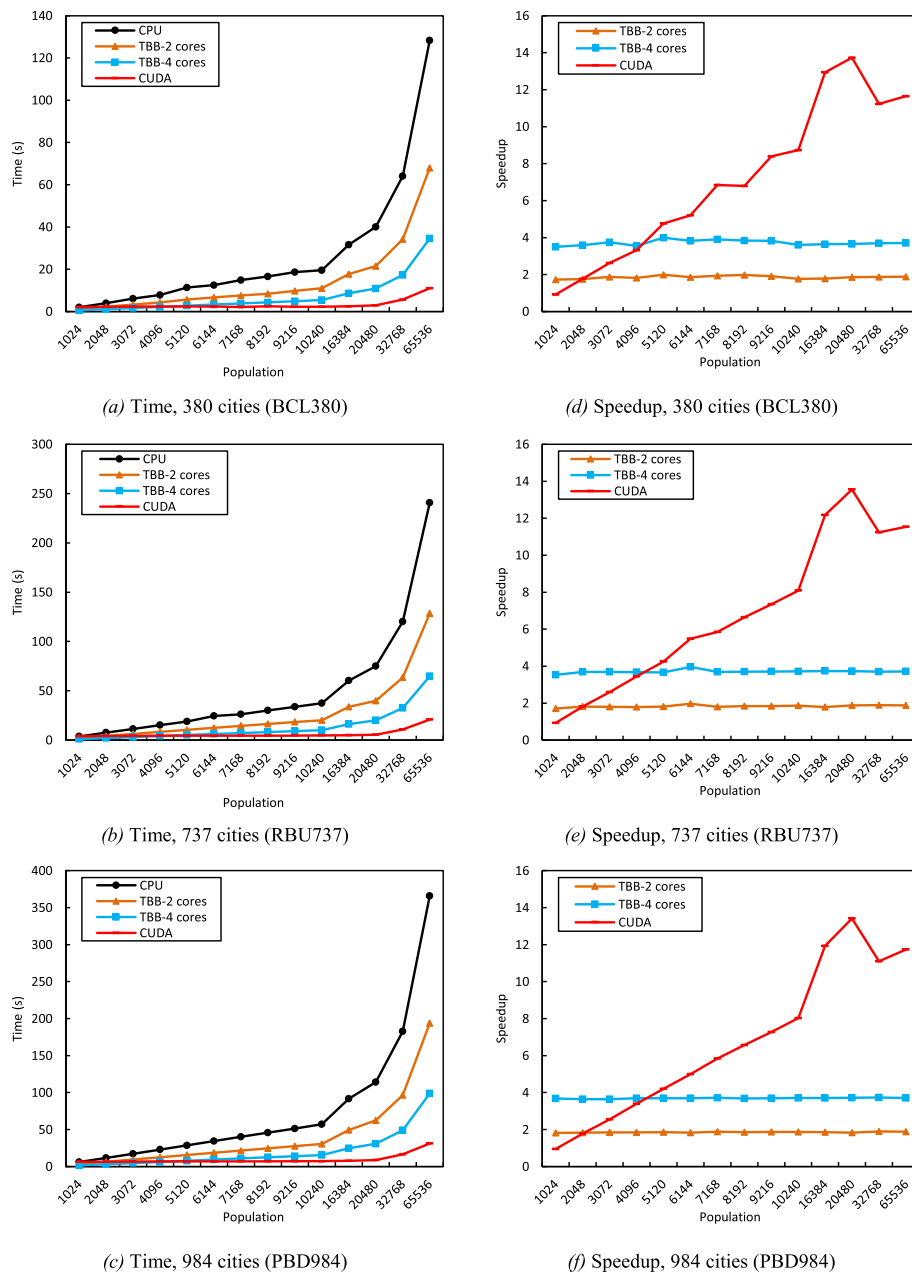
TBB-based TSP on dual-core and quad-core CPUs are 1.99 and 3.99, respectively, while the maximum speedup of CUDA-based TSP on 1280 cores is 13.73.

Finally, according to the experiments, it can be stated that when a genetic algorithm is applied on a small population, the TBB-based method has the best performance. Otherwise, considering the ability of CUDA to define the maximum required number of threads for calculations, the use of CUDA is more efficient.

#### Efficiency evaluation and comparison

To provide a more intuitive illustration of the efficiency of the proposed parallelization method, the efficiency of the

proposed parallelization of the GA solution of TSP on TBB platform is computed and compared in Table 3 with the state-of-the-art methods. As shown in Table 3, the efficiency of the proposed parallel GA on a quad-core system using TBB platform is the highest as compared to the other parallel GA solutions of TSP on multi-core systems. Moreover, the efficiency of multi-core parallelization of GA solution of TSP using TBB platform is 0.9975, which is considerably higher than that of the OpenMP-based parallelization as well as TBB-based parallelization done by Zhu [16]. This result shows that the proposed parallelization method could more optimally exploit the parallel resources of the multi-core CPUs and consequently achieve higher efficiency.



**Fig. 8** The impact of population size on the speedup of parallel methods. Each subplot is labeled with the corresponding population size

**Table 3** Comparing the efficiency of state-of-art parallelizations of GA solution of TSP

Year	Method	Reference	# of Cores	Speed up	efficiency
2013	TBB	Zhu [16]	4	2.55	0.6375
2019	OpenMP	Saxena [25]	4	2	0.5
2019	TBB	Proposed Method	4	3.99	0.9975
2019	TBB	Proposed Method	2	1.99	0.995

## Conclusion

Using efficiently parallelizable optimization algorithms for solving equivalent TSPs of vehicle routing problems is a key in minimizing the costs of any intelligent transportation system with limited profitability margins. Especially, providing a solution exploitable on vehicular cloud computing platform has been recently attracted many interests.

In this paper we presented an enhanced GA solution to TSP problem in VRPs which could be easily and highly parallelized on multi-core and many-core

machines of suitable VCC platforms. We show that the efficient parallelization of genetic algorithms (GAs) on multi-core or many-core systems is affected by the efficiency of the scheduling of hardware resources regarding the concurrency of threads. We proposed a method for efficient parallelization of genetic algorithms on multi-core and many-core systems. In the proposed method, the fitness functions, crossover, mutation and selection functions are implemented in parallel. Next, the proposed method is implementable on many-core and multi-core processors. The second aim of this study was to arrange synchronizable kernels which can use the maximum resources of many-core processors for accelerating the computations. In this regard, three separate kernels were designed to compute the various functions of the genetic algorithm concurrently. These kernels can be replicated on different CUDA blocks on GPU. To synchronize the threads of these blocks, a switching mechanism is used. The time taken to switch between the kernels of different blocks is negligible with regard to the running time of the genetic algorithm. Therefore, the proposed method is highly efficient in synchronizing collaborative threads in the processing of genetic algorithms.

The proposed method was tested for parallelizing a GA-based solution of Traveling Salesman Problem (TSP) on CUDA and TBB platforms with the same settings including the same number of primary population and generations as well as the same ratio of children created by crossover and mutation operators on the same data set. The performance of these two platforms were evaluated based on different criteria such as the running time and speedup of the parallel GA over each of them.

According to the results, the highest speedup of the parallel algorithm on the GPU, the dual-core processor, and the quad-core processor is 13.73, 1.99, and 3.99, respectively. Another significant finding of this study is that the performance of a parallel algorithm on a GPU-like many-core processor, when the initial number of population in the genetic algorithm is low, is much lower than that of a multi-core processor. The reason is that, in a low initial population, parallelization resources in multi-core processors are more efficiently utilized than in the GPU-like many-core system.

The CPU/GPU clusters have recently evolved to become high-performance accelerators for computation-intensive programs [39, 40]. Therefore, future studies should investigate how to extend the proposed method to best use the resources of GPU cluster for GA computations.

#### Abbreviations

API: Application Interface; CPU: Central Processing Unit; CUDA: Compute Unified Device Architecture; GA: Genetic Algorithm; GPU: Graphics Processing Unit; TBB: Threading Building Blocks; TMS: Traffic Management System; TSP: Travelling Salesman Problem; VANET: Vehicular Ad-hoc Network; VCC: Vehicular Cloud Computing; VRP: Vehicle Routing Problem

#### Acknowledgements

Authors thank editor and reviewers for their time and consideration.

#### Authors' contributions

MA, MK, AJ and VM have participated in design of the proposed method and practical implementation. MR and JM has coded the method. MA, MK, AJ and VM have completed the first draft of this paper. All authors have read and approved the manuscript.

#### Authors' information

Not applicable.

#### Funding

Not applicable.

#### Availability of data and materials

Not applicable.

#### Competing interests

The authors declare that they have no competing interests.

#### Author details

<sup>1</sup>Department of Computer Engineering, Engineering Faculty, Bu-Ali Sina University, Hamedan, Iran. <sup>2</sup>Department of Electrical and Electronic Engineering, Shiraz University of Technology, Shiraz, Iran. <sup>3</sup>Computer Engineering Department, Persian Gulf University, Bushehr, Iran. <sup>4</sup>Department of Computing, Macquarie University, Sydney, NSW, Australia. <sup>5</sup>Department of Computer Science and Engineering, SCMS School of Engineering and Technology, Ernakulam, Kerala 683582, India.

Received: 30 November 2019 Accepted: 22 January 2020

Published online: 03 February 2020

#### References

1. Wu J, Zhou L, Du Z, Lv Y (2019) Mixed steepest descent algorithm for the traveling salesman problem and application in air logistics. *Transport Res Part E Logistic Transport Rev* 126:87–102
2. Menon VG, Prathap J (2018) Vehicular fog computing: challenges applications and future directions. In: fog computing: breakthroughs in research and practice. IGI global, pp 220–229
3. Vidal T, Laporte G, Matl P (2019) A concise guide to existing and emerging vehicle routing problem variants. *Eur J Oper Res*
4. Vu DM, Hewitt M, Boland N, Savelsbergh M (2019) Dynamic discretization discovery for solving the time-dependent traveling salesman problem with time windows. *Transp Sci*
5. Li B, Peng Z, Hou P, He M, Anisetti M, Jeon G (2019) Reliability and capability based computation offloading strategy for vehicular ad hoc clouds. *J Cloud Comput* 8(1):21. <https://doi.org/10.1186/s13677-019-0147-6>
6. Whaiduzzaman M, Sookhak M, Gani A, Buyya R (2014) A survey on vehicular cloud computing. *J Netw Comput Appl* 40:325–344
7. Mistareehi H, Manivannan D (2019) Classification, challenges and critical comparison of proposed solutions for vehicular clouds. *Int J Next Gen Comput* 10(1)
8. Ahmed ZE, Saeed RA, Mukherjee A (2019) Challenges and opportunities in vehicular cloud computing. In: cloud security: concepts, methodologies, tools, and applications. IGI global, pp 2168–2185
9. Avraham E, Raviv T (2019) The data-driven time-dependent traveling salesperson problem
10. Giap CN, Ha DT Parallel genetic algorithm for minimum dominating set problem. In: Computing, Management and Telecommunications (ComManTel), 2014 International Conference on, 2014. IEEE, pp 165–169
11. Jain DK, Jacob S, Alzubi J, Menon V (2019) An efficient and adaptable multimedia system for converting PAL to VGA in real-time video processing. *J Real-Time Image Proc* 1–13
12. Munroe S, Sandoval K, Martens DE, Sijkema D, Pomponi SA (2019) Genetic algorithm as an optimization tool for the development of sponge cell culture media. *In Vitro Cell Dev Biol Anim* 55(3):149–158
13. Yasmin S (2019) Linear colour image processing in Hypercomplex algebra guided by genetic algorithms. Dissertaion, University of Essex



14. Lima AA, de Barros FK, Yoshizumi VH, Spatti DH, Dajer ME (2019) Optimized artificial neural network for biosignals classification using genetic algorithm. *J Control Autom Electric Syst* 30(3):371–379
15. Arif MH, Li J, Iqbal M, Liu K (2018) Sentiment analysis and spam detection in short informal text using learning classifier systems. *Soft Comput* 22(21):7281–7291
16. Zhu J, Li Q Application of Hybrid MPI+ TBB Parallel Programming Model for Traveling Salesman Problem. In: *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCoM), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, 2013. IEEE, pp 2164–2167
17. Fujimoto N, Tsutsui S A highly-parallel TSP solver for a GPU computing platform. In: *International Conference on Numerical Methods and Applications*, 2010. Springer, pp 264–271
18. Chen S, Davis S, Jiang H, Novobilski A (2011) CUDA-based genetic algorithm on traveling salesman problem. In: *computer and information science 2011*. Springer, pp 241–252
19. Sánchez LNG, Armenta JJT, Ramírez VHD (2015) Parallel genetic algorithms on a GPU to solve the travelling salesman problem. *Difu100ci@ Revista en Ingeniería y Tecnología, UAZ* 8 (2)
20. Kang S, Kim S-S, Won J, Kang Y-M (2016) GPU-based parallel genetic approach to large-scale travelling salesman problem. *J Supercomput* 72(11):4399–4414
21. Moumen Y, Abdoun O, Daanoun A Parallel approach for genetic algorithm to solve the Asymmetric Traveling Salesman Problems. In: *Proceedings of the 2nd International Conference on Computing and Wireless Communication Systems*, 2017. ACM, p 24
22. Cekmez U, Ozsiginan M, Sahingoz OK Adapting the GA approach to solve Traveling Salesman Problems on CUDA architecture. In: *Computational Intelligence and Informatics (CINTI), 2013 IEEE 14th International Symposium on*, 2013. IEEE, pp 423–428
23. Radford D, Calvert D (2017) A comparative analysis of the performance of scalable parallel patterns applied to genetic algorithms and configured for NVIDIA GPUs. *Procedia Comput Sci* 114:65–72
24. Li C-C, Lin C-H, Liu J-C (2017) Parallel genetic algorithms on the graphics processing units using island model and simulated annealing. *Adv Mech Eng* 9(7):1687814017707413
25. Saxena R, Jain M, Sharma D, Jaidka S (2019) A review on VANET routing protocols and proposing a parallelized genetic algorithm based heuristic modification to mobicast routing for real time message passing. *J Intell Fuzzy Systems* 36(3):2387–2398
26. NVIDIA. NVIDIA CUDA (Compute Unified Device Architecture) Programming Guide, (accessed September 2019) [http://docs.nvidia.com/cuda/pdf/CUDA\\_C\\_Programming\\_Guide.pdf](http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf). Accessed 1 Sept 2019
27. Abbasi M, Rafiee M (2019) A calibrated asymptotic framework for analyzing packet classification algorithms on GPUs. *The Journal of Supercomputing*:1–38
28. Jam S, Shahbahrani A, Ziyabari S (2017) Parallel implementation of particle swarm optimization variants using graphics processing unit platform. *Int J Eng Trans A Basic* 30(1):48–56
29. Abbasi M, Tahouri R, Rafiee M (2019) Enhancing the performance of the aggregated bit vector algorithm in network packet classification using GPU. *Peer J Comput Sci* 5:e185
30. Yip CM, Asaduzzaman A A promising CUDA-accelerated vehicular area network simulator using NS-3. In: *Performance Computing and Communications Conference (IPCCC), 2014 IEEE International*, 2014. IEEE, pp 1–2
31. Intel T (2018) Intel Threading Building Blocks. Available: <http://threadingbuildingblocks.org/>. Accessed 16 Mar 2019
32. Kim CG, Kim JG, Lee DH (2014) Optimizing image processing on multi-core CPUs with Intel parallel programming technologies. *Multimed Tools Appl* 68(2):237–251
33. Hougardy S, Wilde M (2014) On the nearest neighbor rule for the metric traveling salesman problem. *Discret Appl Math*
34. Groba C, Sartal A, Vázquez XH (2015) Solving the dynamic traveling salesman problem using a genetic algorithm with trajectory prediction: an application to fish aggregating devices. *Comput Oper Res* 56:22–32
35. Hussain A, Muhammad YS (2019) Trade-off between exploration and exploitation with genetic algorithm using a novel selection operator. *Complex Intell Syst*:1–14
36. Contreras-Bolton C, Parada V (2015) Automatic combination of operators in a genetic algorithm to solve the traveling salesman problem. *PLoS One* 10(9):e0137724–e0137724. <https://doi.org/10.1371/journal.pone.0137724>
37. Lee E, Lee E-K, Gerla M, Oh SY (2014) Vehicular cloud networking: architecture and design principles. *IEEE Commun Mag* 52(2):148–155
38. VLSI TSP Collection, September 2019, [online] Available: <http://www.math.uwaterloo.ca/tsp/vlsi/index.html>. Accessed 2 Feb 2018
39. Jaros J Multi-GPU island-based genetic algorithm for solving the knapsack problem. In: *Evolutionary Computation (CEC), 2012 IEEE congress on*, 2012. IEEE, pp 1–8
40. Orts F, Ortega G, Garzón EM, Puertas A (2019) Finite size effects in active microrheology in colloids. *Comput Phys Commun* 236:8–14

## Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

**Submit your manuscript to a SpringerOpen<sup>®</sup> journal and benefit from:**

- Convenient online submission
- Rigorous peer review
- Open access: articles freely available online
- High visibility within the field
- Retaining the copyright to your article

---

Submit your next manuscript at ► [springeropen.com](https://www.springeropen.com)