HOSTED BY
ELSEVIER

Alexandria University

Alexandria Engineering Journal

www.elsevier.com/locate/aej
www.sciencedirect.com

Multi-core-, multi-thread-based optimization algorithm for large-scale traveling salesman problem

Xin Wei [*], Liang Ma, Huizhen Zhang, Yong Liu

School of Management, University of Shanghai for Science and Technology, Shanghai 200093, China

Received 29 April 2020; revised 24 June 2020; accepted 28 June 2020
Available online 18 July 2020

KEYWORDS

Multi-core;
Multi-thread;
Traveling Salesman Problem;
Optimization Algorithm

Abstract With the rapid development of general hardware technology, microcomputers with multi-core CPUs have been widely applied in commercial services and household usage in the last ten years. Multi-core chips could, theoretically, lead to much better performance and computational efficiency than single-core chips. But so far, they have not shown general advantages for users, other than for operating systems and some specialized software. It is not easy to transform traditional single-core-based algorithms into multi-core-, multi-thread-based algorithms that can greatly improve efficiency, because of difficulties in computation and scheduling of hardware kernels, and because some programming languages cannot support multi-core, multi-thread programming. Therefore, a kind of multi-core-, multi-thread-based fast algorithm was designed and coded with Delphi language for the medium- and large-scale traveling salesman problem instances from TSPLIB, which can fully speed up the searching process without loss of quality. Experimental results show that the algorithm proposed can, under the given hardware limitations, take full advantage of multi-core chips and effectively balance the conflict between increasing problem size and computational efficiency and thus acquire satisfactory solutions.

1. Introduction

The traveling salesman problem (TSP) is a classic combinatorial optimization problem in operational research [1–3], where the objective is to find the lowest cost of a tour path in which the traveling salesman passes all the given cities exactly once and returns back to the starting one.

As a traditional research subject, many problems can be converted into the TSP or sub-problems of the TSP, in academic fields or practical applications, such as the pre-distribution of logistics [4–6], the process and customization of integrated circuit chips or circuit boards [7,8], production task arrangement [9–11], and even astronomical observations [12]. As a matter of fact, theoretical discussion and application research on the TSP are also beneficial to the solution of other combinatorial optimization problems. Much research progress in this field is derived from the in-depth study of this issue. For example, the TSP is a sub-problem of the Vehicle Routing

[*] Corresponding author.
E-mail address: usstwx@126.com (X. Wei).

Peer review under responsibility of Faculty of Engineering, Alexandria University.

https://doi.org/10.1016/j.aej.2020.06.055

Problem (VRP) providing optimal solutions for traffic prediction and planning [13,14].

With the coming of the big data era, these practical problems now tend to be on a large scale. Therefore, it is of practical significance to study the large-scale TSP and the method used to solve it, which can also be used as a reference for other related optimization problems. When the problem size is small, the optimal solution can be obtained by precise algorithms such as linear programming, dynamic programming, and the branch and bound methods. However, these precise algorithms are all exponential in nature; in other words, the so-called combinatorial explosion will be triggered as the problem size gets larger, which makes their application to larger-sized problems practically impossible.

With the development of hardware technology, as well as the pursuit of high performance of multi-processors, the issue of how to use existing hardware technology to improve the efficiency of algorithms has become a very worthwhile research topic. Based on Turing machines, modern computer architecture is serial in nature, and even with the advent of multi-core processors, it is difficult for traditional serial algorithms to benefit from them. Therefore, corresponding changes in the design and implementation of algorithms are needed in order to adapt to the development of hardware technology, in order to make full use of hardware resources so as to effectively improve the performance and execution efficiency of algorithms.

In the last ten years or so, computers with multi-core CPUs have been widely applied in household and commercial areas. Theoretically, compared to single-core chips, the multi-core chips should lead to better performance and computational efficiency. But so far, except for some operating systems or specialized software, these multi-core processors have not shown general advantages for users. It is not easy to transform traditional single-core-based algorithms into multi-core-, multi-thread-based algorithms that can greatly improve efficiency, because of the difficulties in computation and scheduling of hardware kernels, and because some programming languages cannot support multi-core, multi-thread programming.

Take the TSP, the typical NP-hard problem, for example. A kind of multi-core-, multi-thread-based fast algorithm was designed in this paper and coded with the Delphi language for medium- and large-scale TSP instances from TSPLIB. The algorithm can fully speed up the searching process without loss of quality. Theoretically, the multi-core- and multi-thread-based algorithm can be K times faster than the traditional single-core- and single-thread-based algorithm, where K is the number of CPU cores [15]. Also, the experimental results show that the fast algorithm based on multi-core and multi-thread can, under the given hardware limitations, take full advantage of multi-core chips and effectively balance the conflict between the increasing problem size and computational efficiency and thus acquire satisfactory solutions.

2. The technology of multi-core and multi-threading

The year 1946 witnessed the birth of ENIAC, the first electronic computer. Since then, computer system structures have evolved from mainframes, to vector super machines, to mini-computers, to microcomputers, to computer clusters. And microcomputers have been improved by the upgrading of microprocessors. Meanwhile, the way of software programming is changing from serial programming to parallel programming. Generally speaking, the development of circuit technology can improve the performance of processors by about 20 times, while the development of architecture can improve it by about 4 times, and the development of compilation technology can improve it by about 1.4 times. But now it is hard to maintain these increases mentioned above, and the increase in frequency is limited. Moore's law, which dominated the semiconductor market for nearly four decades, is likely to fail to work in the next decade or two, and multi-core CPUs are starting to take over. The year 2006 was known as "the first year of dual-core," and now, computers equipped with 4 cores, 8 cores, or even 16 cores have already been used in the daily life of ordinary families. Therefore, multi-core processors have become the inevitable products of technology development and application requirements.

Generally speaking, the main parts of computer communication are the processes, which communicate through pipelines or networks, running either on parallel computers or on one node of a computer group. Multi-thread communication consists of a series of threads belonging to the same process, with each thread sharing all the memory of the process and its global variables. Compared with inter-process communication, multiple threads could communicate by sharing global variables with each other, which can be accessed instantly and enjoy higher efficiency in synchronization control, data exchange, and other aspects. Therefore, the greatest advantage of multi-thread-based optimization algorithms lies in the convenient communication between different threads, which is suitable for concurrent operation. Nowadays, the idea of multi-thread has been widely used in some business server areas such as online transaction processing and web services and has played an important role in those areas. It is still relatively rare, however, to use multi-thread algorithms in the field of traditional operations research optimization. Even if such algorithms are occasionally used, they are limited to run in a single-core hardware environment, which has no parallel effect and lacks practical value [16].

As general computer hardware enters into the multi-core era, the biggest challenges for designers of optimization algorithms are as follows: (1) the transition from single-threaded design to multi-threaded design; (2) the transition from single-core-, multi-thread-based algorithm to multi-core-, multi-thread-based programming; (3) the transformation from serial thinking to parallel thinking; (4) the programming capability of distributed algorithm optimization; and (5) paying more attention to the call and design capability of the underlying hardware.

In multi-thread-based algorithm design, the basic unit participating in scheduling and execution is the thread, where multiple threads belonging to the same process are both independent and dependent on each other. There are many advantages to the application of the multi-thread strategy. First of all, the threads are parallel and concurrent and can share the resources of the process. Therefore, there is no need to allocate additional memory space and other resources when creating a thread, which saves time and resources. Second, threads, as the most basic execution units of the operating system, load less content while switching between each other, with fast switching speed and low switching cost. Third, multi-threads running at the same time can effectively take advantage of the hardware resources of multi-core processors and thus can significantly improve the operational efficiency of optimization algorithms. In addition, threads communicate more efficiently because of sharing the same address space and global data.

The famous speedup ratio laws in the high-performance computing field refer to the ratio of the execution time of the optimal serial algorithm to the execution time of parallel programming under the same environment and the same problem scale. The speedup ratio is the core index used to measure the effect of parallel computing, accurately describing the performance gains after the parallelization of algorithmic programs. In a single-core CPU configuration, with regard to some client software, the purpose of multi-threading is to create threads to run some computations in parallel in the background, so as to avoid interfering with the user's operations and improve operational performance. In a multi-core CPU configuration, splitting multiple threads allows optimal computation to be allocated to each CPU core for execution. The number of threads is related to the number of cores. If the number of threads is less than the number of CPU cores, some cores must be idle, resulting in a decreased speedup ratio.

In this paper, the algorithm flow is redesigned according to the characteristics of the large-scale TSP. Thus, in the multi-thread parallel design pattern, the parallelizable parts are identified, and different data blocks are allocated to multiple threads to perform the same operation. In other words, the iterative process of each generation in the algorithm cycle module is reasonably distributed to each thread, so as to reach load balance, maintain effective communication, and conduct lock operations as needed. In this way, if the hardware has K kernels capable of executing threads simultaneously, the execution speed can be theoretically increased by up to K times; of course, the small amount of time required for inter-thread communication must be deducted.

3. New algorithm for the large-scale traveling salesman problem

3.1. λ-opt of the TSP

The optimal solution of the TSP has always been a hot topic in academic circles. As mentioned above, an accurate algorithm with exponential complexity is not suitable for the solution of large-scale problems. Therefore, to meet actual demands, approximate methods or heuristic algorithms are often used to obtain satisfactory solutions to the TSP, such as swarm intelligent optimization, the nearest-neighbor algorithms, the greedy algorithms, and the λ-*opt* algorithm.

The λ-*opt* algorithm was first put forward by Lin in 1965 [12]. It replaces the λ edges on the current path with other λ edges, in order to make the new path shorter than the current one. One of the most effective search algorithms—the famous Lin-Kernighan algorithm—is derived from this idea [17–19]. At each step, this algorithm removes λ edges chosen from the original path and then recombines the $2*\lambda$ points connected to these edges into λ new edges in order to obtain a new path that is shorter than the current solution.

According to the λ-Optimality Principle, when the value of the objective function cannot be improved if any λ new edges are used to replace any existing λ edges in order to obtain a new feasible solution, then the existing feasible solution satisfies λ-Optimality, abbreviated as λ-*opt*. It is not difficult to deduce that if λ-*opt* is satisfied, then λ'-*opt* will also be satisfied, $1 \le \lambda' \le \lambda$; i.e., the downward compatibility is satisfied. Hence, for a TSP with n vertices, as long as we continue attempting to exchange λ edges among the existing feasible paths, then when $\lambda = n$, we can determine the optimal travel path satisfying n-*opt* as a solution of the TSP. However, the amount of work of the λ-exchanges increases exponentially with the increase of λ, and the time complexity of the n-exchanges is up to $O(n^{\lambda})$.

And after a lot of calculations, it turns out that a bigger λ is not always better, especially in the application and solution of large-scale TSPs. Once the problem size gets larger, algorithms of $O(n^3)$ order and above are no longer practical, while algorithms of $O(n^2)$ order or below are not very effective, even some practical algorithms, such as polynomial time algorithms, so there is always a tradeoff between execution speed and solution quality for any algorithm. In practical applications, it is often necessary to design the algorithm flow according to different characteristics of large-scale TSPs and various heuristic strategies.

Some existing methods are as follows:

(1) Clustering large-scale problems into several small-scale TSPs that can be solved separately and easily and then merging these solutions according to some rules in order to generate the solution to the original problem [20];
(2) With the help of some swarm intelligence optimization methods such as bee colony algorithms and particle swarm algorithms, exploring the performance advantages of such heuristic strategies derived from nature or bionics in large-scale problems [21,22];
(3) Adapting some parallel immunity and self-adaptive reduction mechanisms to large-scale problems [23,24].

Although each method mentioned above has a different emphasis, all of them should consider the characteristics of a certain specific TSP or design algorithms according to specific examples. These factors lead to limitations.

This paper takes another point of view and draws performance advantages from the high efficiency of *2-opt* or *3-opt* algorithms combined with other heuristic strategies, based on the principle of λ-*opt* and considering the time and calculation requirements of large-scale problems [25]. Meanwhile, the potential for optimization in existing hardware resources is explored, and a multi-core- and multi-thread-based parallel optimization strategy is integrated into the algorithm design.

The possible iteration cycle in the algorithm is parallelized according to the established communication regulation; that is, the time-consuming information communication in large-scale problems is directly delivered to the hardware layer for implementation, which alleviates the time delay caused by information exchange to a certain extent, not only saving calculation time but also improving solution efficiency.

3.2. Multi-core-, multi-thread-based fast algorithm

Based on the ideas mentioned above and combined with the design strategies of multi-core/multi-thread, this paper develops a kind of near-parallel fast algorithm that can run on general multi-core computers. Focusing on the characteristics of large-scale TSPs, and identifying parallel opportunities for multi-thread algorithms, this paper also presents an efficient multi-core, multi-threaded parallel computing scheme. After comparing with the traditional single-core and single-threaded implementation approaches, experimental results show that the algorithm proposed in this paper can improve calculation efficiency significantly and effectively balance the tradeoff between computational complexity and computing time for large-scale problems.

The core idea of the algorithm includes two levels: path construction and path improvement. First, an extended random k-nearest neighbor method is designed to construct the initial path, and its time complexity is $O(n\log n)$. Next, based on the principle of *λ-opt*, the *3-opt* algorithm with time complexity of $O(n^3)$ is reduced to form a new second-order algorithm for loop improvement, and the time complexity of the new *3-opt* algorithm is $O(n^2)$.

The main order-reduction strategy originated from Dueck's experiment on the TSP for 442 cities [26]. The study showed that there was little significance to carrying out the edge exchange between cities that were far away, and that it was necessary only to select several cities that were closer to each other to carry out this exchange.

In view of the reasons mentioned above, the order-reduction strategy in this paper is that a subset of the possible edges for exchange is randomly selected from the nearest edges, instead of traditionally traversing all the possible edges in a large-scale problem. In this way, the time and resources consumed by edge exchange between cities with a long distance can be avoided. Meanwhile, the influence of random factors can be offset by the increased cyclical iteration under a multi-threaded strategy.

In addition, with the aid of multi-core-, multi-thread-based parallel optimization, it can not only resolve the defect that the iterative computation time is too long for large-scale problems but also improve the quality and accuracy of the solution without increasing computation time.

In this paper, the multi-core- and multi-thread-based fast algorithm for the large-scale TSP and its solution flow can be described and expressed in Fig. 1.

In Step 2, k is determined by the current number of iterations myK: when $k <$ the default $s$max, set $k = myK$; otherwise, $k = random(myK)$. The default $s$max can be preset to different values, and the heuristic rules can be used for TSP problems of different sizes. For example, when $s\text{max} = 50$, the 50 nearest edges will be considered. Of course, with the increase of problem size, the value of $s$max can be adjusted according to different situations. In this way, the classical nearest-neighbor strategy can be integrated into the algorithm. If the number of iterations is no more than the number of nearest neighbor points considered, then try the latter with every possible number; otherwise, it is generated randomly within the preset range.

In Step 3, the strategy of transition probability was borrowed from ant colony optimization, and the transfer factor $PValue$, according to characteristics of large-scale problems, was improved as follows:

```
for $j = 1 \to mys\text{max}$
begin
  $PValue \leftarrow 0$;
  for $i = 1 \to mys\text{max}$
$PValue = PValue + 1/i$;
  $PValue = (1/j)/PValue$;
end
```

There are two factors that affect transition probability $PValue$: the distance between nodes and the number of nearest neighbor points $mysmax$. In the large-scale problem, the larger the distance (i.e., larger j), the smaller the probability of being selected; the larger the number of nearest neighbor points (i.e., larger mysmax), the greater the diversity of choice, which reduces the risk of falling into local optimization.

If the complete *3-opt* algorithm is embedded, the time complexity of the whole new algorithm is $O(L\bullet n^3)$, while if the improved *3-opt* algorithm is embedded, the time complexity is $O(L\bullet n^2)$, where L is the preset number of iterations.

4. Experiments

Since the 1950s, many numerical test examples have been generated in the study of TSP algorithms, most of which have concerned practical problems of various countries around the world. These test datasets can be downloaded from http://www.math.uwaterloo.ca/tsp/world/countries.html, which contains TSP examples from selected countries around the globe.

We mainly use some data instances from this website to verify the performance of the modified algorithm. The new algorithm, applicable to both Euclidean distance and absolute distance (Manhattan distance), is coded with Embarcadero Delphi for Windows 7 and Windows10 environments; in addition, we provided single-thread and multi-thread options for comparison of running times. The hardware platform is a mobile workstation with Intel Xeon E-2186 M CPU, 12 logic processors, and 32G RAM.

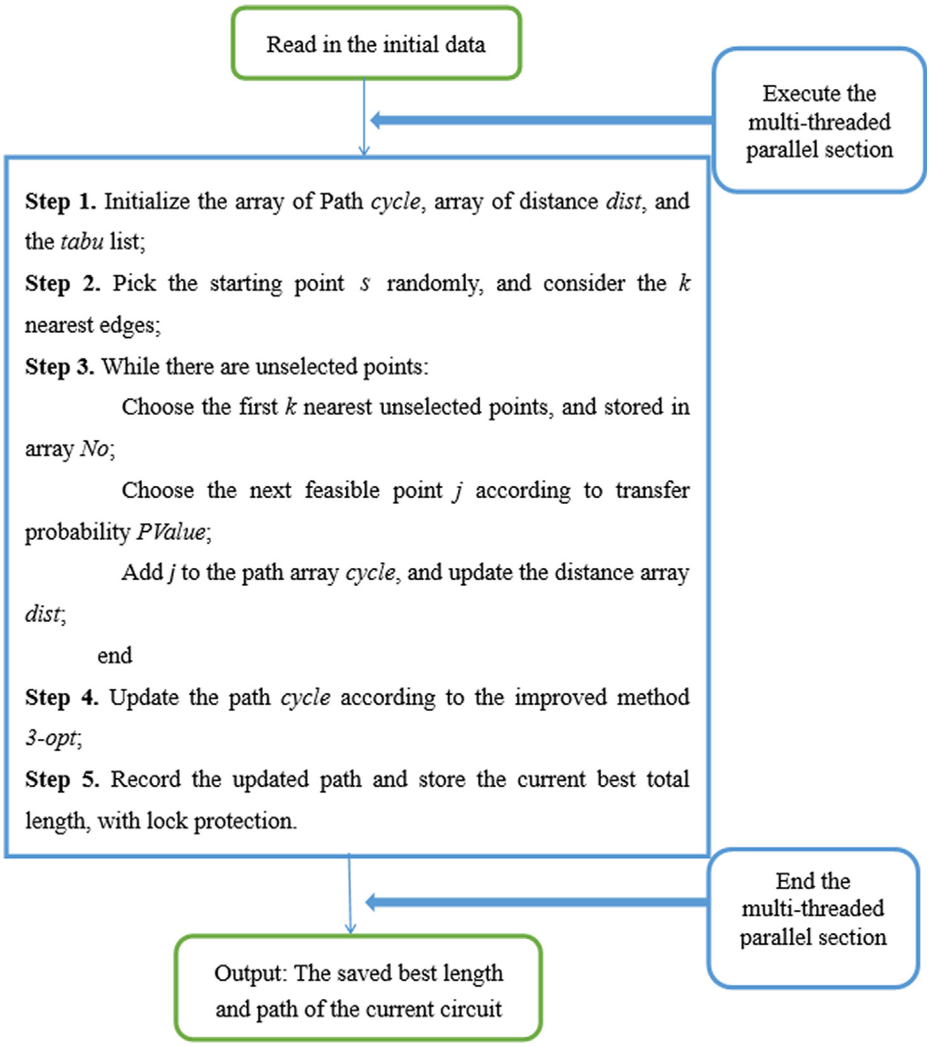
Experiment 1: Small- and medium-sized TSP for verification of the effectiveness of the algorithm

First, some small- and medium-sized problems in TSPLIB were selected in order to show the effectiveness of the multi-core- and multi-thread-based optimization algorithm designed in this paper. Table 1 shows the results after 1200 iterations, each running for 10 rounds, where n is the scale of the problem. We can compute b, the deviation ratio of a particular example, as follows:

$$b = \frac{L_2 - L_1}{L_1} \tag{1}$$

L_1 represents the optimal solution of each example, and L_2 represents the optimal solution obtained by the proposed algorithm. M is the average value of the results after 10 rounds of operation.

It can be seen that the multi-core- and multi-thread-based optimization algorithm proposed in this paper has excellent performance for small- and medium-sized problems and can obtain the optimal solutions in TSPLIB.


Read in the initial data
Execute the multi-threaded parallel section
End the multi-threaded parallel section
Output: The saved best length and path of the current circuit

```
Step 1. Initialize the array of Path cycle, array of distance dist, and the tabu list;
Step 2. Pick the starting point s randomly, and consider the k nearest edges;
Step 3. While there are unselected points:
            Choose the first k nearest unselected points, and stored in array No;
            Choose the next feasible point j according to transfer probability PValue;
            Add j to the path array cycle, and update the distance array dist;
        end
Step 4. Update the path cycle according to the improved method 3-opt;
Step 5. Record the updated path and store the current best total length, with lock protection.
```

Fig. 1 Flow diagram of the multi-core and multi-thread fast algorithm.

Table 1 Results of Small- and Medium-sized TSPs.

No.	Instance	n	L_1	L_2	b	M
1	Eil51	51	426	426	0.00	426
2	Berlin52	52	7542	7542	0.00	7542
3	St70	70	675	675	0.00	675
4	Pr76	76	108,159	108,159	0.00	108,159
5	Kroa100	100	21,282	21,282	0.00	212,282
6	Krob100	100	22,141	22,141	0.00	22,157
7	Kroc100	100	20,749	20,749	0.00	20,749
8	Krod100	100	21,294	21,294	0.00	21,300
9	Kroe100	100	22,068	22,068	0.00	22,070
10	Eil101	101	629	629	0.00	629

In those experiments, we found that the influence of random factors in the algorithm can be offset by increasing the number of iterations, so that the optimal solution of the problem can be directly reached in only one run, especially for small- and medium-sized problems.

The multi-core and multi-threading method reasonably allocates the added iterative part of the algorithm to execute in parallel in the multi-threads; therefore, it will not affect the running speed too much and can still obtain the optimal solution of the problem quickly.

To investigate the advantage of the improved algorithm and reflect accurately the performance gains after parallelization, we can compute the speedup ratio under the multi-core and multi-threaded mode. If T_1 is the computation time of parallel execution of multi-threading, and T_2 is the traditional single-thread execution time, then the speedup ratio S can be computed as follows:

$$S = \frac{T_2}{T_1} \tag{2}$$

The execution time of each example and the speedup ratio are shown in Table 2.

Experiment 2: Large-scale TSP for further verification of the optimization performance of the new algorithm

In fact, there are different requirements in practical applications having different TSP scales. It is more important to obtain the optimal solution of the problem within an acceptable time if the problem size is small, such as in Experiment 1. For large-scale problems, however, it is even more critical to shorten the computation time, as long as the deviation rate is acceptable. The multi-core and multi-threaded parallel optimization strategy in this paper can improve the operational efficiency greatly and save execution time of the algorithm effectively, especially for some large-scale scalability problems, where an acceptable approximate solution can be found in a limited time.

We selected some large-scale problems in TSPLIB and ran each for 10 rounds. In order to balance the calculation effect and running time, each round was iterated 10 times, and the results are shown in Table 3. Just like Experiment 1, the execution time of each example and the speedup ratio of the large-scale problem are shown in Table 4.

It can be seen that the results in the table can be obtained within 10 s for hundreds of scale TSP, and thousands of scale problems can also be solved in a reasonable time. The deviation between the solution results and the published best results is less than 0.04, which can meet the demands of practical use.

In addition, tens of thousands of TSP scale examples were also solved in this study, in order to further validate the proposed multi-core and multi-thread-based optimization algorithm. Here, two representative results are presented:

(1) vm22775: the 22775 cities in Vietnam

This took 85.169 s after 10 multi-threaded iterations. The total length of the loop was 715,280. The known optimal value was 569,288 and the deviation rate was 0.256.

(2) cn71009: the 71009 cities in China

This took 412.825 s after 10 multi-threaded iterations. The total length of the loop was 553,990. The known optimal value was 4,566,563 and the deviation rate was 0.213.

It can be seen that both of these larger-scale examples obtained acceptable solutions within a reasonable length of time by means of the multi-core and multi-threaded method. The time cost would be increased several dozen times or even more if the traditional single-core and single-threaded method were adopted Fig. 1.

In terms of running time, the multi-core and multi-threaded strategy enjoys obvious advantages over the single-threaded one for TSPs of any size. In single-threaded mode, the running time increases sharply with the increase of problem size, but in multi-threaded mode, it increases slowly, which effectively avoids the defect of sensitivity to problem size in traditional single-threaded mode. Figs. 2 and 3 compare the running time variation trend of TSPs of different sizes, under multi-threaded and single-threaded strategies, respectively.

Meanwhile, the experimental results show that the larger the scale of problem solved by the optimization algorithm based on multi-core and multi-threading strategy, the more obvious the efficiency advantage of parallel computing. And the performance of the speedup ratio improves more and more significantly with the increase of problem size. As shown in Fig. 4, the speedup ratio of different problem sizes indicates this variation trend clearly.

The reason for these improvements is that the application of the multi-core and multi-threading strategy described in this paper can maximize the utilization of system hardware resources: the experiments show that the utilization can reach 100%. That is, it can make full use of the free CPU kernels and allocate information communication directly to the bottom of the hardware system for implementation, greatly reducing the running time of the algorithm. Therefore, it has obvious advantages compared with the traditional single-threaded algorithm.

We found that the actual running time of the algorithm is related to the hardware environment of the system, i.e., the

Table 2 Comparison of Computation Time for Small- and Medium-sized TSPs.

	Instance	n	T_1	T_2	S
1	Eil51	51	0.677	5.625	8.309
2	Berlin52	52	0.695	6.667	9.593
3	St70	70	1.645	14.866	9.037
4	Pr76	76	2.088	19.334	9.26
5	Kroa100	100	4.557	44.208	9.701
6	Krob100	100	4.365	43.581	9.984
7	Kroc100	100	4.671	44.208	9.464
8	Krod100	100	4.773	43.27	9.066
9	Kroe100	100	4.986	44.501	8.925
10	Eil101	101	4.453	40.126	9.011

Table 3 Results of Large-scale TSPs.

	Instance	n	L_1	L_2	b	M
11	Lin318	318	42,029	42,603	0.01	42,920.3
12	Rat575	575	6773	7026	0.04	7056.8
13	D657	657	48,912	50,562	0.03	50,734.6
14	Rat783	783	8806	9143	0.04	9168.4
15	Pr1002	1002	259,045	266,148	0.03	269,071.7
16	Pr2392	2392	378,032	393,269	0.04	395,604.3
17	Pcb3038	3038	137,694	143,328	0.04	143,815.1
18	Fl3795	3795	28,772	29,274	0.02	29,366.8
19	Rl5915	5915	565,530	583,695	0.03	589,464.6
20	Rl5934	5934	556,045	574,556	0.03	577,839.1

Table 4 Comparison of Computation Time for Large-scale TSPs.

	Instance	n	T_1	T_2	S
11	Lin318	318	0.8	7.71	9.638
12	Rat575	575	4.117	39.851	9.68
13	D657	657	6.857	72.374	10.555
14	Rat783	783	9.135	96.594	10.574
15	Pr1002	1002	26.851	324.09	12.07
16	Pr2392	2392	365.578	4579.101	12.526
17	Pcb3038	3038	708.465	8933.463	12.61
18	Fl3795	3795	1099.239	14,189.498	12.908
19	Rl5915	5915	2792.875	43,206.443	15.47
20	Rl5934	5934	2897.924	47,928.989	16.539

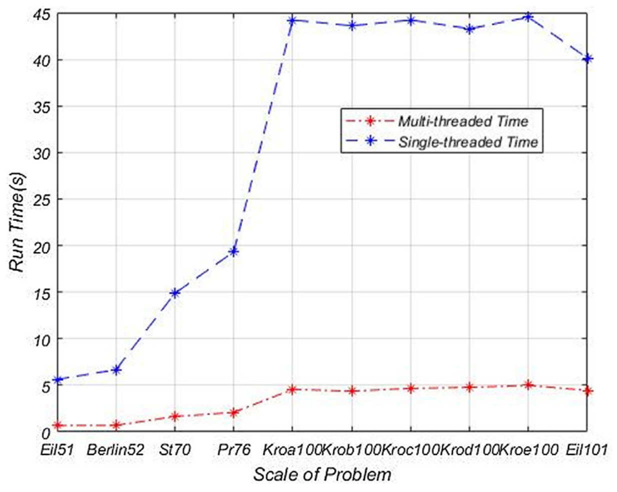


Fig. 2 Comparison of Running Time for Small- and Medium-sized TSPs.

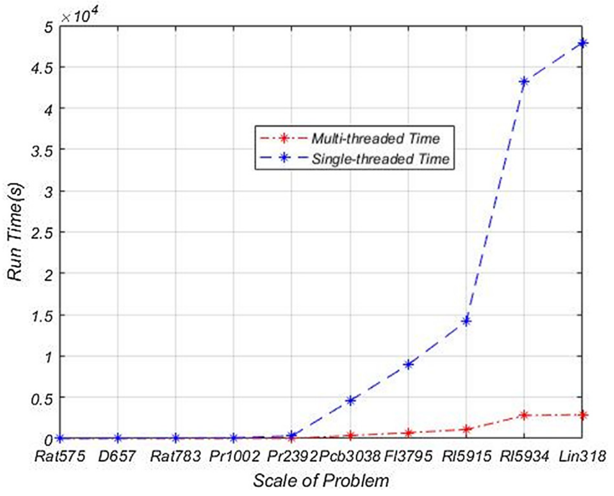


Fig. 3 Comparison of Running Time for Large-scale TSPs.

number of multi-cores and the number of available threads. The more the threads involved in the computation, the more the advantages, and the more the effective balance between problem size growth and computational efficiency.

Of course, the algorithm experiments in this study were run on current general computing equipment. If the computing equipment were upgraded and the algorithm were run on computer clusters, it could be applied to much larger-scale problems and obtain more efficient optimization results. However, the multi-core and multi-threaded parallel strategy is one of the main approaches for improving the efficiency of the algorithm without increasing the hardware cost.

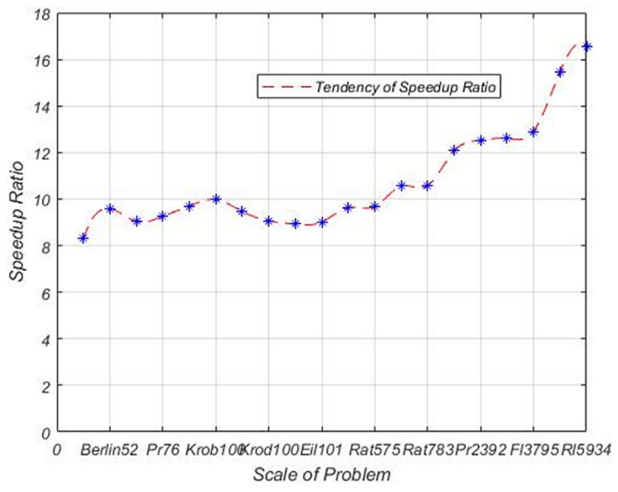


Fig. 4 Tendency of Speedup Ratio.

5. Conclusions

With the coming of the big data era, large-scale problems present a severe challenge to all the optimization methods. For large-scale problems, it has become a dilemma: how to improve the solution efficiency and look for effective balance between solution speed and effect. Meanwhile, with the development of hardware technologies, there are more and more processor cores integrated on one single chip. Thus, the question of how to make full use of the existing hardware resources to improve the execution efficiency of algorithms has become a new idea of algorithm improvement.

Taking full advantage of hardware technologies and integrating the multi-core and multi-threading strategy into the design of optimization algorithms, so as to combine the bottom hardware technologies with the top algorithm design and realize the parallel optimization of ‘‘from bottom to up,’’ is one of the ideas to solve this dilemma. Here, ‘‘bottom’’ refers to the underlying hardware technology, while ‘‘top’’ refers to the top-level hardware technology. Multi-core and multi-threading computation can be faster and more efficient, but it also brings new opportunities and challenges to algorithm designers.

This paper describes and implements a kind of multi-core-, multi-thread-based optimization algorithm, which realizes parallel computation by running the multi-threads at the same time. Experimental results show that this algorithm can greatly improve computational performance and obtain satisfactory results in a short time. It is believed that in the era of big data, there are more and more demands for computer performance. The implementation of multi-core, multi-threaded strategy with multi-core technology will be one of the important directions in future optimization calculations, especially in the field of intelligent optimization.

Declaration of Competing Interest

The authors declared that there is no conflict of interest.

Acknowledgments

This work was supported in part by the Ministry of Education’s Fund for Research in the Humanities and Social Sciences of China (Grant No. 16YJA630037) and in part by the Key Project of the Shanghai Soft Science Research Program (Grant No. 18692110500).

References

[1] A. Subramanyam, C.E. Gounaris, A branch-and-cut framework for the consistent traveling salesman problem, Eur. J. Oper. Res. 248 (2) (2016) 384–395.
[2] N.K. Santosh, P.P. Abraham, The bottleneck TSP, Travel. Salesman Problem Variat. 12 (1) (2007) 697–735.
[3] Q. Wang, X.P. Liu, Solution for travelling salesman problem based on flowing water algorithm, Forecasting 33 (1) (2014) 65–69.
[4] M. Francesca, P. Guido, T. Roberto, The multi-path traveling salesman problem with stochastic travel costs: building realistic instances for city logistics applications, Transp. Res. Procedia 3 (1) (2014) 528–536.
[5] H. Liu, M. Chen, Research on the distribution system simulation of large company’s logistics under internet of things based on traveling salesman problem solution, Cybern. Inf. Technol. 16 (5) (2016) 78–87.
[6] M. Liu, P.Y. Zhang, New hybrid genetic algorithm for solving the multiple travelling salesman problem: an example of distribution of emergence materials, J. Syst. Manage. 23 (2) (2014) 247–254.
[7] A.F. Alkaya, E. Duman, Application of sequence-dependent traveling salesman problem in printed circuit board assembly, IEEE Trans. Compon. Packag. Manuf. Technol. 3 (6) (2013) 1063–1076.
[8] S. Anand, S. Hartmut, M. Christoph, Alternating tsp and printed circuit board assembly, Electr. Notes Discrete Math. 3 (1) (1999) 179–183.
[9] S. Das, N. Ahmed, A travelling salesman problem (TSP) with multiple job facilities, Opsearch 38 (4) (2001) 394–406.
[10] N. Zhang, Design and implementation of walking beam manipulator in automatic production line based on PLC, Journal Européen des Systèmes Automatisés 52 (2) (2019) 173–178.
[11] F.G.E. Anaya, G.E.S. Hernandez, M. Tuoh, J.C. Seek, Solution of the job-shop scheduling problem through the traveling salesman problem, Revista Iberoamaricana De Automatica e Informatica Industrial 13 (4) (2016) 430–437.
[12] W.J. Cook, Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation, Princeton University Press, 2011.
[13] A. Hajjam, J.C. Creput, A. Koukam, From the TSP to the dynamic VRP: an application of neural networks in population based metaheuristic, Metaheurist. Dyn. Optimiz. 433 (1) (2013) 309–339.
[14] Y.P. Lu, X. Pei, C.Z. Zhang, H.Y. Luo, B. Liu, Z.D. Ma, Design of multimodal transport path optimization model and dual pheromone hybrid algorithm, Journal Européen des Systèmes Automatisés 52 (5) (2019) 477–484.
[15] W.M. Zhou, Multi-Core Computing and Programing, Huazhong University of Science and Technology, Wuhan, 2009.
[16] C.J. Li, Q.M. Zhang, Multi-thread evolutionary algorithm to TSP, Comput. Eng. Design 26 (7) (2005) 1744–1750.
[17] S. Lin, Computer solutions of the traveling Salesman Problem, Bell Syst. Tech. J. 44 (1) (1965) 2245–2269.
[18] S. Lin, B. Kernighan, An effective heuristic algorithm for the travelling salesman problem, Oper. Res. 21 (2) (1973) 498–516.

[19] D.S. Johnson, Local optimization and the traveling salesman problem, in: International Colloquium on Automata, Languages, and Programming, Springer, Berlin, Heidelberg, 1990, pp. 446–461.

[20] X.B. Hu, X.Y. Huang, R. Yuan, J.J. Yi, PASCP apply to large-scale TSP problem, Comput. Simulat. 21 (7) (2004) 52–54, 185.

[21] J.W. Zhang, Solving large-scale TSP with adaptive hybrid PSO, Comput. Appl. Softw. 32 (12) (2015) 265–269.

[22] W.Y. Dong, X.S. Dong, Y.F. Wang, Improved artificial bee colony algorithm for large scale colored bottleneck travelling salesman problem, J. Commun. 39 (12) (2018) 18–29.

[23] Y.T. Qi, L.C. Jiao, F. Liu, Parallel artificial immune algorithm for large-scale TSP, ACTA Electron. Sin. 36 (8) (2008) 1552–1558.

[24] Y.T. Qi, F. Liu, L.C. Jiao, Immune algorithm with self-adaptive reduction for large-scale TSP, J. Softw. 19 (6) (2008) 1265–1273.

[25] H.P. Sheng, L. Ma, Modified great deluge algorithm for large-scale travelling salesman problem, J. Chin. Comput. Syst. 33 (2) (2012) 259–262.

[26] G. Dueck, New optimization Heuristics: the great deluge algorithm and the record-to-record travel, J. Comput. Phys. 104 (1) (1993) 86–92.