Received: 24 January 2023 | Revised: 24 July 2023 | Accepted: 31 July 2023

DOI: 10.1002/cpe.7891

RESEARCH ARTICLE

WILEY

# A new parallel tabu search algorithm for the optimization of the maximum vertex weight clique problem

Özcan Dülger[1,2] | Tansel Dökeroğlu[3]

[1]Department of Computer Engineering, Middle East Technical University, Ankara, Turkey

[2]Department of Computer Engineering, Artvin Coruh University, Artvin, Turkey

[3]Department of Software Engineering, TED University, Ankara, Turkey

**Correspondence**
Özcan Dülger, Department of Computer Engineering, Middle East Technical University, Ankara, Turkey.
Email: odulger@ceng.metu.edu.tr

**Present address**
Özcan Dülger, Faculty of Engineering, Artvin Coruh University, Seyitler, Artvin, Turkey

## Summary

The efficiency of metaheuristic algorithms depends significantly on the number of fitness value evaluations performed on candidate solutions. In addition to various intelligent techniques used to obtain better results, parallelization of calculations can substantially improve the solutions in cases where the problem is NP-hard and requires many evaluations. This study proposes a new parallel tabu search method for solving the Maximum Vertex Weight Clique Problem (MVWCP) on the Non-Uniform Memory Access (NUMA) architectures using the OpenMP parallel programming paradigm. Achieving scalability in the NUMA architectures presents significant challenges due to the high complexity of their memory systems, which can lead to performance loss. However, our proposed Tabu-NUMA algorithm provides up to 18× speed-up with 64 cores for ten basic problem instances in DIMACS-W and BHOSLIB-W benchmarks. And it improves the performance of the serial Multi Neighborhood Tabu Search (MN/TS) algorithm for 38 problem instances in DIMACS-W and BHOSLIB-W benchmarks. We further evaluate our algorithm on larger datasets with thousands of edges and vertices from Network Data Repository benchmark problem instances, and we report significant improvements in terms of speed up. Our results confirm that the Tabu-NUMA algorithm is among the best recent algorithms for solving MVWCP on the NUMA architectures.

**KEYWORDS**

maximum vertex weight clique problem, OpenMP, optimization, parallel search, tabu search

## 1 | INTRODUCTION

Finding the complete subgraph of maximum cardinality in a graph is called the Maximum Clique Problem (MCP).[1] The Maximum Vertex Weight Clique Problem (MVWCP) is a different version of MCP that aims to discover a clique having the largest total weight value of vertices.[2-5] Formally, in a graph $G = (V, E)$ with a set of vertices $V = \{1, \ldots, n\}$ and a set of edges, $E \subseteq V \times V$. $w : V \rightarrow Z^+$ is a function that gives a positive value to each vertex $v \in V$. Simply, we try to find a clique with a maximum value that is the total sum of numbers in the vertices of the clique. MVWCP is a maximization problem that tries to find a clique in $G$ with a maximum $w$. Bioinformatics,[6] coding theory,[7] protein structure prediction,[8] computer vision,[9] combinatorial auctions,[10] cluster detection[11] are some of the areas that MVWCP has been applied. Due to the intractable nature of MVWCP, its large problem instances cannot be solved to the best value in reasonable execution times.

Metaheuristic algorithms are the best tools to find solutions for large NP-Hard problems when it is impossible to solve them with exact algorithms.[12,13] Tabu Search is one of the best metaheuristic algorithm to deal with these kinds of problems.[14] Tabu Search can accept worsening moves if no improving move is available. In addition to this, there is a tabu list that discourages the search for the duplicate solutions. Tabu Search uses discrete structures that mark the previously visited solutions. If a candidate solution is visited before in a defined period, it is not permitted to visit this

  

configuration again. Wu et al.[5] introduce the Multi-Neighborhood Tabu Search (MN/TS) algorithm for the solution of MVWCP. MN/TS uses a unified neighborhood and a tabu search mechanism with a restart strategy to provide diversification. The MN/TS algorithm is evaluated on 136 problem instances (DIMACS, BHOSLIB and set packing). The MN/TS algorithm is reported to outperform state-of-the-art algorithms.

The performance of the MN/TS algorithm depends on explorations and exploitation techniques and the number of fitness evaluations it performs. As the number of diversified fitness calculations increases, the probability of obtaining a (near)-best solution also increases. In this study, we develop a Non-Uniform Memory Access (NUMA) architecture-specific parallel version of the MN/TS algorithm. We aim to report better results with more fitness calculations using many cores. And we also aim to obtain results the same as the MN/TS algorithm with shorter execution times. The challenging part of the parallel algorithms is the difficulty in designing scalable shared-memory parallel algorithms (i.e., NUMA). Although numerous cores exist on the NUMA architecture, due to the contention during the access of the memory segments, scalability aims to decrease as we add a new core to the problem's solution. We mention this issue in detail in Section 3.

We have mainly two aims in this study. In the first one, we distribute the workload of the serial MN/TS algorithm to the cores (up to 64 cores) in the NUMA architecture. We obtain similar results with the serial MN/TS algorithm along with better execution times. We ensure speed-up ratios up to 18 times for ten basic problem instances in DIMACS-W and BHOSLIB-W benchmarks. In this approach, the processes (threads) co-operate with each other to obtain improved execution times. In the second one, we find more solutions by running the different instances of the serial MN/TS algorithm in all cores (64) at the same time. We obtain 64 times more solutions along with similar execution times compared to the serial MN/TS algorithm. The processes (threads) do not cooperate with each other directly. However, their solutions are reduced (the maximum ones are selected) in order to find the overall result of the MN/TS algorithm. Getting many new solutions at a time helps us to obtain better solutions compared to the serial MN/TS algorithm. To verify the performance of Tabu-NUMA in this approach, we focus on 38 problems of DIMACS-W and BHOSLIB-W benchmarks that have not been solved to the best value in every restart of the MN/TS algorithm. We improve the success rate of 21 problem instances out of 38 in near best known values. The comparisons with state-of-the-art metaheuristic algorithms show that our proposed Tabu-NUMA algorithm is competitive and has the potential to improve its performance as additional cores are provided. Our contributions to this study can be summarized as follows:

- A new parallel tabu search algorithm for optimizing MVWCP is developed using the NUMA architectures in the literature for the first time.
- The performance of the serial Multi Neighborhood Tabu Search (MN/TS) algorithm is improved.
- Significant speed-up is provided although this is a bottleneck of the NUMA architectures.
- Alternative random number generators are inspected, and their performances are analyzed.
- Shortest execution times in the literature are obtained for large undirected graphs to optimize MVWCP.

Related studies are reported in Section 2. Our proposed algorithm Tabu-NUMA, OpenMP and the NUMA architecture are introduced in Section 3. Section 4 details the performance evaluation of the experiments and the comparisons of Tabu-NUMA to state-of-the-art algorithms. Concluding remarks and future works are provided in Section 5.

## 2 | RELATED WORK

Wu et al.[5] introduce a tabu search with a unified neighborhood using a restart technique to diversify the initial solutions. The algorithm is verified to be effective on problem instances of DIMACS and BHOSLIB benchmarks. This is the algorithm for which we develop a parallel version using OpenMP. Hosseinian et al.[15] explore the MCP and the Maximum Edge Weight Clique (MEWC) problem. The authors propose an exact algorithm for the MEWC problem. Chu et al.[16] present a Local Search technique for MVWCP by assembling clique construction, and graph reduction.

Wang et al.[17] develop two local heuristic search algorithms (LSCC) for MVWCP. The authors propose a heuristic to reduce cycling in local searches. Zhou et al.[18] introduce PUSH to generalize the well-known add and swap operators for MVWCP. The PUSH operator is tested with two tabu search algorithms. The results on DIMACS and BHOSLIB benchmarks indicate that the algorithms are competitive with contemporary algorithms. Nogueira and Pinheiro[19] propose a local search technique for massive graphs with Central Processing Unit-Graphics Processing Unit (CPU-GPU) architectures for MWCP. Two new neighborhood structures are introduced in the algorithm. The neighborhoods are inspected using a procedure mapped onto GPU-based massively parallel architecture.

Jiang et al.[20,21] describe a branch-and-bound solution model for MVWCP that incorporates a novel two-stage Maximum Satisfiability problem (MaxSAT) reasoning approach. McCreesh et al.[22] propose algorithms for other families of benchmark instances from graph coloring. Li et al.[23] design a local search to solve the MEWCP problem.

Cai and Lin[24] explore techniques for solving MWCP quickly in very large graphs. The authors move between the graph reduction and the construction of cliques. They propose three new methods in the algorithm, FastWClq. Nogueira et al.[25] propose a hybrid technique for maximum weight independent set (MWIS). The new algorithm is better than the well-known heuristics for MWIS and works well for MWCP. Wang et al.[26] reshape

MVWCP into the General Binary Quadratic Programming (BQP) model. Later, the authors solve the problem using a probabilistic Tabu Search metaheuristic. Wang et al.[27] propose local search techniques for large graphs using strong configuration checking and walk perturbation methods. Chu et al.[28] develop a method using Programming by Optimization combining many effective heuristics. They achieve substantial improvements in solving MVWCP.

An algorithm is developed by Aksan et al.[29] using multi-core shared-memory architectures. The authors enhance the Breakout Local Search (BLS) heuristic. In this study, similar and local optima solutions are kept in memory and are not considered new starting points for the optimization process. The BLS Algorithm is implemented with OpenMP. The proposed algorithm has obtained 57 of the 59 (near)-best solutions.

Parallel metaheuristic algorithms can report better solutions than their sequential versions.[30,31] Sevinc and Dokeroglu[32] propose a parallel local search method (Par-LS) for finding MVWCP in large graphs. The authors show that the methods are adequate to deal with large graphs. A new best solution for frb100-40 problem instance in the BHOSLIB benchmark is reported. Kiziloz and Dokeroglu[33] propose a Message Passing Interfaces (MPI) parallel version of the Tabu Search algorithm for MVWCP. With MPI techniques and a unified neighborhood approach, add, swap and drop operators are improved. El Baz et al.[34] propose a parallel ant colony optimization algorithm for MVWCP. The parallel algorithm cooperates among many ant colonies. There exists a center of messages and colonies of ants. Encouraging results are observed in the study.

## 3 | PROPOSED ALGORITHM, TABU-NUMA

In this section, at first we give information about the OpenMP parallel programming paradigm and we present the Non-Uniform Memory Access (NUMA) architecture. Then, we describe our proposed Tabu-NUMA algorithm in detail.

### 3.1 | Open multi-processing

Open multi-processing (OpenMP) extends the programming language C++ to provide shared memory parallelism with compiler directives and run-time library routines.[35] OpenMP supports shared-memory multithreaded programming on many platforms.[36,37] OpenMP has a scalable model. There is a master thread in OpenMP that forks slave threads and shares a task with them. The threads run concurrently. In Figure 1, the master thread forks $n-1$ threads. They share the task of the Tabu-NUMA algorithm including the master thread (Note that Thread 0 is the master thread). Task and data parallelism can be provided using OpenMP which provides structured parallelism. The tasking model of OpenMP provides flexibility and huge potential for scalability and performance.

### 3.2 | Non-uniform memory access architecture

In non-uniform memory access (NUMA) systems, multiple processors share a memory system via a high-speed interconnect, but accessing memory can vary in speed depending on its location relative to the requesting processor. This leads to contention, where multiple processors try to access the same memory location simultaneously, causing performance degradation through increased latency and reduced throughput. Mitigating contention involves various techniques, including partitioning memory into local and remote regions and distributing memory access. Hardware support like cache coherence protocols and directory-based schemes can also help to reduce contention. Cache coherence protocols maintain coherence among multiple processors' caches to ensure a consistent view of memory, while directory-based schemes track memory block locations and state for

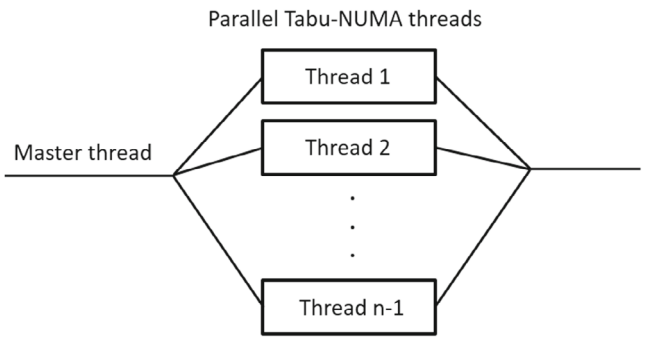


**FIGURE 1** The workflow of the threads in the Tabu-NUMA algorithm.

efficient sharing across processors. Software optimization techniques like memory affinity and load balancing algorithms can minimize contention by scheduling threads on processors with high memory affinity and distributing workloads evenly. Effective contention management is crucial to achieve optimal performance in multi-processor systems.

We use an AMD Opteron(tm) Processor 6376 board. It has a NUMA architecture. In its architecture, there are four sockets and in each socket there are eight cores. In a single core, two threads can run simultaneously. So, it is possible to run 64 threads simultaneously. There are eight NUMA nodes in this board. Each NUMA node has a memory bank and 8 CPUs. These CPUs share 6 MB Last Level Cache (LLC). The board has 64 GB RAM divided into 8 NUMA nodes. Ids of the threads start from 0 to 7 for NUMA node0, start from 8 to 15 for NUMA node1 and so on. The clock rate is 1400 MHz.

There is a study about the data and memory allocation performances of AMD Opteron Processor 6376 (Abu-Dhabi) in the literature.[38] This processor has similar architecture with our AMD Opteron(tm) Processor 6376. In this study, the architecture of NUMA is explained in detail. In Figure 2, the architecture of AMD Opteron Processor 6376 (Abu-Dhabi) is given. In the NUMA architecture, the global memory is divided evenly into the NUMA nodes. Each NUMA node has a processor and a memory bank. A program can access the memory bank of another node coherently with some latency. Running a program in a NUMA architecture is not a trivial task. Most of the applications and platforms, including OpenMP do not take the NUMA architecture into account.[38] They consider the physical memory as a single unit and try to allocate the places in memory without considering latency. In our experiments, we use the NUMA library (libnuma) along with OpenMP.[39] We run the threads evenly across the NUMA nodes. We use the formula (*tid mod* 8) and the *numa_run_on_node* instruction in libnuma in order to assigning the threads to the NUMA nodes evenly. To allocate the threads' private variables in their NUMA node's memory bank, we use the *numa_set_preferred(-1)* instruction.

### 3.3 | Tabu-NUMA algorithm

The Tabu-NUMA algorithm is an OpenMP version of the MN/TS algorithm introduced by Wu et al.[5] The Tabu-NUMA algorithm is written by using OpenMP libraries of C++. The developed algorithm is specific to the NUMA architectures.[38] The MN/TS algorithm uses the rand() function to generate random integer numbers. However, the rand() function is not thread-safe for OpenMP applications. The threads have a single hidden state updated by all threads. Because of this reason, we cannot obtain faster results with this approach. We should create a distinct hidden state for each thread to overcome the thread-unsafe problem. We create a distinct seed for each thread using the thread id parameter to do this. Hence, each thread has one distinct random number generator.[40] We use the Linear Congruential Generator (LCG) algorithm to create our random number

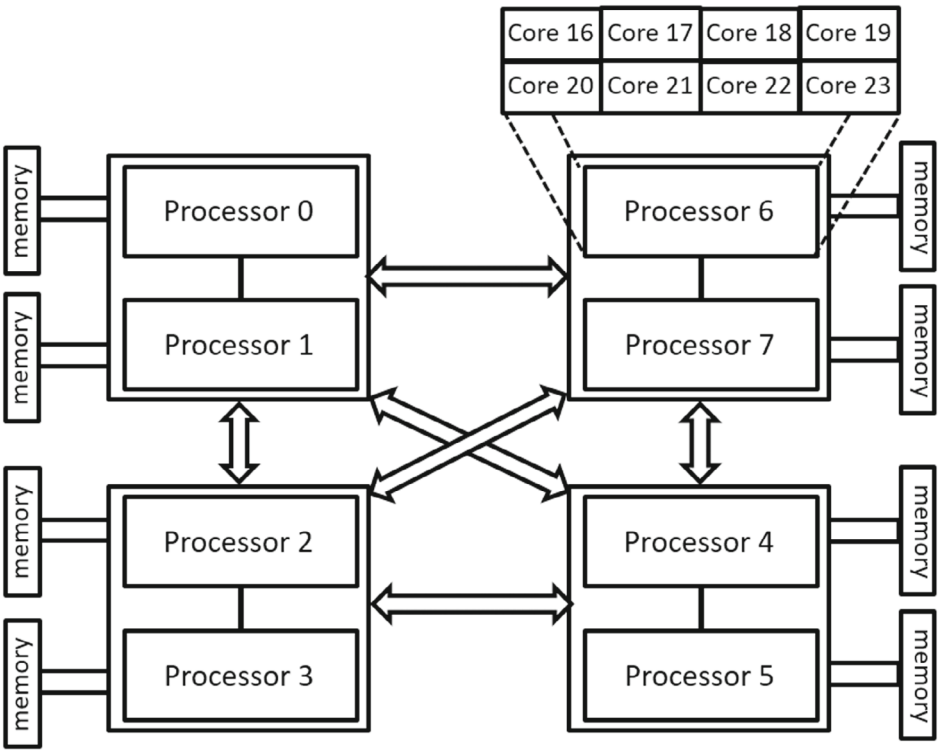


**FIGURE 2** The architecture of AMD Opteron Processor 6376 (Abu-Dhabi).[38]

generator for Tabu-NUMA. There is a comparison of the speed-up performances of the random number generators including our random number generator and the rand() function in Section 4.

The Tabu-NUMA algorithm starts constructing a clique, $C$, from a different vertex for each thread. This is ensured by generating distinct random numbers at each thread's process area. The more processors Tabu-NUMA has, the faster it explores MVWCP. To construct clique $C$, the Tabu-NUMA algorithm randomly selects vertex $i$ from $G$. Adding the neighboring vertex $v \notin C$ continues to make a bigger clique. This process goes on until there exists no vertex to be added to the current clique. Each thread employs this method with distinct random number generators, which means that unlike cliques are constituted at each thread's working areas.

There are mainly three operators (*add*, *swap* and *drop*) of the Tabu-NUMA algorithm. *Add* operator introduces a new vertex to the clique by increasing the total sum of the weights. This is a constructive movement on the current clique to the (near)-best solution. The complexity of the add operator is O($n$) with $n$ vertices. See Figure 3 for the execution of the add, swap, and drop operators. *Swap* operator exchanges a vertex in the current

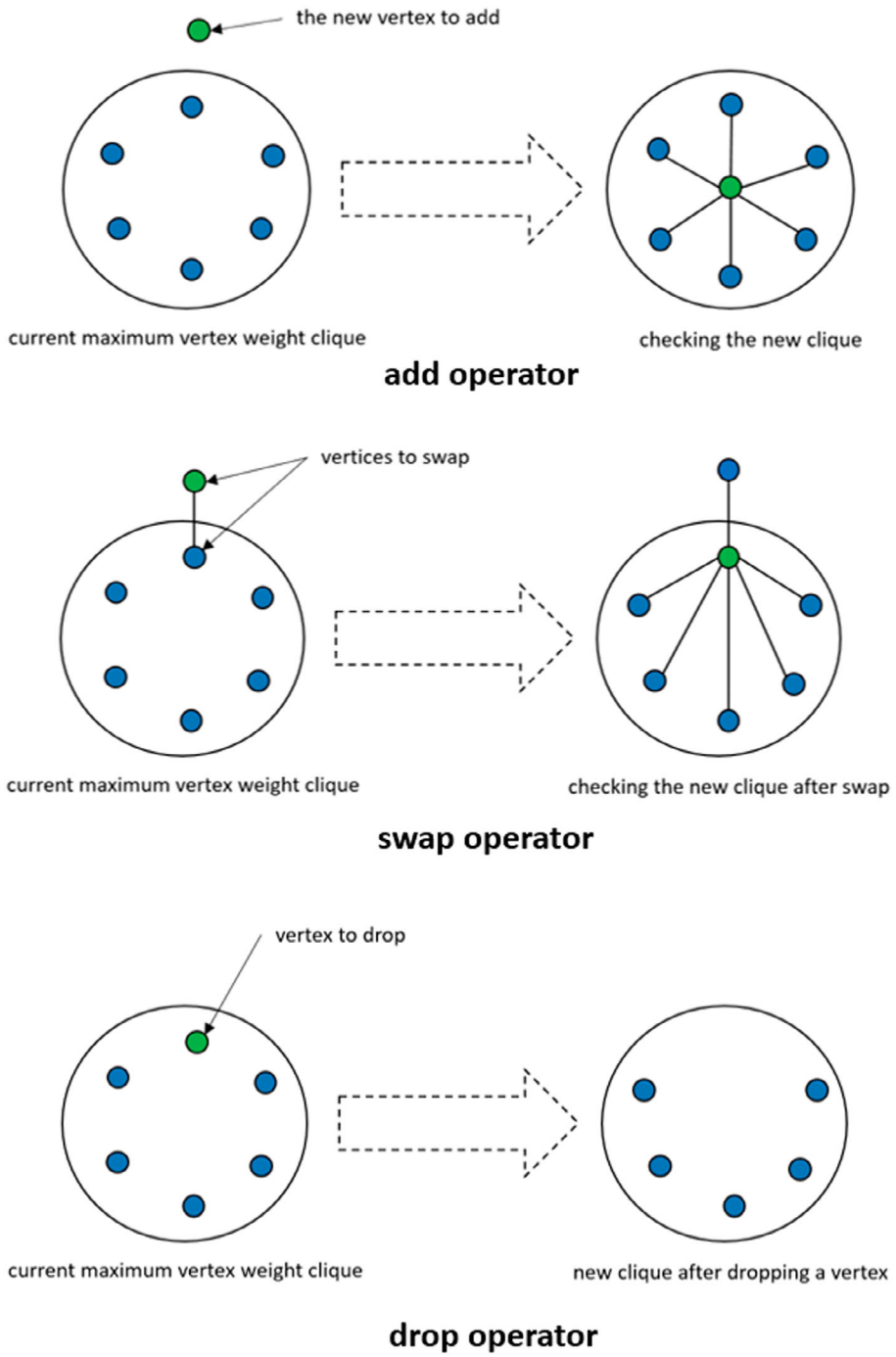


**FIGURE 3** An example executions of the add, swap, and drop operators.

clique with another vertex. This action may lead to a worse or better solution. It depends on the weight and the neighboring vertices of the nodes that are *swapped*. After the swap operation, the new maximum vertex weight clique must be checked whether it constitutes a new clique or not. The complexity of the swap operator is O($n$) with $n$ vertices. Occasionally, it may be possible to have better cliques instead of applying *add* operator. *Drop* action deletes a vertex from the current clique. It decreases the weight of the current clique by removing the vertices. This operator diversifies the solution space of the algorithm by escaping from local optima. Therefore, it may positively affect the exploration abilities of the algorithm. See Figure 3 for an example execution of the drop operator. There is no need to check the remaining clique. Therefore, there is no additional computation cost for this operator.

The *add* operator does not always positively influence the optimization of the problem. Every new vertex attempted to be added to the current clique will be accepted after verifying that it has connections with all the vertices in the current clique. In cases *add* operator cannot be useful, *swap* and *drop* operators can change the search space and direct us to some other spaces. Therefore, there is no superiority of a specific operator to others. The operators' performances are related to the landscape of the search space and the behavior of initial cliques. In the Tabu-NUMA algorithm, these operators are used in a union fashion to provide a more efficient way to optimize MVWCP.[5]

To prevent consecutive iterations of the search that cannot improve the clique weight, the parameter $L$ (search depth) is used to limit useless attempts. This means that the Tabu-NUMA algorithm will not continue exploring if there is no progress in the results. The Tabu-NUMA algorithm has a private tabu list at each thread's memory to control revisiting of previous solutions. When a vertex is deleted from the clique, it is not permitted to include the same vertex in the current clique for the defined number of iterations. This process is named tabu tenure.

Any vertex can be removed from the clique via *drop* and *swap* operators. Adding a new vertex is called tabu if it is in the tabu list. The aspiration criterion allows a move (even if it is in the tabu list) that generates a better solution. At each thread, it attempts three neighborhoods at each trial and accepts the best one. Three neighboring vertices improve the algorithm to make a better exploration. Tabu list members should not be revisited. This ensures diversification for better exploitation. These operations are performed simultaneously by all threads. The details of the MN/TS algorithm are presented in Algorithms 1 and 2.

**Algorithm 1.** *Max_Tabu* function[5]

```
1: G = (V, E, w) // weighted graph
2: L is search depth,
3: Iter_max // maximum number of iterations
4: C* is a Clique
5: Weight of a clique is W(C)
6: Iter = 0
7: C* = ∅
8: while (Iter<Iter_max) do
9:    C = Initialize()
10:   Initiate_tabu_list
11:   NI = 0 // the iterations that W(C) is not improved
12:   C_local_best = C
13:   while (NI<L) do
14:      N1, N2 and N3 are neighborhoods of C
15:      Select the best neighbor C'εN1 ∪ N2 ∪ N3
16:      C = C' // use a new solution
17:      NI + +
18:      Iter + +
19:      Update tabu_list
20:      if (W(C)>W(C_local_best))then
21:         NI = 0
22:         C_local_best = C
23:      end if
24:   end while
25:   if (W(C_local_best)>W(C*))then
26:      C* = C_local_best
27:   end if
28: end while
29: Return (Clique C*)
```

**Algorithm 2.** The MN/TS Algorithm[5]

1: *srand*(*time*()) // Set seed
2: *Initialize_Data*() // Input data and parameters
3: **for** ($i = 0; i<trial; i++$) **do**
4: $C^* = Max\_Tabu()$ // Algorithm 1
5: $W\_used[i] = C^*$ // Weight value
6: **end for**
7: *Output_Data*() // Calculate the overall results

Algorithm 1 is the *Max_Tabu* function of the MN/TS algorithm. Algorithm 2 is the MN/TS algorithm.[5] The *Max_Tabu* function is called 100 times (the value of *trial*) by Algorithm 2 and it calculates the weight of the corresponding clique at each iteration. The best one in the weights is the best weight value of the MN/TS algorithm. The other results of the MN/TS algorithm such as the cardinality, the average weight of the objective function, the success rate, the average number of iterations, and the average time are also calculated.

Tabu-NUMA distributes the workload of "for" loop across the threads by using OpenMP. The details of the Tabu-NUMA algorithm are given in Algorithm 3. At first, we determine how many threads are created with the *omp_set_num_threads*(*nthreads*) instruction of OpenMP. In this instruction, the variable *nthreads* represents the number of threads. Then, we open a private instruction space for each thread with **#pragma omp parallel private(tid)** directive of OpenMP. In this directive, *tid* is a private variable that keeps the id of the corresponding thread. In the following lines of the algorithm, every instruction is executed simultaneously by each thread unless a collective directive or an instruction is executed or the end of **#pragma omp parallel private(tid)** is reached. Each thread has a unique thread id and by using this id we can assign each thread to the NUMA nodes evenly. The $tid = omp\_get\_thread\_num()$ instruction returns the id of the corresponding thread and save it to its private *tid* variable. With the $numa\_run\_on\_node(tid \bmod 8)$ instruction, the threads are assigned to different NUMA nodes evenly. ($tid \bmod 8$) is the formula for this process where 8 is the number of the NUMA nodes in the architecture. When the number of threads is 4, thread 0 is assigned to NUMA node 0, thread 1 is assigned to NUMA node 1 and so on. When the number of threads is 8, all NUMA nodes run a single thread. When the number of threads is 16, all NUMA nodes run two threads and so on. In order to allocate spaces for the private data of the thread in the memory bank of its NUMA node, we can use the $numa\_set\_preferred(-1)$ instruction. The input data and parameters of the MN/TS algorithm are same and private for each thread and are allocated in the private memory spaces of each thread. So accessing them becomes fast for each thread. Furthermore, we should create a distinct seed for each thread using thread id. $seed = (unsigned)time(NULL)^{tid}$ instruction is a way to achieve distinct seeds. With **#pragma omp for schedule(static) nowait** OpenMP directive, the workload of "for" loop in Algorithm 2 is distributed across the threads evenly using a static scheduling policy. In this policy, the iterations are distributed consecutively across the threads. For example, remember the value of *trial* is 100 and assume the number of threads is equal to 32, the first four threads run four consecutive iterations of the loop. And the remaining ones run three consecutive iterations of the loop. To be more specific, thread 0 runs iterations 0, 1, 2, 3, thread 1 runs iterations 4, 5, 6, 7, thread 31 runs iterations 97, 98, 99. There is an implicit barrier at the end of this directive. In order to calculate the execution time of a thread correctly, we should disable this barrier. **nowait** clause helps us to disable this barrier. The execution time of the slowest thread is considered as the execution time of the parallel "for" loop. Once all the threads finish their work, the threads are destroyed except the master thread. The master thread receives the results of the slave threads from the arrays in shared memory such as *W_used* and calculate the overall results as in Algorithm 2

**Algorithm 3.** The Tabu-NUMA algorithm

1: *Set_number_of_threads*() // The number of threads is set
2: **#pragma omp parallel private(tid)** // The private instruction space is opened for each thread
3: $tid = Get\_thread\_id()$ // The id of the corresponding thread is set to *tid*
4: *Assign_thread_to_node*() // The thread is assigned to a NUMA node based on its thread id (*tid*)
5: *Set_memory_policy*() // Force the thread's private data to be allocated in its NUMA node's memory bank.
6: *Initialize_Data*() // Input data and parameters of the MN/TS algorithm are allocated for each thread privately
7: *Create_distinct_seed*() // Create a distinct seed for each thread using thread id (*tid*)
8: **#pragma omp for schedule(static) nowait** // The workload of "for" loop is distributed across the threads evenly
9: **for** ($i = 0; i<trial; i++$) **do**
10: $C^* = Max\_Tabu()$ // Algorithm 1
11: $W\_used[i] = C^*$ // Weight value
12: **end for**
13: **end #pragma omp parallel**// End of the private instruction space. All the threads except the master thread are destroyed
14: *Output_Data*() // Calculate the overall results

## 4 | EXPERIMENTAL SETUP AND RESULTS

The details of our experiments, the computation environment, the problem instances, the speed-up, and scalability analyses are given in this section. We compare the performance of the Tabu-NUMA algorithm with MN/TS, and recent MVWCP algorithms with 38 problems from DIMACS and BHOSLIB benchmark libraries. These problems have not been solved to the best value in every restart of the MN/TS algorithm. Originally, the instances are unweighted. Each vertex $i$ is given a weight $((i \ mod \ 200) + 1)$ as it is performed with other studies.[41] The results of the other algorithms are obtained from their original papers. The hardware platforms are different from each other. However, it still gives an idea about the execution times of the algorithms to the interested readers/researchers.

We use an AMD Opteron(tm) Processor 6376 board mentioned in Section 3.2. We use a Scientific Ubuntu 18.04.4 Long Term Service (LTS) 64-bit operating system. g++ version is 7.5. The total execution time of all the experiments we have carried out is about 360 h (15 days) for a single thread execution time.

Random number generators have some effects on the performance of our proposed algorithm. The random number generator rand() (used in the MN/TS algorithm) is not a thread-safe function for OpenMP. This function prevents us from achieving speed-up as the number of threads increases in our experiments. We test three different LCG random number generators and rand() in our Tabu-NUMA algorithm. We choose the parameters of glibc used by GCC, C++11's minstd_rand, and glibc rand48_r as the parameters of our LCG random number generator. We generate ten million random numbers, store them on the global memory, and implement the random number generators with OpenMP for the number of threads from 2 to 64. The speed up of the random number generators are reported in Table 1. It is seen that the speed up of the rand() function decreases substantially. This affects the execution time of Tabu-NUMA adversely. We achieve up to 30.38× speed up with three LCG random number generators. Although three generators achieve similar speed ups, we achieve high-quality results with the glibc rand48_r generator. Thus, we select it as our random number generator. Further, our random number generator achieves successful results in terms of quality and speed in an empirical test.[42]

### 4.1 | Analyzing the speed-up performance of the Tabu-NUMA algorithm

In this part of our experiments, we verify the speed-up performance of the Tabu-NUMA algorithm on ten problem instances with different numbers of vertices and edges. In the serial implementation of the MN/TS algorithm, there is a dependency caused by the random number generator between each run of *Max_Tabu*() call. In our Tabu-NUMA algorithm, there is no dependency between the threads because of using a distinct random number generator for each thread. The execution time of the experiment is the execution time of the slowest thread. We measure the speed-up ratio of an experiment by dividing the execution time of the sequential implementation by the execution time of the parallel implementation. We increase the number of threads as the power of 2 (starting from 2 to 64) and set the number of trials to 100. Table 2 presents the set of selected problem instances and their speed-ups with an increasing number of threads. We achieve up to 18.35× speed-up with 64 threads. The speed-up ratio becomes worse when the number of threads is 16 and above. At most one thread runs in a NUMA node up to 16 threads. When the number of threads is 16, more than one thread runs in a NUMA node. This situation has small negative impact on the execution time of the algorithm. But the main reason for this problem is hidden in the speed-up performance of the random number generators. Since the speed-up ratio becomes worse in our random number generator when the number of threads is 16 and above, this situation affects the speed-up performance of Tabu-NUMA directly. However, we can still evaluate the Tabu-NUMA algorithm as a scalable parallel algorithm. The average speed-up and average scalability of 10 selected problem instances on each number of threads are given in Figures 4 and 5, respectively. The scalability of the random number generators that we analyzed before is given in Figure 6. The effects of the random number generators on the speed-up of Tabu-NUMA can be seen in Figures 5 and 6.

**TABLE 1** The speed-up of the random number generators.

| #threads | rand() | LCG glibc | LCG minstd_rand | LCG rand48_r |
|---|---|---|---|---|
| Serial exec. time | 0.38 | 1.05 | 1.05 | 1.10 |
| 2 | 0.24 | 2.01 | 2.03 | 2.01 |
| 4 | 0.19 | 3.91 | 3.97 | 3.95 |
| 8 | 0.10 | 7.58 | 7.71 | 7.60 |
| 16 | 0.09 | 12.50 | 12.51 | 12.65 |
| 32 | 0.09 | 19.28 | 19.37 | 19.33 |
| 64 | 0.09 | 29.23 | 28.89 | 30.38 |

**TABLE 2** The speed-up performance of the proposed algorithm on ten different problems.

| #threads | brock800_1 | brock800_2 | brock800_3 | C2000.5 | keller5 | frb30-15-1 | frb30-15-5 | frb35-17-3 | frb35-17-2 | frb35-17-4 |
|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 1.64 | 2.00 | 1.58 | 1.61 | 1.65 | 1.69 | 1.47 | 1.67 | 1.51 | 1.57 |
| 4 | 3.26 | 3.46 | 2.74 | 3.06 | 2.78 | 2.66 | 2.91 | 2.91 | 2.63 | 3.05 |
| 8 | 5.13 | 6.06 | 5.19 | 4.91 | 5.09 | 4.68 | 4.53 | 4.97 | 4.26 | 5.67 |
| 16 | 7.89 | 10.17 | 6.67 | 7.58 | 7.88 | 8.21 | 6.99 | 8.02 | 6.71 | 9.27 |
| 32 | 10.33 | 13.74 | 9.33 | 11.11 | 12.21 | 11.05 | 8.87 | 11.79 | 9.73 | 12.71 |
| 64 | 12.05 | 15.87 | 11.64 | 13.57 | 14.30 | 13.88 | 11.73 | 14.47 | 13.79 | 18.35 |

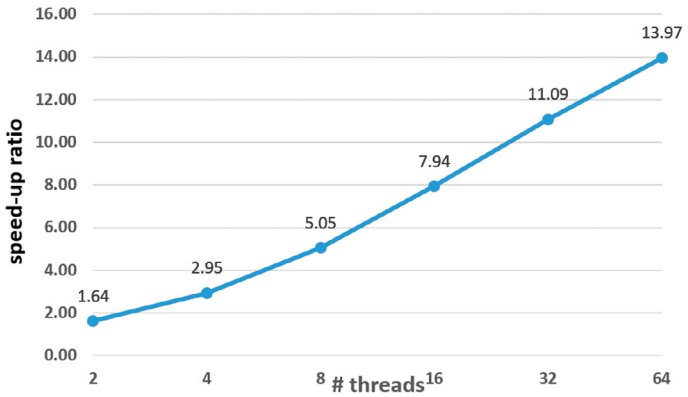


**FIGURE 4** The average speed-up performance of 10 selected problem instances.

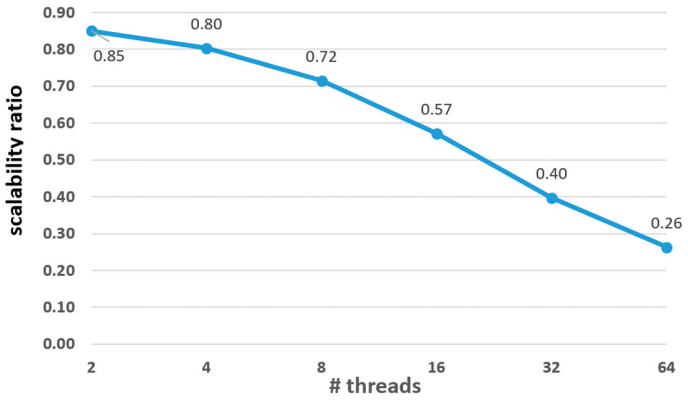


**FIGURE 5** The average scalability performance of 10 selected problem instances.

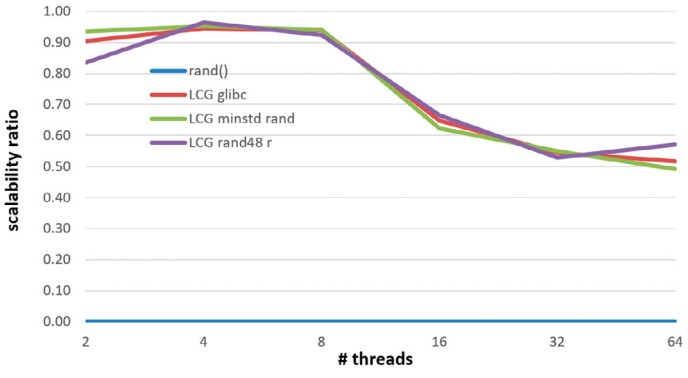


**FIGURE 6** The scalability performance of the random number generators.

## 4.2 | Observing the solution quality of Tabu-NUMA with an increasing number of threads

So far, we use multiple threads to reduce the execution time of the serial implementation without changing the number of trials. On the other hand, we can use multiple threads to increase the number of trials of the serial implementation without affecting its execution time. So, we are able to explore more candidate solutions at the same duration. This part of our experiments displays that we can obtain better solutions by sampling more candidates with high-performance computation. Figure 7 presents the results of our experiment on problem instance *frb45-21-3* with increasing number of threads starting from 2 to 16. The number of trials is determined as *#threads* times 100. So, each thread performs 100 *Max_Tabu*() calls. We select 100 solutions from the solutions of all threads by selecting the best solution from the first *#threads* solutions, the best solution from the second *#threads* solutions and so on until we select 100 better solutions. We obtain 26 best solutions with one thread configuration. When the number of threads reaches 8, we obtain 100 best solutions. This shows us that increasing the number of trials of the Tabu-NUMA algorithm and the number of threads enables us to obtain better solutions along with similar duration compared to the serial implementation. In addition, the objective function's average weight also increases from 4759.8 to 4765.0 (0.2% increase).

## 4.3 | Experiments on DIMACS-W and BHLOBS-W benchmark instances

In this part of our experiments, we select 38 problem instances from DIMACS-W and BHLOBS-W benchmark libraries. These problems have not been solved to the best value in every restart of the MN/TS algorithm. Therefore, we intend to compare our results, especially on these problem instances. We use the same approach given in Section 4.2. We set the number of threads to 64. Table 3 presents our results on the selected 38 problem instances. The sizes of the vertices and edges are given in this table. The averages of the sizes of the vertices and edges are 1272 and 643,575, respectively. The maximum number of vertices is 3361, and the maximum number of edges is more than 5.5 million. The columns represent the instance name, the number of successful trials in 100 trials or success rate of MN/TS (succ.), the average weight of the objective function of MN/TS, the number of successful trials in 100 trials or success rate of Tabu-NUMA (succ.), the average weight of the objective function of Tabu-NUMA, the percentage of the improvements in the number of successful trials of Tabu-NUMA and the percentage of the improvements in the average weight of the objective function of Tabu-NUMA, respectively. We select 100 better solutions of Tabu-NUMA as in Section 4.2. Then, we compute the overall results as in the MN/TS algorithm. We improve the success rate of 21 instances of 38. The highest improvement is obtained in instance *frb40-19-3* with an 80% value in the number of successful trials. The average improvement is 19.9% in 100 trials. Although having more best solutions in some problems is impossible, we improve the average solution quality in 12 instances of 17. Furthermore, we obtain 100 best solutions with the best-known objective function values in 12 instances.

## 4.4 | Comparison with state-of-the-art algorithms

The Tabu-NUMA algorithm is compared with state-of-the-art MVWCP algorithms. The selected algorithms are the versions of Tabu Search, metaheuristic or parallel algorithms in the literature. Breakout Local Search (BLS),[43] Parallel Tabu Search (PTC),[33] Restart Tabu Search-I (ReTS-I),[18] Binary Quadratic Programming-Probabilistic Tabu Search (BQP-PTS),[26] Phased Local Search (PLS),[41] and Parallel Local Search (Par-LS)[32] are the algorithms we compare Tabu-NUMA with. The results of these algorithms are provided in their manuscripts with the same problem instances. The results of

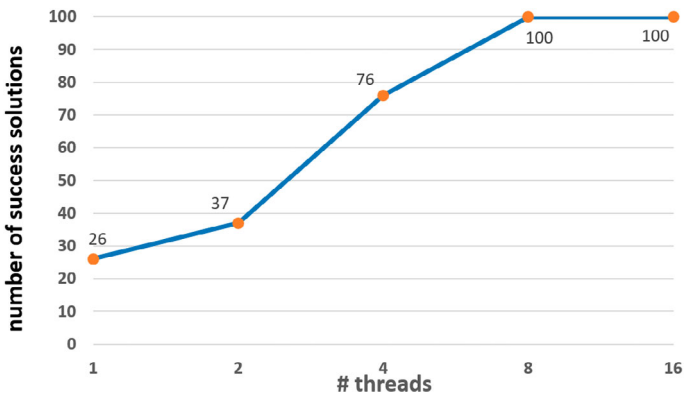


**FIGURE 7** The performance of the Tabu-NUMA algorithm with increasing number of threads on problem instance frb45-21-3.

**TABLE 3** The comparison of Tabu-NUMA with MN/TS on 38 problems. The problems are selected from the set of DIMACS-W and BHOSLIB-W benchmark libraries that have not been solved to the best value in every restart of the MN/TS algorithm. The percentage of successful trials (hits) are indicated with succ.

| | | | MN/TS | | Tabu-NUMA | | Improvement % | |
|---|---|---|---|---|---|---|---|---|
| **Instance** | **\|V\|** | **\|E\|** | **succ.** | **avg.** | **succ.** | **avg.** | **succ.** | **avg.** |
| C2000.9 | 2000 | 1,800,000 | 22 | 10, 971.9 | 65 | 10, 988.9 | **43** | **0.2** |
| keller6 | 3361 | 4,619,898 | 5 | 7939.5 | 0 | 80, 18.5 | 0 | **1.0** |
| MANN_a27 | 378 | 70,551 | 1 | 12, 273.3 | 17 | 12, 277.1 | **16** | 0.0 |
| MANN_a45 | 1035 | 533,115 | 1 | 34, 172.9 | 64 | 34, 187.8 | **63** | 0.0 |
| MANN_a81 | 3321 | 5,506,380 | 1 | 111, 108.3 | 53 | 111, 124.2 | **52** | 0.0 |
| p_hat1500-3 | 1500 | 847,244 | 96 | 10, 319.9 | 100 | 10, 321.0 | **4** | 0.0 |
| frb35-17-2 | 595 | 83200 | 96 | 3736.8 | 100 | 3738.0 | **4** | 0.0 |
| frb35-17-4 | 595 | 149,000 | 77 | 3678.3 | 100 | 3683.0 | **23** | **0.1** |
| frb40-19-1 | 760 | 249,000 | 83 | 4062.2 | 100 | 4063.0 | **17** | 0.0 |
| frb40-19-2 | 760 | 249,000 | 87 | 4111.2 | 100 | 4112.0 | **13** | 0.0 |
| frb40-19-3 | 760 | 249,000 | 19 | 4108.3 | 99 | 4114.9 | **80** | **0.2** |
| frb40-19-4 | 760 | 249,000 | 89 | 4135.6 | 100 | 4136.0 | **11** | 0.0 |
| frb40-19-5 | 760 | 249,000 | 90 | 4117.6 | 100 | 4118.0 | **10** | 0.0 |
| frb45-21-1 | 945 | 389,900 | 44 | 4748.7 | 100 | 4760.0 | **56** | **0.2** |
| frb45-21-2 | 945 | 387,400 | 47 | 4775.9 | 91 | 4783.6 | **44** | **0.2** |
| frb45-21-3 | 945 | 387,800 | 26 | 4756.9 | 100 | 4765.0 | **74** | **0.2** |
| frb45-21-4 | 945 | 387,500 | 43 | 4772.4 | 100 | 4799.0 | **57** | **0.6** |
| frb45-21-5 | 945 | 387,500 | 82 | 4777.4 | 100 | 4779.0 | **18** | **0.0** |
| frb50-23-1 | 1150 | 580,600 | 6 | 5484.7 | 0 | 5486.3 | 0 | 0.0 |
| frb50-23-2 | 1150 | 589,800 | 3 | 5434.1 | 1 | 5448.6 | 0 | **0.3** |
| frb50-23-3 | 1150 | 579,600 | 53 | 5480.3 | 86 | 5485.7 | **33** | **0.1** |
| frb50-23-4 | 1150 | 580,400 | 9 | 5451.7 | 2 | 5453.0 | 0 | 0.0 |
| frb50-23-5 | 1150 | 580,600 | 89 | 5495.7 | 100 | 5498.0 | **11** | 0.0 |
| frb53-24-1 | 1272 | 714,100 | 5 | 5637.9 | 1 | 5649.9 | 0 | **0.2** |
| frb53-24-2 | 1272 | 714,100 | 6 | 5676.6 | 0 | 5684.4 | 0 | **0.1** |
| frb53-24-3 | 1272 | 714,200 | 15 | 5610.8 | 88 | 5637.9 | **73** | **0.5** |
| frb53-24-4 | 1272 | 714,000 | 7 | 5645.6 | 0 | 5655.9 | 0 | **0.2** |
| frb53-24-5 | 1272 | 714,100 | 5 | 5628.8 | 0 | 5641.0 | 0 | **0.2** |
| frb56-25-1 | 1400 | 109,676 | 3 | 5836.9 | 0 | 4525.0 | 0 | 0.0 |
| frb56-25-2 | 1400 | 109,401 | 1 | 5807.7 | 0 | 4525.0 | 0 | 0.0 |
| frb56-25-3 | 1400 | 109,379 | 1 | 5799.4 | 0 | 4525.0 | 0 | 0.0 |
| frb56-25-4 | 1400 | 110,038 | 3 | 5839.2 | 0 | 4525.0 | 0 | 0.0 |
| frb56-25-5 | 1400 | 109,601 | 1 | 5768.4 | 0 | 4525.0 | 0 | 0.0 |
| frb59-26-1 | 1534 | 126,555 | 3 | 6547.5 | 0 | 6571.0 | 0 | **0.4** |
| frb59-26-2 | 1534 | 126,163 | 3 | 6567.1 | 59 | 6611.6 | **56** | **0.7** |
| frb59-26-3 | 1534 | 126,082 | 1 | 6514.2 | 0 | 6547.6 | 0 | **0.5** |
| frb59-26-4 | 1534 | 127,011 | 1 | 6498.4 | 0 | 6568.2 | 0 | **1.1** |
| frb59-26-5 | 1534 | 125,982 | 1 | 6522.6 | 0 | 6539.1 | 0 | **0.3** |
| average | 1265 | 643,575 | 29.6 | 9363.5 | 48.1 | 9207.2 | **19.9** | **0.2** |

MN/TS[5] have already been presented in Table 3. In Table 4, we can observe the comparisons of Tabu-NUMA with other algorithms. There are 33 problem instances. We compare the best and the average values of the weights obtained by the algorithms. This technique is applied by previous manuscripts while evaluating the performance of the algorithms. In the columns, the best and the average values in terms of weights found by the algorithms are reported in the format [best-value(average-value)]. In 22 problem instances, Tabu-NUMA is in the best three algorithms that obtain the best values. Furthermore, Tabu-NUMA is in the best three algorithms that solve 22 problem instances in the average values. The deviation of the results of Tabu-NUMA from the best values is not more than 0.5% for all the problem instances.

**TABLE 4** The comparison of Tabu-NUMA with recent algorithms in the literature. In the columns, the best and the average values in terms of weight found by the algorithms are reported in the format [best-value(average-value)]. The best algorithm is given in boldface.

| Instance | Tabu-NUMA | BLS | PTC | ReTS-I | BQP-PTS | PLS | Par-LS |
|---|---|---|---|---|---|---|---|
| C2000.9 | **10,999** (10,988.9) | **10,999** (10,989.9) | **10,999** (10,999) | **10,999** (10,996.4) | **10,999** | 10,028 | **10,999** |
| keller6 | 8038 (8018.5) | **8062** (8027.2) | **8062** (8062) | **8062** (8062) | **8062** | 7382 | **8062** |
| MANN_a27 | 12,281 (12277.1) | 12,281 (12,277) | **12,283** (12,283) | **12,283** (12282.8) | 12,277 | 12,264 | **12,283** |
| MANN_a45 | 34,190 (34,187.8) | 34,229 (34,211) | 34,205 (34,192) | **34,259** (34,253.6) | 34,194 | 34,129 | 34,254 |
| MANN_a81 | 111,128 (111,124.2) | 111,237 (111,188) | 111,146 (111,128) | 111,137 | 111,137 | 110,564 | **111,400** |
| p_hat1500-3 | **10,321** (10,321) | **10,321** (10,321) | **10,321** (10,321) | **10,321** (10,321) | **10,321** | 10,014 | **10,321** |
| frb35-17-2 | **3738** (3738) | **3738** (3738) | **3738** (3738) | **3738** (3738) | **3738** | - | **3738** |
| frb35-17-4 | **3683** (3683) | **3683** (3683) | **3683** (3683) | **3683** (3683) | **3683** | - | **3683** |
| frb40-19-1 | **4063** (4063) | **4063** (4062.8) | **4063** (4063) | **4063** (4063) | **4063** | - | **4063** |
| frb40-19-2 | **4112** (4112) | **4112** (4112) | **4112** (4112) | **4112** (4112) | **4112** | - | **4112** |
| frb40-19-3 | **4115** (4114.9) | **4115** (4111.7) | **4115** (4115) | **4115** (4114.9) | **4115** | - | **4115** |
| frb40-19-4 | **4136** (4136) | **4136** (4135.9) | **4136** (4136) | **4136** (4135.9) | **4136** | - | **4136** |
| frb40-19-5 | **4118** (4118) | **4118** (4117.52) | **4118** (4118) | **4118** (4118) | **4118** | - | **4118** |
| frb45-21-1 | **4760** (4760) | **4760** (4754.3) | **4760** (4760) | **4760** (4759.8) | **4760** | - | **4760** |
| frb45-21-2 | **4784** (4783.6) | **4784** (4784) | **4784** (4784) | **4784** (4784) | **4784** | - | **4784** |
| frb45-21-3 | **4765** (4765) | **4765** (4764.8) | **4765** (4765) | **4765** (4764.8) | **4765** | - | **4765** |
| frb45-21-4 | **4799** (4799) | **4799** (4797.24) | **4799** (4799) | **4799** (4799) | **4799** | - | **4799** |
| frb45-21-5 | 4779(4779) | 4779(4779) | **4799** (4799) | **4799** (4799) | 4779 | - | 4779 |
| frb50-23-1 | 5487(5486.3) | **5494** (5486.4) | **5494** (5491.4) | **5494** (5485.2) | **5494** | - | **5494** |
| frb50-23-2 | **5462** (5448.6) | **5462** (5440.2) | **5462** (5462) | **5462** (5451.9) | **5462** | - | **5462** |
| frb50-23-3 | **5486** (5485.7) | **5486** (5485.9) | **5486** (5486) | **5486** (5485.2) | **5486** | - | **5486** |
| frb50-23-4 | **5454** (5453) | **5454** (5453.1) | **5454** (5454) | 5453(5452.5) | **5454** | - | **5454** |
| frb50-23-5 | **5498** (5498) | **5498** (5498) | **5498** (5498) | **5498** (5498) | **5498** | - | **5498** |
| frb53-24-1 | **5670** (5649.9) | **5670** (5652.2) | **5670** (5670) | **5670** (5661.4) | **5670** | - | **5670** |
| frb53-24-2 | 5688 (5684.4) | **5707** (5685.2) | **5707** (5707) | **5707** (5707) | **5707** | - | **5707** |
| frb53-24-3 | 5640 (5637.9) | 5640 (5629.4) | 5640 (5640) | **5655** (5636.5) | 5640 | - | 5640 |
| frb53-24-4 | 5665 (5655.9) | **5714** (5676.2) | **5714** (5704.4) | **5714** (5696.9) | **5714** | - | **5714** |
| frb53-24-5 | 5650 (5641) | **5659** (5642.5) | **5659** (5658.2) | **5659** (5651.4) | **5659** | - | **5659** |
| frb59-26-1 | 6575 (6571) | **6591** (6571.6) | **6591** (6583.3) | **6591** (6578.7) | **6591** | - | **6591** |
| frb59-26-2 | **6645** (6611.6) | **6645** (6602.3) | **6645** (6617.4) | **6645** (6689.1) | 6545 | - | **6645** |
| frb59-26-3 | 6606 (6547.6) | **6608** (6542.7) | **6608** (6582.7) | **6608** (6579.1) | **6608** | - | **6608** |
| frb59-26-4 | 6575 (6568.2) | **6592** (6526.5) | **6592** (6542.1) | **6592** (6585.1) | **6592** | - | **6592** |
| frb59-26-5 | 6541 (6539.1) | **6584** (6546.9) | **6584** (6562.4) | **6584** (6558.5) | **6584** | - | **6584** |

Parallel ant colony optimization-based metaheuristic (PACOM) is a recent algorithm.[34] PACOM is only better with *keller6* and *MANN_a45* problem instances. With other problems, Tabu-NUMA is equal to or better than the PACOM algorithm. Weighted Large Maximum Clique (WLMC) is an exact algorithm whose results are reported with 3.600-second optimization process limits.[21] For the problems *C2000.9*, *p-hat1500-3*, and *keller6*, Tabu-NUMA performs better than WLMC. WLMC is better than Tabu-NUMA in the problem instances of MANN. FastWClq is a heuristic solver.[24] Tabu-NUMA outperforms this heuristic in all problem instances reported by the researchers. LSCC+BMS is a recent algorithm that uses the heuristic, Best from Multiple Selection.[17] For the problem instance *C2000.9*, Tabu-NUMA obtains the same solution with LSCC+BMS. For the problem instances, *MANN_a45*, *MANN_a81*, and *phat1500-3*, Tabu-NUMA has better performance than LSCC+BMS.

Table 5 presents the average time of the successful trials of state-of-the-art algorithms. We use our approach in Section 4.1. The results of Tabu-NUMA are obtained with 64 threads and 100 number of trials.

Compared with the other algorithms, the average times of the successful trials of the Tabu-NUMA algorithm are reasonable. Out of 9 of 16 instances, Tabu-NUMA achieves the best three results. With *MANN_a81* and *c-fat200-1* problem instances, Tabu-NUMA has the longest average time of the successful trials.

## 4.5 | Performance analysis on large graphs

We also perform an experiment by using our approach in Section 4.1 on the large graph instances by setting the thread configuration to 64 and setting the number of trials to 100. Experiments are carried out with the large instances given in this study.[32] We choose 13 large graph instances whose size of vertices is at most 11,500 by considering the capacity of our memory (RAM). The results of the experiment in terms of the best and the average values of the weights and their comparisons with the results of the MN/TS algorithm given in this study[32] are presented in Table 6. The sizes of the vertices and edges of the instances and the speed up results are also presented in the table.

We achieve the same results with the MN/TS algorithm in most of the instances and achieve better results in 2 instances in terms of the best and the average values of the weights. Moreover, we obtain up to 21.38× speed up. It is seen that we achieve the same results with the MN/TS algorithm along with remarkable speed up on the large graph instances. The datasets can be downloaded from Network Data Repository.[44]

**TABLE 5** Average time of the successful trials of state-of-the-art algorithms for selected problem instances (in seconds). Tabu-NUMA is executed with 64 threads and 100 iterations. The best algorithm is given in boldface.

| Instance | Tabu-NUMA | BQP-PTS | PLS | MN/TS | BLS | ReTS-I | ILS-VND | PTC | Par-LS |
|---|---|---|---|---|---|---|---|---|---|
| brock200_1 | 0.00 | 0.02 | 0.19 | 0.01 | 0.01 | **0** | 0.01 | 0.7 | 0.1 |
| brock800_4 | 73.16 | 105.53 | 3.77 | 49.7 | 339.07 | 835.03 | 25.4 | 0.4 | **0.1** |
| keller4 | 0.02 | 0.05 | 0.02 | 0.03 | 0.04 | **0** | 0.01 | 0.1 | 0.1 |
| keller6 | 810.75 | 3,418.36 | - | 606.15 | 1,980.16 | 532.74 | **53.3** | 610.4 | 58.4 |
| MANN_a9 | 0.00 | 0.01 | 0.01 | 0.01 | - | **0** | 0.01 | 0.1 | 0.2 |
| MANN_a81 | 6,457.65 | 6,167.28 | - | 832.24 | 2,942.54 | 990.02 | **74.99** | 954.8 | 3401.8 |
| hamming6-2 | 0.00 | 0.01 | 0.01 | 0.001 | 0.01 | **0** | 0.01 | **0** | 0.1 |
| hamming10-4 | 2.43 | 32.49 | 1,433.07 | 2.21 | 26.86 | 26.25 | 6.83 | **1.2** | 8.5 |
| gen200_p0.9_44 | 0.00 | 0.02 | 4.44 | 0.01 | 0.01 | **0** | 0.01 | **0** | 0.1 |
| gen400_p0.9_75 | 0.59 | 0.67 | **0.01** | 0.88 | 0.43 | 0.03 | **0.01** | 0.3 | 0.1 |
| c-fat200-1 | 0.34 | 0.01 | 0.01 | 0.14 | - | **0** | 0.01 | 0.2 | 0.1 |
| c-fat500-10 | 0.05 | 1.29 | 0.01 | 0.06 | - | 0.11 | 0.01 | **0** | 0.1 |
| johnson8-2-4 | 0.01 | 0.01 | 0.01 | 0.01 | 0.01 | **0** | 0.01 | **0** | 0.1 |
| johnson32-2-4 | 0.82 | 26.71 | 44.68 | 0.53 | 0.48 | 0.04 | **0.01** | 0.3 | 0.1 |
| p_hat300-1 | 0.01 | 0.03 | 0.01 | 0.02 | 0.01 | **0** | 0.01 | **0** | 0.1 |
| p_hat1500-3 | 100.72 | 34.14 | - | 188.38 | 1.78 | 2.06 | **0.06** | 246.3 | 0.1 |

**TABLE 6** The comparison of Tabu-NUMA with MN/TS in terms of the best and the average values of the weights found by the algorithms. The speed up performance of Tabu-NUMA with 64 thread configuration is also presented.

| Instance | \|V\| | \|E\| | MN/TS (Best) | MN/TS (Avg.) | Tabu-NUMA (Best) | Tabu-NUMA (Avg.) | Tabu-NUMA (Speed Up) |
|---|---|---|---|---|---|---|---|
| bio-dmela | 7393 | 25,569 | 805 | 805 | 805 | 805 | 1.45 |
| c4000-5 | 4000 | 4,000,268 | 2792 | 2788.4 | 2792 | 2792 | 13.07 |
| ca-Erdos992 | 6100 | 7515 | 958 | 958 | 958 | 958 | 16.31 |
| ca-HepPh | 11,204 | 117,619 | 24,533 | 24,533 | 24,533 | 24,533 | 9.98 |
| ia-reality | 6809 | 7680 | 374 | 368.875 | 374 | 374 | 21.35 |
| inf-power | 4941 | 6594 | 888 | 888 | 888 | 888 | 12.94 |
| socfb-CMU | 6621 | 249,959 | 4141 | 4141 | 4141 | 4141 | 17.02 |
| socfb-Duke14 | 9885 | 506,437 | 3694 | 3694 | 3694 | 3694 | 11.96 |
| socfb-MIT | 6402 | 251,230 | 3658 | 3658 | 3658 | 3658 | 16.12 |
| tech-WHOIS | 7476 | 56,943 | 6154 | 6154 | 6154 | 6154 | 15.25 |
| web-edu | 3031 | 6474 | 2077 | 2077 | 2077 | 2077 | 16.78 |
| web-indochina-2004 | 11,358 | 47,606 | 6997 | 6997 | 6997 | 6997 | 21.38 |
| web-spam | 4767 | 37,375 | 2503 | 2503 | 2503 | 2503 | 10.20 |

## 5 | CONCLUSION AND FUTURE WORK

This study introduces the Tabu-NUMA algorithm, a novel approach to improve the optimization skills of MVWCP on the NUMA architectures. We achieve scalable algorithms on the NUMA architectures using OpenMP. To evaluate the performance of the Tabu-NUMA algorithm, we compare it with the MN/TS algorithm on 38 graph instances from DIMACS-W and BHOSLIB-W benchmarks. The ratio of the best solutions (reported in the literature) found by the algorithms in 100 trials is used as the criterion. The Tabu-NUMA algorithm outperforms the MN/TS algorithm in terms of execution times and improves the performance of the MN/TS algorithm using multi-core architectures. Specifically, the average success rate of the MN/TS algorithm increases by 19.9% with Tabu-NUMA on the selected set of problem instances. In 12 problem instances, Tabu-NUMA achieves the best results in the literature. Additionally, we observe that the success rate of the Tabu-NUMA algorithm improves as the number of cores increases and we suggest adding more computation power to yield better results for the tested instances. On the other hand, we distribute the workload of the MN/TS algorithm to the cores (up to 64 cores) evenly with Tabu-NUMA on our NUMA architecture. We obtain similar results with the MN/TS algorithm along with better execution times. We ensure up to 18.35× speed-up for ten basic problem instances from DIMACS-W and BHOSLIB-W benchmarks and ensure up to 21.38× speed-up for the large graph instances.

In the future, new generation metaheuristics such as artificial bee colonies and social spider web, as well as their hybrid versions with Tabu Search, can be developed and parallelized for the NUMA architectures. This can open up new avenues for the research in this field. Other combinatorial optimization problems, such as graph coloring and quadratic assignment, can also be applied to this class of parallel metaheuristic algorithms on the NUMA architectures.

### ACKNOWLEDGMENTS

We would like to express our sincere thanks to Prof. Dr. Murat Manguoğlu for providing us their AMD Opteron(tm) Processor 6376 board for our experiments.

### DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon reasonable request.

### ORCID

*Özcan Dülger* https://orcid.org/0000-0001-7525-1064

### REFERENCES

1. Pardalos PM, Xue J. The maximum clique problem. *J Glob Optim*. 1994;4(3):301-328. doi:10.1007/BF01098364
2. Wu Q, Hao JK. A clique-based exact method for optimal winner determination in combinatorial auctions. *Inf Sci*. 2016;33:103-121. doi:10.1016/j.ins.2015.11.029

3. Zhou Y, Hao JK, Duval B. Opposition-based memetic search for the maximum diversity problem. *IEEE Trans Evol Comput*. 2017;21(5):731-745. doi:10.1109/TEVC.2017.2674800
4. Wu Q, Hao JK. A review on algorithms for maximum clique problems. *Eur J Oper Res*. 2015;242(3):693-709. doi:10.1016/j.ejor.2014.09.064
5. Wu Q, Hao JK, Glover F. Multi-neighborhood tabu search for the maximum weight clique problem. *Ann Oper Res*. 2012;196(1):611-634. doi:10.1007/s10479-012-1124-3
6. Zheng C, Zhu Q, Sankoff D. Removing noise and ambiguities from comparative maps in rearrangement analysis. *IEEE/ACM Trans Comput Biol Bioinform*. 2007;4(4):515-522. doi:10.1109/TCBB.2007.1075
7. Zhian H, Sabaei M, Javan NT, Tavallaie O. Increasing coding opportunities using maximum-weight clique. *Computer Science and Electronic Engineering Conference (CEEC)*. Vol 2013. IEEE; 2013:168-173.
8. Mascia F, Cilia E, Brunato M, Passerini A. Predicting structural and functional sites in proteins by searching for maximum-weight cliques. *Proceedings of the AAAI Conference on Artificial Intelligence*. AAAI Press; 2010:1274-1279.
9. Ma T, Latecki LJ. Maximum weight cliques with mutex constraints for video object segmentation. *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference*. IEEE; 2012:670-677.
10. Li CM, Liu Y, Jiang H, Manyà F, Li Y. A new upper bound for the maximum weight clique problem. *Eur J Oper Res*. 2018;270(1):66-77. doi:10.1016/j.ejor.2018.03.020
11. Tepper M, Sapiro G. Ants crawling to discover the community structure in networks. *Iberoamerican Congress on Pattern Recognition*. Springer; 2013:552-559.
12. Dokeroglu T, Sevinc E, Kucukyilmaz T, Cosar A. A survey on new generation metaheuristic algorithms. *Comput Ind Eng*. 2019;137:106040. doi:10.1016/j.cie.2019.106040
13. Jemai M, Dimassi S, Ouni B, Mtibaa A. A metaheuristic based on the tabu search for hardware-software partitioning. *Turk J Electr Eng Comput Sci*. 2017;25(2):901-912. doi:10.3906/elk-1501-64
14. Glover F, Laguna M. Tabu search. *Handbook of combinatorial optimization*. Springer; 1998:2093-2229. doi:10.1007/978-1-4613-0303-9_33
15. Hosseinian S, Fontes DB, Butenko S. A Lagrangian bound on the clique number and an exact algorithm for the maximum edge weight clique problem. *INFORMS J Comput*. 2020;32(3):747-762. doi:10.1287/ijoc.2019.0898
16. Chu Y, Liu B, Cai S, Luo C, You H. An efficient local search algorithm for solving maximum edge weight clique problem in large graphs. *J Comb Optim*. 2020;39(4):933-954. doi:10.1007/s10878-020-00529-9
17. Wang Y, Cai S, Yin M. *Two Efficient Local Search Algorithms for Maximum Weight Clique Problem*. AAAI; 2016:805-811.
18. Zhou Y, Hao JK, Goëffon A. A generalized operator for the maximum vertex weight clique problem. *Eur J Oper Res*. 2017;257(1):41-54. doi:10.1016/j.ejor.2016.07.056
19. Nogueira B, Pinheiro RG. A CPU-GPU local search heuristic for the maximum weight clique problem on massive graphs. *Comput Oper Res*. 2018;90:232-248. doi:10.1016/j.cor.2017.09.023
20. Jiang H, Li CM, Liu Y, Manya F. A two-stage maxsat reasoning approach for the maximum weight clique problem. *Thirty-Second AAAI Conference on Artificial Intelligence*. Vol 32. AAAI Press; 2018.
21. Jiang H, Li CM, Manya F. *An Exact Algorithm for the Maximum Weight Clique Problem in Large Graphs*. AAAI; 2017:830-838.
22. McCreesh C, Prosser P, Simpson K, Trimble J. On maximum weight clique algorithms, and how they are evaluated. *International Conference on Principles and Practice of Constraint Programming*. Springer; 2017:206-225. doi:10.1007/978-3-319-66158-2_14
23. Li R, Wu X, Liu H, Wu J, Yin M. An efficient local search for the maximum edge weighted clique problem. *IEEE Access*. 2018;6:10743-10753. doi:10.1109/ACCESS.2018.2799953
24. Cai S, Lin J. *Fast Solving Maximum Weight Clique Problem in Massive Graphs*. IJCAI; 2016:568-574.
25. Nogueira B, Pinheiro RG, Subramanian A. A hybrid iterated local search heuristic for the maximum weight independent set problem. *Optim Lett*. 2018;12(3):567-583. doi:10.1007/s11590-017-1128-7
26. Wang Y, Hao JK, Glover F, Lü Z, Wu Q. Solving the maximum vertex weight clique problem via binary quadratic programming. *J Comb Optim*. 2016;32(2):531-549. doi:10.1007/s10878-016-9990-2
27. Wang Y, Cai S, Chen J, Yin M. SCCWalk: An efficient local search algorithm and its improvements for maximum weight clique problem. *Artif Intell*. 2020;280:103230. doi:10.1016/j.artint.2019.103230
28. Chu Y, Luo C, Hoos HH, Lin Q, You H. Improving the performance of stochastic local search for maximum vertex weight clique problem using programming by optimization. arXiv preprint, arXiv.2002.11909, 2020.
29. Aksan Y, Dokeroglu T, Cosar A. A stagnation-aware cooperative parallel breakout local search algorithm for the quadratic assignment problem. *Comput Ind Eng*. 2017;103:105-115. doi:10.1016/j.cie.2016.11.023
30. Dokeroglu T, Pehlivan S, Avenoglu B. Robust parallel hybrid artificial bee colony algorithms for the multi-dimensional numerical optimization. *J Supercomput*. 2020;76(9):7026-7046. doi:10.1007/s11227-019-03127-7
31. Arslan H, Manguoglu M. A hybrid single-source shortest path algorithm. *Turk J Electr Eng Comput Sci*. 2019;27(4):2636-2647.
32. Sevinc E, Dokeroglu T. A novel parallel local search algorithm for the maximum vertex weight clique problem in large graphs. *Soft Comput*. 2020;24(5):3551-3567. doi:10.1007/s00500-019-04122-z
33. Kiziloz HE, Dokeroglu T. A robust and cooperative parallel tabu search algorithm for the maximum vertex weight clique problem. *Comput Ind Eng*. 2018;118:54-66. doi:10.1016/j.cie.2018.02.018
34. El Baz D, Hifi M, Wu L, Shi X. A parallel ant colony optimization for the maximum-weight clique problem. *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE; 2016:796-800. doi:10.1109/IPDPSW.2016.111
35. Dagum L, Menon R. Openmp: an industry standard API for shared-memory programming. *IEEE Comput Sci Eng*. 1998;5(1):46-55. doi:10.1109/99.660313
36. Chandra R, Dagum L, Kohr D, et al. *Parallel programming in OpenMP*. Morgan Kaufmann; 2001.
37. Ayguadé E, Copty N, Duran A, et al. The design of OpenMP tasks. *IEEE Trans Parallel Distributed Syst*. 2008;20(3):404-418. doi:10.1109/TPDS.2008.105
38. Lenis J, Senar MA. A performance comparison of data and memory allocation strategies for sequence aligners on NUMA architectures. *Clust Comput*. 2017;20(3):1909-1924. doi:10.1007/s10586-017-1015-0
39. Kleen A. *A Numa API for Linux*. Novel Inc.; 2005.
40. Trobec R, Slivnik B, Bulić P, Robič B. *Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms*. Springer Nature; 2018.

41. Pullan W. Approximating the maximum vertex/edge weighted clique using local search. *J Heuristics*. 2008;14(2):117-134. doi:10.1007/s10732-007-9026-2
42. L'Ecuyer P, Simard R. Testu01: AC library for empirical testing of random number generators. *ACM Trans Math Software*. 2007;33(4):1-40. doi:10.1145/1268776.1268777
43. Benlic U, Hao JK. Breakout local search for maximum clique problems. *Comput Oper Res*. 2013;40(1):192-206. doi:10.1016/j.cor.2012.06.002
44. Rossi Ryan A, Ahmed NK. *The Network Data Repository with Interactive Graph Analytics and Visualization*. AAAI; 2015 https://networkrepository.com

**How to cite this article:** Dülger Ö, Dökeroğlu T. A new parallel tabu search algorithm for the optimization of the maximum vertex weight clique problem. *Concurrency Computat Pract Exper*. 2024;36(2):e7891. doi: 10.1002/cpe.7891