Parallel approach for brute force perfect matching

Alan Kunz Cechinel
PPGEAS - UFSC
cechinel.a.k@gmail.com

***Abstract*—The Chinese Postman Problem (CPP) is a classic coverage problem that has several applications. This paper presents two parallel implementations (V1 and V2) that use brute force to solve the CPP [1]. These versions reached up to 4 times speedup compared to the sequential one. However, experiments in a computer with NUMA architecture had results significantly worse than the computer with UMA. Further studies are needed to explain the cause of these results.**

***Index Terms*—Eulerization, Parallelism, Multicore**

I. Introduction

The Chinese Postman Problem (CPP) also knew as the weighted-eulerization problem, received this name regarding its creator, the Chinese mathematician Mei-Ko Kwan. In [1], Mei-Ko Kwan raised the question of how a mail carrier can visit all streets in a postal zone, coming back to his point of origin, having travelled the minimum distance possible in his route. The first solution to this problem come in [2].

Solving the CPP is useful for applications where an exhaustive route is needed, such as mail delivery, garbage collection, 3D printing, and coverage path-planing [3], [4].

Formally, the CPP is to find the shortest path in a graph, in which each edge is visited at least one time and ends at the starting vertex. This path is known as the eulerian circuit. For this path to be possible, the graph must be Eulerian. In this case, it has all vertices with even degrees [4]. In practical applications, graphs are not Eulerian. Therefore, they need to go through the eulerization process. This process adds edges to the graph, making it Eulerian.

The eulerization process can be done using brute force [4]. However, even in small problems, it can be costly. In this context, this work proposes a parallelization to the brute force eulerization process. It has as objective speedup this brute force technique and, in the future, compare it with more sophisticated methods.

II. Related Work

Edmonds and Johnson [2] proposed an algorithm to solve the CPP. The method first finds the minimum weight matching of the graph, and then construct the Euler graph. This sequential method gives the perfect matching algorithm about the minimum weight of CPP with n vertices in $O(n^3)$.

In [5], a bio-inspired algorithm based on molecular computation is proposed to solve the CPP. This approach has the advantages of high computational efficiency, large storage capacity, and strong parallel computing ability. Due to the good parallelism of DNA computing, this method solves the CPP with n vertices in $O(n^2)$.

The brute force method to solve the CPP has $O(2^{n/2})$ time complexity in its sequential version. Therefore, it is interesting the parallelization of the method to reduce the global time complexity. After that, it could be compared with [2] and [5] in a future work using different problem sizes.

III. Problem

The weighted-eulerization problem, also known as the Chinese Postman Problem, can be solved using Algorithm 1.

Through Steps 1 to 4, it is created a complete graph using the odd vertices. Dijkstra's algorithm can be used to find the shortest path and its total weight for each pair of odd vertices in the complete graph.

The perfect matching, in Step 5, can be found listing all combinations of odd vertex pairs. Then, the one with the smallest total weight is selected.

In Step 6, the edges corresponding to the shortest path between pairs in the perfect matching are duplicated. Thus, Fleury's or Hierholzer's algorithm can be used to define the Eulerian circuit in Step 7.

Algorithm 1: Postman Algorithm – Adapted from [4].

```
Data: Weighted graph $G = (V; E; w)$.
Steps:
  1) Find the set $S$ of odd vertices in $G$.
  2) Form the complete graph $K_n$ where $n = |S|$.
  3) For each distinct pair $x, y \in S$, find the shortest path
     $P_{xy}$ and its total weight $w(P_{xy})$.
  4) Define the weight of the edge $xy$ in $K_n$ to be
     $w(xy) = w(P_{xy}))$.
  5) Find a perfect matching $M$ of $K_n$ of least total weight.
  6) For each edge $e = xy \in M$, duplicate the edges of $P_{xy}$
     corresponding to the shortest path from $x$ to $y$,
     creating $G^*$.
  7) Find an eulerian circuit of $G^*$.
Result: Eulerian circuit; Total distance
```

Preliminary experiments showed that, when using brute force, the Step 5 consumes over 99% of the execution time for problems with sixteen or more odd vertices. Therefore, this work focused on speedup the runtime of Step 5.

Listings 1 and 2 combined, implement Step 5 using brute force in the C++ programming language. The method LPC, in Listing 1, lists combinations of odd vertex pairs. For instance,

[1]Source code on *https://github.com/alankc/ChinesePostmanProblem*

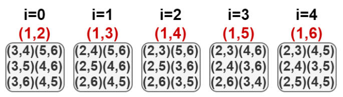

Fig. 1: The subset of combinations by iteration. In red elements of the first pair. In grey, subset generated using LPC.

given vertices 1, 2, 3, and 4. The pair combinations would be $\{(1,2)(3,4)\}$, $\{(1,3)(2,4)\}$, and $\{(1,4)(2,3)\}$. Despite the simple description, the number of combinations for n odd vertices is given by $(n-1)!!$. Therefore, the problem reaches millions of combinations with only sixteen odd vertices.

The method LPCB, in Listing 2, iteratively searches the perfect matching in subsets of the universe of combinations. In each iteration, the algorithm gets the subset of combinations (lines 20 to 24). After that, it defines the local perfect matching (lines 27 to 40). Then, it compares local and global minimum matching and maintains the best as the global minimum matching (lines 43 to 50). For instance, considering the odd vertices as $\{1,2,3,4,5,6\}$, LPCB will perform five iterations searching in the subsets showed in Figure 1. Taking iteration $i = 2$ as an example, the subset will be composed of the combinations $\{(1,4)(2,3)(5,6)\}$, $\{(1,4)(2,5)(3,6)\}$, and $\{(1,4)(2,6)(3,5)\}$. The number of combinations in each subset for n odd vertices is given by $(n-3)!!$.

```
 1  vector<vector<pair<uint16_t, uint16_t>>>
 2  LPC(vector<uint16_t> &odd)
 3  {
 4      vector<vector<pair<uint16_t, uint16_t>>> final;
 5      if (odd.size() == 2)
 6      {
 7          vector<pair<uint16_t, uint16_t>> buffer;
 8          buffer.push_back(make_pair(odd[0], odd[1]));
 9          final.push_back(buffer);
10      }
11      else
12      {
13          uint16_t first = *odd.begin();
14          odd.erase(odd.begin());
15
16          for (uint16_t i = 0; i < odd.size(); i++)
17          {
18              auto odd_j = odd;
19              uint16_t second = odd[i];
20              odd_j.erase(odd_j.begin() + i);
21
22              auto final_tmp = LPC(odd_j);
23              for (auto &el : final_tmp)
24              {
25                  vector<pair<uint16_t, uint16_t>> buffer;
26                  buffer.push_back(make_pair(first, second));
27                  final.push_back(buffer);
28                  copy(el.begin(), el.end(),
29                      back_inserter(final[final.size()-1]));
30              }
31          }
32      }
33      return final;
34  }
```

Listing 1: List Pairs Combination

```
 1  vector<pair<uint16_t, uint16_t>>
 2  LPCB(vector<uint16_t> &odd,
 3      vector<vector<uint16_t>> &distances)
 4  {
 5      vector<pair<uint16_t, uint16_t>> final;
 6      if (odd.size() == 2)
 7      {
 8          final.push_back(make_pair(odd[0], odd[1]));
 9      }
10      else
11      {
12          uint16_t first = *odd.begin();
13          odd.erase(odd.begin());
14
15          uint16_t min_distance =
16              numeric_limits<uint16_t>::max();
17
18          for (uint16_t i = 0; i < odd.size(); i++)
19          {
20              auto odd_j = odd;
21              uint16_t second = odd[i];
22              odd_j.erase(odd_j.begin() + i);
23
24              auto final_tmp = LPC(odd_j);
25
26              //defining local minimum
27              uint32_t min_id = 0;
28              uint16_t min_distance_local =
29                  numeric_limits<uint16_t>::max();
30              for (uint32_t j = 0; j < final_tmp.size(); j++)
31              {
32                  uint16_t total_distance =
33                      distances[first][second] +
34                      totalDistance(final_tmp[j], distances);
35                  if (total_distance < min_distance_local)
36                  {
37                      min_id = j;
38                      min_distance_local = total_distance;
39                  }
40              }
41
42              //defining global minimum
43              if (min_distance_local < min_distance)
44              {
45                  final.clear();
46                  final.push_back(make_pair(first, second));
47                  copy(final_tmp[min_id].begin(), final_tmp[min_id].end(),
48                      back_inserter(final));
49                  min_distance = min_distance_local;
50              }
51          }
52      }
53      return final;
54  }
```

Listing 2: List Pairs Combination Base

IV. Parallel Approach

In this section, two parallel versions of the Step 5 of the Postman Algorithm are described.

A. Parallel Version 1

The parallel version 1 (V1) has a simple implementation and executes sets of LPCB iterations in parallel. In this version, the *for* (in Listing 2) that starts on line 18 and ends on line 51 is parallelized using the directive *#pragma omp parallel for*.

To make this version functional, a reduction for the variable *final* was developed. The variables minimum distance and vector of pairs (lines 45 to 49), in this version, are stored in a data structure in each thread. At the end of the execution of the threads, occurs the reduction and the best result is placed in the variable *final*.

The parallelization implemented in this version can lead to limitations on scalability. For example, running LPCB-V1 for 18 odd vertices in a computer with 20 cores makes two cores useless to speed up the LPCB.

B. Parallel Version 2

The parallel version 2 (V2) tries to overcome the scalability problem from V1. This parallelization is divided into two parts. The first one calls the LPC method (line 24 from Listing 2) using *#pragma omp single*. Then, the method LPC launches the content inside the *for* (line 16 from Listing 1) as a task, using *#pragma omp task*, until the size of the variable odd became equal to a parameter called min_seq. After that, the recursive calls run sequentially.

Figure 2 exemplifies the parallel recursion tree considering $min\_seq = 4$. For a problem with ten odd vertices, LPCB will make nine LPC calls with eight odd vertices. In each one, LPC will create seven tasks composed of six odd vertices. In the next recursion levels, as the variable odd will have a size

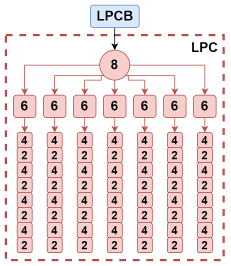

Fig. 2: Parallel recursion tree for 8 odd vertices, $min\_seq = 4$.

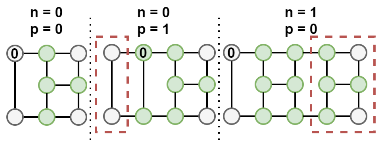

Fig. 3: Graph generation example. Extra vertices added by changing p and n are highlighted in red. Odd vertices in green.

smaller or equal to four, and $min\_seq = 4$, the recursions will occur sequentially.

The second part of the parallelization V2 parallelizes the *for* that starts on line 30 and ends on line 40, in Listing 2, using the directive *#pragma omp parallel for*. Besides that, a reduction for the variables min_id and $min\_distance\_local$ (lines 32 to 39) was developed. It works analogously to the reduction in V1.

V. Experimental Environment

The algorithm was implemented using the C++ programming language. The parallel version of the algorithm uses the OpenMP 5.1 API. Two computer scenarios were used to perform the experiments.

Computer A: Architecture UMA with CPU Intel i7-9750H@2.6GHz (6 cores), 16GB of RAM, using the Ubuntu 18.04 LTS OS. Experiments were repeated 50 times.

Computer B: Architecture NUMA with two nodes using CPU Intel Xeon E5-2640v4@2.4GHz (10 cores), 128GB of RAM, using Ubuntu 16.04 LTS OS. Experiments were repeated 25 times.

The experiments were conducted using undirected graphs following a pattern. Because of the algorithm nature, the system must compute all possibilities to odd vertices independent of patterns in the input. Therefore, this simplification in the graph generation does not influence the results.

The algorithm that generates graphs makes undirected graphs using two parameters n and p. Figure 3 shows examples of graphs generated varying p and n. The parameter p adds vertices on the left side of the base graph, while parameter n adds vertices on the right side.

The experiments were performed using five graph sizes. They are summarised in Table I.

VI. Results and Discussions

The experiments are divided into parameter definitions and comparisons between versions. Subsection VI-A shows parameter definition. Next, Subsections VI-B and VI-C perform comparisons between the parallel versions V1 and V2, using Computer A and B, for different numbers of threads. The speedup results were computed using the data from Table II. Experiments using 20 odd vertices were conducted only in V2 because of limitations in memory imposed by V1.

TABLE I: Graphs used in the experiments.

n	p	Vertices	Odd Vertices	Edges
0	1	10	6	13
1	0	14	10	19
2	0	20	16	28
2	1	22	18	31
2	2	24	20	34

TABLE II: LPCB sequential version results.

	Computer A		Computer B	
Odd Vertices	AVG Time	STD Time	AVG Time	STD Time
6	5,165 us	3,7971 us	-	-
10	296,566 us	16,2373 us	-	-
16	1,265 s	0,0245 s	-	-
18	24,663 s	0,4377 s	45,158 s	0,2820 s
20	562,715 s	6,8997 s	1018,465 s	3,8720 s

A. Parameter definition

The experiments performed in this section were executed using Computer A running with six threads. These experiments used this computer because it is a machine that could be accessed locally. Besides that, it was used 18 or 20 odd vertices in most of the experiments because they were cases with high runtime. Also, these experiments help to define some parameters from the parallel versions, however, the results probably are not optimum.

The iterations of LPCB can have similar runtimes. However, when the *if* on line 36 is true most of the time, the iteration runtime grows. Differently, when the *if* is false most of the time, the runtime decreases. Static scheduling is a good approach for iteration runtimes well defined. Dynamic scheduling deals better with different iteration runtimes, but with an extra cost to allocate iterations. Therefore, in version V1, it is interesting to compare dynamic and static scheduling types. In the dynamic scheduling experiment, the chunk size was defined as one, once that the number of iterations is given by the number of odd vertices. For 18 odd vertices, the results showed that the static scheduling reach a speedup of 3.79 times and the dynamic scheduling 3.81 times. Therefore, the dynamic scheduling was chosen as default in V1.

In the parallelization V2, the value of min_seq can influence the speedup. The best value for min_seq depends on factors like the number of threads or odd vertices. Therefore, it was conducted an experiment to define a value for min_seq to be used in the next experiments. The results are summarised in Table III. For cases with 6 or 10 odd vertices, the parallel version had a worse performance than the sequential, probably because of the cost to create tasks. Besides that, using min_seq equal to 6 or 8 brings the best results. However, for 20 odd vertices, $min\_seq = 8$ has the biggest speedup. Thus, this value was chosen as default in V2.

TABLE III: Speedup measurements using different min_seq values, varying number of odd vertices.

Odd Vertices	min_seq 6	8	10	12
6	0,001	0,001	0,001	0,001
10	0,048	0,055	0,054	0,050
16	4,223	4,170	4,048	3,331
18	3,806	3,799	3,741	3,647
20	3,929	3,946	3,911	3,828

TABLE IV: Speedup for static and guided scheduling in V2.

Odd Vertices	Static	Guided
18	2,470	2,479
20	2,232	2,249

In the *for* (Listing 2) parallelized in V2, it was tested static and guided scheduling. The number of iterations in this *for* is given by $(n-3)!!$ according to the number of odd vertices. Also, as in V1, there are scenarios where iterations have very different runtimes. Therefore, the guided scheduling can help the system to adapt to different problem sizes and iteration runtimes. The results in Table IV confirmed that using guided scheduling brings the best results.

B. *V1 vs V2 – Computer A*

In Computer A, the number of threads was varied from 2 to 6. The results in Figure 4 shows that the speedup is limited to approximately 4 times. Besides that, for 18 odd vertices, both parallelization had similar results. This may happen because 18 is a multiple of 6, this way the work is probably divided equally among threads.

C. *V1 vs V2 – Computer B*

Figure 5 shows experiments results performed with 18 odd vertices. In this scenario, it could be observed that V1 had a bigger limitation on scalability than V2, as foreseen in its development. Also, there is a speedup drop in 12 threads, probably caused by the memory allocation in the NUMA architecture. This way, when the system uses over 10 threads, they need to access remote memory blocks, which reduces the performance gain.

In the experiment with 20 odd vertices, Figure 6, the speedup curve had similar behaviour to the experiment with 18 odd vertices.

Theoretically, the influence in memory accesses, caused by the architecture NUMA, should not be observed in experiments up to 10 threads. Therefore, these experiments should have had better results compared with the ones conducted in Computer A. However, in all cases, Computer A had better results than Computer B. Thus, it is necessary a deeper investigation to determine what is causing these worse results.

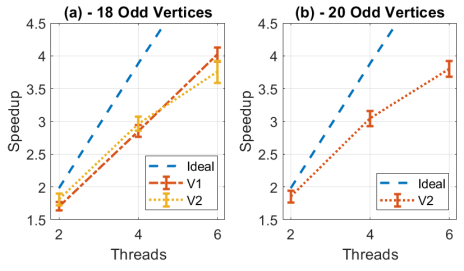

Fig. 4: Computer A – Speedup for 18 and 20 odd vertices.

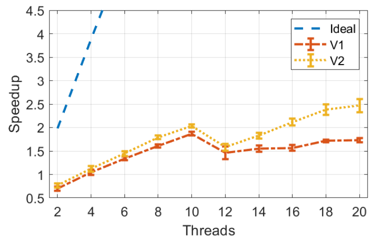

Fig. 5: Computer B – Speedup for 18 odd vertices.

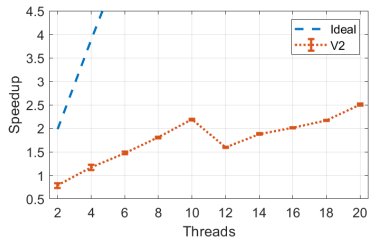

Fig. 6: Computer B – Speedup for 20 odd vertices.

VII. Conclusion

This work aimed to speed up Step 5 in Algorithm 1, developing two parallel versions, V1 and V2. It was performed experiments to define parameters as scheduling type and parallelization level in recursion. After that, V1 and V2 were compared in UMA and NUMA computer architectures, computers A and B, respectively. When varied the number of threads, Computer A showed better results reaching up 4 times speedup, against only 2.5 times for Computer B. Besides that, it was proven that V1 has scalability limitations. Also, it showed that the NUMA architecture limited the performance because of remote memory access.

The speedup aim was reached in both versions. However, parallelization V2 is the most promising. Further study is needed to understand the cause of its poor results in Computer B. Besides that, it would interest to adapt V2 to allow bigger problem sizes and perform more detailed studies on its parameters. Then, it could be compared with other techniques.

References

[1] Mei-Ko Kwan, "Graphic programming using odd or even points," *Chinese Mathematics*, vol. 1, pp. 273–277, 1962.
[2] J. Edmonds and E. L. Johnson, "Matching, euler tours and the chinese postman," *Mathematical Programming*, vol. 5, no. 1, pp. 88–124, 1973.
[3] R. Mannadiar and I. Rekleitis, "Optimal coverage of a known arbitrary environment," in *2010 IEEE International Conference on Robotics and Automation*, 2010, pp. 5525–5530.
[4] K. R. Saoub, *Graph theory : an introduction to proofs, algorithms, and applications*. CRC Press, 2021.
[5] Z. Wang, X. Bao, and T. Wu, "A parallel bioinspired algorithm for chinese postman problem based on molecular computing," *Computational Intelligence and Neuroscience*, vol. 2021, 2021.