Parallel implementations of TSP brute force algorithm

Rafael Almenara Ventura Alves[1], **Maria Clicia S. Castro**[2], **Cristiana Bentes**[1]

[1]Faculdade de Engenharia

[2]Instituto de Matemática e Estatística
Universidade do Estado do Rio de Janeiro (UERJ), Rio de Janeiro, Brasil

almenara.r@gmail.com, clicia@ime.uerj.br, cris@eng.uerj.br

Abstract. *The traveling salesman problem is a famous combinatorial optimization problem. Despite its great importance, there is no good exact solution for the generic case. Considering the existence not only of CPUs with many cores but also of GPUs allowing a more generic use and being more popular, this work tries to answer the question: up to what size of problem is it possible to solve in parallel through brute force in a timely manner? For this, implementations were developed in OpenMP, MPI, and CUDA.*

1. Introduction

The Traveling Salesman Problem is a classic mathematical problem. Its main goal is to find the shortest route that a salesman can take when visiting a set of cities. The salesman starts at an initial city, should visit each city only once, and returns to the starting city. It is a well-known combinatorial optimization problem that has many applications, from finding better logistic routes to improving chemical processes and integrated circuit fabrication [Cook et al. 2011].

Despite its great importance, TSP is an NP-Hard problem in which the number of possible orders of visits to the cities grows exponentially with the number of cities. For this reason, there is no efficient algorithm that produces the exact solution for a generic case. The only algorithm that is able to give the exact solution to the generic case of TSP is the brute force algorithm. The brute force algorithm checks all of the $(n-1)!$ route possibilities of visiting n cities, and establishes which is the shortest route among all of them. Therefore, for big values of n, it becomes infeasible to find the best route of TSP using the brute force algorithm [Garey and Johnson 1979].

In this work, we propose to explore parallel processing in order to make the brute force algorithm feasible for a greater value of n than the ones usually handled by single machines that solve the problem sequentially. We propose to exploit the parallelism present in the most common computer architectures today: multicore processors, clusters of computers, and graphics processing units (GPUs). For each of these architectures, we propose the use of a different parallel programming model.

Cadernos do IME - Série Informática
e-ISSN: 2317-2193 (online)
DOI: 10.12957/cadinf.2024.79821

For the multi-core architecture, we propose a shared memory algorithm implemented using OpenMP. For the distributed architecture of a cluster of computers, we propose a message-passing algorithm using MPI. For the massively parallel architecture of a GPU, we propose a fine-grain parallel algorithm using CUDA. We evaluate the performance of the brute force algorithm in these three parallel environments and show some important optimizations provided in each programming model.

Our results show that it is possible to gain speed in solving brute force TSPs through parallelism but at the cost of large clusters and a lot of energy. Even so, for still small sets of cities, in the order of dozens. Modern applications require sets of hundreds and even thousands of cities, which still seems a long way off.

The remainder of this paper is organized as follows. Section 2 shows some of the most related works. Section 3 presents the brute force algorithm to solve the TSP. Section 4 presents the parallel algorithms proposed: OpenMP, MPI and CUDA. Section 5 shows the performance results of our parallel algorithms. Finally, in Section 6 we conclude our work and show some directions for future work.

2. Related Work

The Travelling Salesman Problem (TSP) has traditionally deterministic methods, such as brute force or Greedy approach [Sahalot and Shrimali 2014, Baidoo and Oppong 2016], to solve it with the exact solution. They are the oldest solutions that do not require auxiliary information or probabilistic procedures. The brute force methods at all times return the correct result. They are good for testing the correctness of faster algorithms.

Brute force algorithms require exponential computation for every new city added. Gohil et al. [Gohil et al. 2022] proposed parallel implementations of the brute force algorithm that are close to our solutions. They used the shared memory, distributed memory, and GPU paradigms. Our proposal exploits the same paradigms and APIs but with different algorithms and execution environments. In terms of the permutation generation, we exploit and compare two approaches: the lexicographic order [Djamegni and Tchuenté 1997] and the Johnson and Trotter algorithm [Johnson 1963]. Our GPU implementation is also different because we propose different strategies for the reduction operation.

Currently, there are methods to solve the TSP problem using heuristics with different approaches to avoid the exponential computational time [Laporte 1992]. Although these heuristics are not exact methods, they can deal with a larger number of cities. They focus on various aspects such as processing time, development effort, and solution quality. Examples of these solutions, with different implementations, are Genetic Algorithm (GA) [Grefenstette et al. 2014], Ant Colony Optimization (ACO) [Bianchi et al. 2002, Mavrovouniotis et al. 2017], and Simulated Annealing (SA) [Johnson et al. 1989, Johnson et al. 1991].

3. The brute force algorithm for the Traveling Salesman Problem

The main idea behind the brute force algorithm is to search from all possible routes for the one with the smallest cost. The algorithm is quite simple. From an input of n cities, each permutation of the n cities is a possible route. The algorithm computes the route cost assembled by each permutation and keeps the one with the lowest one. The details of each step of the algorithm are described in the following sections.

3.1. Read Input

The first step of the program is to receive the input data. The input data is a set of geographical locations, called cities, represented by cartesian coordinates. Each input line contains a string of type "x y", where x and y are integers representing city coordinates. The x coordinates are stored in an array X and the y coordinates are stored in an array Y, where the index of these arrays is the index of the city.

3.2. Calculate Distances

In this step, the Euclidean distance between all pairs of cities is computed and stored in a distance matrix, D. The distances will be used in the next steps to evaluate the cost of the route. The computation of the Euclidean distances of each pair of cities is described in Algorithm 1.

Algorithm 1: Calculate distances between coordinates.

```
1 Initialize matrix $D$ with zeros;
2 $n \leftarrow$ length of arrays $X$ and $Y$;
3 for $i \leftarrow 1$ to $n$ do
4     for $j \leftarrow 1$ to $n$ do
5         $D[i][j] \leftarrow \sqrt{(X[i] - X[j])^2 + (Y[i] - Y[j])^2}$;
6     end
7 end
```

3.3. Build the Initial Route

The initial route is built arbitrarily as an increasing sequence of integers from 0 to $n - 1$, where n is the number of cities. For example, for 5 cities, an array [0 1 2 3 4 0] is built, where the first position of the array represents the first city, the second position is the second city on the route, and so on. Table 1 shows an example with 3 cities and its coordinates, a route [0 2 1 0] is: [(325,492), (878,1204), (552,890), (325,492)].

City Index	Coordinates
0	(325,492)
1	(552,890)
2	(878,1204)

Table 1. Example with 3 cities and its coordinates

3.4. Compute the Route Cost

The computation of the cost of each route is very simple. The pre-calculated values of the distances between each two cities are recovered from the distances matrix D and added together to compose the route cost. Then the cost of the route is compared with the best route found so far, if the cost is smaller than the best route, it is saved as the best route. The Algorithm 2 describes this step.

Algorithm 2: Calculate sum of distances among the cities in the route.

```
1 Initialize sum $S$ to 0;
2 $n \leftarrow$ length of array $A$;
3 for $i \leftarrow 1$ to $n$ do
4     $S \leftarrow S + D[A[i]][A[i+1]]$;
5 end
6 if $S < BestValue$ then
7     $BestValue \leftarrow S$;
8 end
```

3.5. Next permutation calculation

The most important step in the brute force algorithm is the computation of the next permutation (or route) to be analyzed. It is the most expensive step in the brute force computation and special attention should be given to it.

There are plenty of different algorithms proposed in the literature to generate the permutations of a given set of numbers [Sedgewick 1977]. Only a few algorithms, however, have proven to be good candidates for the brute force TSP since the problem requires certain conditions, and they need to be able to be parallelized. For example, a great number of permutation algorithms are based on recursion. However, the recursive algorithm is not usually well suited to parallelism, since each iteration of the recursion depends on the memory of the previous iteration, making it difficult to distribute the tasks. In addition, not defining the total number of iterations can lead to rapid exhaustion of the scarce individual resources of each thread. In this work, we studied two different permutation generation algorithms: Johnson and Trotter [Johnson 1963] and Lexicographic generation [Djamegni and Tchuenté 1997].

3.5.1. Johnson and Trotter Algorithm

The Johnson and Trotter algorithm has an iterative version and a recursive one. The Algorithm 3 shows the iterative version. At the core of the algorithm are the concepts of mobile elements and their directions. An element is mobile if it is larger than the element it points to in its designated direction. The direction of an element is determined by the relative order of the elements in the permutation. The algorithm starts by initializing a permutation, often as the identity permutation (i.e., elements in ascending order). It then identifies the mobile elements within the permutation. The algorithm systematically rearranges the mobile elements to produce the next permutation, repeating until no more mobile elements can be found. This procedure ensures the generation of all permutations.

3.5.2. Lexicographic Generation

The lexicographic order generation algorithm calculates the permutations without using auxiliary structures. Another important aspect of this algorithm is that it is able to calculate any permutation k without having to calculate all the previous permutations. This

Algorithm 3: Johnson-Trotter permutation generation algorithm.

```
Initialize array $D$ (list of decreasing values from $n$ to 1);
Initialize array $P'$ as a copy of $P$;
Initialize a list $permutations$;
while There are mobile elements in $P'$ do
    Find the largest mobile element $m$ in $P'$;
    for Each element $x$ in $P'$ do
        if $x > m$ then
            Swap $x$ and $m$ in $P'$;
            Swap the corresponding elements in $D$;
            Add a copy of $P'$ to $permutations$;
            Break;
        end
    end
    Reverse the direction of all elements in $P'$ and $D$ that are greater than $m$;
end
return $permutations$;
```

possibility has some advantages for a parallel implementation. The Lexicographic generation algorithm is described in Algorithm 4 and Algorithm 5.

The algorithm begins with the initial permutation, often sorted in ascending order. It continuously generates the next permutation by finding the rightmost element in the sequence that can be modified to produce a lexicographically larger permutation. The selected element is replaced with the smallest element to its right that is greater than itself, ensuring a minimal change that maintains the lexicographic order, a sequence where each permutation follows the previous one based on its natural order. This process repeats until all permutations are generated.

Algorithm 4: Lexicographic permutation generation.

```
Sort $P$ in ascending order;
Initialize a list $permutations$;
while NextPermutation($P$) is true do
    Add a copy of $P$ to $permutations$;
end
return $permutations$;
```

4. Parallelizing the brute force TSP

The brute force algorithm can be parallelized using two strategies: (1) parallelize the search step for the next permutation, or (2) divide the problem into sub-tasks, where each sub-task is composed of a set of routes, the final solution is the best among the local solutions of each sub-tasks [Cesari 1996]. In this work, we used strategy 1 with Johnson and Trotter, and strategy 2 with Lexicographic Generation and compared both.

Algorithm 5: Finding the next permutation in lexicographic generation.

```
1 Find the largest index $i$ such that $P[i] < P[i+1]$;
2 if No such index $i$ exists then
3     return false;
4 end
5 Find the largest index $j$ such that $P[i] < P[j]$;
6 Swap elements $P[i]$ and $P[j]$;
7 Reverse the subarray from $P[i+1]$ to the end of the array $P$;
8 return true;
```

Following, we describe the three parallel implementations proposed that exploit different parallel programming paradigms: shared memory, message passing, and GPU computing. For all of them, the read input and the build of the distance matrix are made sequentially.

4.1. Shared Memory Implementation

The parallel implementation of the brute force algorithm that exploits the shared memory paradigm is based on the widely adopted parallel programming specifications OpenMP [Chapman et al. 2007].

4.1.1. Implementation with Johnson and Trotter

In the parallel implementation of the Johnson and Trotter permutation generation algorithm, it is not possible to divide the routes into sub-tasks because each permutation depends on the previous one to be found. However, since the process of finding the next permutation is iterative, we can parallelize the loop that iterates through each element of the array P' (line 6 of Algorithm 3).

In this implementation, the construction of the initial route and the computation of the route cost are made sequentially. When a new permutation is required, the parallel algorithm creates several threads to find the next permutation. Each thread compares an element of P' with the largest mobile M, swapping elements to find the next permutation if it's bigger. Then, all threads are synchronized to the main thread to calculate the cost.

4.1.2. Implementation with Lexicographic Generation

For lexicographic generation, the total number of possible routes, R, is divided by the number of threads, T. So, each thread should calculate the cost of R/T routes. The threads create their first permutation (route) based on the thread ID and the number of routes per thread. Lexicographic generation allows us to get a K^{th} permutation without passing by the previous permutations.

For example, for 5 cities and 3 threads, there are 120 possible routes, and each thread is responsible for computing the cost of 40 routes. Thread 0 starts at route 0, thread 1 starts at the 40^{th} route, and thread 2 at the 80^{th} route. Each thread generates its initial route, calculates the route cost, generates the next permutation, and so on, repeating this R/T times. The work done by each thread is described in Algorithm 6.

Each thread computes the local best route and saves this value in a shared array indexed by the thread ID. After all threads finish their computation, the main thread compares the local best routes to find the best one. This procedure is described in Algorithm 7.

Algorithm 6: Work performed by a single thread.

```
1 $Permutation \leftarrow$ FindKPermutation($ThreadID * RoutesPerThread$);
2 for $i \leftarrow 1$ to RoutesPerThread do
3     AnalyzePermutation($Permutation$);
4     $Permutation \leftarrow$ NextPermutation($Permutation$)
5 end
```

Algorithm 7: Split the routes among threads and find the global result.

```
1 $BestValue \leftarrow \infty$ ;
2 Initialize empty array $BestLocalResults$;
3 $RoutesPerThread \leftarrow NumberOfCities!/NumberOfThreads$;
4 for 1 to NumberOfThreads do
5     CreateThread($RoutesPerThread, BestLocalResults$)
6 end
7 Await threads to finish;
8 for $i \leftarrow 1$ to NumberOfThreads do
9     if $BestLocalResults[i] < BestValue$ then
10        $BestValue \leftarrow BestLocalResults[i]$;
11    end
12 end
```

4.2. MPI implementation

In the message-passing paradigm, we decided to implement only the lexicographic permutation generation, since this is the computation of each route independent and does not require communication between processes. Therefore, the MPI implementation has a work division similar to the OpenMP implementation. For P MPI processes and R possible routes to be evaluated, each process evaluates R/P routes. The distributed implementation, however, requires that the number of routes be analyzed and the matrix with the distances between the cities are sent to each process. At the end of the local calculations, each process sends its best route back to process 0. After process 0 receives the messages from all processes, it finds the best route among the local solutions. The Algorithm 8 describes the MPI implementation.

4.3. GPU implementation

In order to take advantage of the GPU massive parallel environment, we chose the lexicographic permutation generation algorithm for the parallel implementation. Since the

Algorithm 8: Split the routes among processes and find the global result.

```
1  $BestValue, BestLocalValue \leftarrow \infty$ ;
2  Initialize empty array $ProcessesID$;
3  $RoutesPerProcess \leftarrow NumberOfCities!/NumberOfProcesses$;
4  for 1 to NumberOfProcesses do
5      $ProcessID \leftarrow$ CreateProcess($RoutesPerProcess$) ;
6      SendData($ProcessID$, $DistanceMatrix$);
7      Append $ProcessID$ to $ProcessesID$ array;
8  end
9  for $i \leftarrow 1$ to length($ProcessesID$) do
10     $BestLocalValue \leftarrow$ ReceiveData($ProcessID$) ;
11     if $BestLocalValue < BestValue$ then
12         $BestValue \leftarrow BestLocalValue$;
13     end
14 end
```

lexicographic generation allows evaluating a number of routes in parallel, the GPU implementation assigns a number of routes to each thread.

The main algorithm starts reading the input data and building the distance matrix in the CPU. The next step is to transfer the distance matrix to the GPU global memory. After that, the CUDA threads are started, and each of them is responsible for evaluating a set of routes based on the thread ID. In the end, there is a need for a reduction operation in order to determine the best route from all the threads. We implemented two versions of this algorithm that employ different ways to perform the reduction operation in finding the best route.

4.3.1. Reduction in the CPU

The first version implements the reduction operation in the GPU global memory. The idea is to create a global array with one entry for each thread. So, each thread saves in this array its local best route found. After all threads have finished, this array is sent to the CPU, which goes through all the elements to find the smallest.

4.3.2. Reduction in the GPU

We also implemented an alternative version for finding the best route. As the data to be reduced is on the GPU, it is possible to use the massive parallelism of the GPU to perform the reduction operation. After all the CUDA threads have found their local best solution, the first half of the CUDA threads, starting from thread 0, are selected to search for the global solution. Each of these threads compares elements i and $i + T$, where i is the thread ID and T is the number of threads operating the reduction at that moment. After the comparison, the smallest value is saved in the position i of the array. As the number of elements in the array has been halved, one-quarter of the threads will now be used to repeat the process. This procedure continues until the array has only one element. So, position 0 stores the best global solution. This value is sent to the CPU to display the result and end the program.

4.3.3. Optimizing Memory Access

We also implemented a third version of the GPU implementation that takes advantage of the faster-access memory inside the GPU, the shared memory. Since each CUDA thread accesses many times the distance matrix, we transferred part of this matrix to the shared memory of each Streaming Multiprocessor of the GPU. The idea is that thread 0 from each block transfers part of the distance matrix to the respective shared memory. When the routes are evaluated by threads, they read the distance between the cities in a faster way. For this implementation, we implemented the reduction operation in the CPU.

5. Experimental Results

The goal of any parallelization strategy is to reduce the execution time for a given problem. One way of measuring this is by calculating *speedup*, a performance metric that tells you how much parallel processing gains over sequential processing. There are various definitions of *speedup* and, consequently, various ways of calculating it. This work uses the definition of absolute *speedup*: the *speedup* is equal to the time spent by the serial implementation divided by the time spent by the parallel implementation [Sun and Ni 1990].

5.1. Execution environment

Two execution environments were used in our experiments. Machine A consists of an Intel® Core™ i5-13400F CPU, 16GB of RAM, and an Nvidia GeForce RTX 4060 GPU using Windows 10. Machine B consists of an AMD Ryzen 7 2700 CPU, 16GB of RAM, and an Nvidia GeForce RTX 3060 GPU using Ubuntu 18.04. Table 2 shows the detailed information about the CPUs used, and Table 3 shows the information about the GPUs used.

Machine	CPU	Number of cores	Clock Base	L1 Cache	L2 Cache
A	Intel® Core™ i5-13400F	10	2.5GHz	800KB	12MB
B	AMD Ryzen 7 2700	8	3.2GHz	768KB	4MB

Table 2. CPUs used in our experiments

Machine	GPU	Architecture	CUDA Cores	Clock Base	Global memory
A	RTX 4060	Ada Lovelace	3072	1830MHz	8GB
B	RTX 3060	Ampere	3584	1320MHz	12GB

Table 3. GPUs used in our experiments

We executed all the experiments ten times in each machine. The results are the average of these ten executions.

5.2. Comparing the Permutation Algorithms

Table 4 and 5 shows the execution times of the sequential brute force TSP algorithm for up to 15 cities using the two forms of generating the permutations, Johnson and Trotter and Lexicographic. We can observe in this table that Lexicograph generation performs much better than the Johnson and Trotter algorithm. For 13 cities, it is more than 5 times faster. It was not possible to complete the execution for 14 cities or more with Johnson and Trotter algorithm.

Cities	Johnson and Trotter (s)	Lexicographic (s)
6	0.001	0.001
7	0.001	0.001
8	0.008	0.001
10	0.473	0.011
11	4.487	0.088
12	49.582	0.987
13	878.641	12.329
14	>999.999	161.214
15	>9999.999	2361.746

Table 4. Execution times (in seconds) of the two sequential permutation generation algorithms in machine A

Cities	Johnson and Trotter (s)	Lexicographic (s)
6	<0.000	<0.000
7	<0.000	0.001
8	0.003	0.001
9	0.015	0.004
10	0.087	0.022
11	0.774	0.165
12	8.644	1.820
13	103.350	23.183
14	>999.999	323.226
15	>9999.999	4803.881

Table 5. Execution times (in seconds) of the two sequential permutation generation algorithms in machine B

5.3. OpenMP Results

Table 6 and 7 shows the execution times in seconds of the parallel OpenMP implementations of the brute force algorithm with 10 and 70 threads. The overly elevated execution times of the parallel implementation with the Johnson and Trotter algorithm prevented us from executing it with more than 13 cities.

Figure 1 shows the speedup of these executions when compared to the sequential algorithm. We can observe in this figure that the parallel implementation with the Johnson and Trotter algorithm provides speedups below 1, which means that the parallel algorithm is slower than the sequential one.

In the Johnson and Trotter algorithm, the most compute-intensive loop is executed several times, but each execution contains only a few iterations. In this case, the loop parallelization implies overheads of creating and terminating the threads, but the amount of work computed by each thread is not significant.

Also, in Figure 2, it can be seen that when the lexicographic permutation algorithm is used, we obtained performance gains when compared to the sequential algorithm. For

Cities	Johnson and Trotter (10 threads)	Lexicographic (10 threads)	Lexicographic (70 threads)
6	0.019	0.016	0.033
7	0.038	0.016	0.033
8	0.079	0.016	0.033
9	0.611	0.016	0.033
10	5.155	0.016	0.033
11	50.228	0.030	0.049
12	571.566	0.260	0.208
13	>999.999	2.687	2.240
14	>9999.999	38.780	30.879
15	>99999.999	595.980	445.360

Table 6. Execution times (in seconds) of OpenMP implementations in machine A

Cities	Johnson and Trotter (10 threads)	Lexicographic (10 threads)	Lexicographic (70 threads)
6	<0.000	0.001	0.004
7	0.001	0.001	0.004
8	0.007	0.001	0.004
9	0.029	0.001	0.004
10	0.252	0.007	0.009
11	2.667	0.038	0.031
12	31.498	0.315	0.238
13	408.252	3.759	2.799
14	>9999.999	50.626	37.797
15	>99999.999	700.732	563.176

Table 7. Execution times (in seconds) of OpenMP implementations in machine B

this implementation, there is a substantial amount of work for each thread to compute and the overhead of creating and terminating the threads is compensated. When we compare the speedups obtained for using 10 and 70 threads, we can observe that using 70 threads provides higher speedups because this is a memory-bound application and the memory access latency can be hidden by processing another thread.

5.4. MPI Results

Table 8 shows the execution times in seconds of the parallel MPI implementation of the brute force algorithm running in machine A with 10 (number of physical CPU cores) and 16 (physical plus performance CPU cores) processes. Table 9 shows the same implementation in machine B with 8 (total CPU cores) processes. In Figure 3, we plot the speedup curves of all the MPI implementations.

However, it is important to notice that this is a small environment for the MPI execution. Running this implementation on a cluster of computers would be more correct, where more scalable results would be achieved. The use of a cluster is a possible future work.

5.5. GPU Results

Table 10 shows the execution times in seconds for execution in machine A of the three CUDA implementations of the brute force algorithm: (1) Reduction in the CPU; (2) Reduction in the GPU; (3) Shared Memory. Table 11 shows the execution times for the three implementations running in machine B.

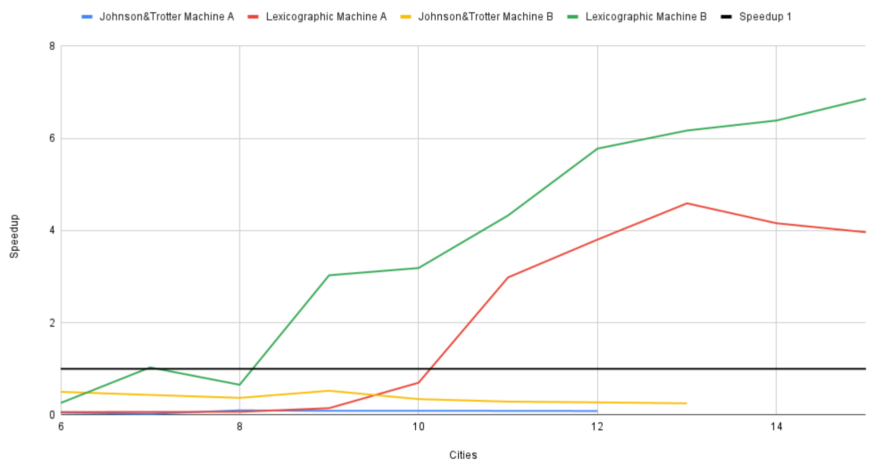

Figure 1. Comparing the speedup of OpenMP implementations on machine A and B for both algorithm strategies with different cities size

Cities	Time for 10 Processes (sec)	Time for 16 Processes (sec)
6	0.001	0.001
7	0.001	0.001
8	0.001	0.001
9	0.001	0.001
10	0.004	0.003
11	0.016	0.012
12	0.144	0.115
13	1.780	1.348
14	25.450	17.730
15	406.122	287.661

Table 8. Execution times (in seconds) of MPI implementation in machine A

In both tables, we can observe that there was no significant difference between the execution times of the two different strategies for the reduction operation (performed by the CPU or by the GPU). The reduction performed by the GPU may be preferable when the amount of data grows. In the cases tested, the implementation of the reduction in the CPU is still quite efficient for the reduction of an *array* in the order of a few tens of thousands of items. Because of this, and the lower complexity of its implementation, the shared memory version of the CUDA implementation performs the reduction in the CPU.

Figure 4 shows the speedups of the CUDA implementations in the two machines. Firstly, the speedups obtained in machine B are impressive, over 70x. This confirms that the massively parallel environment of the GPU is adequate to accelerate the brute force algorithm and compute the exact solution of TSP in a reasonable time (for up to 16 cities at least). Secondly, exploiting the GPU shared memory for the distance matrix has a substantial impact on the brute force performance that is even more pronounced in machine B where the speedup increases from around 70x to over 100x. Thirdly, when we compare the performance of machine A and machine B, we can observe that the increased number of cores of the GPU of machine B makes a huge difference in the overall performance.

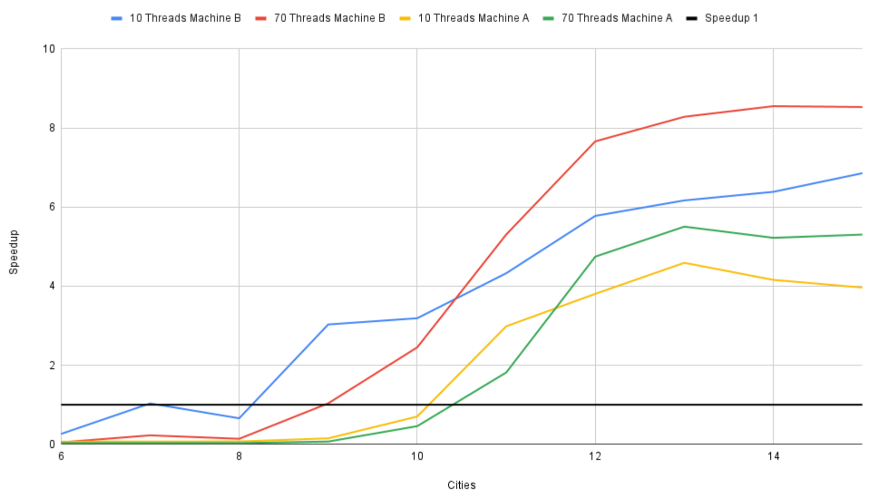

Figure 2. Comparing the speedup of OpenMP implementations on machine A and B with Lexicographic Algorithm with different cities size

Cities	Time for 8 Processes (sec)
6	<0.000
7	<0.000
8	<0.000
9	0.001
10	0.006
11	0.030
12	0.288
13	3.635
14	49.800
15	746.017

Table 9. Execution times (in seconds) of MPI implementation in machine B

Cities	Reduction in the CPU	Reduction in the GPU	Shared memory
10	0.015	0.016	0.012
11	0.018	0.018	0.013
12	0.053	0.054	0.032
13	0.445	0.442	0.234
14	5.662	5.668	3.219
15	83.386	84.555	48.826

Table 10. Execution times of CUDA implementations (in seconds) for machine A

6. Conclusions

In this work, we presented parallel implementations of the TSP brute force algorithm using different parallel programming paradigms, OpenMP, MPI, and GPU programming with CUDA. We carried out experiments on two different machines and obtained speedups for the three parallel paradigms. We observed that there is a minimum size of 12 cities to achieve reasonable speedups. We also demonstrated that the massive power of a modern GPU allowed us to evaluate 16! routes in a few minutes and also allowed us to achieve speedups of more than 70 times.

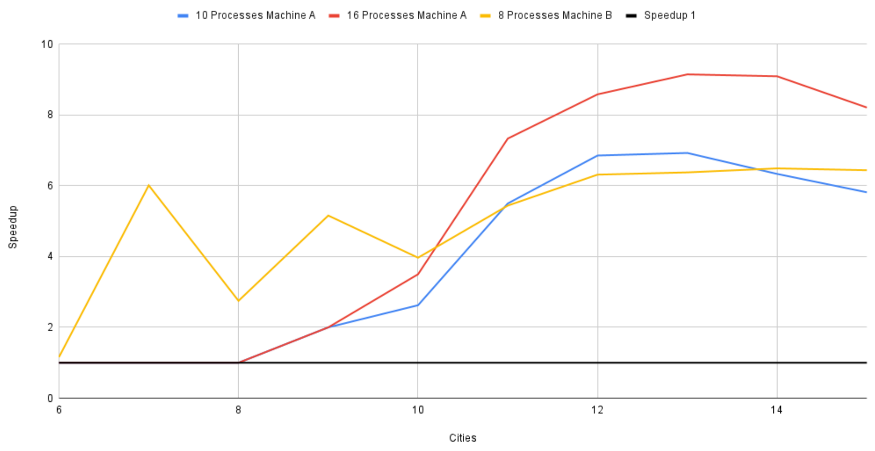

Figure 3. Comparing the speedup of MPI implementations on machine A and B in different cities size

Cities	Reduction in the CPU	Reduction in the GPU	Shared memory
10	0.011	0.011	0.012
11	0.012	0.011	0.013
12	0.032	0.032	0.032
13	0.296	0.296	0.234
14	4.648	4.650	3.219
15	69.074	69.077	48.826

Table 11. Execution times of CUDA implementations (in seconds) for machine B

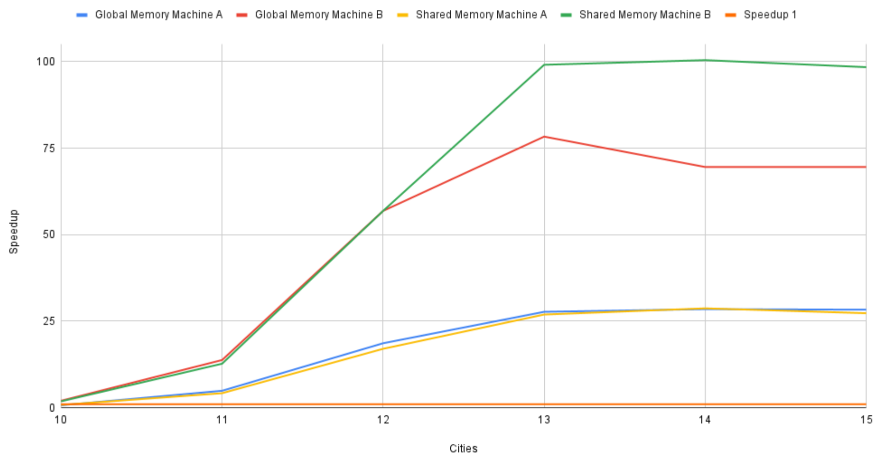

Figure 4. Comparing the speedup of CUDA implementations on machines A and B in different cities sizes

When the number of cities increases, however, the brute force solution would require a high computational cost. In addition, values greater than $20!$ do not fit in current registers. In this case, it would be necessary to partition this number into multiple registers, increasing the amount of processing and memory accesses, which may even compromise the gains of parallel implementations.

For future work, we envision a solution that combines all the parallel programming paradigms and takes advantage of a distributed system that contains multicore processors and GPUs.

References

Baidoo, E. and Oppong, S. O. (2016). Solving the tsp using traditional computing approach. *International Journal of Computer Applications*, 152(8):13–19.

Bianchi, L., Gambardella, L. M., and Dorigo, M. (2002). An ant colony optimization approach to the probabilistic traveling salesman problem. In Guervós, J. J. M., Adamidis, P., Beyer, H.-G., Schwefel, H.-P., and Fernández-Villacañas, J.-L., editors, *Parallel Problem Solving from Nature — PPSN VII*, pages 883–892, Berlin, Heidelberg. Springer Berlin Heidelberg.

Cesari, G. (1996). Divide and conquer strategies for parallel tsp heuristics. *Computers & operations research*, 23(7):681–694.

Chapman, B., Jost, G., and Van Der Pas, R. (2007). *Using OpenMP: portable shared memory parallel programming*. MIT press.

Cook, W. J., Applegate, D. L., Bixby, R. E., and Chvátal, V. (2011). *The traveling salesman problem: a computational study*. Princeton university press.

Djamegni, C. T. and Tchuenté, M. (1997). A cost-optimal pipeline algorithm for permutation generation in lexicographic order. *Journal of Parallel and Distributed Computing*, 44(2):153–159.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, San Francisco.

Gohil, A., Tayal, M., Sahu, T., and Sawalpurkar, V. (2022). Travelling Salesman Problem: Parallel Implementations & Analysis. DOI:10.48550/arXiv.2205.14352.

Grefenstette, J., Gopal, R., Rosmaita, B., and Van Gucht, D. (2014). Genetic algorithms for the traveling salesman problem. In *Proceedings of the first International Conference on Genetic Algorithms and their Applications*, pages 160–168. Psychology Press.

Johnson, D. S., Aragon, C. R., McGeoch, L. A., and Schevon, C. (1989). Optimization by simulated annealing: An experimental evaluation; part i, graph partitioning. *Operations research*, 37(6):865–892.

Johnson, D. S., Aragon, C. R., McGeoch, L. A., and Schevon, C. (1991). Optimization by simulated annealing: an experimental evaluation; part ii, graph coloring and number partitioning. *Operations research*, 39(3):378–406.

Johnson, S. M. (1963). Generation of permutations by adjacent transposition. *Mathematics of computation*, 17(83):282–285.

Laporte, G. (1992). The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(2):231–247.

Mavrovouniotis, M., Müller, F. M., and Yang, S. (2017). Ant colony optimization with local search for dynamic traveling salesman problems. *IEEE Transactions on Cybernetics*, 47(7):1743–1756.

Sahalot, A. and Shrimali, S. (2014). A comparative study of brute force method, nearest neighbour and greedy algorithms to solve the travelling salesman problem. *International Journal of Research in Engineering & Technology*, 2(6):59–72.

Sedgewick, R. (1977). Permutation generation methods. *ACM Computing Surveys (CSUR)*, 9(2):137–164.

Sun, X.-H. and Ni, L. M. (1990). Another view on parallel speedup. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 324–333.