Parallelizing VRPH: A Vehicle Routing Heuristics Library using OpenMP

Venkatesh Mahadevan, Tanzim Mokammel, Yasser Khan
Department of Electrical and Computer Engineering
University of Toronto, Toronto, Canada
{ venkatesh.mahadevan, tanzim.mokammel, yasserm.khan}@utoronto.ca

***Abstract*—This paper describes the general nature of VRPH and the various algorithms used for implementing the same. The test methodology and the test datasets used are also described, along with the results obtained. The paper concludes by drawing upon inferences obtained during the testing procedure and how they can be used for solving NP-complete problems similar to VRPH in various fields.**

***Keywords* — VRPH, Clarke-Wright, Sweep, datasets, speedup**.

I. INTRODUCTION

VRPH (Vehicle Routing Problem Heuristics) is a combinatorial optimization and integer programming problem. First introduced in 1959 by Dantzig and Ramser [1], and referred to as *'classical VRP'*, the problem attempts to find a minimal cost route using a given fleet of homogenous vehicles and a set of customer locations and their demands, while adhering to vehicle capacity and route length constraints. Such problems are very important in the fields of transportation, supply chain distribution, and logistics. Simple real life examples of the usage of variants of the VRP include: school bus route determination, garbage collection within a city, shipment delivery routes, and mailman delivery routes. Given the nature of VRPH, the following questions arise:

1) What is the approach taken when solving VRP?
2) What is the time complexity of these solutions?
3) What can we do to speed up the solution finding process?
4) If speed up via software is not possible, can we make use of hardware as well i.e achieve parallelism?

These are the questions that instigate us to probe further into the nature of VRP, and in finding new methodologies for solving the same, as the following sections will describe.

II. NEED FOR PARALLELIZATION

Although the VRP is easy to understand, it is very difficult to solve in practice. In the field of combinatorial optimization, it is an NP-complete problem. This indicates the problem cannot be solved in polynomial time, but if there is a solution, it can be verified within polynomial time [2]. Due to the difficulty in solving VRP, exact solutions are only possible through integer and dynamic programming methods. However, these can only be used for less than 100 customers, and cannot account for various modifications that must be made to the VRP. Therefore, in practical scenarios containing thousands of customers (or nodes), heuristics are required [3]. Heuristics produce a solution within a reasonable time frame and within acceptable levels of accuracy. However, heuristic algorithms often exhibit time complexities that are non-linear with respect to input size and in many cases, grow exponentially. Hence if the time taken to arrive at a solution using heuristic-based approaches can be optimized, we can hope to have reasonable solutions in finite time. Since the trend nowadays in the field of software is to adopt a parallel-based approach towards solving problems of such a nature as described above, we felt that doing so would benefit us in terms of finding solutions in polynomial time, and allow the scientific community to make use of VRPH in solving problems in various fields.

III. REASONS FOR CHOOSING OPENMP

For parallelizing VRPH, we chose OpenMP. OpenMP is an application programming interface (API), and supports shared memory multiprocessor programming in C, C++, and Fortran. It is an implementation of multithreading. With this method of multithreading, a master thread allocates slave threads at specified locations of the code. The specified code then runs concurrently according to the work sharing methodology, which may also be designed. This is called the fork-join model, and it executes serial parts of the code sequentially, and parallelized parts concurrently [4]. OpenMP allows for both task and data parallelism. Fig. 3.1 below shows the fork-join model which OpenMP uses.

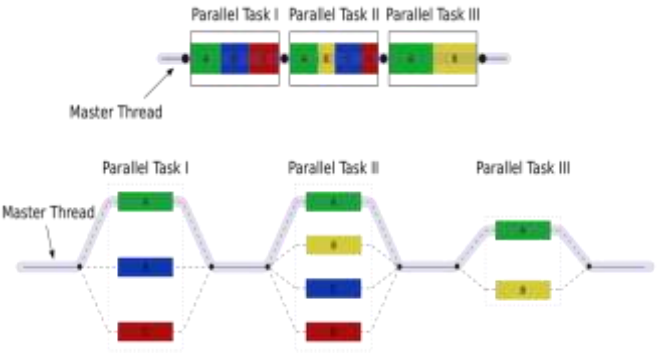


Fig 3.1: Fork-join model of OpenMP

OpenMP was considered suitable for parallelizing the VRPH library due to the following reasons:

1) OpenMP allows for easier development. With OpenMP, the developer does not need to worry about message passing, as it is done automatically.
2) OpenMP uses the pragma system, due to which no dramatic changes are made to the original code. This reduces the chances of bugs being introduced during the code development cycle.
3) The layout and decomposition of memory in OpenMP is handled automatically, allowing the programmer to focus solely on improving execution.

The following sections describe the algorithms used by VRPH, their subsequent parallelization using OpenMP and the performance enhancements obtained with the same.

IV. ALGORITHMS USED FOR VRPH

Since VRP is an NP-complete problem, sophisticated algorithms are used to generate good heuristics, so that the problem can be solved in polynomial time. Because the problem is "harder" than the TSP, exact methods are suitable for small instances only. The Integer Programming formulation can be based on:

1) partition of routes, or
2) construction of routes for vehicles (cf. TSP models), or
3) product flow

Many heuristics use similar principles as the TSP heuristics, keeping an eye on the capacity constraints. Most methods can be classified into the following categories:

• Constructive methods: tours are built up by adding nodes to partial tours or combining sub-tours, regarding capacities and costs.

• Two-phase methods: consisting of 1) clustering of vertices and 2) route construction. The order of these phases may be "clustering first, route later" or "route first, clustering later".

Two main algorithms which are often used are Clarke-Wright, which is an example method of the first type, and Sweep, which is an example method of the second type. Also, since multiple solutions are generated in VRP using the above two algorithms, a decision must be made to select the best possible solution out of all the proposed solutions. To aid in this, local search operators are often used for route optimizations and to help identify a unique and cost-effective solution. Examples of search operators include OnePointMove, TwoPointMove, ThreePointMove, and OrOpt, amongst others. We go on to describe the Clarke-Wright algorithm in the next section.

V. CLARKE-WRIGHT SAVINGS ALGORITHM

As stated in Sec. IV, the Clarke-Wright algorithm follows the principle of "clustering first, route later". For this algorithm, we proceed by assuming a complete undirected graph with symmetric costs and an unlimited number of vehicles with capacity K. The algorithm proceeds as follows:

1. Starting solution: each of the n vehicles serves one customer.

2. For all pairs of nodes i, j, i…j, calculate the savings for joining the cycles using edge [i, j]:

$$s_{ij} = c_{0i} + c_{0j} - c_{ij} \quad (5.1)$$

3. Sort the savings in decreasing order.

4. Take edge [i, j] from the top of the savings list. Join two separate cycles with edge [i, j], if

4.1) the nodes belong to separate cycles

4.2) the maximum capacity of the vehicle is not exceeded

4.3) i and j are first or last customer on their cycles.

5. Repeat 4 until the savings list is handled or the capacities don't allow more merging.

The algorithm iterates if i and j are NOT united in step 4 or if the nodes belong to the same cycle, OR if the capacity is exceeded, OR if either node is an interior node of the cycle. The algorithm thus tries to improve each route by applying a TSP heuristic to the same. A visual representation of how the Clarke-Wright algorithm works in the case of our VRP library is shown below:

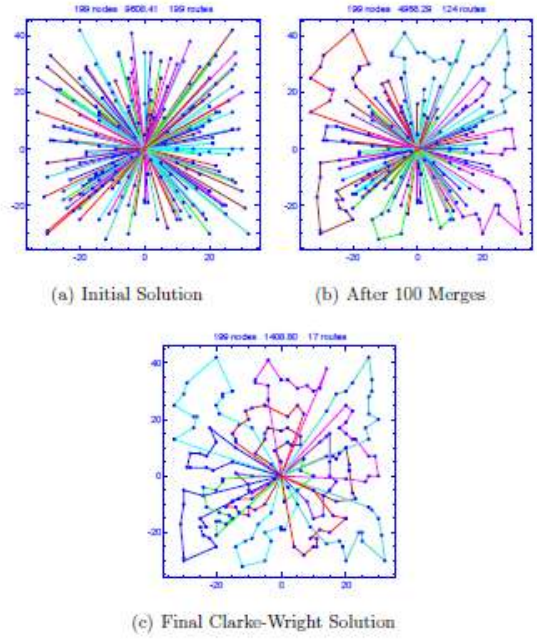


Fig 5.1. Clarke-Wright algorithm

Here, we see that in Fig 5.1 c), the final solution generated allows every vehicle to cover more nodes by combining routes before returning to the depot.

The next section describes the Sweep algorithm.

VI. SWEEP ALGORITHM

As stated in Sec. IV, the Sweep algorithm follows the principle of "route first, clustering later". The algorithm begins by assuming the customers are points in a 2-D plane with Euclidean distances as costs. The distance between (x_i, y_i) and (x_j, y_j) is:

$$c_{ij}^2 = (x_i - x_j)^2 + (y_i - y_j)^2 \quad (6.1)$$

The algorithm then proceeds as follows:

1) Compute the polar coordinates of each customer with respect to the depot. Sort the customers by increasing polar angle.

2) Add loads to the first vehicle from the top of the list as long as the capacity allows. Continue with the next vehicle until all customers are included. Now the customers have been clustered by vehicles.

3) For each vehicle, optimize its route by a suitable TSP method.

Graphically, in step 1 we rotate a ray centered at the depot. The starting angle can be chosen arbitrarily. A visual representation of how the Sweep algorithm works in the case of our VRP library is shown in Fig 6.1. Fig 6.1 c) shows that the final solution generated forces every vehicle to return back to the depot and then service the next node, which increases the runtime in comparison to Clarke-Wright.

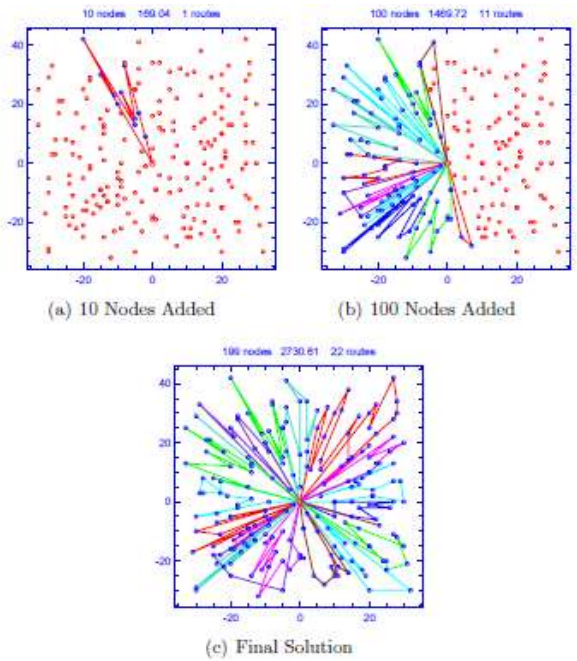
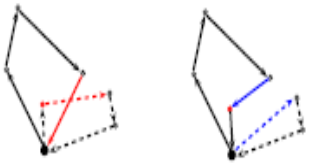


Fig 6.1. Sweep algorithm

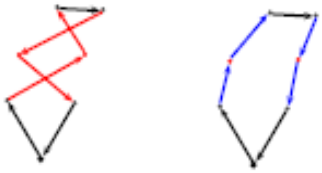
In order to understand how the above algorithms produce solutions to an NP complete problem in polynomial time, it is important to understand the optimization techniques used by them. The next section describes the local search operators used in the optimization techniques employed by the above algorithms, and how they differ from each other.

VII. LOCAL SEARCH OPERATORS

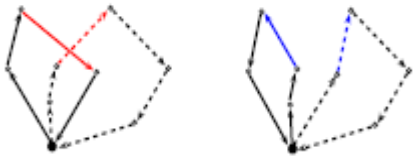
The above algorithms try to improve upon the initial solutions found by applying local search operators to improve the quality of the same. This is a commonly used technique in combinatorial optimization, and has proven to be very effective in VRP. In the VRP library, the local search operators perform solution modifications based on user defined criteria. For example, the user may define the library to only accept optimizations resulting from applying the operators, if it decreases the total route length. The most important optimization techniques are listed below:



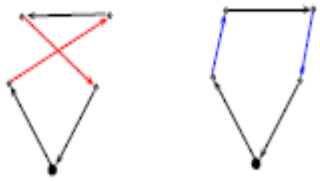
a) **One-Point Move**: Relocate an existing node into a new position.



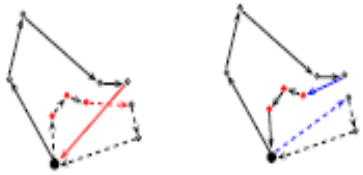
b) **Two-Point Move**: Swap the position of two nodes.



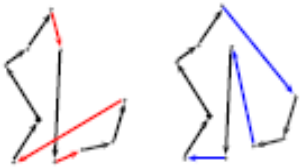
c) **Inter-Route Two-Opt Move**: Remove one edge from two different routes and replace them with two new edges.



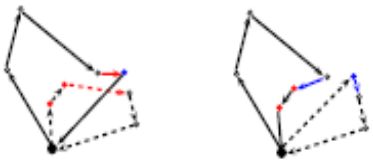
d) **Intra-Route Two-Opt Move**: Remove two edges from a single route and replace them with two new edges.



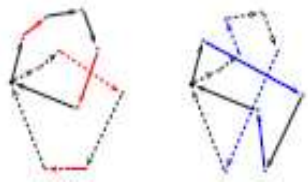
e) **Or-Opt Move**: Remove a string of two, three of four nodes and insert the string into a new position.



f) **Three-Opt Move**: Remove three edges from a route and replace them with three new edges.



g) **Three-Point Move**: Swap the position of a pair of adjacent nodes with the position of a third node.



h) **Cross-Exchange Move**: Remove four edges from two different routes and replace them with four new edges.

Fig 7.1. Local search operators used in VRP

The upcoming sections describe the parallelization methodology and its applications with reference to the VRPH API.

VIII. PARALLELIZATION METHODOLOGY

The VRPH API is written in C and C++ and proves to have great potential when it comes to applying parallelization. However, in order to ensure comparable or greater performance than the serial implementation, care must be taken in identifying the core regions which would benefit from parallelizing and studying the resulting effects on overall performance. To do this, we must first understand the input file format, and its relationship with the VRPH API.

IX. VRPH INPUT FILE FORMAT

The VRPH implementation takes a VRP file as input. A VRP file is a modification of the widely used TSPLIB format, which offers several methods of representing VRP problems. The VRPH input file includes a few modifications specific for *CVRPH* (capacitated *VRPH*). The input files contain field headers, which contain the name of the field, followed by the data pertinent to the problem type. An example TSPLIB file format is shown in Table 9.1.

TABLE 9.1 Input TSPLIB file format

Header	Description
Name	String used to describe the problem
Type	Must be *CVRP*
Dimension	Used to specify total number of nodes
Capacity	Used to specify vehicle capacity
Edge_Weight_Format	Used to describe the distance function used
Edge_Weight_Type	Used to determine the inter-node distance (*FUNCTION* in this project)
Node_Coord_Section	Used to define the section containing the node coordinates
Demand_Section	Used to define the section containing the demands of each node
Depot_Section	Used to specify the coordinates of the depot
EOF	Used to denote end of file

Table 9.1 shows the valid headers and sections of a TSPLIB file. The *Edge_Weight_Format* function supports five different edge weight calculations:
Euclidean distance: Straight line distance
Geographical distance: Node locations are latitude and longitude coordinates
Manhattan distance: Right angle distance.
Maximum distance: max(|x1 − x2|, |y1 − y2|) .
Rounded Euclidean distance: sqrt((x1 − x2)2 + (y1 −y2)).

These edge-weight calculating functions are used to create the distance matrix. The *DEMAND* section comprises multiple scenarios, such as demands spread over multiple days and demands to be satisfied with a certain time period. However, we do not test those features since our main focus is on capacitated VRP with the time period restricted to a single day.

The next section describes the VRPH API, their role in the algorithms described in Sec. V and Sec. VI, and possible speedups that could be obtained via parallelization.

X. VRPH API

As described in the previous sections, VRPH relies on a set of heuristics to predict feasible solutions. Therefore, if performance is to be improved, we need to find out potential bottlenecks and parallelize them without introducing any additional penalties. Hence our main focus is to find out the regions in the VRPH API that were either compute-intensive or I/O intensive and try to speed them up. For this purpose, we analyze our code using Valgrind [5], which is an open source instrumentation framework that can detect memory leaks and threading bugs. We use a tool in the Valgrind toolkit known as Callgrind [6], which can generate a dump of the program along with the time spent executing each portion.

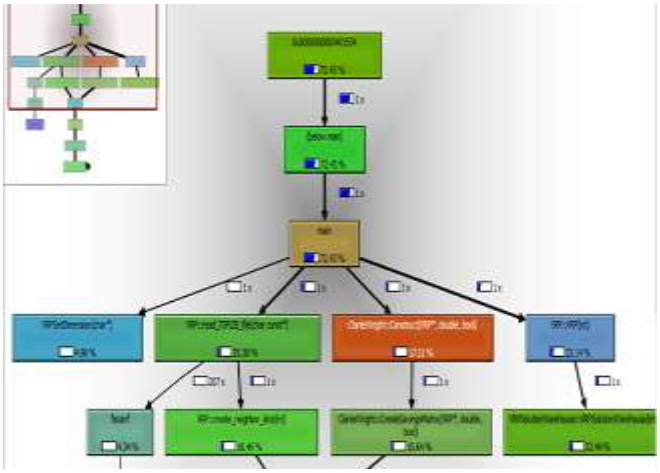


Fig 10.1 Example call graph produced by Callgrind and Kcachegrind

The dump file produced by Callgrind can be fed to Kcachegrind [7], which produces an effective visualization as shown in Fig 10.1. This helps isolate the overheads associated with linking to the OpenMP library and thread activity, and allows us to solely focus on improving the execution time of the functions at the first level below main. The following sections go on to describe each of the above routines, the performance enhancements that were implemented, along with the speedup achieved. Note that the speedup was measured using both the Clarke-Wright and Sweep implementations, the results of which will be discussed in Sec. XIV.

A. VRPGetDimension(char):*

This function is an example of an I/O intensive function**.** It reads in the input file (which is in TSPLIB format) and tries to scan for the string '*DIMENSION*' (which indicates the size of the problem). It also tries to find the keyword *'EOF*' which would indicate the end of the input dataset. It then returns the list of VRPH nodes present in the input file, which is later used to create the optimal route. The time taken to process

the TSPLIB file format can vary depending on the size of the dataset, and slows down the serial implementation. Hence, we choose to parallelize the initialization of the array containing the characters for the '*DIMENSION*' string separately, which would later be used for returning the number of VRPH nodes. The results of parallelization and the speedup obtained are shown in the Figure 10.2.

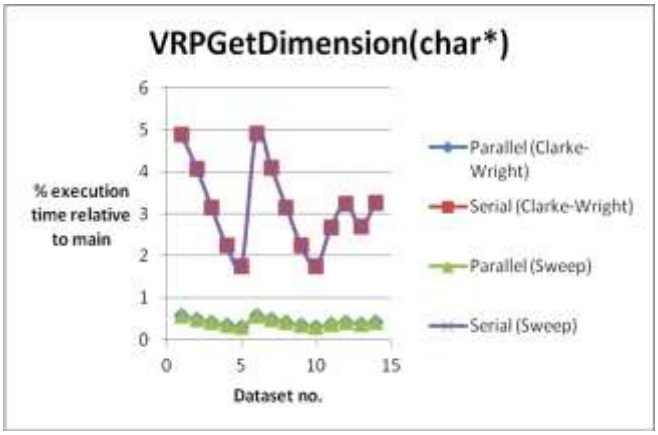


Fig 10.2. Speedup of VRPGetDimension (using Clarke-Wright and Sweep)

Here, we see that the serial implementations of both Clarke-Wright and Sweep have the same execution time percentages; the same goes for the parallel implementations. Furthermore, we notice a speedup factor of 10 in the relative execution times when we parallelize the serial implementation. Since this is the first function that is called before computation, it is evident that the speedup obtained by parallelizing this function would have a considerable benefit in terms of reducing overall computation time, barring any thread-based overheads.

B. VRP::VRP(int):

This function is an example of an I/O and compute intensive function**.** It serves as the constructor for the nodes specified in the '*DIMENSION*' field above. The current solution at any stage of the procedure is stored in a doubly linked list containing two arrays of length '*DIMENSION*', *next_array* and *pred_array*. Along with these 2 arrays, several other properties are kept track of, such as the total route length and the number of routes. The solution warehouse is then created using *VRPSolutionWarehouse*. The routes and the associated heuristics are then stored using *VRPTabuList*. This information is used later on when new routes are generated, in order to help choose the best route subject to certain constraints. Out of all the possible solutions generated, only the best 50 are stored in *VRPTabuList*. Here the maximum overhead is incurred in the initializations of the *fixed* and *routed* arrays, and while updating the hash tables (*this->hash_vals1* and *this->hash_vals2*) with the values for 2 different seeds, namely *SALT1* and *SALT2*. The results of the speedup obtained by parallelizing these bottlenecks in comparison with serial is shown in Fig 10.3. Here we observe that the serial implementations follow a similar trend and the same applies to the parallel version. The serial versions also seem to match the parallel performance for datasets 5 and 10 before diverging. This is because the dimension and demand section for both datasets is the same. Since the demand section determines the nature of the routes taken, the serial versions display similar behaviour for both datasets and so do the parallel versions. The serial versions also have the same execution time as parallel for these datasets since there is a lot of divergence in the parallel case. This is because only 1 thread is needed to perform all the work, making the parallel performance match the serial performance. The speedup factor is thus seen to be equal to 9 in the case of Sweep and 12 in the case of Clarke-Wright.

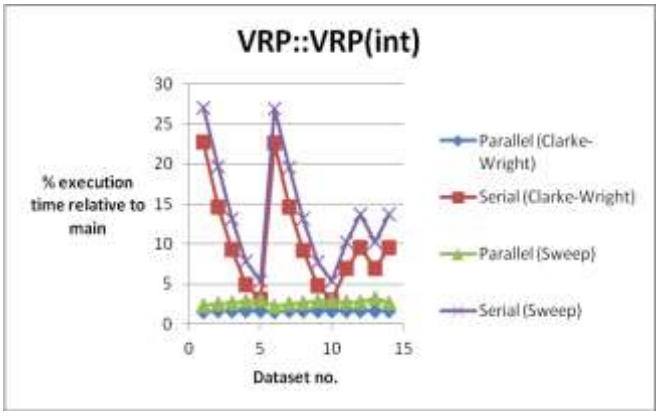


Fig 10.3. Speedup of VRP::VRP (using Clarke-Wright and Sweep)

C. ClarkeWright::Construct(VRP, double, bool):*

The role of this function is to construct the routes via the Clarke-Wright savings algorithm with the parameter *lambda*. It starts by creating the initial set of routes for all non-*VRPH DEPOT* nodes (i.e nodes that are not at the origin). In order to avoid redundancy in routing, it checks to see if a node is already routed in the TSPLIB file and if so, assumes that further computation will be performed only on the set of nodes that were already routed. If the above condition is not satisfied, it means the node can be used for routing. Further checking is performed to see if it is in the set of nodes that can be serviced by the default set of routes.

The routes are then merged under 4 conditions:

1) Both the source i and destination j are marked *UNUSED*.
2) The source i has been marked *VRPH_ADDED* and the destination j has been marked *VRPH_UNUSED*.
3) The source i has been marked *VRPH_UNUSED* and the destination j has been marked *VRPH_ADDED*.
4) Both the source i and destination j are marked as *VRPH_ADDED*.

A check is also performed to see if the source and/or destination lie on a route that passes through them. If so, their status is changed to *VRPH_INTERIOR* before routing occurs. After routing the entire set of nodes, the total route length is recorded and the intra-cluster and inter-cluster route numbers are normalized relative to the *VRPH_DEPOT*.

The main causes of lack of speedup in the function were in checking if a node was previously routed, and in the merging of the routes under the 4 conditions described previously. The former was parallelized using OpenMP pragmas, while the logic in the latter was modified to avoid route reversals wherever possible when trying to find the source and destination nodes for all routes. The results of the applied parallelization are shown in Fig. 10.4.

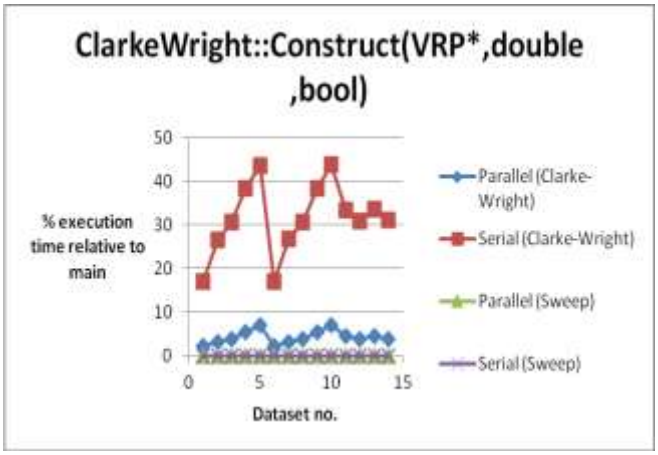


Fig 10.4. Speedup of ClarkeWright::Construct (using Clarke-Wright and Sweep)

Here, we notice that parallelizing Construct provides a speedup factor of almost 5 in the case wherein the Clark-Wright implementation is actually used. In the case of Sweep, a separate constructor is used that is seen to have minimal overhead in all cases, and therefore has not been considered for analysis. Therefore, the execution time is seen to be 0 for both the parallel and serial implementations of ClarkWright::Construct in the case of the Sweep algorithm.

D. VRP::read_TSPLIB_file(char const):*

This function processes each section of the TSPLIB file provided by the user, and records the relevant data in the VRP structure. It begins by checking if the list of strings in the TSPLIB file are supported, and if so, begins to classify them into various categories, explained in Sec. 9.

The function then begins by normalizing all nodes to place *VRPH_DEPOT* at the origin. Next, it attempts to place all the nodes using polar co-ordinates (in case of Sweep) or Cartesian co-ordinates (in case of Clarke-Wright), and calculates their neighbor lists as well. The service times and capacities for multiple days worth of demands are also stored in case the type of problem encountered in the TSPLIB file is a *CVRP* (capacitated VRP), in which case the heuristics need to be calculated over multiple days. In this function, an attempt was made to remove logical redundancy in the case statements when a properly formatted TSPLIB file was encountered. To ensure that thread divergence did not occur, dummy rows or columns were added in the case of VRPH_DEPOT. The results of the above modifications are shown in the Fig 10.5. Here, we see that the serial implementation of Sweep performs worse than the same for Clarke-Wright. This is because Sweep needs to read in the TSPLIB file and then create a sorted list of nodes and their respective theta's to '*sweep*' through before constructing the routes. These are not needed when using Clarke-Wright. We also see that the parallel implementation is faster than serial in both cases, since the processing of the co-ordinates and neighbor list generation can be done separately by removing any dependencies between nodes. The speedup factor that we get by parallelizing is thus seen to be close to 10 in the case of Clarke-Wright and 14 in case of Sweep.

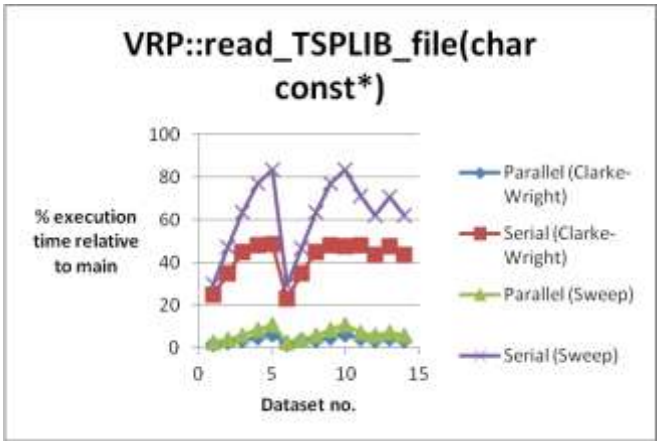


Fig 10.5. Speedup of VRP::read_TSPLIB_file (using Clarke-Wright and Sweep)

The following sections describe the test platform, the datasets used for testing and the subsequent results obtained along with a detailed analysis of the same.

XI. TEST PLATFORM

The tests mentioned above were performed on an AMD FX 8 core processor. There are 2 hardware threads per core, allowing 16 hardware threads to be available at any time. The OS used was Ubuntu version 14.04, while the compiler used was GCC 4.8. This version of GCC includes OpenMP 3.1, which is sufficient for the parallelization we intended. PLplot, a cross-platform open source scientific plot generator was used to generate the routing graphs. The output file produced by VRPH can be provided as input to PLplot to produce a graphical representation. of the solution space.

XII. DATASETS USED FOR TESTING

To accurately test the difference between the serial and parallel implementations, we use large and small versions of four different real world dataset types. These dataset types are Christofides (50-100 nodes), Golden (100-250 nodes), Taillard (400-580 nodes), and Li (780-1200 nodes). These eight datasets are chosen to effectively tax different parts of the code, ensuring accurate results for each portion of the VRP implementation.

Fig 12.1 shows the Clarke Wright parallel solutions for the first dataset type used, called Christofides. Christofides has the fewest number of nodes amongst all datasets. Also, the nodes are evenly spaced out, so that most neighborhoods are equally sized.

The second dataset type is Taillard, shown in Fig 12.2. Both the small and large versions of this dataset contain nodes spaced very closely together, with several nodes on the outer limits. The Clarke Wright parallel solution for the large Taillard dataset contains many route intersections, which pose a problem for the Sweep algorithm, since the latter works on the principle of how far and at what angle a node is from VRPH_DEPOT and does not handle inter-node routing for non VRPH-DEPOT type nodes

Fig. 12.3 shows the third dataset type, Golden. It contains more nodes than in the previous two types. Golden also features a distinctive wheel spoke pattern for the routes. Creating these routes is a challenge for Clarke-Wright and Sweep due to the nature of their initial solution gathering, which doesn't lend itself well to these types of routes.

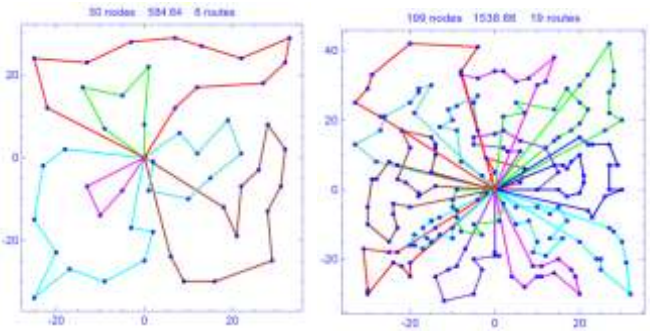


Fig 12.1. Parallel (small) vs Parallel (large) solutions for Christofides using Clarke-Wright

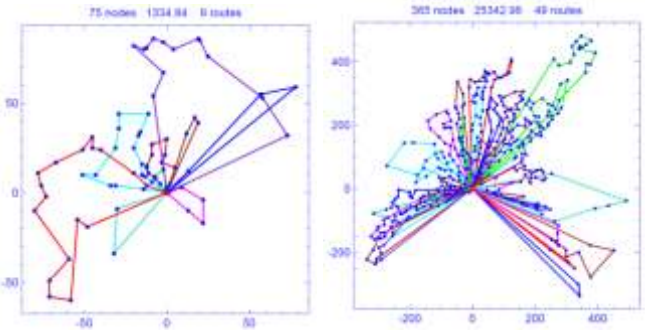


Fig 12.2. Parallel (small) vs Parallel (large) solutions for Taillard using Clarke-Wright

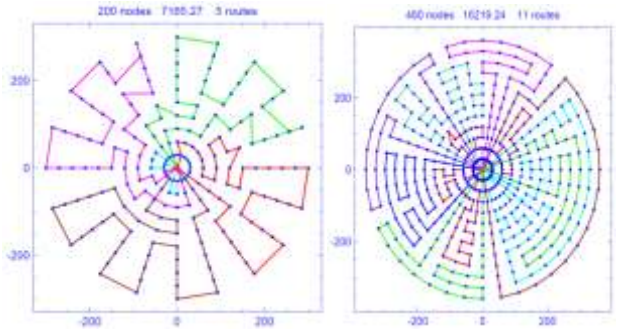


Fig 12.3. Parallel (small) vs Parallel (large) solutions for Golden using Clarke-Wright

The final dataset type is called Li, and is shown in Fig 12.4. This type contains the most number of nodes out of all our datasets. The pattern is very similar to Golden, so all comments for Golden apply here as well. Since there are many more nodes in Li than Golden, they are spaced very close together.

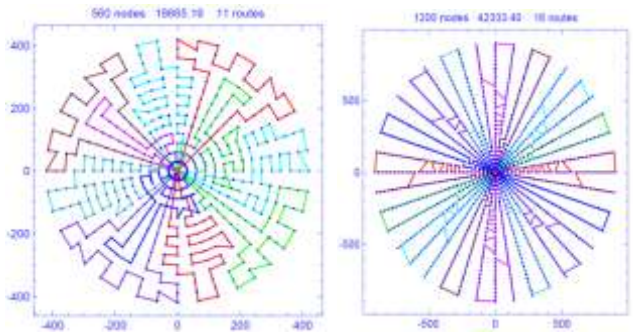


Fig 12.4. Parallel (small) vs Parallel (large) solutions for Li using Clarke-Wright

XIII. RESULTS OBTAINED VIA TESTING

To measure our success in parallelizing VRPH, we test each of the datasets described previously with the parallel and serial VRPH implementations. The similarity of the solutions can be verified by determining the delta in the number of routes and route distances between the serial and parallel implementations. In Fig 13.1, we see that the number of routes between the serial and parallel implementations for Clarke-Wright does not change, implying that the solution is unchanged in both versions.

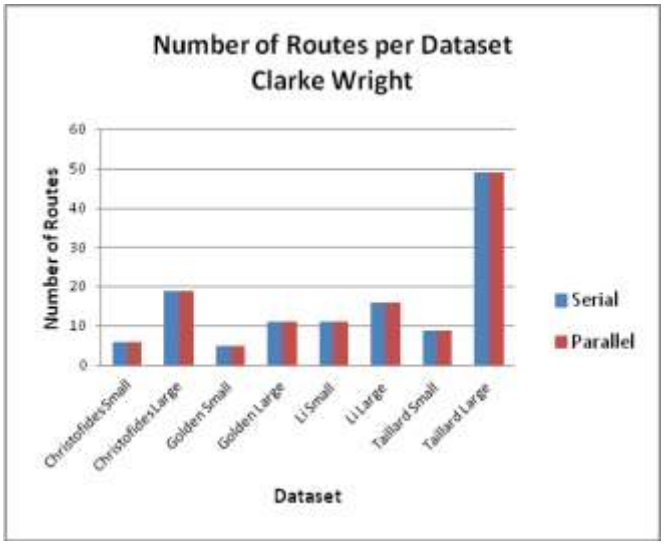


Fig 13.1. Number of routes generated by Clarke-Wright for each dataset.

Fig. 13.2 further proves the preceding observation, since both number of routes and route length do not change

Sweep has a different outcome than Clarke Wright with respect to the solution as shown in Fig 13.3. The parallel implementation appears to exhibit a few more routes in comparison with the serial version, signifying a change in the solution space.

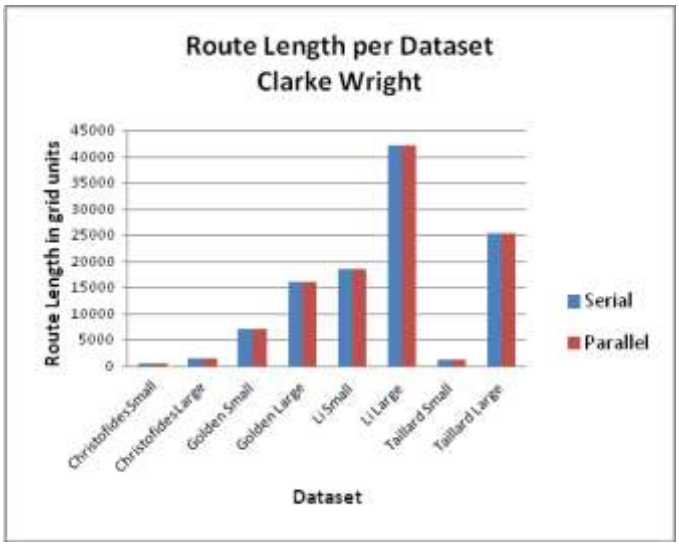


Fig 13.2. Route length generated by Clarke-Wright for each dataset.

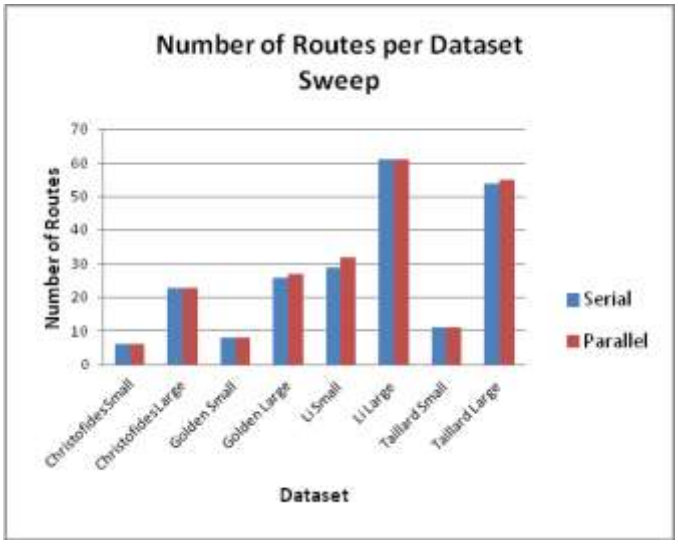


Fig 13.3. Number of routes generated by Sweep for each dataset.

Fig. 13.4 below confirms that any changes in the solution (as shown by Fig. 13.3) are minimal since the route length has not been affected significantly. The large Golden dataset has a visible change in route length but other datasets look extremely similar. The previous four figures prove that our parallelization has not affected the validity of the logic but has introduced a small perturbation in the solution space.

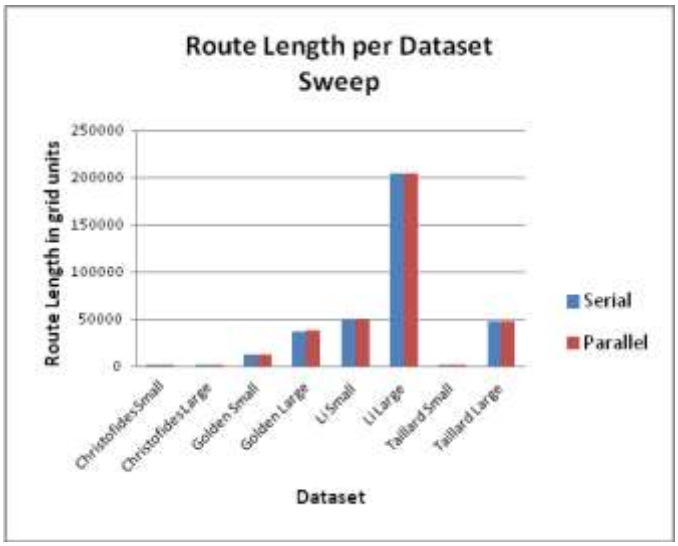


Fig 13.4. Route length generated by Sweep for each dataset.

XIV. SPEEDUP MEASUREMENTS

This section presents the speedup accomplished by parallelization. Using Callgrind, the most time consuming functions are graphed for every dataset type. Percentages are used instead of absolute times to determine speedup for two reasons. First, more hardware resources are needed to effectively use the parallelism we have programmed. Therefore using absolute times in this case does not result in an accurate comparison.. Secondly, this approach allows us to ignore the time spent in initialization and only focus on the speedup of the logic.

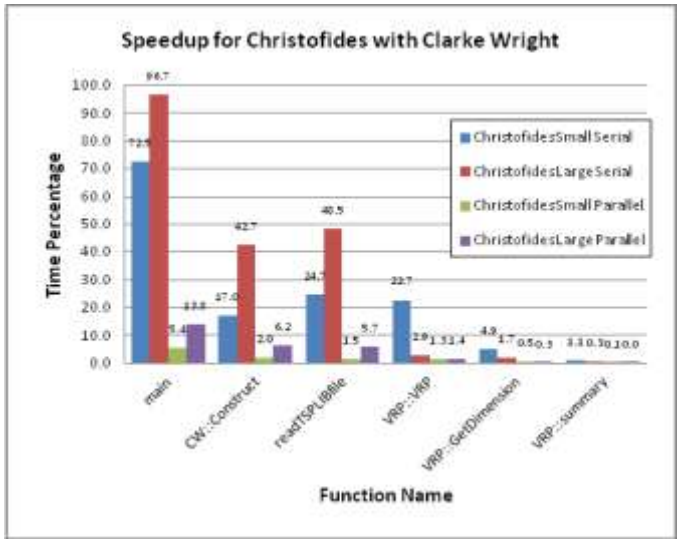


Fig 14.1. Function times for Christofides using Clarke- Wright.

Figure 14.1 shows us the major effect of parallelization. The serial results show that as the data size increases, more time is spent in main. That is also true for the parallel results, but even on the large Christofides dataset the amount of time in main is almost $1/8^{th}$ of serial for large datasets. The rest of the functions (called within main) show similar results; there is a higher speedup in the parallel version for functions that spend more time in the serial version.

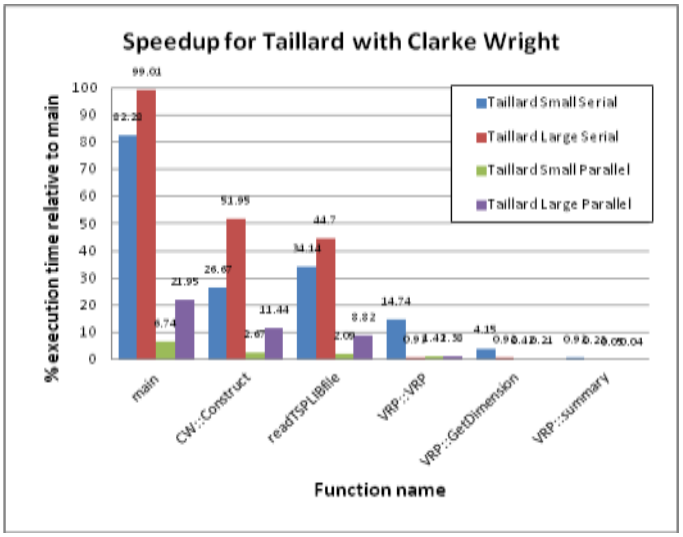


Fig 14.2. Function times for Taillard using Clarke-Wright.

Speedup results for Taillard using Clarke Wright are shown in Fig. 14.2. The dataset is larger than Christofides, and more complex in design so as the dataset gets larger more time is spent in main for both serial and parallel. The speedup for main now is only 5x for the largest dataset. Other functions exhibit similar behaviour.

Figure 14.3 shows us that Golden spends almost the same time as Taillard in main, even though it uses a larger dataset. This shows that the dataset complexity has an effect on execution time. Since Golden is simpler to route, 5x speedup is seen, similar to Taillard.

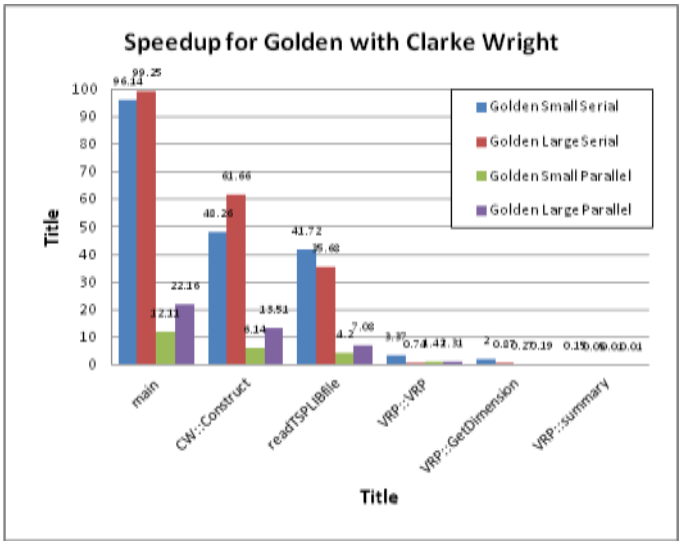


Fig 14.3. Function times for Golden using Clarke-Wright.

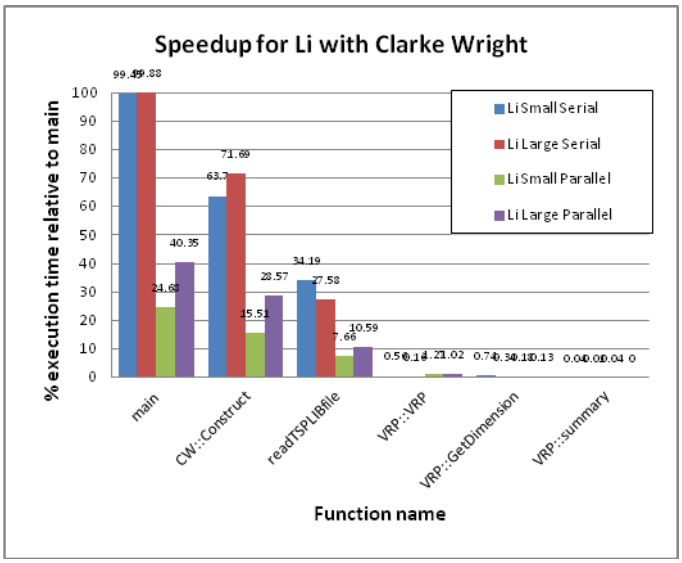


Fig 14.4. Function times for Li using Clarke-Wright

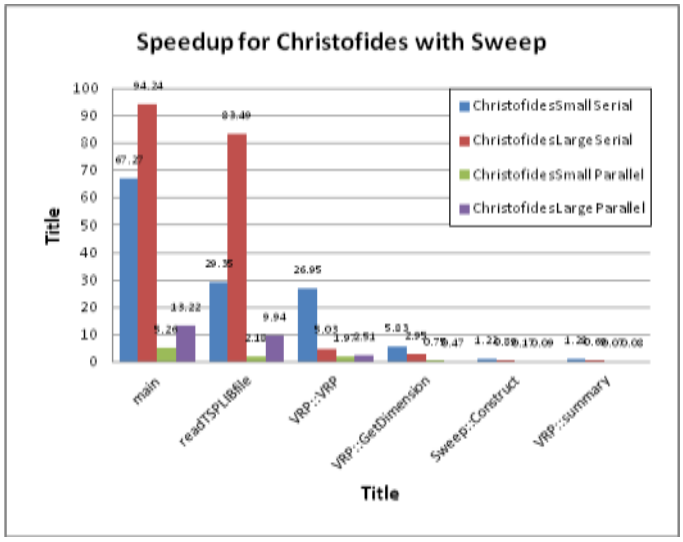


Fig 14.5. Function times for Christofides using Sweep

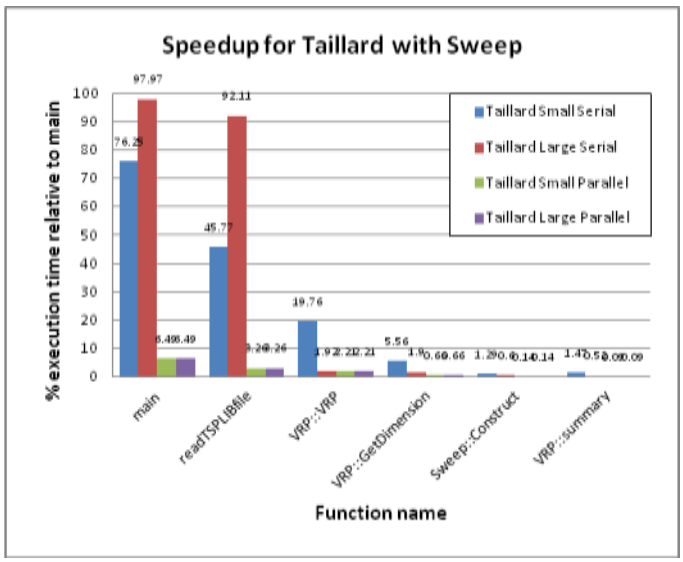


Fig 14.6. Function times for Taillard using Sweep

Li has the biggest dataset, and the effect it has on parallelism is seen in Fig. 14.4. More computation per thread means longer time to run, and on the large Li dataset there is only 4x speedup in main, compared to the 5x for Taillard. Specific functions fare better: reading the TSPLIB file is 3x faster, and the same applies for Construct (where the actual solution is calculated).

Sweep is a very different algorithm than Clarke Wright, and the effect of parallelism is more pronounced for smaller datasets. The bulk of time spent in Sweep is in reading the TSPLIB file, as seen in Fig. 14.5. It is trivial to parallelize file reading, and as such we can achieve great speedups, over 9x in the readTSPLIBfile function alone. Along with the other functions, main is almost 15x faster in the parallel case.

Taillard has a larger dataset than Christofides, so we see in Fig. 14.6 that reading the TSPLIB file serially takes even more time, almost all the time of main for the large dataset. The effect of parallelism is very pronounced here, almost a 30x speedup for reading the TSPLIB file.

Fig. 14.7 shows that when the dataset size for Sweep grows, the effects of parallelism become less visible. Here, reading the TSPLIB file in parallel is taking more time than in serial, resulting in a slowdown of 10x in comparison to Taillard for large datasets.

We finish with the largest dataset, Li, shown in Fig. 14.8. Sweep still uses the majority of the main time in reading the TSPLIB file, and we can now confirm that increasing the dataset size does indeed cause slowdown effects. It is now only 5x, down from the 30x speedup in Taillard. We can extrapolate that the maximum speedup for Sweep can only be achieved with a certain dataset size, beyond which the performance drops sharply.

We thus surmise that while parallelizing does indeed improve performance in the case of VRP, there is an inflection point beyond which performance is saturated and no further gains are achievable. We conclude by summarizing all that we have learnt, and on how we can develop on it, in the next section.

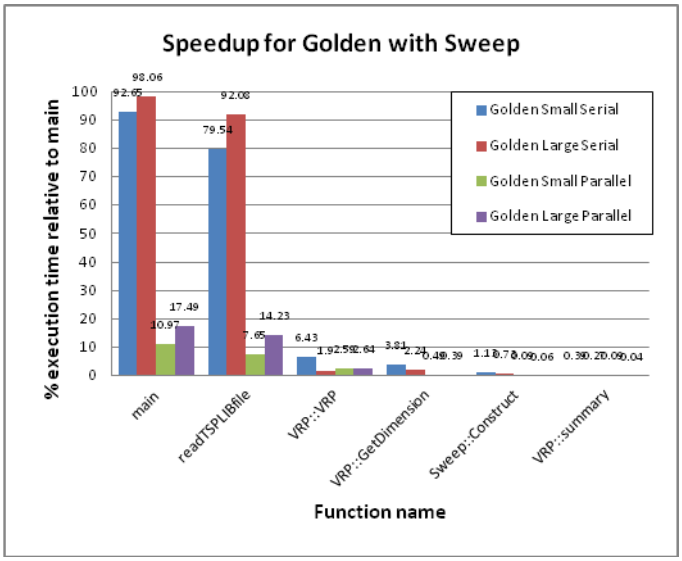


Fig 14.7. Function times for Golden using Sweep

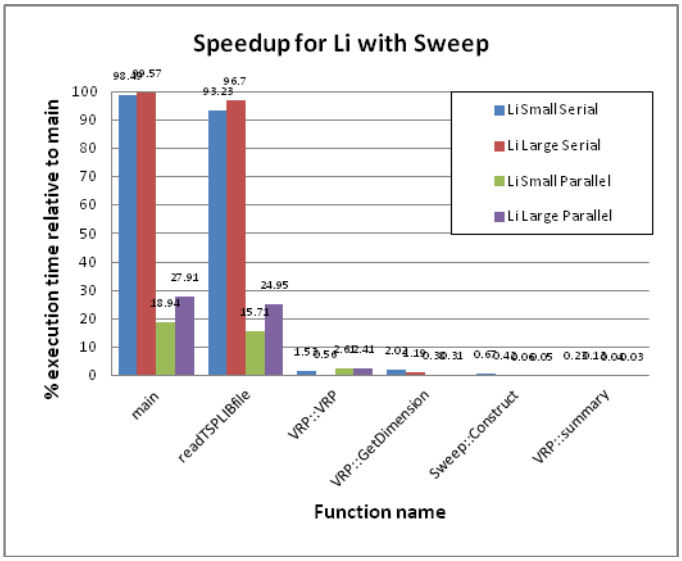


Fig 14.8. Function times for Li using Sweep

XV. CONCLUSION

Through the course of this paper, we hope that the reader has secured an insight on the intricacies involving VRPH and its implementation. The need to parallelize VRP was touched upon in detail, along with the reasons chosen for doing the same with OpenMP. The Clarke-Wright and Sweep algorithms, which are the main ones used in classical VRP, were elaborated upon, along with the local search operators used to perform route optimization.

The input file format was also described in detail, along with the relationship it bore to the VRPH API. The datasets and the platform used for testing were then elucidated, with a graphical representation of the effects that resulted from parallelizing the same. We learnt that improving the performance of a serial program via parallelization was not a trivial task, and careful analysis and consideration was needed to ensure comparable or greater performance. It is hoped that our work would prove to be a useful reference to those who wish to study and improve the performance of NP-complete problems, and for those who wish to apply aspects of VRPH to other fields.

XVI. ACKNOWLEDGEMENTS

We would like to thank Chris Groer (cgroer@gmail.com) for graciously allowing us to use his VRPH library written in C and C++ for parallelization. This was developed as part of his Phd thesis at the University of Maryland in 2008. Further information regarding the same can be found in Reference [1] under the References section. We would also like to thank Professor Amza for allowing us to take on this project. The parallel version of the source code for this project can be found under https://github.com/venkatesh20/VRPH, which is a public repository on github.

XVII. REFERENCES

[1] Classical VRP: http://drum.lib.umd.edu/bitstream /1903/9011/1/ Groer_umd_0117E_10068.pdf
[2] NP-complete: http://en.wikipedia.org/wiki/NP-hard
[3] Vehicle routing: http://en.wikipedia.org/wiki/ Vehicle_routing_problem
[4] OpenMP: http://en.wikipedia.org/wiki/OpenMP
[5] Valgrind: http://valgrind.org/
[6] Callgrind: http://valgrind.org/docs/manual/cl-manual.html
[7] Kcachegrind: http://kcachegrind.sourceforge.net/html/ Home.html