A PARALLEL BRANCH AND BOUND ALGORITHM

FOR THE SEQUENTIAL ORDERING PROBLEM

A Project

Presented to the faculty of the Department of Computer Science

California State University, Sacramento

Submitted in partial satisfaction of
the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

by

Radoyeh Shojaei

SPRING
2019

A PARALLEL BRANCH AND BOUND ALGORITHM

FOR THE SEQUENTIAL ORDERING PROBLEM

A Project

by

Radoyeh Shojaei

Approved by:

_____, Committee Chair
Dr. Ghassan Shobaki

_____, Second reader
Dr. Pinar Muyan-Ozcelik

_____
Date

Student: Radoyeh Shojaei

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the project.

_____, Graduate Coordinator _____
Dr. Jinsong Ouyang Date

Department of Computer Science

Abstract

of

A PARALLEL BRANCH AND BOUND ALGORITHM

FOR THE SEQUENTIAL ORDERING PROBLEM

by

Radoyeh Shojaei

The Sequential Ordering Problem (SOP) is a combinatorial optimization problem. Given a directed weighted graph and an unweighted directed graph representing precedence constraints among vertices, find a minimum-cost Hamiltonian path that satisfies the precedence constraints. Previous work on this problem included heuristic solutions and sequential optimal algorithms. To the best of our knowledge, there is no parallel optimal algorithm for the SOP. In this work, we propose a parallel optimal algorithm for the SOP using a branch-and-bound approach. Our current experimental results using 116 standard benchmark instances show that with a 2-hour time limit, the proposed parallel algorithm can solve 75 instances compared to 66 instances solved by the existing sequential algorithm. The average speedup across all solved instances is 2.05 for 4 threads. The best speedup with 4 threads is over 21.

_____, Committee Chair
Dr. Ghassan Shobaki

_____
Date

ACKNOWLEDGMENTS

I would like to thank Dr. Ghassan Shobaki, Dr. Pinar Muyan-Ozcelik and the CSUS Computer Science Department for providing the advice and resources needed to complete this project.

TABLE OF CONTENTS

Page

LIST OF TABLES

Tables Page

LIST OF FIGURES

Figures Page

Chapter 1: Introduction

The Sequential Ordering Problem (SOP) is a generalization of the Asymmetric Traveling Salesman Problem (TSP) [1]. The SOP, like the TSP, can be thought of as the problem of finding a path that visits a given set of cities. Given a pair-wise cost between cities, find a minimum-cost path that visits each city exactly once. The SOP adds precedence constraints to the problem, where certain cities must be visited prior to other cities.

There are several real-world applications that can be modeled as SOPs. In operation and production decisions, the scheduling of tasks in a manufacturing system [2], [3] or painting vehicles in an automotive paint shop [4] where the cost of switching tasks or paint is minimized. In transportation and routing, a helicopter visiting offshore oil platforms minimizes the total distanced travelled while satisfying the scheduling constraints of the platforms to visit [5]. In compiler instruction scheduling, minimizing the switching cost for a stream of instructions may be modeled as a SOP [6].

Like the TSP, the SOP is also NP-hard [3], [6]. Initial work focused on heuristic solutions to the SOP [1], [5]. In later work, exact algorithms using branch-and-bound (B&B) [6], [7] and integer linear programming (ILP) [3] have been proposed. However, to the best of our knowledge, no parallel exact algorithm has been proposed for the SOP. In this paper, we describe a parallel exact algorithm for solving the SOP based on the sequential B&B algorithm originally proposed by Shobaki and Jamal [6] and later enhanced by Jamal et al. [7]. Unlike the sequential algorithm that is based on a pure

depth-first search (DFS) of the solution space (represented by a tree), the proposed parallel algorithm uses a combination of breadth-first search (BFS) and DFS. It uses BFS to create multiple threads that simultaneously explore multiple sub-regions in the solution space. Dynamically generated subtrees of the solution space are divided among the created threads. Within each thread, DFS is used to search the sub-region that is assigned to that thread. The proposed algorithm is run on a multi-core processor, and the number of threads created does not exceed the number of cores to ensure that each thread is run on a separate physical core.

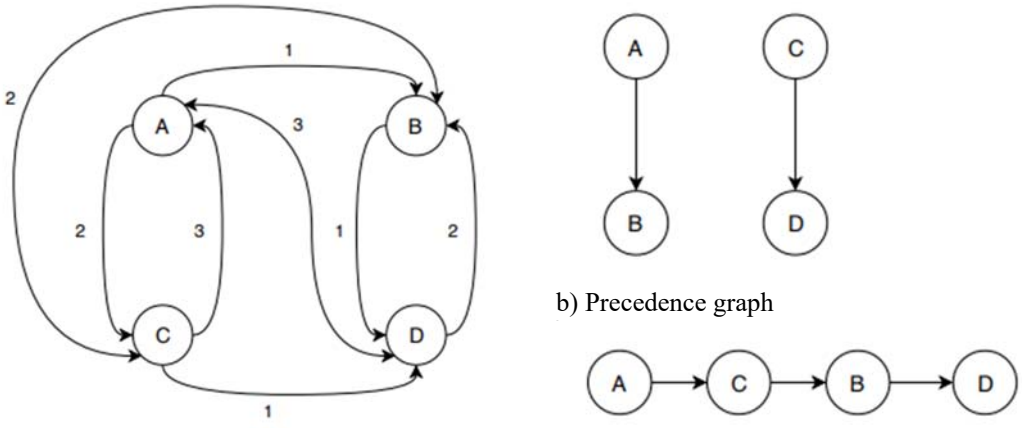
One motivation for parallelizing B&B algorithms is the possibility of super-linear speedups [8], meaning that the speedup could be greater than the number of parallel threads used during execution. Super-linear speedups may occur if the parallel algorithm finds good solutions earlier than the sequential algorithm. The result is that the parallel algorithm searches a smaller solution space relative to the sequential algorithm

The proposed parallel B&B algorithm is evaluated experimentally relative to the sequential B&B algorithm on three benchmark suites instances including TSPLIB [9], SOPLIB [3] and Compilers [6]. The results of this experimental evaluation on a quad-core processor with a time limit of 2 hours per instance show that the proposed parallel algorithm solves 9 instances that are not solved by the sequential algorithm. Super-linear speedups by up to 21x are seen on some instances. The average speedup across all solved instances is 2.05.

In subsequent chapters we present the following. In Chapter 2, we give a formal description of the SOP. In Chapter 3, we give a summary of previous work on the SOP and parallel B&B for problems related to the SOP. Chapter 4 describes the sequential B&B algorithm, and Chapter 5 describes the proposed parallel B&B algorithm. In Chapter 6, we present and analyze our experimental results. Finally, we give the conclusions and future work in Chapter 7.

Chapter 2: Formal Problem Description

The SOP is given by a weighted directed Graph G = (V, E) and a precedence Graph P (V, R), where V is a set of vertices, E is a set of weighted edges representing costs and R is a set of directed unweighted edges representing precedence constraints. The objective is to find a minimum-cost Hamiltonian path in G that satisfies the precedence relations in P. A Hamiltonian path in a graph is a path that visits each vertex in the graph exactly once. An example of an SOP instance with a feasible solution is given in Figure 1. In this example the cost graph in Figure 1c is shown with weighted directed edges. Symmetric edges are shown as bidirectional to simplify the image. The precedence relations in Figure 1b require the source vertex to precede the destination vertex. In Figure 1b, vertex A to be visited before vertex B and vertex C to be visited before vertex D. A possible solution is given by the path visiting A, C, B and D with a total cost of 5 in Figure 1c.



a) Cost graph

b) Precedence graph

c) Feasible solution

Figure 1: An instance of the SOP

Chapter 3: Previous Work

The literature review will focus on two aspects related to this report. First is the previous work on the SOP using heuristic and optimal algorithms, including B&B and dynamic programming. Second is the review of parallel B&B for other combinatorial optimization problems.

3.1 Algorithms for the SOP

The SOP was first described by Escudero and was solved with a heuristic method. Precedence constraints are first ignored which reduces the SOP to the asymmetric-TSP. If this solution does not satisfy precedence constraints, the path is modified to produce a valid solution [1]. Other heuristic solutions include ant colony systems [10], [11] and a parallel rollout algorithm [12]. More recently, Helsgaun adapted the Lin-Kernighan-Helsgaun (LKH) heuristic used for the TSP to solve related problems including the SOP. Helsgaun found the LKH heuristic to find optimal or best-known solutions on a high percentage of benchmark instances and even improved the best-known solution for some problems [13].

Ascheuer et al. were the first to create an optimal algorithm to the SOP based on ILP. They were able to solve problem instances up to 200 vertices from TSPLIB [14]. Mojana et al. [3] solved the SOP by recursively splitting the problem into smaller instances and combining the sub-problem solutions, referred to as decomposition. The sub-problems are solved with an ILP solver. Their algorithm closed an additional 8 open SOPLIB instances [3].

Shobaki and Jamal solved the SOP using a B&B approach for energy switching cost minimization in compiler instruction scheduling [6]. Three sets of lower bounds are used, two edge-based bound and a minimum spanning tree bound. For an instance of *n* vertices, the solution will require *n*-1 edges. Each vertex other than the first will have an incoming edge and each vertex other than the last will have an outgoing edge. The minimum incoming edge bound is calculated by taking the minimum cost incoming edges for each vertex and removing the maximum cost edge selected. The outgoing edge is calculated the same way, yielding the second lower bound. A minimum spanning tree (MST) will connect each vertex in the graph with *n*-1 edges. The MST is computed by creating an undirected graph from the directed graph and using Prim's algorithm to find a MST. With these three upper bounds, the largest lower bound is used to prune the solution tree [6].

In addition to using lower bounds, Shobaki and Jamal use a dominance-based pruning technique [6]. In this technique, information about previously visited nodes in the solution space are stored in a hash table and then used to either prune the current node or construct a better solution without having to explore the subtree below the current node. The current node may be pruned if it is dominated by a previously visited node. If the current node dominates the previously visited node, the current node replaces the previous entry in the hash table.

Papapanagiotou et al. experimentally compare the decomposition-based algorithm from Mojana et al. [3] and the B&B algorithm from Shobaki and Jamal [6] on three sets of SOP instances, SOPLIB, Compilers and TSPLIB [15]. Both algorithms were able to

solve instances the other algorithm could not. Overall, they found that the edge-based B&B approach tended to solve instances fairly quickly or would not improve the starting solution very much within the 48-hour time limit [15].

Jamal et al. extended their previous work [6], by adding a minimum-cost perfect matching (MCPM) lower bound [7] to the B&B algorithm. The SOP constraints are relaxed to reduce it to the assignment problem. The MCPM lower bound is an enhancement of the edge-based lower bounds introduced by Shobaki and Jamal [6]. Instead of finding a minimum-weight incoming/outgoing edge for each vertex, a minimum-cost set of edges is found by relaxing the SOP as a MCPM problem. The MCPM can be solved using the Hungarian algorithm [16], [17] and an existing solution can be updated efficiently using the dynamic Hungarian algorithm [18]. Jamal et al. use the dynamic Hungarian algorithm to incrementally compute MCPM-based lower bounds during a B&B search. Compared to their previous work, Jamal et al. solve five open instances in SOPLIB and achieve significant speedups on many previously solved instances [7].

The most recent work on the SOP is the work of Salii [19], who implemented a dynamic programming algorithm for the SOP and was able to close an open instance in TSPLIB.

3.2 Parallel B&B Algorithms

Parallel B&B algorithms have an interesting property. There is potential for super-linear speedups depending on the order the nodes in the solution tree are explored. Li and Wah [8] discuss theoretically how super-linear and sub-linear speedups can occur.

When searching the solution space in parallel, a good solution may be found on one of the first explored branches of the solution space. This may result in pruning subtrees explored later that would have to be searched in a sequential solver, resulting in better-than-linear speedups. There is also potential for degradation because a parallel solver may search subtrees that would have been pruned with a sequential search. Detrimental anomalies refer to situations where degradation results in parallel searches producing worse results than sequential. Detrimental anomalies occur due to inefficient search orders that may be due to either the sequential or parallel search order. Search orders that can produce anomalies are those that may not prioritize nodes with minimum lower bounds [8].

Tschoke et al. solve the TSP with a parallel B&B approach in a network of processors [20]. They identify three partly contrary goals for an efficient parallel algorithm. Minimizing idle processors, minimizing communication between processors and distributing sub-problems with the lowest bounds to the different processors first. The goal with the last requirement is to prioritize subtrees that will yield better solutions. Their method of distributing workloads among processors was to assign subtrees of the solution space dynamically. Processors in the network would communicate to only neighboring systems in the network to reduce network communication. The communication included updated upper and lower bounds and distributing subtrees between processors to maintain load balancing. [20]. Finding better solutions earlier will make it more likely to find accelerating anomalies as described by Li and Wah [8]. Tschoke et al. found close to linear speedups in a 1024 processor network [20].

Pekny and Miller solve the asymmetric-TSP with a parallel B&B algorithm using the assignment problem as a lower bound. In their parallel algorithm, they distribute nodes in the solution space needed to calculated lower bounds calculations. The subtree of the node is not distributed as children of nodes are put back into a global queue. Using a 10-CPU system, they find speedups ranging from 1-100 for various generated TSP instances. Overall, they found the best speedups for randomly generated problems [21].

McCreesh and Prosser analyzed parallel B&B algorithms for the maximum clique problem [22]. They found that load balancing can affect the performance of an algorithm, but the parallel search order is usually the dominating factor in speedups. In some cases, they found load balancing strategies could worsen the performance of an instance because the search order would change due to the balancing. Their approach to maintaining balance among threads was to start splitting the solution space into subtrees one level below the root. If a thread became idle, then the idle thread would steal work from active threads at levels further down from the root. This strategy produced good load balancing and run times compared to other balancing methods they tested [22].

De Bruin et al. use a parallel B&B algorithm on the TSP that uses lower bounds and dominance to prune nodes in the solution space [23]. Extending the work of Li and Wah [8], De Bruin et al. proved that with dominance, a parallel B&B can avoid detrimental anomalies if dominant nodes have smaller lower bounds than the dominated node [23]. Their experiments were on networks of workstations and found degradation related to hardware. As more workstations were added to the network, the performance usually improved. The exception was when a slower workstation was added to the

network. The slower workstation may have been assigned a part of the solution space with good solutions and the result is finding good solutions later due to the hardware [23].

Lalami and El-Baz use a parallel B&B algorithm to solve the *0-1* knapsack problem. They use a CPU-GPU hybrid where the GPU calculates lower bounds and branching. Using breadth, nodes in the solution space are transferred to the GPU to perform calculations. They report speedups from 8 to 20 on the CPU-GPU hybrid compared to the pure CPU algorithm [24].

Chapter 4: Sequential Algorithm

The sequential B&B algorithm consists of three steps based on the previous work of Shobaki and Jamal [6] and Jamal et al. [7]. Before performing B&B, the cost graph is pre-processed to remove the edges that cannot be in a valid solution. An edge is infeasible if it violates a precedence constraint in the problem. During B&B, the solution space is pruned by two methods. The first is based on dominance relations between nodes and the second is a lower bound that is based on relaxing the SOP to an assignment problem.

4.1 Cost Graph Preprocessing

The cost graph of the SOP may contain infeasible edges because of the way precedence constraints are encoded in the input file. We describe how to remove these infeasible edges based on the source code of Jamal et al.'s previous work [7].

Each instance of the SOP is stored in a files that contains a cost matrix. Each element in the matrix represents an outgoing edge from the row to the column (conversely an incoming edge from the column to the row). The edge weight determines whether the edge is in the cost graph or in the precedence graph. Non-negative weights are edges in the cost graph and negative weights are edges in the precedence graph. An example of the cost matrix is given in Figure 2. Vertices are labeled *0-3*. Elements in the table are edge weights with negative values indicating precedence relations.

Vertices	0	1	2	3
0	0	0	0	0
1	-1	0	10	20
2	-1	-1	0	14
3	-1	4	-1	0

a) The cost matrix as it appears in the instance file.

Vertices	0	1	2	3
0	0	0	21	21
1	-1	0	10	20
2	-1	-1	0	14
3	-1	-1	-1	0

b) The cost matrix after removing infeasible edges.

Figure 2: Example cost matrix from a SOP instance of 4 vertices

In Figure 2a *vertex-1* has an outgoing edge to *vertex-2* with a weight of 10. The outgoing *edge-(1, 0)* with weight 1 indicates *vertex-0* must precede *vertex-1* in a feasible solution. The cost matrix in each file may contain infeasible edges demonstrated in Figure 2. *Vertex-0* has an outgoing edge to *vertex-2* with a weight of 0. However, note that *vertex-2* must be preceded by *vertex-0* and *vertex-1* while *vertex-1* must be preceded by *vertex-0*. If *edge-(0-2)* was taken in a solution, it would violate a precedence constraint. If *vertex-1* is visited before *vertex-0* or after *vertex-2*, would violate the *(0, 1)* or *(1, 2)* precedence respectively. So we can then remove the *edge-(0, 2)* from the cost graph. In this case *vertex-0* was a recursive predecessor but not an immediate predecessor of *vertex-2* because *vertex-0* preceded one of *vertex-2*'s predecessors (vertex 1). In general, we can remove edges from recursive predecessors that are not immediate predecessors. To remove an edge, we can set the weight to infinity or, in practical terms, the largest edge weight plus one. In this example, the largest edge weight in the cost matrix is 20. So, we can set an edge weight to 21 to effectively eliminate it from the graph.

The other instance of infeasible edges are implied precedence relations. Note that *vertex-3* must be preceded by *vertex2* and has an outgoing edge of weight 4 to *vertex-1*. However, *vertex-1* must precede 2 so anything that 2 precedes, 1 must also precede. In this case we can remove the outgoing edge weight from *(3, 1)* and instead add an edge to the precedence graph from *(1, 3)*. In the cost matrix, this would change the weight to -1. *Vertex-3* was a recursive successor to *vertex-1* because *vertex-2* preceded *vertex-3* and *vertex-1* preceded *vertex-2*. After removing the recursive successors and predecessors we are left with the cost matrix in Figure 2b. Note that edges in the cost graph are removed and improves the use of edges to determine lower bounds for the problem.

4.2 Dominance Relations

Two nodes in the solution space are similar if they have visited the same vertices and are currently at the same vertex. Figure 3 shows an example of two partial paths. Both have visited the same vertices A, B and C and are currently at vertex B. Both have the same vertices to visit from the same location, so the remaining problem is identical between the similar nodes. The node with the lower partial path cost is strictly better since it will always find a better solution. In Figure 3 this would mean searching the path A, C, B while ignoring the path of C, A, B.

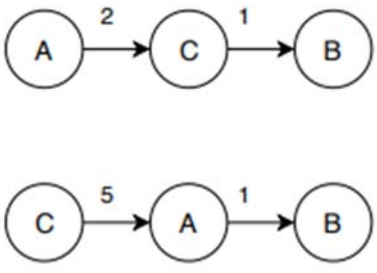


Figure 3: Example of two partial solutions

The path A, C, B has a cost of 3 and is strictly better than C, A, B which has a cost of 6.

We store similar nodes in a hash table. The key has two parts, a bit vector representing the visited vertices and a number to identify the current vertex. The stored value is both the partial cost of the path and the lower bound calculated at the partial path. For the rest of the paper we refer to the hash table of similar nodes as the history table. In general, the number of nodes in the solution space is too large to store in memory so we stop adding new entries after a threshold. Existing nodes are still updated if better partial paths are found.

4.3 Lower Bound

To determine a lower bound, two constraints are relaxed in the problem. The precedence and path requirements are relaxed. The cost graph is used to pair wise match vertices. This is the assignment problem and we use the Hungarian algorithm to solve it, which is $O(n^3)$ [17]. Figure 4a contains a cost matrix used in the Hungarian algorithm. For edges not in the cost graph, an infinity edge (or max weight plus one) is used. In Figure 4 the infinity edge weight is 9. The matching in Figure 4a produces a matching between the four vertices, *(0, 2)*, *(1, 3)*, *(2, 1)* and *(3, 0)*. Since an actual solution path will select three edges and not four, we can remove the largest among the selected edges for a lower bound. In Figure 4a, the *edge-(3, 0)* is discarded for a lower bound of 1.

During enumeration through the solution space, the cost graph is modified to fix the matching corresponding to edges taken in the partial solution. When we select another vertex to visit, the row and column in the cost matrix is adjusted to fix the new edge in the partial path. The cost of the taken edge is left as is, while the cost of the rest of the

row and column are set to infinity to force the matching between the two vertices. In Figure 4b this is shown after forcing the match from the edge-*(0, 1)*. The row of *vertex-0* and column of *vertex-1* are set to infinity except for the entry matching *(0, 1)*. This ensures *(0, 1)* are matching during the algorithm. The Hungarian algorithm is performed again to solve the new assignment problem but since only one row and column are changed, we can use the dynamic Hungarian algorithm which solves the Hungarian algorithm after an initial matching solution. The result is the matching of *(0, 1)*, *(1, 3)*, *(2, 0)* and *(3, 2)*. The *edge-(2, 0)* is discarded for a lower bound of 7. Dynamic Hungarian is $O(kn^2)$ where k is the number of matches removed from the previous matching [18]. In Figure 4b, the fixed edge removed two matches, so the Dynamic Hungarian would be $O(2n^2)$.

Vertices	**0**	**1**	**2**	**3**
0	9	0	0	0
1	9	9	5	0
2	9	1	9	3
3	9	8	7	9

a) Matching (0, 2), (1, 3), (2, 1) and (3, 0).

Vertices	**0**	**1**	**2**	**3**
0	9	0	9	9
1	9	9	5	0
2	9	9	9	3
3	9	9	7	9

b) Matching (0, 1), (1, 3), (2, 0) and (3, 2).

Figure 4: Cost matrices for the assignment problem

4.4 Sequential Algorithm Summary

The enumeration of the solution space is done with depth first search (DFS). The search order prioritizes the nearest neighbor with ties broken arbitrarily. At each node, the hash table is checked for a similar node. If we don't find an existing similar node,

then we calculate the bound and add an entry into the history table for the node. If a similar node is found with a lower partial cost, we simply prune the subtree and backtrack. If we find a similar node with a larger partial cost, we can use the stored bound to calculate the new bound without using the Dynamic Hungarian algorithm. The new lower bound will be the stored lower bound minus the difference in the costs of the partial paths. With the lower bound, we compare with the best solution to determine if we should search or prune the subtree.

Chapter 5: Parallel Algorithm

The parallel B&B algorithm utilizes the sequential algorithm as the basis for each thread. Each thread is assigned a subtree that is a portion of the solution space. In parallel each thread executes the sequential algorithm on its assigned subtree. The rest of this chapter describes how the subtrees of the solution space is divided between the threads, an example of how super-linear speedup is possible and how threads synchronize global data to minimize resource contention.

5.1 Super-linear Speedup

Figure 5 shows an example of sub-linear and super-linear speedups in a search with a tree and four children. Each node in this tree represents a subtree to search for a good solution. For simplicity we assume the subtrees take an equal amount of time to search. The circles represent subtrees where the green subtree contains a good solution. Once the green subtree is searched, the rest can be pruned. Blue and red arrows indicate searched and pruned subtrees respectively. Sequential search is done from left to right. Parallel search is done by splitting the subtrees into left and right halves where each half is searched from left to right. Once the subtree with a good solution is found, the rest of the subtrees are pruned. Figure 5a and 5b show how sub-linear speedups can occur. The good subtree is searched immediately in the sequential search. When adding a second thread to search two subtrees in parallel there is no speedup. The parallel search explored an extra subtree without knowing about the good solution it ended up searching one extra subtree compared to the sequential search. Figure 5c and 5d show how super linear search occurs. Now the sequential search explores three subtrees before finding the good

subtree. In parallel, the second thread finds the good subtree immediately and each thread explores only one subtree. In this case, the parallel search explored two subtrees, one less than the sequential search. The speedup would be a factor of three with two only threads.

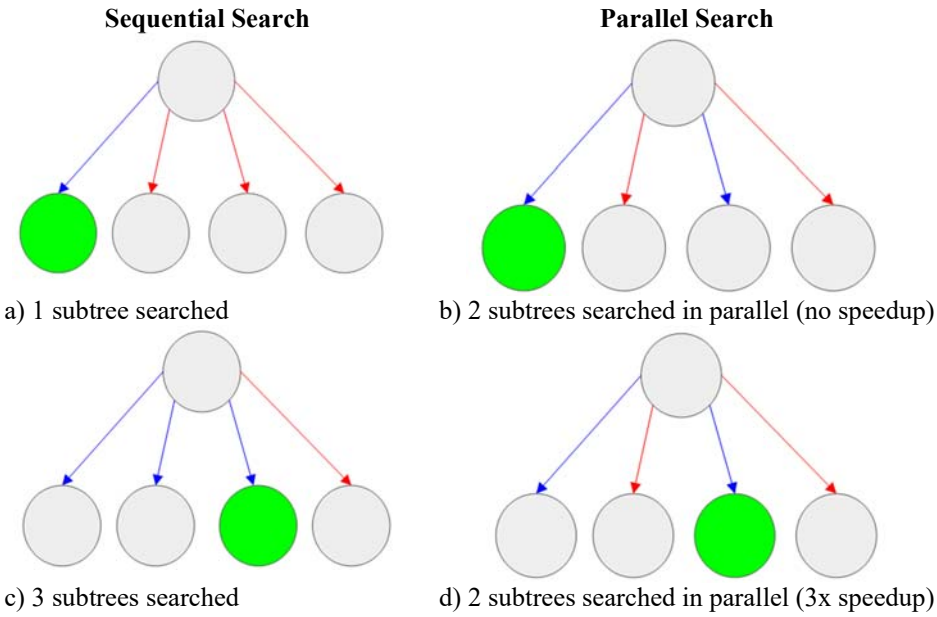


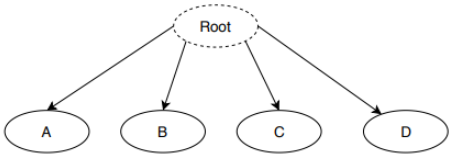
Figure 5: An example of sequential and parallel search on a tree

5.2 Subtree Division

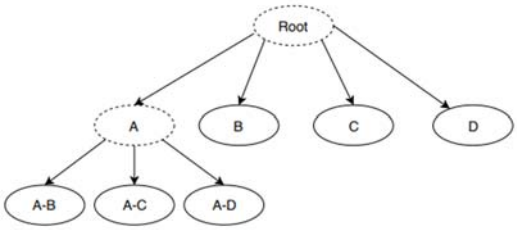
Subtrees of the solution space are stored in a global priority queue (GPQ) where the subtrees are prioritized by closeness to the root node (largest height). This priority provides a limited breadth first search (BFS). Threads are assigned subtrees as tasks to perform a DFS to find an optimal solution. Starting from the root node, feasible vertices are populated into the GPQ. The threads keep the GPQ populated by splitting subtrees if the GPQ would otherwise have fewer subtrees than the number of threads. If a thread needs to split a subtree, existing subtrees in the queue will be split into multiple subtrees to ensure the GPQ maintains tasks for the threads. An example of task splitting is given

in Figure 6. In Figure 6, solid circles represent the subtree in the queue. Dotted circles represent paths leading up to the subtree. The priority for each subtree is listed for each from most to least priority. Load balancing dynamically, when needed, follows the ideas of McCreesh and Prosser where doing unnecessary splitting can worsen the search order [22].

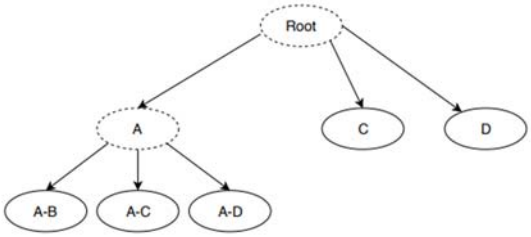
Suppose we start with the GPQ populated with four subtrees all one level below the root node, we label the subtrees A-D corresponding to the vertex visited after the root, shown in Figure 6a. Assume we have four active threads and *thread-1* requires a task. Since the size of the GPQ would be three if *thread-1* removed a subtree, one of the subtrees is split. Since all subtrees are one level below the root, they have the same priority. We can assume the subtree labeled A is split. The GPQ is left with subtrees B, C and D as before with A split into the possible paths following A as shown in Figure 6b. Afterwards *thread-1* removes a task and would prioritize B, C and D since those are closest to the root. We can arbitrarily select B to remove resulting in the GPQ in Figure 6c. Now *thread-2* requires a task and there will be at least four subtrees after removing one from the GPQ so *thread-2* simply removes a subtree. C and D are prioritized, and we arbitrarily pick C. The resulting GPQ is shown in Figure 6d with subtree D having the highest priority.



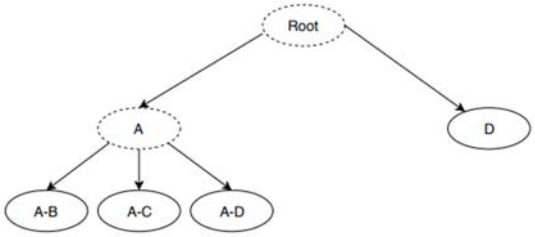
a) Starting priority, A, B, C, D (A=B=C=D)



b) Priority B, C, D (B=C=D), A-B, A-C, A-D (A-B=A-C=A-D) after *thread-1* splits A



c) Priority C, D (C=D), A-B, A-C, A-D (A-B=A-C=A-D) after *thread-1* removes B



d) Priority D, A-B, A-C, A-D (A-B=A-C=A-D) after *thread-2* removes C

Figure 6: Example subtrees in the GPQ

5.3 Managing Shared Data

The GPQ, global best solution and the history table are all shared data structures that require synchronization between the threads. The GPQ is locked by a thread anytime it is accessed. The global best solution is read very often to determine if a node's subtree should be pruned. Writes are much less common as this only happens when a better solution is found. To avoid thread contention, we allow reads without locks and only lock on writes. This allows a read to happen during a write, but an invalid read won't invalidate the solution. If the read is invalid it will read the cost of a previous best solution, but at worst a subtree that would have been pruned will be searched instead. The optimal solution will still be found with some effort wasted searching the subtree. This

potential waste is acceptable since it won't happen often, and the time taken for threads read locking will be high. If a read is done to determine whether to write, the read is performed again after locking to ensure the write condition is still being met.

The same method of avoiding read locks is applied to the hash table where preventing locks is just as important. To ensure reads in the hash table are referencing valid memory, the hash table uses separate chaining with a fixed number of buckets. The buckets are organized in an array with no resizing to ensure the memory locations do not change. The chains in a bucket use a linked list so as entries are added, the memory locations of previous entries do not change.

4.4 Parallel Algorithm Summary

Figure 7 shows pseudocode for the parallel algorithm. Each thread begins by obtaining a subtree from the GPQ to search, splitting a subtree if necessary. During DFS, the threads check against the history table and lower bounds to decide which nodes to enumerate. The best solution is an upper bound on the problem. If lower bounds are greater than or equal to the upper bounds the node can be pruned. Note during execution, the history table and best solution cost are read without locks. After locking the resource and before doing an update, the value is read again to ensure it was not updated between the thread's read and resource lock.

```
Repeat until GPQ is empty:
        Lock GPQ
                If size of GPQ is less than or equal to the thread count:
                        Split a subtree in the GPQ
                Remove a subtree from the GPQ
        Unlock GPQ
        Perform DFS on new subtree

DFS:
        Insert subtree root on stack
        Repeat until stack is empty:
                Remove node from stack
                Set enumerate to true
                If current node is a leaf of the subtree:
                        set enumerate to false
                        If current solution less than the best solution:
                                Lock the best solution
                                        If current solution cost less than upper bound:
                                                Update best solution
                                Unlock the best solution
                If similar node is found in history table:
                        If current node partial cost less than history node partial cost:
                        Lock history table bucket
                                        If current node cost less than history node cost:
                                                Update history node
                                                set lower bound to history node lower bound -
(history node cost - current partial cost)
                                Unlock history table bucket
                                If current node lower bound is not less than upper bound:
                                        set enumerate to false
                        Else:
                                set enumerate to false
                Else:
                        Calculate lower bound
                                If lower bound is not less than upper bound:
                                        set enumerate to false
                        Add new entry to history table
                If enumerate:
                        Add children of current node to the stack from the furthest to closest
```

Figure 7: Pseudocode of the parallel algorithm

Chapter 6: Experimental Results and Analysis

6.1 Testing Environment and Benchmarks

The parallel algorithm was tested on three benchmark sets (TSPLIB, SOPLIB, Compilers) for a total of 116 SOP instances. TSPLIB is a set of instances for the TSP and related problems maintained by the Heidelberg University. We use the TSPLIB sample set for the SOP [9]. SOPLIB is a set of SOP instances maintained by the Dalle Molle Institute for Artificial Intelligence of the University Of Applied Sciences Of Southern Switzerland [3]. The Compilers benchmark is a set of SOP instances taken from the work of Shobaki and Jamal on switching energy minimization in compilers [6].

The proposed parallel algorithm was run with a two-hour time limit per instance. The platform was a virtual machine with an Ubuntu 16.04.5 operating system. The hardware was an Intel i7-7700k CPU with 28 GB of main memory.

6.2 Results

With the sequential algorithm, a total of 66 instances were solved optimally. Using two threads, three additional instances were solved and with four threads a total of nine instances were solved that were not solved sequentially. All the instances that were solved sequentially were solved using the parallel algorithm with two or four threads.

Table 1 summarizes the results for each set of benchmarks. The speedup is the average of the sequential time divided by the parallel time across instances solved sequentially. The speedups for the individual instances are shown in the appendix. Table

1 shows the number of instances solved optimally using each algorithm (sequential, two-thread parallel and four-thread parallel). It also shows the speedup of the parallel algorithm for each number of threads (two or four) relative to the sequential algorithm. For the problems that were solved in parallel but not sequentially, the speedup will be at least the ratio of the time limit to the parallel solution time and we refer to this as a lower bound on the speedup. The best speedup lower bound was 21.6 with four threads on instance R.200.100.1 in SOPLIB. Within the time limit used in this test, no instance was improved relative to the previous best-known solution. Since the CPU on the system had four cores, tests were done with at most four threads. Otherwise, the hardware would limit speedup.

Table 1: Speedups of 2 and 4 threads

Benchmark	Total Instances	Solved Instances			Speedup	
		Sequential	2-Thread	4-Thread	2-Thread	4-Thread
TSPLIB	41	15	16	17	1.04	1.44
SOPLIB	48	25	27	32	1.39	2.52
Compilers	27	26	26	26	1.27	1.95
All	116	66	69	75	1.26	2.05

The last row in Table 1 shows the average speedup of parallelization is 1.26 with two threads and 2.05 with four threads, which is lower than expected. There are multiple factors that could cause this low speedup. One factor is thread contention on shared resource, including the GPQ and the global best solution. Another factor is that imbalanced workloads between threads can leave some threads idle while other threads are searching larger subtrees (sub-spaces). This can occur when the GPQ is empty and the threads that complete searching their assigned subtrees terminate while other threads

are active. A third factor is that the parallel algorithm may search more nodes as shown earlier in Figure 2.

6.3 Analysis

To investigate the degree of balance in the parallel algorithm, we added code that counts the number of nodes enumerated by each thread. The results are shown in Table 2. The *actual node ratio* is the ratio of the number of nodes explored sequentially to the number of nodes explored in the *bottleneck thread*. The bottleneck thread is the thread that explores the largest number of nodes. The actual node ratio is an upper bound on the speedup that can be achieved using the parallel algorithm. For example, suppose that the sequential algorithm enumerates 1000 nodes, while the four threads in the parallel algorithm enumerate 300, 350, 250, and 275 nodes. We would not expect a speedup larger than 1000/350 or 2.86.

The *ideal node ratio* is the ratio of the number of nodes explored sequentially to the average number of nodes explored by a thread. The ideal node ratio assumes that the thread workload is perfectly balanced and gives a theoretical limit on the speedup of the algorithm. In the previous example, the average number of nodes enumerated by a thread is 294, which gives an ideal speedup limit of 1000/294 or 3.4. The ideal node ratio also reflects the number of extra nodes searched by the parallel algorithm. If the parallel algorithm searches the same number of nodes as the sequential algorithm, the ideal node ratio will be equal to the number of parallel threads, which corresponds to linear speedup. Sub/super-linear speedups result in ratios smaller/larger than the number of threads.

Comparing the actual node ratio with the ideal node ratio gives an indication about the degree of imbalance. If the actual node ratio is significantly less than the ideal node ratio, that indicates a high degree of imbalance in the workload among the threads.

In Table 2, we see that the actual node ratio is less than the ideal node ratio among all instances. With four threads, the actual node ratio of enumerated nodes is 2.73 versus an ideal node ratio of 3.77. This is a significant imbalance that shows the need to improve load balancing in the algorithm.

One important consideration in studying load balancing is the relation between an instance's level of difficulty and the degree of imbalance. Table 2 shows average results for instances with different levels of difficulty. Instances that take longer times to solve are at higher levels of difficulty. The first row in the table shows the results with all solved instances included (difficulty level 0). The second row shows the results for only those instances that took at least 100 ms to solve (difficulty level 1), and the third row shows the results for the instances that needed at least 1 second to be solved optimally (difficulty level 2), and so on. The results in this table show that as the level of difficulty increases, the speedup and the actual node ratio become closer to the ideal. This means that the algorithm achieves a better balance for the harder instances that take longer times to solve.

Balancing trivial instance is expected to be harder, because as solution times get smaller the communication and synchronization overhead may overweigh the benefit from parallelization. For the fastest instances, background processes can even affect the

results. For the highest level of difficulty in the table (instances that take longer than 1 minute to solve), the average speedup with four threads is 3.29, which is reasonably good.

The ideal node ratio with four threads ranges between 3.77 and 3.9, which indicates that, on average, the parallel algorithm enumerates more nodes than the sequential algorithm. However, the fact that the ideal node ratios are close to 4 suggests that the major cause of the low speedup values is not enumerating more nodes; it is likely to be thread imbalance and the communication and synchronization overhead, which have a higher impact on the instances that are solved in less time. Therefore, we would expect the speedup and the degree of balance to be even better for the harder instances that take more than one minute to solve.

Table 2: Speedups and explored node ratios for different levels of difficulty

Solution Time of Instances	2-Thread Averages			4-Thread Averages		
	Speedup	**Actual Node Ratio**	**Ideal Node Ratio**	**Speedup**	**Actual Node Ratio**	**Ideal Node Ratio**
All	1.26	1.57	1.79	2.05	2.73	3.77
> 100ms	1.35	1.56	1.76	2.28	2.8	3.75
> 1 sec	1.4	1.59	1.73	2.41	2.98	3.78
> 10 sec	1.52	1.72	1.83	2.84	3.37	3.9
> 1 minute	1.62	1.75	1.9	3.29	3.58	3.85

For the harder instances (greater than 1 minute to solve), the average actual node ratio of 3.58 is only slightly below the ideal ratio of 3.85. This suggests that improving load balancing would not improve the solution time much on average, but it may improve the solution times of specific instances. Data for the individual instances shows that the

actual node ratio with four threads is much lower than the ideal ratio for some instances, such as R.700.100.60 in SOPLIB.

We next describe the attempts that we have to improve load balancing. The parallel algorithm was changed such that when the GPQ is empty, threads will wait instead of terminating. An active thread will detect that there are waiting threads and repopulate the GPQ with a portion of the subtree assigned to the active thread. This is similar to the work stealing of threads done by McCreesh and Prosser [22], except that in our case active threads give up work instead of idle threads stealing work. So far, the results show a 10-20 percent solution time improvement on some instances. The explored nodes become almost perfectly balanced with this change. However, the effectiveness of our load balancing technique still needs to be tested on harder instances that take longer running times to solve.

We have investigated resource contention as another possible cause of the low speedup, but the results of our investigation showed that the impact of contention is very low. On some instances, the time taken to access the GPQ was found to be on the order of 1 millisecond for instances that take minutes to solve. Because this time constitutes such a small fraction of the total solution time, we did not consider exploring techniques for reducing the contention overhead.

The speedup on the instances that are solved in parallel but time out sequentially indicates harder problems with longer running times have better speedups. The lower bound of speedups on these instances ranges from 1.2 to 21.6 with four threads.

However, it is important to note that search order with the parallel algorithm is different than the sequential search order, and this can be a disadvantage of the parallel algorithm in some cases. The parallel algorithm first performs a BFS to create multiple nodes and assign them to threads, while the sequential algorithm does a strict DFS. If the sequential search order happened to be good, the parallel algorithm may miss a good search order. This is evident on instances that time out but for which the sequential algorithm finds a better solution than the parallel algorithm. In TSPLIB, there are seven instances where the best sequential solution is better than the best parallel solution (kro124p.1, p43.1, p43.2, p43.3, ry48p.1, ry48p.2, ry48p.3). In general, prioritizing the search order to match the sequential search would conflict with the breadth-first creation and assignment of nodes to threads in the parallel algorithm, but different search orders give better results on different instances. On some other instances, the parallel algorithm finds better solutions than the sequential algorithm; it may even optimally solve some instances on which the sequential algorithm times out.

Chapter 7: Conclusions and Future Work

The parallel algorithm presented for the SOP has an overall speedup of 2.05 of four threads relative to the sequential algorithm with a time limit of two hours. Excluding instances that solve in less than one minute, the speedup increases to 3.29. Imbalanced workloads between threads are the main reason for lower speedups across all instances. Many of the imbalanced problems are solved quickly and harder instances show the best speedup. One instance solved with four threads but not sequentially has a speedup of at least 21.6. However, specific instances may be improved significantly with better load balancing.

Current work is focused on improving the load balancing and running tests with longer running times. A 48-hour time limit is needed to compare with previous work [7]. To aid load balancing, additional dynamic balancing is done where threads without assigned subtrees will wait instead of terminating if the GPQ is empty. Active threads will repopulate the GPQ with subtrees by sharing their current workload. More testing and experimentation is needed to know if this balancing method is enhancing the parallel algorithm described in Chapter 5.

Future work can build on the parallel algorithm to reduce the extra solution nodes explored in parallel. One method of doing this is to improve the search order. The current nearest neighbor search order is relatively fast to compute but using lower bounds or other methods could prove better. The better the search order, the faster good solutions are found. Improving the starting solution can also reduce the extra searching done in

parallel. The closer the starting solution's cost is to the optimal cost, then fewer solution nodes would be searched sequentially or in parallel. This would reduce both the potential for sub-linear and super-linear speedup because there is less opportunity to find better solutions when the starting solution is close to optimal.

Finally, more parallelization can be added to the problem. Using the GPU to compute the lower bounds, the bounds for many solution nodes can be calculated in parallel, adding more breadth to the parallel search. GPUs can also be used to do computations other than calculating lower bounds. In general, B&B involves lots of irregular control statements and branching. This would cause thread divergence on a GPU, degrading its performance. More research into performing branching or other function on a GPU could improve the usage of GPUs in parallel B&B.

Appendix: Tables of Experimental Results

Table 3: TSPLIB solution times

	Sequential		2-Threads		4-Threads	
Instance	Best Solution Cost	Time (sec)	Best Solution Cost	Time (sec)	Best Solution Cost	Time (sec)
br17.10	55	0.03	55	0.04	55	0.03
br17.12	55	0.02	55	0.02	55	0.01
ESC07	2125	0.00	2125	0.00	2125	0.00
ESC11	2075	0.00	2075	0.00	2075	0.00
ESC12	1675	0.00	1675	0.00	1675	0.00
ESC25	1681	0.01	1681	0.01	1681	0.00
ESC47	1288	1.30	1288	1.13	1288	1.08
ESC63	62	0.02	62	0.07	62	0.04
ESC78	21060	7200.02	20435	7200.05	18315	7200.03
ft53.1	8902	7200.01	8770	7200.06	8587	7200.03
ft53.2	10829	7200.01	11393	7200.06	9380	7200.03
ft53.3	11939	7200.01	12177	7200.03	11914	7200.02
ft53.4	14425	22.99	14425	19.47	14425	13.41
ft70.1	42163	7200.01	42013	7200.22	40953	7200.13
ft70.2	43566	7200.02	42500	7200.20	42602	7200.11
ft70.3	46847	7200.02	46827	7200.13	46451	7200.08
ft70.4	53530	1426.52	53530	1069.91	53530	779.36
rbg048a	363	7200.00	363	7200.02	352	7200.00
rbg050c	489	7200.00	468	7200.00	467	470.95
rbg109a	1038	15.68	1038	9.68	1038	6.04
rbg150a	1750	22.96	1750	14.98	1750	7.63
rbg174a	2120	7200.11	2033	5946.23	2033	2957.41
rbg253a	3201	7200.32	3184	7200.79	3151	7200.60
rbg323a	3667	7200.76	3681	7202.30	3665	7201.80
rbg341a	3238	7200.87	3201	7202.32	3121	7201.76
rbg358a	3889	7201.05	3887	7203.48	3814	7202.41
rbg378a	3939	7201.25	3909	7203.97	3899	7202.90
kro124p.1	45674	7200.02	48466	7200.27	47684	7200.16
kro124p.2	51271	7200.02	49404	7200.27	49404	7200.14
kro124p.3	64124	7200.03	60919	7200.19	61726	7200.12
kro124p.4	85663	7200.03	87008	7200.11	84694	7200.08
p43.1	28310	7200.00	28630	7200.02	28575	7200.01
p43.2	28545	7200.00	28795	7200.01	28825	7200.01
p43.3	28885	7200.00	29250	7200.01	29260	7200.00
p43.4	83005	5.69	83005	6.79	83005	3.74
prob.100	1808	7200.03	1751	7200.11	1754	7200.08
prob.42	243	3067.60	243	2551.56	243	1838.83
ry48p.1	17238	7200.00	19578	7200.03	18393	7200.01
ry48p.2	18049	7200.00	20127	7200.02	18109	7200.01
ry48p.3	20772	7200.00	20984	7200.01	20841	7200.01
ry48p.4	31446	30.44	31446	28.50	31446	23.67

Table 4: SOPLIB solution times

	Sequential		2-Threads		4-Threads	
Instance	Best Solution Cost	Time (sec)	Best Solution Cost	Time (sec)	Best Solution Cost	Time (sec)
R.200.100.1	71	7200.13	61	390.93	61	333.13
R.200.100.15	1968	7200.17	1792	5027.33	1792	2836.48
R.200.100.30	4216	7.12	4216	4.21	4216	2.14
R.200.100.60	71749	0.17	71749	0.18	71749	0.17
R.200.1000.1	1511	7200.22	1429	7200.59	1404	5295.56
R.200.1000.15	20481	3617.10	20481	1961.13	20481	1030.58
R.200.1000.30	41196	5.81	41196	3.78	41196	2.41
R.200.1000.60	71556	0.34	71556	0.29	71556	0.22
R.300.100.1	82	7200.49	74	7201.33	26	1752.56
R.300.100.15	4320	7200.69	3699	7201.14	3152	5978.54
R.300.100.30	6120	36.56	6120	19.45	6120	10.53
R.300.100.60	9726	0.62	9726	0.68	9726	0.59
R.300.1000.1	2853	7200.66	1539	7202.89	1344	7201.46
R.300.1000.15	36286	7200.81	35419	7201.29	33304	7201.08
R.300.1000.30	54147	107.33	54147	73.64	54147	42.15
R.300.1000.60	109471	1.31	109471	1.37	109471	0.78
R.400.100.1	46	7201.20	20	7202.49	13	2577.82
R.400.100.15	6366	7201.54	6347	7202.92	5902	7202.47
R.400.100.30	8165	102.83	8165	60.53	8165	28.89
R.400.100.60	15228	2.50	15228	2.22	15228	1.65
R.400.1000.1	2268	7201.88	2191	7208.00	2179	7205.70
R.400.1000.15	65187	7202.48	63778	7204.79	58129	7203.85
R.400.1000.30	85128	173.29	85128	100.21	85128	41.14
R.400.1000.60	140816	3.07	140816	3.19	140816	1.76
R.500.100.1	147	7202.35	38	7206.19	4	5042.90
R.500.100.15	8565	7203.54	8308	7206.73	7853	7205.65
R.500.100.30	9665	305.88	9665	195.43	9665	100.78
R.500.100.60	18240	7.61	18240	7.93	18240	3.80
R.500.1000.1	2244	7203.16	2246	7211.93	2201	7212.06
R.500.1000.15	81628	7204.25	76117	7208.01	71733	7206.95
R.500.1000.30	98987	618.06	98987	326.98	98987	148.90
R.500.1000.60	178212	5.09	178212	4.89	178212	4.07
R.600.100.1	164	7204.06	84	7208.61	21	7207.85
R.600.100.15	11813	7205.58	11387	7211.04	10574	7208.88
R.600.100.30	12465	385.07	12465	231.27	12465	141.13
R.600.100.60	23293	9.23	23293	8.18	23293	6.98
R.600.1000.1	3212	7205.76	2750	7230.91	2733	7224.93
R.600.1000.15	99632	7207.36	99819	7213.28	93042	7211.92
R.600.1000.30	126798	744.93	126798	538.12	126798	275.46
R.600.1000.60	214608	10.83	214608	9.20	214608	6.97
R.700.100.1	131	7206.95	109	7218.44	57	7213.13
R.700.100.15	13130	7210.16	12929	7219.98	12518	7217.06
R.700.100.30	14510	1855.55	14510	866.63	14510	323.64
R.700.100.60	24102	17.27	24102	14.33	24102	11.24
R.700.1000.1	3049	7209.00	2853	7243.28	2829	7227.72

	Sequential		2-Threads		4-Threads	
Instance	Best Solution Cost	Time (sec)	Best Solution Cost	Time (sec)	Best Solution Cost	Time (sec)
R.700.1000.15	131180	7211.71	126435	7223.49	119744	7219.73
R.700.1000.30	134474	1264.56	134474	855.69	134474	332.55
R.700.1000.60	245589	17.70	245589	14.75	245589	10.62

Table 5: Compilers solution times

	Sequential		2-Threads		4-Threads	
Instance	Best Solution Cost	Time (sec)	Best Solution Cost	Time (sec)	Best Solution Cost	Time (sec)
gsm.153.124	1109	0.09	1109	0.10	1109	0.08
gsm.444.350	2745	0.48	2745	0.51	2745	0.52
gsm.462.77	577	1.37	577	0.92	577	0.63
jpeg.1483.25	93	0.06	93	0.06	93	0.04
jpeg.3184.107	791	5.00	791	2.95	791	1.71
jpeg.3195.85	68	1.55	68	1.09	68	0.62
jpeg.3198.93	312	3.06	312	1.75	312	0.92
jpeg.3203.135	850	10.50	850	9.16	850	3.85
jpeg.3740.15	40	0.11	40	0.10	40	0.10
jpeg.4154.36	167	0.16	167	0.09	167	0.10
jpeg.4753.54	245	1.31	245	0.86	245	0.52
susan.248.197	1338	53.91	1338	39.35	1338	28.93
susan.260.158	1016	6.63	1016	3.92	1016	3.12
susan.343.182	1207	9.36	1207	8.05	1207	3.71
typeset.10192.123	630	7200.04	628	7200.16	627	7200.09
typeset.10835.26	127	1.18	127	0.72	127	0.32
typeset.12395.43	174	8.52	174	5.82	174	4.83
typeset.15087.23	98	0.00	98	0.00	98	0.00
typeset.15577.36	155	0.46	155	0.49	155	0.20
typeset.16000.68	84	0.92	84	0.70	84	0.52
typeset.1723.25	64	0.31	64	0.24	64	0.20
typeset.19972.246	2018	0.19	2018	0.23	2018	0.24
typeset.4391.240	1605	3.38	1605	1.98	1605	1.22
typeset.4597.45	184	5.46	184	4.23	184	3.28
typeset.4724.433	3466	1.13	3466	1.29	3466	1.20
typeset.5797.33	131	0.00	131	0.01	131	0.00
typeset.5881.246	1732	1.86	1732	1.23	1732	0.95

Table 6: TSPLIB speedups and enumerated node ratios
Lower bound speedups are shown in bold

	2-Threads			4-Threads		
Instance	Speedup	Actual Node Ratio	Ideal Node Ratio	Speedup	Actual Node Ratio	Ideal Node Ratio
br17.10	0.85	1.72	1.94	1.35	2.45	3.88
br17.12	1.27	1.58	1.51	1.41	1.48	3.01
ESC07	0.19	0.91	0.90	0.14	0.98	1.79
ESC11	0.67	1.10	1.68	0.59	1.78	3.37
ESC12	1.14	1.42	1.66	0.94	1.75	3.32
ESC25	1.30	1.95	2.15	1.92	2.92	4.30
ESC47	1.15	1.27	0.89	1.20	1.60	1.77
ESC63	0.31	2.27	2.34	0.49	2.40	4.68
ESC78						
ft53.1						
ft53.2						
ft53.3						
ft53.4	1.18	1.26	1.20	1.71	1.86	2.40
ft70.1						
ft70.2						
ft70.3						
ft70.4	1.33	1.53	1.29	1.83	2.09	2.57
rbg048a						
rbg050c				**15.29**		
rbg109a	1.62	1.73	1.79	2.60	2.80	3.59
rbg150a	1.53	1.62	1.70	3.01	3.26	3.40
rbg174a	**1.21**			**2.43**		
rbg253a						
rbg323a						
rbg341a						
rbg358a						
rbg378a						
kro124p.1						
kro124p.2						
kro124p.3						
kro124p.4						
p43.1						
p43.2						
p43.3						
p43.4	0.84	0.85	1.24	1.52	1.57	2.47
prob.100						
prob.42	1.20	1.23	1.07	1.67	1.65	2.15
ry48p.1						
ry48p.2						
ry48p.3						
ry48p.4	1.07	1.16	1.39	1.29	1.39	2.77

Table 7: SOPLIB speedups and enumerated node ratios
Lower bound speedups are shown in bold

	2-Threads			4-Threads		
Instance	Speedup	Actual Node Ratio	Ideal Node Ratio	Speedup	Actual Node Ratio	Ideal Node Ratio
R.200.100.1	**18.42**			**21.61**		
R.200.100.15	**1.43**			**2.54**		
R.200.100.30	1.69	1.77	1.91	3.32	3.83	4.09
R.200.100.60	0.91	1.11	2.19	1.01	1.37	3.80
R.200.1000.1				**1.36**		
R.200.1000.15	1.84	1.98	2.06	3.51	3.91	4.20
R.200.1000.30	1.54	1.72	1.74	2.41	2.67	3.21
R.200.1000.60	1.14	2.05	2.15	1.51	3.52	6.32
R.300.100.1				**4.11**		
R.300.100.15				**1.20**		
R.300.100.30	1.88	2.05	2.15	3.47	4.23	4.29
R.300.100.60	0.91	1.46	1.80	1.05	2.30	3.91
R.300.1000.1						
R.300.1000.15						
R.300.1000.30	1.46	1.66	1.81	2.55	2.88	3.84
R.300.1000.60	0.96	1.00	1.99	1.69	2.57	4.78
R.400.100.1				**2.79**		
R.400.100.15						
R.400.100.30	1.70	1.82	1.91	3.56	4.00	4.29
R.400.100.60	1.13	1.84	1.89	1.51	2.88	4.56
R.400.1000.1						
R.400.1000.15						
R.400.1000.30	1.73	1.88	1.92	4.21	4.74	5.02
R.400.1000.60	0.96	1.00	2.00	1.75	3.38	5.78
R.500.100.1				**1.43**		
R.500.100.15						
R.500.100.30	1.57	1.72	1.73	3.03	3.40	3.61
R.500.100.60	0.96	1.00	2.00	2.00	3.42	6.93
R.500.1000.1						
R.500.1000.15						
R.500.1000.30	1.89	2.08	2.08	4.15	4.58	4.72
R.500.1000.60	1.04	1.56	1.70	1.25	1.60	3.63
R.600.100.1						
R.600.100.15						
R.600.100.30	1.67	1.80	1.81	2.73	3.03	3.44
R.600.100.60	1.13	1.86	2.17	1.32	3.13	4.17
R.600.1000.1						
R.600.1000.15						
R.600.1000.30	1.38	1.52	1.53	2.70	3.01	3.05
R.600.1000.60	1.18	1.82	1.97	1.55	3.52	4.06
R.700.100.1						
R.700.100.15						
R.700.100.30	2.14	2.31	2.34	5.73	6.41	6.58
R.700.100.60	1.21	2.42	2.45	1.54	5.22	5.95
R.700.1000.1						

	2-Threads			4-Threads		
Instance	Speedup	Actual Node Ratio	Ideal Node Ratio	Speedup	Actual Node Ratio	Ideal Node Ratio
R.700.1000.15						
R.700.1000.30	1.48	1.57	1.61	3.80	4.20	4.33
R.700.1000.60	1.20	1.77	1.84	1.67	2.61	4.36

Table 8: Compilers speedups and enumerated node ratios

	2-Threads			4-Threads		
Instance	Speedup	Actual Node Ratio	Ideal Node Ratio	Speedup	Actual Node Ratio	Ideal Node Ratio
gsm.153.124	0.94	1.87	1.93	1.11	1.58	2.47
gsm.444.350	0.94	1.58	1.81	0.92	1.48	2.27
gsm.462.77	1.49	1.30	1.64	2.16	1.64	2.99
jpeg.1483.25	0.98	1.95	2.11	1.47	3.73	4.66
jpeg.3184.107	1.69	1.77	1.85	2.93	3.12	3.59
jpeg.3195.85	1.43	1.96	1.96	2.50	2.19	2.74
jpeg.3198.93	1.74	1.64	1.97	3.33	3.12	3.88
jpeg.3203.135	1.15	1.19	1.22	2.73	3.02	3.11
jpeg.3740.15	1.16	1.71	1.79	1.19	3.02	3.82
jpeg.4154.36	1.70	1.75	1.88	1.58	2.63	3.69
jpeg.4753.54	1.51	1.58	1.67	2.52	2.41	3.15
susan.248.197	1.37	1.27	1.71	1.86	1.46	2.68
susan.260.158	1.69	1.55	1.88	2.12	2.06	2.97
susan.343.182	1.16	1.27	1.29	2.52	2.96	3.08
typeset.10192.123						
typeset.10835.26	1.64	2.12	2.22	3.71	4.71	5.18
typeset.12395.43	1.46	1.57	1.61	1.76	1.69	3.34
typeset.15087.23	0.70	1.29	2.56	1.74	5.29	7.54
typeset.15577.36	0.95	1.08	1.30	2.35	2.06	3.35
typeset.16000.68	1.32	1.57	1.59	1.79	1.45	1.92
typeset.1723.25	1.31	1.54	1.71	1.58	3.10	4.24
typeset.19972.246	0.83	1.00	1.98	0.80	1.70	3.41
typeset.4391.240	1.71	1.63	1.98	2.77	2.66	3.42
typeset.4597.45	1.29	1.30	1.43	1.66	1.59	2.83
typeset.4724.433	0.88	1.11	1.35	0.94	1.69	3.36
typeset.5797.33	0.58	1.19	2.38	0.85	1.77	3.91
typeset.5881.246	1.51	1.41	2.04	1.96	1.49	3.02

References

1. L.F. Escudero, “An inexact algorithm for the sequential ordering problem,” *European Journal of Operational Research*, vol. 37, no. 2, pp. 236–249, 1988.

2. N. Ascheuer, L. Escudero, M. Grötschel and M. Stoer, "A Cutting Plane Approach to the Sequential Ordering Problem (with Applications to Job Scheduling in Manufacturing)," *SIAM Journal on Optimization*, vol. 3, no. 1, pp. 25-42, 1993.

3. M. Mojana, R. Montemanni, G.A. Di Caro and L.M. Gambardella, “A branch and bound approach for the sequential ordering problem,” *Lecture Notes in Management Science*, vol. 4, pp. 266–273, 2012.

4. S. Spieckermann, K. Gutenschwager and S. Voß “A sequential ordering problem in automotive paint shops,” *International Journal of Production Research*, vol. 42, no. 9, pp. 1865-1878, (Feb 2007).

5. M. F. Timlin and W. Pulleyblank, "*Precedence Constrained Routing and Helicopter Scheduling: Heuristic Design,*" *Interfaces*, vol. 22, no. 3, pp. 100-111, 1992.

6. G. Shobaki and J. Jamal, “An exact algorithm for the sequential ordering problem and its application to switching energy minimization in compilers,” *Computational Optimization and Applications*, vol. 61, no 2, pp. 343-372, 2015.

7. J. Jamal, G. Shobaki, V. Papapanagiotou, L. M. Gambardella and R. Montemanni, "Solving the sequential ordering problem using branch and bound,"*2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, Honolulu, HI, pp. 1-9, 2017.

8. G. Li and B. W. Wah, "Coping with Anomalies in Parallel Branch-and-Bound Algorithms" *IEEE Transactions on Computers*, vol. C-35, no. 6, pp. 568-573, June 1986.

9. G. Reinhelt. “TSPLIB: a library of sample instances for the TSP (and related problems) from various sources and of various types." [Online]. Available: https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/ (Accessed: April 2019).

10. L. M. Gambardella and M. Dorigo. “Has-Sop: Hybrid Ant System for the Sequential Ordering Problem,” Technical Report, Dalle Molle Institute for Artificial Intelligence, 1997.

11. L. M. Gambardella and M. Dorigo. “An Ant Colony System Hybridized with a New Local Search for the Sequential Ordering Problem,” *INFORMS J. on Computing,* vol. 12, no. 3, pp. 237-255, August 2000.

12. F. Guerriero and M. Mancini. “A cooperative parallel rollout algorithm for the sequential ordering problem,” Parallel Computing, vol. 29, 5, pp. 663-677, May 2003.

13. K. Helsgaun. “An Extension of the Lin-Kernighan-Helsgaun TSP Solver for Constrained Traveling Salesman and Vehicle Routing Problems,” Technical Report, Roskilde University, 2017.

14. N. Ascheuer, M. Jünger and G. Reinelt. "A branch & cut algorithm for the asymmetric traveling salesman problem with precedence constraints." *Computational Optimization and Applications*, vol. 17, no. 1, pp. 61-84, 2000.

15. V. Papapanagiotou, J. Jamal, R. Montemanni, G. Shobaki and L. M. Gambardella, "A comparison of two exact algorithms for the sequential ordering problem," *2015 IEEE Conference on Systems, Process and Control (ICSPC)*, Bandar Sunway, 2015, pp. 73-78.

16. H. W. Kuhn. "The Hungarian method for the assignment problem." *Naval research logistics quarterly*, vol. 2, no. 1-2, pp. 83-97. 1955.

17. C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, 1998.

18. G. Mills-Tettey, G. Ayorkor, A. Stentz, and M. B. Dias. "The Dynamic Hungarian Algorithm for the Assignment Problem with Changing Costs" Robotics Institute, Carnegie Mellon University 2007.

19. Y. Salii. "Revisiting dynamic programming for precedence-constrained traveling salesman problem and its time-dependent generalization," *European Journal of Operational Research*, Elsevier, vol. 272, no. 1, pp. 32-42, 2019.

20. S. Tschoke, R. Lubling and B. Monien, "Solving the traveling salesman problem with a distributed branch-and-bound algorithm on a 1024 processor network" *Proceedings of 9th International Parallel Processing Symposium*, Santa Barbara, CA, pp. 182-189, 1995.

21. J. F. Pekny and D. L. Miller. “A parallel branch and bound algorithm for solving large asymmetric traveling salesman problems,” *Mathematical Programming*, vol. 55, pp. 17-33, 1992.

22. C. McCreesh and P. Prosser. “The Shape of the Search Tree for the Maximum Clique Problem and the Implications for Parallel Branch and Bound,” *ACM Trans. Parallel Computing*, vol. 2, no. 1, April 2015.

23. A. de Bruin, G. A. P. Kindervater and H. W. J. M. Trienekens. “Asynchronous Parallel Branch and Bound and Anomalies,” *In Proceedings of the Second International Workshop on Parallel Algorithms for Irregularly Structured Problems,* London, UK, pp. 363-377, 1995.

24. M. E. Lalami and D. El-Baz, "GPU Implementation of the Branch and Bound Method for Knapsack Problems," *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, Shanghai, 2012, pp. 1769-1777.