Contents lists available at ScienceDirect
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

A parallel branch-and-bound algorithm with history-based domination and its application to the sequential ordering problem

Taspon Gonggiatgul, Ghassan Shobaki, Pınar Muyan-Özçelik

Department of Computer Science, California State University, 6000 J street, Sacramento, 95819, CA, USA

ARTICLE INFO

Article history:
Received 24 March 2022
Received in revised form 10 September 2022
Accepted 11 October 2022
Available online xxxx

Keywords:
Parallel branch-and-bound
Sequential ordering problem
Combinatorial optimization
NP-complete problems
History domination

ABSTRACT

In this paper, we describe the first parallel Branch-and-Bound (B&B) algorithm with a history-based domination technique. Although history-based domination substantially speeds up a B&B search, it makes parallelization much more challenging. Our algorithm is the first parallel exact algorithm for the Sequential Ordering Problem using a pure B&B approach. To effectively explore the solution space, we have developed three novel parallelization techniques: thread restart, parallel history domination, and history-table memory management. The proposed algorithm was experimentally evaluated using the SOPLIB and TSPLIB benchmarks on multi-core processors. Using 32 threads with a time limit of one hour, the algorithm gives geometric-mean speedups of 72x and 20x on the medium-difficulty SOPLIB and TSPLIB instances, respectively. On the hard instances, it solves 12 instances that the sequential algorithm does not solve, with geometric-mean speedups of 16x on SOPLIB and 32x on TSPLIB. Super-linear speedups up to 366x are seen on 16 instances.

1. Introduction

In this paper, we show how to effectively parallelize a Branch-and-Bound (B&B) algorithm that has a history-based domination technique on a multicore processor. Although history-based domination substantially speeds up a B&B search, it creates great challenges in parallelizing the algorithm. The proposed parallel B&B algorithm is applied to the Sequential Ordering Problem (SOP). To the best of our knowledge, the proposed algorithm is the first parallel B&B algorithm that includes history-based domination and is the first parallel exact algorithm for solving the SOP using a pure B&B approach.

The SOP is a generalization of the Traveling Salesman Problem (TSP), which is a well-known NP-hard combinatorial optimization problem. Given a weighted graph and a dependence graph representing precedence constraints among the vertices, the objective in the SOP is finding a minimum-cost Hamiltonian path in the weighted graph that satisfies the precedence constraints in the dependence graph.

Precedence constraints make developing a parallel B&B algorithm more challenging, because they make it harder to estimate and balance thread loads. Parallel B&B algorithms for optimization problems with precedence constraints are understudied in the literature. We believe that the parallelization techniques presented in this paper are general enough to be applicable to many precedence-constrained combinatorial optimization problems.

The proposed parallel algorithm is a pool-based algorithm that consists of a collection of techniques that we designed to effectively search the solution space using multiple parallel threads. The techniques used in our algorithm include three novel techniques: thread restart (Section 4.2), parallel history-based domination (Section 5.1), and history table memory management (Section 5.2). In addition to developing these new techniques, we have experimented with multiple methods for assigning global-pool nodes to threads (Section 3.4) and balancing the load by work stealing (Section 4.1) [35].

The proposed parallel algorithm is based on the sequential B&B algorithm that was originally proposed by Shobaki and Jamal [46] and later enhanced by Jamal et al. [30] for solving the SOP. This sequential algorithm includes a history-based domination technique that stores information about previously explored sub-problems in a history table and uses that information to speedup the processing of similar sub-problems. That technique was a generalization of the history-based domination technique that was originally introduced by Shobaki and Wilken [47] for solving the compiler instruction scheduling problem. Although this history technique is an effective technique that greatly reduces the size of the search tree, it makes parallelizing the algorithm much more challenging.

E-mail address: tagonggiatgul@ucdavis.edu (T. Gonggiatgul).

https://doi.org/10.1016/j.jpdc.2022.10.007
0743-7315/© 2022 Published by Elsevier Inc.

These challenges and the techniques that we have developed to tackle them are discussed in Section 5.

What distinguishes parallel search algorithms like B&B from other types of parallel computation is the possibility of achieving *super-linear speedup* (also known as acceleration anomaly [34,16]) relative to the sequential algorithm, that is, a speedup ratio that is greater than the number of parallel threads. Super-linear speedup is possible in search algorithms, because the performance is highly dependent on the search order, and the search order of the parallel algorithm can be better than that of the sequential algorithm. The search order has a high impact on performance, because the best solution found so far is dependent on this order, and the degree of pruning at any given point depends on the value of the best solution at the point.

When history-based domination is used, super-linear speedup becomes even more likely. Within the same time period, a parallel algorithm can explore different sub-spaces simultaneously. This does not only increase the chances of finding a better best solution, but it also stores more diverse information in the history table, thus increasing the chances of applying history-based pruning. However, filling the history table at a faster rate makes it more challenging to use the available space to store the most useful information in the table. The contributions of this paper may be summarized as:

1. We propose the first parallel B&B algorithm with history-based domination. The algorithm includes three novel techniques that we have developed as well as our experimentation-based versions of two known techniques.
2. We apply the proposed algorithm to the SOP, which is an NP-hard sequencing problem with precedence constraints, and thus we provide insights into the understudied area of parallel B&B algorithms for precedence-constrained sequencing problems.
3. We present a thorough experimental evaluation of the proposed parallel algorithm on the whole SOPLIB and TSPLIB benchmark suites. The results show that with our parallelization techniques, super-linear speedup is not an anomaly; it can be achieved on many instances. We report super-linear speedup ratios that are much greater than the number of threads, with the highest ratio being 366 on a 32-core processor.

2. Previous work

The SOP was introduced by Escudero [17]. Various sequential algorithms using both heuristic and exact approaches have been proposed for solving the SOP. Heuristic approaches use different techniques, including ant colony optimization [19,50,20], particle swarm optimization [2], and the Lin-Kernighan-Helsgaun algorithm [28]. On the other hand, exact approaches use various methods such as the Lagrangian relaxation [18], cutting plane [5,25], branch-and-cut [6,26], branch-and-bound [38,46,30], constraint programming [32], and dynamic programming [37,44].

Parallel algorithms have also been proposed for solving the SOP. Guerriero and Mancini [27] propose a heuristic parallel rollout algorithm. Exact parallel algorithms for the SOP are understudied. The only work that we are aware of on exact parallel algorithms for the SOP is the recent work of Salii and Sheka [45], who propose a hybrid algorithm combining dynamic programming with a Morin-Marsten B&B scheme (DPBB). In contrast, our proposed parallel algorithm is based on a pure B&B approach. Since our algorithm does not use dynamic programming, it does not have a strict memory restriction. Our history table management technique described in Section 5.2 is an adaptive technique that is designed to make the best use of available memory. To the best of our knowledge, no previous parallel algorithm for solving the SOP using a pure B&B approach has been proposed.

However, various parallel B&B algorithms have been proposed for solving other optimization problems, such as the Traveling Salesman Problem (TSP) [41,52,16], Maximum-Clique [35], 01-Knapsack [31,33], Blocking Job Shop Scheduling [14], Optimal Batch Plants Design [9], Quadratic Assignment [7,13,22], N-Queens [22], and Flow-Shop Scheduling [22,15,24,11,23]. These algorithms are run on different parallel architectures, including multiprocessors [41], processor networks [52,16], distributed/shared memory parallel systems [8], multi-core CPUs [35,24,9], clusters [9], FPGAs [15], and GPUs [9,23,33]. Some previous papers propose parallel B&B algorithms on a hybrid platform with a combination of multi-core CPUs and GPUs [11,22,14]. Bader et al. [7] discuss how the design of parallel B&B algorithms is influenced by the nature of the computational platform.

Gmys et al. [24] indicate that B&B can be parallelized in different ways, including parallelizing the tree search, parallelizing child-node evaluation, and parallelizing lower bound computation. Janakiram et al. [31] concurrently explore multiple search trees that use a different selection operator.

A survey of early parallel B&B algorithms was conducted by Gendron and Crainic [21]. In a later paper, Crainic et al. [13] present various parallelization strategies for B&B and review frameworks that help implement these strategies. A more recent survey of these frameworks and other multi-core CPU B&B approaches is provided by Gmys [22]. These frameworks ease the development but cannot provide the high performance that custom algorithms achieve as indicated by Herrera et al. [29].

Previous work on parallel B&B algorithms for precedence-constrained problems like the SOP is quite limited. Among the above-mentioned algorithms, only Dabah et al. [14] propose a parallel B&B approach for a precedence-constrained problem, which is Blocking Job Shop Scheduling.

Anderson et al. [1] use tree estimation to develop a restart technique for achieving a better search order in B&B. However, the work of Anderson et al. was within a sequential algorithm. Archibald et al. [4] propose a parallel restart algorithm that restarts the search from the beginning if no progress has been made. Archibald et al. [3] also propose a technique for avoiding a poor parallel search order by forcing a sequential ordering on a thread. Chu et al. [12] propose a parallel restart algorithm that focuses the search on the most promising sub-spaces. In Section 4.2 of the current paper, we describe a parallel restart algorithm that utilizes multiple threads to balance focusing the search on the most promising sub-spaces with exploring new sub-spaces.

Morrison et al. [40] provide a list of B&B approaches that use memory-based and nonmemory-based dominance relations. Tomazella and Nagano [51] provide a list of dominance rules used in B&B algorithms for flowshop scheduling problems. Those techniques were used in sequential B&B algorithms. In the current paper, we present a parallel version of the history-based domination technique proposed by Shobaki and Jamal [46].

3. Basic algorithm description

3.1. Problem description

An instance of the Sequential Ordering Problem (SOP) consists of a cost graph $G=(V,E)$ and a precedence graph $P=(V,R)$ defined on the same set of vertices V, as well as a start vertex s and a final vertex f that both belong to V.

The cost graph is a complete directed weighted graph $G=(V,E)$ in which each edge (i,j) in E is assigned a weight $w(i,j)$. A path in the graph is a sequence of edges from E. The cost of a path is the sum of the weights of the edges that constitute that

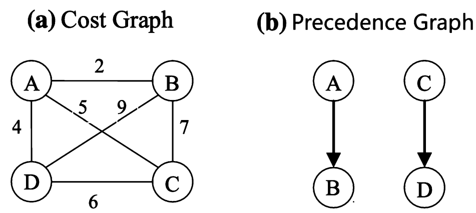

Fig. 1. Example SOP instance.

path. A Hamiltonian path is a path that visits *every* vertex in the graph exactly once. A Hamiltonian path is guaranteed to exist in a complete graph.

The precedence graph $P = (V, R)$ is a directed graph in which an edge (x, y) in R indicates that vertex x must appear before vertex y in any *feasible path*. If the precedence constraints in P imply that vertex n cannot appear immediately after vertex m in any feasible path, the weight of edge (m, n) in G will be irrelevant, and we follow the convention of setting this weight to -1.

The SOP is the problem of finding a minimum-cost Hamiltonian path in G that starts with s, ends with f, and satisfies the precedence constraints imposed by P.

A small example of a SOP instance is shown in Fig. 1, including the cost graph (Fig. 1a) and the precedence graph (Fig. 1b). Since in this instance, the edge weight is the same in either direction ($w(i, j) = w(j, i)$ for every i and j), the graph is drawn as an undirected graph. Assuming that the start vertex is A and the final vertex is D, there will be only two feasible Hamiltonian paths, namely ⟨A, B, C, D⟩ of cost 15, and ⟨A, C, B, D⟩ of cost 21. Therefore, the solution to this SOP instance is ⟨A, B, C, D⟩, because it is the feasible Hamiltonian path with the lowest cost.

3.2. Sequential algorithm summary

The proposed parallel algorithm is based on the sequential B&B algorithm that was originally proposed by Shobaki and Jamal [46] and later enhanced by Jamal et al. [30]. The enhanced algorithm uses a lower bound (LB) based on relaxing the SOP into a Minimum-Cost Perfect Matching (MCPM) problem and then solving it using the dynamic Hungarian algorithm [36].

The sequential algorithm used in our work is based on the enhanced B&B algorithm proposed by Jamal et al. with Best-First Search (BestFS) [43] used to explore the solution space. BestFS is used because it experimentally gave better results for the SOP. The sequential algorithm is summarized in this sub-section. The details may be found in the original papers [46,30].

The sequential algorithm is based on a B&B approach that exhaustively explores the solution space by constructing an *enumeration tree*. Each leaf in the tree represents a complete feasible solution, and each internal node represents a partial solution. A solution is constructed incrementally by adding one vertex to the current partial solution. At each tree node, a partial path has been constructed by selecting a sequence of vertices. The subproblem to be solved at that node is finding the optimal order for the remaining vertices. The LBs of all possible next vertices at that node are computed and the vertex with the lowest LB is added to the current partial path. The feasible solutions in a subproblem's solution space (subspace) are the leaves of the sub-tree below that node. In this paper, tree nodes, subproblems, and subspaces are used interchangeably.

To speed up the search, pruning techniques are applied at each node. If a pruning technique indicates that no better solution than the current best solution can be found below the current node, the algorithm backtracks to the previous node, thus pruning the sub-tree below the current node.

The two pruning techniques used in the sequential B&B algorithm are history-based domination and the MCPM LB. History domination stores information about previously visited nodes in a history table and then uses them to quickly process *similar* nodes that are visited later. Two tree nodes are similar if their partial solutions (prefix paths) are permutations of the same set of vertices and they end with the same vertex. For example, a tree node with a prefix path ⟨A, B, C, D⟩ is similar to a tree node with a prefix path ⟨A, C, B, D⟩, because the nodes' partial paths are two different permutations of the same set of vertices {A, B, C, D} and they both end with Vertex D. The history table is implemented as a hash table. The second pruning technique is relaxing the SOP into a MCPM problem and then solving it using the dynamic Hungarian algorithm that runs in $O(n^2)$ time [36]. History domination is always applied before the dynamic Hungarian algorithm, because it is much faster.

3.3. Parallel algorithm overview

The proposed parallel B&B algorithm is a pool-based algorithm, in which Breadth-First-Search (BFS) is initially used to split the problem into smaller subproblems that are stored in a global pool. The subproblems (tree nodes) in the global pool are then assigned to threads using the scheme described in Section 3.4. Each thread explores the sub-tree below its assigned tree node in a BestFS order as in the sequential algorithm. When a thread completes exploring its assigned tree node, it is assigned a new node from the global pool. If the global pool is empty, a thread that has completed exploring its assigned node will steal part of the load of an active thread. The details are described in the next subsections.

3.4. Global-pool initial assignment

As mentioned above, BFS is used to split the tree nodes and store them in a global pool. First, the children of the root node (Depth 1) are placed in the pool. If the number of children is less than the number of threads, all the children will be split, and the nodes in the global pool will then be the root node's grandchildren (Depth 2). Splitting continues until we reach a depth at which the number of nodes in the pool is greater than or equal to the number of threads. Splitting is done for all the nodes at each depth to ensure that all the nodes in the global pool are always at the same depth.

Global pool nodes are assigned to threads in a manner that ensures *diversity* [35]. By diversity, we mean covering as many *primary subspaces* as possible and distributing the threads among the covered primary subspaces as fairly as possible. A primary subspace is a subspace that corresponds to an immediate child of the root node. If the number of threads is greater than the number of primary subspaces, threads are divided among primary subspaces as fairly as possible. Within each primary subspace, nodes are assigned to threads in ascending LB order. Nodes with smaller LBs are assigned first, because a smaller LB for a given node indicates that the subspace below that node is more promising.

Hence, our initial node assignment treats diversity as a primary criterion and LBs as a secondary criterion. We have experimented with both treating diversity as a primary criterion and treating the LB as a primary criterion, and the results were, on average, about the same. Two examples of tree splitting and global-pool initial assignment with four threads are shown in Fig. 2. Each leaf node is labeled by its LB.

In Fig. 1.a, the root node is initially split into five nodes: Nodes A through E. Since the number of nodes at that depth is greater than the number of threads, no further splitting is needed. Furthermore, since there are four threads and five primary subspaces, the most promising subspaces (based on LBs) will be assigned to

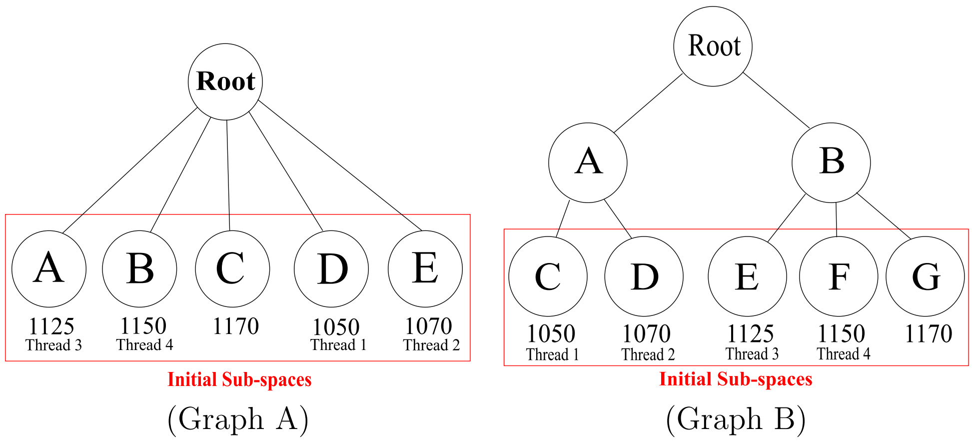

Fig. 2. Node splitting and global-pool initial assignment example.

threads, and the subspace of Node C, which has the highest LB, will not be assigned to any thread at that point.

In Fig. 1.b, the initial splitting only gives two nodes. Therefore, all the nodes at Depth 1 are split to produce five nodes at Depth 2. Since at that depth, the number of nodes is greater than the number of threads, no further splitting is needed. With two primary subspaces and four threads, each primary supspace will be covered by two threads. Within each primary subspace, nodes are assigned in ascending LB order. The first primary subspace (the sub-tree below Node A) has two nodes, and these are assigned to Threads 1 and 2. The second primary subspace (the sub-tree below Node B) has three nodes. The nodes with the smaller LBs (Nodes E and F) are assigned to Threads 3 and 4. Node G, which has the largest LB in that subspace, will not be assigned to a thread at that point. Nodes that are not initially assigned to threads remain in the global pool to be assigned later to the threads that complete exploring their assigned subspaces.

4. Load balancing and reassignment

4.1. Work stealing

Balancing the load among threads is necessary for utilizing all threads, and thus maximizing the speedup delivered by a parallel algorithm. Load balancing is particularly challenging in B&B, because the time taken by a thread to process a given subspace (tree node) depends on the number of feasible solutions (tree leaves) in that subspace and the degree of pruning that takes place while exploring it. The number of feasible solutions depends on precedence constraints, and the problem of counting the number of feasible solutions in the presence of precedence constraints is intractable [10]. The degree of pruning depends on the structure of the assigned subproblem, the quality of the solutions in its subspace, the search order and the effectiveness of the pruning techniques. Due to this combination of factors, it is not possible to predict the time that a thread will take to process a given tree node. Some threads may complete much faster than others. To address this problem, we have developed a dynamic load balancing technique.

On a high level, our approach to load balancing is based on the work-stealing strategy proposed by McCreesh and Prosser [35]. However, the low-level algorithmic details are based on our experimentation with alternative implementation methods. The details are as follows.

If a thread completes processing its assigned node, it becomes temporarily *idle*. An idle thread checks the global pool. If the global pool is non-empty, one of the nodes in the pool will be assigned to the idle thread. If the global pool is empty, the idle thread will steal work from one of the active threads.

Work stealing is implemented in our algorithm by maintaining a *local pool* of unprocessed tree nodes in each thread. Each thread's local pool holds the nodes that have been visited but have not been processed yet. Recall that within each thread, nodes are processed in a BestFS manner. At a given tree node, the LBs of that node's children are computed, and the child with the lowest LB (the most promising child) is explored first. The rest of the children are inserted into the thread's local pool to be used for work stealing.

When an idle thread requests work stealing, it steals work from a victim thread. A victim thread is randomly selected from the active threads that have non-empty local pools. We experimented with multiple schemes for selecting the victim thread, and random selection gave the best results. Once the victim thread has been selected, the idle thread takes a certain number of nodes from the victim thread's local pool. This number is computed using the formula $k(1 - p)$, where k is the number of available nodes in the victim thread's local pool, and p is the density of precedence constraints in the given instance. When the density of precedence constraints is higher, threads are likely to complete exploring stolen nodes faster, and thus more threads are likely to become idle within a given period of time. Therefeore, the precedence-constriant density is used in the above equation to limit the number of nodes stolen by a single idle thread and make it possible to satisfy the needs of more idle threads.

Ideally, the stolen nodes should have a sufficiently high load to keep the stealing thread active for a long period of time and avoid invoking work stealing too frequently. For the reasons explained above, it is not possible to precisely estimate the amount of time that will be taken to process a given node. Therefore, we chose to use the depth of a tree node as a simple indicator of the time needed to process a node, and a node is inserted into the local pool only if its depth is less than a certain threshold. The threshold is chosen based on experimentation.

The search is complete when the global pool and all local pools are empty, and when every thread has completely explored its assigned tree nodes, including unstolen nodes in its local pool.

4.2. Search order and thread restart

The order in which a B&B algorithm searches the solution space is an important factor that greatly affects the search speed. For an NP-hard problem, we shouldn't expect to find a perfect search order. However, some heuristics may be used to achieve a good search order that causes the algorithm to visit better subspaces earlier. In our sequential algorithm, the heuristic used to determine the search order is the MCPM LB.

If the search order of a parallel B&B algorithm is better than the sequential search order, super-linear speedup may be achieved, while if the parallel search order is worse, the speedup will be low. In the worst case, a poor parallel search order may cause the parallel algorithm to run slower that than the sequential algorithm (*detrimental anomaly* [34,16]).

The search order of our parallel algorithm is inherently different than that of the sequential algorithm, because the sequential algorithm strictly uses BestFS, while the parallel algorithm uses a combination of BFS and BestFS and includes a work-stealing technique that reassigns nodes to threads. Initially, the search order of the parallel algorithm is determined by the scheme used to assign global-pool nodes to threads.

Our experimental investigation has shown that a good scheme for assigning global-pool nodes to threads is not enough for achieving the best search order. A dynamic and adaptive technique is needed to guide the search. To maximize the benefit from parallelization, the parallel algorithm should intelligently take advantage of the information that becomes available about how promising each subspace is. Focusing the search on the most promising subspaces can greatly improve the parallel search order, and thus increase the chances of achieving super-linear speedup. In addition to focusing on the most promising subspaces known so far, the parallel algorithm should use some threads to explore new subspaces, as that may lead to discovering more promising subspaces.

To accomplish this, we designed a dynamic thread restart technique that uses multiple threads to balance the exploitation of promisingness information and the exploration of new subspaces. Our technique is, to some extent, similar to the restart technique proposed by Chu et al. [12]. However, the technique of Chu et al. uses different criteria for reassigning loads to threads. Their criteria are based on solution-density estimates and user-defined confidence, while our criteria are based on the LBs and the number of updates made to the global best solution.

Although our restart technique focuses on the most promising subspaces, it does not ignore other subspaces. It continues to search other promising subspaces and also explores new subspaces. Balancing exploitation and exploration is a unique feature that distinguishes our algorithm from previous parallel combinatorial algorithms, including the algorithm of Chu et al. [12].

The proposed algorithm temporarily abandons non-promising subspaces by moving the corresponding nodes back to the global pool. Then, the algorithm assigns the threads which were processing non-promising subspaces to either most promising subspaces or new subspaces that have not been explored yet.

In the proposed restart algorithm, there are two modes of operation: the *pre-update* mode and the *normal mode*. The modes differ in the way promising subspaces are identified. Initially, the algorithm is in the pre-update node, and it switches to the normal mode as soon as the first update is made to the global best solution.

In the pre-update mode, two global variables are used to track the lowest LB in any active thread and the depth at which this LB is found. The depth is updated whenever the same LB is found at a greater depth. Then, the number of updates that a thread makes to the depth of the lowest LB is used as a metric to measure how promising a thread's subspace is. The rational is that the LB of a given subspace is a good indicator of how promising that subspace is, and that a tighter LB is a stronger indicator of promisingness. Since the LBs found at greater depths are tighter, finding the lowest LB at a greater depth below a given node indicates a more promising subspace below that node.

Once a thread makes an update to the global best solution, the restart algorithm switches to the normal mode, in which the number of updates to the global best solution is used as the metric for measuring how promising a thread's sub-space is. Experimentally, this metric has been found to be the best indicator of promisingness.

In both modes, the number of updates (either to the depth of the lowest LB or to the global best solution) is measured within a certain period of time, called *the sampling period*. Each period defines a *restart cycle*. The search in the next restart cycle focuses on the most promising subspaces in the current restart cycle by assigning more threads to them.

In the pre-update mode, the threads assigned to the most promising subspaces share the computation of the LB, while in the normal mode, they share the search in those subspaces. This was based on the experimental observation that in the pre-update mode, the best use of parallelism is sharing the computation of the LBs in the most promising subspaces.

In both modes, each thread is classified into one of the following categories based on the number of updates that it made to either the best solution or to the depth of the lowest LB in the current sampling period.

Non-Promising: The thread did not make any update.

Promising: The thread made at least one update.

Most Promising: The thread is one of the top k threads in terms of the number of updates made to the best solution or to the depth of the lowest LB. Ties are broken in favor of the thread that made the most recent update. Usually, the most promising threads make multiple updates.

The parameter k is the number of most-promising subspaces that the search will focus on in the next cycle. Experimentally, the best results were achieved by setting k to four when the number of threads is 32, to two when the number of threads is 16, and to one when the number of threads is 8.

In the normal mode, promising threads continue to explore their subspaces in the next cycle. The most promising threads place part of their loads (some of the nodes that they have not explored) in the global pool, and these nodes are labeled *promising*. So, the global pool will have promising nodes and unexplored nodes (nodes that have never been assigned to threads).

Non-promising threads temporarily abandon their current nodes, move the current contents of their local pools to the global pool, and then take new nodes from the global pool. Some of the non-promising threads are assigned to promising nodes from the global pool, while the rest of the non-promising threads are assigned to unexplored nodes. If the global pool does not have any promising nodes, all non-promising threads will be assigned to unexplored nodes. Experimentally, the best results were obtained by assigning 50% of the non-promising threads to promising nodes (exploitation) and the other 50% to unexplored nodes (exploration).

Fig. 3 shows an example of thread restart in the normal mode. In this example, the total number of threads is eight, and k is set to one. The box representing each thread shows the number of global solution updates that the thread made during the sampling period. Threads 2, 4, 5, and 6 will be labeled "non-promising", because they did not make any updates to the global solution. Threads 1, 3, and 8 will be labeled "promising", because each of them made at least one update, and Thread 7 will be labeled "most promising", because it ranks first in terms of the number of updates.

Thread 7 will then place part of its load in the global pool's "promising" section. Assuming 50% exploitation and 50% exploration, two of the nonpromising threads (Threads 5 and 6) will be assigned to the promising nodes shared by Thread 7, and the other two (Threads 2 and 4) will be assigned to unexplored nodes from the global pool. Threads 1, 3, and 8, which are labeled "promising", will continue to explore their current subspaces.

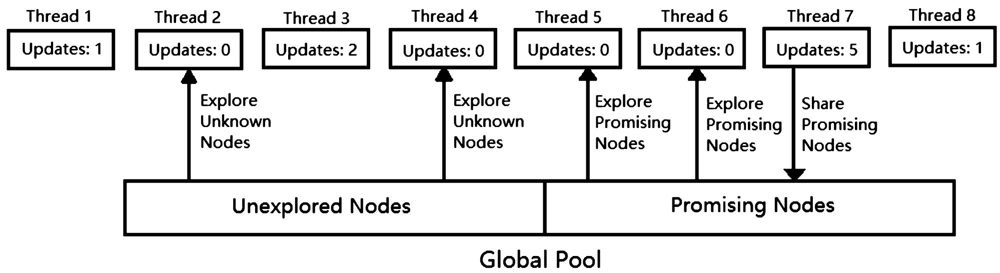

Fig. 3. Thread restart example.

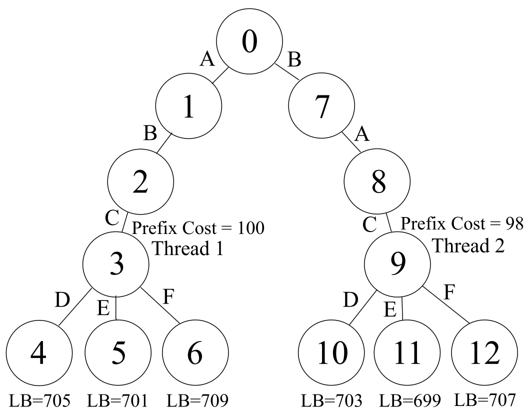

Fig. 4. Thread stop and resume example.

5. Parallelization of history domination

5.1. *Thread stop/resume technique*

When the enumeration tree is explored in parallel, multiple threads may explore the same sub-tree below similar tree nodes, which is redundant work that should be avoided if possible. Ideally, only the sub-tree below the most dominant tree node (the node with the lowest prefix cost) need to be explored. In reality, however, this ideal goal cannot be achieved exactly. In this subsection, we describe the algorithm that we have developed to minimize the overlap in exploring similar subspaces.

An example is shown in Fig. 4. In this example, tree Nodes 3 and 9 are similar, because their partial paths (⟨A, B, C⟩ and ⟨B, A, C⟩) are two different permutations of the same set of vertices (the set {A, B, C}) and both partial paths end with the same vertex (vertex C). Both Nodes 3 and 9 have the same sub-tree below them. The objective is to explore this sub-tree only once, or at least, avoid completely exploring it twice. In this case Node 9 dominates Node 3, because the prefix cost of Node 9 (the cost of partial path ⟨B, A, C⟩) is less than the prefix cost of Node 3 (the cost of partial path ⟨A, B, C⟩). Suppose that Node 3 is assigned to Thread 1 and Node 9 is assigned to Thread 2. In general, we cannot control the order in which Threads 1 and 2 are executed. So, both orders must be considered.

First, assume that Thread 2 starts executing before Thread 1. When Thread 1 starts executing, the sub-tree below Node 3 will be considered for exploration. As part of that consideration, our algorithm will search the history table for a node that is similar to Node 3, and Node 9 will be found. Applying the domination condition (comparing the prefix costs of Node 3 and Node 9) will show that Node 9 dominates Node 3. In this case, the right action to take is obvious. The sub-tree below a dominated node does not need to be explored. Therefore, our parallel algorithm will simply prune Node 3 and then assign a different node to Thread 1.

Now, assume that Thread 1 starts executing before Thread 2. When Thread 2 starts executing, the algorithm will search the history table for a node that is similar to Node 9, and Node 3 will be found. Applying the domination condition will show that the current node (Node 9) dominates the history node (Node 3). The right action in this case will depend on whether the exploration of the sub-tree below the dominated node (Node 3) has been completed or is still active. If it is still active, our parallel algorithm will stop Thread 1, because it is exploring a sub-tree below a dominated tree node. Implementing this thread stopping is non-trivial. It requires special data structures and low-level algorithmic details that are described next.

Every entry in the history table has a flag that indicates whether the exploration of the sub-tree below the corresponding tree node has been completed or is still active. This flag is set to *active*, when the algorithm starts exploring the sub-tree and is changed to *completed* when the algorithm backtracks to that node after exploring of all of its children. It should be noted that because of the reassignment techniques described in the previous section, different children may be explored by different threads, and this further complicates the implementation.

The implementation requires an active-tree data structure that links parents with their children even if they are processed by different threads. At a given point, a node is active if the sub-tree below it is currently being explored by one or more threads. Each active node has a link to the corresponding history entry along with its exploration state (whether the sub-tree below it has been fully explored). Due to work stealing, it is possible for the children of an active node to be explored by different threads. So, the active tree structure is used to update the history entry's exploration flag during the parallel exploration of the search tree.

A thread that is about to explore the sub-tree below a dominant node (Node 9 in our example) sends a stop request if the history entry of the dominated node is labeled *active*. The stop request is stored in a stop-request buffer. Each stop request contains the prefix of the history node (so that it can be found in the table) and the latest prefix cost (the prefix cost of the dominant node that is about to be explored).

Each active thread periodically checks the stop-request buffer for a similar prefix. If a thread finds a stop request with a similar prefix and a better prefix cost, it will invoke the stopping procedure to stop exploring its current sub-tree. The stopping procedure backtracks to the root of the sub-tree being explored (the dominated node). It also involves checking the global and local pools for any children of the dominated node and deleting them.

Stopping a thread that is exploring a dominated node is a powerful technique that allows the algorithm to avoid unnecessary exploration. However, stopping the exploration of the dominated node is not enough to minimize redundancy. A more efficient algorithm should take advantage of the pruning that has taken place while exploring the sub-tree below the dominated node. Some or all of that pruning may still be valid in the sub-tree below the dominant node. If we can prove that every pruning that has taken place below the dominated node is still valid below the dominant node, the exploration of the dominant node may pick up from where the exploration of the dominated node left off, instead of re-exploring the whole sub-tree from scratch.

In Fig. 4, the prefix cost of the dominant node (Node 9) is better than the prefix cost of the dominated node (Node 3) by 2 (100–98). This implies that every LB in the sub-tree below Node 9 is less than the LB of the corresponding node in the sub-tree below Node 3 by 2. So, if the LBs of Nodes 4 and 5 are 705 and 701 respectively, the LBs of Nodes 10 and 11 will be 703 and 699, respectively. Assuming that the global best cost when Nodes 4 and 5 were explored was 700, whether Nodes 10 and 11 can be pruned will depend on the global best cost when Nodes 10 and 11 are explored. If that best cost is 699 or less, both Nodes 10 an 11 can be pruned, while if that best cost remains 700, Node 10 can be pruned but Node 11 cannot.

Keeping track of the LBs of all the nodes that have been explored below the dominated node would require an excessive amount of memory, because the number of nodes in a sub-tree is an exponential function of the sub-tree's height. However, storing the lowest LB that was used in any pruning below a history node makes it possible to reuse pruning information with minimal additional memory. Every LB-based pruning that has taken place in the sub-tree below the dominated node will still be valid in the sub-tree below the dominant node if the following condition is satisfied:

$$LB_{min} - imp \geq best$$

where LB_{min} is the minimum LB used for any pruning below the dominated node, *imp* is the prefix cost improvement (prefix cost of the dominated node minus prefix cost of the dominant node), *best* is the current best cost. This technique is referred to as the *thread resume* technique. To implement this technique, the lowest LB used for pruning is propagated up the active tree and stored in the history entries for the sub-tree nodes.

The thread that is about to explore the dominant node sends a resume request and waits until it receives information about the current position of the exploration of the dominated node. The current position is represented by the current node's partial solution. Once this information has been received, the waiting thread can use it to prune any redundant sub-trees until its current partial solution matches the stopped thread's partial solution.

5.2. History table memory management

The history table is implemented as an array of buckets. Each bucket has a list of history entries, and each history entry is a pair that consists of a key and a history node. The history node contains atomic data including the cost of the corresponding prefix path, the LB at that node, and the ID of the thread that is currently exploring the sub-tree below that node. The main data structures used to implement the history table are shown in Fig. 5.

When many threads explore the solution space in parallel, nodes will be inserted in the history table at a faster rate. This has the advantage of providing more history information, and thus enabling more pruning. However, this advantage comes at the cost of increasing the table size at a faster rate, and thus exhausting memory in less time. Moreover, adding more entries to the history table results in more collisions, and that slows the search.

Ideally, the objective is using the available memory to store the most useful entries in the history table. A history table entry is a previously explored tree node. Two factors that determine the usefullness of an explored tree node are the node's depth and the size of the sub-tree that was enumerated below that node. A shallower node is more useful, because it makes it possible to prune a similar node earlier. A node with a larger sub-tree below it is more useful, because pruning based on that node will save more computation. To maximize useful entries in the history table, the algorithm sets a restriction on the nodes inserted into the history table after a certain percentage of the available memory has been used. Experimentally, the setting that gave the best results was 80% of available memory. When that percentage is used, the algorithm inserts a node in the history table only if the depth of the explored sub-tree below it is greater than zero. When available memory is used completely, no more nodes are inserted into the history table.

```
typedef pair<Bitvector,Int> Key;
typedef pair<Key,HistoryNode*> Entry;
typedef list<Entry> Bucket;

struct Content {
    int PrefixCost;
    int LowerBound;
};

struct HistoryNode {
    atomic<data> Content;
    atomic<bool> Explored;
    atomic<int> ActiveTID;
};

class HistoryTable {
    private:
        size_t TableSize;
        vector<Bucket*> Map;
        vector<Lock> TableLock;
        vector<MemModule> MemAllocator;
    public:
        void InitializeTable(size_t TableSize, size_t ThreadCount);
        void InsertNode(Key& Val, node& CurrentNode);
        HistoryNode* RetrieveNode(Key& Val);
};
```

Fig. 5. Simple illustration of the history table data structures.

5.3. Memory access and protection

The proposed parallel algorithm uses four main shared data structures, namely, the global pool, the global best solution found so far, the history table, and a local pool for each thread. These data structures are implemented using the C++ standard library. Locks and atomic variables from the C++ standard multi-threading library are used to synchronize access to these shared data structures, and various techniques are implemented to minimize the waiting time on locks as described below.

The global pool is locked during reading and writing to avoid conflicts when multiple threads try to access it at the same time. The local pool of each thread is also read-protected and write-protected during work stealing.

The best solution found so far is only write-protected, since reading an older value of the best solution does not affect correctness and only minimally affects performance. If a thread reads an older value of the best cost found so far, it may miss a pruning opportunity at the current node, but it will, most likely, make up for the missed opportunity within a short period of time by reading the updated value at the next node. Since reading the best solution occurs much more frequently than writing (updating), limiting locking to write accesses can give a significant performance gain.

The history table is both read-protected and write-protected. In the history table, every x adjacent buckets are protected by a lock during reading (history node retrieval) and writing (history node insertion), where x is a parameter that is currently set to 10. An

Table 1
Instance classification.

Benchmark Suite	Total Instances	Easy Instances	Medium Instances	Hard Instances
SOPLIB	48	13	19	16
TSPLIB	41	12	5	24

entry in the table is implemented as an atomic structure rather than a lock-protected structure to reduce the synchronization overhead during concurrent reads to the entry. This also reduces the overhead of acquiring locks via system calls. In addition to atomic structures, we have experimented with basic locks and reader-writer locks, and atomic structures gave the best results.

To minimize the system-call overhead, history table entries are allocated in blocks instead of allocating one entry at a time. This block-based allocation is managed by a memory allocation module, which is a small handwritten software layer that manages access to dynamically allocated memory with minimal system calls. This module allocates blocks of frequently allocated objects, such as history entries, and returns one object from that block to a requesting thread. When the current block is used completely, the memory allocation module allocates a new block, and so on. In the parallel algorithm, each thread has its own memory allocation module instead of having one global module for the entire program. This decision was made to minimize the synchronization cost. A shared global allocation module must be protected with a lock, and the time spent waiting on that lock may cause a significant performance degradation.

6. Experimental results

6.1. Experimental setup

The proposed parallel algorithm was tested on both the TSPLIB [42] and the SOPLIB [39] benchmark suites. Two different machines were used: the *main machine* that is owned by our group and has a 32-core AMD Threadripper 2990WX processor with 128 GB of memory and an AWS machine (c6g.8xlarge) that has a 32-core Graviton 2 ARM processor with 64 GB of memory. The operating system on both machines is Ubuntu 20.04.

The AWS machine has faster memory, while the main machine has larger memory capacity. The AWS machine was used for the shorter tests that measure speedup, while the main machine was used for the longer tests that measure cost improvements. As explained in Subsection 5.2, when a parallel B&B algorithm with history domination is run for a longer period of time, more memory is needed due to the growth of the history table.

The TSPLIB and SOPLIB instances were classified into *easy*, *medium*, and *hard* based on the sequential algorithm's running time. An easy instance is an instance that the sequential algorithm solves within 10 seconds. A medium-difficulty instance is an instance that the sequential algorithm solves in more than 10 seconds but in less than an hour. An instance that the sequential algorithm cannot solve within an hour is classified as a hard instance. Easy instances are not considered in this paper, as a parallel algorithm is not needed to solve these instances. The number of instances that belong to each difficulty category is shown in Table 1. The instances that the parallel algorithm could not solve in an hour on the AWS machine were run on the main machine using a time limit of five hours.

On both machines, the depth constraint for inserting nodes into the history table was triggered once 80% of memory is used. The sampling period for the restart algorithm was set to 200 seconds for TSPLIB and 25 seconds for SOPLIB. The depth threshold for inserting nodes into the local pool was set to 250 levels for SOPLIB and 150 levels for TSPLIB. We also note that only the thread stopping part of the algorithm described in Section 5.1 was used in the experimental evaluation, as the thread resume part did not work effectively due to the high overhead of tracking LBs.

Table 2
Speedup of medium-difficulty instances.

SOPLIB	8 Threads	16 Threads	32 Threads
Geo-mean speedup	27.8	42.9	71.9
Maximum speedup	88.9	187.9	366.3
Minimum speedup	5.6	9.7	15.8
Super-linear speedups	17	16	13
TSPLIB	**8 Threads**	**16 Threads**	**32 Threads**
Geo-mean speedup	7.3	13.4	19.5
Maximum speedup	27.1	61.3	116.2
Minimum speedup	3.7	6.4	7.1
Super-linear speedups	1	1	1

6.2. Medium-difficulty instances

The proposed parallel algorithm was applied to the medium-difficulty instances in SOPLIB and TSPLIB using 8, 16, and 32 threads with a time limit of one hour on the AWS machine. The speedup delivered by the proposed parallel algorithm relative to the sequential algorithm is shown in Table 2.

On SOPLIB, the geometric-mean speedup of the parallel algorithm relative to the sequential algorithm is super-linear for any number of threads. For example, with 32 threads, the geo-mean speedup across the medium SOPLIB instances is 71.9, which is super-linear, and the maximum speedup is 366.3. These results confirm the hypothesis that a parallel algorithm with history domination and a dynamic restart technique can deliver super-linear speedup on many instances. Out of 19 medium SOPLIB instances, superlinear speedup was achieved on 13 instances with 32 threads, on 16 instances with 16 threads, and on 17 instances with 8 threads.

The results in Table 2 show that there were some instances on which the speedup was lower than expected. For example, the lowest speedup with 32 threads is 15.8, which is significant but much smaller than the number of threads. This is primarily attributed to the search order. As explained in Section 4.2, it is unlikely to find a way of ensuring that the parallel search order is always at least as good as the sequential search order.

Comparing the geometric-mean speedup for the different numbers of threads shows that the proposed algorithm scales up very well when the number of threads is increased. The geometric-mean speedup is 27.8 with 8 threads, 42.9 with 16 threads, and 71.9 with 32 threads. The increase in the geometric-mean speedup is not perfectly proportional to the number of threads, because the overhead of our parallelization techniques increases as the number of threads is increased. For example, when the number of threads is increased, more threads will be requesting work stealing and thread restart. As a result, more time will be spent handling these requests and the threads will be spending more time waiting on the locks that protect shared variables.

The speedup ratios for the medium-difficulty TSPLIB instances in Table 2 are not as high as the SOPLIB speedup ratios. However, the TSPLIB speedup ratios scale up better than the SOPLIB speedup ratios, and superlinear speedup is achieved on one instance with 32, 16, and 8 threads.

6.3. Hard instances

The proposed algorithm was applied to the hard SOPLIB and TSPLIB instances using 8, 16, and 32 threads. The Speedup measurement tests with a one-hour time limit were run on the AWS

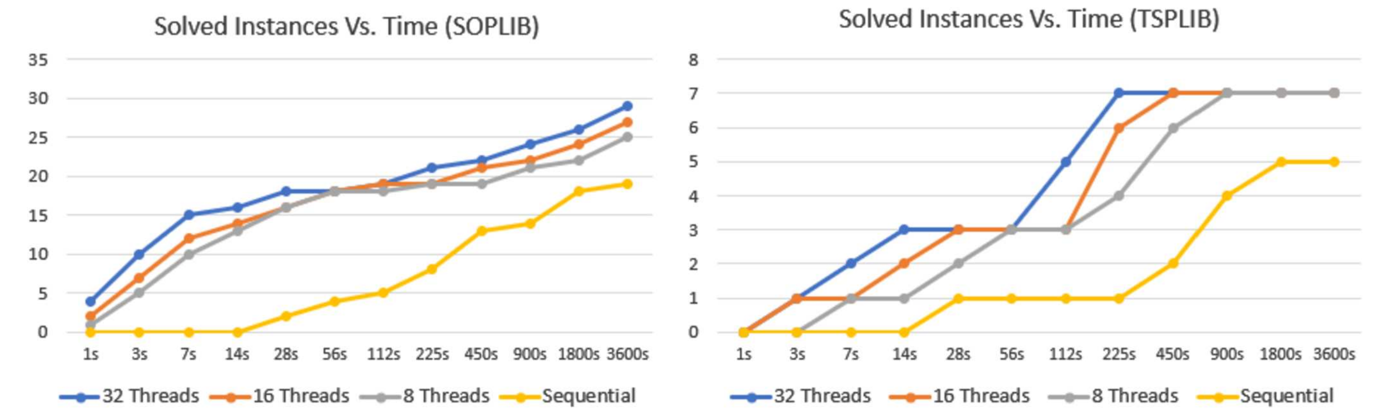

Fig. 6. Cumulative plots for the instances that are solved within an hour.

machine, while the cost-comparison tests and the 5-hour tests were run on the main machine. Recall that the AWS machine has faster memory, while the main machine has larger memory capacity. Table 3 shows the number of instances solved in each case.

With a time limit of one hour, the parallel algorithm with 32 threads solved 10 out of 16 hard SOPLIB instances and two out of 24 hard TSPLIB instances. Recall that these hard instances are the instances that the sequential algorithm could not solve on the same machine within an hour.

Fig. 6 shows the cumulative plots for all the instances (including medium and hard instances) that are solved optimally by the parallel algorithm within an hour. For example, the SOPLIB plot shows that the parallel algorithm solves much more instances than the sequential algorithm within the first 28 seconds. At the end of the one-hour period, the parallel algorithm solves 12 more SOPLIB instances than the sequential algorithm.

To further test the power of the proposed parallel algorithm, the hard instances that the parallel algorithm could not solve with 32-threads within an hour on the AWS machine were run on the main machine with a time limit of five hours. The results are shown in the last row in Table 3, with a five-hour time limit, the parallel algorithm solved 14 out of 16 hard SOPLIB instances and four out of 24 hard TSPLIB instances. Two SOPLIB instances and 20 TSPLIB instances remain unsolved. The number of unsolved TSPLIB instances is significantly greater than the number of unsolved SOPLIB instances, because most hard TSPLIB instances are less precedence constrained, and thus have larger solution spaces to explore.

Table 4 shows the speedup delivered by the parallel algorithm relative to the sequential algorithm with a time limit of one hour. Since, by definition, the sequential algorithm times out on hard instances, only a LB on the speedup can be computed for these instances. The ratio between the one-hour time limit and the parallel execution time is a LB on the actual speedup, because the sequential solution time is not known exactly but is certainly greater than the time limit.

On SOPLIB, the geometric-mean speedup LB is 4.5 with 8 threads, 8.4 with 16 threads, and 15.6 with 32 threads. This shows that the algorithm scales up very well on these instances. The highest speedup LB is 269.9, which shows the great potential of the proposed parallel algorithm.

On TSPLIB, the geometric-mean speedup ratios are significantly higher than the SOPLIB ratios, and the scaling is excellent. The largest speedup on the hard TSPLIB instances with 32 threads is 49.7, which is super-linear.

It is noted in Table 3 that some hard instances are not solved to optimality by the proposed algorithm. To study the effect of the parallel algorithm on these unsolved instances, we compared the final cost computed by the parallel algorithm with that computed by the sequential algorithm. Tables 5 and 6 show the results of this comparison for the hard instances that the parallel algorithm with 32 threads could not solve within an hour.

Table 3
Hard instances solved by the parallel algorithm.

	SOPLIB	TSPLIB
Total hard instances	16	24
Solved by 8 threads in one hour	6	2
Solved by 16 threads in one hour	8	2
Solved by 32 threads in one hour	10	2
Solved by 32 threads in five hours	14	4

Table 4
Speedup lower bounds on hard instances.

SOPLIB	8 Threads	16 Threads	32 Threads
Geo-mean speedup LB	4.5	8.4	15.6
Maximum speedup LB	233.6	217.1	269.9
Minimum speedup LB	1.1	1.9	3.7
Super-linear speedups	1	1	1
TSPLIB	**8 Threads**	**16 Threads**	**32 Threads**
Geo-mean speedup LB	7.3	16.9	31.6
Maximum speedup LB	10.7	27.1	49.7
Minimum speedup LB	5.0	10.5	20.0
Super-linear speedups	1	1	1

Table 5
Effect of the parallel algorithm on timed out hard instances.

Instance Group	Instances Studied	Instances Improved	Instances Regressed	Instances with No Improvement
SOPLIB	6	5	1	0
TSPLIB	21	20	0	1

The results in Table 5 show that although the parallel algorithm times out on 27 hard instances, it finds better solutions than the sequential algorithm on 25 instances. Table 6 shows that on SOPLIB, the average improvement is 14.5%, and the best improvement on any instance is 28.6%, while on TSPLIB, the average improvement is 4.8%, and the best improvement is 15.5%.

A slight regression of 0.4% is seen on one SOPLIB instance. This is attributed to the search order. As explained in Section 4.2, our restart algorithm is designed to achieve a better search order than the sequential algorithm, but, due to the heuristic nature of the algorithm, achieving a better search order is not guaranteed in all cases. These results show that a better search order is achieved on most but not all instances.

Table 6
Cost improvement by the parallel algorithm on timed out hard instances.

Instance Group	Avg Cost Improvement [%]	Max Cost Improvement [%]	Min Cost Improvement [%]
SOPLIB	14.5	28.6	−0.4
TSPLIB	4.8	15.5	0.0

Table 7
Random variation in the solution times of medium instances.

Instance Group	Random Variation in Geo-mean [%]	Highest Random Variation [%]	Lowest Random Variation [%]
SOPLIB	3.4	22.0	0.1
TSPLIB	1.1	14.4	1.1

Table 8
Effect of thread restart on the speedup of medium SOPLIB instances.

SOPLIB	Restart Enabled	Restart Disabled	Speedup Improvement
Instances studied	6	6	
Geo-mean speedup	21.9	14.5	51.0%
Maximum speedup	46.3	18.7	147.5%
Minimum speedup	15.8	7.8	102.6%

6.4. Random variation

In this sub-section, we study random variation in the proposed algorithm's solution time. Random variation is inherent to parallel algorithms, due to race conditions among threads and operation system intervention. To measure the amount of random variation in the solution times of the proposed algorithm, the algorithm with 32 threads was applied to the medium SOPLIB and TSPLIB instances 3 times on the AWS machine. Table 7 shows the percentage random variation in aggregate (the geo-mean) and at the instance level (the highest and lowest percentage variation on any instance). The percentage random variation is defined as the difference between the highest reading and the lowest reading as a percentage of the lowest reading.

At the instance level, the maximum random variation on any instance (the worst case) is 22.0%. In aggregate, random variation in the geo-mean is 3.4% for SOPLIB and 1.1% for TSPLIB. Random variation in the geo-mean is limited because random variations on individual instances tend to average out (positive variations cancel negative variations). It is important to emphasize here that the amount of random variation in the results is so limited that all the super-linear speedups reported in this paper would still be obtained even if the test was repeated many times.

6.5. Effect of thread restart

To test the effectiveness of the thread restart technique described in Section 4.2, we applied the proposed algorithm with thread restart disabled to two sets of instances. The first set is the set of medium-difficulty SOPLIB instances on which thread restart is activated, and the second set is the set of hard instances that the parallel algorithm could not close in an hour. In this test, we used 32 threads and a time limit of one hour.

Table 8 shows the speedup delivered by the parallel algorithm relative to the sequential algorithm with and without thread restart on the studied medium SOPLIB instances. The results show that enabling the thread restart technique increases the geometric-mean speedup on these instances by 51.0%, the maximum speedup by 147.5% and the minimum speedup by 102.6%.

Tables 9 and 10 show the effect of the thread restart technique on the hard instances that the parallel algorithm could not close within an hour. Table 9 shows that the thread restart technique improved the final cost for a total of 17 out of 27 timed out hard instances. Seven instances were negatively affected by the algorithm, while three instances were not affected. Regressions on some instances are unavoidable due to the heuristic nature of the technique. As explained earlier, it is possible to find a heuristic technique that gives a better search order in most cases, but it is unlikely to find a technique that gives a better search order in all cases.

Table 9
Effect of thread restart on timed out hard instances.

Instance Group	Instances Studied	Instances Improved	Instances Regressed	Instances with No Improvement
SOPLIB	6	3	2	1
TSPLIB	21	14	5	2

Table 10
Effect of thread restart on the final cost of timed out hard instances.

Instance Group	Avg Cost Improvement [%]	Max Cost Improvement [%]	Min Cost Improvement [%]
SOPLIB	6.7	16.3	−0.5
TSPLIB	0.7	8.4	−5.0

Table 11
Effect of thread stop on the speedup of medium instances.

SOPLIB	Stop Enabled	Stop Disabled	Speedup Imp
Geo-mean speedup	71.9	42.9	67.6%
Maximum speedup	366.3	112.3	226.2%
Minimum speedup	15.8	14.1	12.1%
TSPLIB	**Stop Enabled**	**Stop Disabled**	**Speedup Imp**
Geo-mean speedup	19.5	18.7	4.3%
Maximum speedup	116.2	84.7	37.2%
Minimum speedup	7.1	7.1	0%

The cost improvements achieved when the thread restart technique is enabled are shown in Table 10. The results in this table show that the thread restart technique produces better final costs on average. The average cost improvement is 6.7% on SOPLIB and 0.7% on TSPLIB. The best improvement is 16.3% on SOPLIB and 8.4% on TSPLIB.

6.6. Effect of thread stop

To test the effectiveness of the thread stop technique described in Section 5.1, the parallel algorithm with 32 threads and a one-hour time limit was applied to all the medium-difficulty instances with thread stop disabled. Table 11 shows the speedup delivered by the parallel algorithm relative to the sequential algorithm on the medium instances in SOPLIB and TSPLIB with and without thread stop. The last column in each table shows the percentage improvement in speedup when thread stop is enabled.

On SOPLIB, enabling the thread stop technique increases the geometric-mean speedup by 67.6% and increases the highest speedup by 226.2%. On TSPLIB, enabling thread stop increases the geometric-mean speedup by 4.3% and increases the highest speedup by 37.2%. These results clearly show the effectiveness of the thread stop algorithm.

6.7. Optimality proving vs solution improving

To test the optimality-proving performance, as opposed to the solution improving performance, of the proposed algorithm, we ran a test on the medium-difficulty instances with the optimal solution fed as an initial solution into the sequential and the parallel algorithm. Table 12 shows the improvements made by the

Table 12
Improvement with an optimal initial solution.

SOPLIB	Geo-mean	Max	Min
Improvement			
Solution Time	45.6	140.4	10.0
Enumerated Nodes	9.9	46.9	1.0
Node Processing Speed	4.6	21.0	0.2
TSPLIB	**Geo-mean**	**Max**	**Min**
Improvement			
Solution Time	21.0	59.5	8.9
Enumerated Nodes	1.0	1.7	0.8
Node Processing Speed	21.5	36.1	10.8

parallel algorithm with 32 threads relative to the sequential algorithm in three different metrics: the total solution time, the number of enumerated nodes, and the node processing speed (defined as nodes processed per second). The improvements in the table are shown as ratios (sequential-to-parallel ratio for the first two metrics where smaller is better, and parallel-to-sequential ratio for the third metric where larger is better). The experiment was performed on the AWS machine.

Overall, the results in Table 12 show that the proposed algorithm delivers significant improvements relative to the sequential algorithm even when the initial solution is optimal in both cases (parallel and sequential). For example, on average, the parallel algorithm proved the optimality of the initial solution 45.6 times faster than the sequential algorithm on the SOPLIB instances. Some speedup comes from the search order, but significant improvement in node processing speed is also achieved. On average, the improvement in the total solution time is approximately equal to the product of the improvement in enumerated nodes and the improvement in node processing speed.

It is important to note that when the parallel algorithm makes a greater improvement (reduction) in the number of enumerated nodes, the improvement in node processing speed will be smaller, because when the parallel algorithm makes more aggressive pruning, it will be processing shallower nodes. Shallower nodes take more time to process, as the LBs need to be computed for larger sub-graphs. Therefore, node processing speeds are comparable only when the number of enumerated nodes is about the same. This is the case for the TSPLIB instances, where the number of enumerated nodes is, on average, the same for both the parallel and the sequential algorithm. In that case, the node processing speed of the parallel algorithm is 21 times faster than the node processing speed of the sequential algorithm. This is a reasonably good speedup, considering the overhead of the parallel algorithm, especially shared-memory contention, cache conflicts and waiting on locks.

7. Conclusions and future work

In this paper, we propose a parallel B&B algorithm with history-based domination and apply it to the SOP. The proposed algorithm with 32 threads solves 10 SOPLIB instances and two TSPLIB instances that the sequential algorithm does not solve in an hour. It gives super-linear speedup on 16 instances with a maximum speedup of 366x on a 32-core processor.

In future work, we will continue to work on enhancing the proposed parallel algorithm, and we will explore a GPU version of it. Furthermore, we plan on extending the proposed parallel approach to solve the instruction scheduling problem in compilers [48,49], which is somewhat similar to the SOP but involves additional challenges and complexities.

Table 13
Effect of the percentage of stolen load on performance.

Benchmarks	Geo-mean Speedup			
	25%	50%	100%	Dynamic
SOPLIB	61.2	55.1	51.2	71.9
TSPLIB	20.4	20.0	20.0	19.5

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work was supported in part by the US National Science Foundation (NSF) Award No 1911235. This work benefited from technical discussions with Jafar Jamal and Jeffrey Byrnes. The authors also thank Patrick Brannan and Lynne Koropp for the technical support that they provided. Finally, we thank the anonymous reviewers for their thorough and insightful comments that greatly improved the final version of this paper.

Appendix A. Work stealing

To study the relation between the amount of stolen load and performance, the parallel algorithm was applied to the medium-difficulty instances on the AWS machine with a time limit of one hour using three different percentages of stolen load: 25%, 50% and 100%. The results are shown in Table 13. The table shows the geometric-mean speedup achieved using the parallel algorithm for each percentage of stolen load.

The results in the table show that on SOPLIB, stealing all the load gives the worst overall results. By examining the individual instances, we observed that there is a relation between an instance's precedence-constraint density and the load percentage that gives the best results for that instance. When the density of precedence constraints is higher, stealing fewer nodes gives better results. This is attributed to the fact that with a higher precedence density, work stealing occurs more frequently, and thus more threads are likely to become idle and request work stealing at about the same time. In this case, better performance is achieved when a victim thread's load is distributed among multiple idle threads instead of giving all of that load to one idle thread.

Based on this observation, we experimented with dynamically setting the percentage of stolen load based on an instance's density of precedence constraints. As shown in the last column of Table 13, this gave better results on than any fixed percentage on SOPLIB. This is the scheme that was used to generate the results in the paper. On TSPLIB, performance is not sensitive to the percentage of stolen nodes. Approximately the same results are achieved using any percentage of stolen load.

Appendix B. Thread restart period

We experimented with the effect of the restart period on the performance of the restart technique. The test was performed on the hard instances that could not be solved within an hour by the proposed parallel algorithm with 32 threads on the main machine. Table 14 shows the percentage improvement in the overall best cost found within an hour using thread restart periods of 25s, 50s,

Table 14
Cost improvement of the restart algorithm with different restart periods.

Benchmarks	Cost improvement			
	25 s	50 s	100 s	200 s
SOPLIB	6.7%	5.9%	5.2%	3.9%
TSPLIB	0.0%	−0.1%	−0.5%	0.7%

Table 15
Cost improvement of the restart algorithm with different percentages of exploitation.

Benchmarks	Cost Improvement		
	20%	50%	80%
	Exploitation	Exploitation	Exploitation
SOPLIB	4.4%	6.7%	6.9%
TSPLIB	−0.6%	0.7%	−0.3%

100s, and 200s. The cost improvements in the table are relative to the costs obtained with thread restart disabled.

On SOPLIB, using a short restart period of 25s gives the best overall results, while on TSPLIB, the longest period of 200s gives the best overall results. This is attributed to the fact that updates to the best solution happen at a much higher rate in SOPLIB. When updates to the best solution happen infrequently, a longer restart period is needed to identify the most promising supspaces.

Appendix C. Balancing exploitation and exploration

Table 15 shows the results of our experimentation with the balance between exploitation and exploration in the restart technique. The test was performed on all the instances that the parallel algorithm with 32 threads could not solve within an hour on the main machine using three different percentages of exploitation: 20%, 50%, 80%). The cost improvements in the table are relative to the costs obtained with thread restart disabled.

The results in the table show that dividing the threads equally between exploitation and exploration gives the best overall results. On SOPLIB, however, doing exploitation in 80% of the threads gives slightly better results.

References

[1] D. Anderson, G. Hendel, P.L. Bodic, M. Viernickel, Clairvoyant restarts in branch-and-bound search using online tree-size estimation, in: Proceedings of the AAAI Conference on Artificial Intelligence, 2019, pp. 1427–1434.
[2] D. Anghinolfi, R. Montemanni, M. Paolucci, L. Gambardella, A hybrid particle swarm optimization approach for the sequential ordering problem, Comput. Oper. Res. 38 (2011) 1076–1085.
[3] B. Archibald, P. Maier, C. McCreesh, R. Stewart, P. Trinder, Replicable parallel branch and bound search, J. Parallel Distrib. Comput. 113 (2018) 92–114.
[4] B. Archibald, F. Dunlop, R. Hoffmann, C. McCreesh, P. Prosser, J. Trimble, Sequential and parallel solution-biased search for subgraph algorithms, in: Integration of Constraint Programming, Artificial Intelligence, and Operations Research, 2019, pp. 20–38.
[5] N. Ascheuer, L. Escudero, M. Grötschel, M. Stoer, A cutting plane approach to the sequential ordering problem (with applications to job scheduling in manufacturing), SIAM J. Optim. 3 (1993) 25–42.
[6] N. Ascheuer, M. Jünger, G. Reinelt, A branch & cut algorithm for the asymmetric traveling salesman problem with precedence constraints, Comput. Optim. Appl. 17 (2000) 61–84.
[7] D. Bader, W. Hart, C. Phillips, Parallel algorithm design for branch and bound, in: Tutorials on Emerging Methodologies and Applications in Operations Research, in: International Series in Operations Research & Management Science, vol. 76, 2005.
[8] L. Barreto, M. Bauer, Parallel branch and bound algorithm - a comparison between serial, OpenMP and MPI implementations, J. Phys. Conf. Ser. 256 (2010).
[9] A. Borisenko, S. Gorlatch, Optimal batch plants design on parallel systems: a comparative study, in: IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2019, pp. 549–558.
[10] G. Brightwell, P. Winkler, Counting linear extensions is #p-complete, in: Proceedings of the ACM Symposium on Theory of Computing, 1991, pp. 175–181.
[11] I. Chakroun, N. Melab, Towards a heterogeneous and adaptive parallel branch-and-bound algorithm, J. Comput. Syst. Sci. 81 (2015) 72–84.
[12] G. Chu, C. Schulte, P.J. Stuckey, Confidence-based work stealing in parallel constraint programming, in: Principles and Practice of Constraint Programming, 2009, pp. 226–241.
[13] T. Crainic, B. Le Cun, C. Roucairol, Parallel branch-and-bound algorithms, in: Parallel Combinatorial Optimization, 2006, pp. 1–28.
[14] A. Dabah, A. Bendjoudi, A. AitZai, D. El-Baz, N. Taboudjemat, Hybrid multi-core CPU and GPU-based b&b approaches for the blocking job shop scheduling problem, J. Parallel Distrib. Comput. 117 (2018) 73–86.
[15] M. Daouri, F. Escobar, X. Chang, C. Valderrama, A hardware architecture for the branch and bound flow-shop scheduling algorithm, in: 2015 Nordic Circuits and Systems Conference (NORCAS): NORCHIP International Symposium on System-on-Chip (SoC), 2015, pp. 1–4.
[16] A. de Bruin, G. Kindervater, H. Trienekens, Asynchronous parallel branch and bound and anomalies, in: Parallel Algorithms for Irregularly Structured Problems, in: Lecture Notes in Computer Science, vol. 980, 1995, pp. 363–377.
[17] L. Escudero, An inexact algorithm for the sequential ordering problem, Eur. J. Oper. Res. 37 (1988) 236–249.
[18] L. Escudero, M. Guignard, K. Malik, A Lagrangian relax-and-cut approach for the sequential ordering problem with precedence relationships, Ann. Oper. Res. 50 (1994) 219–237.
[19] L. Gambardella, M. Dorigo, An ant colony system hybridized with a new local search for the sequential ordering problem, INFORMS J. Comput. 12 (2000) 237–255.
[20] L. Gambardella, R. Montemanni, D. Weyland, An enhanced ant colony system for the sequential ordering problem, in: Operations Research Proceedings 2011, 2012, pp. 355–360.
[21] B. Gendron, T. Crainic, Parallel branch-and-bound algorithms: survey and synthesis, Oper. Res. 42 (1994) 1042–1066.
[22] J. Gmys, Heterogeneous cluster computing for many-task exact optimization - Application to permutation problems, Ph.D. thesis, Université de Mons (UMONS); Université de Lille, 2017.
[23] J. Gmys, M. Mezmaz, N. Melab, D. Tuyttens, A GPU-based branchand-bound algorithm using integer-vector-matrix data structure, Parallel Comput. 59 (2016) 119–139.
[24] J. Gmys, M. Mezmaz, N. Melab, D. Tuyttens, A computationally efficient branch-and-bound algorithm for the permutation flow-shop scheduling problem, Eur. J. Oper. Res. 284 (2020) 814–833.
[25] L. Gouveia, P. Pesneau, On extended formulations for the precedence constrained asymmetric traveling salesman problem, Networks 48 (2006) 77–89.
[26] L. Gouveia, M. Ruthmair, Load-dependent and precedence-based models for pickup and delivery problems, Comput. Oper. Res. 63 (2015) 56–71.
[27] F. Guerriero, M. Mancini, A cooperative parallel rollout algorithm for the sequential ordering problem, Parallel Comput. 29 (2003) 663–677.
[28] K. Helsgaun, An extension of the Lin-Kernighan-Helsgaun TSP Solver for constrained traveling salesman and vehicle routing problems, Technical Report, Roskilde Universitet, 2017.
[29] J. Herrera, J. Salmerón, E. Hendrix, R. Asenjo, L. Casado, On parallel branch and bound frameworks for global optimization, J. Glob. Optim. 69 (2017) 547–560.
[30] J. Jamal, G. Shobaki, V. Papapanagiotou, L. Gambardella, R. Montemanni, Solving the sequential ordering problem using branch and bound, in: IEEE Symposium Series on Computational Intelligence, 2017, pp. 1–9.
[31] V. Janakiram, E. Gehringer, D. Agrawal, R. Mehrotra, A randomized parallel branch-and-bound algorithm, Int. J. Parallel Program. 17 (1988) 277–301.
[32] J. Kinable, A. Ciré, W. van Hoeve, Hybrid optimization methods for time-dependent sequencing problems, Eur. J. Oper. Res. 259 (2017) 887–897.
[33] M.E. Lalami, D. El-Baz, GPU implementation of the branch and bound method for knapsack problems, in: IEEE International Parallel and Distributed Processing Symposium Workshops PhD Forum, 2012, pp. 1769–1777.
[34] G. Li, B. Wah, Coping with anomalies in parallel branch-and-bound algorithms, IEEE Trans. Comput. C-35 (1986) 568–573.
[35] C. McCreesh, P. Prosser, The shape of the search tree for the maximum clique problem and the implications for parallel branch and bound, ACM Trans. Parallel Comput. 2 (2015) 8:1–8:27.
[36] G. Mills-Tettey, A. Stentz, M. Dias, The dynamic Hungarian algorithm for the assignment problem with changing costs, Technical Report, Carnegie Mellon University, 2007.
[37] A. Mingozzi, L. Bianco, S. Ricciardelli, Dynamic programming strategies for the traveling salesman problem with time window and precedence constraints, Oper. Res. 45 (1997) 365–377.
[38] M. Mojana, R. Montemanni, G. Caro, L. Gambardella, A branch and bound approach for the sequential ordering problem, in: Proceedings of the International Conference on Applied Operational Research, in: Lecture Notes in Management Science, vol. 4, 2012, pp. 266–273.
[39] R. Montemanni, D. Smith, L. Gambardella, A heuristic manipulation technique for the sequential ordering problem, Comput. Oper. Res. 35 (2008) 3931–3944.
[40] D. Morrison, S. Jacobson, J. Sauppe, E. Sewell, Branch-and-bound algorithms: a survey of recent advances in searching, branching, and pruning, Discrete Optim. 19 (2016) 79–102.

[41] J. Pekny, D. Miller, A parallel branch and bound algorithm for solving large asymmetric traveling salesman problems, Math. Program. 55 (1992) 17–33.
[42] G. Reinelt, TSPLIB—a traveling salesman problem library, INFORMS J. Comput. 3 (1991) 376–384.
[43] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, 4th edition, Pearson, 2020.
[44] Y. Salii, Revisiting dynamic programming for precedence-constrained traveling salesman problem and its time-dependent generalization, Eur. J. Oper. Res. 272 (2019) 32–42.
[45] Y. Salii, A. Sheka, Improving dynamic programming for travelling salesman with precedence constraints: parallel Morin–Marsten bounding, Optim. Methods Softw. (2020) 1–27.
[46] G. Shobaki, J. Jamal, An exact algorithm for the sequential ordering problem and its application to switching energy minimization in compilers, Comput. Optim. Appl. 61 (2015) 343–372.
[47] G. Shobaki, K. Wilken, Optimal superblock scheduling using enumeration, in: Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture, 2004, pp. 283–293.
[48] G. Shobaki, M. Shawabkehand, N. Rmaileh, Preallocation instruction scheduling with register pressure minimization using a combinatorial optimization approach, ACM Trans. Archit. Code Optim. 10 (2013) 31.
[49] G. Shobaki, A. Kerbow, S. Mekhanoshin, Optimizing occupancy and ILP on the GPU using a combinatorial approach, in: Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization, 2020, pp. 133–144.
[50] R. Skinderowicz, An improved ant colony system for the sequential ordering problem, Comput. Oper. Res. 86 (2017) 1–17.
[51] C. Tomazella, M. Nagano, A comprehensive review of branch-and-bound algorithms: guidelines and directions for further research on the flowshop scheduling problem, Expert Syst. Appl. 158 (2020) 113556.
[52] S. Tschoke, R. Lubling, B. Monien, Solving the traveling salesman problem with a distributed branch-and-bound algorithm on a 1024 processor network, in: Proceedings of the International Parallel Processing Symposium, 1995, pp. 182–189.

Taspon Gonggiatgul is a research assistant at California State University, Sacramento. He received a bachelor's degree in Computer Engineering from University of California, Davis. He is interested in the parallization of combinatorial optimization algorithms along with other topics in the area of embedded systems.

Ghassan Shobaki is an Associate Professor in the Computer Science (CS) dept. at California State University, Sacramento. He received his Ph.D and Master of Science degrees in CS from the University of California, Davis in 2002 and 2006, respectively. He also holds a Master of Science degree in Electrical Engineering (EE) from the University of Houston and a B.Sc degree in EE from the University of Jordan.

Dr. Shobaki's research interests are in the areas of compiler optimizations, combinatorial optimization algorithms and parallel computing. He is generally interested in developing exact or precise algorithms for solving NP-hard combinatorial optimization problems. More specifically, he is interested in applying combinatorial optimization techniques to compiler optimization problems and studying their performance compared to the heuristic approaches used in production compilers.

Pınar Muyan-Özçelik is an Associate Professor at the Department of Computer Science of the California State University, Sacramento. She earned her Ph.D degree in Computer Science from the University of California, Davis. She received her MSc degree from the Department of Computer Science at the University of British Columbia and completed her B.Sc degree in Computer Engineering at the Ege University. She is interested in the parallelization of combinatorial optimization algorithms and benchmarking autonomous driving systems. She also conducts research on the use of GPU computing in various domains such as compiler optimizations, electric power systems, automotive computing, embedded systems, and medical imaging.

Sponsor names

Do not correct this page. Please mark corrections to sponsor names and grant numbers in the main text.

US National Science Foundation, *country*=United States, *grants*=1911235

Highlights

- We propose the first parallel B&B algorithm that involves a history-based domination technique.
- The proposed algorithm includes three novel parallelization techniques: thread restart, parallel history-based domination, and history-table memory management.
- The proposed algorithm is the first parallel B&B algorithm for the Sequential Ordering Problem, which is an NP-hard sequencing problem with precedence constraints.
- We present a thorough experimental evaluation of the proposed parallel algorithm on SOPLIB and TSPLIB. The results show that super-linear speedup is not an anomaly; it can be achieved on many instances. We report a speedup ratio as high as 285 on a 32-core processor.