CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Scalable Parallel Branch-and-Bound with Relaxed Concurrent Queues

A Study of Relaxed Concurrent Queues in Branch-and-Bound using a Generic Solver Framework

Master's thesis in Computer science and engineering

LUDVIG SANDH

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

Scalable Parallel Branch-and-Bound with Relaxed Concurrent Queues

A Study of Relaxed Concurrent Queues in Branch-and-Bound using a Generic Solver Framework

LUDVIG SANDH

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Scalable Parallel Branch-and-Bound with Relaxed Concurrent Queues
A Study of Relaxed Concurrent Queues in Branch-and-Bound using a Generic Solver Framework
LUDVIG SANDH

Supervisor: Kåre von Geijer, CSE
Examiner: Philippas Tsigas, CSE

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Scalable Parallel Branch-and-Bound with Relaxed Concurrent Queues
A Study of Relaxed Concurrent Queues in Branch-and-Bound using a Generic Solver Framework
LUDVIG SANDH
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Relaxed concurrent queues improve scalability by weakening strict ordering guarantees in exchange for reduced synchronisation overhead and contention. While previous work has demonstrated strong performance for such data structures in isolation, their applicability within complete algorithms and practical workloads remains less explored. This thesis investigates the use of relaxed concurrent queues, particularly the MultiQueue and related relaxed concurrent data structures, in branch-and-bound (BnB) algorithms. A literature study identifies BnB as a suitable application domain due to its tolerance for non-strict exploration order, after which a generic modular framework for parallel BnB solving is developed. The framework separates problem-specific logic from parallel execution and queue management, greatly simplifying the development of high-performing branch-and-bound solvers. The framework is evaluated primarily through a case study on the Maximum Clique Problem using DIMACS benchmark instances on a 512-thread multicore system, alongside demonstrations on knapsack variants. The experimental results show that relaxed concurrent designs generally scale better than strict concurrent alternatives under high contention, with relaxed queues significantly outperforming strict concurrent counterparts while maintaining competitive search quality. Solvers using our framework with relaxed concurrent data structures sometimes outperform competitors, showing the value and further potential of our framework. The results demonstrate that data-structure-level parallelism based on relaxed concurrent queues can compete with algorithm-level parallelisation strategies in several cases, particularly at very high thread counts.

Keywords: computer science, engineering, branch-and-bound, MultiQueue, relaxed concurrent priority queues, parallel computing, benchmarking, scalability

Acknowledgements

I would like to thank my supervisor, Kåre von Geijer, for his guidance, feedback, and support throughout this thesis project. I also thank my examiner, Philippas Tsigas, for valuable input and discussions. Finally, I would like to thank my family and friends for their support during the course of this work.

Ludvig Sandh, Gothenburg, 2026-07-21

Contents

List of Figures

List of Tables

1

Introduction

The end of frequency scaling in the early 2000s has shifted performance improvements toward parallel computing [1], [2], [3]. As a result, efficiently utilising multicore hardware has become a central challenge in modern computing. However, designing scalable parallel programs remains difficult due to synchronisation and communication overheads.

Priority queues are a key component in many parallel algorithms, particularly in search, scheduling, and optimisation problems. In concurrent settings, strict priority queues impose global ordering constraints on their operations [4], [5], which introduce serialisation points and limit scalability.

Relaxed concurrent priority queues address this limitation by weakening ordering guarantees in exchange for improved scalability [6]. Rather than always returning the globally minimal element, these structures allow deviations from strict priority order to reduce contention.

One prominent example is the MultiQueue [5], [7], a relaxed concurrent priority queue that achieves high throughput by distributing elements across multiple internal queues. While prior work has demonstrated strong performance at the data-structure level, there is still limited understanding of how relaxed concurrent queues behave when integrated into complete algorithms and real workloads. This motivates a systematic investigation of their applicability and trade-offs in practice.

The first aim of this thesis is to identify algorithmic domains in which relaxed concurrent priority queues can be applied effectively without compromising correctness under relaxed ordering semantics. This is addressed through a literature study, presented in Chapter 3. From this investigation, BnB algorithms emerge as a particularly suitable candidate, since they can tolerate deviations from strict exploration order while still producing correct solutions [8].

Existing parallel BnB solvers are often specialised to individual problems and tightly couple parallelisation mechanisms with problem-specific logic. Based on this insight, the thesis, secondly, aims to develop a generic modular framework for solving BnB problems using relaxed queues. The proposed framework separates concerns by supporting interchangeable problem definitions, queue implementations, and solver configurations, simplifying the development and evaluation of parallel BnB solvers. Finally, the project aims to use the framework to instantiate a range of BnB solvers

and benchmark them against state-of-the-art approaches. In particular, the Maximum Clique Problem serves as the primary case study for deeper analysis.

Although the project initally focused on relaxed concurrent *priority queues*, the scope grew to cover relaxed concurrent *stacks* as well, hence the use of the general term *queues* throughout the report.

The study is guided by the following research questions:

- **RQ1:** Which algorithmic domains can benefit from relaxed concurrent priority queues while tolerating deviations from strict priority ordering?
- **RQ2:** How do different forms of queue relaxation and search-order strategies influence performance and exploration behaviour in branch-and-bound algorithms?
- **RQ3:** Can branch-and-bound solvers using relaxed concurrent queues as the primary source of parallelism compete with state-of-the-art algorithm-level parallelisation strategies?

2

Background

This chapter provides background on the architectural and algorithmic context that motivates the thesis work, and explains the relevance of studying relaxed concurrent queues. It first outlines the limitations of frequency scaling and the resulting need for efficient parallel programming techniques. It then introduces the role of priority queues in parallel algorithms and discusses why strict concurrent priority queue semantics do not scale on modern multicore hardware. These limitations motivate studying whether relaxed priority queues can be used effectively inside real parallel algorithms. Lastly, this chapter explains branch-and-bound, and presents three problems that will be implemented and evaluated in the context of relaxed queues in this thesis.

2.1 Parallelism and the Limits of Frequency Scaling

Since the early 2000s, computers have started experiencing the practical limits of frequency scaling [1], [2]. Today, engineers can no longer rely on consistent yearly increases in CPU frequency to obtain performance improvements [3]. To effectively scale the performance of systems in today's multicore era, one must turn to the field of parallel programming [1]. In an ideal scenario, programs can be perfectly distributed onto multiple chips to run in parallel. This would promise an increase in compute performance directly proportional to the number of compute units involved. In reality, however, designing programs that execute efficiently on parallel systems can be highly complicated. This is due to overheads such as communication and synchronisation between threads. For example, some parts of a program must be executed in mutual exclusion, hence the need to synchronise. Since some data structures are commonly used in a wide range of programs, a great deal of work has been put into designing efficient concurrent versions of them [9]. Such designs can be used to increase overall throughput, reduce latency, and in general improve performance-critical systems further.

2.2 Priority Queues

One of the most foundational data structures is the priority queue [10]. It is commonly used in algorithms that require dynamically keeping track of elements with

associated priorities. Examples of such algorithms are priority scheduling [4] and graph searching [11]. Two key operations are provided by the priority queue: `insert` and `deleteMin`. `insert` inserts a key-value pair into the queue, and `deleteMin` extracts the key-value pair with the smallest key from the queue. In the case of an empty queue, `deleteMin` can return a sentinel value that indicates emptiness. The key acts as the priority for the corresponding value, and without loss of generality, the smallest key is considered to be of highest priority. In fact, one can define the priority direction arbitrarily, even without modification to the underlying data-structure implementation: by negating all keys before insertion and after deletion, a `deleteMin` operation can be made to behave as a delete-max operation. Classical priority queue implementations such as binary heaps offer $O(\log n)$ time for insertion and deletion [10]. More advanced designs such as Fibonacci heaps [12] achieve amortised $O(1)$ insertion, but tend to perform worse in practice compared to binary heaps.

2.3 Relaxed Concurrent Priority Queues

The problem of efficiently supporting priority-queue operations under high levels of concurrency has been studied extensively, and existing work shows that strict, fully linearisable [13] concurrent priority queues cannot scale on modern multicore hardware [4], [5]. The fundamental reason is that strict priority-queue semantics impose a global ordering constraint on `deleteMin` operations: each extraction must return the globally smallest element with respect to a single total order. Enforcing a globally minimal delete operation creates serialisation points that limit parallel execution, as the correctness of each operation depends on the outcome of all preceding extractions. As shown by Attiya et al. [14] and Ellen et al. [15], such global ordering constraints induce synchronisation costs that cannot be eliminated by finer-grained locking or lock-free techniques. Consequently, strict concurrent priority queues suffer from high contention and limited parallelism, making scalable performance unattainable in practice.

Relaxation weakens the semantic guarantees of the data structure in order to reduce synchronisation and improve scalability [6]. By relaxing the specifications of the priority queue, it is possible to design highly efficient and scalable concurrent priority queues [5], [7], [16], [17], [18], [19], [20]. This relaxation may cause the `deleteMin` operation to return an incorrect element: one that exists in the queue but does not correspond to the highest priority. Whether such relaxation is acceptable depends on how priority ordering is used by the surrounding algorithm, a question discussed in more detail in Section 2.7.

2.4 The MultiQueue Data Structure

A range of concurrent priority queue designs address this scalability problem through different underlying structures. Skip-list based approaches retain a single global structure but reduce contention around it in different ways. The SprayList [16]

performs deletions via a biased random walk down the levels of a skip list instead of always removing the leftmost element, bounding the rank error at $O(p \log^3 p)$ with high probability, although insertions can still cause heavy contention near the head of the list. The Contention Avoiding Priority Queue (CAPQ) [17] instead switches dynamically between a centralised skip list and thread-local insertion and deletion buffers depending on the level of contention, guaranteeing a rank error of $O(p^2)$ only once every $\Theta(p)$ operations, with no guarantee for the remaining fraction of operations served from local buffers. The k-LSM [18] combines thread-local log-structured merge-trees with a shared global structure that threads periodically synchronise with, bounding the rank error by $p \cdot k$ for a local buffer of size k, though this does not account for the periods during which the shared structure is inaccessible due to merging, which can lead to large delays in practice.

Other designs move away from skip lists and heaps entirely. Building on the same load-balancing idea as the MultiQueue, the Multi Bucket Queue [20] distributes elements across multiple lock-protected bucket queues, each grouping same-priority elements into contiguous buffers, avoiding the logarithmic per-operation overhead of skip-list- or heap-based designs altogether.

The relaxed concurrent priority queue studied most extensively in this project is the MultiQueue [5], [7]. The MultiQueue is a state-of-the-art relaxed concurrent priority queue, and has been shown to perform well when other designs struggle. It is designed to achieve high scalability on multicore systems by weakening strict priority ordering guarantees. Instead of maintaining a single globally ordered structure, the MultiQueue consists of multiple independent sequential priority queues, referred to as sub-queues.

Insert operations place elements into a randomly chosen sub-queue. `deleteMin` operations sample a small number of sub-queues (typically two) and return the minimum element among their local minima. Each sub-queue is protected by a lock, allowing concurrent operations on different sub-queues while avoiding global synchronisation. In both operations, sub-queue selection is retried until an unlocked sub-queue is successfully acquired. To ensure that operations can typically acquire a sub-queue without blocking, the number of sub-queues is chosen to be at least as many as the number of concurrent threads operating on the MultiQueue.

The quality of a relaxed priority queue is commonly measured in terms of *rank error*, defined as follows: if a `deleteMin` operation returns an element of rank r among all elements currently present in the queue, the rank error of the operation is $r - 1$. In other words, the rank error quantifies how many elements with strictly higher priority remain in the queue when a deletion occurs.

Rank error describes how far a single deletion deviates from the ideal choice, but it does not show how long high-priority elements may have to wait. Another commonly used measure is therefore *delay*. The delay of an element is the number of `deleteMin` operations that happen after the element has effectively become the minimum, but before it is actually removed. In other words, delay measures how long an important element is postponed while other elements are processed first. Together, rank error and delay describe both how accurate individual deletions are and how fairly high-

priority elements are treated over time [7].

The concept of a randomised selection strategy for deletions significantly reduces contention and enables scalable parallel performance, at the cost of slight rank error and delay. The effectiveness of this design is closely related to the *power-of-choice* principle [21], which shows that sampling a small number of candidates (most notably two) is sufficient to obtain strong bias toward high-priority elements while dramatically reducing contention compared to global selection. Theoretical analyses show that the expected rank error of a deletion is bounded by $\Theta(n)$, where n denotes the number of internal queues, and can be characterised exactly in the long run [22]. In particular, for the standard two-choice MultiQueue, the expected rank error converges to $\frac{5}{6}n - \Theta(1)$. Moreover, concentration bounds show that the maximum rank error observed over long execution sequences is $O(n \log n)$ with high probability [22].

Several design parameters influence the performance–quality trade-off of the MultiQueue. These include the number of sub-queues relative to the number of threads (known as the queue factor), the choice of internal sequential priority queue, and optimisations such as buffering and stickiness introduced in later work [7]. These parameters affect both contention behaviour and the degree of priority relaxation, making the MultiQueue a flexible but non-trivial building block for parallel algorithms.

Because the MultiQueue exposes controlled and quantifiable relaxation while providing high throughput, it serves as a natural candidate for studying how relaxed priority semantics interact with algorithm-level correctness and performance.

While prior work [23] has demonstrated that relaxed concurrent priority queues such as the MultiQueue can achieve high scalability, existing research has focused primarily on data-structure design and performance evaluation in isolation. Comparatively less attention has been given to understanding how these relaxed structures behave when integrated into full algorithms and application-level workloads. For many parallel algorithms, strict priority ordering is assumed but not always required for correctness, suggesting that controlled relaxation may offer performance benefits without compromising acceptable algorithmic behaviour.

2.4.1 Development of the MultiQueue Design

The MultiQueue was introduced in late 2014 [24] as a preprint by Rihani, Sanders, and Dementiev, and was later published at SPAA 2015 [5], marking the first formal presentation of the data structure. Their paper proposes the concept of building a relaxed concurrent priority queue from multiple sequential priority queues, where each operation propagates to a randomly sampled internal queue. When the number of sub-queues exceeds the number of threads, this approach yields probabilistically wait-free insertions and deletions. Simulations of this design show high scalability and significantly reduced contention [5] compared to previous skip-list-based relaxed structures such as the SprayList [16].

Williams, Sanders, and Dementiev extend this line of work in 2021 by publishing

a full-length paper in ESA [7]. Their perspective addresses practical shortcomings of the original MultiQueue, most notably poor cache locality and the cost of random queue accesses. They introduce three separate improvements: buffering, bulk operations, and stickiness. Buffering augments each sequential sub-queue with an insertion buffer and a deletion buffer; the deletion buffer stores the smallest elements of the sub-queue, so deleteMin operations can usually access this small buffer instead of the underlying priority queue. This reduces cache-line accesses and improves locality, while occasional refills and flushes are handled in batches. Bulk operations process multiple elements at once, amortising the cost of queue accesses across several insertions or deletions. Stickiness biases threads toward repeatedly accessing the same subset of local queues instead of selecting queues uniformly at random for every operation. A stickiness value determines for how many consecutive operations a thread continues using the same queue choices before new random queues are selected. This improves cache locality and reduces contention, while still preserving the relaxed semantics of the data structure.

The authors also explore different strategies in their more recent journal paper [23] for remapping threads to sub-queues after a stickiness period expires. In particular, they compare a simple randomised reassignment strategy with a swap-based strategy designed to reduce contention between threads accessing the same sub-queues simultaneously. These improvements aim to increase locality while maintaining relaxed semantics. They also present an expanded quality analysis that incorporates both rank error and delay, the latter measuring how long high-priority elements can be overtaken by lower-priority ones. Their evaluations [7], [23] show that the engineered MultiQueue offers a favourable throughput-quality trade-off and outperforms competing approaches such as the CAPQ [17], SprayList [16], and k-LSM [18] on modern hardware.

2.5 The MultiFIFO Data Structure

Similar to the MultiQueue, the MultiFIFO [25] data structure also employs relaxation by allowing deletes to return elements slightly out of order. However, the difference is that with the MultiQueue, ordering is based on priority of the keys, whereas with the MultiFIFO, ordering is based on insertion timestamps. During insertions, a timestamp is recorded and saved along with the element to be used later for comparisons during deletions. The MultiFIFO works as a concurrent FIFO queue by using a set of internal sequential FIFO queues, all guarded by their own lock, just like the MultiQueue uses internal sequential priority queues. Within the internal queues, the order is correctly maintained, and the relaxation comes from deleting from an internal queue that does not necessarily contain the globally oldest element.

2.6 The 2D-stack Data Structure

The 2D-stack was first introduced as a brief announcement in 2018 [26] and later generalised into a broader framework for continuously relaxing concurrent data structures [27]. It is a relaxed lock-free concurrent stack that improves scalability by weakening strict LIFO semantics. Instead of always returning the globally most recently inserted element, a pop operation may remove one of several recently inserted elements while maintaining deterministic bounds on the amount of relaxation.

Like several other relaxed concurrent data structures, the 2D-stack distributes operations over multiple independent sub-stacks [27]. Each sub-stack is implemented as a lock-free Treiber stack, and operations on different sub-stacks can therefore proceed independently, substantially reducing contention compared to a single shared stack.

The defining feature of the 2D-stack is that relaxation is controlled along two independent dimensions. The *width* parameter specifies the number of sub-stacks, thereby increasing the number of independent access points available for concurrent operations. Larger widths improve disjoint-access parallelism by spreading operations across more sub-stacks, reducing contention between threads. The *depth* parameter instead controls the amount of work that may accumulate within each sub-stack before operations are directed elsewhere. This second dimension exploits locality by allowing a thread to continue operating on the same sub-stack while contention remains low, thereby improving cache locality without requiring permanent thread-to-stack assignments [27].

To coordinate these two dimensions, the algorithm maintains a global operational window that determines which sub-stacks are currently eligible for push and pop operations. Threads first attempt to continue operating on their previously successful sub-stack. If contention is encountered, they randomly select another sub-stack, allowing successful threads to retain locality while unsuccessful threads migrate to reduce contention. If no suitable sub-stack is found, the global window is shifted, continuously trading stronger stack semantics for higher throughput [27].

Unlike many previous relaxed stack designs, which primarily exploit either disjoint-access parallelism or locality, the 2D-stack combines both mechanisms within the same algorithm. The authors prove that the resulting data structure remains lock-free and linearizable with respect to the k-Out-of-Order stack semantics, while experimental evaluations demonstrate higher throughput and better observed relaxation quality than previous relaxed stack designs over a wide range of relaxation levels [27].

2.7 Tolerance to Relaxed Priority Semantics

While relaxed concurrent priority queues improve scalability by weakening ordering guarantees, their suitability depends strongly on how priority ordering is used by the surrounding algorithm. Different algorithms rely on priority queues in fundamentally different ways: in some cases, strict priority ordering is essential for correctness,

whereas in others it primarily influences efficiency and convergence speed [23]. As a result, the effects of relaxing priority semantics must be understood in the context of the specific algorithm in which the priority queue is embedded.

A classical example where strict priority ordering is required is Dijkstra's shortest-path algorithm [11]. Dijkstra's correctness relies on the fact that vertices are processed in non-decreasing order of tentative distance, which guarantees that once a vertex is extracted from the priority queue, its shortest-path distance is final. If elements are extracted out of order, this invariant is violated and the algorithm may produce incorrect results. Similarly, optimal variants of the A* algorithm [28] depend on strict priority ordering to justify early termination when the target node is first extracted. In these settings, relaxed priority queues cannot be substituted directly without altering the algorithm's correctness guarantees.

At the same time, several relaxed or asynchronous variants of shortest-path and graph-search algorithms have been proposed that tolerate deviations from strict priority order. These algorithms typically compensate for out-of-order processing by allowing vertices to be relaxed multiple times and by modifying termination conditions. An example is the Relax and Don't Stop framework [29], which presents a highly competitive parallel solution to the single-source shortest path problem. The approach incorporates MultiQueue-based scheduling and adapts dynamically to graph structure, demonstrating that relaxed priority queues can achieve strong performance in a full algorithmic setting.

Similarly, the Wasp [30] algorithm shows that asynchronous shortest-path algorithms can tolerate priority drifting by relying on repeated relaxations and global convergence rather than strict extraction order. Wasp introduces priority relaxation on demand through a priority-aware work-stealing mechanism, allowing lower-priority work to be processed only when no higher-priority work is available, thereby limiting redundant work while preserving correctness. Similarly, recent work on parallel A* and point-to-point shortest-path algorithms shows that strict priority ordering can be weakened at the algorithmic level to improve scalability, provided that termination and pruning conditions are carefully redesigned. In particular, Dong et al. propose Orionet [31], a parallel framework for point-to-point shortest paths that processes vertices in parallel batches rather than in strict priority order, while ensuring correctness through global bounds and convergence criteria. In such algorithms, relaxation primarily increases the amount of work performed but does not compromise the correctness of the final result.

Other algorithmic paradigms tolerate relaxed priority semantics more naturally. Branch-and-bound search is a prominent example. It is described in Section 2.8, and the reason behind its relaxation tolerance can be found in Section 3.6.

These examples illustrate that the applicability of relaxed concurrent priority queues should be evaluated at the algorithmic level rather than solely at the data-structure level. Understanding how different algorithms respond to relaxed priority semantics is therefore essential for identifying application domains where relaxed priority queues such as the MultiQueue can be integrated effectively to improve parallel performance.

2.8 Branch-and-Bound

In branch-and-bound (BnB) search [8], [32], a problem is broken up into a series of decisions. These decisions can be modelled as a decision tree, where every node corresponds to a partial solution and each branch represents a decision on how to advance to the next partial solution. Leaf nodes correspond with complete solutions. Solving a BnB problem involves searching through such a tree until finding the leaf node corresponding to the best solution. The nature of the best solution depends on the problem, but is typically about minimising or maximising an objective function.

The term BnB refers both to the branching of decisions and to the use of bounds for pruning the search space [8], [32]. Assume, without loss of generality, that a given BnB problem is a maximisation problem. At any node, an upper and lower bound on the best solution in the node's subtree can be computed. Also, the maximum previously seen value, known as the incumbent, is maintained. Whenever a new best solution is found, the incumbent is updated with the new best solution. If the upper bound computed at a node does not exceed the incumbent, there is no need to search in that node's subtree, since the upper bound tells us none of the complete solutions are better than what is already found. On the other hand, if the computed lower bound exceeds the incumbent, it can be updated to the lower bound value, since the actual solution is guaranteed to be at least as good as that. When all nodes have been explored or pruned away, the algorithm is done and the incumbent corresponds to the global solution. The tighter the lower and upper bound calculations are, the more pruning can be done, leading to a more efficient solution since fewer nodes have to be visited [33].

An important observation is that since the algorithm runs until there is nothing left to explore, the search (starting from the root) can be performed in any order [32]. Correctness is ensured by bounding conditions and globally shared incumbent solutions rather than by strict exploration order. There are two main search orders typically used, each with its own benefits and drawbacks. A depth-first search prioritises traversing the tree deeply before backtracking, thus finding a leaf node and hence an incumbent early, which can be used to quickly eliminate parts of the search tree. A best-first search can compare the upper bounds on all children in the search front and prioritise the most promising node, potentially guiding the search in the right direction. Since the bounds are approximate, it is generally difficult to determine which search order will be most efficient. However, a major benefit of the DFS strategy is that its memory footprint stays bounded proportional to the maximum tree depth. With best-first search, the entire search front will be stored, usually in a priority queue, and can grow until the program runs out of available memory. On the other hand, a best-first exploration order may visit fewer nodes overall, leading to less work and potentially faster solutions. Many practical BnB solvers incorporate depth-first search to control memory usage [34], [35], [36], [37].

In the following sections, a few example BnB problems are described, as they give relevant context for the rest of the report, in particular Chapter 6 where their implementation is described.

2.8.1 0–1 Knapsack Problem

The 0–1 knapsack problem is a classical combinatorial optimisation problem and a canonical example of a problem solved using BnB techniques [38]. Given a set of n items, where each item i has a value v_i and a weight w_i, and a capacity constraint C, the objective is to select a subset of items maximising total value while satisfying the capacity constraint:

$$\max \sum_{i=1}^{n} v_i x_i \quad \text{s.t.} \quad \sum_{i=1}^{n} w_i x_i \leq C, \quad x_i \in \{0, 1\}.$$

The search space can be represented as a binary decision tree, where each level corresponds to deciding whether to include or exclude a particular item. BnB algorithms explore this tree while pruning subproblems using bounds derived from relaxations. A standard upper bound is obtained from the fractional knapsack relaxation, where items may be taken fractionally, yielding a tight and efficiently computable bound [38]. A lower bound can be computed by greedily including items until the capacity constraint would be violated.

2.8.2 Multidimensional Knapsack Problem

The multidimensional knapsack problem (MDKP) generalises the 0–1 knapsack problem by introducing multiple resource constraints [38]. Each item i is associated with a value v_i and a vector of weights $(w_{i1}, \ldots, w_{im})$, and the objective is to maximise total value subject to m capacity constraints:

$$\max \sum_{i=1}^{n} v_i x_i \quad \text{s.t.} \quad \sum_{i=1}^{n} w_{ij} x_i \leq C_j \;\; \forall j \in \{1, \ldots, m\}, \quad x_i \in \{0, 1\}.$$

Compared to the single-constraint case, MDKP generally exhibits weaker relaxations. Bounding is often performed by solving surrogate or one-dimensional relaxations, or by combining multiple fractional bounds, which increases computational cost and reduces pruning effectiveness [38].

From a BnB perspective, MDKP preserves the same binary branching structure as the 0–1 problem but introduces a more challenging trade-off between bound quality and computational overhead.

2.8.3 Maximum Clique Problem

The maximum clique problem [39] is a classical combinatorial optimisation problem defined on an undirected graph. Given a graph $G = (V, E)$, the goal is to find the largest subset of vertices $C \subseteq V$ such that every pair of vertices in C is connected by an edge, i.e., C forms a clique. The problem is known to be NP-hard, and it has been studied extensively due to its theoretical significance and applications in areas such as bioinformatics, social network analysis, and pattern recognition.

Exact algorithms for maximum clique commonly rely on BnB techniques [35], [39]. The search space consists of subsets of vertices, and partial solutions are extended

by adding vertices that are adjacent to all vertices in the current clique. The incumbent stores the size of the largest clique found so far, and bounding is used to prune subproblems that cannot lead to larger cliques than the incumbent. This combination of recursive search and pruning makes the problem a natural candidate for BnB formulations.

The open-source dataset DIMACS [40] is commonly used to evaluate maximum clique solvers, since it contains multiple graphs with different characteristics, ranging from easy to still unsolved.

3

Candidate Domains

A central goal of this thesis is to identify application domains and algorithms in which relaxed concurrent priority queues, and the MultiQueue in particular, can be integrated effectively. This chapter presents the domain study and its findings, and explains why BnB was chosen for further work. If the reader is interested in learning about the conclusions regarding other potential domains, and how this study eventually led to the choice of BnB, this chapter will likely be of interest. Otherwise, the reader can continue to Chapter 4.

Several candidate domains were considered based on two main criteria: whether the application can tolerate deviations from strict priority ordering without compromising correctness, and whether relaxed concurrent priority queues are likely to provide meaningful scalability or performance benefits in practice. This section discusses several potential application areas and their relevance, and assesses their feasibility within the scope of the thesis. Among these, BnB emerges as the most promising domain and will serve as the primary focus of the subsequent work. The Maximum Clique Problem is selected as the main BnB problem for benchmark evaluation, while, for example, knapsack variants are selected for implementation and demonstration only.

3.1 Graph Traversal and Search

Graph traversal and graph search algorithms constitute one of the most common evaluation domains for concurrent and relaxed priority-queue designs [19], [20], [29], [30], [31], [41], [42]. Many classical graph algorithms, including Dijkstra's shortest-path algorithm and A*, rely on priority queues to dynamically maintain a set of vertices ordered by tentative distance or estimated cost [11]. As a result, priority queues form a central performance bottleneck when these algorithms are parallelised, making graph processing a natural testbed for relaxed concurrent priority queues.

This domain has already been studied extensively in the context of relaxed priority semantics. Several parallel and asynchronous shortest-path algorithms weaken strict priority ordering in order to improve scalability, while compensating for out-of-order processing through repeated relaxations or modified termination conditions. Examples include multi-bucket [20] and Δ-stepping [41] approaches, relaxed priority schedulers such as Wasp [29], and parallel A* variants that process vertices in batches rather than strictly in priority order such as Orionet [31]. Recent work [19],

[30] has also explored priority-aware work stealing and graph-specific optimisations to further reduce redundant work.

The MultiQueue itself has already been applied successfully in this domain as a building block for parallel Dijkstra-style algorithms, demonstrating that controlled priority relaxation can improve scalability without compromising correctness. More recently, the Wasp [30] framework proposes a different approach to asynchronous single-source shortest-path computation, showing that priority relaxation can be introduced more selectively through a priority-aware work-stealing scheme rather than through a relaxed priority queue. Wasp claims to outperform the MultiQueue-based parallel Dijkstra implementations by introducing priority drifting only when no higher-priority work is locally available, thereby reducing redundant work while maintaining high thread occupancy. Related work [42] has also explored relaxed or approximate priority mechanisms in specialised settings such as FPGA-based routing, further demonstrating that shortest-path computations can tolerate controlled deviations from strict priority order under appropriate conditions.

Because of this substantial body of existing work, graph traversal and search algorithms are less attractive as the primary focus of this thesis. Simply substituting the MultiQueue into a standard shortest-path or A* implementation is unlikely to yield novel results, and achieving meaningful contributions would require either new algorithmic insights or substantial domain-specific adaptations. Nevertheless, this domain remains useful as a case study on how relaxed data structures have successfully been integrated to yield higher-performance and more scalable algorithms.

3.2 Priority-based Task Scheduling

Priority-based scheduling is a natural application area for concurrent priority queues: tasks are continuously inserted with an associated priority (e.g., deadline, estimated remaining work, or algorithm-specific key), and worker threads repeatedly extract the currently most urgent task. A single shared strict priority queue is conceptually simple, but in highly concurrent settings it quickly becomes a scalability bottleneck, as discussed in the background section. Consequently, practical schedulers often relax strict global ordering by using distributed run queues (typically one per worker) combined with load balancing or work stealing. For example, modern operating-system schedulers maintain per-CPU scheduling structures and periodically rebalance across CPUs to reduce contention and improve locality [43].

Recent research has shown that relaxed priority queues can be used effectively as the core abstraction in such schedulers. In particular, Postnikova et al. [19] demonstrate that MultiQueue-based designs can achieve state-of-the-art performance for graph traversal priority scheduling by combining local priority ordering with randomised selection and work stealing. A key insight of their work is that providing probabilistic rank guarantees enables bounding the amount of additional work induced by relaxed scheduling, allowing high throughput without excessive wasted computation in practice.

For this thesis, the question is whether there remains a meaningful research gap

beyond existing MultiQueue-based schedulers. One promising direction is workload sensitivity: scheduling workloads differ widely in priority distributions, task granularity, and locality requirements, and the optimal relaxation parameters (e.g., number of sub-queues, deletion choice count, or stickiness) may vary substantially across domains. Another open aspect is end-to-end impact: many prior evaluations focus on scheduler throughput and abstract quality metrics, whereas the relevant outcome is application-level behaviour such as tail latency, convergence time, or total work under realistic workloads. Therefore, priority-based scheduling is viable, but only with carefully chosen scope.

3.3 Discrete Event Simulation

Discrete event simulation (DES) models system evolution as a sequence of timestamped events [44]. A simulator maintains an *event queue* of scheduled future events (typically a priority queue ordered by timestamp) and repeatedly processes the event with the smallest timestamp, advancing the simulation clock and potentially generating new future events that represent causal consequences.

Parallel discrete event simulation (PDES) distributes the model across multiple *logical processes* (LPs). Each LP is responsible for a subset of the state and maintaining its own local event queue [44]. The central difficulty is enforcing *causality*: events must be processed in non-decreasing timestamp order *per causal dependency*, even though different LPs execute concurrently and dependencies are data-dependent and difficult to predict. Two main synchronisation families dominate the PDES literature. *Conservative* approaches avoid causality violations by only processing events that are provably safe, using mechanisms such as lookahead and (in classical formulations) null-message schemes to prevent deadlock [45], [46]. *Optimistic* approaches (e.g., Time Warp) allow LPs to process events speculatively; if a causality violation is later detected, the simulator rolls back state and cancels erroneously generated events [47]. A large body of work studies the performance trade-offs between these approaches at scale; for example, Carothers and Perumalla observe PDES has produced over 5000 publications and evaluate conservative vs. optimistic synchronisation on large supercomputer platforms [48]. This highlights that PDES has been extensively studied, suggesting that new contributions must carefully identify niches beyond existing synchronisation frameworks.

These causality requirements make DES a demanding test case for evaluating relaxed priority semantics. The MultiQueue (or any relaxed priority queue) can return events out of timestamp order, which in general corresponds to a causality violation and can change the outcome of the simulation. Consequently, relaxed priority queues are *not* a drop-in replacement for the timestamp-ordered event queue in general-purpose DES. However, they may still be useful in *structured* settings where limited reordering is semantically acceptable or can be compensated for. One possible direction is to explore models with bounded time-window tolerance, where events whose timestamps differ by at most a small Δ can be processed in any order without affecting observable results. Alternatively, relaxed execution can be paired with explicit correction mechanisms similar to optimistic rollback, where causality vi-

olations are detected and repaired at runtime. This suggests the following feasibility assessment: relaxed priority queues could be well suited for DES only when paired with an algorithmic mechanism that (i) restricts relaxed deletions to a proven-safe window, or (ii) detects and repairs causality violations, so that relaxation primarily trades additional work and bookkeeping for higher throughput.

3.4 Data Compression

Data compression algorithms provide a classical example of priority-queue usage, most notably in the construction of Huffman codes. Huffman coding [49] is a greedy algorithm that assigns variable-length binary codes to symbols such that the total encoded size is minimised, given known symbol frequencies. After counting symbol frequencies, the algorithm inserts them into a priority queue and repeatedly merges the two least frequent entries into a new node with combined weight, which is then reinserted into the queue. This process continues until a single tree remains, defining the optimal prefix code. Throughout the construction, the priority queue enforces the strict frequency ordering required for optimality.

The reliance on strict priority ordering makes Huffman coding a challenging domain for relaxed priority queues. Extracting elements out of order can lead to suboptimal merge decisions and, consequently, to codes that are no longer optimal in terms of expected code length. There is no natural correction mechanism that can compensate for out-of-order deletions without fundamentally altering the algorithm. As a result, relaxed priority queues cannot be substituted directly into the classical Huffman construction while preserving its optimality guarantees.

Nevertheless, this domain remains of some exploratory interest. In settings with extremely high throughput requirements or approximate compression goals, it may be acceptable to trade compression efficiency for construction speed. For example, in large-scale or streaming scenarios where encoding speed dominates and slight increases in compressed size are tolerable, relaxed priority semantics could potentially be employed to accelerate tree construction. However, reasoning about the impact of relaxation on average-case compression quality is non-trivial, and formal guarantees would be difficult to establish.

Parallel construction of Huffman trees has been studied previously, typically through algorithmic restructuring rather than relaxed priority queues. For instance, existing work on parallel Huffman tree construction [50] achieves sublinear depth and polylogarithmic factors by reorganising the merge process and exploiting parallel prefix techniques, while still enforcing strict ordering where required. These approaches suggest that parallelism in data compression is better addressed through specialised algorithmic designs than through relaxed concurrent priority queues. Consequently, while data compression illustrates the limits of relaxation tolerance, it is unlikely to serve as a primary application domain for this thesis, though small-scale experiments may still provide useful contrast.

3.5 Relaxed Priority Queues on GPUs

Graphics Processing Units (GPUs) offer massive parallelism through a single-instruction, multiple-thread (SIMT) execution model, in which groups of threads execute in lock-step and achieve high throughput when memory accesses and control flow are regular [51]. While GPUs excel at data-parallel workloads with predictable access patterns, synchronisation and fine-grained control dependencies are comparatively expensive, making it challenging to map pointer-heavy or highly irregular data structures efficiently to GPU architectures.

Priority queues are a particularly poor fit for GPUs in their classical form. The `deleteMin` operation is inherently sequential, and tree-based implementations rely on irregular memory accesses and frequent global synchronisation. Even relaxed concurrent priority queues such as the MultiQueue retain internal sequential priority queues and depend on randomised sub-queue selection, which introduces irregular access patterns and control divergence that are poorly suited to GPU execution. In particular, the "power-of-two-choices" strategy used in the MultiQueue requires sampling and comparing multiple candidate queues, an operation that maps poorly to SIMD-style execution and exacerbates branch divergence.

As a result, many parallel priority-based designs avoid fine-grained tree structures and instead rely on bucketed formulations. Bucket queues exploit the fact that selecting a bucket is a simple arithmetic operation, allowing many threads to process elements in parallel with minimal synchronisation. This approach is central to parallel shortest-path algorithms such as Δ-stepping [41], where priorities are grouped into ranges that can be processed concurrently. Such structured relaxation exposes substantial parallelism and maps naturally to throughput-oriented architectures, in contrast to designs that rely on strict global ordering. Recent GPU work [52] follows a similar philosophy: rather than implementing a fully general priority queue, algorithms are reformulated to use work-efficient frontier processing and lightweight ordering mechanisms that better match GPU execution models.

Nevertheless, recent work indicates that priority-queue-style abstractions can be realised on GPUs under carefully constrained conditions. The BGPQ design [53] proposes a heap-based priority queue for GPUs that achieves parallelism by allowing multiple thread blocks to operate concurrently on disjoint heapify paths. In this design, nodes are locked at the granularity of entire thread blocks rather than individual threads, reducing synchronisation overhead while still preserving correctness. Although BGPQ does not employ relaxed priority semantics, it demonstrates that limited forms of concurrency are possible for heap-based structures on GPUs and provides useful insights into synchronisation granularity and memory layout.

Overall, relaxed priority queues on GPUs appear to be a high-risk, exploratory research direction. While certain specialised workloads may benefit from relaxed ordering combined with GPU-friendly data layouts, adapting MultiQueue-style designs directly to GPUs is unlikely to yield competitive performance. This domain therefore serves primarily as a source of architectural insight and contrast, rather than as a primary target for the thesis.

3.6 Branch-and-Bound

BnB is a general algorithmic framework for solving combinatorial optimisation and decision problems by systematically exploring a search space while pruning subproblems that cannot lead to better solutions than those already found [54]. A central feature of some BnB implementations is the use of a priority queue to determine the order in which subproblems (nodes of the search tree) are explored, typically prioritising nodes with the most promising bounds. From an algorithmic perspective, BnB is particularly attractive for relaxed priority queues. This section provides the necessary background on BnB to motivate its suitability for relaxed concurrent priority queues and to support the framework design introduced later in the thesis.

3.6.1 Tolerance to Relaxed Priority Ordering

In a typical BnB algorithm, correctness is ensured by two global mechanisms: bounding and incumbents. Strict priority ordering is not required for correctness [55]; instead it primarily affects how quickly strong incumbents are discovered and how effectively pruning reduces the search space.

This property makes BnB a natural fit for relaxed concurrent priority queues. If subproblems are extracted out of order, the algorithm may explore less promising regions of the search space earlier than ideal, but it will still converge to the optimal solution. The primary cost of relaxation is therefore additional work, not incorrect results. This aligns closely with the design philosophy of the MultiQueue, which trades ordering precision for throughput while offering probabilistic bounds on priority deviation.

3.6.2 Representative Problem Families

BnB is not a single algorithm but a meta-algorithm that underlies a wide range of classical NP-hard problems. Prominent examples include knapsack variants, integer programming, satisfiability and pseudo-Boolean optimisation, constraint programming, scheduling, graph optimisation, and combinatorial design problems [38], [54]. These domains share a common structure: a large combinatorial search space, a bounding function that estimates solution quality, and a search strategy that benefits from prioritisation but does not require strict ordering.

The 0–1 knapsack problem is a canonical example [38]. Given a set of items with associated values and weights, the goal is to select a subset maximising total value subject to a capacity constraint. BnB algorithms solve this problem by recursively branching on item inclusion and using upper bounds on achievable value to prune subtrees. Many related problems, including change-making, bin packing, and multiple knapsack variants, can be formulated in a similar manner. Another closely related problem family is subset sum, where the objective is to determine whether a subset of integers sums to a target value [38]. Although NP-complete in the worst case, subset sum and its variants are often tackled using BnB-style search with bounding and heuristic ordering.

Because these problem families admit substantial parallelism and tolerate relaxed exploration order, they offer a rich and relatively unexplored design space for relaxed concurrent priority queues. A key difficulty is that the size and structure of the search tree are highly irregular and data-dependent, making load balancing and coordination among threads challenging in practice [56]. Unlike graph search or scheduling, where MultiQueue-based designs have already been studied extensively, BnB algorithms vary widely in structure and workload characteristics. This makes them particularly well suited for investigating how relaxation behaviour interacts with algorithm-level performance and pruning efficiency. For these reasons, BnB constitutes the most promising primary application domain for this thesis.

Because these problem families share a common structural pattern - combinatorial search with bounding and pruning - they provide a natural setting for BnB algorithms. At the same time, they exhibit significant variation in state representation, branching structure, and bound computation, which can lead to different performance characteristics in practice. This diversity makes them useful for evaluating how algorithmic structure interacts with implementation choices in BnB frameworks.

3.6.3 Parallel Branch-and-Bound

In parallel implementations of BnB, multiple workers explore the search tree concurrently. A shared work pool is often used to store active subproblems, and synchronisation is required to coordinate access to this pool and to maintain the global incumbent. A key property enabling parallelisation is that subproblems can be explored independently, as long as bounding information and the incumbent are shared correctly.

Parallelisations of BnB have long been studied as a means of accelerating combinatorial optimisation, with standard texts noting that the search tree can be explored concurrently by multiple workers, provided that bounding information is shared effectively [56]. A key difficulty is that the size and structure of the search tree are highly irregular and data-dependent, making load balancing and coordination among threads challenging in practice [56].

As a result, many existing approaches [36], [37] focus on algorithm-level parallelism, such as work stealing, dynamic load balancing, and decentralised task pools, in order to reduce contention on the global work pool. These techniques aim to distribute work efficiently while preserving enough structure to benefit from priority-based exploration.

In contrast, this thesis explores parallelism at the data-structure level. Rather than avoiding contention on a shared work pool, relaxed concurrent data structures are designed to handle concurrent access efficiently by relaxing strict priority ordering. This enables a different design point: a single logical work pool backed by a relaxed concurrent data structure in which threads can insert and extract subproblems with low synchronisation overhead.

3.6.4 Selected Benchmark Problems

While BnB provides a general strategy applicable to a wide range of combinatorial optimisation problems, only a small subset can be realistically explored within the scope of this thesis. The selection of benchmark problems is therefore guided by three main criteria: (i) implementation complexity, (ii) suitability for studying relaxed priority-queue behaviour, and (iii) representativeness of broader application domains.

A natural starting point is the knapsack problem family. Knapsack variants are among the simplest problems that admit efficient BnB formulations: the state representation is compact, branching decisions are straightforward (typically binary include–exclude choices), and strong bounds can be computed efficiently using well-known relaxations. In particular, fractional knapsack bounds provide tight upper bounds at low computational cost. This makes knapsack an ideal baseline for evaluating the interaction between search order and pruning effectiveness. Moreover, the simplicity of the implementation ensures that performance differences can be attributed primarily to scheduling and priority-queue behaviour rather than to complex domain-specific optimisations.

However, focusing exclusively on knapsack would limit the generality of the conclusions. To complement this domain, a graph-based combinatorial problem is also included, namely the maximum clique problem. Maximum clique, as described in Section 2.8.3, is a classical NP-hard problem with a rich literature on exact BnB algorithms, and it differs significantly from knapsack in several respects. In particular, the search space is defined over subsets of vertices rather than sequences of inclusion–exclusion decisions, and effective bounding relies on more involved techniques such as graph colouring heuristics. At the same time, it remains comparatively accessible from an implementation perspective, and well-performing BnB algorithms can be expressed with relatively concise code. Preliminary experiments further indicate that maximum clique exhibits non-trivial sensitivity to exploration order, making it a suitable candidate for studying relaxation effects.

Several additional problem classes were considered but ultimately excluded. Mixed Integer Linear Programming (MILP) is one of the most prominent application areas of BnB, with roots in the foundational work of Land and Doig [8]. In modern solvers, upper bounds are obtained by solving linear programming relaxations, while lower bounds are derived from feasible integer solutions. However, state-of-the-art MILP solvers incorporate a wide range of advanced techniques, including cutting planes, presolve reductions, sophisticated branching strategies, and heuristic solution methods. Understanding and reimplementing even a simplified version of such a solver requires substantial background in mathematical optimisation and is generally considered beyond the scope of a master's thesis. Furthermore, the complexity of these solvers would make it difficult to isolate the effects of relaxed priority scheduling.

Another class of candidates is scheduling problems, such as permutation flow-shop [57] or job-shop scheduling. These problems admit BnB formulations and are of practical relevance in operations research. However, efficient implementations typically rely on intricate bounding functions and problem-specific dominance rules. While promis-

ing, incorporating such problems would require additional background research and careful engineering to achieve competitive performance, and was therefore deferred in favour of more accessible domains.

Similarly, the travelling salesman problem (TSP) was considered due to its prominence and extensive study. However, high-performance exact TSP solvers rely on complex techniques such as branch-and-cut and advanced bounding procedures, making them less suitable for a clean and controlled evaluation of priority relaxation effects.

3.7 Summary

This section summarises the findings in Table 3.1, where each domain is listed, along with the reason for whether it was picked.

Table 3.1: Overview of candidate domains for relaxed concurrent priority queues.

Domain	**Assessment**	**Reason**
Graph traversal and search	Not selected	Highly relevant but already extensively studied.
Priority-based task scheduling	Not selected	Existing MultiQueue-based schedulers already cover much of this space.
Discrete event simulation	Not selected	Complex and risky.
Data compression	Not selected	Serves little purpose. The need for trading compression quality for faster compression would be difficult to establish.
Relaxed priority queues on GPUs	Not selected	Randomised queue selection and irregular access patterns involved in MultiQueue-style designs do not fit GPUs well.
Branch-and-bound	Selected	Strong fit. Tolerates relaxed exploration order and is likely to benefit from it in at least one problem.

4

Related Work

This chapter focuses on related work regarding relaxed data structures. Then, the chapter also highlights the history of exact maximum clique solvers.

4.1 Research on the MultiQueue

Research on concurrent priority queues has shown that strict semantics fundamentally limit scalability on modern multicore systems. Early work by Hagit Attiya et al. [14] and Faith Ellen et al. [15] established that enforcing a globally consistent `deleteMin` operation induces unavoidable synchronisation overhead, motivating the development of relaxed alternatives. In response, a large body of work has explored data structures that weaken ordering guarantees in exchange for improved throughput, including skip-list-based designs such as the SprayList [16] and k-LSM [18].

The MultiQueue [5], [7], [23], first introduced by Rihani, Sanders, and Dementiev, represents a particularly simple and effective design point. By distributing elements across multiple internal queues and using randomised selection for deletions, it achieves high scalability while providing probabilistic guarantees on rank error. Subsequent work by Williams et al. [7], [23] refines the design with engineering optimisations such as buffering and stickiness, and provides a more complete analysis of the throughput–quality trade-off. More recent studies [19], [20] further demonstrate that MultiQueue-based designs can act as competitive priority schedulers in practice.

The idea of distributing priority queues across processors predates the MultiQueue. An early example is the BnB algorithm by Karp and Zhang [58], where each processor maintains a local priority queue and newly generated subproblems are distributed randomly across processors. Their work demonstrates that decentralised randomised work distribution can achieve near-optimal parallel execution times with high probability, while avoiding global shared data structures and complex synchronisation. However, the algorithm assumes a synchronous message-passing model and does not provide the relaxed `deleteMin` semantics studied in later concurrent priority-queue research. Similar distributed priority-queue designs have also been studied by Sanders [59], [60].

Beyond data-structure design, relaxed priority queues have been studied in some algorithmic contexts where strict ordering is not required for correctness. In graph

processing, several asynchronous shortest-path algorithms tolerate relaxed priority semantics by allowing repeated relaxations or modifying termination conditions. Frameworks such as Wasp [30] demonstrate, with their single-source shortest path solver, that controlled relaxation can improve scalability while maintaining correctness through convergence guarantees.

4.1.1 Theory on Relaxed Concurrent Priority Queues in Branch-and-Bound

The interaction between relaxed priority queues and BnB has been analysed conceptually in recent work on the MultiQueue by Williams and Sanders [23]. In best-first BnB, nodes are explored in order of their most promising bound using a priority queue. Assuming, without loss of generality, that the problem is a maximisation problem, nodes are prioritised according to decreasing upper bounds.

Let H denote the search tree, and let $\mathrm{UB}(v)$ denote the upper bound associated with a node $v \in H$. Further, let OPT denote the value of the optimal solution. Define

$$H_{\geq} = \{v \in H \mid \mathrm{UB}(v) \geq \mathrm{OPT}\},$$

that is, the set of nodes whose upper bounds are sufficiently large that they may still contain the optimal solution. The remaining nodes are defined as

$$H_{<} = H \setminus H_{\geq},$$

corresponding to nodes whose upper bounds are strictly smaller than the optimal solution value. Such nodes cannot contain an optimal solution and will eventually be pruned by the sequential algorithm once the optimum has been discovered.

With relaxed priority queues, nodes from $H_{<}$ may be explored prematurely due to delays in processing higher-priority nodes [23].

Using delay as a quality measure, Williams and Sanders [23] show that if h is the depth of the optimal solution and D is the expected delay of the priority queue, then at most Dh additional nodes from $H_{<}$ are explored in expectation. This yields a total of $|H_{\geq}| + Dh$ explored nodes. Assuming each node operation has cost $O(\log n)$, where n denotes the current number of elements in the priority queue, the expected parallel runtime with p worker threads becomes

$$T_{\mathrm{par}} = \frac{T_{\mathrm{seq}}}{p} + O\left(\frac{|H_{\geq}| + Dh}{p} \log n\right).$$

For the MultiQueue, where $D \in O(p)$, this simplifies to

$$T_{\mathrm{par}} = \frac{T_{\mathrm{seq}}}{p} + O\left(\left(\frac{|H_{\geq}|}{p} + h\right) \log n\right).$$

This bound is comparable to earlier analyses of bulk-parallel priority queues [59], but applies without requiring uniform node processing times.

In contrast to this theoretical analysis, many existing parallel BnB implementations focus on work distribution strategies such as work stealing and decentralised task pools [36], [37]. These approaches aim to reduce contention rather than relax priority semantics within a shared queue.

A complementary theoretical perspective on relaxed scheduling is provided by Alistarh, Nadiradze, and Koval [61]. They analyse how bounded relaxation affects the total work of incremental algorithms when tasks are executed out of order. Using a model of a *k-relaxed scheduler*, which bounds the maximum priority inversion allowed during task execution, they show that for several algorithms, including Delaunay triangulation and insertion-based sorting, the additional wasted work introduced by relaxation is only $O(\log n \cdot \text{poly}(k))$. For single-source shortest paths, the additional work is bounded by $O(\text{poly}(k)\, d_{\max}/w_{\min})$, where $d_{\max}$ is the maximum shortest-path distance and $w_{\min}$ is the minimum edge weight. These results provide theoretical justification for using relaxed priority queues: when the number of tasks is large relative to the relaxation factor, the overhead introduced by relaxation can remain small compared to the gains in parallel scalability. The authors also establish lower bounds, showing that some overhead from relaxation is unavoidable.

This thesis builds on these lines of work by evaluating relaxed concurrent queues as a general-purpose scheduling mechanism in BnB, focusing on how relaxation affects search efficiency, pruning behaviour, and scalability.

4.2 Generic Branch-and-Bound Frameworks

To the best of our knowledge, there is also little prior work on generic reusable frameworks for exact BnB in the sense considered in this thesis: a framework that separates problem-specific components such as node representation, branching, and bounding from shared execution logic, while allowing the same solver infrastructure to be reused across multiple problem domains and queue implementations. Existing implementations are typically developed as problem-specific solvers or as experimental codes tied to a single application, making systematic comparisons across problems and scheduling strategies more difficult.

4.3 Branch-and-Bound Search Strategy

Another relevant observation is that many exact BnB solvers rely on depth-first search rather than best-first exploration. This is particularly visible in the maximum clique literature, where highly optimised exact solvers are usually built around depth-first BnB schemes [34], [35], [36], [37], [39]. A major reason is memory efficiency: depth-first search only needs to store a path in the search tree together with limited auxiliary state, whereas best-first search requires maintaining a potentially much larger pool of open nodes. As a result, best-first BnB is comparatively less common in high-performance exact solvers, despite its potential to discover strong incumbents earlier and thereby improve pruning. This suggests a possible gap in the literature: if relaxed concurrent priority queues can make best-first exploration practical at scale,

they may offer a competitive alternative to more traditional depth-first parallel BnB designs.

4.4 Historical Development of Exact Maximum Clique Algorithms

Early exact algorithms for the maximum clique problem follow a simple recursive scheme. Starting from a candidate set of vertices, a vertex is selected and added to the current clique, and the search continues on the subgraph induced by its neighbours. By exploring all possible choices in this manner, all maximal cliques can be enumerated, and the largest one can be identified. This basic approach, often described as a depth-first search over the space of vertex subsets, forms the foundation of many later algorithms [39].

Subsequent work focused on improving pruning efficiency within this search framework. Östergård [34] introduced a refinement that maintains auxiliary information about the maximum clique size achievable from subsets of vertices, enabling stronger pruning decisions during the search. Rather than fundamentally changing the search strategy, this work demonstrates how tighter bounds can significantly reduce the explored search space.

Further advances were made by Tomita and Kameda [35], who introduced techniques based on approximate vertex colouring to compute upper bounds on clique size. The idea is that if a graph can be coloured with k distinct colours, it cannot contain a clique of size larger than k. This is because a clique of size r requires r distinct colours, since every pair of vertices in the clique is adjacent. An upper bound k can be found by greedily colouring a graph, resulting in k colours. Combined with improved vertex ordering heuristics, these methods lead to substantial performance improvements while still adhering to the same underlying depth-first BnB structure. This line of work forms the basis of many modern exact maximum clique solvers.

San Segundo proposed the solvers BBMCI and BBMCR [62], [63] in his two papers where he introduces bit-level optimisations to compute graph operations faster. This research contributed to a constant factor speedup that most maximum clique solvers can easily enjoy by handling the graph operations with bitsets.

A particularly strong sequential exact solver is MoMC, proposed by Jiang et al. [64]. MoMC is a BnB algorithm that combines approximate graph colouring with incremental MaxSAT reasoning to compute tighter upper bounds for pruning. In addition, the solver dynamically selects between different vertex orderings depending on graph structure, using either degeneracy ordering or a maximum-independent-set-based ordering. These techniques substantially reduce the explored search space and make MoMC highly effective on difficult benchmark instances.

Another recent sequential exact maximum clique solver is CliSAT, proposed by San Segundo et al. [65]. CliSAT is a sequential BnB solver that strengthens pruning through graph-colouring bounds, partial MaxSAT reasoning, and constraint-propagation-based filtering. Its accompanying repository provides benchmark in-

stances, binaries, and supplementary results [66]. The paper evaluates both CliSAT and MoMC on the same machine and presents the results. Their results show that CliSAT is slightly faster than MoMC on most DIMACS inputs, but performs roughly on par with it.

Parallel exact algorithms have also been developed, typically by building on these sequential methods. McCreesh and Prosser [36] present a multi-threaded variant that distributes subproblems across workers using a global work pool, while preserving the sequential search logic within each subproblem. Work distribution is achieved through work donation, where an idle worker sets a shared atomic flag to advertise when the global work pool is empty. When the flag is set, non-idle workers can place parts of their search tree onto the global work pool and unset the flag. An incumbent is shared between threads via an atomic to prune the thread-local searches. The main idea behind this approach is to avoid communication overhead as much as possible, by letting the threads work disjointly and only accessing the global work pool to avoid idleness. The paper is not specific in its implementation details. For instance, it does not disclose the concrete data structure for the global queue, nor the synchronisation mechanism used to protect it. It also does not explain how its idle-worker detection works.

What is particularly interesting is comparing the results in the paper on MoMC with the paper by McCreesh and Prosser. On a single thread, MoMC seems to strongly outperform the solver by McCreesh and Prosser. They both present execution times on the DIMACS [40] dataset. Although their results are not based on the same machine, MoMC seems to perform around twice as fast on most DIMACS inputs. On some inputs, MoMC is unexpectedly fast. For instance on the input named gen400_p0.9_65, where MoMC solves it in 0.32s, the solver by McCreesh and Prosser reports 431,310.7s on a single thread.

More recent work by Jiang et al. [37] proposes a parallel BnB algorithm for the maximum clique problem called *Parallel Bounded Search* (PBS). The algorithm executes multiple BnB searches in parallel, where each worker maintains its own search subproblem while sharing a global incumbent corresponding to the best clique found so far. Each active search also maintains an upper bound on the largest clique it can still discover. Whenever the shared incumbent reaches or exceeds this upper bound, the corresponding search can terminate early because it can no longer improve the global solution. The main idea of PBS is to organise parallel searches so that incumbent updates can invalidate many active searches simultaneously, thereby reducing the total explored search space. The implementation is based on the previously described state-of-the-art sequential maximum clique solver MoMC, and uses message passing to propagate incumbent updates between workers. Experimental results on DIMACS [40] benchmark instances show near-linear speedups on many instances and super-linear speedups on several particularly hard instances. The authors attribute these super-linear speedups to earlier discovery of strong incumbents during parallel execution, which increases pruning efficiency and reduces the amount of search required overall. Unlike approaches based on shared concurrent priority queues, however, PBS still relies primarily on decentralised depth-first BnB searches rather than globally coordinated best-first exploration.

PBS doesn't report absolute execution time for their parallel solver. They only report execution times for MoMC, and then show plots showing speedups when moving from MoMC to PBS. Their table of execution times for MoMC closely resemble the one in the paper on MoMC, which is expected. The speedups reported when switching to PBS are sublinear but close to linear on the majority of DIMACS input graphs. They observed superlinear speedup on only one of the inputs. Although surprising, these results lead us to expect PBS to be the best performing solver, with MoMC coming second out of the two, and McCreesh and Prosser behind MoMC.

A notable observation across this line of research is that the core search procedure remains largely unchanged: a depth-first BnB exploration of the search tree. Parallelisation efforts have primarily focused on distributing work efficiently and minimising synchronisation overhead, often resulting in designs where access to shared data structures is relatively infrequent. This raises the question of whether alternative approaches, such as relaxed concurrent queues, can provide additional benefits by enabling different exploration orders and reducing coordination costs.

5

Branch-and-Bound Framework

The purpose of our generic BnB framework is to easily create and benchmark solvers for different BnB problems and experiment with different types of queues used to facilitate the exploration through the search tree. To begin the implementation of such a framework, inspiration was taken from the existing source code [67] by Marvin Williams, written in C++. The codebase contains an implementation of a configurable MultiQueue and a small set of benchmarks presented in their paper [23]. It is extended into a generic and reusable framework by separating problem-specific logic from the execution and measurement infrastructure, and generalising it into a modular design.

The key idea is to encapsulate all problem-dependent aspects of a BnB algorithm – such as the representation of partial solutions, branching rules, and bound computations – into a single *problem definition*, while keeping the search strategy, scheduling, and instrumentation (the *driver code*) fixed. This allows new benchmarks to be added with minimal effort. Implementing a new problem requires only defining how nodes are constructed, expanded, and evaluated, without modifying the surrounding infrastructure.

This design also makes it straightforward to benchmark different queue implementations under identical workloads. Since the framework exposes a uniform interface, the same driver code can be reused across multiple problems and queue configurations, enabling consistent and reproducible comparisons. Essentially, the abstraction enables building solvers and benchmarking them by selecting and piecing together a problem definition, driver code, and a data structure. In short, the driver code accesses the data structure to fetch work, and consults the problem definition to complete the work, making it a modular design. For example, to experiment with the MultiQueue on the Travelling Salesman Problem (TSP), one could simply implement the problem-specific parts for TSP, such as how to compute bounds, and the generic framework allows connecting it together with driver code and selecting the MultiQueue as the queue type. The rest of the report will refer to the queue type as the *underlying queue*.

This chapter is devoted to explaining the framework itself and the driver code, while Chapter 6 describes the implementation of problem definitions and underlying queues.

5.1 Framework Abstraction

The abstraction is centred around the `BnBProblem` interface, which is implemented using what is commonly known among C++ developers as the Curiously Recurring Template Pattern (CRTP) [68]. Each concrete problem instance needs to provide a derived class from `BnBProblem` that defines the necessary operations for BnB search.

A problem definition implementation must specify:

- A `data_type` representing the objective value.
- A `node_type` representing a partial solution, also storing a lower and upper bound.
- A method `root_impl()` that constructs and returns the root node along with its bounds stored in it.
- A method `branch_impl(node, incumbent, out)` that generates the children of a node and adds them to the `out` container passed as an argument. The children's bounds are also computed and stored during this call.

The framework exposes these operations through a thin wrapper, ensuring a consistent interface while allowing the compiler to inline all calls. This avoids the overhead of virtual dispatch and is particularly important for high-performance benchmarking.

Each node stores both a lower bound and an upper bound, representing bounds on the best solution that can be found in the subtree rooted at that node. The upper bound is used as the priority key in best-first search, while the lower bound is used to update the incumbent solution. The lower bound represents a feasible solution value and is therefore used to update the incumbent. The correctness of pruning relies on the bounds being valid, but the framework does not impose any restrictions on how they are computed.

It is assumed that the problem to be solved is a maximisation problem, thus a larger upper bound is prioritised over a smaller upper bound. This is currently a limitation with the framework but can be easily fixed by extending the problem definition to also specify if it is a maximisation or minimisation problem. Just like a max-heap can be used as a min-heap by negating all insertions and deletions, this framework limitation can be worked around in the same way to use it for minimisation problems too.

Branching is performed by generating all feasible child nodes from a given node. The current incumbent value is passed to the branching function, allowing problem-specific implementations to perform early pruning or avoid generating dominated children.

The use of CRTP enables static polymorphism: all problem-specific logic is resolved at compile time, allowing the compiler to inline calls and optimise across abstraction boundaries without introducing indirection overhead. Since node expansion and bound computation lie on the critical path of the search and are executed extremely

frequently, avoiding runtime polymorphism is important for maintaining high performance. In contrast, an implementation based on virtual functions would require dynamic dispatch through virtual table lookups, limiting optimisation opportunities and introducing additional overhead on every call. The result is modularity and abstraction without runtime overhead or sacrificed performance, one of the great strengths of C++.

5.2 Driver Code

The framework provides two driver implementations: a sequential version and a parallel version. Both share the same high-level structure and rely exclusively on the problem interface described above.

5.2.1 Sequential Driver

The sequential driver has access to a selected underlying queue and executes the core BnB search loop shown in Algorithm 1. In addition to the search itself, the full driver implementation also contains surrounding infrastructure such as queue initialisation, timing and metric collection, and result aggregation.

Algorithm 1 Main Branch-and-Bound Search Loop

```
1: r ← ROOT
2: incumbent ← r.lower_bound
3: if r.upper_bound > incumbent then
4:     PUSH(Q, r)
5: end if
6: while Q is not empty do
7:     node ← POP(Q)
8:     if node.upper_bound ≤ incumbent then
9:         continue
10:    end if
11:    children ← BRANCH(node, incumbent)
12:    for all child ∈ children do
13:        incumbent ← max(incumbent, child.lower_bound)
14:        if child.upper_bound > incumbent then
15:            PUSH(Q, child)
16:        end if
17:    end for
18: end while
19: return incumbent
```

The execution continues until the underlying queue is empty. This implementation serves both as a baseline and as a correctness reference for the parallel version. The upper and lower bounds are stored directly in the node so that they can be accessed by the driver code without recomputation when nodes are inserted into or removed

from the underlying queue. Note that during line 7, the `pop` behaviour belongs to the underlying queue. Thus, the search order (DFS or best-first) is determined by the underlying queue, not the driver code.

5.2.2 Parallel Driver

The parallel driver generalises the search loop from Algorithm 1 to multiple worker threads executing concurrently over a shared thread-safe underlying queue. Each thread repeatedly extracts nodes, expands them, updates the incumbent, and inserts newly generated nodes until global termination is detected.

The shared state between threads includes the global incumbent value and termination detection mechanisms. The incumbent is maintained using atomic operations (an `std::atomic` type) to ensure correctness under concurrent updates. When a thread discovers a better lower bound, it attempts to update the incumbent using a compare-and-swap operation.

Termination detection is handled explicitly, as the relaxed nature of the concurrent queues means that threads may temporarily observe an empty queue even when work remains. The termination detection mechanism uses atomic counters to track the number of idle threads. The framework uses this as the primary detection, and uses an extra safeguard for when the underlying data structure doesn't provide the same guarantees as for example the MultiQueue. This safeguard involves tracking the number of outstanding nodes and if all threads are idle but some nodes are still outstanding, the termination criteria is not met. The algorithm continues until all threads are idle and no more work is waiting.

The driver code is fully generic with respect to the problem definition. To run a benchmark on a new problem, it suffices to implement a class conforming to the `BnBProblem` interface and pass it as a template parameter to the driver. No changes to the driver logic are required.

This separation of concerns ensures that all benchmarks share identical setup and measurement logic, while allowing the problem-specific aspects of BnB to be defined independently, steering the actual search.

5.2.3 Recorded Metrics

Regardless of the selected underlying queue and the problem instance, the driver code records a few metrics useful for benchmarking. The first and perhaps most interesting metric is the execution time of the solver itself. Initial setup like constructing the underlying queue and possibly heap-allocating space for its items happens before the timer is started. Only the solver time is recorded as part of the execution time. As soon as the answer is returned in the driver code, the timer is stopped. The timer used is a `std::chrono::steady_clock` from the C++ standard library.

Along with the execution time, the number of processed nodes in the BnB search tree is also recorded. Each thread has its own thread-local counter `processed_nodes` that it increments every time it processes a node. Whenever a node is about to

be processed, but the comparison with the incumbent shows it should be pruned instead, another thread-local counter `ignored_nodes` is incremented instead. After the answer has been found, the threads coordinate to accumulate their counters and the final sums are presented in the program's standard output, along with the execution time.

These metrics are useful for understanding the effects of relaxation and how much extra work is done by different solver configurations.

5.2.4 Batching

A simple optimisation that in theory can benefit all solvers regardless of the selected underlying queue is something we call batching. Batching is part of the generic framework, implemented in the driver code, and controls how frequently the threads access the underlying queue. Whenever a thread fetches work from the underlying queue, shared by all threads, the batching value determines how long the thread works locally before pushing new nodes into the underlying queue. The batching value is a positive integer and directly tells each thread how many nodes to process locally. A batching value of one correlates to no batching at all, which means each thread pops a node from the underlying queue, processes it to generate its children, and inserts the children back into the underlying queue. For instance, a batching value of 100 means that a thread pops a node from the underlying queue and then continues processing locally until 100 nodes have been processed. To achieve this, each thread maintains a local work queue for storing and retrieving work. When the batch has been processed, the remaining nodes in the local work queue are inserted into the global work queue (the underlying queue). Algorithm 2 illustrates the high-level execution flow of a worker thread using batching.

In our implementation, the framework uses a FIFO queue as the local work queue, meaning the local search behaviour follows a DFS order. For best-first searches, batching also acts as another layer of relaxation, since threads work with batches of nodes instead. The batches follow priority order, since the priority of a batch is determined by its root. Within the batches, priority is not followed due to the DFS order enabled by the local work FIFO queue. Future work could adapt the implementation so that the local queue reflects the search order (FIFO queue for DFS, and priority queue for best-first) of the underlying queue. This would maintain the same search order strategy for each solver. Further research could also investigate whether this is beneficial.

Algorithm 2 Batched Worker Execution

```
1: while termination not detected do
2:     root ← POP(Q_global)
3:     if root = null then
4:         continue
5:     end if
6:     PUSH(Q_local, root)
7:     processed ← 0
8:     while processed < batch_size ∧ Q_local ≠ ∅ do
9:         node ← POP(Q_local)
10:        if node.upper_bound ≤ incumbent then
11:            ignored ← ignored + 1
12:            continue
13:        end if
14:        children ← BRANCH(node, incumbent)
15:        for all child ∈ children do
16:            if child.lower_bound > incumbent then
17:                incumbent ← child.lower_bound
18:            end if
19:            PUSH(Q_local, child)
20:        end for
21:        processed ← processed + 1
22:    end while
23:    while Q_local ≠ ∅ do
24:        node ← POP(Q_local)
25:        if node.upper_bound > incumbent then
26:            PUSH(Q_global, node)
27:        end if
28:    end while
29: end while
```

6

Implementation

This chapter describes the concrete problem implementations built on top of the general BnB framework introduced in Chapter 5. For each selected benchmark problem, the implementation must define how a search node is represented, how child nodes are generated during branching, and how valid lower and upper bounds are computed. These components are the problem-specific parts of the solver, while the surrounding driver code, underlying queue integration, and benchmarking infrastructure are shared across all benchmarks. The selected BnB problems to implement, as mentioned in Chapter 3, are a couple of knapsack variants and the maximum clique problem. This chapter focuses on the implementation details that determine the search-space structure and pruning effectiveness for these problems, with particular emphasis on the maximum clique problem.

6.1 0–1 Knapsack Problem

The 0–1 knapsack solver is already present in the original benchmark suite, but as a standalone benchmark rather than as an instantiation of the generic BnB framework. In this work, that implementation is refactored to conform to the common `BnBProblem` interface, so that it can be executed by the same generic driver code as the other benchmark problems. This makes it possible to evaluate the 0–1 knapsack problem under the same scheduling, instrumentation, and underlying queue abstractions as any other solver using the framework. The following sections explain the implementation of the 0–1 knapsack problem definition.

6.1.1 Instance Representation

The implementation reads standard 0–1 knapsack benchmark instances from KPLIB [69], a widely used library of knapsack test problems [70]. Each instance consists of a maximum capacity and a set of items, where each item has a value and a weight. Recall that the goal of 0–1 knapsack is to pick a subset of items to maximise the sum of their values, while keeping the sum of their weights under the maximum capacity.

To enable efficient bounding, the items are sorted in non-increasing order of value-to-weight ratio. This ordering is used consistently throughout the search when computing bounds and generating branches.

6.1.2 Node Representation

Each search node stores five pieces of information:

- an upper bound,
- a lower bound,
- the current item index,
- the remaining free capacity, and
- the total value accumulated so far.

The index indicates the current position in the sorted item sequence. A node therefore represents a partial assignment in which all items before the current index have already been decided, while the remaining items are still available for future branching. The accumulated value corresponds to the profit of the current partial solution, and the free capacity determines how much additional weight may still be packed. A custom comparator associating larger upper bounds with higher priority is implemented, to support the case that the underlying queue is a priority queue.

6.1.3 Root Node

The root node corresponds to the empty knapsack. No items have yet been fixed, the accumulated value is zero, and the remaining capacity equals the full knapsack capacity. If the capacity of the instance is denoted by C, the root node is initialised as

$$i = 0, \qquad \mathit{value} = 0, \qquad \mathit{free\_capacity} = C.$$

The bounds of the root node are computed from all items using the same bounding routine as for internal nodes. These bounds define both the initial incumbent and the potential priority of the root node, in case the search is best-first.

6.1.4 Bounds from the Fractional Relaxation

The implementation computes bounds using a fractional-knapsack relaxation. Starting from a node with remaining capacity c and next undecided item index i, items are considered in the sorted value-density order.

The lower bound is obtained by greedily inserting full items as long as they fit within the remaining capacity. This produces a feasible 0–1 solution, and therefore a valid lower bound.

The upper bound is derived from the same greedy process, but allows the next item to be taken fractionally if it does not fully fit. If the remaining capacity after packing full items is r, and the next item has value v and weight w, then the fractional contribution is

$$\frac{v}{w}\, r.$$

The upper bound is thus the value of the current partial solution plus the value of all fully packed items and the fractional contribution of the next item, which is

guaranteed to be at least as good as any feasible 0–1 solution obtainable from that node.

This relaxation corresponds to solving the fractional knapsack problem, which provides a tight and computationally inexpensive upper bound. The effectiveness of this bound is largely due to the ordering of items by value-to-weight ratio.

In the implementation, these bounds are computed efficiently using prefix-sum arrays over the sorted items, allowing constant-time access to cumulative weights and values when evaluating how many items can be packed.

6.1.5 Branching

Branching is performed by considering the next undecided item and creating up to two children corresponding to the two possible decisions: exclude the item or include it.

For a node at item index i, the first child represents the branch where item i is excluded. In this case, the accumulated value and remaining capacity remain unchanged, while the index advances to $i + 1$. The bounds are then recomputed for the subproblem consisting of the remaining items.

The second child represents the branch where item i is included. This child is generated only if the item fits within the remaining capacity. Its value is increased by the value of item i, its free capacity is reduced by the weight of item i, and its index is advanced to $i + 1$.

6.2 Multidimensional Knapsack Problem

The multidimensional knapsack problem (MDKP) extends the classical 0–1 knapsack problem by replacing the single capacity constraint with multiple resource constraints. Each item therefore has one profit value but a vector of weights, and a feasible solution must satisfy all capacity limits simultaneously. The MDKP problem definition is implemented as a second knapsack-based instantiation of the generic `BnBProblem` interface. The implementation supports standard benchmark input instances from the OR-Library [71], whose multidimensional knapsack data files follow the same format assumed by the parser used here.

6.2.1 Instance and Node Representation

As in the 0–1 knapsack implementation, a node represents a partial assignment over a fixed item order. Each node stores an upper bound, a lower bound, the current item index, the accumulated value, and a vector of remaining capacities—one for each constraint. The root node corresponds to the empty solution, with value zero, index zero, and remaining capacities equal to the full instance capacities. Its bounds are computed from the complete set of items.

Unlike the one-dimensional case, there is no single canonical value-to-weight ratio. The implementation therefore precomputes two kinds of orderings. First, it builds a global heuristic order based on an aggregated efficiency measure, defined as the item value divided by the sum of normalised weights across all constraints. Second, for each individual constraint, it builds a separate order sorted by one-dimensional value-to-weight ratio for that constraint.

6.2.2 Bounds

The lower bound is obtained greedily using the global item order. Starting from the remaining capacity vector of the current node, the algorithm scans the globally sorted items that have not yet been decided and inserts every item that fits in all constraints. This produces a feasible solution and therefore a valid lower bound.

The upper bound is based on one-dimensional fractional relaxations. For each constraint separately, the remaining subproblem is viewed as an ordinary fractional knapsack problem using only that constraint. Items are then packed in the precomputed order for that constraint, allowing the final item to be taken fractionally if necessary. This yields one upper bound per constraint, and the final node upper bound is taken as the minimum of these values. Since every feasible MDKP solution must satisfy every single constraint, the minimum over these one-dimensional relaxations remains a valid upper bound.

6.2.3 Branching

Branching follows the same binary structure as in the 0–1 knapsack benchmark. At a node with current index i, one child excludes item i, leaving the current value and remaining capacities unchanged while advancing to the next index. The other child includes item i, but only if the item fits in every remaining capacity. In that case, the child value is increased by the item's profit and each remaining capacity is reduced by the corresponding weight component.

Overall, the MDKP benchmark can be viewed as a direct generalisation of the 0–1 knapsack implementation. The branching structure remains the same, but the state representation and bounding procedures must account for multiple simultaneous resource constraints.

6.3 Maximum Clique Problem

The maximum clique benchmark is once again implemented as a concrete instantiation of the generic `BnBProblem` interface. In the implementation, each node represents a partial clique together with a set of candidate vertices that may still extend it.

6.3.1 Graph Representation

The graph instance is stored as an adjacency structure based on bitsets. For a graph with n vertices, each vertex v is associated with a bitset representing its set of neighbours. This representation makes set intersection operations efficient, which is important because BnB for maximum clique repeatedly computes the intersection between the current candidate set and the neighbourhood of a selected vertex. These bit-level optimisations can be attributed to the work by San Segundo [62], [63].

The implementation accepts DIMACS-style graph instances as input, following the benchmark set [40] from the Second DIMACS Implementation Challenge [72]. In this format, a problem line specifies the number of vertices, and each edge line gives an undirected edge between two vertices. A simpler fallback edge-list format is also supported. Internally, both formats are converted into the same adjacency-bitset representation. Using bitsets makes it possible to implement operations such as intersection, membership tests, and iteration over candidate vertices with low overhead, which is particularly beneficial since these operations are on the critical path of the search.

6.3.2 Node Representation

Each search node stores four pieces of information:

- an upper bound,
- a lower bound,
- the size of the current partial clique, and
- a bitset of candidate vertices.

The current clique itself is not stored explicitly. Instead, the implementation keeps only its size, denoted here by $|C|$, together with the candidate set P. The set P contains exactly those vertices that are adjacent to every vertex already chosen for the clique, and therefore each vertex in P can be added to the current partial solution without violating the clique property.

This compact representation is sufficient for optimisation, since the benchmark is only concerned with the clique size and not with reconstructing the exact vertex set. Storing only $|C|$ and P reduces per-node memory usage and keeps the cost of branching low.

6.3.3 Root Node

The root node corresponds to the empty clique. Its candidate set initially contains all vertices of the graph, since any vertex may be chosen as the first element of a clique. The clique size of the root is therefore zero, while the lower and upper bounds are computed from the full vertex set.

Formally, the root node is initialised as

$$C = \emptyset, \qquad P = V, \qquad |C| = 0.$$

where V is the full vertex set of the input graph.

The lower bound is obtained by greedily completing a clique from P, and the upper bound is obtained by greedily colouring the subgraph induced by P. These two bounds determine both the initial incumbent value and the priority of the root node.

6.3.4 Upper Bound by Greedy Colouring

The upper bound used in the implementation is based on a greedy sequential colouring of the subgraph induced by the candidate set P. The number of colours needed by such a colouring is an upper bound on the size of any clique contained entirely in P, because vertices in a clique must all receive different colours. Consequently,

$$\omega(P) \leq \chi(P),$$

where $\omega(P)$ denotes the maximum clique size of the induced subgraph on P and $\chi(P)$ denotes its *chromatic number* (minimum number of colours for a valid colouring). Since the implementation uses a greedy colouring heuristic rather than an exact colouring algorithm, the computed number of colours is not necessarily minimal, but it is still a valid upper bound.

The colouring routine repeatedly constructs one colour class at a time. Starting from the set of remaining candidate vertices, it greedily builds a maximal independent set: a vertex is selected, assigned the current colour, and all of its neighbours are removed from the set of vertices that can still receive that same colour. Once no further vertices can be added to the current colour class, a new colour is started and the process repeats until all vertices in P have been coloured.

If the greedy colouring of P uses k colours, then the node upper bound becomes

$$\mathrm{UB}(C, P) = |C| + k.$$

Intuitively, it represents the largest clique size that could still be obtained by extending the current partial clique under an optimistic view of the remaining candidates.

6.3.5 Lower Bound by Greedy Clique Completion

To obtain a feasible lower bound, the implementation greedily completes a clique starting from the candidate set P. The method repeatedly selects an arbitrary vertex from P, adds it to the tentative completion, and then replaces P by its intersection with the neighbourhood of the selected vertex. This ensures that all remaining vertices are adjacent to every vertex chosen so far. The procedure stops when no candidates remain.

If the greedy completion adds g vertices to the current partial clique, the resulting lower bound is

$$\mathrm{LB}(C, P) = |C| + g.$$

Because this construction always produces a valid clique, it yields a feasible solution and is therefore a valid lower bound. Although this bound is generally weaker than

what might be obtained by more sophisticated clique heuristics, it is inexpensive to compute and suitable for frequent use during the search.

6.3.6 Colour-Based Branching Order

The branching step follows a Tomita-style [35] expansion strategy. For a node (C, P), the candidate set P is first greedily coloured again, but this time the procedure additionally records two arrays:

- an ordered list of candidate vertices, and
- for each position in that list, the colour bound associated with the vertex.

The colour bound at a given position indicates an upper bound on how many vertices can still be added from that suffix of the candidate order. This information is used to prune branching decisions early. If

$$|C| + b_i \leq \mathit{incumbent},$$

where b_i is the stored colour bound at position i, then no branch from that point can improve the incumbent and the remaining vertices can be skipped.

The implementation expands vertices in reverse colouring order. For each selected vertex v, a child node is generated by adding v to the current clique and restricting the candidate set to those vertices that are both still available and adjacent to v:

$$C' = C \cup \{v\}, \qquad P' = P_{\text{rem}} \cap N(v),$$

where P_{rem} denotes the set of candidates not yet excluded by earlier branches in the ordered expansion.

The child node then has clique size

$$|C'| = |C| + 1,$$

and its lower and upper bounds are recomputed as

$$\text{LB}(C', P') = |C'| + \text{greedy-completion}(P'),$$

$$\text{UB}(C', P') = |C'| + \text{greedy-colouring}(P').$$

After a branch on vertex v has been created, v is removed from the remaining candidate set before the next branch is considered. This corresponds to the usual BnB decomposition in which each branch fixes one choice and the subsequent branches exclude previously considered vertices.

6.4 Data Structures

As explained in Chapter 5, the generic BnB framework supports connecting different underlying queues with the driver code. The underlying queue is selected during compilation via a `--target` flag. The meta-build system **CMake** is used for

compilation, since it provides flexibility and supports several build systems. New underlying queues can be added by simply providing a type having `push()` and `try_pop()` functions, and making the type selectable via `--target` during compilation in the CMake configuration files, called `CMakeLists.txt` files or CML files. These functions need to conform to specific signatures so that the driver code can use them. This means that integrating a new underlying queue that provides `push()` and `try_pop()` methods, but with slightly different function signatures, will lead to compilation errors. The easiest way to fix this mismatch is to wrap the underlying queue into a new type that conforms to the function signatures the driver code expects. These functions can forward the operations to the internal underlying queue in a way that masks the differences in function signatures, acting as an adapter. Integrating new underlying queues usually means adding their source code to the project, writing wrapper types for them, and making them visible in the build system via the CML files.

Since this project aims to compare a variety of underlying queues, they first have to be integrated with the project. The following sections outline which underlying queues were selected to be integrated and why.

6.4.1 Sequential Stack & Priority Queue

The most simple baseline for DFS-behaviour possible would be a completely regular stack in a single-threaded environment. Running benchmarks with the sequential stack serves as a baseline to compare other stack-based data structures against. Such a stack contains no synchronisation mechanisms and hence no synchronisation overhead. In our implementation, an `std::stack` from the C++ standard library is used.

The direct equivalent to the sequential stack but supporting best-first search behaviour would be the sequential priority queue. Similarly, it supports no synchronisation and will only be run sequentially with a single thread. The `std::priority_queue` from the STL is used.

6.4.2 Globally Locked Stack & Priority Queue

The sequential stack and priority queue serve as baselines for single-threaded contexts. For baselines in multi-threaded contexts we can turn to globally locked data structures. Such data structures ensure thread-safety by wrapping an underlying data structure with a mutex. This way, threads accessing the data structure must first wait to acquire the lock, and upon completing the access, they must relinquish it. This pattern is applied to both an `std::stack` and a `std::priority_queue` as wrapper types to create a globally locked stack and a globally locked priority queue. These two data structures are still inherently sequential since they can only serve one thread at a time, but they can be used in a multi-threaded environment. Implementation-wise, this is achieved with an `std::mutex` and an `std::lock_guard`.

6.4.3 Treiber Stack

To compare the effects of relaxation, it is important to integrate non-relaxed concurrent data structures to compare against. For the non-relaxed concurrent stack, the Treiber Stack is selected since it is one of the most well-known and widely used lock-free concurrent stack algorithms [73]. The Treiber Stack is a lock free stack based on a singly linked list whose top pointer is updated using atomic compare-and-swap operations, providing a simple lock-free implementation with strict LIFO semantics.

An implementation [74] is found in a popular GitHub repository containing a library of concurrent data structures and integrated into the project. The implementation additionally uses an elimination optimisation, where concurrent push and pop operations can potentially cancel each other out directly without accessing the central stack structure. Threads may exchange values through auxiliary elimination slots, reducing contention on the shared stack top. Under high concurrency, this can substantially improve scalability while still preserving the observable semantics of the stack.

6.4.4 Lindén

The same motivation applies to priority queues. To evaluate the effects of relaxation, it is useful to compare relaxed concurrent priority queues against a strict linearisable baseline. Lock-free skiplist-based concurrent priority queues were first proposed by Sundell and Tsigas [75], [76], whose design ensures linearisability by preventing threads from advancing past a list element that has not yet been fully removed, with concurrent threads helping to complete the removal. Building on this line of work, a later lock-free skiplist-based concurrent priority queue developed by Lindén and Jonsson [77] is integrated into the framework using an existing GitHub implementation [78]. The data structure is based on a concurrent skiplist and provides linearisable priority-queue semantics. Its main design goal is to reduce contention on `deleteMin` operations by minimising the number of shared-memory updates required during deletions. Skiplists are well suited for concurrent priority queues because they maintain sorted order while allowing different threads to operate on different regions of the structure concurrently. Throughout the remainder of this report, this priority-queue implementation will be referred to simply as Lindén.

6.4.5 The MultiQueue

The driver code is written with the MultiQueue operations' type signatures in mind, thus the MultiQueue is automatically natively supported by the generic framework. The MultiQueue as an underlying queue supports specifying parameters such as the stickiness value. Since the MultiQueue is a type of priority queue, it enables a best-first search ordering through the BnB search tree.

The MultiQueue exposes a particularly interesting parameter, namely the stickiness value. As described in Section 2.4.1, the stickiness value controls how long a thread continues working on a sub-queue before moving on to another. A high stickiness value biases the MultiQueue towards improved cache locality and less contention.

An experiment is made and presented further on to compare the result of varying stickiness values. The parameter controlling the number of sub-queues (referred to as the queue factor) is left at its default value in the implementation, which is two. This means the number of sub-queues will be twice as many as the number of threads. The reason this parameter is fixed is because we believe it already strikes an adequate balance between relaxation and contention. Results reported in [23] indicate that a queue factor of two achieves the best overall balance between throughput and quality, whereas larger queue factors lead to substantially worse rank errors.

Another parameter exposed by the MultiQueue implementation is the sub-queue selection strategy. The available strategies are summarised below:

`mq_random` The baseline randomised strategy performs independent random sub-queue selection for every operation. Insertions are placed into a uniformly random sub-queue, while deletions sample a fixed number of distinct random sub-queues and remove the element with the best observed priority among them.

`mq_random_strict` This strategy uses the same random sampling procedure as `mq_random`, but rejects stale deletion decisions. Since candidate priorities are read before locking, another thread may modify the selected sub-queue before the lock is acquired. The strict variant verifies that the locked sub-queue still contains the sampled top key; otherwise, the operation is retried.

`mq_stick_random` The sticky variants reduce reselection frequency by reusing the same candidate sub-queues for multiple operations. In `mq_stick_random`, each handle stores a private random candidate set that is reused for both insertions and deletions until either the stickiness counter expires or a lock conflict forces reselection.

`mq_stick_mark` This strategy extends sticky random selection with soft ownership. sub-queues remember the identifier of the last handle that marked them. After the first operation in a sticky period, a handle avoids sub-queues marked by other handles, thereby reducing interference between workers while still permitting reassignment at sticky-period boundaries.

`mq_stick_swap` Sticky selection is implemented through a shared permutation of sub-queue assignments. Each handle uses its assigned positions as candidates, while reselection is performed by swapping assignments with random positions in the permutation. Compared with independent random reselection, this approach keeps candidate assignments globally coordinated and avoids duplicate ownership patterns caused by independent random draws.

An important detail is that stickiness only affects the strategies whose names contain `mq_stick_`.

6.4.6 The MultiLIFO

To support a data structure analogous to the MultiQueue but suited to depth-first search, we turn to the MultiFIFO queue. As explained in Section 2.5, it works roughly the same as the MultiQueue, but replaces the internal priority queues with FIFO queues, and compares timestamps of insertions instead of minimum keys during pops. To model DFS ordering, the MultiFIFO queue must be transformed from FIFO to LIFO ordering, meaning the internal queues must be turned into stacks, and the comparison of timestamps must be reversed. This allows the most recent elements to be popped instead of the least recent, which would enable DFS through the search tree. A fork is made of the MultiFIFO repository [79], and the changes just described were made. Let us call the new data structure the *MultiLIFO*. Both the MultiQueue and the MultiLIFO data structures are available to benchmark, and they support best-first and depth-first search orderings respectively.

6.4.7 2D-stack

The 2D-stack is also worth integrating as a selectable data structure, since it may perform even better than the MultiLIFO. As explained in Section 2.6, it is a state-of-the-art relaxed concurrent stack. A simple wrapper is built around it to integrate it into the project. The width parameter is automatically set to twice the number of threads, since it controls the number of sub-stacks used by the threads directly. The depth parameter is configurable during compilation using a `--depth` command-line argument.

6.4.8 Simple Work Stealing

As another point of comparison, a simple work-stealing solver is created. In this approach, each thread has its own local queue that it uses to enable a DFS search. If a pop fails because the local queue is empty, the thread tries to steal work from other threads' queues. The data structure used for stealing is a Chase–Lev [80] work-stealing deque, implemented by Conor Williams in the `ConcurrentDeque` GitHub repository [81]. Unlike the other competitor solvers, this one is built by us. This solver serves as a simple reference point for a simple-to-implement alternative that is also generic.

6.5 Existing Parallel Max Clique Solvers

In order to ground the solvers that our generic framework can produce and put them into a frame of reference, other existing solvers are compared against. The following sections introduce these alternative solvers and how they were implemented.

6.5.1 PMC

PMC [82] is an open-source high-performance maximum clique solver written in C++. The library cannot parse DIMACS instance files directly, so a fork had to

be created along with an extension to support this feature. This fork is imported into our project as a Git submodule. The PMC library exposes a function that returns the maximum clique size of a given graph instance file, which allows it to be integrated as an external benchmark target within the codebase. This way, PMC can be executed and benchmarked using the same command-line interface as the other implementations through the `--target` flag, despite not using the internal framework abstractions.

6.5.2 PBS

Since PBS [37] is identified in Chapter 4 as one of the most competitive parallel exact maximum clique solvers, it is also included as a competitor benchmark. The original implementation is obtained from the authors and is kept in a separate repository to keep it private. Instructions for how to compile and run PBS were given by the authors and no changes were made to the source code. Thus, PBS is not integrated into the same benchmarking suite as the rest and has to be run individually. PBS serves as an important reference point when comparing solvers, since it represents the state-of-the-art for solvers specifically engineered for the maximum clique problem. Recall that PBS is based on MoMC [64], thus we do not include MoMC as a separate solver for benchmarking.

6.5.3 McCreesh

The source code [83] for the parallel maximum clique solver by McCreesh and Prosser [36], raised in Section 4.4, is also found. Despite not being as recent as PBS, it still serves as an interesting point of comparison. It is integrated into the project as an external benchmark into the benchmark suite. Throughout the rest of the report, this solver will be referred to simply as *McCreesh*.

McCreesh's maximum-clique solver was configured to use the threaded `tbmcsa1` algorithm. This is a bitset-based threaded implementation of San Segundo's BBMCI [63] algorithm using the simpler BBMC1 initial vertex ordering. The algorithm performs branch-and-bound search using bitset colour sorting to calculate upper bounds and an atomic shared incumbent.

Work donation was disabled. Instead, the search was initially partitioned at depth one, with first-level subproblems placed into a shared queue and processed by the configured number of worker threads. Consequently, workers that became idle could consume queued first-level subproblems but could not receive deeper subproblems donated by other workers. The solver additionally used one producer thread while generating the initial work.

No initial incumbent, early termination target, or internal timeout was configured. The benchmark batch-size parameter was ignored by this solver. Scalability experiments used between 1 and 512 worker threads, while the comparison experiments used 512 worker threads.

Further work could test other configuration of McCreesh's maximum-clique solver,

since it is out of scope for this thesis.

6.5.4 CliSAT

We do not include CliSAT as a benchmarkable solver due to time constraints. However, future work could add it as a sequential competitor, along with MoMC.

6.6 Artifact Availability

The source code developed in this thesis, including the generic branch-and-bound framework, queue implementations, benchmarking infrastructure, and experimental scripts, is available as open source [84].

7

Experimental Evaluation

This section describes the environment and experimental setup used to evaluate the proposed implementations. It outlines the execution environment, benchmark selection, and evaluation metrics used to assess performance, scalability, and the effects of relaxed priority semantics. It then presents benchmark evaluations of the selected maximum clique BnB problem, together with the measured metrics. These evaluations are compared against existing solvers and in particular, state-of-the-art within maximum clique solvers. Finally, it presents benchmark evaluations on a high level for the other BnB problems implemented in this project.

7.1 Benchmark Environment

This section outlines the benchmark environment used for running the benchmarks presented in this chapter. It starts with a description of the benchmarking server, then continues with information regarding compilation, and lastly highlights the solver configurations used for the benchmarks.

7.1.1 System Description

All experiments are conducted on a multicore benchmarking server named Athena. Athena is a dual-socket system equipped with two AMD EPYC 9754 processors running at 2.25 GHz. Each processor contains 128 physical cores with two-way simultaneous multithreading (SMT), yielding a total of 256 physical cores and 512 hardware threads available to the operating system. Each processor provides 256 MB of shared L3 cache, for a total of 512 MB L3 cache across the system. The machine is equipped with approximately 755 GB RAM and runs OpenSUSE Tumbleweed with the Linux 7.0.1-1-default kernel.

7.1.2 Language Environment and Compilation

The generic BnB framework is written in C++ and compiled using the C++20 standard. Some of the integrated implementations are compiled with other versions and are linked together. All benchmarks are compiled with architecture-specific optimisations enabled through the `-march=native` compiler flag in order to fully utilise the instruction set and microarchitectural features available on the benchmarking server. As mentioned earlier, `CMake` is used as the meta build system for all compilation.

The implementation uses POSIX threads through a custom `pthread` wrapper. Worker threads are created by the framework dispatcher, which by default assigns each thread a specific CPU affinity mask using the `FarL1CloseL3` policy. This policy orders hardware threads according to the detected cache and socket hierarchy, and the resulting affinity mask is applied during thread creation using `pthread_attr_setaffinity_np()`.

7.1.3 Solver Configurations

Solvers are instantiated from the generic framework by connecting the problem instance to different underlying queues. The underlying queues used for benchmarking are the ones described under Section 6.4, and are reiterated in Table 7.1

Queue type	Thread-Safe	Relaxed	Exploration Order
Sequential Stack	No	No	DFS
Globally Locked Stack	Yes	No	DFS
Treiber Stack	Yes	No	DFS
MultiLIFO	Yes	Yes	DFS
2D-stack	Yes	Yes	DFS
Sequential Priority Queue	No	No	Best-first
Globally Locked PQ	Yes	No	Best-first
Lindén	Yes	No	Best-first
MultiQueue	Yes	Yes	Best-first

Table 7.1: Overview of the integrated underlying queue data structures.

Whenever relevant, the solvers built from our generic framework are compared against other complete solvers, highlighted in Section 6.5, namely:

- PMC
- Simple Work Stealing
- PBS
- McCreesh

7.1.4 Evaluation Metrics

The evaluation metrics presented in the benchmarks are firstly those the driver code records, described in Section 5.2.3, which are execution time and counters for the number of processed and ignored nodes. Also, other metrics derived from these are presented, such as speedups or slowdowns compared between configurations.

7.2 Benchmarking Scripts

To simplify and streamline the benchmarking process, a set of python scripts were written. The python scripts execute benchmark runs and parse their metrics written

to standard output. They also catch and report crashes and out-of-memory errors. Out-of-memory errors occur when the solver process tries to use more memory than the operating system allows. With these scripts, we can also set timeouts for how long each individual run is allowed to take. If a run exceeds the timeout, it is simply recorded as a timeout. If the solver crashes, the exit code of the solver process is recorded.

7.3 Evaluation of the Maximum Clique Case Study

The following section presents the results of the case study on the maximum clique problem. Since some of the DIMACS graph instances are still unsolved, an initial filtering is done to avoid wasting computational resources on known extremely hard inputs during subsequent analyses. An analysis and discussion on the effects of the stickiness and batching parameters is also done in preparation for the final experiments. A smaller set of graphs are chosen to analyse how the different implementations compare in terms of scalability and single-threaded performance. Finally, a comparison is made on all the filtered inputs for a more complete view on the effectiveness of each implementation. In all following experiments, a timeout of 30 minutes was set for the individual runs.

7.3.1 Filtering of Input Graphs

Filtering is done by excluding instances for which a sequential solver takes longer than ten minutes, or shorter than a second to solve. To save time, results are inspected from McCreesh's paper on parallel max clique [36], where sequential runtimes are reported. The instances from said results which obey our one-second to ten-minute filtering rule are noted down. After this filtering, 21 out of the original 80 instances are left. The filtered DIMACS graph instances are presented in Table 7.2 These are used in the comparison benchmark that is presented in Section 7.3.5. Thus our analyses can more accurately show how the implementations compare without having to wait for days. Instances with very short solution times are not very interesting because there is not much room for differences in performance to have time to show.

For the more investigative benchmarks, it would be time-consuming to run them on all these 21 inputs. Besides, it is possible to get insight even using the same input, if the purpose is to find out how implementations scale as parameters are increased. Thus, three random inputs are selected for these detailed analyses. These are `brock400_4`, `DSJC1000_5`, `gen200_p0.9_44`.

7.3.2 Analysing the Effects of Stickiness

To analyse the effects of stickiness, a benchmark runs the framework-instantiated solvers with the MultiQueue and MultiLIFO as the underlying queues, using the parallel driver code. The stickiness parameter ranges from one to 1024, increasing by a factor of four in each benchmark. The thread count is fixed to the maximum

Table 7.2: Filtered DIMACS maximum clique benchmark instances used in the evaluation.

Instance	**Nodes**	**Edges**	**Density**	**Solution**
DSJC1000_5	1,000	499,652	0.500	15
DSJC500_5	500	125,248	0.502	13
MANN_a45	1,035	533,115	0.996	345
brock400_1	400	59,723	0.748	27
brock400_2	400	59,786	0.749	29
brock400_3	400	59,681	0.748	31
brock400_4	400	59,765	0.749	33
gen200_p0.9_44	200	17,910	0.900	44
p_hat1000-2	1,000	244,799	0.490	46
p_hat1500-1	1,500	284,923	0.253	12
p_hat300-3	300	33,390	0.744	36
p_hat500-3	500	93,800	0.752	50
p_hat700-2	700	121,728	0.498	44
p_hat700-3	700	183,010	0.748	62
san1000	1,000	250,500	0.502	15
san200_0.9_3	200	17,910	0.900	44
san400_0.7_2	400	55,860	0.700	30
san400_0.7_3	400	55,860	0.700	22
san400_0.9_1	400	71,820	0.900	100
sanr200_0.9	200	17,863	0.898	42
sanr400_0.7	400	55,869	0.700	21

available on Athena, which is 512. Each run is repeated 25 times and the reported times are averaged and presented in Figure 7.1, for one of the selected inputs. In addition to varying the stickiness value, the experiment also tests different stickiness reassignment strategies described in Section 2.4.1.

In the figure it can be seen that the different sub-queue reselection strategies and stickiness values do not play a significant role up to a stickiness value of 64, for the MultiQueue. At 256 and 1024, the stick random and stick swap strategies begin to suffer from degraded performance. For the MultiLIFO, it seems to benefit specifically from a stickiness value of 64. Recall that stickiness only affects the strategies stick mark, stick random and stick swap.

The default stickiness value for both the MultiQueue and the MultiLIFO is 16, and the default sub-queue selection strategy is stick swap. Due to a timeline oversight, we chose to use the default stickiness and sub-queue selection strategies for both the MultiQueue and MultiLIFO. For the MultiQueue, this is acceptable since no other configuration performed much better. For the MultiLIFO however, going from the stickiness value of 16 to 64 would–according to the result on brock400_4–more than double the performance. Thus, the default stickiness value might not have been an optimal choice for the MultiLIFO. On the other hand, we have to keep in mind that this stickiness experiment is by no means exhaustive and what performs best on this specific input might not be the best in all scenarios. In addition, the default values are the ones recommended by the papers [23], [25].

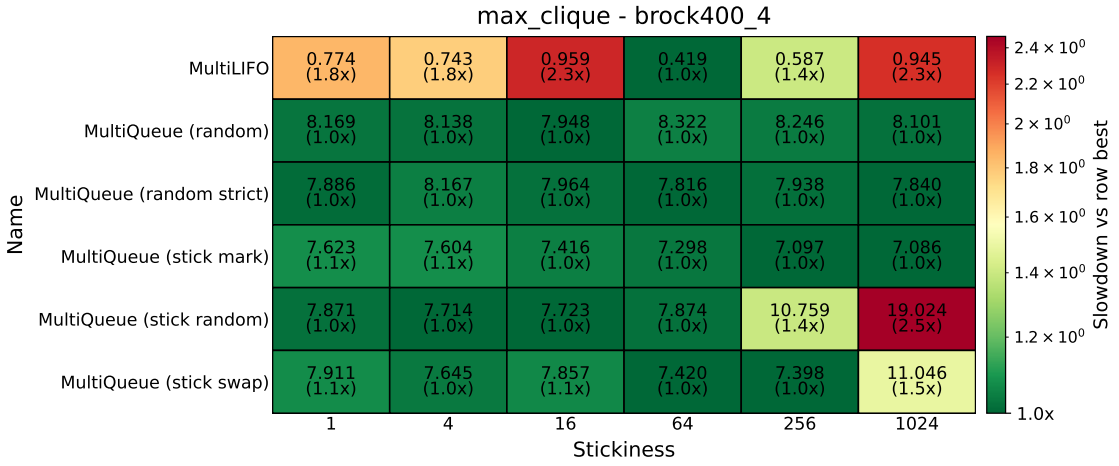

Figure 7.1: Maximum clique benchmark results for the stickiness analysis on the brock400_4 instance, showing execution time for different configurations of the MultiQueue, and the MultiLIFO, for different stickiness values. Each tile displays the execution time and the slowdown compared to the fastest execution time in the same row.

7.3.3 Analysing the Effects of Batching

To analyse the effects of batching, as introduced in Section 5.2.4, a benchmark very similar to the stickiness benchmark is run. This time, however, solvers are

instantiated from all selectable underlying queues, since the batching parameter controls driver code behaviour and applies to all individual queues. Similar to the stickiness experiment, the batching parameter ranges from one to 1024, increasing by a factor of four in each experiment. The thread count is once again fixed to 512. Each run is repeated 25 times and the reported times are averaged and presented in the heatmaps shown below, one for each of the three selected inputs.

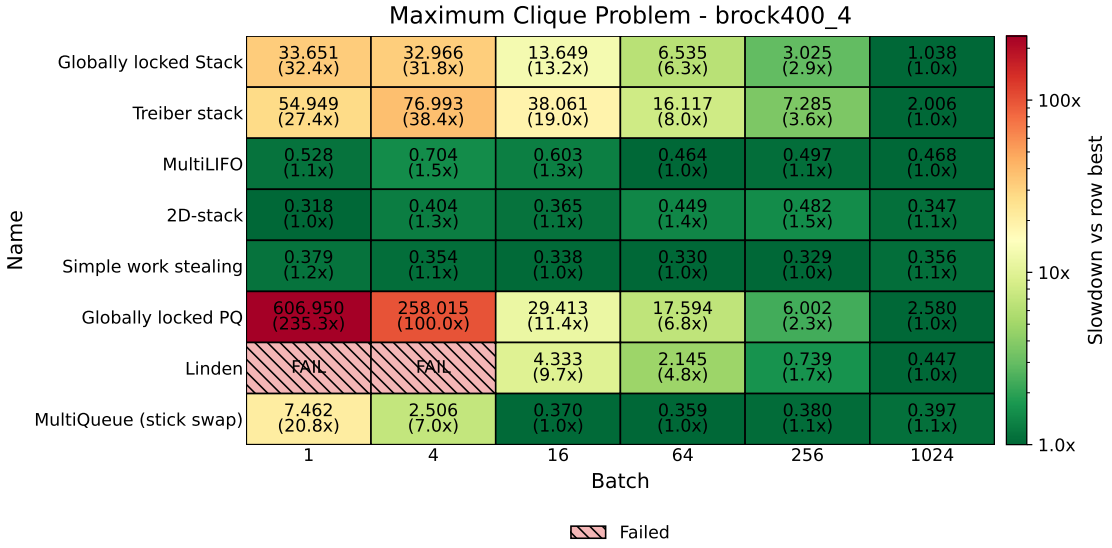

Figure 7.2: Maximum clique benchmark results for the batch analysis on the brock400_4 instance. The heatmap shows execution time for each underlying queue for different batch sizes. Each tile also displays the slowdown compared to the fastest execution time in the same row.

Figures 7.2, 7.3, and 7.4 show heatmaps of execution times for different batch sizes, for each framework-based parallel solver. The figures show that some underlying queues benefit more from increased batching values than others, in the max clique problem. Since batching effectively reduces the frequency at which the algorithm accesses the underlying queue, one could expect the underlying queues that react poorly to contention to benefit most from it. This is essentially what the figure shows. The globally locked stack and priority queue only allow access from one thread at a time, and do not scale well with increasing thread counts at all. These two show the highest relative speedup as the batch value increases from one to 1024, however their absolute run times are still slow compared to the rest.

Interestingly, some underlying queues do not seem to benefit from batching at all. Simple Work Stealing, MultiLIFO and the 2D-stack have approximately constant execution time regardless of the batch value. This shows that a higher batch size is not necessarily better.

The MultiQueue is already built to handle contention well. However it still seems to enjoy a performance boost with a higher batch. One conclusion from this is that the MultiLIFO handles contention better than the MultiQueue, probably due to internal stack operations being quicker than internal priority-queue operations.

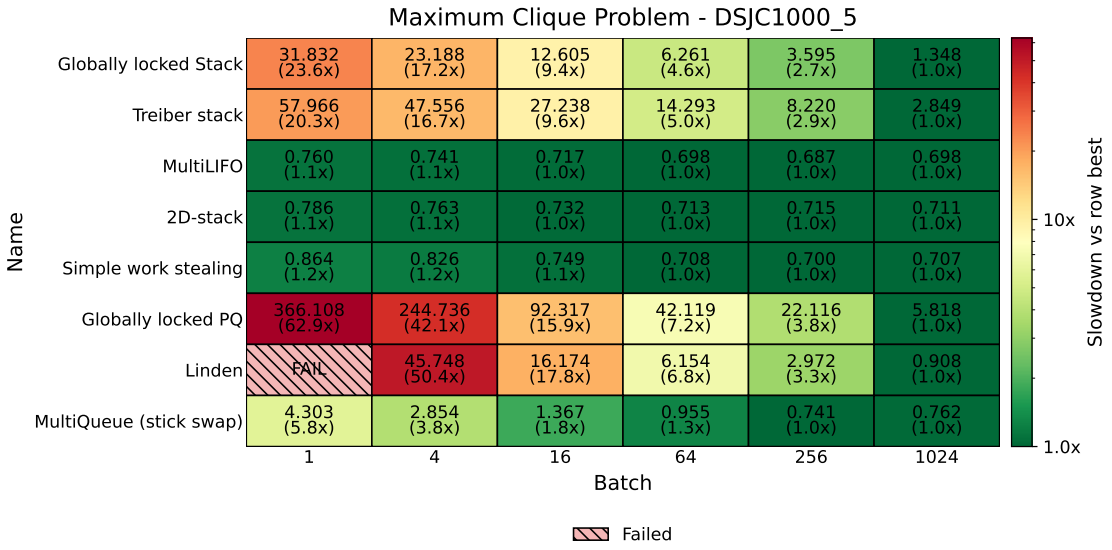

Figure 7.3: Maximum clique benchmark results for the batch analysis on the DSJC1000_5 instance. The heatmap shows execution time for each underlying queue for different batch sizes. Each tile also displays the slowdown compared to the fastest execution time in the same row.

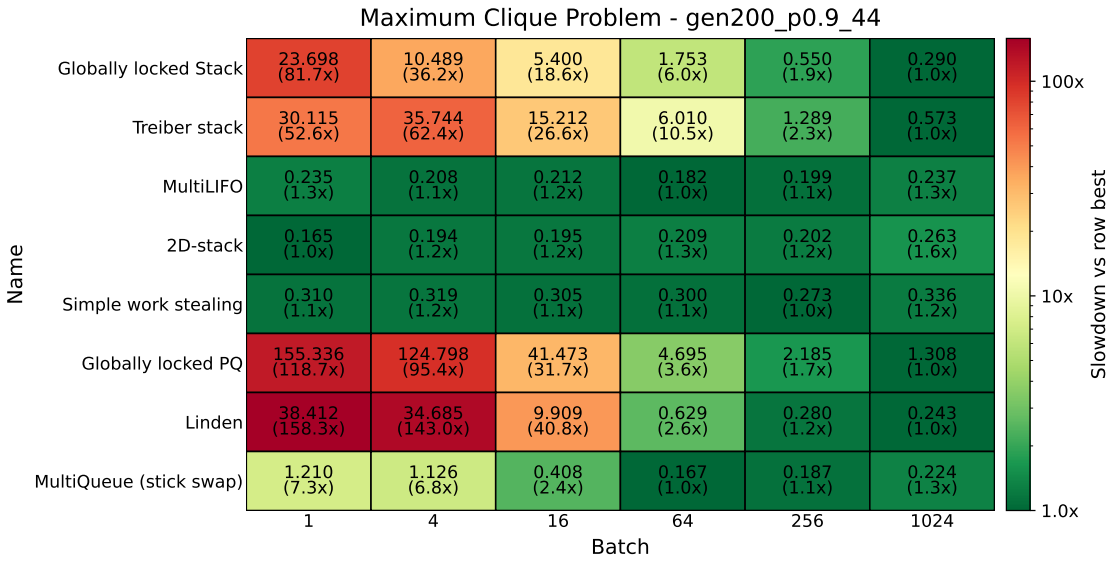

Figure 7.4: Maximum clique benchmark results for the batch analysis on the gen200_p0.9_44 instance. The heatmap shows execution time for each underlying queue for different batch sizes. Each tile also displays the slowdown compared to the fastest execution time in the same row..

Another could be that batching introduces DFS-behaviour via a local stack, which has a lower time complexity than a priority queue. This could lead to faster execution time, but we think that would only benefit by a small factor.

The question is whether batching is reasonable to use in our main evaluations, and if so, how the value should be chosen for each underlying queue. None of the underlying queues seem to worsen with a higher batch value (at least to 1024), so it seems like the higher the better. On the other hand, the higher the batch size, the less frequent the work gets shared. Frequent work distribution is one of the strong points of our framework. Also, avoiding access to the underlying queue defeats the aim of comparing different data structures. This is because the higher the batch size, the more time is spent doing a local DFS instead of using the shared underlying queue. Barely accessing the shared underlying queue means its ability to handle contention and ordering of nodes plays less of a role. Since the goal is to lean into and measure the effects of relaxation on the underlying queue level, it would be fine to alleviate the contention slightly but not try to avoid it too much. The decision was made to keep using batching, but fix the batch size to a small but still beneficial value. The batching value was fixed to 16 in the rest of the experiments, since this seemed like a reasonable middle ground between work distribution and contention avoidance.

Note how the Treiber stack seems to perform worse than a globally locked stack, even though the thread count is 512. This is surprising, since the Treiber stack is designed to solve the concurrency issues of a globally locked stack. This could be due to an implementation or setup issue, but it is not certain. Also note that the Lindén underlying queue shows `FAIL` in some places. In the benchmark script used to get the results, out-of-memory and timeout errors are recorded, but any other error/crash is recorded as "FAIL" along with the program return code. The return codes for the fails were `6`, corresponding with the POSIX signal `SIGABRT`. The causes have not been identified, and no attempts were made to do so since the rest of the results were satisfactory.

7.3.4 Scalability Analysis

The scalability analysis aims to show how the different underlying queues respond as thread counts increase, and how the competing solvers behave. The batch size is fixed to 16 in the generic framework and the same three inputs as in the batch analysis are used. The thread count ranges from 1 up to 512, increasing by a factor of two. To prevent this benchmark from taking longer than a day, only five repetitions are performed for each configuration this time, rather than 25, before taking the average.

Figure 7.5 shows both a heatmap and a line plot of execution times on the brock400_4 instance for different solvers, varying the thread count. Both execution time and speedup compared to each solver's single-threaded performance is shown in this figure. The sequential solvers (PQ and stack) intentionally only show up as a dot in the line plot since they were designed to run with a single thread only. One can see that the globally locked data structures perform better up until a certain point, where

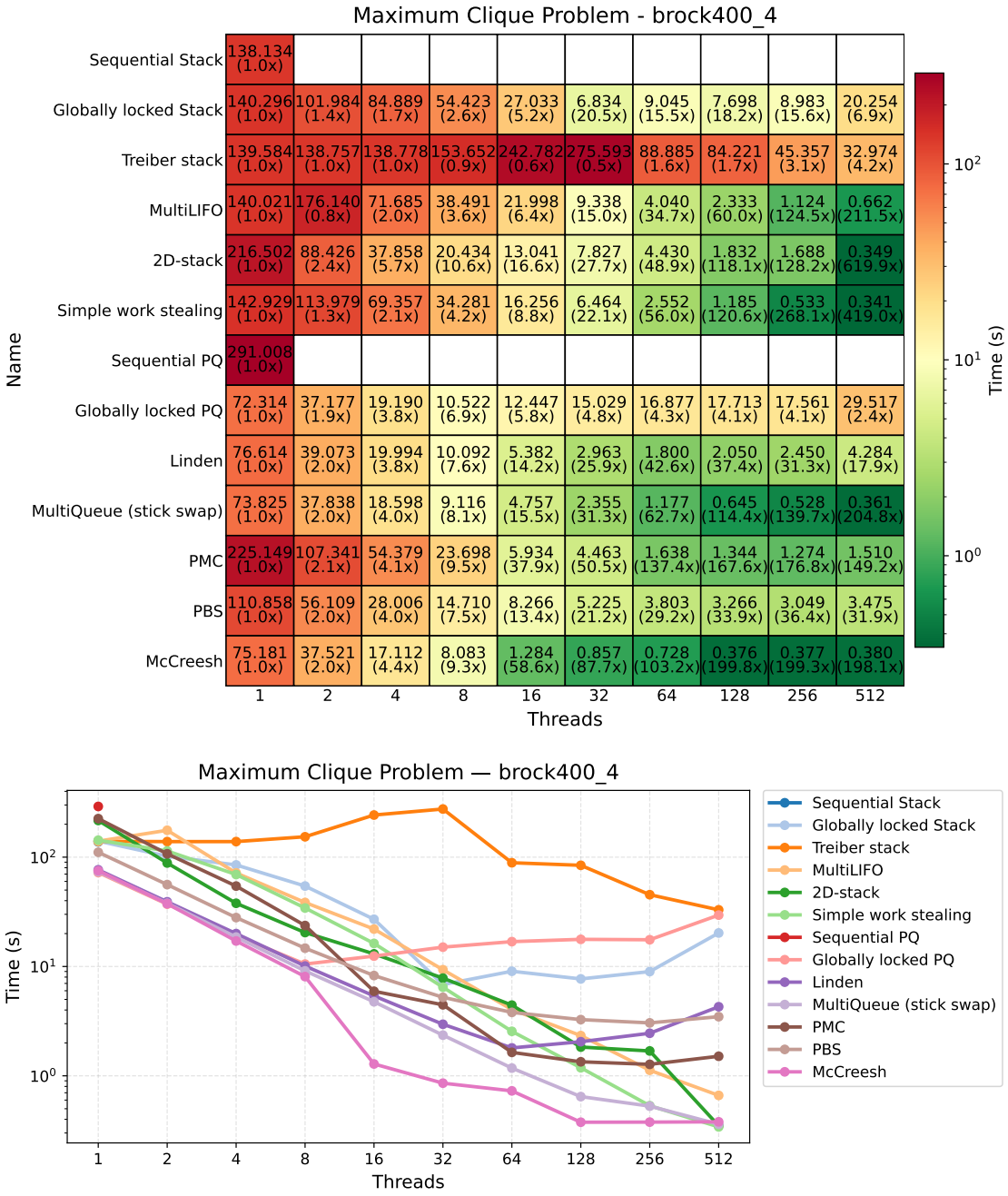

Figure 7.5: Maximum clique benchmark results for the scalability analysis on the brock400_4 instance. Both plots show execution time as a function of the thread count. The heatmap also shows the speedup compared to the single-threaded performance in the first column.

they quickly fall off and cannot handle the increased thread count. Once again, the Treiber stack seems to be an anomaly and is mostly disregarded in this analysis. Only a few implementations are capable of holding up to the high thread count of 512, namely the MultiQueue, Simple Work Stealing, the 2D-stack, McCreesh, and possibly the MultiLIFO, while most of the remaining implementations fail to scale similarly at higher thread counts. The highest speedup belongs to the 2D-stack, with a 619.9X speedup over its single-threaded execution time. It is the only one in the group with a superlinear speedup. To better see the scalability of each solver, Figure 7.6 displays a heatmap over their strong scalability. Strong scaling is computed as speedup divided by thread count. A strong scalability of 100% correlates with linear speedup, and any value over that indicates a superlinear speedup.

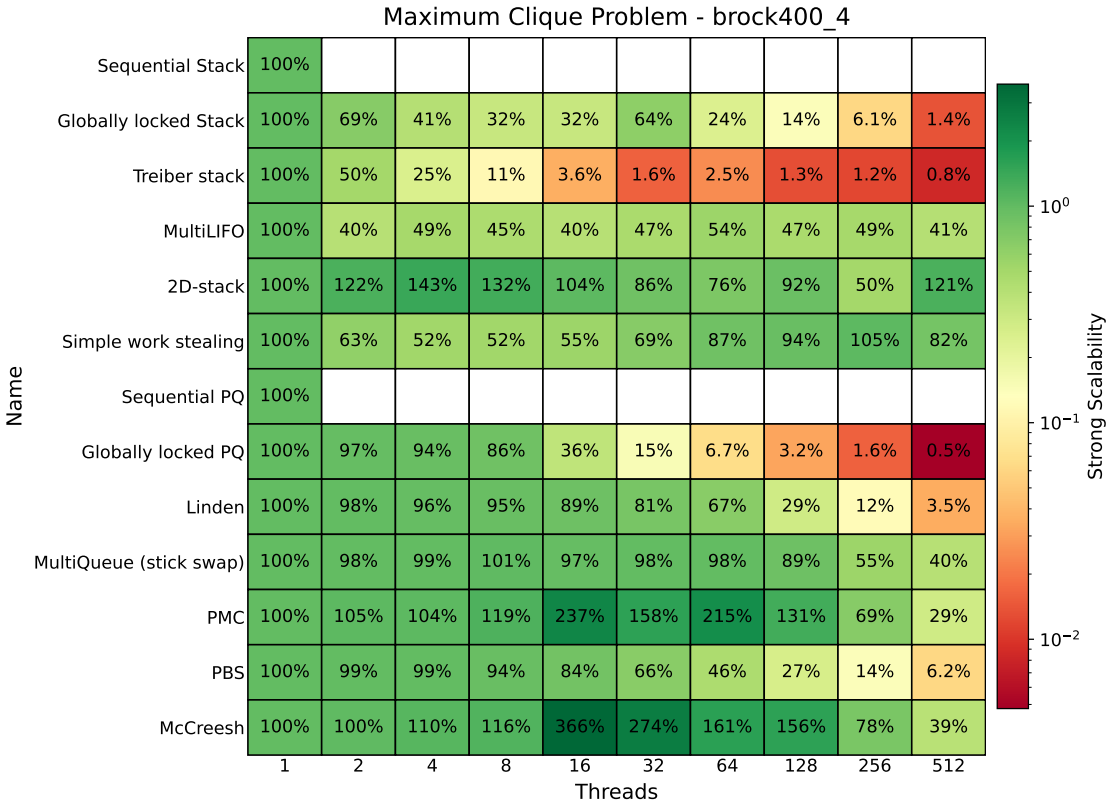

Figure 7.6: Maximum clique benchmark results for the scalability analysis on the brock400_4 instance, showing strong scaling for all solvers over a range of thread counts. Strong scaling is computed as speedup divided by thread count.

Figure 7.7 presents execution time of the different solvers as the thread count increases from one to 512, for the DSJC1000_5 graph instance. The figure shows broadly similar scaling behaviour to that observed for `brock400_4`. However, the MultiQueue performs slightly worse than before, and starts to fall off after 128 threads. The winner here is McCreesh with a final time of 0.464, with a speedup of 156.4x. What is interesting is that McCreesh stops scaling linearly after 128 threads, and only performs slightly better with 512 than 128, and even worse at 512 compared to 256. This shows that it cannot untilize SMT that well, and is especially not optimized for NUMA. The 2D-stack, Simple Work Stealing and MultiLIFO come right

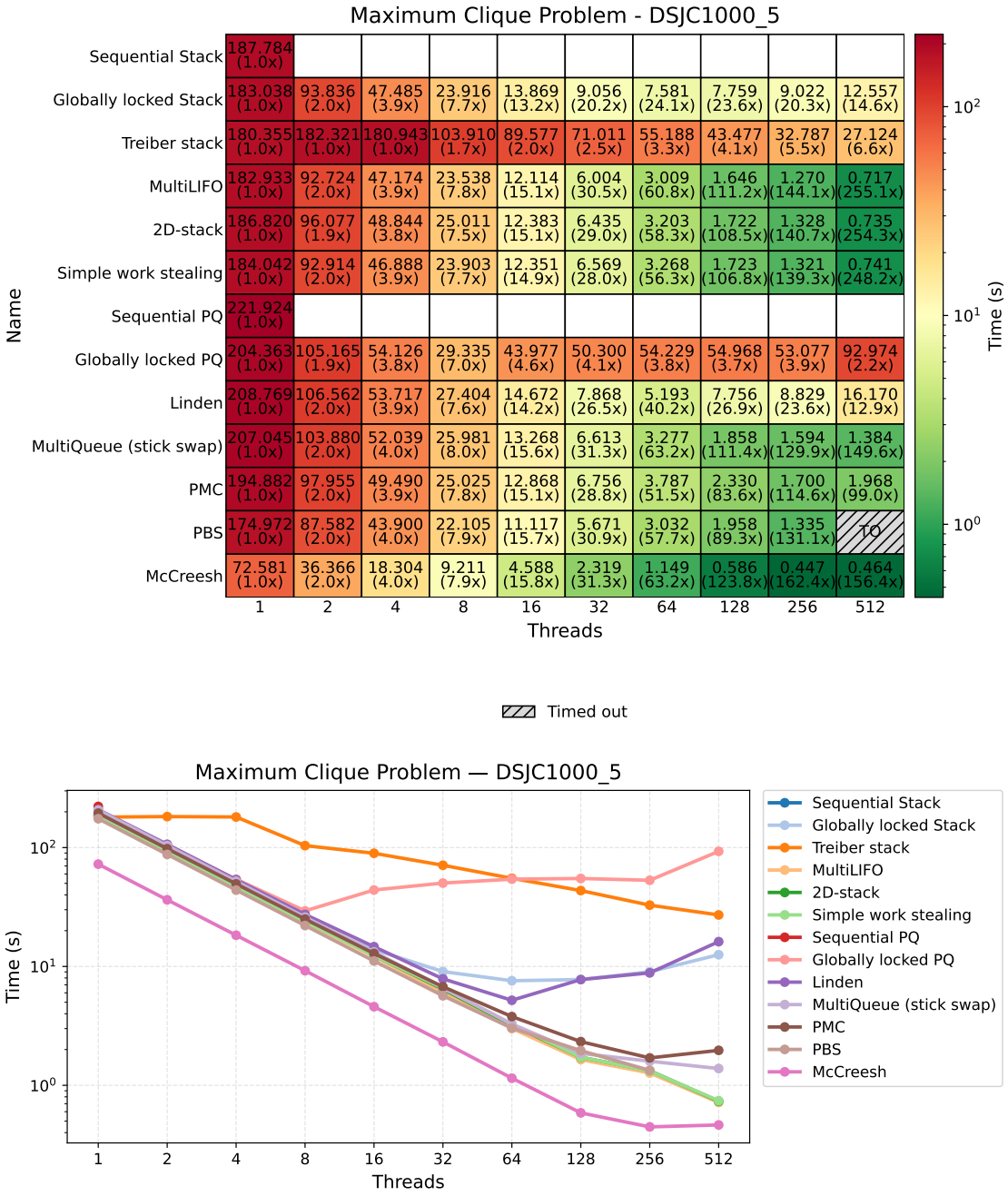

Figure 7.7: Maximum clique benchmark results for the scalability analysis on the DSJC1000_5 instance. Both plots show execution time as a function of the thread count. The heatmap also shows the speedup compared to the single-threaded performance in the first column.

after, all of them with a speedup of around 250X compared to their single-threaded execution time. Note how these continue to scale past 128 threads, however they do not achieve linear speedup at 512 threads. Interestingly, PBS timed out with 512 threads, even when the trend showed it should have completed in around one second. We ran PBS without any changes to the source code and used the compilation and running instructions given by the authors.

Figure 7.8 once again presents execution time for different thread counts, but for the gen200_p0.9_44 graph instance. This time, we see a very different picture. At a single thread, PBS is several orders of magnitude faster than our implementations, with PMC and McCreesh being somewhere in between in the logarithmic scale, at around one second. At 512 threads, McCreesh seems to achieve the best performance with 42ms solve time, and only a 61.2x speedup compared to its single-threaded performance. However, McCreesh performed best with only 32 threads, meaning the additional 480 threads did nothing to help scale the performance. The rest of the solvers follow roughly the same pattern as seen with the previous two graphs, but need all 512 threads to come even close to state-of-the-art at its best. It is interesting however that PBS becomes slower as the number of threads surpasses eight. The reason PBS performs so much better in the single-threaded run could simply be because its sequential algorithm is better. Although we implemented bound computations with vertex orderings, colouring, and bit-level optimisations from earlier papers, our baseline implementation is still not as sophisticated as PBS's, which uses MoMC [64]. A slightly better vertex ordering or bounds computation with MaxSAT reasoning could prune huge parts of the search tree, leading to faster execution times similar to the one seen. Thus, it is not very surprising that some graphs could lead to this kind of behaviour, while others do not. Further research could focus on building a comparable sequential baseline implementation and see if that one could scale to compete with PBS on this instance.

Another reason PBS could perform so well for this specific graph is simply luck. It could happen that PBS visits nodes in a favourable search order that leads to the solution quickly specifically on this input graph, and any overhead from parallelism just slows down this process. We think it is a combination of both this and the fact that its baseline is better. However the fact still remains that other solvers (McCreesh, PMC, 2D-stack and MultiLIFO) could handle the immense thread count, when PBS could not, even when PBS had favourable single-threaded performance. This experiment shows that there could still exist a research gap to find a solver that has both strong single-threaded performance and can scale well even up to 512 threads.

Figure 7.9 shows the strong scaling on the gen200_p0.9_44 instance for the solvers over different thread counts. This figure shows very clearly how PBS scales so poorly on this specific instance, while the MultiLIFO, 2D-stack and Simple Work Stealing solver scale superlinearly. This does not necessarily mean that their parallelizations are better, as they never reach the same speed. However it does show that they scale well, and that PBS has a harder time parallelizing its sequential algorithm.

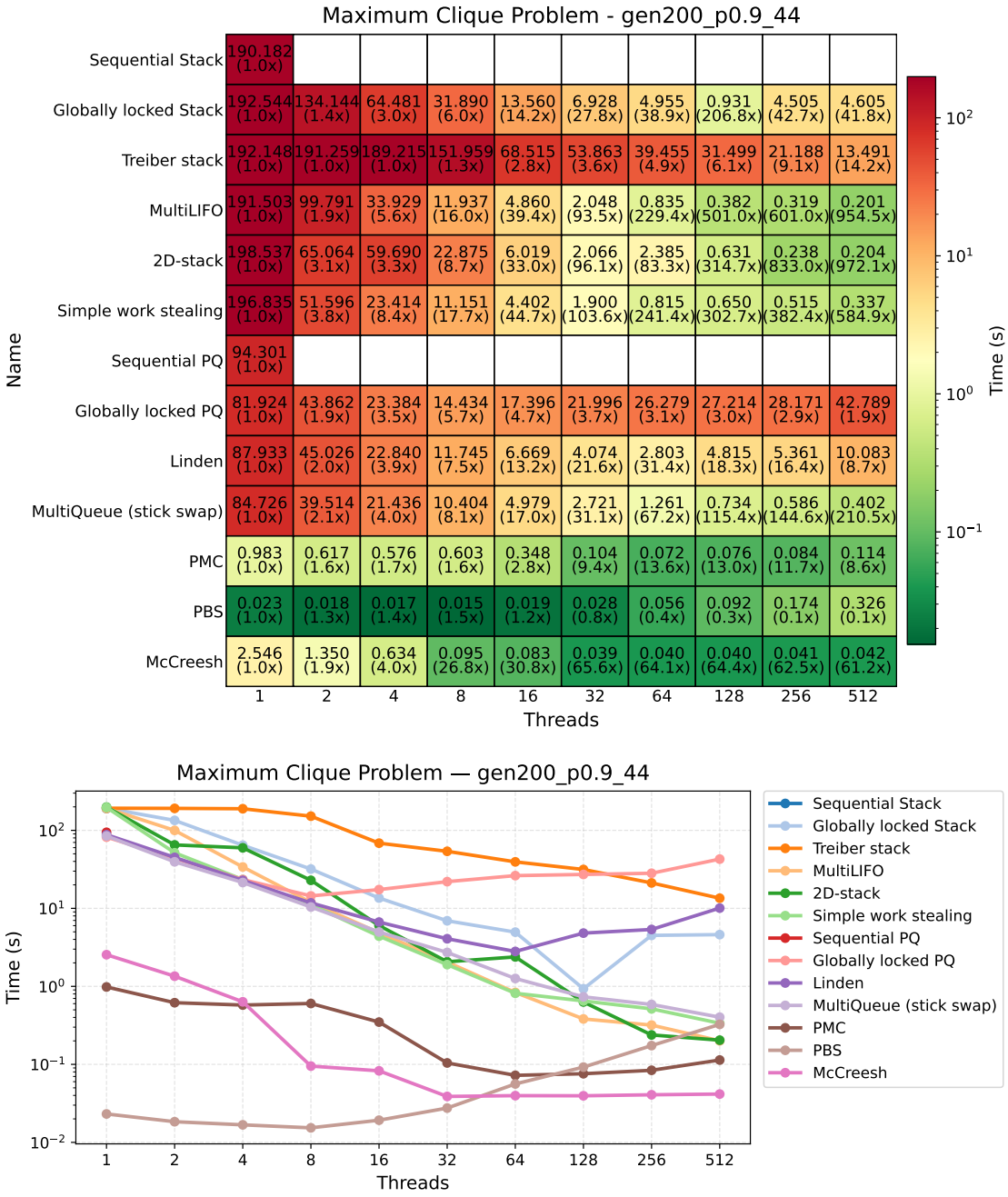

Figure 7.8: Maximum clique benchmark results for the scalability analysis on the gen200_p0.9_44 instance. Both plots show execution time as a function of the thread count. The heatmap also shows the speedup compared to the single-threaded performance in the first column.

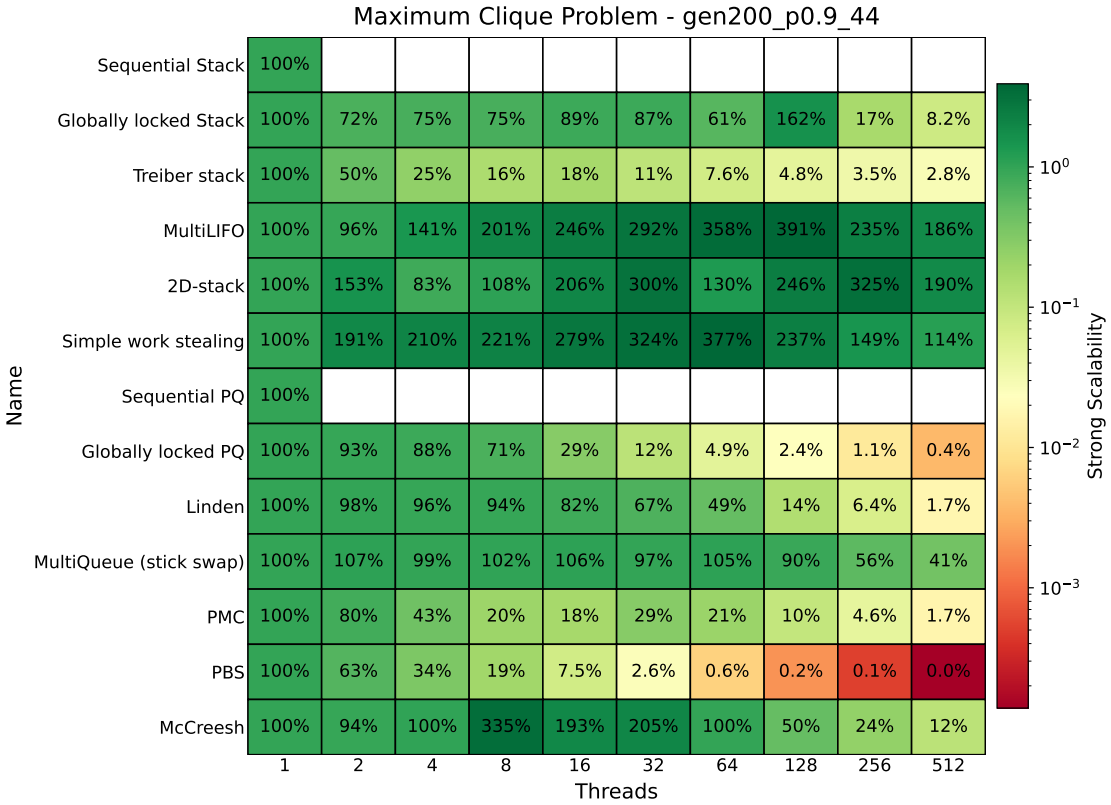

Figure 7.9: Maximum clique benchmark results for the scalability analysis on the gen200_p0.9_44 instance, showing strong scaling for all solvers over a range of thread counts. Strong scaling is computed as speedup divided by thread count.

Another reasonable metric to look at for the scalability experiment is the number of visited nodes. Figure 7.10 displays a heatmap over the number of visited nodes, which is defined as the number of processed nodes, plus the number of ignored nodes. Below each visited node count in the figure, the fraction of processed nodes compared to all visited nodes is reported as a percentage. With ignored nodes, we mean those about to be processed, but being pruned instead due to the incumbent comparison. Nodes within pruned subtrees, never even generated, do not count towards ignored nodes. Thus, the number of ignored nodes serves as a measure of how much pruning the solver performed. Our generic framework records the number of processed and ignored nodes. However, not all alternative solvers record and report this. For example, PMC does not tell us the number of visited nodes, while McCreesh reports the number of visited nodes, but does not separate them into processed and ignored. PMC is therefore excluded from the figure, while McCreesh is included but there is no information about the split between processed and ignored nodes.

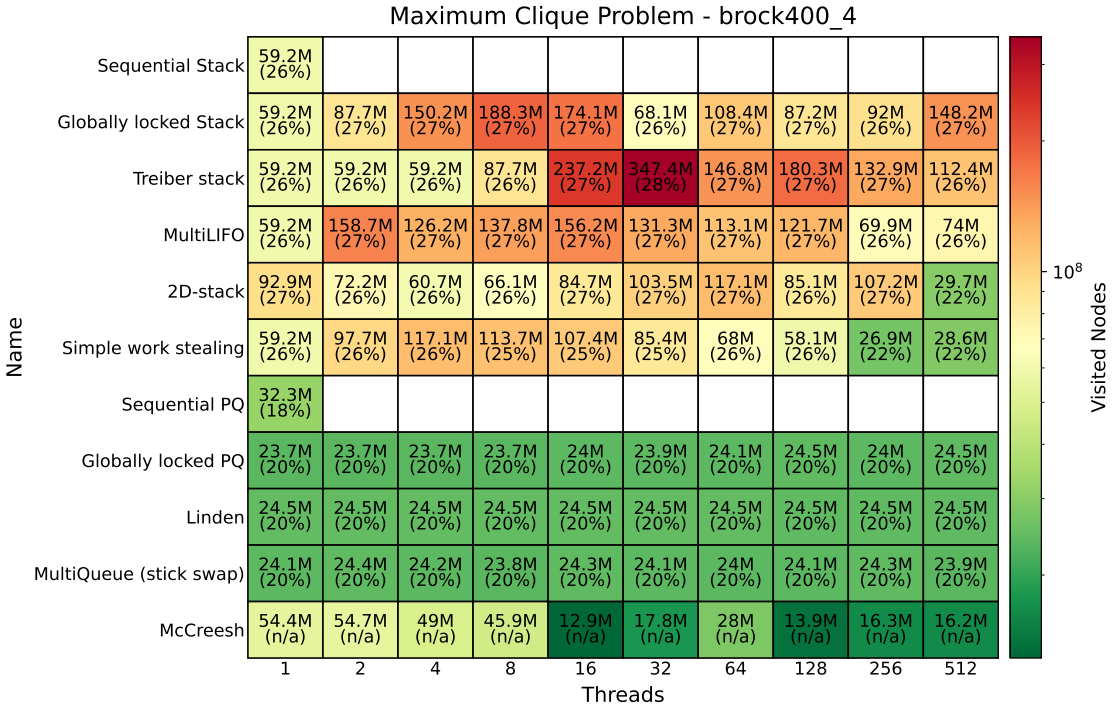

Figure 7.10: Maximum clique benchmark results for the scalability analysis on the brock400_4 instance, showing visited node counts, and the fraction of those being processed rather than ignored/pruned. The suffix M denotes millions.

In Figure 7.10, it can be seen that the priority-queue implementations (Globally Locked PQ, Lindén, and MultiQueue) have a relatively similar number of visited nodes. Thus, it seems like the effects of relaxation do not introduce more work in this case. One can draw the conclusion that with the same visited node count across all number of threads, the work is distributed across all threads, which should lead to a linear speedup. This is not entirely correct, since the node count does not take thread synchronisation overhead into account. In fact, Recall that in Figure 7.5, Lindén and the MultiQueue scale quite differently in terms of execution time, with

the MultiQueue reaching over 200x speedup whereas Lindén only reaches about 18x, even though they start off at roughly the same single-threaded performance. Also note that as expected, priority-queue-based designs visit far fewer nodes, showing how approximate upper-bounds treated as priorities guide the search space more effectively towards the final answer than stack-based designs. However, this did not necessarily translate into faster execution times, since the stack-based designs generally sustain higher throughput and lower synchronisation overhead. Nonetheless, this result serves as an indication for how much work is being done.

While the priority-queue-based solvers have stable and efficient work behavior, the depth-first versions vary significantly across thread counts. This is likely because depth-first search is unguided. In branch and bound, finding a good incumbent early is important, since it enables larger parts of the search tree to be pruned. With DFS, however, the search is not directed toward promising nodes, so different runs may find good incumbents at very different times. This highlights another benefit of using priority queues in branch and bound. They provide more predictable performance and usually require less total work, although they come with trade-offs such as lower throughput and higher memory usage.

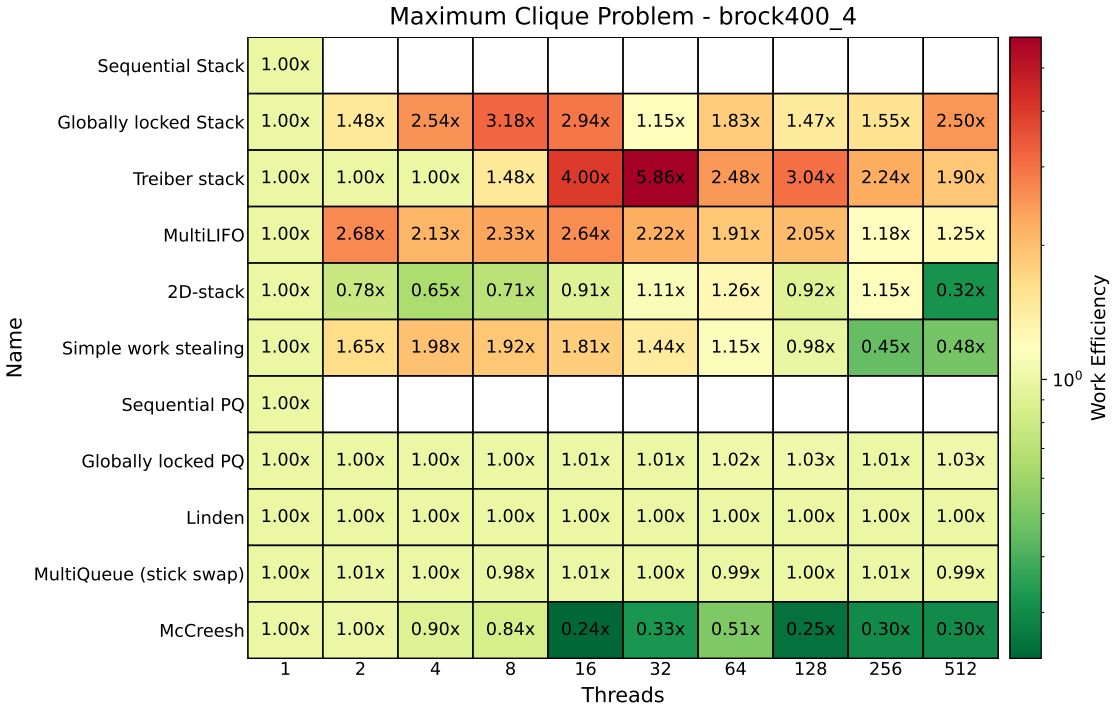

Figure 7.11: Maximum clique benchmark results for the scalability analysis on the brock400_4 instance, showing the fraction of visited node counts compared to the corresponding single-threaded visited node count.

From this, another heatmap can be made that shows the work efficiency introduced by parallelism, shown in Figure 7.11. We define work efficiency as the number of nodes visited by the parallel algorithm divided by the nodes visited by its single-threaded execution. This figure displays very clearly which solvers introduce more

or less work when the thread counts increases. McCreesh very clearly needs less work, while the priority-queue designs need the same amount of work. Once again, we see the instability of stack based designs. When comparing with the work on one thread, the 2D-stack performs less work at 512 threads, but more work at 64. The MultiLIFO needs to do more work for all its multi-threaded executions compared to its single-threaded work.

7.3.5 Comparison Across all Graphs

To obtain a better overview of the competing solvers, an experiment is run where the thread count is pinned to 512, and all parallel solvers are run on all filtered DIMACS graph instances. Batching is fixed to 16 for all the framework-instantiated solvers once again. The result of this experiment is presented as a heatmap in Figure 7.12.

Interestingly, the solver by McCreesh [36] seems to perform overall the best among the solvers, beating the other solvers on 13 out of the 21 graph instances. This is surprising since it is not as recent as PBS [37], and from our related work, we expected PBS to be the absolutely best performing solver.

The solvers instantiated from our generic framework only win five out of the 21 times. Among these, it is the MultiLIFO, 2D-stack, Simple Work Stealing and MultiQueue that performs best on those five instances. Both PMC and PBS wins once each. In general, the 2D-stack performs slightly better than the MultiLIFO on almost all instances, showing that the 2D-stack seems to be the more performant relaxed concurrent stack overall, out of the two. This is not surprising, since the 2D-stack doesn't have the overhead of creating or comparing timestamps each operation.

The MultiQueue performance seems to be roughly on par with the 2D-stack and MultiLIFO, being only slightly slower than them on most instances. This result is interesting since the MultiQueue follows a completely different search strategy than them. Relying on best-first searching via priorities of upper bounds, we would expect the MultiQueue to find the solution much faster than those relying on depth-first search behaviour. One reason could be that a MultiQueue operation is slower than with the stack-based concurrent queues, since the time complexity of a priority-queue operation is $O(\log(n))$, whereas it is $O(1)$ for stacks, where n is the number of elements in the data structure. This does not explain the result, but generally priority-queue operations need more work than stack operations, which could offset the more efficient searching strategy. Another plausible reason could be that the priorities do not play a big role in the search. This could happen if a lot of nodes have the same upper bounds, such that their priorities are equal and the search cannot be effectively guided anymore. Perhaps this could come from the approximate bounds computation, which could be tighter, or it could be for the most part unavoidable on these instances.

Something clear is that the MultiQueue seems to perform much better than Lindén. Recall that they both are concurrent priority queues, but the MultiQueue is relaxed while Lindén is not. Here we can see clear effects of relaxation. In general, the relaxed designs perform better than the non-relaxed designs, despite deviations from

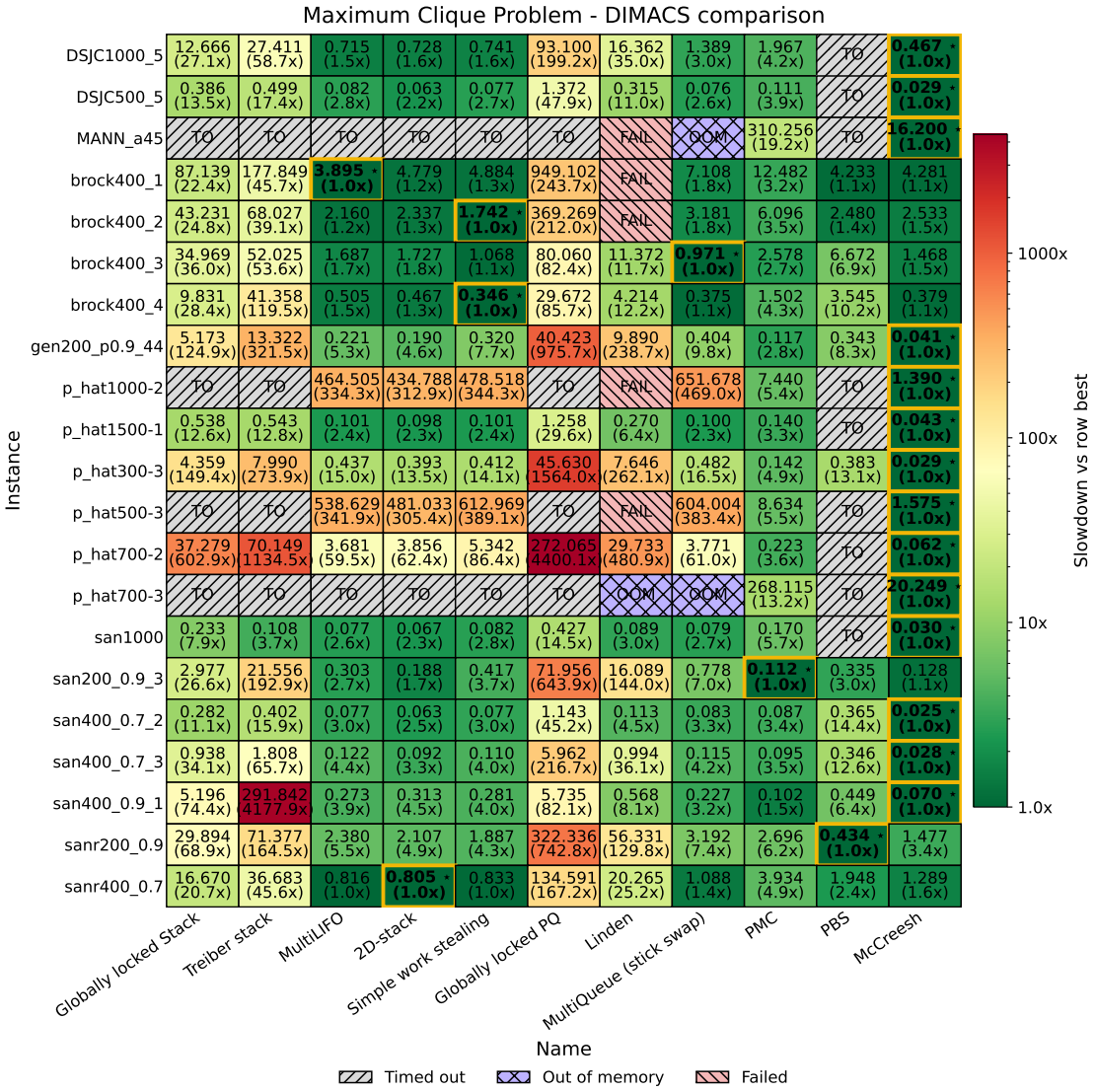

Figure 7.12: Maximum clique benchmark results for the comparison analysis on all filtered DIMACS instances. Execution time is presented for all solvers, as well as slowdowns compared to the fastest solver on the same graph instance. The fastest solver for each instance is highlighted with a golden border and a star.

strict ordering semantics.

Observe that some of the priority queue runs show up as out-of-memory (OOM), meaning they needed more memory than the system could provide. This is expected with priority queues, as they may require more and more memory as they store a growing search front. With DFS, only a memory footprint size proportional to a single path through the search tree is required, meaning it stays bounded and is not expected to run out of memory. In this experiment, this flaw with best-first designs did not matter, since the maximum solving time was capped at 30 minutes, and whenever there was an OOM, most other solvers timed out anyways. However, if the timeout was removed, the DFS-based solvers would continue to eventually find the solution, whereas the best-first-based solvers would still run out of memory.

Note that PMC and McCreesh stand out as the only solvers that could solve all instances in time. Since PMC is an open-source library with the purpose of providing a reliable maximum clique solver, this shows that the authors of PMC succeeded in a sense. What is very interesting however, is how much better McCreesh performs compared to PMC (most often 3x-6x faster). The source code for McCreesh is open source [83], but compiles to a standalone program rather than to be used as an extension to other code. Thus, one could contribute to the open-source community by building an easy-to-use importable library for solving the maximum clique problem just like PMC, but with the performance of McCreesh.

7.4 Knapsack Problem

This section presents a scalability experiment done on the knapsack problem. This experiment demonstrates the effectiveness of the generic framework, reinforcing that it can be used to solve and benchmark any BnB problem. Figure 7.13 shows the execution time and speedups different solvers instantiated from the framework. Each benchmark is run five times and the average execution time is reported. This time, no existing solvers are compared against. The input used is an arbitrary input found in the KPLIB [69] dataset by the path `kplib/04AlmostStronglyCorrelated/n00500/R01000/s069.kp`. The same batching and stickiness configurations as used in the maximum clique scalability experiments are used in this experiment as well.

Figure 7.13 shows that for this problem, the DFS behaviour is even more unpredictable than in the maximum clique problem. Some of the 2D-stack and MultiLIFO runs didn't finish within the 30 minutes required to avoid time out. The MultiQueue displays much more stable and predictable performance, once again showing why best-first searching can be beneficial in BnB. The speedups do not quite scale linearly with the number of threads. This is expected since the bounds approximation is basic and not very tight. A more sophisticated bounding component could help reach significantly better speedups.

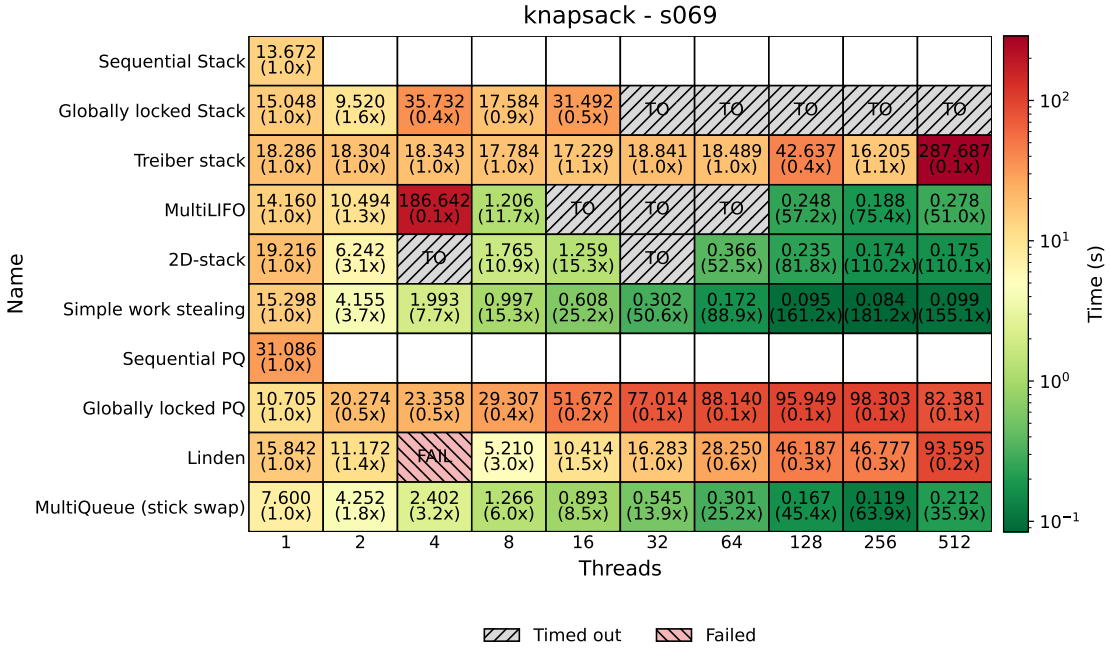

Figure 7.13: Results for the scalability experiment on the knapsack problem for the input named s069.

7.5 Multidimensional Knapsack Problem

Just like the knapsack problem results, this section shows the results for the MDKP problem. Figure 7.14 shows a similar plot as for the knapsack problem, where a heatmap displays execution times and speedup across thread counts. However, since the input data contains floating point values, and Lindén does not support non-integer key types, the Lindén solver is left out in this experiment. The same configuration is used as for the knapsack experiment, and the input instance used is called `SENTO2.DAT` within the `mknap2.txt` file in the previously mentioned OR-Library [71]. This instance was selected because it is the first instance to actually take the solvers any significant time to solve.

Again, like with the knapsack experiment, Figure 7.14 shows that we do not quite reach linear speedups, and the reason is the same. However, this time both depth-first and best-first show stability in terms of execution time, and neither seems to perform much better than the other. All the solvers with relaxed underlying queues reach roughly 200ms with 512 threads, except the MultiLIFO which takes 259ms. None of the other solvers come close to this performance, which again strengthens the evidence of using relaxed concurrent data structures within BnB. Regarding search order, there is no clear overall winner between depth-first and best-first. This can be seen as a more general observation within BnB. The drawbacks and benefits of both search strategies have to be taken into account, and even then it's hard to predict which one will perform the best.

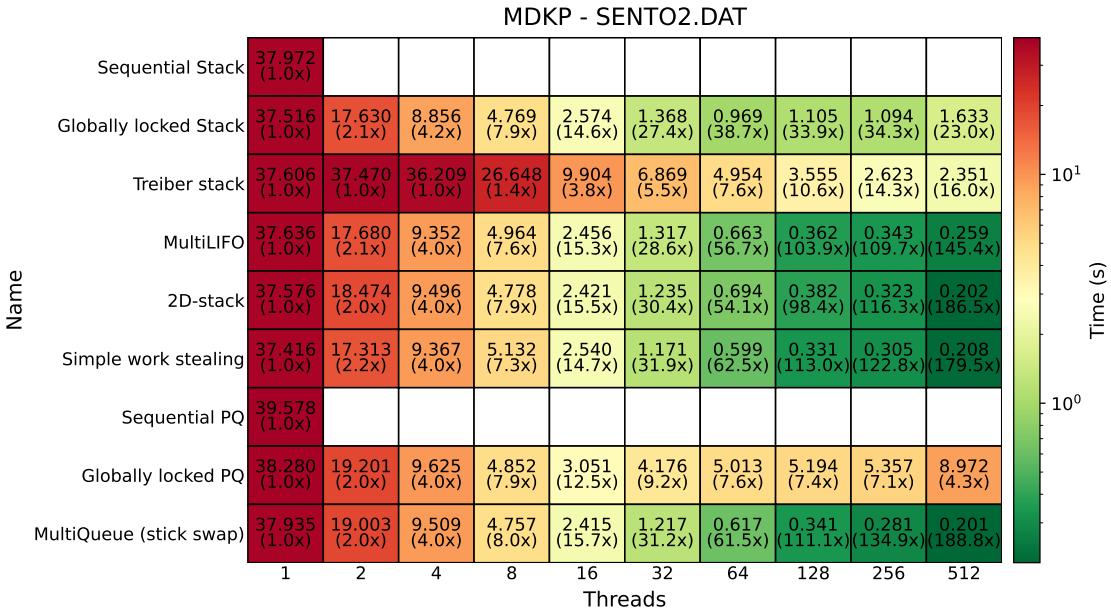

Figure 7.14: Results for the scalability experiment on the MDKP problem for the input named SENTO2.DAT.

8

Conclusions

This thesis investigated the applicability of relaxed concurrent queues in branch-and-bound (BnB) algorithms, with particular focus on the MultiQueue and related relaxed concurrent stack and priority-queue designs. The work combined a literature study with the design and evaluation of a generic modular framework for parallel BnB solving. Experimental evaluations were conducted primarily on the Maximum Clique Problem, alongside demonstrations on the knapsack problem and the multidimensional knapsack problem.

The results indicate that BnB algorithms form a suitable domain for relaxed concurrent queues. The literature study identified BnB as a class of algorithms where strict exploration order is not required for correctness, making relaxed ordering semantics applicable. The experimental results further demonstrated that both relaxed concurrent stacks and relaxed concurrent priority queues can be integrated into BnB solvers while still producing correct solutions and competitive performance. In particular, the relaxed designs showed strong scalability at high thread counts compared to strictly synchronised alternatives.

The experiments demonstrated that algorithmic and data-structure design choices strongly influence performance in parallel BnB workloads. Search strategy, batching, queue configuration, and the degree of relaxation all affected scalability and execution time. The batching experiments showed that reducing contention through local work processing could substantially improve scalability, while the stickiness experiments demonstrated that queue-specific configuration parameters can significantly affect performance. Overall, the relaxed concurrent designs consistently scaled better than their strict counterparts under high contention. The experiments also highlighted trade-offs between best-first and depth-first exploration strategies, where best-first approaches often reduced explored work, while DFS-based approaches exhibited lower synchronisation overhead and stronger scalability. Interestingly, relaxed concurrent stack designs such as the 2D-stack and MultiLIFO often achieved stronger scalability than the relaxed priority-queue-based approaches, demonstrating that relaxed DFS-oriented exploration can be highly effective in parallel BnB workloads. In fact, we haven't found any other use cases for relaxed stacks specifically. Finding and successfully applying them within BnB is a contribution in itself, motivating further interest in applying them to similar problems, and perhaps designing more of them.

The results show that data-structure-level parallelism can compete with state-of-

the-art algorithm-level parallelisation strategies in several cases, particularly at very high thread counts. Although the proposed solvers did not consistently outperform specialised state-of-the-art implementations such as McCreesh or PBS, some framework-instantiated solvers achieved competitive scalability and occasionally outperformed existing approaches on individual benchmark instances. The experiments further showed clear benefits of relaxation in practice, with the MultiQueue significantly outperforming the strict Lindén priority queue at high thread counts despite similar single-threaded performance. The experiments also showed that some state-of-the-art solvers scale poorly at very high thread counts despite strong sequential performance, indicating that there remains room for further research into scalable parallel BnB solving. Furthermore, the proposed implementations were not built upon the strongest known sequential maximum clique solvers, suggesting that combining the presented parallelisation strategies with more sophisticated sequential baselines could potentially yield substantially stronger overall performance.

An additional contribution of this thesis is the development of a modular generic framework for parallel BnB algorithms. The framework was demonstrated on multiple BnB problems and greatly simplified the process of developing high-performing BnB solvers with a common infrastructure. This design shift—putting the parallelisation logic in the data structure layer—shifts the burden away from the programmer, which can more easily design parallel applications. Additionally, the work contributed modifications to the existing open-source MultiFIFO implementation, transforming it into a relaxed concurrent stack variant proposed as the MultiLIFO, which showed strong scalability and competitive performance throughout the experiments.

This paragraph outlines possible future work, to continue where this thesis left off. Firstly, future work can focus on finding other algorithmic domains that benefit from relaxed concurrent data structures. The candidate domain study can serve as a starting point for those interested in investigating other possible domains. Secondly, future work could extend evaluation with more problem definitions to cover a broader range of branch-and-bound applications. Although they gave useful insight, our evaluations were only done on three BnB problems. A broader study can investigate whether the findings of this study is a general trend within BnB. It could also enable deeper understanding of relaxation behaviour in general. Lastly, improving the bounding implementation for the maximum clique problem to match state-of-the-art methods would enable a fair comparison of whether our parallelisation strategy outperforms existing approaches. Our problem definition for the maximum clique problem isn't very sophisticated, and the comparison between the single-threaded performance of our solvers with competitor solvers showed evidence that it can be improved. Thus, we believe that a more sophisticated problem definition for the maximum clique problem could compete and possibly outperform state-of-the-art parallel solvers.

Bibliography

[1] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, 2005.

[2] K. e. a. Asanovi, "The landscape of parallel computing research: A view from berkeley," UC Berkeley, Tech. Rep., 2009.

[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th. Morgan Kaufmann, 2019.

[4] A. Lenharth, D. Nguyen, and K. Pingali, "Priority queues are not good concurrent priority schedulers," in *Euro-Par 2015: Parallel Processing*, J. L. Träff, S. Hunold, and F. Versaci, Eds., ser. Lecture Notes in Computer Science, vol. 9233, Springer, 2015, pp. 209–221. DOI: 10.1007/978-3-662-48096-0_17.

[5] H. Rihani, P. Sanders, and R. Dementiev, "Brief announcement: Multiqueues: Simple relaxed concurrent priority queues," in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2015)*, ACM, 2015, pp. 80–82. DOI: 10.1145/2755573.2755616.

[6] N. Shavit, "Data structures in the multicore age," *Communications of the ACM*, vol. 54, no. 3, pp. 76–84, 2011. DOI: 10.1145/1897852.1897873.

[7] M. Williams, P. Sanders, and R. Dementiev, "Engineering multiqueues: Fast relaxed concurrent priority queues," in *29th Annual European Symposium on Algorithms (ESA 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 204, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 81:1–81:17. DOI: 10.4230/LIPIcs.ESA.2021.81.

[8] A. H. Land and A. G. Doig, "An automatic method of solving discrete programming problems," *Econometrica*, vol. 28, no. 3, pp. 497–520, 1960. DOI: 10.2307/1910129.

[9] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, Revised 1st. Morgan Kaufmann, 2012, ISBN: 978-0123973375.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, Jul. 31, 2009, ISBN: 9780262033848.

[11] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959. DOI: 10.1007/BF01386390.

[12] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM*, vol. 34, no. 3, pp. 596–615, 1987.

[13] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990. DOI: 10.1145/78969.78972.

[14] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev, "Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated," in *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*, 2011, pp. 487–498. DOI: 10.1145/1926385.1926442.

[15] F. Ellen, D. Hendler, and N. Shavit, "On the inherent sequentiality of concurrent objects," *SIAM Journal on Computing*, vol. 41, no. 3, pp. 519–536, 2012. DOI: 10.1137/08072646X.

[16] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit, "The spraylist: A scalable relaxed priority queue," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2015. DOI: 10.1145/2688500.2688523.

[17] K. Sagonas and K. Winblad, "The contention avoiding concurrent priority queue," in *Languages and Compilers for Parallel Computing*, 2017. DOI: 10.1007/978-3-319-52709-3_23.

[18] M. Wimmer, J. Gruber, J. L. Träff, and P. Tsigas, "The lock-free k-lsm relaxed priority queue," *ACM SIGPLAN Notices*, vol. 50, no. 8, 2015. DOI: 10.1145/2858788.2688547.

[19] A. Postnikova, N. Koval, G. Nadiradze, and D.-A. Alistarh, "Multi-queues can be state-of-the-art priority schedulers," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2022)*, 2022, pp. 353–367. DOI: 10.1145/3503221.3508432.

[20] G. Zhang, G. Posluns, and M. C. Jeffrey, "Multi bucket queues: Efficient concurrent priority scheduling," in *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '24)*, Nantes, France: ACM, 2024, pp. 113–124. DOI: 10.1145/3626183.3659962. [Online]. Available: https://doi.org/10.1145/3626183.3659962.

[21] D. Alistarh, J. Kopinsky, J. Z. Li, and G. Nadiradze, "The power of choice in priority scheduling," in *Proceedings of the 36th ACM Symposium on Principles of Distributed Computing (PODC 2017)*, 2017, pp. 461–470. DOI: 10.1145/3087801.3087810.

[22] S. Walzer and M. Williams, "A simple yet exact analysis of the multiqueue," in *SPAA '21*, 2021, pp. 145–156. DOI: 10.1145/3409964.3461807.

[23] M. Williams and P. Sanders, "The multiqueue: A simple and fast relaxed concurrent priority queue," *ACM Transactions on Parallel Computing*, 2024. DOI: 10.1145/3771738. [Online]. Available: https://dl.acm.org/doi/10.1145/3771738.

[24] H. Rihani, P. Sanders, and R. Dementiev, *Multiqueues: Simpler, faster, and better relaxed concurrent priority queues*, 2014. DOI: 10.48550/arXiv.1411.1209. arXiv: 1411.1209 [cs.DS].

[25] S. Koch, P. Sanders, and M. Williams, "Blockfifo & multififo: Scalable relaxed queues," in *Proceedings of the SIAM Symposium on Algorithm Engineering*

and Experiments (ALENEX), Society for Industrial and Applied Mathematics (SIAM), 2026, pp. 31–43. DOI: 10.1137/1.9781611978957.3.

[26] A. Rukundo, A. Atalar, and P. Tsigas, "Brief announcement: 2d-stack - A scalable lock-free stack design that continuously relaxes semantics for better performance," in *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, C. Newport and I. Keidar, Eds., ACM, 2018, pp. 407–409. [Online]. Available: https://dl.acm.org/citation.cfm?id=3212794.

[27] A. Rukundo, A. Atalar, and P. Tsigas, "Monotonically relaxing concurrent data-structure semantics for increasing performance: An efficient 2d design framework," in *33rd International Symposium on Distributed Computing, DISC 2019, Budapest, Hungary, October 14-18, 2019*, J. Suomela, Ed., ser. LIPIcs, vol. 146, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 31:1–31:15. DOI: 10.4230/LIPICS.DISC.2019.31. [Online]. Available: https://doi.org/10.4230/LIPIcs.DISC.2019.31.

[28] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968. DOI: 10.1109/TSSC.1968.300136.

[29] M. D'Antonio, K. von Geijer, T. S. Mai, P. Tsigas, and H. Vandierendonck, "Relax and don't stop: Graph-aware asynchronous sssp," in *Proceedings of the 1st FastCode Programming Challenge (FCPC 2025)*, ACM, 2025, pp. 43–47. DOI: 10.1145/3711708.3723446.

[30] M. D'Antonio, T. S. Mai, P. Tsigas, and H. Vandierendonck, "Wasp: Efficient asynchronous single-source shortest path on multicore systems via work stealing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25)*, New York, NY, USA: ACM, 2025, pp. 2109–2125, ISBN: 979-8-4007-1466-5. DOI: 10.1145/3712285.3759872.

[31] X. Dong, A. Li, Y. Gu, and Y. Sun, "Parallel point-to-point shortest paths and batch queries," in *Proceedings of the 37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '25)*, 2025. DOI: 10.1145/3694906.3743311.

[32] E. L. Lawler and D. E. Wood, "Branch-and-bound methods: A survey," *Operations Research*, vol. 14, no. 4, pp. 699–719, 1966. DOI: 10.1287/opre.14.4.699.

[33] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*. New York: John Wiley & Sons, 1988, ISBN: 978-0-471-82819-8. DOI: 10.1002/9781118627372.

[34] P. R. J. Östergård, "A fast algorithm for the maximum clique problem," *Discrete Applied Mathematics*, vol. 120, no. 1–3, pp. 197–207, 2002. DOI: 10.1016/S0166-218X(01)00290-6.

[35] E. Tomita and T. Kameda, "An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments," *Journal of Global Optimization*, vol. 37, no. 1, pp. 95–111, 2007. DOI: 10.1007/s10898-006-9039-7.

[36] C. McCreesh and P. Prosser, "Multi-threading a state-of-the-art maximum clique algorithm," *Algorithms*, vol. 6, no. 4, pp. 618–635, 2013. DOI: 10.3390/a6040618. [Online]. Available: https://www.mdpi.com/1999-4893/6/4/618.

[37] H. Jiang, K. Bai, H.-J. Liu, C.-M. Li, F. Manyà, and Z.-H. Fu, "Parallel bounded search for the maximum clique problem," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 7, pp. 1532–1546, 2018. DOI: 10.1109/TPDS.2018.2794319.

[38] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, 1990, ISBN: 9780471924203.

[39] R. Carraghan and P. M. Pardalos, "An exact algorithm for the maximum clique problem," *Operations Research Letters*, vol. 9, no. 6, pp. 375–382, 1990. DOI: 10.1016/0167-6377(90)90057-C.

[40] IRIDIA, CoDE, Université Libre de Bruxelles, *The maximum clique problem – dimacs benchmark set*, https://iridia.ulb.ac.be/~fmascia/maximum_clique/, Selected instances from the Second DIMACS Implementation Challenge. Accessed 2026-03-27, 2026.

[41] U. Meyer and P. Sanders, "Δ-stepping: A parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114–152, 2003. DOI: 10.1016/S0196-6774(03)00076-2. [Online]. Available: https://doi.org/10.1016/S0196-6774(03)00076-2.

[42] A. Singer, H. Yan, G. Zhang, M. C. Jeffrey, M. Stojilovi, and V. Betz, "Multiqueue-based fpga routing: Relaxed a* priority ordering for improved parallelism," in *Proceedings of the 23rd International Conference on Field-Programmable Technology (FPT 2024)*, IEEE, 2024, pp. 1–9.

[43] Linux Kernel Project, *Cfs scheduler*, Kernel documentation, 2026. Accessed: Feb. 8, 2026. [Online]. Available: https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html.

[44] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*. John Wiley & Sons, 2000, ISBN: 9780471183839.

[45] K. M. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 5, pp. 440–452, 1979.

[46] R. E. Bryant, "Simulation of packet communication architecture computer systems," Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, MA, USA, Tech. Rep. MIT-LCS-TR-188, 1977.

[47] D. R. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, pp. 404–425, 1985. DOI: 10.1145/3916.3988.

[48] C. D. Carothers and K. S. Perumalla, "On deciding between conservative and optimistic approaches on massively parallel platforms," in *Proceedings of the 2010 Winter Simulation Conference*, Winter Simulation Conference, 2010, pp. 678–687. DOI: 10.1109/WSC.2010.5679016.

[49] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952. DOI: 10.1109/JRPROC.1952.273898.

[50] P. Klein and P. Sanders, "Fast parallel construction of huffman codes," in *Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and*

Architectures, ser. SPAA '03, ACM, 2003, pp. 195–204. DOI: 10.1145/777412.777446.

[51] NVIDIA Corporation, *Cuda c++ programming guide*, https://docs.nvidia.com/cuda/, Accessed: 2026-02-08, 2026.

[52] K. Wang, D. Fussell, and C. Lin, "A fast work-efficient sssp algorithm for gpus," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '21)*, New York, NY, USA: Association for Computing Machinery, 2021, pp. 133–146. DOI: 10.1145/3437801.3441605.

[53] X. Xu, Y. Tang, X. Cui, and W. Zhang, "Bgpq: A heap-based priority queue design for gpus," in *Proceedings of the 2022 IEEE International Parallel and Distributed Processing Symposium*, IEEE, 2022, pp. 808–818. DOI: 10.1109/IPDPS53621.2022.00083.

[54] P. M. Pardalos and D.-Z. Du, Eds., *Handbook of Combinatorial Optimization* (Springer Reference), 2nd ed. Springer, 2013. DOI: 10.1007/978-1-4419-7997-1.

[55] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*. New York: Wiley-Interscience, 1988, ISBN: 9780471359432.

[56] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Boston, MA: Addison-Wesley, 2003.

[57] S. M. Johnson, "Optimal two- and three-stage production schedules with setup times included," *Naval Research Logistics Quarterly*, vol. 1, no. 1, pp. 61–68, 1954. DOI: 10.1002/nav.3800010110.

[58] R. M. Karp and Y. Zhang, "Randomized parallel algorithms for backtrack search and branch-and-bound computation," *Journal of the ACM*, vol. 40, no. 3, Jul. 1993. DOI: 10.1145/174130.174145.

[59] P. Sanders, "Fast priority queues for parallel branch-and-bound," in *Parallel Algorithms for Irregularly Structured Problems*, Springer, 1995. DOI: 10.1007/3-540-60321-2_30.

[60] P. Sanders, "Randomized priority queues for fast parallel access," *Journal of Parallel and Distributed Computing*, vol. 49, no. 1, Feb. 1998. DOI: 10.1006/jpdc.1998.1429.

[61] D. Alistarh, G. Nadiradze, and N. Koval, "Efficiency guarantees for parallel incremental algorithms under relaxed schedulers," in *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, ACM, 2019, pp. 377–386. DOI: 10.1145/3323165.3323201.

[62] P. San Segundo, D. Rodríguez-Losada, and A. Jiménez, "An exact bit-parallel algorithm for the maximum clique problem," *Computers & Operations Research*, vol. 38, no. 2, pp. 571–581, 2011. DOI: 10.1016/j.cor.2010.07.019.

[63] P. San Segundo, F. Matía, D. Rodríguez-Losada, and M. Hernando, "An improved bit parallel exact maximum clique algorithm," *Optimization Letters*, vol. 7, no. 3, pp. 467–479, 2013. DOI: 10.1007/s11590-011-0431-y.

[64] C.-M. Li, H. Jiang, and F. Manyà, "On minimization of the number of branches in branch-and-bound algorithms for the maximum clique problem," *Computers & Operations Research*, vol. 84, pp. 1–15, 2017. DOI: 10.1016/j.cor.2017.02.017.

[65] P. San Segundo, F. Furini, D. Álvarez, and P. M. Pardalos, "CliSAT: A new exact algorithm for hard maximum clique problems," *European Journal of Operational Research*, vol. 307, no. 3, pp. 1008–1025, 2023. DOI: 10.1016/j.ejor.2022.10.028.

[66] P. San Segundo, *CliSAT: An exact algorithm for the maximum clique problem*, https://github.com/psanse/CliSAT, GitHub repository, accessed 2026-06-11.

[67] M. Williams, *Experimental evaluation of the multiqueue*, GitHub repository, 2021. Accessed: Feb. 8, 2026. [Online]. Available: https://github.com/marvinwilliams/multiqueue_experiments.

[68] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001, ISBN: 9780201704310.

[69] likr, *Kplib: Test instances for knapsack problems*, https://github.com/likr/kplib, Accessed 2026-03-27, 2026.

[70] H. Kellerer, U. Pferschy, and D. Pisinger, "Exact solution of the knapsack problem," in *Knapsack Problems*, Berlin, Heidelberg: Springer, 2004, pp. 117–160. DOI: 10.1007/978-3-540-24777-7_5.

[71] J. E. Beasley, *Or-library: Multidimensional knapsack problem*, https://people.brunel.ac.uk/~mastjjb/jeb/orlib/mknapinfo.html, Accessed 2026-03-27, 2026.

[72] D. S. Johnson and M. A. Trick, Eds., *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge* (DIMACS Series in Discrete Mathematics and Theoretical Computer Science). Providence, RI: American Mathematical Society, 1996, vol. 26, ISBN: 9780821806609.

[73] R. K. Treiber, "Systems programming: Coping with parallelism," IBM Thomas J. Watson Research Center, Tech. Rep. RJ 5118, 1986.

[74] M. Khizhinsky, *Libcds: Concurrent data structures library*, https://github.com/khizmax/libcds, GitHub repository, accessed 2026-05-27, 2026.

[75] H. Sundell and P. Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," in *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings*, IEEE Computer Society, 2003, p. 84. DOI: 10.1109/IPDPS.2003.1213189. [Online]. Available: https://doi.org/10.1109/IPDPS.2003.1213189.

[76] H. Sundell and P. Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," *Journal of Parallel and Distributed Computing*, vol. 65, no. 5, pp. 609–627, 2005, ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2004.12.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731504002333.

[77] J. Lindén and B. Jonsson, "A skiplist-based concurrent priority queue with minimal memory contention," *Journal of Parallel and Distributed Computing*, vol. 73, no. 12, pp. 1518–1528, 2013. DOI: 10.1016/j.jpdc.2013.08.001.

[78] J. Lindén, *Pr: A lock-free priority queue implementation*, https://github.com/jonatanlinden/PR, GitHub repository, accessed 2026-05-27, 2026.

[79] S. Saalvage, *Block_based_queue: A novel approach to relaxing a fifo queue*, https://github.com/Saalvage/block_based_queue, GitHub repository, accessed 2026-05-27, 2025.

[80] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '05, New York, NY, USA: Association for Computing Machinery, 2005, pp. 21–28. DOI: 10.1145/1073970.1073974.

[81] C. Williams, *Concurrentdeque: Work-stealing deque implementations in c++*, https://github.com/ConorWilliams/ConcurrentDeque, GitHub repository, accessed 2026-05-27, 2026.

[82] R. Rossi, *Pmc: Parallel maximum clique solver*, https://github.com/ryanrossi/pmc, GitHub repository, accessed 2026-05-27, 2026.

[83] C. McCreesh, *Multi-threaded maximum clique solver*, https://github.com/ciaranm/multithreadedmaximumclique, GitHub repository, accessed 2026-05-29, 2013.

[84] L. Sandh, *Source code for "scalable parallel branch-and-bound with relaxed concurrent queues"*, https://github.com/ludvig-sandh/multiqueue_experiments, Accessed: 2026-06-03, 2026.