YewPar: Skeletons for Exact Combinatorial Search

Blair Archibald
University of Glasgow
Blair.Archibald@glasgow.ac.uk

Patrick Maier
University of Stirling
Patrick.Maier@stir.ac.uk

Robert Stewart
Heriot-Watt University
R.Stewart@hw.ac.uk

Phil Trinder
University of Glasgow
Phil.Trinder@glasgow.ac.uk

Abstract

Combinatorial search is central to many applications, yet the huge irregular search trees and the need to respect search heuristics make it hard to parallelise. We aim to improve the *reuse* of intricate parallel search implementations by providing the first general purpose scalable parallel framework for exact combinatorial search, YewPar.

We make the following contributions. (1) We present a novel formal model of parallel backtracking search, covering enumeration, decision, and optimisation search. (2) We introduce Lazy Node Generators as a uniform API for search tree generation. (3) We present the design and implementation of 12 widely applicable algorithmic skeletons for tree search on shared and distributed memory architectures. (4) Uniquely in the field we demonstrate how a wide range of parallel search applications can easily be constructed by composing Lazy Node Generators and the search skeletons. (5) We report a systematic performance analysis of all 12 YewPar skeletons on standard instances of 7 search applications, investigating skeleton overheads and scalability up to 255 workers on 17 distributed locations.

CCS Concepts • **Computing methodologies → Combinatorial algorithms**; **Parallel programming languages**;

Keywords Combinatorial Search, Algorithmic Skeletons, Distributed Memory Parallelism, HPX

1 Introduction

Exact combinatorial search is essential to a wide range of applications including constraint programming, graph matching, and computational algebra. Combinatorial problems are solved by systematically exploring a search space, and doing so is computationally hard both in theory and in practice, encouraging the use of *approximate* algorithms that quickly provide answers yet with no guarantee of optimality. Alternatively, *exact* search explores the entire search space

and delivers provably optimal answers. Conceptually exact combinatorial search proceeds by generating and traversing a (huge) tree representing alternative options. Combining parallelism, on-demand tree generation, search heuristics, and pruning can reduce the execution time of exact search.

There are three main types of search: *enumeration*, which searches for all solutions matching some property, e.g. all maximal cliques in a graph; *decision*, which looks for a specific solution, e.g. a clique of size k; and *optimisation*, which looks for a solution that minimises/maximises an objective function, e.g. finding a maximum clique.

Parallelising exact combinatorial search is extremely challenging due to huge and highly irregular search trees, the need to preserve search order heuristics that bias search towards likely solutions, and pruning heuristics that dynamically alter the workload to avoid unnecessary search. State-of-the-art parallel searches (1) are single purpose, i.e. for a specific search application, e.g. Embarrassingly Parallel Search [24] supports constraint programming only; (2) use hand crafted parallelism, e.g. parallel MaxClique [28], with almost no reuse of sophisticated parallel coordination between search applications; and (3) target a single scale of parallelism, often shared memory multicores. Hence typically an application is parallelised just once, on a specific architecture, in a heroic effort.

In contrast this paper presents YewPar, the *first scalable parallel framework for exact combinatorial search*. YewPar is designed to allow non-expert users to benefit from parallelism; to reuse parallel search patterns encoded as *algorithmic skeletons*; and to run on multiple parallel architectures. The paper makes the following contributions.

A new formal model of parallel backtracking search uniformly characterising enumeration, decision and optimisation searches as a fold of the search tree into a monoid (Section 3). The formal model combines an encoding of search trees and a nondeterministic small-step semantics. We prove that the semantics correctly models parallel search, i.e. reductions terminate and yield correct/optimal results, regardless of the interleaving of parallel reductions. Correctness is far from obvious because pruning reductions may alter the shape of the search tree nondeterministically. We use the model to derive a programming interface for the skeletons

(Section 4.1) and to succinctly describe the work generation behaviours of the skeletons (Section 4.2).

A widely applicable API for search tree generation (Section 4.1). We introduce *Lazy Node Generators* as a uniform abstraction for application developers to specify how the search trees for a specific application are created. The generators are based on a formalisation of backtracking tree traversal and implicitly encode application-specific search order heuristics. Search tree nodes are constructed lazily to avoid materialising the entire (huge) tree, such that pruning removes redundant search by eliminating subtrees before they are materialised.

12 widely applicable algorithmic skeletons for backtracking search (Section 4). The skeletons use sophisticated parallel *search coordinations* inspired by the literature, currently: *Sequential*, *Depth-Bounded*, *Stack-Stealing* and *Budget* (Section 4.2). The 12 skeletons are parametric combinations of the 3 search types, and the 4 search coordinations. To scale on both shared and distributed memory architectures, YewPar builds on the HPX C++ distributed execution framework [22], and features custom work-stealing schedulers that handle search irregularity and carefully manage search order heuristics and knowledge exchange.

A demonstration of the easy construction of parallel search applications using *Lazy Node Generators* and *search skeletons* (Sections 4.4, 5.1 and A.3). We show how to parallelise exact combinatorial search applications by composing a Lazy Node Generator (to create the search tree), with an appropriate skeleton to traverse the tree (Fig. 3). The generality of YewPar is shown with 7 search applications covering the 3 search types. Prior implementations like [1, 34, 40] support just one search type, and most provide only one search coordination, the exception being Muesli [33] which supports two for optimisation search. Most parallel implementations demonstrate performance for a single search application, a minority do two applications, and very few [16, 23] do more.

A systematic performance analysis of the skeletons and YewPar (Section 5). The evaluation is entirely novel and measures the performance of the 12 skeletons across 7 search applications. The applications are evaluated on a set of approximately 30 standard challenge search instances, e.g. from the DIMACS benchmark suite [21]. To evaluate the performance overheads of the general purpose skeletons we compare YewPar with a state-of-the-art hand-coded C++ implementation on 18 instances. We demonstrate YewPar execution on a multicore and a Beowulf cluster, showing good scaling and efficiency up to 255 workers on 17 localities (physical machines). We show that YewPar allows users to easily explore alternate parallelisations, and compare the performance of each search application with every skeleton.

This paper focuses on the YewPar programming model and formalisation. For in-depth performance analyses of the implementation, including workload management, see [3, 5]. YewPar is available as an evaluated artifact (Appendix A).

2 Background

To illustrate combinatorial search, consider the problem of finding cliques within a graph such as the tiny graph shown in Fig. 1. A clique is a set of vertices C such that all vertices in C are pairwise adjacent. Example cliques in Fig. 1 are $\{a\}$ and $\{a, b, c\}$, but $\{a, b, c, g\}$ is not a clique as there is no edge between c and g.

The search space is huge, the powerset of the set of vertices, but we can minimise the space to be explored by generating only valid cliques. So given a clique, extend it by adding any vertex that is adjacent to all elements of the clique. This leads naturally to backtracking search: extend the clique until no more additions are possible; thereafter remove the most recently added element and try a different vertex.

We can view the search space as a tree where each node corresponds to a clique, e.g. level 1 of the tree for the graph in Fig. 1 represents the cliques $\{c\}$, $\{f\}$, $\{g\}$ etc, and the direct descendants of clique $\{c\}$ are $\{c, a\}$, $\{c, b\}$ and $\{c, e\}$.

It is not always necessary to traverse the entire search tree. For example, to check whether the graph has a 3-clique, a *decision* search, we only need to generate nodes $\{c\}$, $\{c, a\}$ and $\{c, a, b\}$. Likewise we can use bounding to *prune* the search space, i.e. we do not need to explore a branch if we can prove that a solution cannot exist in that branch.

The *search order* determines the sequence in which nodes in the search tree are explored, and is extremely important as finding a solution quickly allows early termination in decision searches and improved pruning for optimisation searches. Like most search frameworks YewPar primarily uses depth-first search as it requires less memory than other search strategies such as breadth-first.

Search spaces are often huge and grow exponentially, e.g. 378×10^{12} nodes at depth 67 with a growth rate of 1.62 between successive depths in [17]. In practise YewPar materialises only small fragments of the tree during search, saving both memory and runtime. Search problems are hard in both theory, usually $\mathcal{NP}$-hard [38], and in practice, with instances often taking hours, or longer, to solve. They may be solved using *approximate methods* like simulated annealing or ant colony optimisation [9] that quickly provide non-exact answers. In contrast *exact methods* explore the entire search space and provide *proofs* of optimality or infeasibility. YewPar aims to provide the benefits of exact search while using parallelism to reduce runtime.

2.1 Parallel Combinatorial Search

Parallelising search solves existing instances faster and allows larger instances to be practically solved, e.g. reducing runtimes from hours to minutes. Parallel tree searches can be classified [18] as follows: (1) Parallel Node Processing
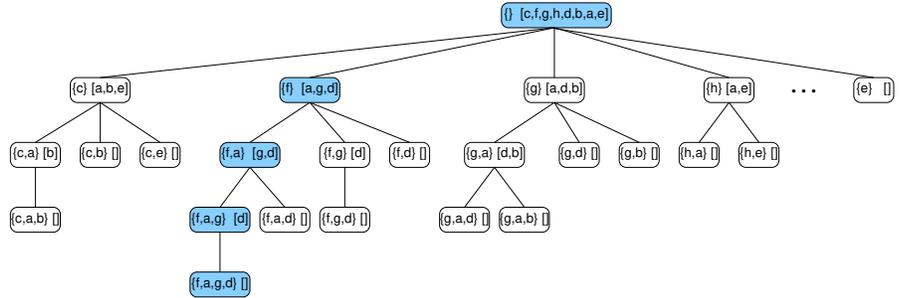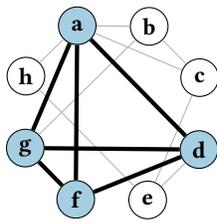
Figure 1. Maximum clique instance. Input graph with clique $\{a, d, f, g\}$ to the left and corresponding search tree to the right. Each tree node displays the current clique and a list of candidate vertices (in heuristic order) to extend that clique.

that parallelises the branching/bounding, e.g. computing the bounds for a Flowshop problem on a GPU [20]. (2) Space-Splitting where parallel workers *speculatively* explore subtrees of the search tree. While the subtrees are explored independently, knowledge such as improved bounds is often shared between workers to improve performance. (3) Portfolio approaches run competing searches in parallel, often with different heuristics or bounding methods. Hybrid approaches combining these approaches are also used.

Both parallel node processing and portfolio approaches require domain-specific knowledge, e.g. to vary bounding functions. YewPar uses space-splitting as it is independent of specific search domains, and hence suitable for general-purpose search skeletons.

While space-splitting maps naturally to task parallelism, it poses some very significant challenges. Search trees are highly irregular and the time required to search a particular subtree is both unpredictable and highly variable [31]. Careful choice of an initial work distribution, or methods to dynamically generate more work, are essential to keep workers busy. Knowledge transfer further complicates matters. (1) New knowledge dynamically changes the shape of the search tree: tasks that were predicted to be long running (often those near the root of the tree) may become trivial, and many tasks may be invalidated simultaneously, requiring work redistribution. (2) Gaining new knowledge early on greatly reduces search time by reducing the search space. Heuristics guide search to promising areas and essentially impose an ordering on tasks that should be preserved as far as possible. (3) Knowledge is shared globally which can be expensive on large (distributed memory) systems, although [5] shows that in many important searches there are few global knowledge updates.

Performance Anomalies. Search algorithms rely on ordering heuristics to find *useful* nodes as early as possible, e.g. the target node in a decision problem, or one with a strong bound for a branch and bound optimisation problem. To take advantage of these heuristics, search proceeds in a left-to-right order (at all depths of the tree). Hence a sequential search visiting a node has full information about the search

tree, e.g. incumbents, from all nodes to-the-left of it, and the search is deterministic.

Parallel searches *speculatively* search subtrees without full information to-the-left, and may benefit from information flowing right-to-left. However the speculation means that the parallel search may perform significantly more work than an equivalent sequential search. For this reason, parallel search is notorious for performance anomalies [14]. *Detrimental Anomalies* occur when the runtime on w workers is more than for $w - 1$ workers. Here the additional work may outweigh the benefit of the additional compute resource, or the additional compute resource may disrupt the search heuristic. *Acceleration Anomalies* are superlinear speedups, typically due to knowledge flowing right-to-left, reducing overall work by enabling more pruning than in a sequential search.

The presence of anomalies makes it difficult to reason about the parallel performance of search applications. YewPar aims to avoid detrimental anomalies while allowing acceleration anomalies; [4] reports a specialised search skeleton that carefully controls anomalies to provide replicable performance guarantees.

2.2 Existing Parallel Space-Splitting Searches

While reviews of parallel search often focus on a particular search domain e.g. [19], a multi-domain survey can be found in [3]. Space-splitting searches are classified by synchronicity, workpool layout, and load-balancing [3, 18, 37].

Static approaches e.g. EPS [35] or MapReduce [39] partially explore the search tree, usually breadth first, to generate a set of search tasks. They are simple, use minimal communication (often to/from a single leader node), and scale well on distributed architectures e.g. to 512 workers for EPS [35]. Many static searches rely on domain-specific heuristics to determine how many tasks to generate.

Dynamic approaches divide the search space during search, adapting to the search state to avoid starvation. Hence they may effectively parallelise searches that cannot be statically partitioned. Their implementations are more complex, and using techniques like work-stealing means that the searches

are no longer deterministic. Dynamic approaches are classified by their load-balancing strategy: (1) centralised strategies as in the BOB framework [29] are common; (2) decentralised strategies as in ZRAM [11] and [1] often use work-stealing; (3) periodic strategies redistribute work at intervals during the search as in mts [8]. YewPar also provides dynamic work generation, with general purpose search coordinations that use random distributed work-stealing and on-demand tree splitting, and coordinations that periodically repartition work based on a *backtracking budget* similar to mts [8].

2.3 Why a New Parallel Search Framework?

As implementing parallel tree search is so intricate, it would be a huge saving to reuse effective implementations. Unfortunately while general purpose task parallel frameworks like Cilk [10] or X10 [12] provide features like work-stealing, they use techniques inappropriate for search, e.g. standard deque-based work-stealing breaks heuristic search orders. Moreover a search-specific framework makes it easier to fulfil parallel search requirements, e.g. dynamically splitting the search tree, and sharing knowledge between search tasks.

A careful analysis of 31 parallel tree search implementations (Table 2.1 in [3]) reveals that most are application specific like [34, 40], and there are just two generic frameworks: Muesli [13] and MaLLBa [2]. Both are designed for branch and bound optimisation and, unlike YewPar, do not currently support decision or enumeration searches.

3 Formalising Parallel Tree Search

At an abstract level tree search can be viewed as an unfold/map/fold computation. The unfold generates the search tree, and the map and fold operate on a monoid that accumulates search information. That is, the map applies an *objective function* to turn each search tree node into a monoid value, and the fold reduces the resulting tree to a single monoid value, e.g. an optimal search result.

However, such an abstract formulation cannot readily describe the nondeterminism and pruning that are crucial for efficient optimisation and decision searches. Hence we construct a more concrete semantics as follows. Section 3.1 specifies the unfold, i.e. search tree generation and traversal. Section 3.2 specifies the monoids used to accumulate information in enumeration, optimisation and decision searches. Section 3.3 defines a new parallel operational semantics for multi-threaded tree traversal. This semantics generalises the semantics in [3] by classifying search types by their monoids, and by providing correctness proofs (Section 3.7).

3.1 Trees

To represent search trees, we formalise trees and tree traversals. Let X be a non-empty alphabet. X^* denotes the set of finite words over X, and $\leq$ denotes the prefix order on X^*.

A *tree* T is a non-empty finite prefix-closed subset of X^*. We call words $w \in T$ *nodes* of T. A node w is a *parent* of a node u, and u is a *child* of w, if there is $a \in X$ such that $u = wa$. Nodes u and v are *siblings* if they share the same parent. The *root* of T is the empty word ϵ, and the *depth* of a node w in T is the length of word w, denoted by $|w|$.

We assume that a tree T is *ordered*, i.e. equipped with a partial order $\prec_T$ that linearly orders siblings; we call $\prec_T$ a *sibling order*. We define the *traversal order* $\ll_T$ for T as a linear extension of the prefix order $\prec$ and the sibling order $\prec_T$ as follows.

$$u \ll_T v \;\; \text{iff} \;\; \begin{cases} u \prec v \text{ or} \\ \exists w, u', v' \in X^* \; \exists a, b \in X \text{ such that} \\ \quad u = wau' \text{ and } v = wbv' \text{ and } wa \prec_T wb \end{cases}$$

We omit the subscript if the tree T is obvious from the context. Traversing T in $\ll$ order is a depth-first traversal that visits the children of each node in sibling order.

A subset S of X^* is a *subtree* if S has a least element w.r.t. the prefix order, the *root* u, and is prefix-closed above the root, i.e. whenever $u \leq v \leq w$ and $w \in S$ then $v \in S$; we call S a subtree *rooted at* u. If a subtree S is a subset of a tree T then S inherits T's sibling order $\prec_T$ and traversal order $\ll_T$.

Let S be a subtree rooted at u, and let $v \in S$. We define $children(S, v)$ as the set of nodes in S that are children of v. We define $subtree(S, v) = \{w \mid w \in S \text{ and } v \leq w\}$ as the subtree of S rooted at v. If $v \neq u$ then $S \setminus subtree(S, v)$ is a subtree rooted at u. We define $succ(S, v) = \{w \mid w \in S \text{ and } v \ll w\}$ to be the set of nodes in S that follow v in traversal order. We define $next(S, v) = \min_{\ll} succ(S, v)$ as the node in S that immediately follows v in traversal order, writing $next(S, v) = \bot$ if no such node exists.

We write $lowest(S, v)$ for the subset of nodes in $succ(S, v)$ at minimum depth, that is closest to the root of S. We define $nextLowest(S, v) = \min_{\ll} lowest(S, v)$ as the "first" (in traversal order) of the minimum depth nodes in $succ(S, v)$, writing $nextLowest(S, v) = \bot$ if there is no such node.

Tree generators. While the semantics models a fully materialised search tree, any realistic implementation must generate the search tree on demand as in Section 4.1. We call a function $g : X^* \to X^*$ an *ordered tree generator* if all images of g are *isograms*, i.e. have no repeating letters. We define T_g, *the* tree generated by such a g as the smallest subset of X^* that contains ϵ and is closed under g, that is, if $u \in T_g$ and $g(u) = a_1 \ldots a_n, a_i \in X$, then all $ua_i \in T_g$. We equip T_g with *the* sibling order $\prec_{T_g}$ induced by g, defined as $ua_i \prec_{T_g} ua_j$ iff $g(u) = a_1 \ldots a_n$ and $i < j$. This defines a total order on siblings because images of g are isograms.

3.2 Search types

YewPar supports enumeration, optimisation and decision searches. All three search types can be characterised by a *commutative monoid* M for accumulating information and

an *objective function h* for mapping search tree nodes into the monoid.

Enumeration search traverses the entire search tree and gathers information by summing the values of the objective function. Such a search is defined by a commutative monoid $\langle M, +, 0 \rangle$ and an objective function $h : X^* \to M$, and searching an initial tree S_0 amounts to computing the sum $\sum \{h(v) \mid v \in S_0\}$.

Examples. To count the nodes in a search tree the monoid M is the natural numbers with addition and the objective function h is $h(v) = 1$. To count the nodes at a given depth d requires the same monoid M but a different objective function: $h(v) = 1$ if the depth of v is d, i.e. $|v| = d$, and $h(v) = 0$ otherwise.

Optimisation search computes the maximal value of the objective function across the search tree. This requires the commutative monoid $\langle M, +, 0 \rangle$ to induce a total order $\langle M, \sqsubseteq \rangle$ with least element 0, and with + acting as the max operator. Optimisation search typically returns not the maximal value but a search tree node witnessing that value. (There may be many such witnesses; optimisation search may pick one nondeterministically.) This requires the tracking of incumbents, and also provides opportunities for pruning the search tree (Section 3.5).

Example. A simple optimisation search computes tree depth. The monoid M is the natural numbers with maximum, which induces the usual total order, and the objective function h maps each node to its depth, i.e. $h(v) = |v|$.

Decision search, like optimisation search, computes the maximum of the objective function while traversing the search tree. (Like optimisation search, decision search typically returns a nondeterministically chosen node witnessing that maximum.) However, decision search requires the total order $\langle M, \sqsubseteq \rangle$ to be bounded, and terminates as soon as the objective function reaches the greatest element.

Example. A simple decision search decides whether a tree is at least k levels deep. The bounded total order is the set $\{0, \ldots, k\}$ with the usual order, i.e. k is the greatest element. The objective function h maps each node to its depth, cut off at level k, i.e. $h(v) = \min\{|v|, k\}$

3.3 Configurations

The semantics captures the current state of a parallel search in a *configuration* of the form $\langle \sigma, Tasks, \theta_1, \ldots, \theta_n \rangle$, where $n \geq 1$ is the fixed number of parallel threads. The components of a configuration are as follows.

θ_i is the state of the ith *thread*. It is either $\perp$ to denote an idle thread, or $\langle S, v \rangle^k$ to denote an active thread that is executing task S, i.e. searching subtree S in traversal order. The node currently being explored is v, and the superscript k records how often the search of S has backtracked (k may be omitted if not relevant).

Tasks is a queue of pending *tasks*, i.e. subtrees yet to be searched. We use list notation, so $[]$ is an empty queue, $[S]$

is a singleton queue, $S{:}Tasks$ is a queue with S at the head, and $Tasks{:}S$ is a queue with S at the tail.

σ is the current *global knowledge*, and is either of the form $\langle x \rangle$ or $\{u\}$. For enumeration searches, the *accumulator* $\langle x \rangle$ is an element of a commutative monoid M that sums the current knowledge. For optimisation and decision searches, the *incumbent* $\{u\}$ is a search tree node that currently maximises the objective function h.

Search begins with all threads idle, a singleton task queue and global knowledge being either the root node incumbent or the zero accumulator. That is, an initial configuration takes the form $\langle \sigma_0, [S_0], \perp, \ldots, \perp \rangle$, where S_0 is the entire search tree and σ_0 is either $\{\epsilon\}$ or $\langle 0 \rangle$. Search ends when the task queue is empty and all threads are idle, i.e. a final configuration is $\langle \sigma, [], \perp, \ldots, \perp \rangle$, where σ is a search result.

3.4 Reduction rules

Figure 2 lists the reduction rules of the multi-threaded semantics. The rules are divided into four categories and define reduction relations $\to_i^T$, $\to_i^N$, $\to_i^P$ and $\to_i^S$ for tree traversal, node processing, pruning and spawning respectively. The subscript i indicates the active thread performing a reduction step. The per-thread and overall reduction relations $\to_i$ and $\to$ are defined as follows.

$$\to_i = (\to_i^T \circ \to_i^N) \cup \to_i^P \cup \to_i^S$$
$$\to = \to_1 \cup \cdots \cup \to_n$$

Every $\to$ reduction is a per-thread reduction for some thread i, which is either a spawn reduction, a prune reduction, or a traversal reduction followed immediately by a node processing reduction.

The traversal rules encode standard backtracking, searching a subtree S in traversal order, starting at the root of S (schedule), expanding the current branch (expand), backtracking to another branch (backtrack), and terminating once S is explored (terminate). The search type determines which node processing rules are applicable. Enumeration searches accumulate the value of the objective function using the monoid addition + (accumulate). Optimisation and decision searches update the incumbent after comparing its objective value to the current node using the total order $\sqsubseteq$ (strengthen/skip). The (noop) rule prevents node processing getting stuck after (terminate).

3.5 Pruning

Optimisation and decision searches admit *pruning* the search tree, i.e. removing subtrees that can never improve the current incumbent. Semantically, this is reflected by the (prune) rule. What to prune is decided by search-specific heuristics. The semantics abstracts the heuristics as a binary relation ▷ on search tree nodes, where $u \rhd v$ states that u *justifies pruning* v. The ▷ relation must satisfy the following admissibility conditions w.r.t. the objective function h and the total order $\langle M, \sqsubseteq \rangle$.

$$(\text{schedule}_i) \ \frac{v = \text{root of } S}{\langle \sigma, S{:}Tasks, \ldots, \perp, \ldots \rangle \rightarrow_i^T \langle \sigma, Tasks, \ldots, \langle S, v \rangle^0, \ldots \rangle}$$

$$(\text{expand}_i) \ \frac{v' = next(S, v) \qquad v' \neq \perp \qquad v \leq v'}{\langle \sigma, Tasks, \ldots, \langle S, v \rangle^k, \ldots \rangle \rightarrow_i^T \langle \sigma, Tasks, \ldots, \langle S, v' \rangle^k, \ldots \rangle}$$

$$(\text{backtrack}_i) \ \frac{v' = next(S, v) \qquad v' \neq \perp \qquad v \not\leq v'}{\langle \sigma, Tasks, \ldots, \langle S, v \rangle^k, \ldots \rangle \rightarrow_i^T \langle \sigma, Tasks, \ldots, \langle S, v' \rangle^{k+1}, \ldots \rangle}$$

$$(\text{terminate}_i) \ \frac{next(S, v) = \perp}{\langle \sigma, Tasks, \ldots, \langle S, v \rangle^k, \ldots \rangle \rightarrow_i^T \langle \sigma, Tasks, \ldots, \perp, \ldots \rangle}$$

$$(\text{accumulate}_i) \ \frac{}{\langle \langle x \rangle, Tasks, \ldots, \langle S, v \rangle, \ldots \rangle \rightarrow_i^N \langle \langle x + h(v) \rangle, Tasks, \ldots, \langle S, v \rangle, \ldots \rangle}$$

$$(\text{strengthen}_i) \ \frac{h(v) \sqsupset h(u)}{\langle \{u\}, Tasks, \ldots, \langle S, v \rangle, \ldots \rangle \rightarrow_i^N \langle \{v\}, Tasks, \ldots, \langle S, v \rangle, \ldots \rangle}$$

$$(\text{skip}_i) \ \frac{h(v) \sqsubseteq h(u)}{\langle \{u\}, Tasks, \ldots, \langle S, v \rangle, \ldots \rangle \rightarrow_i^N \langle \{u\}, Tasks, \ldots, \langle S, v \rangle, \ldots \rangle}$$

$$(\text{noop}_i) \ \frac{}{\langle \sigma, Tasks, \ldots, \perp, \ldots \rangle \rightarrow_i^N \langle \sigma, Tasks, \ldots, \perp, \ldots \rangle}$$

$$(\text{prune}_i) \ \frac{u \triangleright v \qquad S' = subtree(S, v) \setminus \{v\} \qquad S' \neq \emptyset}{\langle \{u\}, Tasks, \ldots, \langle S, v \rangle, \ldots \rangle \rightarrow_i^P \langle \{u\}, Tasks, \ldots, \langle S \setminus S', v \rangle, \ldots \rangle}$$

$$(\text{shortcircuit}_i) \ \frac{h(u) \text{ is greatest element}}{\langle \{u\}, Tasks, \ldots, \langle S, v \rangle, \ldots \rangle \rightarrow_i^P \langle \{u\}, [], \perp, \ldots, \perp \rangle}$$

$$(\text{spawn}_i) \ \frac{u \in S \qquad v \ll u \qquad S_u = subtree(S, u)}{\langle \sigma, Tasks, \ldots, \langle S, v \rangle, \ldots \rangle \rightarrow_i^S \langle \sigma, Tasks{:}S_u, \ldots, \langle S \setminus S_u, v \rangle, \ldots \rangle}$$

$$(\text{spawn-depth}_i) \ \frac{|v| < d_{cutoff} \qquad \{u_1, \ldots, u_m\} = children(S, v) \neq \emptyset \qquad u_1 \ll \cdots \ll u_m \qquad S_1 = subtree(S, u_1) \ \ldots \ S_m = subtree(S, u_m)}{\langle \sigma, Tasks, \ldots, \langle S, v \rangle, \ldots \rangle \rightarrow_i^S \langle \sigma, Tasks{:}S_1{:}\ldots{:}S_m, \ldots, \langle S \setminus S_1 \setminus \cdots \setminus S_m, v \rangle, \ldots \rangle}$$

$$(\text{spawn-budget}_i) \ \frac{k \geq k_{budget} \qquad \{u_1, \ldots, u_m\} = lowest(S, v) \neq \emptyset \qquad u_1 \ll \cdots \ll u_m \qquad S_1 = subtree(S, u_1) \ \ldots \ S_m = subtree(S, u_m)}{\langle \sigma, Tasks, \ldots, \langle S, v \rangle^k, \ldots \rangle \rightarrow_i^S \langle \sigma, Tasks{:}S_1{:}\ldots{:}S_m, \ldots, \langle S \setminus S_1 \setminus \cdots \setminus S_m, v \rangle^0, \ldots \rangle}$$

$$(\text{spawn-stack}_i) \ \frac{u = nextLowest(S, v) \neq \perp \qquad S_u = subtree(S, u)}{\langle \sigma, [], \ldots, \langle S, v \rangle, \ldots \rangle \rightarrow_i^S \langle \sigma, [S_u], \ldots, \langle S \setminus S_u, v \rangle, \ldots \rangle}$$

Figure 2. Reduction rules of the operational semantics.

1. For all u and v, if $u \triangleright v$ then $h(u) \sqsupseteq h(v)$.
2. For all u', u and v, if $h(u') \sqsupseteq h(u)$ and $u \triangleright v$ then $u' \triangleright v$.
3. For all u, v and v', if $u \triangleright v$ and $v \leq v'$ then $u \triangleright v'$.

Condition 1 states correctness of pruning w.r.t. maximising the objective function: if u justifies pruning v then $h(u)$ dominates $h(v)$. Condition 2 allows strengthening of incumbents: if u justifies pruning v then any stronger incumbent u' also will. Condition 3 allows pruning entire subtrees: if u justifies pruning v then any descendent of v can also be pruned.

3.6 Spawning

The semantics includes a (spawn) rule to model space-splitting parallel search. A (spawn) reduction hives off some subtree S_u of the current thread $\langle S, v \rangle$ into a new task S_u that is added to the task queue. S_u must be *unexplored*, i.e. its root u is visited after the current node v in traversal order.

The search coordinations used in YewPar skeletons (Section 4.2) implement more complex space-splitting behaviours, selecting specific groups of subtrees to spawn as tasks, in a specific order. We model these coordination behaviours as derived spawn rules. Semantically, these rules are redundant; they are included to faithfully model the coordination behaviour of particular YewPar skeletons.

The (spawn-depth) rule models Depth-Bounded search coordination. The rule fires if the depth of the current node v is less than the d_{cutoff} parameter. The rule spawns all subtrees of S rooted at children of v and queues them in traversal order. The rule causes the eager spawning of the top d_{cutoff} levels of the search tree, queued in heuristic search order.

The (spawn-budget) rule models Budget search coordination. The rule fires if the backtrack counter k of the current

thread exceeds the k_{budget} parameter. The rule spawns all unexplored subtrees of the current thread $\langle S, v \rangle$ at lowest depth, i.e. closest to the root of S. New tasks are queued in traversal order, and the backtrack counter of the current thread is reset to 0. The rule periodically generates new tasks from threads that contain significant amounts of work since search has not completed within the backtrack budget.

The (spawn-stack) rule models Stack-Stealing search coordination. It differs from the other rules in that it only fires when the task queue is empty, and only spawns one new task. That task is the first (in traversal order) of the unexplored lowest-depth subtrees of the current thread $\langle S, v \rangle$. The rule is designed to split the search space on demand, generating a task to be stolen by an idle thread.

The YewPar implementation of Stack-Stealing lets a thief steal directly from the victim. Semantically, this behaviour corresponds to a (spawn-stack$_i$) reduction followed by a (schedule$_j$) reduction, modelling idle thread j stealing from victim thread i, with the task queue acting as a transit buffer for the stolen task.

3.7 Correctness

The semantics is correct if every sequence of reductions on a search tree eventually computes the same sum or maximum of the objective function, independent of the particular reduction sequence. For enumeration searches, correctness amounts to termination and confluence of the reduction relation, but optimisation and decision searches may nondeterministically return any optimal witness, hence the reduction relation cannot be confluent in general. The following theorems formalise this statement of correctness.

Theorem 3.1. *Let S_0 be a search tree for an enumeration search. If $\langle\langle 0\rangle, [S_0], \bot, \ldots, \bot\rangle \rightarrow^* \langle\langle x\rangle, [], \bot, \ldots, \bot\rangle$ then $x = \sum\{h(v) \mid v \in S_0\}$.*

Proof sketch. All rules except (terminate) keep the total set of nodes in a configuration invariant. The traversal order ensures that nodes removed by (terminate) have been visited and processed by (accumulate) exactly once. The claim follows as every node is eventually removed by (terminate). □

Theorem 3.2. *Let S_0 be a search tree for an optimisation or decision search. If $\langle\{\epsilon\}, [S_0], \bot, \ldots, \bot\rangle \rightarrow^* \langle\{\hat{u}\}, [], \bot, \ldots, \bot\rangle$ then $h(\hat{u}) = \max\{h(v) \mid v \in S_0\}$.*

Proof sketch. The claim is obvious if $h(\hat{u})$ is the greatest element and the (shortcircuit) rule fires. Otherwise, arguments similar to the proof of Theorem 3.1 show that any nodes removed by rules (terminate) and (prune) cannot beat the current incumbent, let alone the final incumbent $\hat{u}$, hence the claim follows. □

Theorem 3.3. *The reduction relation $\rightarrow$ is terminating.*

Proof sketch. We map each configuration to a *measure*, namely a multiset containing the number of nodes of each task and the number of unexplored nodes of each thread. The claim holds as each $\rightarrow$ step strictly decreases the measure w.r.t. Dershowitz and Manna's well-founded multiset order [15]. □

4 Composing Tree Searches
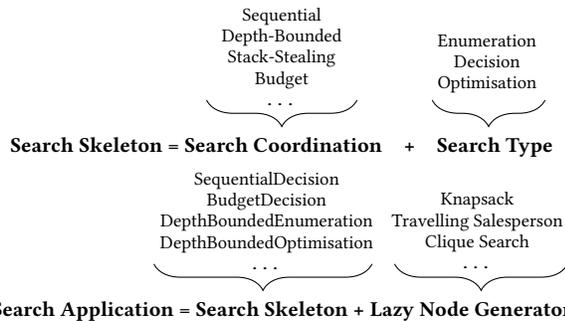
Figure 3. Structure of Search Skeletons and Applications.

YewPar search applications are constructed by combining a search skeleton with a Lazy Node Generator, as shown in Fig. 3. Skeletons in turn comprise a search type, e.g *Decision*, and a search coordination, e.g. *Depth-Bounded* (Section 4.2). Most of the search is parameterised generic code. Notable exceptions are the Lazy Node Generator that is application-specific, and is provided by the user to specify how to generate the search tree (on demand) and in which order to traverse the tree. Likewise the bounding functions or pruning predicates are application-specific, and must be specified.

The decomposition of a search application into a Lazy Node Generator, search coordination and search type reflects how the reduction rules in Figure 2 are factored into groups for traversal, node processing, pruning and work generation. The traversal rules are implemented by all search coordination methods and call the node generator. The node processing and pruning rules are determined by the search type. Each work generation rule is implemented by a specific search coordination. The search skeleton library is extensible, allowing the addition of new search coordination methods. For example, new coordination methods may provide best-first search or random task creation.

4.1 Lazy Node Generators

Let T_g be a search tree generated by ordered tree generator g (Section 3.1). Given a parent node $u \in T_g$, the list $[ua_1, \ldots, ua_n]$ enumerates the children of u in traversal order, where $g(u) = a_1 \ldots a_n$.

A *Lazy Node Generator* for u is a data structure that provides iteration over the list $[ua_1, \ldots, ua_n]$. Lazy Node Generators implement the following interface, parameterised by types for the search space and the search tree nodes.

```
1  template <typename SearchSpace, typename Node>
2  struct NodeGenerator {
3    // Returns true if there are more children
4    virtual bool hasNext();
5
6    // Returns the next child, provided hasNext() == true
7    virtual Node next();
8
9    // Constructs a NodeGenerator for a given parent node
10   NodeGenerator(const SearchSpace & space,
11                 const Node & node);
12 };
```

Node generators only *generate* the children of a node, in the order in which they are to be traversed. They do not determine *how* and *when* the search tree is constructed; these are specified by the skeleton implementations (Section 4.2). In particular, depth-first backtracking tree traversal is implemented by maintaining a stack of node generators, as exemplified by the *Sequential* search coordination (Listing 2). In each iteration, the node generator at the top of the stack is advanced and a new node generator for the child is pushed onto the stack, corresponding to the (expand) rule. When the node generator at the top of the stack is empty, the search coordination pops the empty generator off the stack and advances the generator below, corresponding to the (backtrack) rule.

Besides backtracking, the generator stack is also helpful for identifying which subtrees to spawn as tasks, as exemplified by the *Budget* and *Stack-Stealing* search coordinations.

A Lazy Node Generator does not compute the entire list of children of a node upfront. Instead, the children are materialised *lazily* as the node generator is advanced using the next method, rather like a Python generator. This has two advantages: (1) A stack of Lazy Node Generators typically uses less memory than a stack of fully materialised lists of children. (2) Often the full list of children is not required, e.g. it is possible to prune all "future" children "to-the-right" once

Listing 1. Lazy Node Generator for maximum clique.

```cpp
1   typedef std::bitset<N>         VertexSet;
2   typedef std::vector<VertexSet> Graph;
3
4   // Greedily colours the subgraph induced by vertex set p.
5   // On return, array p_vertex enumerates the set p, and
6   // p_colour[i] is the number of colours used to colour
7   // the set of vertices p_vertex[0], ..., p_vertex[i].
8   void greedy_colour(const Graph & graph,
9                      const VertexSet & p,
10                     std::array<int,N> & p_vertex,
11                     std::array<int,N> & p_colour);
12
13  // Search tree node
14  struct Node {
15    VertexSet clique;     // current clique
16    int size;             // size of current clique
17    VertexSet candidates; // candidates to extend clique
18    int bound;            // bound on clique extensions
19
20    // Objective function (to be maximized)
21    int getObj() const { return size; }
22
23    // Not shown: Node constructor, serialisation
24  };
25
26  // Upper bound function (determines when to prune)
27  int upperBound(const Graph & graph, const Node & node) {
28    return node.getObj() + node.bound;
29  }
30
31  // Lazy node generator
32  struct Gen : NodeGenerator<Graph, Node> {
33    std::reference_wrapper<const Graph> graph;
34    std::reference_wrapper<const Node> parent;
35    std::array<int,N> p_vertex; // candidates (ordered)
36    std::array<int,N> p_colour; // number of colours used
37    VertexSet remaining;  // set of remaining candidates
38    int k;  // current index into array p_vertex
39
40    // Constructor
41    Gen(const Graph & graph, const Node & parent) :
42      graph(std::cref(graph)), parent(std::cref(parent))
43    {
44      remaining = parent.candidates;
45      greedy_colour(graph, remaining, p_vertex, p_colour);
46      k = remaining.count();
47    }
48
49    bool hasNext() override {  return k > 0; }
50
51    Node next() override {
52      k--;
53      int v = p_vertex[k];  // candidate to be added
54      remaining.reset(v);
55      VertexSet clique = parent.get().clique;
56      clique.set(v) ;
57      int size = parent.get().size + 1;
58      VertexSet candidates = remaining;
59      candidates &= graph.get()[v];
60      int bound = p_colour[k];
61      return Node(clique, size, candidates, bound);
62    }
63  };
```

a bounds check establishes that the current node cannot beat the incumbent.

Lazy Node Generator Example. Listing 1 demonstrates a Lazy Node Generator implementing a state-of-the-art Maximum Clique algorithm [26]. Graphs (of up to N vertices) are represented as vectors mapping vertices to sets of adjacent vertices, and vertex sets are represented as fixed-size bitsets. The bitset representation enables vectorisation of set

operations, which is known to speed up Maximum Clique implementations up to 20-fold [36]. The Maximum Clique algorithm relies on a standard greedy colouring heuristic for pruning; the prototype is declared on line 8, see [26] for the specific algorithm.

The Node struct (line 14) represents a search tree node. It stores the current clique (including its size), a set of candidate vertices that may extend the clique, and an upper bound on the number of vertices that can yet be added to the current clique. Node also provides the getObj method (line 21) which returns the current objective value of the node, and which is maximised by optimisation and decision searches. Branch-and-bound pruning decisions are taken by comparing the best objective value found so far with the result of the upperBound function (line 27).

The Lazy Node Generator, struct Gen, starts at line 32. It stores a reference to the graph (i.e. the search space) and to its parent node. The constructor (line 41) copies the parent's set of candidates and calls a greedy heuristic to colour the subgraph induced by these candidates. On return, array p_vertex stores the candidates in reverse heuristic order, and p_colour[i] stores the number of colours used to colour the set of vertices $\{p\_vertex[0], \ldots, p\_vertex[i]\}$, which is an upper bound on the number of vertices that can be added to the parent's clique. Finally, the constructor initialises the iterator index k to the end of array p_vertex, i.e. k points to the heuristically best candidate.

The lazy node generator implements the NodeGenerator interface by iterating over array p_vertex in reverse order. The hasNext method (line 49) simply checks the iterator index. The next method (line 51) advances the iterator index, removes the current candidate v from the remaining candidates (line 54), and adds it to a copy of the parent's clique (line 56). Then, it intersects a copy of the set of remaining vertices with the vertices that are adjacent to the current candidate v (line 59), forming a new set of candidates for extending the extended clique. Finally, next returns a new Node (line 61) containing the extended clique, the new set of candidates, and an upper bound on the number of vertices that can yet be added to the extended clique.

4.2 Search Coordination Methods

During a search potentially any node in the search tree may be converted to a task, but to minimise search time it is critical to choose heuristically a *good* node. We follow prior work e.g. [32], and use both application heuristics (as encoded in the Lazy Node Generator), and select large subtrees (to minimise communication and scheduling overheads) that we expect to find close to the root of the search tree.

Sequential search coordination corresponds to a semantics without spawn rules, i.e. it executes in a single thread that performs a depth-first search from the root node. The pseudocode is given in Listing 2, where the effect of processNode

Listing 2. Sequential Search Coordination Pseudocode.

```
1  function Sequential(SearchType stype, SearchSpace space,
2                       Node root):
3    processNode(stype, root)
4    genStack.push(NodeGenerator(space, root))
5    while not genStack.empty() do
6      gen ← genStack.top()
7      if gen.hasNext() then
8        child ← gen.next()
9        processNode(stype, child)
10       genStack.push(NodeGenerator(space, child))
11     else
12       genStack.pop()          // Backtrack
```

Listing 3. Stack-Stealing Search Coordination Pseudocode.

```
1  function StackStealing(SearchType stype, SearchSpace
       space, Node root):
2    steal ← getWorkerStealChannel()
3    processNode(stype, root)
4    genStack.push(NodeGenerator(space, root))
5    while not genStack.empty() do
6      if response = steal.non_blocking_get() then
7        for gen ← genStack.bottom() to genStack.top() do
8          if gen.hasNext() then
9            tRoot ← gen.next()
10           t ← StackStealing(stype, space, tRoot)
11           response.put(t)
12           break
13         response.put(Nothing)
14       else                      // Continue Seq. Search
15         gen ← genStack.top()
16         if gen.hasNext() then
17           child ← gen.next()
18           processNode(stype, child)
19           genStack.push(NodeGenerator(space, child))
20         else
21           genStack.pop()      // Backtrack
```

Listing 4. Budget Search Coordination Pseudocode.

```
1  function Budget(SearchType stype, SearchSpace space,
2                  Node root, int btBudget):
3    backtracks ← 0
4    processNode(stype, root)
5    genStack.push(NodeGenerator(space, root))
6    while not genStack.empty() do
7      if backtracks ≥ btBudget then
8        for gen ← genStack.bottom() to genStack.top() do
9          if gen.hasNext() then
10           while gen.hasNext() do
11             tRoot ← gen.next()
12             t ← Budget(stype, space, tRoot, btBudget)
13             spawn(t)
14           break
15         backtracks ← 0
16       else                      // Continue Seq. Search
17         gen ← genStack.top()
18         if gen.hasNext() then
19           child ← gen.next()
20           processNode(stype, child)
21           genStack.push(NodeGenerator(space, child))
22         else
23           genStack.pop()      // Backtrack
24           backtracks ← backtracks + 1
```

function (lines 3 and 9) is determined by the particular type of search, e.g. optimisation.

Depth-Bounded search coordination converts all nodes below a cut-off depth d_{cutoff} into tasks. The YewPar implementation is similar to sequential, except that it tracks the current search depth, and, up until the cutoff depth, spawns tasks for each child node. These spawns occur as tasks execute rather than generating all work upfront. Depth-Bounded is a direct implementation of the (spawn-depth) rule, except that it uses a distributed workpool to achieve scalability.

Stack-Stealing search coordination is a dynamic work generation approach that splits the search tree on receipt of a work-stealing request. YewPar implements the (spawn-stack) rule and the pseudocode is given in Listing 3. It checks for a stealRequest on *every* search expansion step (line 6). If a steal is requested the generator stack is searched from bottom to top (line 7), i.e. nodes (one node, or all at the lowest depth if the *chunked* flag is set) closest to the root first. Either work is returned to the thief (line 11), or the steal fails (line 13). Steal requests are sent via atomic channels between thieves and victims.

The YewPar implementation combines work pushing to initially distribute search tasks to workers, and once these are exhausted the workers switch to work (stack) stealing. Victim selection is random, although in a distributed environment remote workers are only selected if there are no active local workers. As workers communicate directly, no work pool is required, and the implementation is similar to [1].

Budget search coordination is a dynamic approach with asynchronous periodic load-balancing. Each worker has a period based on the number of backtracks performed. Workers search subtrees until the task is complete or the task has backtracked as many times as specified in a user-defined *budget*, btBudget in the Listing 4 pseudocode. If the budget is exhausted (line 7), all nodes at the lowest depth are spawned and the budget is reset (lines 8–15). YewPar implements the (spawn-budget) rule, except for using a distributed workpool to achieve scalability.

4.3 YewPar Realisation

The YewPar parallel C++ search framework[1] not only implements the Lazy Node Generators and search skeletons, but also provides low-level components such as search specific work-stealing schedulers and workpools with which new skeletons can be created. To support distributed memory parallelism, YewPar builds on the HPX [22] task parallelism library and runtime system. HPX is routinely deployed on HPC and Cloud architectures, and YewPar can readily exploit this portability at scale. Complete descriptions of the YewPar design and implementation are available in [3, 5]; it has the following primary components.

Search Specific Schedulers. YewPar layers the search coordination methods as custom schedulers on top of the existing HPX scheduler. That is, the HPX scheduler manages several lightweight YewPar scheduler threads that perform the search. The (local) thread to core mapping is handled transparently by HPX. We divide operating system threads into two types: *worker* threads that continuously seek and

---

[1] https://github.com/BlairArchibald/YewPar

Listing 5. Composing a YewPar Search Application for a Maximising Search, using Stack-Stealing and Optimisation.

```
1  Node maximal_solution =
2    YewPar::Skeletons::StackStealing< // search coord
3      Gen,                            // lazy node gen
4      Optimisation,                   // search type
5      BoundFunction<upperBound>       // bound for pruning
6    >::search(space, root);
```

execute search tasks, and *manager* threads (one per locality) that are managed entirely by HPX and manage aspects like messages and termination. Where workers look for work is determined by the skeleton being executed. Work may come directly from other workers, as in the Stack-Stealing skeleton, or from a distributed workpool, where steals are sent to remote nodes if no work is available locally. The schedulers seek to preserve search order heuristics, e.g. by using a bespoke order-preserving workpool [3, 5].

Knowledge Management. The sharing of solutions and bounds relies on HPX's partitioned global address space (PGAS). To minimise distributed queries, bounds are broadcast to localities that keep track of the last received bound. The local bound does not need to be up-to-date to maintain correctness, hence YewPar can tolerate communication delays at the cost of missing pruning opportunities.

Skeletons API. C++ templates are used to statically specialise the generic skeletons into efficient application-specific implementations. Compile time type information allows stack based memory allocation and compile time elimination of unused branches, e.g. if branch and bound is disabled then all pruning code is removed.

The return type of the skeletons is derived from the template parameters, for example a skeleton implementation called with an *Optimisation* parameter will return the optimal search tree node.

The skeleton APIs expose parameters like depth *cutoff* or backtracking *budget* that control the parallel search. These parameters determine the amount and location of work in the system, and poor parameter choices can starve or overload the system (Section 5.5).

4.4 Composing YewPar Search Applications

Listing 5 illustrates how a YewPar search application is composed following the model outlined in Figure 3; see Appendix A.3 for further, more detailed examples.

The search in Listing 5 returns the maximal node in some space. A YewPar user simply selects the search coordination, in this case `StackStealing`, provides the application-specific Lazy Node Generator `Gen` to generate the space, and chooses the type of search, here `Optimisation`. The listing also illustrates how YewPar implements the (prune) rule from Figure 2. The user provides an application-specific `BoundFunction` that is called on each search tree node and prunes if the bound cannot beat the current objective. For

efficiency the `BoundFunction` pointer is lifted to template level so that the `upperBound` function can be inlined.

5 Evaluation

5.1 Search Applications

We evaluate YewPar on a representative sample of 7 exact combinatorial search applications covering the 3 search types, as follows.

Enumeration: *Unbalanced Tree Search (UTS)* dynamically constructs synthetic irregular tree workloads based on a given *branching factor*, *depth*, and *random seed* [30]. *Numerical Semigroups (NS)* counts how many numerical semigroups there are of a particular genus [17]. A numerical semigroup S is a cofinite set of natural numbers containing 0 and closed under addition; the genus of S is the size of its complement.

Optimisation: *Maximum Clique (MaxClique)* finds the largest clique, i.e. largest set of pairwise adjacent vertices, in a given graph. *0/1 Knapsack* determines the optimal set of items, each with profit and weight, to place in a container such that profit is maximised subject to a given weight limit. *Travelling Salesperson (TSP)* finds a shortest circular tour of N cities.

Decision: The *Subgraph Isomorphism Problem (SIP)* determines whether a copy of a given pattern graph is present in a target graph. The *k-Clique* variant of Maximum Clique finds a clique of k vertices if one exists in the graph.

The baseline implementations for MaxClique [26], SIP [27], NS [17], and UTS [30] use published state-of-the-art algorithms. The sequential C++ implementations were provided by the domain experts, and not written by the authors. A full description of the applications and instances is in [3]. Data underpinning the analysis in this section is openly available [7].

5.2 Experimental Setup

We report measurements on up to 17 machines, each with a dual 8-core Intel Xeon E5-2640v2 2GHz CPU (no hyper-threading), 64GB RAM, running Ubuntu 14.04.3 LTS. We reserve one core for HPX (Version 1.0, commit 51d3d0) for task management, i.e. on 16 cores we use 15 workers.

Caveat. Performance analysis of parallel searches is notoriously difficult as nondeterminism caused by pruning, finding alternate valid solutions, and random work-stealing, can lead to performance anomalies (Section 2.1), manifested as superlinear speedups/slowdowns. We control for this by investigating 7 search applications, running each experiment multiple times, and reporting cumulative statistics.

5.3 YewPar Overheads

Lazy Node Generators incur some overheads compared to search specific implementations as they decouple search tree generation and traversal. For example, they copy search tree nodes instead of updating in-place. We evaluate these

Table 1. Comparing YewPar runtimes (s) with Hand-written Maximum Clique Implementations: Sequential and OpenMP (15 workers). Mean parallel slowdown computed for instances with runtimes over 1.5s (in bold).

Instance	Seq C++	Seq YewPar	Slow-down (%)	OpenMP C++	Depth-Bounded YewPar	Slow-down (%)
MANN_a45	169.90	160.49	−5.54	**42.69**	**42.25**	−1.02
brock400_1	285.32	340.59	19.37	**28.91**	**29.99**	3.73
brock400_2	213.89	249.49	16.64	**11.34**	**12.57**	10.86
brock400_3	163.72	197.42	20.58	**6.16**	**7.97**	29.50
brock400_4	79.82	95.33	19.43	**1.82**	**3.44**	89.50
brock800_4	1,433.26	1,572.94	9.75	**115.84**	**147.36**	27.21
p_hat1000-2	127.77	140.43	9.91	**9.94**	**11.68**	17.43
p_hat1500-1	2.86	2.76	−3.29	0.44	1.15	161.02
p_hat300-3	1.13	1.29	13.35	0.12	0.12	7.97
p_hat500-3	115.12	123.71	7.46	**9.57**	**10.97**	14.68
p_hat700-2	2.44	2.60	6.61	0.25	0.26	4.09
p_hat700-3	1,278.03	1,312.76	2.72	**98.96**	**118.93**	20.18
san1000	1.70	1.58	−7.08	0.21	1.09	413.38
san400_0.7_2	2.04	2.27	11.48	0.14	0.15	6.98
san400_0.7_3	1.26	1.35	7.50	0.06	0.12	113.84
san400_0.9_1	24.81	25.08	1.08	0.17	0.53	221.39
sanr200_0.9	26.17	29.30	11.93	**2.66**	**3.01**	13.14
sanr400_0.7	74.95	91.44	22.00	**7.87**	**8.01**	1.70
Geo. Mean			8.76			16.56

Figure 4. Scaling for k-clique: 255 workers, 17 locations. Speedups for 18 additional applications are in Table 2.

overheads on Maximum Clique as a competitive sequential implementation is available [25, 26].

The first 4 columns of Table 1 show the mean sequential runtimes (over 5 runs) for the 18 DIMACS clique instances [21] that take between 1 second and 1 hour to run sequentially. The results show a limited cost of generality, i.e. a maximum overhead of 22.0%, a minimum overhead of -5.5%, and geometric mean overhead of just 8.8%.

Parallel Execution adds additional overheads, e.g. the YewPar skeletons are parametric rather than specialised, and a distributed memory execution framework is relatively heavyweight. To evaluate the scale of these overheads we compare with a search-specific OpenMP version of the maximum clique implementation. It is imperative that the parallel search algorithm and coordination are almost identical, as otherwise performance anomalies will disrupt the comparison. Hence the lazy node generator is carefully crafted to mimic the Maximum Clique implementation [26], and the OpenMP implementation uses the `task` pragma to construct a set of tasks for each node at depth 1, closely analogous to the Depth-Bounded skeleton in the YewPar implementation.

Columns 5-7 of Table 1 compare the runtimes of the YewPar and OpenMP versions for the DIMACS search instances on a single location, i.e. with 15 workers. Instances with very short runtimes can exhibit large relative performance differences, e.g. an increase of only 0.36s in runtime for san400_0.9_1 is a 221% slowdown. To avoid skewing the summative performance comparison, we compute the mean slowdown on instances with a runtime of over 1.5s. The geometric mean slowdown is 16.6%, with a maximum slowdown of 89.5% (brock400_4, absolute slowdown 1.6s). We conclude that for these instances the parallel overheads of YewPar remain moderate.
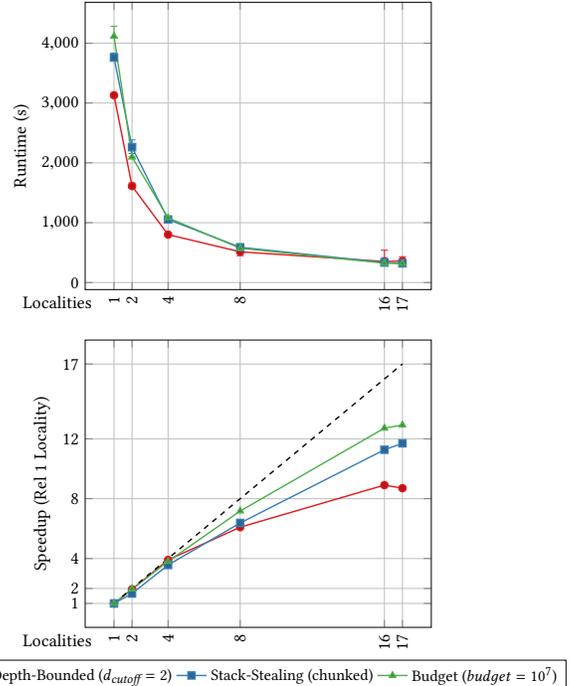
However, the YewPar parallelisation is more flexible than using OpenMP. For a start, OpenMP is restricted to shared memory multicores whereas YewPar scales on distributed memory clusters (Section 5.4). Moreover, experimenting with alternate parallelisations in YewPar entails changing a single line of code (Section 5.5). In contrast, it is far from trivial to engineer performant implementations of advanced search coordinations like Stack-Stealing or Budget in OpenMP.

5.4 YewPar Scalability

To demonstrate scalability Fig. 4 shows the runtime and relative speedups achieved by three skeletons for a large instance (sequential runtimes around 1 hour) of the k-clique benchmark – searching for a spread in H(4, 4) [6] – using 255 workers across 17 localities. Speedup is relative to a single locality with 15 workers.

This example also illustrates YewPar running on multiple architectures, i.e. a single 16-core shared memory machine, and on a cluster of such machines.

These good scalability results are typical, for example Table 2 in the following section reports 18 further scaling results, and others are reported in Chapter 6 of [3]. YewPar commonly achieves a parallel efficiency of >50% even for these highly irregular computations, e.g. the best scaling results for 5 of the 6 benchmarks in Table 2 exceed 60x speedup with 120 workers.

Table 2. Comparing 18 Alternate Application Parallelisations: Mean Speedup on 120 workers

Application	Skeleton	Worst Speedup	Random Speedup	Best Speedup
MaxClique	Depth-Bounded	0.89	14.04	**91.74**
	Stack-Stealing	21.27	28.43	37.67
	Budget	1.38	7.58	17.84
TSP	Depth-Bounded	3.99	38.86	**68.19**
	Stack-Stealing	18.01	20.79	26.27
	Budget	1.25	4.89	39.52
Knapsack	Depth-Bounded	0.81	0.87	0.92
	Stack-Stealing	5.84	5.84	19.17
	Budget	1.06	8.75	**36.85**
SIP	Depth-Bounded	3.92	33.15	68.04
	Stack-Stealing	101.92	105.54	**109.61**
	Budget	1.42	7.70	45.05
NS	Depth-Bounded	0.87	0.87	1.47
	Stack-Stealing	26.37	30.95	30.95
	Budget	3.12	3.12	**59.48**
UTS	Depth-Bounded	1.48	8.72	9.56
	Stack-Stealing	52.68	57.52	57.52
	Budget	29.81	85.85	**85.85**
All	Depth-Bounded	1.67	8.69	26.51
	Stack-Stealing	27.99	34.21	**43.51**
	Budget	1.87	8.99	35.11

5.5 Exploring Search Parallelisations

In the literature different parallelisations are commonly applied to different search applications, if for example there are many tasks near the root (*e.g.,* Maximum Clique), Depth-Bounded often works well. Those that initially have a narrow tree (*e.g.,* NS) require more dynamic parallelism with associated overheads.

In contrast the parametric YewPar skeletons allow users to readily experiment with, and select, an appropriate skeleton parallelisation for a given application. Table 2 illustrates this by reporting cumulative speedups for the three coordinations on each search application. For each application we report the geometric mean speedup for all instances (around 20 in total) on 120 workers (8 physical localities) relative to the Sequential skeleton.

Most YewPar skeletons have parameters like d_{cutoff} and k_{budget} that must be tuned for each search application. To account for this, we run a parameter sweep, e.g. $d_{cutoff} = \{0 \ldots 8\}$, and $k_{budget} = \{10^4, 10^5, 10^6, 10^7\}$ [3]. Table 2 reports the worst, best and random scaling, corresponding to worst, best and some random choice of parameters.

No one skeleton performs best for all applications, with Depth-Bounded coming tops for two applications, Stack-Stealing for one, and Budget for three. Parameter tuning is tricky, and the consequences of getting it wrong are severe, *e.g.,* a slowdown (0.89x) versus a big speedup (91.74x) for MaxClique with Depth-Bounded. Stack-Stealing, with few parameters, minimises runtime variation and is a good choice if good parameters are not known.

6 Conclusion

We aim to improve the *reuse* of intricate parallel search implementations by providing the first general purpose scalable parallel framework for exact combinatorial search, YewPar. The semantics of YewPar backtracking search are defined by, and proved correct in, a novel parallel operational semantics (Section 3), which models parallel combinatorial search as a fold of the search tree into a monoid. The formalisation informs the YewPar design and implementation, and many elements recur: the three search types, search tree construction by ordered tree generator (i.e. lazy node generator), spawn rules specifying the behaviour of corresponding search coordinations.

We define Lazy Node Generators as a uniform API for search tree generation (Section 4.1). Uniquely in the field we demonstrate the ready composition of parallel search applications using Lazy Node Generators and a library of search skeletons, exhibiting 7 search applications (Section 5.1) covering the 3 search types.

We have designed and implemented 12 widely applicable algorithmic skeletons for tree search, and undertaken a systematic performance analysis of the skeletons on standard instances of 7 search applications. We show that the generic YewPar framework has low overheads (8.8% sequential slowdown on average), and that its parallel performance on big instances is similar to a specialised state-of-the-art implementation (Section 5.3). We report results for YewPar on two architectures: a multicore, and a cluster of multicores. Despite the high levels of irregularity, YewPar scales well, e.g. delivering maximal relative speedups of 195 on 255 workers (17 localities) for k-clique, and 110 on 120 workers (8 localities) for SIP (Section 5.4). Comparing the performance of the skeletons across search applications demonstrates the suitability of different skeletons for particular applications, illustrates the challenges of parameter selection, and shows that a skeleton approach makes it easy to explore alternative parallelisations (Section 5.5).

In ongoing work we are applying YewPar to new areas, and specifically to solving large bigraph matching problems.

Acknowledgments

References

[1] Faisal N. Abu-Khzam, Khuzaima Daudjee, Amer E. Mouawad, and Naomi Nishimura. 2015. On scalable parallel recursive backtracking. *J. Parallel and Distrib. Comput.* 84 (2015), 65–75. https://doi.org/10.1016/j.jpdc.2015.07.006

[2] Enrique Alba, Francisco Almeida, Maria J. Blesa, J. Cabeza, Carlos Cotta, Manuel Díaz, Isabel Dorta, Joaquim Gabarró, Coromoto León, J. Luna, Luz Marina Moreno, C. Pablos, Jordi Petit, Angélica Rojas, and Fatos Xhafa. 2002. MALLBA: A Library of Skeletons for Combinatorial Optimisation (Research Note). In *Euro-Par 2002, Parallel Processing, 8th International Euro-Par Conference Paderborn, Germany, August 27-30, 2002, Proceedings.* 927–932. https://doi.org/10.1007/3-540-45706-2_132

[3] Blair Archibald. 2018. *Skeletons for Exact Combinatorial Search at Scale.* Ph.D. Dissertation. University of Glasgow. http://theses.gla.ac.uk/id/eprint/31000

[4] Blair Archibald, Patrick Maier, Ciaran McCreesh, Robert J. Stewart, and Phil Trinder. 2018. Replicable parallel branch and bound search. *J. Parallel Distrib. Comput.* 113 (2018), 92–114. https://doi.org/10.1016/j.jpdc.2017.10.010

[5] Blair Archibald, Patrick Maier, Robert Stewart, and Phil Trinder. 2019. Implementing YewPar: A Framework for Parallel Tree Search [To Appear]. *EuroPar 19* (2019). https://doi.org/10.1007/978-3-030-29400-7_14

[6] Blair Archibald, Patrick Maier, Robert Stewart, Phil Trinder, and Jan De Beule. 2017. Towards Generic Scalable Parallel Combinatorial Search. In *Proceedings of PASCO 2017 (PASCO '17).* ACM, New York, NY, USA. https://doi.org/10.1145/3115936.3115942

[7] Blair Archibald, Patrick Maier, Phil Trinder, and Robert Stewart. 2019. YewPar: Skeletons for Exact Combinatorial Search [Data Collection]. (2019). https://doi.org/10.5525/gla.researchdata.935

[8] David Avis and Charles Jordan. 2017. mts: a light framework for parallelizing tree search codes. *CoRR* abs/1709.07605 (2017). http://arxiv.org/abs/1709.07605

[9] Christian Blum and Andrea Roli. 2003. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.* 35, 3 (2003), 268–308. https://doi.org/10.1145/937503.937505

[10] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. Cilk: An Efficient Multithreaded Runtime System. *J. Parallel Distrib. Comput.* 37, 1 (1996), 55–69. https://doi.org/10.1006/jpdc.1996.0107

[11] Adrian Brüngger, Ambros Marzetta, K. Fukuda, and Jürg Nievergelt. 1999. The parallel search bench ZRAM and its applications. *Annals OR* 90 (1999), 45–63. https://doi.org/10.1023/A3A1018972901171

[12] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA.* 519–538. https://doi.org/10.1145/1094811.1094852

[13] Philipp Ciechanowicz, Michael Poldner, and Herbert Kuchen. 2009. *The Münster Skeleton Library Muesli: A comprehensive overview.* Technical Report. Working Papers, ERCIS-European Research Center for Information Systems.

[14] A. de Bruin, G.A.P. Kindervater, and H.W.J.M. Trienekens. 1995. Asynchronous parallel branch and bound and anomalies. In *Parallel Algorithms for Irregularly Structured Problems*, Afonso Ferreira and José Rolim (Eds.). Lecture Notes in Computer Science, Vol. 980. Springer Berlin Heidelberg, 363–377. https://doi.org/10.1007/3-540-60321-2_29

[15] Nachum Dershowitz and Zohar Manna. 1979. Proving Termination with Multiset Orderings. *Comm. ACM* 22, 8 (1979), 465–476. https://doi.org/10.1145/359138.359142

[16] Raphael A. Finkel and Udi Manber. 1987. DIB - A Distributed Implementation of Backtracking. *ACM Trans. Program. Lang. Syst.* 9, 2 (1987), 235–256. https://doi.org/10.1145/22719.24067

[17] Jean Fromentin and Florent Hivert. 2016. Exploring the tree of numerical semigroups. *Math. Comp.* 85, 301 (2016), 2553–2568. https://doi.org/10.1090/mcom/3075

[18] Bernard Gendron and Teodor Gabriel Crainic. 1994. Parallel Branch-and-Branch Algorithms: Survey and Synthesis. *Operations Research* 42, 6 (1994), 1042–1066. https://doi.org/10.1287/opre.42.6.1042

[19] Ian P. Gent, Ciaran McCreesh, Ian Miguel, Neil C. A. Moore, Peter Nightingale, Patrick Prosser, and Chris Unsworth. 2018. A Review of Literature on Parallel Constraint Solving. *CoRR* abs/1803.10981 (2018).

[20] Jan Gmys, Rudi Leroy, Mohand Mezmaz, Nouredine Melab, and Daniel Tuyttens. 2016. Work stealing with private integer-vector-matrix data structure for multi-core branch-and-bound algorithms. *Concurrency and Computation: Practice and Experience* 28, 18 (2016), 4463–4484. https://doi.org/10.1002/cpe.3771

[21] David J. Johnson and Michael A. Trick (Eds.). 1996. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993.* American Mathematical Society.

[22] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. HPX: A Task Based Programming Model in a Global Address Space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS 2014, Eugene, OR, USA, October 6-10, 2014.* 6:1–6:11. https://doi.org/10.1145/2676870.2676883

[23] Reinhard Lüling, Burkhard Monien, Alexander Reinefeld, and Stefan Tschöke. 1996. Mapping tree-structured combinatorial optimization problems onto parallel computers. In *Solving Combinatorial Optimization Problems in Parallel - Methods and Techniques.* 115–144. https://doi.org/10.1007/BFb0027120

[24] Arnaud Malapert, Jean-Charles Régin, and Mohamed Rezgui. 2016. Embarrassingly Parallel Search in Constraint Programming. *J. Artif. Intell. Res.* 57 (2016), 421–464.

[25] Ciaran McCreesh. 2018. Sequential MCsa1 Maximum Clique Implementation. (2018). https://github.com/ciaranm/sicsa-multicore-challenge-iii/blob/324abebfc3a9144af0bf628077bfb6e5af02444e/c++/voodoo-template-haxx/max_clique.cc Accessed: 04-07-2018.

[26] Ciaran McCreesh and Patrick Prosser. 2013. Multi-threading a state-of-the-art maximum clique algorithm. *Algorithms* 6, 4 (2013), 618–635.

[27] Ciaran McCreesh and Patrick Prosser. 2015. A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs. In *International conference on principles and practice of constraint programming.* Springer, 295–312.

[28] Ciaran McCreesh and Patrick Prosser. 2015. The Shape of the Search Tree for the Maximum Clique Problem and the Implications for Parallel Branch and Bound. *TOPC* 2, 1 (2015), 8:1–8:27. https://doi.org/10.1145/2742359

[29] Tarek Menouer. 2018. Solving combinatorial problems using a parallel framework. *J. Parallel Distrib. Comput.* 112 (2018), 140–153. https://doi.org/10.1016/j.jpdc.2017.05.019

[30] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P Sadayappan, and Chau-Wen Tseng. 2006. UTS: An unbalanced tree search benchmark. In *International Workshop on Languages and Compilers for Parallel Computing.* Springer, 235–250.

[31] Lars Otten and Rina Dechter. 2017. AND/OR Branch-and-Bound on a Computational Grid. *J. Artif. Intell. Res.* 59 (2017), 351–435. https://doi.org/10.1613/jair.5456

[32] Andrea Pietracaprina, Geppino Pucci, Francesco Silvestri, and Fabio Vandin. 2015. Space-efficient parallel algorithms for combinatorial search problems. *J. Parallel Distrib. Comput.* 76 (2015), 58–65. https://doi.org/10.1016/j.jpdc.2014.09.007

[33] Michael Poldner and Herbert Kuchen. 2006. Algorithmic skeletons for branch & bound. In *ICSOFT 2006, First International Conference on Software and Data Technologies, Setúbal, Portugal, September 11-14, 2006.* 291–300. https://doi.org/10.1007/978-3-540-70621-2_17

[34] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. 2013. Embarrassingly Parallel Search. In *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden,*

September 16-20, 2013. Proceedings. 596–610. https://doi.org/10.1007/978-3-642-40627-0_45

[35] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. 2014. Improvement of the Embarrassingly Parallel Search for Data Centers. In *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings.* 622–635. https://doi.org/10.1007/978-3-319-10428-7_45

[36] Pablo San Segundo, Diego Rodríguez-Losada, and Agustín Jiménez. 2011. An exact bit-parallel algorithm for the maximum clique problem. *Computers & OR* 38, 2 (2011), 571–581.

[37] Harry WJM Trienekens and A de Bruin. 1992. *Towards a taxonomy of parallel branch and bound algorithms.* Technical Report. Erasmus School of Economics (ESE).

[38] Gerhard J. Woeginger. 2001. Exact Algorithms for NP-Hard Problems: A Survey. In *Combinatorial Optimization - Eureka, You Shrink!, Papers Dedicated to Jack Edmonds, 5th International Workshop, Aussois, France, March 5-9, 2001, Revised Papers.* 185–208. https://doi.org/10.1007/3-540-36478-1_17

[39] Jingen Xiang, Cong Guo, and Ashraf Aboulnaga. 2013. Scalable maximum clique computation using MapReduce. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013.* 74–85. https://doi.org/10.1109/ICDE.2013.6544815

[40] Yan Xu, Ted K. Ralphs, Laszlo Ladányi, and Matthew J. Saltzman. 2005. *Alps: A Framework for Implementing Parallel Tree Search Algorithms.* Springer US, 319–334. https://doi.org/10.1007/0-387-23529-9_21

A Artifact Appendix

The main artifact, which has been evaluated and assigned the badges **Artifact Available**, **Artifact Functional** and **Results Replicated**, is the *YewPar* framework for parallelising exact combinatorial search applications. Alongside the framework we provide the example search applications used in the paper's evaluation (Section 5). The data and scripts used in the evaluation are available separately at [7].

To facilitate experimentation we have packaged YewPar, and the search applications, as a Docker image. Unfortunately, this makes it difficult to fully replicate the experiments in the paper which require MPI, a 17-node compute cluster and many days of runtime. This document (1) demonstrates parallel execution by showing how to run YewPar on a single machine with 4 cores or more; (2) demonstrates distributed memory execution by showing how to run YewPar under MPI *within* the Docker container.

For more systematic experimentation YewPar is open source (MIT) and available with usage instructions at: https://github.com/BlairArchibald/YewPar

A.1 Downloading and Running the Docker Image

We assume Docker is installed already. First time Docker users may require to run the following commands to ensure that it has the correct permissions:

```
sudo groupadd docker
sudo usermod -aG docker $USER
```

The YewPar Docker image (approx 325MB) can be downloaded from: https://zenodo.org/record/3597463

Once the image is downloaded you should run the following commands which will place you in an interactive bash shell with an environment set up to run the YewPar search applications.

```
docker load < yewpar-artifact-eval.tar.gz
docker run -it yewpar
```

A.2 Docker Image Structure

The YewPar docker image is structured as follows:

File/Directory	Contents
/YewPar	Main working directory (all other files are in here)
src/	YewPar root directory
src/lib	YewPar framework source
src/app	YewPar application sources
c++/	MaxClique comparison code for Table 1
instances/	Instances for the applications, e.g. instances/maxclique/
kclique.sh	Run the kclique example (Fig 4)
kclique-small.sh	Run a shorter kclique example
example_commands.sh	Run a subset of Table 2

A.3 Ease of Use: Creating New Search Applications

Creating Search Trees is the first challenge when implementing a search application. Section 4.1 outlines the Lazy Node Generator abstractions that generate YewPar search trees, and these are relatively simple to use. An example is in /YewPar/src/app/bnb/knapsack/knapsack.hpp, lines 60 to 90. The key function next shows how child nodes are created by adding a candidate to the current solution, updating pos (pointer into the candidate set) and creating a new candidate set (via copy_if). A hasNext function would simply return false if all candidates have been explored, that is if pos > rem.size().

Parallelising Search and Exploring Alternate Parallelisations. A key message of the paper is the ease with which parallel search applications can be built in YewPar, and how easy it is to explore alternate parallelisations. This can be seen in /YewPar/src/app/bnb/maxclique/main.cpp. The code between lines 235 and 343 demonstrates that only the skeleton and some parameters need to be changed to alter the parallel coordination. Crucially the Node Generator GenNode that generates the MaxClique search tree is never changed.

A.4 Recreating Some of the Evaluation Results

A.4.1 Table 1

Table 1 compares the runtimes of a hand-written MaxClique implementation (C++/OpenMP) with the YewPar skeleton-based implementation.

A script runExp.sh to repeat the experiment is available in the c++ sub-directory. Note this script expects a 16 core machine. This can be changed by editing the runSeq and runPar Makefiles.

If you do not wish to use the script you can manually run commands such as

```
maxclique-14 --skeleton seq
  -f /YewPar/instances/maxclique/brock400_4.clq
  --hpx:threads 1

clique-omp /YewPar/instances/maxclique/brock400_4.clq
```

Where maxclique-n can process graphs of up to $64 \times n$ vertices. (To enable optimal use of memory, several maxclique binaries have been compiled, each supporting graphs of different maximal sizes.)

A.4.2 Figure 4

Figure 4 measures the scaling of three parallel YewPar skeletons for a k-clique search instance, namely the budget, stack stealing and depth-bounded skeletons. Parallel executions of the skeletons on four cores, together with a sequential execution, can be obtained as follows:

```
maxclique-14
  -f /YewPar/instances/finitegeo/spreads_H44.clq
  --skeleton seq --decisionBound 33 --hpx:threads 1
```

```
maxclique -14
  -f /YewPar/instances/finitegeo/spreads_H44.clq
  --skeleton budget -b 10000000  --decisionBound 33
  --hpx:threads 4

maxclique -14
  -f /YewPar/instances/finitegeo/spreads_H44.clq
  --skeleton stacksteal --chunked
  --decisionBound  --hpx:threads 4

maxclique -14
  -f /YewPar/instances/finitegeo/spreads_H44.clq
  --skeleton depthbounded -d 2
  --decisionBound 33 --hpx:threads 4
```

To emulate distributed memory execution of the skeletons within the docker container you can run, for example:

```
mpiexec -n 2 -disable-hostname-propagation  maxclique -14
  -f /YewPar/instances/finitegeo/spreads_H44.clq
  --skeleton budget -b 10000000
  --decisionBound 33 --hpx:threads 2
```

And similarly for the DepthBounded/StackStealing skeltons.

Note that -disable-hostname-propagation is required for MPI to work correctly with HPX. These commands are available in the `kclique.sh` script.

To execute on different numbers of cores select a different `--hpx:threads n` parameter. The number of workers (threads performing the search) is n-1 (or 1 if n==1). This implies that hpx:threads 2 may not be faster than hpx:threads 1, but hpx:threads 4 should be.

As Figure 4 illustrates, the k-clique search has long runtimes on small numbers of cores, e.g. around 15 hours on a single fast core. To demonstrate the k-clique application on a smaller example, we provide the `kclique-small.sh` script that runs the k-clique application on the brock400_1 graph (proving the existence of a clique of size 27, and proving the non-existence of cliques of size 28). On a single core, these runs should only take a few minutes each, and around 20 minutes in total.

A.4.3 Table 2

Table 2 reports 9 speedups for each of 6 search benchmarks (54 data points) on 120 workers distributed over a Beowulf cluster. To recreate Table 2 requires a similar sized cluster and many days of running time. To illustrate examples of each of the search applications we provide a list of example commands in the `example_commands.sh` file in the root directory. These commands should give an idea as to what the command line looks like for each application so feel free to vary the parameters. As before you can run MPI locally to experiment with executing YewPar over distributed memory.

If you wish to run the applications on a distributed memory cluster you will need to build YewPar from source; instructions are provided in https://github.com/BlairArchibald/YewPar/blob/master/README.md.