International Journal of Parallel Programming (2023) 51:83–106
https://doi.org/10.1007/s10766-022-00744-3

Generic Exact Combinatorial Search at HPC Scale

Ruairidh MacGregor[1] · **Blair Archibald**[1] · **Phil Trinder**[1]

Received: 9 September 2022 / Accepted: 21 November 2022 / Published online: 7 December 2022
© The Author(s) 2022

Abstract
Exact combinatorial search is essential to a wide range of important applications, and there are many large problems that need to be solved quickly. Searches are extremely challenging to parallelise due to a combination of factors, e.g. searches are non-deterministic, dynamic pruning changes the workload, and search tasks have very different runtimes. YewPar is a C++/HPX framework that generalises parallel search by providing a range of sophisticated search skeletons.This paper demonstrates *generic* high performance combinatorial search, i.e. that a variety of exact combinatorial searches can be easily parallelised for HPC using YewPar. We present a new mechanism for profiling key aspects of YewPar parallel combinatorial search, and demonstrate its value. We exhibit, for the first time, generic exact combinatorial searches at HPC scale. We baseline YewPar against state-of-the-art sequential C++ and C++/OpenMP implementations. We demonstrate that deploying YewPar on an HPC system can dramatically reduce the runtime of large problems, e.g. from days to just 100s. The maximum relative speedups we achieve for an enumeration search are near-linear up to 195(6825) compute-nodes(workers), super-linear for an optimisation search on up to 128(4480) (pruning reduces the workload), and sub-linear for decision searches on up to 64(2240) compute-nodes(workers).

Keywords Combinatorial search · Parallel skeletons · Constraint programming · High performance computing

1 Introduction

Exact combinatorial search is essential to a wide range of important applications including constraint programming, graph matching, and planning. A classic example would be to allocate parcels to vans, and to plan delivery routes for the vans.

✉ Blair Archibald
blair.archibald@glasgow.ac.uk

Phil Trinder
phil.trinder@glasgow.ac.uk

[1] School of Computing Science, University of Glasgow, Glasgow, UK

Combinatorial problems are solved by systematically exploring a space of (partial) solutions, 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 exhaustively explores the search space and delivers provably optimal answers. Conceptually exact combinatorial search proceeds by generating and traversing a (huge) tree representing alternative options. Backtracking branch and bound search is a well known example. Although searches can be time consuming, combining parallelism, on-demand tree generation, search heuristics, and pruning can effectively reduce execution time. Thus a search can be completed to meet time constraints, for example to load and dispatch a fleet of delivery vans each morning.

Exact combinatorial search is very different to classical HPC applications. The problems are NP-hard rather than polynomial. There is almost no use of vectors or matrices: the primary data structure is a huge, dynamically generated, irregular search tree. Almost all values are discrete, e.g. integers or booleans. In place of nested loops there are elaborate recursive control structures. The parallelism is highly *irregular*, that is the number and runtimes of tasks are determined by the search instance, and vary hugely, i.e. by several orders of magnitude [25]. Only a small number of specific exact combinatorial searches have been hand-crafted for HPC scale (around 1000 cores), e.g. [4, 9].

There are three main *search types*: *enumeration*, which searches for all solutions matching some property, e.g. all maximal cliques in a graph[1]; *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. There are standard *instances* for many important *search applications*, e.g. the DIMACS instances [10].

YewPar is a C++ parallel search framework [2, 5]. YewPar generalises search by abstracting search tree generation, and by providing algorithmic skeletons that support the three search types. The skeletons use sophisticated search coordinations that control the parallel search including when new tasks are generated. These are inspired by the literature, and are currently: Sequential, Depth-Bounded, Stack-Stealing and Budget. It also provides low-level search specific schedulers and utilities to deal with the irregularity of search and knowledge exchange between workers. YewPar uses the HPX library for distributed task-parallelism [11], allowing search on multi-cores, clusters, HPC systems etc.

This paper makes the following contributions.

We present *a new mechanism for profiling key aspects of generic parallel combinatorial search* in YewPar. While the extreme irregularity of parallel combinatorial search is well known, it is seldom measured, and then only for specific search applications, e.g. [25]. Key novelties are (1) to provide profiles that quantify the irregularity of various search applications in the generic YewPar framework, e.g. to report

[1] A clique in a graph is a set of vertices C such that all vertices in C are pairwise adjacent. Maximal cliques cannot be extended by including one more adjacent vertex and Maximum cliques are the largest cliques in the graph.

median task runtime, maximum task runtime, etc. (2) to make irregularity characteristics visible to the developer to aid performance tuning (Sect. 5).

We demonstrate for the first time generic exact combinatorial searches at HPC scale. Generic YewPar searches have previously only been demonstrated on relatively small-scale clusters: 17 compute nodes and 270 cores [5]. We use a combination of techniques like repeated measurements and using multiple search instances to address the challenges of parallel search, e.g. non-determinism, non-fixed workloads, irregular parallelism, and the nature of NP-hard problems. Baselining against state-of-the-art sequential C++ and C++/OpenMP implementations on 9 standard (DIMACS) search instances shows that the generality of YewPar incurs a mean sequential slowdown of 9.6%, and a mean parallel slowdown of 27.6% on a single 18-core compute node. Guided by the profiling we effectively parallelise seven standard instances of the three searches, and systematically measure runtime and relative speedups at scale. We demonstrate that deploying YewPar on an HPC system can dramatically reduce the runtime of large problems, e.g. from days to just 100 s. The maximum relative speedups we achieve for the Numerical Semigroups enumeration search are near-linear up to 192(6825) compute-nodes(workers), super-linear for a Maximum Clique optimisation search on up to 128(4480) (pruning reduces the workload), and sub-linear for a k-clique decision search on up to 64(2240) compute-nodes(workers) (Sect. 6).

2 Background

2.1 The Challenges of Exact Combinatorial Search at HPC Scale

An exact combinatorial search generates and explores a massive tree of possible solutions. The search trees are generated on-demand to limit memory requirements, and provide ample opportunities to exploit parallelism. Space-splitting approaches explore subtrees in parallel, speculatively for optimisation and decision searches. Portfolio approaches run multiple searches, with varying search orders/heuristics, and share knowledge between searches.

We focus on space-splitting approaches here. As the search trees are so large (potentially exponential in the input size) we can easily generate millions of parallel tasks: far more than commodity hardware can handle, but enough to keep for modern HPC clusters busy.

However there are many aspects of search that make effective parallelisation extremely challenging in practice. Searches differ from standard parallel workloads due to their heavy use of symbolic/integer data and methods as opposed to floating point, and widespread use of conditionals meaning that neither vectorisation nor GPUs are beneficial.

Here we outline some specific challenges raised by combinatorial search at HPC scale (100s of compute nodes, more than 5000 cores), and show that despite these issues *exact combinatorial search is a fruitful HPC domain*.

Task Irregularity Parallel search tasks are highly *irregular* that is (1) some tasks have short runtimes, e.g. several milliseconds, while others take orders of magnitude longer, *e.g.* many minutes (2) tasks are generated dynamically, and the number of tasks varies depending on the search instance, e.g. as determined by the number of children of a search tree nodes. Many classical HPC workloads, e.g. computing over a homogeneous mesh, are regular with tasks having similar runtimes, and the number of tasks can be statically predicted.

Search tasks explore subtrees, and the sizes of these often varies by orders of magnitude. The problem is compounded as the shape of the search tree changes at runtime as new knowledge, like improved bounds, is learned. In practice improved knowledge can make a significant proportion of existing tasks redundant. We give a detailed analysis of search task irregularity in Sect. 5.

Speculation Tree searches are commonly parallelised by *speculatively* exploring subtrees in parallel. Most searches are compute, rather than memory or communication bound, and to achieve substantial speedups large amounts of speculation are used.

Speculatively exploring subtrees earlier than a sequential algorithm can dramatically improve performance: some parallel task may find a solution or strong bound long before the sequential algorithm would. So superlinear speedups are not uncommon. Conversely, speculation may result in the parallel search performing far more work than the sequential search, and may lead to slowdowns as more cores are used. These superlinear speedups and slowdowns due to increased workload are known as *performance anomalies*.

Preserving Heuristics State of the art algorithms make essential use of *search heuristics* that minimise search tree size by focusing on promising areas first. Parallel searches must preserve the heuristic search ordering as far as possible, so standard random work stealing isn't appropriate. Rather, scheduling is often carefully designed to preserve search heuristics [3, 21].

Global Knowledge Exchange Search tasks discover information that must be shared with other search tasks, e.g. a better bound in a branch and bound search. Sharing global state must be managed carefully at HPC scale to avoid excessive communication and synchronisation overheads. It is likely that some search algorithms that share significant amounts of data, such as clause learning SAT solvers, will struggle to scale onto HPC. However many searches only share small amounts of data globally. Moreover as the global data primarily provides opportunities to optimise (e.g. prune the search tree), there is no strong synchronisation constraint: remote search tasks are neither stalled, nor producing incorrect data prior to receiving the new knowledge.

Programming Challenges State of the art search implementations are intricate [7, 19, 22], and it is unusual to see large scale parallelisations. Moreover few of the parallel search implementations fully utilise modern architectures, e.g. provide two level

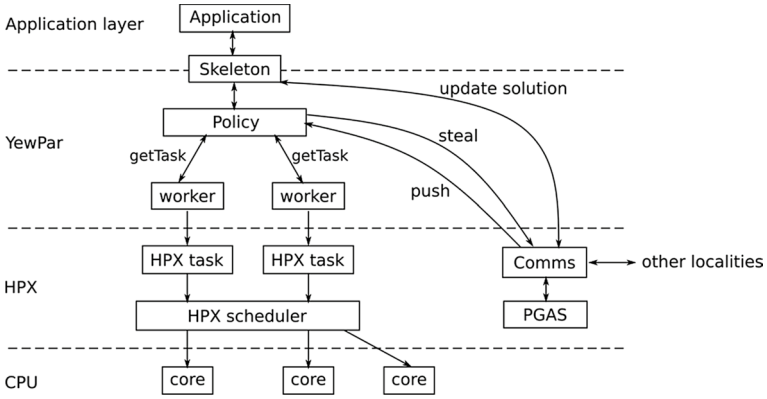

Fig. 1 YewPar system stack from [3]

(thread + process) parallelism. Even when parallel searches share common paradigms, they are typically individually parallelised and there is little code reuse. One approach to minimise development effort is to parallelise existing sequential solvers [23]. Alternatively high level frameworks provide developers generic libraries to compose searches [1, 8, 28].

This paper focuses on YewPar, the latest such framework, and designed to scale to HPC.

2.2 YewPar

YewPar[2] is carefully designed to manage the challenges of parallel search. Lazy Node Generators produce the search trees, and skeletons are provided to efficiently search them in parallel. To support distributed memory parallelism, YewPar builds on the HPX [11] task parallelism library and runtime system. HPX is routinely deployed on HPC and Cloud systems, and YewPar can readily exploit this portability at scale. Complete descriptions of the YewPar design and implementation are available in [2, 3], and we have recreated the architecture diagram from [3] (Fig. 1) for ease of reading; it has the following key components.

Lazy Node Generators Search trees are too large to realise in memory, and searches proceed depth-first lazily generating only the subtrees being searched. The Lazy Node Generator for a specific search application is a data structure that takes a parent search tree node and enumerates its children in traversal (i.e. heuristic) order. While node generators create the children of a node, *how* and *when* the search tree is constructed is determined by the skeletons.

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

Search Coordinations To minimise search time it is critical to choose heuristically a *good* node to search next. We follow prior work e.g. [27], 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. In addition to *sequential* depth first search, YewPar currently provides three parallel search coordinations inspired by approaches in the literature [2]. *Depth-Bounded* search converts all nodes below a cut-off depth d_{cutoff} into tasks in a similar, but more dynamic, style to [30]. *Stack-Stealing* search dynamically generates work by splitting the search tree on receipt of a work-stealing request. In *Budget* search workers search subtrees until either the task completes or the task has backtracked as many times as specified in a user-defined *budget*. At that point new search tasks are spawned for the top-level nodes of the current sub-tree.

YewPar Skeletons compose a search coordination with one of the three search types, for example `BudgetDecision`, or `DepthBoundedOptimisation`. There are currently four search coordinations, and hence 12 (3×4) skeletons. The skeletons are implemented for runtime efficiency, e.g. C++ templates are used to specialise the skeletons at compile-time. The skeleton APIs expose parameters like depth *cutoff* or backtracking *budget* that control the parallel search.

Search Specific Schedulers YewPar layers the search coordination methods as custom schedulers on top of the existing HPX scheduler.[3] That is, the HPX scheduler manages several lightweight YewPar scheduler threads that perform the search. In addition to search worker threads, each compute-node has a *manager* HPX thread that handles aspects like messages and termination. The schedulers seek to preserve search order heuristics, e.g. by using a bespoke order-preserving workpool [2, 3].

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 compute-nodes 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.

3 Ease of Use

YewPar is designed to be easy to use by combinatorial searchers who lack expertise in parallel programming. For example, although each of the three searches we describe in Sect. 4 uses a published state-of-the-art algorithm, they require only around 500 lines of code.[4] A recent example of Flowshop search required on the magnitude of 50 lines of code to move from direct implementation to YewPar parallel [14]. Most of the search application is parameterised generic code. Notable

[3] Using the default thread executor for HPX 1.2.1.

[4] https://github.com/BlairArchibald/YewPar/apps.

exceptions are the node generator that produces the search tree, and the bounding function (pruning predicate), that are search-specific and must be specified.

Crucially a YewPar user only composes extremely high level parallel constructs: they select and parameterise a search skeleton. For example the search specified in the first 6 lines of Listing 1 returns the maximal node in some space. The user specifies the search coordination, in this case `StackStealing`, provides the application-specific Lazy Node Generator `Gen` to generate the search tree, and chooses the type of search, here `Optimisation`. The listing also illustrates how YewPar implements pruning. 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.

Providing such high level abstractions of the parallel search makes it easy to experiment with alternate parallel searches and search parameters. As an example, the last 8 lines of Listing 1 specify a parallel budget version of the maximal node search, where each search task has a budget of 50,000 backtracks. In contrast, hand-written parallel search applications like [7, 21] usually add parallelism constructs directly to the main search algorithm, obfuscating the algorithm and making it very difficult to experiment with alternate parallelisations without major refactoring.

A more complete description of how to specify parallel searches in YewPar are provided in [5, 15]. The former provides a docker image artefact containing 6 example search implementations.

Listing 1: Two YewPar Parallel Search Skeletons

```
1  Node maximal_solution_ws =
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);
7
8  Params params;
9  searchParameters.backtrackBudget = 50000;
10 Node maximal_solution_budget =
11   YewPar::Skeletons::Budget <       // search coord
12     Gen,                            // lazy node gen
13     Optimisation,                   // search type
14     BoundFunction<upperBound>       // bound for pruning
15   >::search(space, root, params);
```

The search coordinations provided by YewPar are significantly more advanced than those commonly used in hand-written parallel searches. For example, in Sect. 6 we baseline our YewPar implementation against a simple OpenMP version that adds a pragma to the main search loop. The programming effort required to produce the OpenMP and YewPar versions is very similar, but the OpenMP implementation is limited to shared memory, and to a simple depth 1 bounded search. Implementing more complex search coordinations, e.g. a heuristic-preserving depth 2 bounded search is far more intricate. Search coordinations such as StackStealing

require access to the scheduler. Moreover conventional parallel schedulers often disrupt search heuristics [21], and we show how the OpenMP scheduler disrupts the baselining search instances in Sect. 6. This highlights the need for custom schedulers as in YewPar.

3.1 Related Frameworks

While the vast majority of parallel searches are handwritten, there are some generic frameworks such as Bob++ [8], MaLLBa [1], TASKWORK [12], and Muesli [28]. These provide a similar level of programming abstraction to YewPar, but are mainly designed for branch and bound optimisation and, unlike YewPar, do not currently support decision or enumeration searches and tend to only support one search coordination. A wider discussion of existing frameworks an their relationship to YewPar is in [2, Chapter 2].

4 Generic Exact Combinatorial Search

Due to the challenges of engineering performant parallel implementations of exact combinatorial search, only a small number of specific exact combinatorial searches have been hand-crafted for HPC scale, e.g. [4, 9]. YewPar is designed to minimise the effort required to engineer performant searches by providing a library of re-usable skeletons and search coordinations. While this genericity has previously been demonstrated by parallelising seven searches at cluster scale (100s of cores) [5], it has never been demonstrated at HPC scale (1000s of cores).

We further demonstrate the ease of construction (Sect. 3) by exhibiting parallel searches covering the three search types.

Numerical Semigroups The first application comes from group theory, and tackles the problem of counting the number of numerical semigroups of a particular genus, which is useful for areas such as algebraic geometry [6]. For mathematical searches such as this, *exactness* is essential: an approximate answer has no value.

A numerical semigroup can be defined as "Let $\mathbb{N}_0$ be the set of non-negative integers. A numerical semigroup is a subset Λ of $\mathbb{N}_0$ which contains 0, is closed under addition and has finite complement, $N_0 \setminus \Lambda$. The elements in $N_0 \setminus \Lambda$ are the gaps of Λ, and the number $g = g(\Lambda)$ of gaps is the genus of Λ" [7]. A numerical semigroup can be viewed as taking a finite set of non-negative integers X such that $\mathbb{N}_0 \setminus X$ remains closed under addition.

A numerical semigroups search enumerates the number of semigroups with genus g, e.g. of genus 46. The YewPar node generator uses Hivert's algorithm [7] and exploits the relation between subsequent numerical semigroups to generate search tree nodes. So the search counts the number of search tree nodes at the specified depth, e.g. at depth 46.

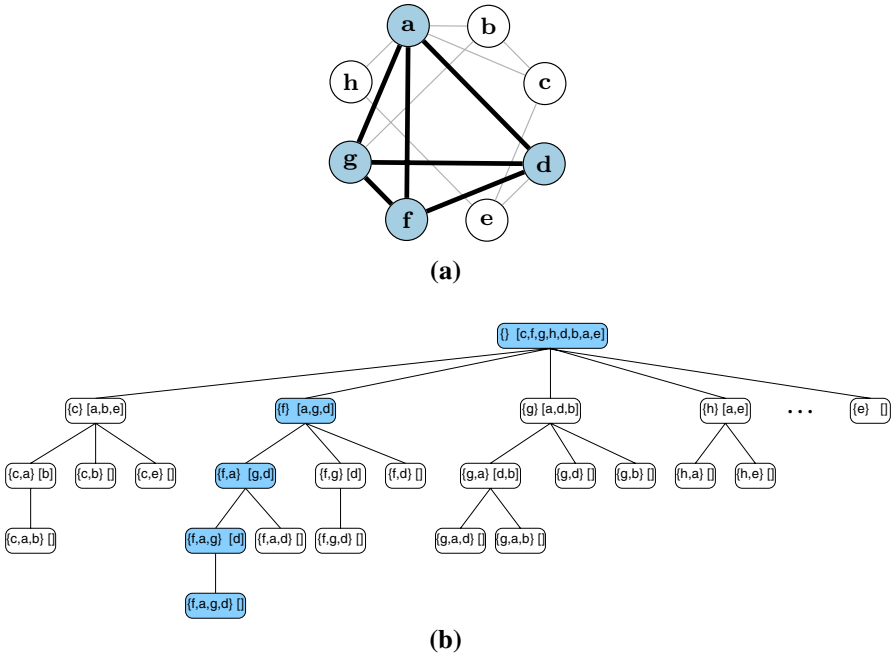

Fig. 2 A maximum clique instance. **a** Input graph with clique $\{a, d, f, g\}$; **b** corresponding search tree. Each tree node displays the current clique and a list of candidate vertices (in heuristic order) to extend that clique

Maximum Clique The maximum clique optimisation problem seeks to find the largest clique in graph G. A clique $C \subseteq V$ such that $\forall u, v \in C(\{u, v\} \in E)$. The node generator implementation is based on the MCSa1 algorithm [29] that exploits graph colouring for bounding/heuristic ordering (Fig. 2).

Maximum clique arises in areas such as computational biology, information retrieval, economics and signal transmission theory [26]. Search instances are drawn from the standard DIMACS challenge instances [10].

k-clique with Finite Geometry. Some applications require a *specifically sized* clique. The k-clique decision search determines whether there is a clique C in a graph G of size k, i.e. $|C| = k$. We apply k-clique to a problem in finite geometry—determining if a *spread* in geometries of the Hermitian variety $H(4, q^2)$ exists [13]. A *spread* is a set of lines L such that every point is incident with exactly one element of L. For $H(4, q^2)$, a spread (if it exists) will have size $q^5 + 1$. Intuitively, a spread forms a partition of the points.

Symmetries in the state space up to depth 3 are broken by pre-processing with GAP [31]. So the clique being searched for is of size $q^5 + 1 - 3$. We consider geometries of the form $H(4, 3^2)$, so $k = 3^5 + 1 - 3 = 241$, and the Maximum Clique node generator is used to generate the search tree.

5 Profiling Irregularity in Exact Combinatorial Search

Parallel exact combinatorial search produces extremely irregular parallelism (Sect. 2.1). Although the irregularity of a small number of specific parallel searches has previously been investigated, e.g. [25, Fig. 4], detailed analysis of irregularity is uncommon, and the extent of the irregularity in most searches is unknown.

To provide detailed information on search task irregularity we have added a small data store on each YewPar compute-node that records key aspects of the parallel search: task runtimes, number of backtracks, number of search tree nodes visited, and total number of tasks spawned. To allow the *shape* of the tree to be investigated, the data is indexed by the tree depth where the task was *spawned*.

5.1 Experimental Setup

Measurements are made on the following platforms. A modified version of YewPar that includes the new profiling techniques[5] and OpenMP are compiled with gcc 8.2.0 and HPX 1.2.1. *Cirrus* is an HPC cluster comprising 228 compute-nodes, each having twin 18-core Intel Xeon (Broadwell) CPUs (2.1Ghz), 256 GB of RAM and running Red Hat Enterprise Linux Version 8.1. The *GPG Beowulf Cluster* comprises 17 compute-nodes, each having dual 8-core Intel Xeon E5-2640v2 CPUs (2Ghz), 64GB of RAM and running Ubuntu 18.04.2 LTS.

5.2 Search Task Runtimes

Search Task Runtime (STR) profiles shows the distribution of task runtimes for tasks spawned at each depth in the search tree. Much of the variance arises from the structure of the search tree. STR allows us to visualise the distribution of task runtimes throughout a search and provides information about the shape of the search tree for a given search instance.

We illustrate the range and distribution of search task runtimes using a violin plot for the tasks spawned at each search tree depth, excluding the time for spawned tasks to complete. The shape of each violin plot represents the distribution of runtimes, e.g. wide sections correspond to frequent runtimes. The white cross represents the median value, and the black rectangle the interquartile range.

As a basis for comparison we record the STR for a relatively *regular* parallel tree search. This synthetic search enumerates the tree nodes down to depth 30 in a balanced binary tree (so all subtrees are the same size), creating tasks down to depth 8. Fig. 3 shows the search task runtimes at different depths. Task runtimes at depths 0 to 7 are uniformly small. The tasks at depth 8 do most of the enumeration, and their runtimes have a compact distribution with median 16.4ms and an interquartile range of 16.3ms to 16.6ms.

[5] https://github.com/ruairidhm98/YewPar.

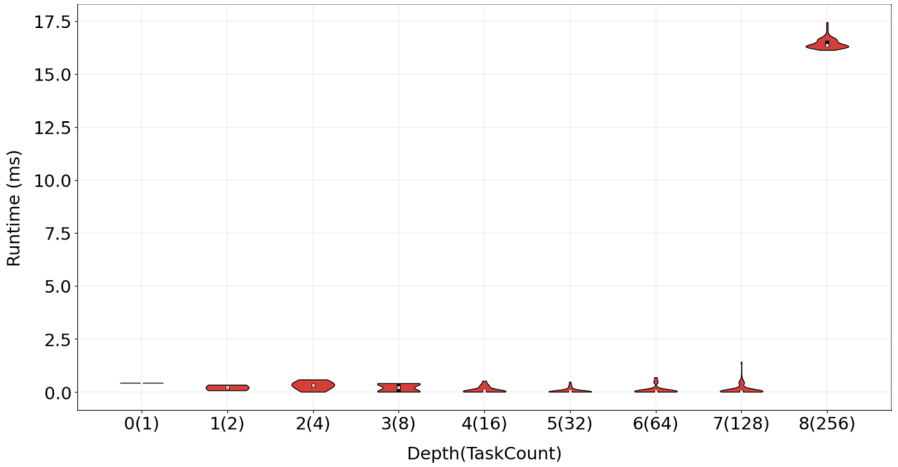

Fig. 3 Regular search task runtimes: task runtime distributions for a balanced binary tree search to depth 30 using the Depthbounded skeleton, $d_{cutoff} = 8$ (1 GPG cluster compute-node)

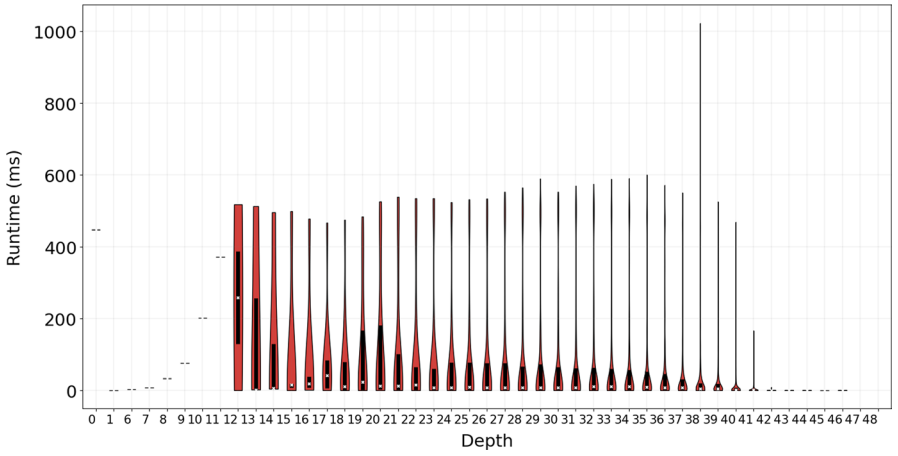

Fig. 4 Search task runtime distributions for a Numerical Semigroups genus 48 search, Budget skeleton with $b = 10^7$ backtracks. Depths 2-5 spawn no tasks and are omitted (1 GPG cluster compute-node, 15 workers)

In contrast, Figs. 4, 5 and 6 show that the STR distributions for typical combinatorial searches are very different. Figure 4 is for a Numerical Semigroups genus 48 search, using the Budget skeleton with a budget of $b = 10^7$ backtracks. The distribution of task runtimes is plotted for each depth in the search tree down to depth 48. Although task sizes increase steadily at depths 1–7, there are few tasks and little variance. This quantifies a known result that the Numerical Semigroups search tree is narrow at low depths [7]. Between depths 12 and 41 there is massive variability in search task runtimes. For example at depth 16 the median task

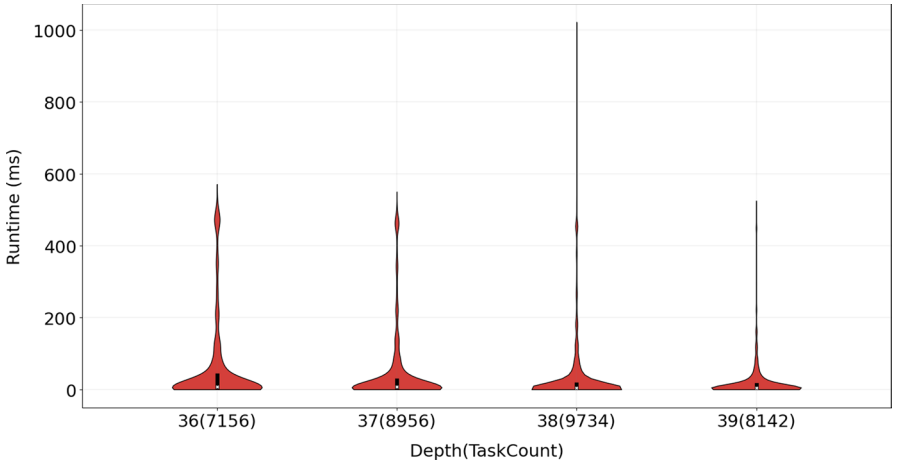

Fig. 5 Search task runtime distributions for depths 36 to 39 of a Numerical Semigroups genus 48 search, Budget skeleton with $b = 10^7$ backtracks (1 GPG cluster compute-node, 15 workers)

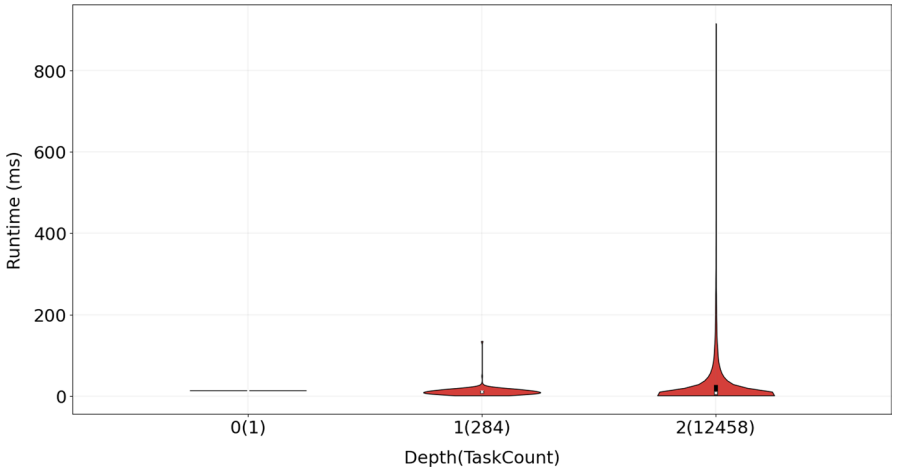

Fig. 6 Search task runtime distributions for a brock400_1 Maximum Clique search, depth 2 Depth-bounded skeleton (1 GPG cluster compute-node, 15 workers)

runtime is 19ms, while the interquartile range is 31.5ms, and the maximum task runtime is 499ms.

Figure 5 provides more details of the Numerical Semigroups search task runtime distributions at depths 36–39. Not only is the massive variability in runtimes clear, but it is far more apparent that the distributions at these levels, as at other levels, are multi-modal. For example the distributions at levels 36 and 37 both have four clear modes.

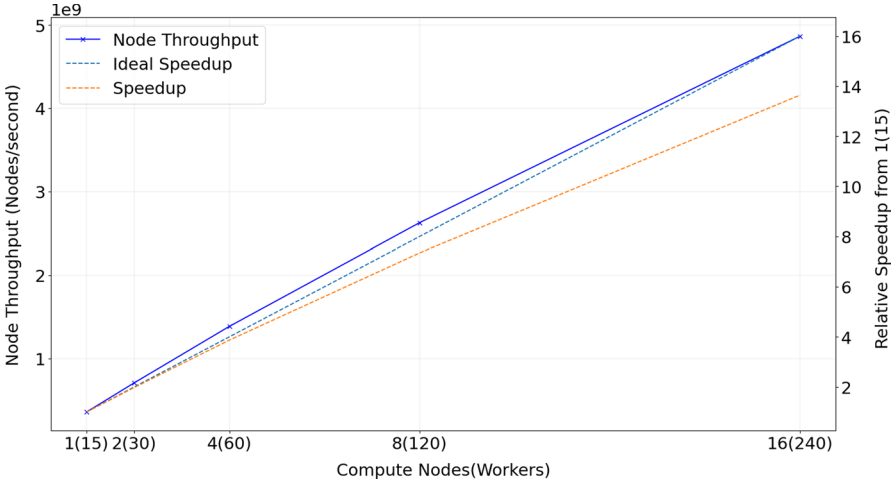

Fig. 7 Node throughput and relative speedup on GPG cluster compute nodes(workers) for a numerical semigroups genus 49 search, Budget skeleton with $b = 10^7$ backtracks

Figure 6 shows the search task runtime distributions for a Maximum Clique search instance using the Depthbounded skeleton at depth 2 ($d_{cutoff} = 2$). This optimisation search instance is brock400_1 from the DIMACS benchmark suite [10]. Tasks are spawned at only three depths and the vast majority of search tasks, 12,458 out of 12,753, are generated at depth 2. Depths 1 and 2 both exhibit massive variability in search task runtimes. Most tasks have short runtimes (less than 70 ms), but a small number have much longer runtimes (over 900ms). At depth 2 the median task runtime is 8ms, while the interquartile range is 26ms, and the maximum task runtime is 916ms.

5.3 Search Tree Node Throughput

Search tree node throughput profiles show the number of nodes visited by some search task per unit time. It is commonly used as a measure of search speed and, indirectly, the size of the workload [17]. As the number of cores grows, increasing node throughput illuminates how parallelism may reduce search runtime.

YewPar has been extended to record node throughput by counting each node visited during the search using a depth-indexed vector of atomic counters in the profiling data store. To minimise the number of atomic operations each search worker maintains a local counter and only updates the atomic counter in the vector on termination.

Figure 7 shows the node throughput and relative speedup for a Numerical Semigroups genus 48 search. This enumeration search again uses the Budget skeleton with a budget b of 10^7 backtracks, and is measured on between 1 and 16 compute nodes of the GPG cluster. This graph, and the other graphs in this section, report throughput as the mean number of nodes visited divided by the median runtime over 5 executions.

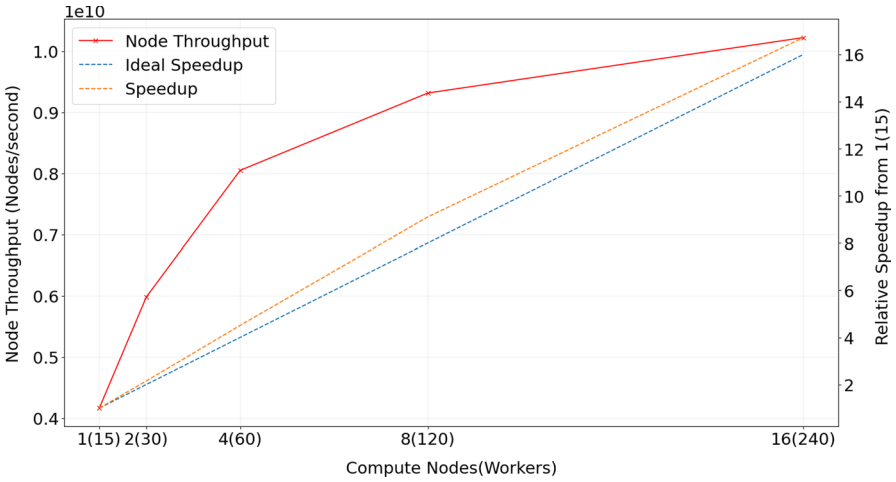

Fig. 8 Node throughput and relative speedup on GPG Cluster Compute Nodes(workers) for a brock800_2 Maximum Clique search, depth 2 Depthbounded skeleton

Node throughput increases linearly as the number of compute nodes and cores increases, and is closely correlated with the speedup. This is as expected for an enumeration search that has a fixed workload, i.e. must visit exactly the same number of search tree nodes in every execution. So for enumeration searches a higher node throughput directly correlates with speedup.

Figure 8 shows the node throughput and speedup for a Maximum Clique search instance using the Depthbounded skeleton at depth 2 ($d_{cutoff} = 2$). This optimisation search instance is DIMACS brock800_2 and is measured on 1–16 compute nodes of the GPG cluster.

Both node throughput and speedup increase superlinearly as the number of compute nodes and cores increases, but are not strongly correlated. Speculative search tasks account for the increase in node throughput as the number of cores increases. We believe that the rate of increase of node throughput falls at high core counts because the speculative threads prune much of the search tree. It is, however, not easy to measure how much of the tree is pruned as the pruned subtrees are never generated. The reduced node throughput is again as expected for an optimisation search where pruning reduces the workload.

5.4 Using Profiles to Select and Parameterise Skeletons

Although far easier that hand coding a parallel search from scratch, a developer using YewPar must both select an appropriate skeleton, and for most skeletons provide appropriate parameters. That is, no one skeleton works best for all search applications [5] and selecting inappropriate parameters results in poor performance [2].

The search task runtime and node throughput profiles above assist the developer to select a skeleton and to parameterise it. They do so by accurately quantifying and

Table 1 Profiling overheads for 4 search applications and 2 search types (1 GPG Cluster Compute-Node, 15 workers)

Instance	STR % overhead	Throughput % overhead
Numerical Semigroups $g = 44$ (Budget)	2	3
Numerical Semigroups $g = 46$ (Budget)	1	3
Maximum Clique brock400_1.clq (depthbounded)	7	9
Maximum Clique brock400_4.clq (depthbounded)	20	10
Geometric mean % overhead	4	5

visualising search task runtime distributions, and hence the search tree shape. We illustrate by example, and use similar techniques to parallelise the searches at HPC scale in the next section.

For the MaxClique instance the STR profile in Fig. 6 shows that the distribution of task runtimes for this instance is not huge, and so it probably does not need highly dynamic parallelism such as Stackstealing or Backtracking. If we select a Depthbounded skeleton, a depth parameter must be specified. We see that most task runtimes are very small (10s of ms), and at depths 0 and 1 there are too few tasks to occupy the 15 workers on a GPG cluster node, i.e. less than 300 tasks. However at depth 2 there are 12K tasks which is likely sufficient.

For the Numerical Semigroups instance the STR profile in Fig. 4 reveals that tasks of widely varying runtimes are generated at many levels of the search tree (levels 12–40). Hence a skeleton with dynamic parallelism is appropriate. As an enumeration search Numerical Semigroups does not need to deal with pruning, and so may not need the fully dynamic Stackstealing skeleton, and the Budget skeleton is likely most appropriate. Here a node throughput profile like Fig. 7 helps to select the backtrack budget. We see that 30 workers are processing approximately 10^9 nodes/s, so each worker processes approximately 10^8 nodes/s. Selecting a budget of 10^7 means that each worker backtracks around 10 times each second: an appealing heuristic value.

5.5 Overheads of Profiling

To understand how profiling impacts search performance we show the percentage overhead for a collection of searches in Table 1. Overhead is calculated as the percentage difference in search runtime with and without profiling. The searches use two different skeletons and feature both enumeration (Numerical Semigroups) and optimisation (Maximum Clique) instances.

The overheads are modest adding just 5% on average. Hence profiling can be used without dramatically effecting runtimes. Measuring node throughput adds more overhead than search runtimes as they must count *every* node that is visited, while STR only requires the time to be recorded at the start and end of the task run. The overheads do vary between searches and brock400_4 has high overheads: up to

20% for STR profiling. It may be that this is showing both profiling overheads and a disrupted search order.

We expect YewPar users to only enable profiling when designing and tuning a search. Hence the profiling implementation is designed to be turned off completely for maximising performance, for example for the HPC executions in the next section.

6 Exact Combinatorial Search at HPC Scale

Measuring parallel searches is challenging primarily due to the non-determinism caused by pruning, random work-stealing, and finding alternate valid solutions. These can lead to performance anomalies (Sect. 2.1) that manifest as dramatic slowdowns or superlinear speedups. We control for this by investigating multiple instances of multiple search applications and selecting the median of 5 executions. The experimental setup is as in Sect. 5.

6.1 Sequential and Single Compute-Node Baselines

YewPar's generality incurs some overheads compared to search specific implementations as it decouples search tree generation and traversal. For example, Lazy Node Generators copy search tree nodes (in case they are stolen) instead of updating in-place. We evaluate these overheads on Maximum Clique as a competitive sequential implementation is available [19, 20].

Sequential Baseline The first 4 columns of Table 2 show the mean sequential runtimes (over 5 executions) of the 9 DIMACS clique instances [10] that take between 100 s and 1 h to run sequentially on Cirrus. The results show a limited cost of generality, i.e. a maximum slowdown of 22.44%, a minimum slowdown of − 2.56%, and geometric mean slowdown of 9.7%. We attribute the small runtime reductions compared with C++ for 2 search instances to optimisations arising from different C++ and C++/HPX compilation schemes.

Parallel Baseline Parallel execution adds additional overheads, e.g. the YewPar skeletons are parametric rather than specialised, and the distributed memory execution framework is relatively heavyweight on a single compute node. 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 [20], and the OpenMP implementation uses a single `task` pragma to construct a set of tasks for each node at depth 1, closely analogous to the DepthBounded skeleton in the YewPar implementation. The penultimate paragraph of Sect. 3 compares these OpenMP and YewPar MaxClique implementations.

Table 2 Comparing YewPar runtimes (s) and slowdown (%) with hand-written maximum clique implementations: sequential and depth-bounded OpenMP implementations (Cirrus)

Search Instance	Sequential C++	Sequential YewPar	Slowdown %	OpenMP (1 Worker) C++	OpenMP (18 Workers) C++	Depth-bounded YewPar	Slowdown %
brock400_1	252.60	301.08	19.19	295.63	18.89	23.77	25.89
brock400_2	183.38	218.32	19.05	217.96	7.80	10.04	28.74
brock400_3	145.44	174.15	19.74	230.71	4.26	5.66	32.86
brock800_1	3790.72	3868.66	2.06	5914.71	195.48	240.86	23.22
brock800_2	3808.82	3764.54	− 1.16	6820.68	232.26	256.52	10.44
brock800_3	3512.92	3511.75	− 0.03	4697.50	183.88	207.45	12.82
brock800_4	1316.16	1282.45	− 2.56	3560.01	86.87	102.15	17.59
C250.9	1725.04	2112.06	22.44	1756.48	115.72	179.57	55.18
p_hat700-3	1139.00	1278.38	12.24	1020.68	77.52	115.49	48.98
Geo. Mean			9.66				27.63

Comparing the second and fifth columns of Table 2 reveals significant slowdowns for OpenMP using a single worker. These arise as the OpenMP scheduler does not preserve the search heuristic, as revealed by the task schedule. That is OpenMP provides no guarantee that the search tasks are executed in the order they are spawned, and this illustrates a common issue when using off-the-shelf parallelism frameworks for search [21]. The effect is smaller in the parallel version as the likelihood that at least one worker follows the heuristic increases.

Columns 5–7 of Table 2 compare the runtimes of the YewPar and OpenMP versions for the DIMACS search instances with 18 search workers on a single Cirrus compute node. We measure the searches on 18 workers/cores rather than on all 36 physical cores available on a Cirrus compute node as experimentation reveals that OpenMP performance reduces above 18 cores. We attribute this to starvation as the depth 1 spawning creates too few tasks to utilise all of the cores. A depth-2 backtracking search would generate far more work, but implementing such a search in OpenMP that is correct, and exactly emulates the YewPar DepthBounded search, is far from trivial especially when trying to maintain search heuristics. The geometric mean slowdown increases to 27.6%, with a maximum slowdown of 55%. We believe these slowdowns are largely caused by increased copying used by YewPar (as any node might become a task so should be self contained) but further investigation is needed.

The sequential and single compute node overheads of YewPar are lower on the GPG Cluster. For the same Maximum Clique codes on a slightly larger set of DIMACS instances the mean sequential slowdown is 8.7%, and the slowdown on a single 16-core compute node is 16.6% [5].

We conclude that for these search instances the parallel overheads of YewPar remain moderate, while facilitating the execution of multiple search applications on multiple platforms: multicores, clusters, or HPC systems.

6.2 HPC Performance Measurements

As exact combinatorial search problems are NP hard the workloads generated by instances vary greatly, often by orders of magnitude. Hence it is not possible to double problem size to measure weak scaling. Hence we report *strong scaling*, and speedups are relative to execution on a small number of Cirrus compute nodes.

We measure the scaling of searches covering the three search types and using different YewPar skeletons. Specifically we measure the searches outlined in Sect. 4, and the search instances measured are as follows. Numerical Semigroups is an Enumeration search at genus 61, using the Budget skeleton with a budget of 10^7 backtracks. Maximum Clique is an Optimisation search for the DIMACS p_hat1000-3 instance, and uses the Depthbounded skeleton with a depth cutoff of 2 for all measurements other than on 4480 workers where we use a cutoff of 3 to minimise starvation. k-clique is the finite geometry Decision search with k=241 and uses the Depthbounded skeleton with a depth cutoff of 3. This relatively high cutoff is selected to generate many search tasks, as searching for a specific clique size induces huge

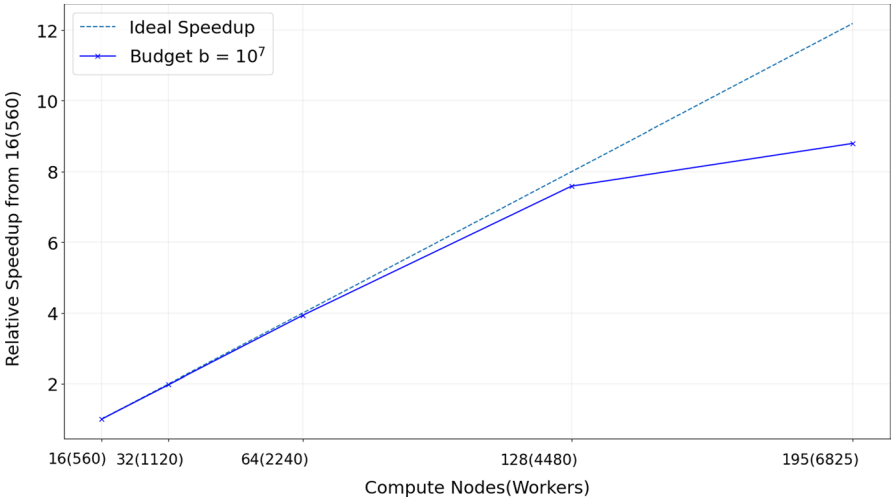

Fig. 9 Numerical semigroups genus 61 search, budget 10^7 backtracks; speedup relative to 16(560) Cirrus compute nodes(workers)

amounts of pruning. While successful decision searches terminate early, here we measure unsuccessful searches that must explore the entire space.

Runtimes Numerical Semigroups runtimes fall from 2649s on 16(560) Cirrus Compute Nodes(workers) to 302s on 195(6825) Compute Nodes(workers). Recall that each YewPar worker is associated with a core. Maximum Clique runtimes fall from 2255s on 8(280) to 102s on 128(4480) compute nodes(workers). For the 178_435 k-clique search, the most significant runtime decrease is from 4563s on 1(35) to 127s on 64(2240) compute nodes(workers). The other k-clique searches have low runtimes: 178_517 has the greatest runtime: 393s on 1(35) and this reduces to 19s on 64(2240) compute nodes(workers). A complete set of runtime and speedup data is available [16].

The results demonstrate that *deploying YewPar on an HPC system can dramatically reduce the runtime of different types of combinatorial search compared with state of the art sequential and parallel implementations*. As a further example, the runtimes for a Maximum Clique p_hat1000-3 have fallen from 130.8h (Sequential), 4.2h (Cilk+) and 3.0h (C++ custom threading) on a dual 32-core Intel Xeon E5-2697A (2.6 GHz) [18] to 102s (4480 YewPar workers on Cirrus). As the Xeon has a faster clock speed than both GPG and Cirrus we would expect even longer sequential runtimes on these platforms.

Speedups Figure 9 shows the speedups for the Numerical Semigroups genus 61 search. The speedups are relative to execution on 16(560) compute nodes(workers). The relative speedup is near linear up to 4480 workers, with parallel efficiency over 90%. By 6825 workers both speedup and efficiency have declined.

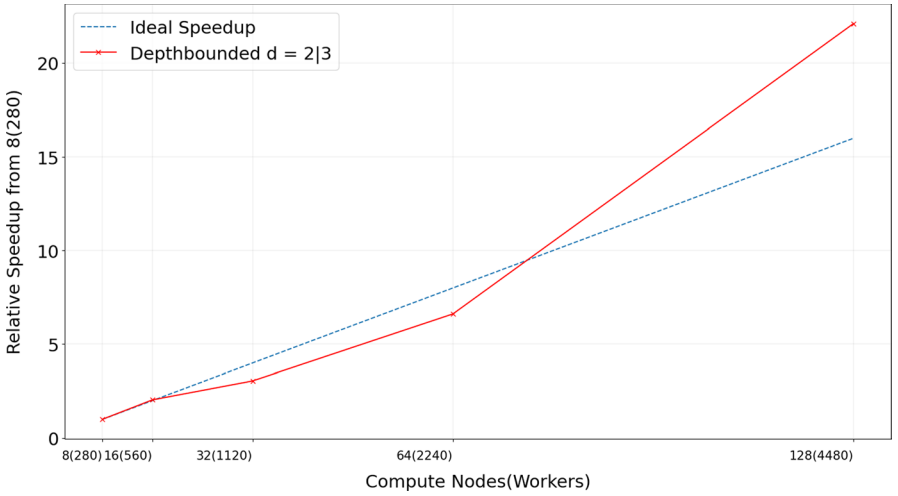

Fig. 10 Maximum Clique p_hat1000-3 search speedup relative to 8(280) Cirrus compute nodes(workers); Depthbounded with cutoff 2, increased to 3 for 16(4480) to minimise starvation

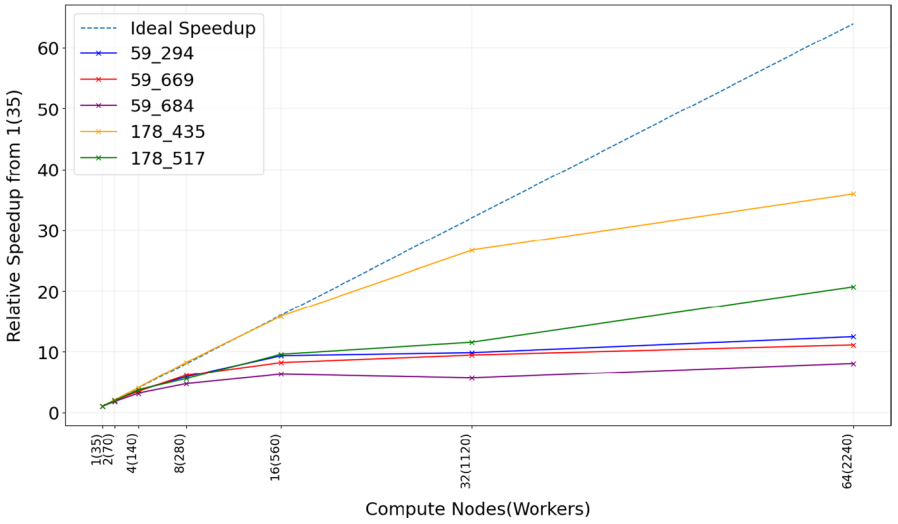

Fig. 11 *k*-clique finite geometry search speedups relative to 1(35) Cirrus compute node(workers); Depthbounded with cutoff 3

Figure 10 shows relative speedups from 8(280) compute nodes (workers) for the Maximum Clique p_hat1000-3 search. Relative speedups increase steadily as the core counts increase up until 4608 cores where super-linear speedups are achieved. Super-linear speedups are common for optimisation searches where pruning can dramatically reduce the workload.

Figure 11 shows relative speedups from 1(35) compute nodes(workers) for 5 instances of the *k*-clique decision search. Speedups vary from instance to instance. The best speedup achieved is for instance 178_435 that has a significantly longer runtime at 1(35) i.e. 4563s. Lower speedups are achieved for instances with lower runtimes on 1(35), and these range from 393s for 178_517 to 113s for 59_684. For these instances runtimes are reduced to between 10s and 20s on 64(2240) compute nodes(workers), and the system is starved of work. So it is likely that far better scaling could be achieved for larger search instances.

7 Conclusions

We report the first ever study of generic combinatorial search at HPC scale, i.e. 100s of compute nodes and more than 4000 cores. The study demonstrates the capacity of the YewPar search framework to scale to HPC.

*We have demonstrated **generic** high performance combinatorial search*, i.e. that a variety of exact combinatorial searches can be easily parallelised for HPC using YewPar. Complete implementations of sophisticated state-of-the art parallel searches require only around 500 lines of code. Previously (1) just a few searches have been individually hand-crafted for HPC scale e.g. [4, 9]; and (2) the genericity of YewPar has only been demonstrated on a modest cluster (100s of cores) by parallelising seven searches [5]. Here we exhibit HPC-scale searches using different YewPar skeletons and covering the three search types: optimisation, enumeration, and decision (Sect. 4).

We have presented *a new mechanism for profiling key aspects of generic parallel combinatorial search in YewPar.* The extreme irregularity of parallel combinatorial search has only rarely been measured, and then only for specific search applications, e.g. [25]. We exhibit profiles that quantify the irregularity of many search applications in the generic YewPar framework. Although implemented for YewPar and in HPX the profiling techniques do not depend directly on either. Search task runtime profiling aids parallelisation by providing information on aspects like the huge differences in search task runtimes, mean task runtime, and the radically different (and frequently multi-modal) task runtime distributions at each search tree depth, e.g. Fig. 4. Profiling node throughput quantifies the dramatic differences in parallel behaviour between enumeration searches with fixed workloads, e.g. Fig. 7, and optimisation searches with variable workloads, e.g. Fig. 8. Profiling has a geometric mean overhead of only 5% (Sect. 5.5) making it accurate and usable in practice. Moreover profiling can be turned off completely to maximise performance (Sect. 5).

We demonstrate, for the first time, generic exact combinatorial searches at HPC scale. Baselining against state-of-the-art sequential C++ and C++/OpenMP implementations on 9 standard (DIMACS) search instances shows that the generality of YewPar incurs a mean sequential slowdown of 9%, and a mean parallel slowdown of 27.6% on a single 18-core compute node (Table 2). Guided by the profiling we effectively parallelise seven standard instances of the three searches, and systematically measure runtime and relative speedups at scale. We show how *deploying YewPar on an HPC system can deliver dramatic reductions in runtime compared with*

state of the art hand crafted search implementations, both sequential and parallelised at smaller scale. For example reducing the p_hat1000-3 sequential search from 131 h to just 102 s using YewPar on 4480 cores (Fig. 10).

Comparing different search types shows similar speedup and scaling characteristics to smaller-scale parallel search [5], e.g. pruning in the Maximum Clique optimisation search reduces workload and hence delivers super-linear speedups up to 128(4480) compute-nodes(workers) (Fig. 10). The maximum relative speedups we achieve for the Numerical Semigroups enumeration search are near-linear up to 192(6825) compute-nodes(workers) (Fig. 9), and sub-linear for five k-clique decision searches on up to 64(2240) compute-nodes(workers) (Fig. 11). It is likely that far better scaling can be achieved for k-clique, and other decision searches, if suitable instances can be found (Sect. 6).

Ongoing Work Currently determining good parameters for a search instance, like depth cutoff or backtrack budget, entails a parameter sweep. Ongoing work seeks to determine whether we can use pre-execution profiling to predict parameters, e.g. is backtracks-per-second-per-worker sufficient to determine an appropriate budget for search instances? We would also like to explore whether performance can be improved by extending YewPar to use the profiling metrics to dynamically adapt the search, e.g. a compute node with ample work may increase the depth cutoff to provide more search tasks.

A benefit of profiling the general framework is that we can explore other search applications such as the Unbalanced-Tree Search benchmark [24] that is already supported in YewPar [2].

Acknowledgements This work was supported EPSRC grants MaRIONet (EP/P006434), STARDUST (EP/T014628) and S4: Science of Sensor Systems Software (EP/N007565).

Author contributions RM: methodology, software, validation, experimentation, visualization, writing—original draft, writing—review & editing; BA: methodology, software, supervision, writing/review/editing, writing—original draft, writing—review and editing; PT: methodology, supervision, writing—original draft, writing—review and editing

Declarations

Conflict of interests The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Alba, E., et al.: MALLBA: a library of skeletons for combinatorial optimisation. In: Euro-Par, Paderborn, Germany, August, 2002, Proceedings (2002)
2. Archibald, B.: Skeletons for Exact Combinatorial Search at Scale. Ph.D. thesis, University of Glasgow (2018) http://theses.gla.ac.uk/id/eprint/31000
3. Archibald, B., Maier, P., Stewart, R., Trinder, P.: Implementing YewPar: A Framework for Parallel Tree Search. Euro-Par, Gottingen (2019)
4. Archibald, B., et al.: Sequential and parallel solution-biased search for subgraph algorithms. In: CPAIOR 16th Thessaloniki, Greece, June 2019 (2019)
5. Archibald, B., et al.: YewPar: skeletons for exact combinatorial search. In: PPoPP'20:, San Diego, California, USA, February, 2020. ACM (2020)
6. Barucci, V., et al.: Maximality Properties in Numerical Semigroups and Applications to One-Dimensional Analytically Irreducible Local Domains, vol. 598. American Mathematical Society, Providence (1997)
7. Fromentin, J., Hivert, F.: Exploring the tree of numerical semigroups. Am. Math. Comput. **85**(301), 2553–2568 (2016)
8. Galea, F., Le Cun, B.: Bob++: a framework for exact combinatorial optimization methods on parallel machines. In: International Conference High Performance Computing and Simulation (HPCS) (2007)
9. Heule, M.J.H., Kullmann, O.: The science of brute force. Commun. ACM **60**(8), 70–79 (2017)
10. Johnson, D.J., Trick, M.A. (eds.): Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October, 1993. American Mathematical Society (1996)
11. Kaiser, H., et al.: HPX: A Task Based Programming Model in a Global Address Space. In: ICPG-ASPM 2014, Eugene, OR, USA, Oct. 2014 (2014)
12. Kehrer, S., Blochinger, W.: Development and operation of elastic parallel tree search applications using TASKWORK. In: Ferguson, D., Muñoz, V.M., Pahl, C., Helfert, M. (eds.) Cloud Computing and Services Science–9th International Conference, CLOSER 2019, Heraklion, Crete, Greece, May 2–4, 2019, Revised Selected Papers. Communications in Computer and Information Science, vol. 1218, pp. 42–65. Springer, Berlin (2019). https://doi.org/10.1007/978-3-030-49432-2_3
13. Klein, A., Storme, L.: Applications of finite geometry in coding theory and cryptography. Inf. Secur. Cod. Theory Rel. Comb. **29**, 38–58 (2011)
14. Knizikevičius, I., Trinder, P., Archibald, B., Yan, J.: Parallel flowshop in YewPar (2022). https://doi.org/10.48550/ARXIV.2207.06902
15. Knizikevičius, I., Trinder, P., Archibald, B., Yan, J.: Parallel Flowshop in YewPar. arXiv preprint arXiv:2207.06902 (2022)
16. MacGregor, R.: Generic High Performance Exact Combinatorial Search [Data Repository]. https://doi.org/10.5281/zenodo.4270336
17. Maher, S.J., Ralphs, T.K., Shinano, Y.: Assessing the effectiveness of (parallel) branch-and-bound algorithms. arXiv preprint arXiv:2104.10025 (2021)
18. McCreesh, C.: Solving hard subgraph problems in parallel. Ph.D. thesis, University of Glasgow (2017)
19. McCreesh, C.: Sequential MCsa1 Maximum Clique Implementation (2018). https://github.com/ciaranm/sicsa-multicore-challenge-iii/c++/
20. McCreesh, C., Prosser, P.: Multi-threading a state-of-the-art maximum clique algorithm. Algorithms **6**(4), 618–635 (2013)
21. McCreesh, C., Prosser, P.: The shape of the search tree for the maximum clique problem and the implications for parallel branch and bound. TOPC (2015). https://doi.org/10.1145/2742359
22. McCreesh, C., Prosser, P., Trimble, J.: The glasgow subgraph solver: Using constraint programming to tackle hard subgraph isomorphism problem variants. In: Gadducci, F., Kehrer, T. (eds.) Graph Transformation—13th International Conference, ICGT 2020, Held as Part of STAF 2020, Bergen, Norway, June 25–26, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12150, pp. 316–324. Springer, Berlin (2020). https://doi.org/10.1007/978-3-030-51372-6_19
23. Menouer, T., et al.: Mixing static and dynamic partitioning to parallelize a constraint programming solver. Int. J. Parallel Program. (2016). https://doi.org/10.1007/s10766-015-0356-7

24. Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J., Sadayappan, P., Tseng, C.W.: UTS: an unbalanced tree search benchmark. In: International Workshop on Languages and Compilers for Parallel Computing, pp. 235–250. Springer, Berlin
25. Otten, L., Dechter, R.: AND/OR branch-and-bound on a computational grid. J. Artif. Intell. Res. **59**, 351–435 (2017)
26. Pardalos, P., Xue, J.: The maximum clique problem. J. Global Optim. **4**, 301–328 (1994)
27. Pietracaprina, A., et al.: Space-efficient parallel algorithms for combinatorial search problems. J. Parallel Distrib. Comput. **76**, 58–65 (2015)
28. Poldner, M., Kuchen, H.: Algorithmic skeletons for branch & bound. In: ICSOFT, Setúbal, Portugal, Sept. 2006 (2006)
29. Prosser, P.: Exact algorithms for maximum clique: a computational study. Algorithms (2012). https://doi.org/10.3390/a5040545
30. Régin, J., Rezgui, M., Malapert, A.: Embarrassingly parallel search. In: Schulte, C. (ed.) Principles and Practice of Constraint Programming—19th International Conference, CP 2013, Uppsala, Sweden, Sept 16–20, vol. 8124, pp. 596–610. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-40627-0_45
31. The GAP Group: GAP - Groups, Algorithms, and Programming, Version 4.8.7 (2017), https://www.gap-system.org/Releases/4.8.7.html

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.