Towards Generic Scalable Parallel Combinatorial Search

ARCHIBALD, Blair, MAIER, Patrick <http://orcid.org/0000-0002-7051-8169>, STEWART, Robert, TRINDER, Phil and DE BEULE, Jan

Published version

ARCHIBALD, Blair, MAIER, Patrick, STEWART, Robert, TRINDER, Phil and DE BEULE, Jan (2017). Towards Generic Scalable Parallel Combinatorial Search. In: PASCO 2017 : Proceedings of the International Workshop on Parallel Symbolic Computation. ACM, 1-10.

Towards Generic Scalable Parallel Combinatorial Search

Blair Archibald
University of Glasgow
b.archibald.1@research.gla.ac.uk

Patrick Maier
University of Glasgow
Patrick.Maier@glasgow.ac.uk

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

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

Jan De Beule
Vrije Universiteit Brussel
Jan.De.Beule@vub.ac.be

ABSTRACT

Combinatorial search problems in mathematics, e.g. in finite geometry, are notoriously hard; a state-of-the-art backtracking search algorithm can easily take months to solve a single problem. There is clearly demand for parallel combinatorial search algorithms scaling to hundreds of cores and beyond. However, backtracking combinatorial searches are challenging to parallelise due to their sensitivity to search order and due to the their irregularly shaped search trees. Moreover, scaling parallel search to hundreds of cores generally requires highly specialist parallel programming expertise.

This paper proposes a generic scalable framework for solving hard combinatorial problems. Key elements are distributed memory task parallelism (to achieve scale), work stealing (to cope with irregularity), and generic algorithmic skeletons for combinatorial search (to reduce the parallelism expertise required). We outline two implementations: a mature Haskell Tree Search Library (HTSL) based around algorithmic skeletons and a prototype C++ Tree Search Library (CTSL) that uses hand coded applications.

Experiments on maximum clique problems and on a problem in finite geometry, the search for spreads in $H(4, 2^2)$, show that (1) CTSL consistently outperforms HTSL on sequential runs, and (2) both libraries scale to 200 cores, e.g. speeding up spreads search by a factor of 81 (HTSL) and 60 (CTSL), respectively. This demonstrates the potential of our generic framework for scaling parallel combinatorial search to large distributed memory platforms.

KEYWORDS

Combinatorics, Backtracking, Distributed Computing, Parallelism, Clique Search, Finite Geometry

1 INTRODUCTION

Many algebraic structures, e.g. graphs, groups, or geometries, possess a rich space of substructures. Exploring this space, e.g. searching for a substructure with certain properties, or simply enumerating all substructures of a certain kind, is a common problem in mathematical research. The sheer size and complexity of the search space makes such exploration a hard combinatorial search problem.

Running combinatorial searches sequentially, i.e. on a single CPU core, may take weeks or months (or longer), which is often too long to be useful. A parallel search algorithm, e.g. running on a shared memory multicore machine, may reduce the search time. Considering the sheer size of the spaces we would like to search, however, the limited parallel capabilities offered by shared-memory machines are insufficient. Instead, a useful parallel combinatorial search must *scale* beyond these limits to distributed memory platforms, e.g. compute clusters and supercomputers, to stand a chance of reducing search times from several months to a few hours.

Combinatorial search algorithms are typically based on backtracking search combined with heuristics for guiding the search order and pruning infeasible branches of the search tree. Parallelising such algorithms is challenging due to the irregular shape of the search tree and due to the dependencies introduced by order and pruning heuristics. Moreover, scaling parallel search to hundreds or thousands of cores generally requires expert parallel programming skills. A *generic* parallel framework for scalable combinatorial search would alleviate these engineering challenges, and make scalable search available to users without extensive parallel programming skills.

In this paper, we propose a framework for parallelising hard combinatorial search problems. For scaling to compute clusters and even supercomputers, the framework is built around a distributed memory task parallel programming model. The framework relies on sophisticated distributed work stealing algorithms to smooth load imbalances caused by the irregularity inherent in parallel combinatorial searches. To the user the framework presents itself as generic *algorithmic skeletons* for parallel combinatorial search, hiding the complex coordination required to scale such searches.

Outline and Contributions. Section 2 introduces a generic API for combinatorial search in terms of generating and pruning a search tree, and goes on to classify combinatorial search problems and discuss the implications for parallel search. The paper then makes the following novel research contributions:

- We a propose a scalable framework for solving hard combinatorial problems. Key elements of the framework are distributed memory task parallelism; work-stealing; and algorithmic skeletons. We outline two implementations: the Haskell Tree Search Library (HTSL) [3] based on HdpH [16] and a prototype C++ Tree Search Library (CTSL) based on HPX [14], the latter being entirely novel (Section 3).
- We demonstrate the *generality* of the framework by exhibiting CTSL and HTSL implementations of maximum clique searches in graphs (Section 4), and deciding if a maximal spread exists in the geometry $H(4, 2^2)$, thereby showing that parallel clique search algorithms are a feasible way

PASCO 2017, Kaiserslautern, Germany

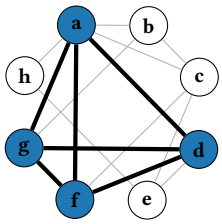

Figure 1: A graph with maximum clique $\{a, d, f, g\}$.

to attack problems in finite geometry (Section 5). To implement the spread search we define a novel skeleton for combinatorial decision problems, and provide an HTSL implementation (Section 5.3).

While the HTSL maximum clique implementation and performance have previously been reported [3], all CTSL results are new, as are all spread search results. To our knowledge this is the first ever *distributed-memory* parallel search for spreads in finite geometries, albeit a relatively naïve search.

- We demonstrate the *scalability* of the approach on a 17-host 272-core Beowulf cluster, scaling the searches to use up to 200 workers. On a benchmark suite of 13 standard instances, the highest maximum clique speedups are 149 for HTSL and 76 for CTSL (Section 4.2). We validate a known result in finite geometry, namely that there is no spread in $H(4, 2^2)$ in around 4.5 minutes (CTSL) and around 20 minutes (HTSL); the highest speedups are 81 for HTSL and 60 for CTSL (Section 5.4). The datasets supporting this evaluation are available from an open access archive [2].

2 TREE SEARCH PROBLEMS

This section presents a generic backtracking search algorithm, sample instantiations to concrete search problems, and an analysis of the challenges for parallelising backtracking search.

Searching for cliques in graphs is core to both case studies in this work: finding a maximum clique in a graph (Section 4) and deciding if a spread exists in $H(4, 2^2)$ (Section 5). To this end, we use clique search as an illustrative example throughout this section.

A *clique* is a set of pairwise adjacent vertices in an undirected graph; for example, the set $\{a, b, c\}$ is a clique of the graph in Figure 1 whereas the set $\{a, b, h\}$ is not. Given a graph and a positive integer k, the *k-clique problem* is to decide whether the graph contains a clique of k vertices. Given a graph, the *maximum clique problem* is to find a clique of maximum size; e.g. $\{a, d, f, g\}$ is such a maximum clique of the graph in Figure 1.

2.1 Combinatorial Search

Combinatorial search problems typically look for a substructure satisfying certain properties within a larger structure or space, for instance a clique of size 33 or a maximum clique in a graph. A typical approach to solving such problems systematically traverses a *search tree*, where each node represents a (partial) substructure, and each of its children extends that substructure. A *backtracking search* descends down a path of the tree as long as the substructure can be extended, otherwise it backs up to the parent node and descends down another path. While traversing the tree, a *maximising* backtracking search selects a substructure (typically the first one) that maximises the desired property. Backtracking searches can often be sped up by *pruning* subtrees that are heuristically shown not to harbour substructures with the desired property.

```
Node max_search(Node incumbent, Node current):
  if prune(incumbent, current):
    return incumbent

  // Maximise property by strengthening incumbent
  if property(current) > property(incumbent):
    incumbent ← current

  // Recurse to children, in the order generated
  for child ← generate(current):
    incumbent ← max_search(incumbent, child)

  return incumbent
```

Algorithm 1: Generic maximising backtracking search

A generic backtracking search, such as `max_search` in Algorithm 1, recursively traverses the search tree below the current node and returns the incumbent. That is, the first node that maximises the desired property. The generic search operates on an abstract Node type and depends on three function parameters: Function `generate` returns a list of all children of the current node, function `property` expresses the property to be maximised, and predicate `prune` decides whether to skip the subtree below the current node because the nodes therein cannot strengthen the incumbent. We illustrate backtracking search by instantiating the generic algorithm to two clique search problems.

2.1.1 Optimality Search. Backtracking search can be used to find an optimal substructure, for example, a clique of maximum size in an undirected graph. The equations below specify the function parameters for solving the maximum clique problem using the generic `max_search` algorithm above. In this case, search tree nodes are triples of the form $\langle G, C, V\rangle$, where G is (a reference to) the input graph, C is a set of vertices forming a clique in G, and V is a set of candidate vertices that may extend C while maintaining the clique property. Calling `max_search`(root, root), where root = $\langle G, \emptyset, V_G\rangle$ is the root node of the search tree and V_G the set of all vertices of G, will return a node $\langle G, C, \emptyset\rangle$ such that C is a clique of maximum size.

$$[\mathit{Node}]\ \texttt{generate}(\mathit{Node}\ \langle G, C, V\rangle) =_{\text{def}} [\langle G, C_1, V_1\rangle, \ldots, \langle G, C_n, V_n\rangle] \text{ where } V = \{u_1, \ldots, u_n\},\ C_i = C \cup \{u_i\} \text{ and } V_i = \{v \in V \setminus \{u_1, \ldots, u_i\} \mid C_i \cup \{v\} \text{ is a clique in } G\}$$

$$\mathit{int}\ \texttt{property}(\mathit{Node}\ \langle G, C, V\rangle) =_{\text{def}} |C|$$

$$\mathit{bool}\ \texttt{prune}(\mathit{Node}\ \langle \_, \widehat{C}, \_\rangle,\ \mathit{Node}\ \langle G, C, V\rangle) =_{\text{def}} |C| + \texttt{colours}(G, V) \leq |\widehat{C}|$$

Function `generate` enumerates all children of the current node as a list; each child node $\langle G, C_i, V_i\rangle$ adds a candidate vertex u_i to

the current clique C and adjusts the set of candidates to maintain the candidate invariant. Function `property` just returns the size of the current clique.

Heuristics for pruning maximum clique searches are well studied; see [26] for a review. The heuristic chosen here, and in current state-of-the-art parallel *branch-and-bound* solvers [8, 17], relies on fast graph colouring algorithms for bounding the clique size by the number of colours used. A subtree is pruned if the bound $|C| + \texttt{colours}(G, V)$ cannot beat $|\widehat{C}|$, the size of incumbent clique.

The effectiveness of pruning depends on the strength of the incumbent; the bigger the incumbent the more can be pruned. Therefore, the search order is important, and many search algorithms [17] prioritise paths in the search tree that are likely to yield larger cliques. The `generate` function offers a way to control search order since it returns the current node's children as an ordered list. For simplicity, the above definition of `generate` picks an arbitrary order.

2.1.2 Decision Problems. Backtracking search can also be used to solve decision problems where we wish to check the existence of substructures within the search space, for example checking whether a graph has a clique of a given size k. This is a seemingly different problem, as finding such a clique requires terminating the search early without traversing the entire search tree. However, early termination can be viewed as pruning, and the k-clique problem can be solved by another instance of the generic `max_search` algorithm. The equations below specify the three function parameters. Search tree nodes are quadruples of the form $\langle k, G, C, V\rangle$, where k is a positive integer specifying the desired clique size, and G, C, V are as before. Calling `max_search`(root, root), where root = $\langle k, G, \emptyset, V_G\rangle$, will either return the root node (in case G has no k-clique) or a node $\langle k, G, C, V\rangle$ such that C is a clique of size k.

$$[\mathit{Node}]\ \texttt{generate}(\mathit{Node}\ \langle k, G, C, V\rangle) =_{\text{def}} [\langle k, G, C_1, V_1\rangle, \ldots, \langle k, G, C_n, V_n\rangle] \text{ where } \ldots$$

$$\mathit{bool}\ \texttt{property}(\mathit{Node}\ \langle k, G, C, V\rangle) =_{\text{def}} |C| = k$$

$$\mathit{bool}\ \texttt{prune}(\mathit{Node}\ \langle \_, \_, \widehat{C}, \_\rangle,\ \mathit{Node}\ \langle k, G, C, V\rangle) =_{\text{def}} |\widehat{C}| = k \text{ or } |C| + \texttt{colours}(G, V) < k$$

Function `generate` is the same as for maximum clique search, mutatis mutandis. Function `property` is a predicate testing whether the current clique is of size k. Note that `property` being a predicate implies that the incumbent can be strengthened at most once. Predicate `prune` is a disjunction of two conditions. The first part ensures early termination once the incumbent has been strengthened by testing whether the incumbent clique $\widehat{C}$ is of size k. The second prunes infeasible subtrees using a colouring-based bounding heuristic similar to that used in a maximum clique search.

2.2 Parallel Tree Search

Parallel tree traversals often follow a divide-and-conquer paradigm, creating a new parallel task for each subtree traversal. This approach does not work very well for search tree traversals, for two reasons.

Firstly, search trees are highly imbalanced, and their exact shape is not predictable prior to search, which results in highly irregular task granularity. Dynamic scheduling, e.g. random work stealing, can load balance irregular tasks but only if most tasks exceed a minimum granularity threshold that depends on the overhead incurred by dynamic scheduling. Unfortunately, pruning can turn many tasks trivial, depending on the strength of the incumbent, exacerbating the irregularity. Moreover, the randomness introduced by dynamic scheduling can disrupt the search order, with dramatic effects on performance [3].

Secondly, backtracking search does not quite fit the divide-and-conquer scheme because subtree traversals aren't independent but share information via the incumbent; e.g. see how the `for` loop of Algorithm 1 threads through the incumbent. Consequently, parallel tasks need to share information, for example broadcasting the incumbent whenever it has been strengthened.

The frequency of incumbent updates is one of the key differences between optimality and decision searches. Optimality searches may strengthen the incumbent many times; for instance, a maximum clique search will update the incumbent k times when finding a clique of size k. In contrast, decision searches will strengthen the incumbent once (if the searched-for substructure exists) or never. Yet, if the incumbent is never updated, subtree traversals do not share information and can run independently in parallel, as in plain divide-and-conquer. This justifies different implementations for parallel backtracking search depending on the search type.

2.3 Skeletons for Combinatorial Search

The differences in parallel coordination between optimality searches and decision searches can be abstracted into *algorithmic skeletons* [7], general parallel coordination patterns that are parameterised by the domain specific computation (e.g. clique search). Table 1 categorises search skeletons implemented in HTSL (Section 3.1) and CTSL (Section 3.2) by search type and by the search order guarantees provided; entries in ***bold*** denote novel implementations contributed by this paper.

Table 1: HTSL and CTSL Skeleton Implementations

	Ordered tasks	Unordered tasks
Optimality search	HTSL	HTSL, ***CTSL***
Decision search		***HTSL***, ***CTSL***

Previous work [3] explores the effect of preserving/not preserving search order on optimality searches, using skeletons implemented in HTSL. This paper is not concerned with preserving search order.

This paper contributes a new HTSL skeleton for decision search and considers the performance of both the decision and optimality searches in HTSL and a prototype implementation in CTSL, a new prototype tree search library. As CTSL does not yet provide skeletons, CTSL applications are implemented directly on top of a low-level library for dynamically scheduling task parallel computations on distributed memory platforms. We wish to show the two implementations, based on the same core search algorithms, give similar performance properties (accounting for sequential baseline differences).

2.4 Related Frameworks

While many tree search parallelisations are application specific, we are not the first to generalise tree search computation patterns into re-useable skeletons/framework, nor the first to support distributed memory platforms.

MallBA [1] provides an entire suite of skeletons, encompassing both *exact* and *heuristic* methods, for solving combinatorial search problems. Problems are specified an object oriented manner with the ability to run on distributed memory architectures via MPI. The branch-and-bound algorithm relies on a central master scheduler, as opposed to distributed work stealing here.

Muesli [21] is similar in taking a skeleton approach, in using C++, and in providing distributed memory parallelism. Where Muesli provides communication using MPI we use HPX. Muesli focuses on branch-and-bound parallelism and studies how the choice of work pool organisation, centralised or distributed, affects scalability and search performance. The distributed model restricts communication to a fixed ring topology, in contrast to our random work stealing approach.

Bob++ [9] provides a high level object orientated model to define various optimality searches including branch-and-bound. Its key feature is the use of a global priority queue structure that decouples the application from the underlying parallelism framework. This allows parallel execution on p-threads, MPI or Anthapascan/Kaapi [11]; the latter is similar to our approach in using distributed work-stealing to balance irregular load.

We address compute bound problems here, but as with symbolic computations in general, many searches are memory bound. In these cases frameworks such as Roomy [15] may be used.

3 TASK PARALLEL LIBRARIES FOR SCALABLE PARALLEL SEARCH

The combinatorial search from the previous section is general enough to encode many well-known problems, for instance Travelling Salesperson and 0/1 Knapsack [3]. This section details a task parallel approach to parallelism and shows how to extract the common computation structure into algorithmic skeleton libraries for parallel combinatorial search.

Tree searches map well to the task parallel model where each task searches a subtree, potentially sharing new results with other tasks. As the size of a subtree is unknown a priori, the time taken to search a subtree is difficult to predict in advance, further complicated by dynamic pruning. To deal with the high degree of irregularity we, like many others [11, 12], use work-stealing scheduling to dynamically manage load. Figure 2 shows the interaction between key components of the distributed task parallel model.

In the task parallel model a large computation is broken down into smaller computations that are then shared between the available cores. These smaller computations may themselves generate new tasks. Scaling to distributed memory architectures, such as high performance computers, entails distributing tasks across multiple *hosts*. Each host may run one or more operating system *processes*, which in turn may be assigned multiple *cores*. Each subtree search task is executed by a *worker* mapped to a core. The systems run by having the scheduler assigning tasks to free workers as they become available.

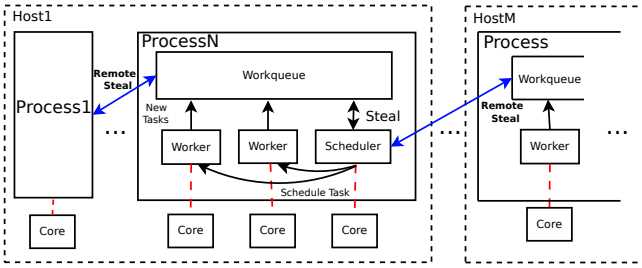

Figure 2: Framework Architecture for Distributed Task Parallelism

The number of tasks to be executed is generally far larger than the number of workers. Additional search tasks are stored in workqueues, where they remain unscheduled. Work stealing scheduling [5] moves unscheduled tasks from the workqueues of heavily loaded processes to processes with empty workqueues. Steals may be local or remote, with remote steals carrying a larger communication cost.

This paper considers two different parallel frameworks/runtimes each embodying the same asynchronous task-parallel model: *HdpH* [16], a domain specific language embedded into Haskell, and *HPX* [14], a parallel runtime system for C++. Specifics related to coordinating parallel combinatorial search applications are layered as libraries on top of these frameworks: the Haskell Tree Search Library (HTSL) and the prototype C++ Tree Search Library (CTSL). We outline the design of these new libraries and their underlying frameworks in Sections 3.1 and 3.2.

3.1 Haskell Tree Search Library

The HdpH framework [16] provides distributed memory task-parallelism for the Haskell programming language. Tasks are managed via two work-stealing schedulers: A local thread scheduler for tasks that have been scheduled for execution within a process, and a *spark* scheduler that manages unscheduled tasks (called "sparks"). Only sparks may be stolen by another process. Distributed work stealing is done at random with the additional optimisation that victim selection is biased to the last successful steal location.

Previous work [3] shows how a generic branch-and-bound API similar to the generic backtracking search of Section 2.1 forms the basis of the Haskell Tree Search Library (HTSL). The API supports many branch-and-bound algorithms, e.g. clique search, travelling salesperson, and knapsack. The parallel coordination of the branch-and-bound pattern has been distilled into an algorithmic skeleton [7] that abstracts over all details of task generation, scheduling and incumbent propagation. It further shows that task scheduling plays a key role in predicting the performance characteristics of a computation.

In Section 5.3 we show how the parallel coordination can be altered to change a branch-and-bound optimality search skeleton into an decision search skeleton.

3.2 C++ Tree Search Library

The overall performance of parallel implementations is crucially dependent on the speed of the sequential tasks. Over a suite of

benchmark problems the sequential (1 worker) Maximum Clique performance of HTSL is between 1.9 and 6.2 times slower than a class-leading, dedicated, and highly optimised C++ Maximum Clique solver [3].

To obtain better sequential performance, and to have better opportunities to move to HPC platforms, we have extended the C++ parallel framework HPX to support HdpH-style *distributed* work stealing and re-implemented Maximum Clique on top. Our work stealing currently uses simple *random* victim selection biased to the location of the last successful steal.

HPX provides support for globally addressable objects via a partitioned global address space, and this feature is used to implement distributed globally accessible work queues. One core per process is dedicated to balance load by invoking steal tasks on remote workqueues.

The C++ Tree Search Library (CTSL) combines the application code with the distributed work-stealing scheduler. The current prototype library provides only a low-level task-parallel coordination layer based on HPX primitives. Higher-level abstractions, such as algorithmic skeletons for combinatorial search based on the generic API of Section 2, are planned for future work.

3.3 Case Studies and Experimental Setup

To show that libraries based on a task parallel, work-stealing, and skeleton approach lead to a framework that is both *general* and *scalable*, we consider two case studies: finding a maximum clique in a graph (Section 4), and checking if a spread exists in a finite geometry (Section 5).

Generality is shown by the fact that two separate libraries, based on the same computation model, implement both case studies and achieve similar performance (when accounting for differences in sequential runtime). Scalability is investigated by subjecting both libraries to the same strong scaling experiments.

CTSL and HTSL are evaluated on a Beowulf cluster consisting of 17 hosts each with dual 8-core Intel Xeon E5-2640v2 CPUs (2Ghz), 64GB of RAM and running Ubuntu 14.04.3 LTS. HTSL is compiled with GHC 8.0.1, CTSL with gcc 6.3.0 and HPX 1.0.

We allocate *workers* + 1 cores per process in order to in have one spare core per process available for handling distributed work-stealing. To avoid over-subscription we allocate a maximum of 3 processes per host, and a maximum of 5 cores per process.[1]

4 PARALLEL MAXIMUM CLIQUE

We investigate the parallel performance of HTSL and CTSL implementations of maximum clique searches, as outlined in Section 2. Although it is possible to guarantee strong performance properties by preserving sequential search ordering [3] this requires a single source of work. For simplicity, and to improve scalability by eliminating the single work source bottleneck, we use an unordered search here.

[1]Performance issues in the GHC runtime system prevent HTSL from scaling to more than 5 cores per process. CTSL could handle more cores; we limit CTSL to 5 cores per process in the interest of a fair comparison.

4.1 Maximum Clique Implementations

Both implementations use the MCsa1 algorithm [22] with bit encoding of candidate vertices to enable fast colouring [23] for both calculating an upper bound and determining a branch ordering.

New tasks are generated for each search tree node above some *depth threshold*. Tasks below the threshold execute sequentially, except for asynchronous bound updates. While simple to implement, this approach requires the setting of an appropriate depth threshold in order to generate sufficiently many tasks. The optimal threshold value depends both on the shape of the search tree and the size of the target architecture.

The HTSL implementation uses generic skeletons, but the CTSL one is a direct implementation, i.e. without skeletons. The source code for the HTSL and CTSL implementations is available at https://doi.org/10.5281/zenodo.556548 and https://doi.org/10.5281/zenodo.556546, respectively.

4.2 Maximum Clique Performance

We evaluate the performance of the CTSL and HTSL maximum clique search implementations on thirteen of the DIMACS clique instances [13]. This set of instances was chosen such that sequential runtimes were no longer than eight hours while also being long enough to enable parallelism to be useful. Results are reported as the mean over ten executions, and the depth threshold is two in all cases. While there is potentially high runtime variance in an unordered search, we have found ten runs give a balance between obtaining a good estimate of the runtime and the time required to obtain more samples.

Figure 3 shows the parallel speedups, relative to a single worker, for the HTSL maximum clique searches.

HTSL scales well especially for the larger instances, achieving a maximum speedup of 149 for 200 workers showing the ability to successfully scale even highly irregular searches. The instances that scale poorly tend to have low sequential runtimes where relatively high communication overheads probably limit the benefits of parallelism.

Figure 4 shows the CTSL relative speedups and, as for HTSL, the instances with the largest sequential times tend to scale the best. While the maximum speedup is only 76 on 200 workers CTSL shows promise towards tackling larger instances.

Table 2 shows a random sample of five maximum clique instances [2]. The first 5 rows show that the CTSL sequential runtimes (in seconds) are significantly lower than the HTSL runtimes. However the second 5 rows of the table show that HTSL scales better than CTSL; the parallel runtimes are far closer, with HTSL even outperforming CTSL on some instances. The scalability differences are likely caused by the immature scheduler/workqueue implementation present in CTSL.

These results should be treated with caution: the CTSL implementation is direct and may therefore avoid some of the overheads of generic skeletons. Moreover, the unordered search used by both libraries gives rise to unpredictable performance as two searches may traverse the search tree in different order [3]. Due to internal workqueue management even the the single worker CTSL and HTSL implementations may follow different search orders.

[2]For lack of space

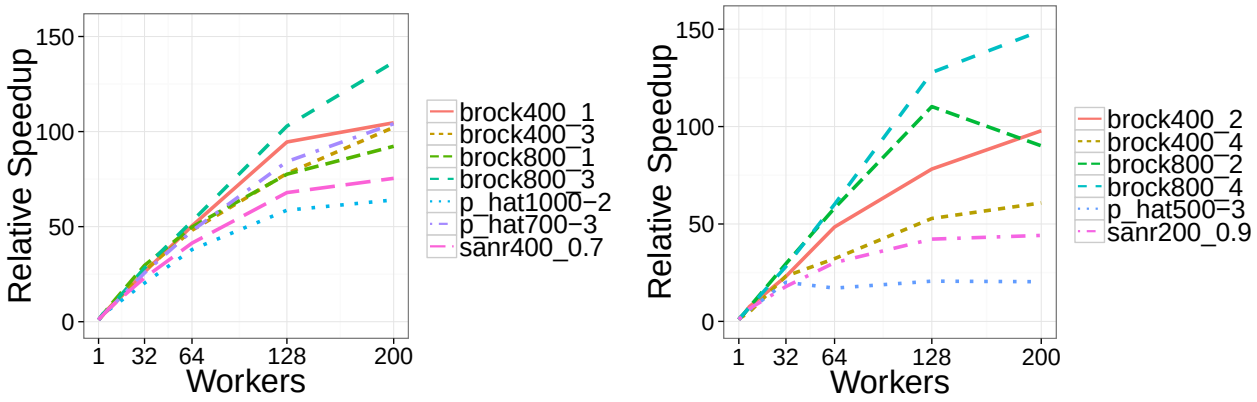

Figure 3: HTSL Maximum Clique Speedups

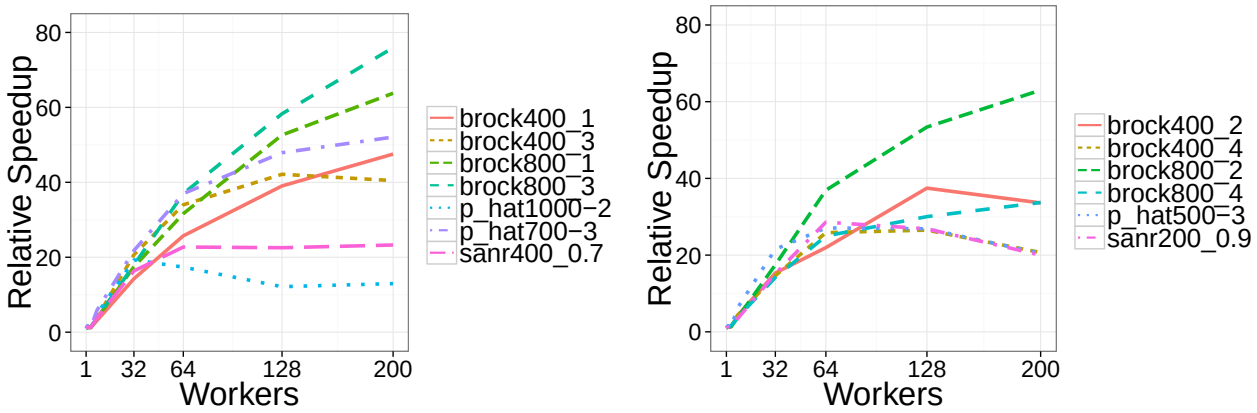

Figure 4: CTSL Maximum Clique Speedups

Table 2: Sample of CTSL/HTSL Maximum Clique Runtimes

		CTSL		HTSL	
Instance/Density	Workers	Time (s)	Speedup	Time (s)	Speedup
brock400_4/0.75	1	84.76	1	245.93	1
brock800_2/0.65	1	3388.08	1	24333.23	1
p_hat1000-2/0.49	1	148.12	1	297.22	1
sanr200_0.9/0.9	1	34.13	1	108.80	1
sanr400_0.7/0.7	1	78.36	1	306.35	1
brock400_4/0.75	200	4.10	21	4.04	61
brock800_2/0.65	200	53.77	63	269.98	90
p_hat1000-2/0.49	200	11.43	13	4.64	64
sanr200_0.9/0.9	200	1.71	20	2.46	44
sanr400_0.7/0.7	200	3.37	23	4.06	75

5 EXISTENCE OF SPREADS IN FINITE GEOMETRIES

We investigate the parallel performance of HTSL and CTSL implementations of a decision problem: seeking spreads in finite geometries. We begin by introducing finite geometry and showing how to map spread searches to clique searches. We then discuss the changes required to convert the optimality search skeleton into a decision search skeleton, and present performance results. Lastly we sketch improvements (Section 5.5) necessary for tackling larger geometries in future.

5.1 Background

Incidence geometry is the study of structures consisting of *elements* (points, lines, planes, etc.) and an *incidence relation* that determines e.g. which points belong to which lines or planes, which lines intersect in a point, or which lines do not have points in common. Incidence geometry has it origin in the axiomatic study of classical Euclidean geometry; its history and development over the 20th century has been extensively described in [6].

Finite geometry is concerned with incidence structures consisting of a finite number of elements; such structures arise naturally when considering algebraic geometry over finite fields. Historically, finite geometry also played a role in the study of finite groups of Lie type, which act in a natural way on certain geometric spaces and give rise to a correspondence between subgroups on the one hand and geometric substructures on the other.

Finite geometry has obvious connections to algebraic combinatorics since many finite geometries give rise to strongly regular

graphs, or more generally, to association schemes. Generally speaking, one is interested in certain substructures of a finite geometry, and such substructures correspond e.g. to certain cliques of the associated graph. The case study described in this section is an example, where the geometric space in question is a so-called *generalised quadrangle*, and the geometric substructure a so-called *spread*.

A *finite generalised quadrangle (GQ)* is a point-line geometry $(\mathcal{P}, \mathcal{B}, \mathrm{I})$, where $\mathcal{P}$ is a set of points, $\mathcal{B}$ is a set of lines, and I is an symmetric incidence relation satisfying the following axioms.

(1) Every point $P \in \mathcal{P}$ is incident with $t + 1$ lines, $t \geq 1$.
(2) Every line $l \in \mathcal{B}$ is incident with $s + 1$ points, $s \geq 1$.
(3) Given a point P and a line l not incident with P, there exists a unique line m incident with P and a unique point Q incident with m such that Q is also incident with l.

The pair (s, t) is called the *order* of the GQ. A GQ of order (s, t) has exactly $(st + 1)(s + 1)$ points and $(st + 1)(t + 1)$ lines. Examples of GQs are found by considering non-degenerate sesquilinear or non-singular quadratic forms of Witt index two on vector spaces over a finite field. These GQs are called the *classical generalised quadrangles*. A well-known classical GQ of order (q^2, q^3), q a prime power, is $\mathrm{H}(4, q^2)$, which arises as the set of projective points of a non-degenerate Hermitian variety in four dimensions over the finite field of size q^2 [20]. Its *collineation group*, i.e. the set of incidence-preserving permutations on points, is denoted by $\mathrm{P\Gamma U}(4, q^2)$. A spread of a GQ of order (s, t) is a set $\mathcal{S}$ of lines, such that every point of the GQ is incident with exactly one line of $\mathcal{S}$; in other words, each point of the *GQ* is covered by exactly one line. By a counting argument, a spread consists of exactly $st + 1$ lines. Some GQs have spreads, others don't. For most classical GQs the existence of spreads is settled; $\mathrm{H}(4, q^2)$ is a notorious open case, where the only known fact is the non-existence of spreads for $q = 2$ [20].

5.2 Spread Search as Clique Search

The question of whether a GQ of order (s, t) has a spread can be reduced to a clique search problem as follows. Let G be the complement of the *line graph* of the GQ. That is, the vertices of G are the lines of the GQ, and two lines are adjacent in G if and only if they do not have any points in common. Note that the graph G is highly symmetric; in fact, its automorphism group coincides with the collineation group of the GQ.

A spread of the GQ is obviously a clique of G of size $st + 1$. Conversely, a clique of size k in G is a set of k lines of the GQ that have no points in common and hence cover $k(s + 1)$ points. Consequently, a clique of size $st + 1$ is a set of $st + 1$ lines covering all points of the GQ, hence it must be a spread.

Using the GAP computational algebra system [25] with packages `fining` [4] and `grape` [24], we have generated the complement line graphs for the GQs $\mathrm{H}(4, 2^2)$ and $\mathrm{H}(4, 3^2)$. Table 3 shows the key properties of these graphs, including the size of the spread/clique to search for. We note that the graphs are dense to very dense and have lots of symmetries.

5.3 Decision Skeleton

The search for spreads takes the form of a decision problem (Section 2.1.2). While the search tree is enumerated in the same divide and conquer way as the maximum clique searches (Section 4) there are two key differences:

(1) Bound propagation is not required; the bound is static and calculated prior to search.
(2) Searches may terminate early, if existence is proved. While an optimality search must traverse (or prune) the full tree to prove optimality, a decision search can terminate as soon as the result is known.

Both libraries realise early termination by passing (a reference to) a distinguished global variable to each worker that, when written to, terminates the search. This differs slightly from the model in Section 2.1.2 where the `prune` predicate ensures that all branches are pruned once existence is proved. The HTSL/CTSL implementations (1) terminate the search instantly without suffering the overheads of individually pruning tens of thousands of tasks, and (2) support instant termination directly in the coordination layer rather than via the user defined `prune` predicate.

As argued in Section 2.1.2, although the *coordination* of the search is different, the decision search API remains the same as the optimality search API, promoting the reuse of existing functionality.

The differences between the optimisation and decision skeletons are highlighted in the pseudo-code listings in Figure 5. Unlike the `max_search` function (Algorithm 1) the incumbent is globally readable by all workers and is atomically updated via the `globalUpdate` function. All parallelism is introduced in the `parfor` loop by constructing a new search task for each child up to the given `maxDepth` threshold. The `killAllWorkers` function explicitly terminates all workers rather than relying on repeated pruning. This allows us to use a completely unmodified pruning function from the optimisation case and can reduce termination time.

Table 3: Properties of the complement line graphs for $\mathrm{H}(4, q^2)$ where $q = 2, 3$

GQ	vertices	density	automorphisms	spread size
$\mathrm{H}(4, 2^2)$	297	0.865	27,371,520	33
$\mathrm{H}(4, 3^2)$	6832	0.960	516,381,143,040	244

5.4 Decision Search Performance

We have implemented the decision search as a skeleton in HTSL and developed a specific clique search implementation in CTSL. Both implementations re-use data structures and functionality of the maximum clique implementations in Section 4.

For both libraries the search is run with 1 – 200 workers (1 – 50 processes) distributed across 17 hosts. Due to the high runtimes, and as we are not investigating performance variability of the runtimes, a single sample is taken for each data point. CTSL runtimes are inclusive of small startup/shutdown overheads (to avoid problems caused by spurious premature shutdown).

The runtime and speedups for both libraries are shown in Table 4. Even on a single core, HTSL can validate that no spread exists in $\mathrm{H}(4, 2^2)$ in approximately a day. The CTSL single core runtimes are far lower: just over 4 hours compared to just over 26.5 hours for HTSL. As for the maximum clique searches (Section 4.2) HTSL scales better than CTSL.

```
global Node incumbent

Node search(int maxDepth, Node root):
  globalUpdate(incumbent, root)

  maxSearch(maxDepth, root)

  return incumbent



void maxSearch(int depth, Node current):
  if prune(incumbent, current):
    return

  if property(current) > property(incumbent):
    globalUpdate(incumbent, current)


  children = generate(current)
  if depth > 0:
    parfor child in children:
      maxSearch(depth - 1, child)
  else:
    for child in children:
      maxSearch(0, child)
```

```
global Node incumbent

Node search(int maxDepth, Node root):
  globalUpdate(incumbent, root)

  decisionSearch(maxDepth, root)

  if incumbent == root:
    return null
  else:
    return incumbent

void decisionSearch(int depth, Node current):
  if prune(incumbent, current):
    return

  if property(current):
    globalUpdate(incumbent, current)
    killAllWorkers()
    return

  children = generate(current)
  if depth > 0:
    parfor child in children:
      decisionSearch(depth - 1, child)
  else:
    for child in children:
      decisionSearch(0, child)
```

Figure 5: Pseudo-code for optimisation skeleton (left) and decision skeleton (right). User provided functions underlined

Table 4: Runtime and Speedup of Decision Search for Spreads in $\mathrm{H}(4, 2^2)$

	CTSL		HTSL	
Workers	Time (s)	Speedup	Time (s)	Speedup
1	16005	1.0	95980	1.0
2	13526	1.2	50946	1.9
4	13234	1.2	27643	3.5
8	3905	4.1	14096	6.8
32	885	18.1	3860	24.9
64	483	33.1	2082	46.1
128	348	46.0	1299	73.9
200	265	60.4	1190	80.6

The results clearly show the benefit of parallelism for spread searches in finite-geometry problems taking the runtime down from just over 26.5 hours to around 20 minutes for HTSL and from just over 4 hours to under 4.5 minutes for CTSL.

5.5 Symmetry Breaking

So far, we have presented a *brute force* approach to checking the existence of spreads, searching the entire graph regardless of symmetries. While this works for $\mathrm{H}(4, 2^2)$, it is unlikely to scale to much larger spaces like $\mathrm{H}(4, 3^2)$, which is more than 20 times as big (Table 3).

The symmetries of the spreads search space are determined by the collineation group of the GQ. For $\mathrm{H}(4, q^2)$ this group is well known and has an efficient representation in the GAP package `fining`. Thus, we can *break symmetry* by *orbital branching* [19], significantly reducing the branching factor of the search tree.

This section shows orbital branching to be an instance of the generic search algorithm from Section 2. It then demonstrates the impact of both symmetry breaking and of the colouring-based pruning heuristic on the performance of spreads search for $\mathrm{H}(4, 2^2)$.

Notation. Let G be the complement line graph of a GQ and let H be a subgroup of G's automorphism group $\mathrm{Aut}(G)$. That is, H is a permutation group on the vertices of G. The set u^H denotes the *orbit* of vertex u under H, i.e. the set of all images of u under the permutations in H; $\mathrm{orbits}_H(V) = \{v^H | v \in V\}$ is the partition of a set of vertices V into orbits under H. Finally, $\mathrm{Aut}(G)_C$ denotes the subgroup H of $\mathrm{Aut}(G)$ that *stabilises* the set of vertices C, i.e. $u^H \subseteq C$ for all $u \in C$.

5.5.1 Symmetry breaking clique search. To take symmetries into account in the k-clique search of Section 2.1.2, we modify the `generate` function to branch on the orbits of candidates rather than on the candidates themselves, as follows. (Functions `property` and `prune` are unchanged.)

$$
\begin{aligned}
&[\mathit{Node}]\ \texttt{generate}(\mathit{Node}\ \langle k, G, C, V\rangle) =_{\text{def}}\\
&\quad [\langle k, G, C_1, V_1\rangle, \ldots, \langle k, G, C_n, V_n\rangle] \text{ where}\\
&\qquad H = \mathrm{Aut}(G)_C, \qquad \mathrm{orbits}_H(V) = \{u_1^H, \ldots, u_n^H\},\\
&\qquad C_i = C \cup \{u_i\} \text{ and}\\
&\qquad V_i = \{v \in V \mid v \notin u_1^H \cup \ldots \cup u_{i-1}^H \cup \{u_i\},\\
&\qquad\qquad\qquad C_i \cup \{v\} \text{ is a clique in } G\}
\end{aligned}
$$

Function `generate` enumerates the orbits of the candidate set V under the automorphims of G that stabilise the current clique C. Each child node $\langle k, G, C_i, V_i\rangle$ adds a representative u_i of the orbit u_i^H to the current clique and adjusts the set of candidates to maintain the candidate invariant. Note that the set of candidates V_i excludes vertices occuring in orbits u_j^H with $j < i$. Ordering the orbits

Table 5: Subproblems at level l of the search tree for spreads in $\mathrm{H}(4, 2^2)$ and $\mathrm{H}(4, 3^2)$

GQ	level	problems	vertices	automorphisms
$\mathrm{H}(4, 2^2)$	0	1	297	27,371,520
	1	1	256	92,106
	2	1	220	720
	3	1	189	18
	4	7	69–161	1–4
$\mathrm{H}(4, 3^2)$	0	1	6832	516,381,143,040
	1	1	6561	75,582,720
	2	1	6300	23,040
	3	3	1433–6049	24–48
	4	329	933–5803	1–12

descending by size is therefore likely to minimize the number of candidates in V_i.

Stabilisers and orbits can be computed by GAP but the cost of these computations can be significant. Fortunately, the stabiliser is getting smaller as the size of C increases and eventually becomes trivial. At that point, orbital branching reverts to ordinary branching, i.e. the `generate` function above behaves exactly like the one in Section 2.1.2. This justifies a two-phase approach to searching for spreads of size k:

(1) Start a symmetry-breaking search in GAP to the level l of the search tree where stabilisers become (nearly) trivial.
(2) For each node $\langle k, G, C, V\rangle$ at level l, generate a $(k-l)$-clique search problem in the subgraph of G induced by the set of candidates V, and solve using HTSL or CTSL.

5.5.2 Performance of two-phase search. Table 5 lists the number and sizes of subproblems generated by the first search phase for $\mathrm{H}(4, 2^2)$. Up to level 3 there is only a single subproblem (that hasn't been pruned away), and there are 7 subproblems at level 4, ranging from 69 to 161 vertices in size. Stabilisers at level 4 are nearly trivial (at most order 4), justifying a switch to the second phase. GAP takes around 20 seconds to generate the entire level 4 of this search tree.

Each of the level 4 subproblems is small enough to be solved by CTSL on a single worker in less than a second. That is, the two-phase approach can reduce the spreads search time from 4 hours to under 1 minute, on a single core.

The level 4 subproblems are small enough to allow us to examine the impact of the pruning heuristic on the search space. Figure 6 compares the sizes of the search trees when using the colouring-based pruning heuristic against a simpler pruning heuristic that only takes into account the size of the candidate set. The graphs show that for these 7 subproblems, colouring-based pruning reduces the size of the search space by four orders of magnitude. This demonstrates that symmetry breaking and pruning are both important for efficient combinatorial search.

5.5.3 Outlook: Spreads in $\mathrm{H}(4, 3^2)$. Table 5 also lists the number and sizes of subproblems generated by the GAP search phase for $\mathrm{H}(4, 3^2)$. The picture is similar to $\mathrm{H}(4, 2^2)$, if scaled up. At level 3 there are 3 subproblems yet stabilisers aren't quite trivial. Level 4 has 329 subproblems, 933 to 5803 vertices big, and stabilisers are nearly trivial (most are of orders 1, 2 or 4, only four are of orders 8 or 12). GAP takes around 90 minutes to generate level 4 of this search tree.

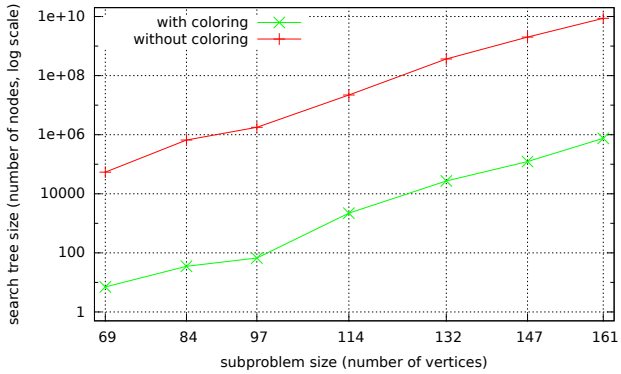

Figure 6: Search tree size of level 4 subproblems of spreads search in $\mathrm{H}(4, 2^2)$, with and without colouring

While GAP is able to compute the upper levels of the search tree in a reasonable time, the subproblems generated are generally too big to be tackled by CTSL in its current state and on our small cluster. Experiments with a random selection of subproblems show that problems with less than 1200 vertices tend to be easy, whereas problems with more than 1300 vertices are computationally hard.

6 CONCLUSION

We propose attacking hard combinatorial problems using a scalable and generic framework comprising distributed memory task parallelism, work-stealing, and algorithmic skeletons. We outline two implementations of the framework: a mature Haskell library (HTSL), and a C++ library (CTSL) in early prototype state (Section 3).

We demonstrate the *generality* of the framework approach with CTSL and HTSL implementations of maximum clique searches in graphs (Section 4), and checking if a spread exists in the generalised quadrangle $\mathrm{H}(4, 2^2)$. This demonstrates that parallel clique search algorithms are a feasible way to attack problems in finite geometry (Section 5).

We demonstrate the *scalability* of the approach by evaluating the performance of the finite geometry decision problem (Section 5.4) and 13 DIMACS maximum clique searches (Section 4.2) on a 17-host 272-core Beowulf cluster. The speedups with up to 200 workers are promising for both libraries. Unsurprisingly CTSL has lower sequential runtimes than HTSL, but does not yet scale as well. However, as C++ is the de-facto standard in high-performance computing, and as a C++ library eases the interoperability with domain-specific C++ libraries, e.g. for breaking symmetries in combinatorial problems, improving CTSL is a worthwhile goal.

Future Work

In the long term our aim is to explore the potential of skeleton-based software libraries for addressing the challenges of parallel

combinatorial search. Much remains to be done to realise this vision, and some immediate steps to advance the work are as follows.

While the improved sequential runtimes of CTSL over HTSL are encouraging, we hope to improve the scaling of CTSL to match HTSL. As a first step we plan to implement search skeletons in CTSL and analyse the overheads. We will exploit low overhead methods like C++ template metaprogramming [10].

Much recent work-stealing scheduler research has focused on the *unbalanced tree search* benchmark [18]. This benchmark has many properties of our searches, although not pruning. The separation of application and scheduler afforded by the skeleton approach will let us investigate many of these techniques further and analyse the effects of the scheduler on the various types of tree search such as enumeration, decision problems and optimisation.

A key step in further investigation of the spreads problem is the use of symmetry breaking to reduce the search space Section 5.5. The API given in Section 2.1 is general enough to encode this by making no distinction on *how* the child nodes are generated. There are many choices on how to best perform symmetry breaking in this context: symmetries can be broken ahead of time, generating N smaller graphs to search, or at runtime. In both cases we require tooling to compute orbits and stabilisers based on subgroups of the collineation group of the geometry. While GAP with the `fining` package is appropriate for a-priori symmetry breaking, it is not clear whether a custom high-performance implementation would be required for scaling search to an HPC environment.

Breaking symmetries also leads to additional computational problems in the search. The reduction in search space will likely reduce the effectiveness of threshold-based work generation, and new methods for generating work will need to be explored. This issue is particularly prevalent in finite geometry problems where after breaking symmetry the tree tends to be very narrow, i.e. have few orbits, at the upper levels.

The next algebraic result we aim to establish using a combination of the CTSL library, symmetry breaking and high performance computing hardware is whether *a spread exists in* $\mathrm{H}(4, 3^2)$.

ACKNOWLEDGMENTS

This work is funded by UK EPSRC grants AJITPar (EP/L000687), CoDiMa (EP/M022641), Glasgow DTA (EP/K503058), MaRIONet (EP/P006434), and Rathlin (EP/K009931). The last author was partially supported by a research grant of the Research Foundation Flanders (Belgium) (FWO) (1504514N). We also thank Ciaran McCreesh, Magnus Morton and the anonymous reviewers for their feedback.

REFERENCES

[1] 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. In *Euro-Par 2002, Paderborn, Germany (LNCS 2400)*. Springer, 927–932. DOI: http://dx.doi.org/10.1007/3-540-45706-2_132

[2] Blair Archibald, Patrick Maier, Robert Stewart, Phil Trinder, and Jan De Beule. 2017. Towards Generic Scalable Parallel Combinatorial Search [Data Collection]. (2017). http://dx.doi.org/10.5525/gla.researchdata.430

[3] Blair Archibald, Ciaran McCreesh, Patrick Maier, Robert Stewart, and Phil Trinder. 2017. Replicable Parallel Branch and Bound Search. (2017). arXiv:1703.05647

[4] John Bamberg, Anton Betten, Philippe Cara, Jan De Beule, Michel Lavrauw, and Max Neunhöffer. 2015. *FinInG – Finite Incidence Geometry, Version 1.3.* http://cage.ugent.be/fining

[5] 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 and Distrib. Comput.* 37, 1 (1996), 55–69. DOI: http://dx.doi.org/10.1006/jpdc.1996.0107

[6] F. Buekenhout (Ed.). 1995. *Handbook of Incidence Geometry.* North-Holland, Amsterdam.

[7] Murray Cole. 1991. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press.

[8] Matjaz Depolli, Janez Konc, Kati Rozman, Roman Trobec, and Dusanka Janezic. 2013. Exact Parallel Maximum Clique Algorithm for General and Protein Graphs. *Journal of Chemical Information and Modeling* 53, 9 (2013), 2217–2228. DOI: http://dx.doi.org/10.1021/ci4002525

[9] A. Djerrah, Bertrand Le Cun, Van-Dat Cung, and Catherine Roucairol. 2006. Bob++: Framework for Solving Optimization Problems with Branch-and-Bound methods. In *High Performance Distributed Computing, HPDC-15, Paris, France.* IEEE, 369–370. DOI: http://dx.doi.org/10.1109/HPDC.2006.1652188

[10] J. Falcou, J. Sérot, T. Chateau, and J.T. Lapresté. 2006. Quaff: efficient C++ design for parallel skeletons. *Parallel Comput.* 32, 7–8 (2006), 604 – 615. DOI: http://dx.doi.org/10.1016/j.parco.2006.06.001

[11] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. 2007. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multiprocessors. In *Parallel Symbolic Computation, PASCO 2007, London, Ontario, Canada.* ACM, 15–23. DOI: http://dx.doi.org/10.1145/1278177.1278182

[12] Yi Guo, Jisheng Zhao, Vincent Cavé, and Vivek Sarkar. 2010. SLAW: A scalable locality-aware adaptive work-stealing scheduler. In *2010 IEEE International Symposium on Parallel Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA.* 1–12. DOI: http://dx.doi.org/10.1109/IPDPS.2010.5470425

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

[14] 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 *Partitioned Global Address Space Programming Models, PGAS 2014, Eugene, Oregon, USA.* ACM, Article 6. DOI: http://dx.doi.org/10.1145/2676870.2676883

[15] Daniel Kunkle. Roomy: A System for Space Limited Computations. In *Parallel Symbolic Computation, PASCO 2010, Grenoble, France.* ACM, 22–25. DOI: http://dx.doi.org/10.1145/1837210.1837216

[16] Patrick Maier, Robert Stewart, and Phil Trinder. 2014. The HdpH DSLs for Scalable Reliable Computation. In *2014 ACM SIGPLAN Symposium on Haskell, Gothenburg, Sweden.* ACM, 65–76. DOI: http://dx.doi.org/10.1145/2633357.2633363

[17] Ciaran McCreesh and Patrick Prosser. 2013. Multi-Threading a State-of-the-Art Maximum Clique Algorithm. *Algorithms* 6, 4 (2013), 618–635. DOI: http://dx.doi.org/10.3390/a6040618

[18] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. 2007. UTS: An unbalanced tree search benchmark. In *Languages and Compilers for Parallel Computing, LCPC 2006, New Orleans, USA (LNCS 4382).* Springer, 235–250.

[19] James Ostrowski, Jeff Linderoth, Fabrizio Rossi, and Stefano Smriglio. 2011. Orbital branching. *Mathematical Programming* 126, 1 (2011), 147–178. DOI: http://dx.doi.org/10.1007/s10107-009-0273-x

[20] Stanley E. Payne and Joseph A. Thas. 2009. *Finite Generalized Quadrangles* (second ed.). European Mathematical Society, Zürich. DOI: http://dx.doi.org/10.4171/066

[21] Michael Poldner and Herbert Kuchen. 2008. Algorithmic skeletons for branch and bound. In *International Conference on Software and Data Technologies, ICSOFT 2006.* Springer, 204–219. DOI: http://dx.doi.org/10.1007/978-3-540-70621-2_17

[22] Patrick Prosser. 2012. Exact Algorithms for Maximum Clique: A Computational Study. *Algorithms* 5, 4 (2012), 545–587. DOI: http://dx.doi.org/10.3390/a5040545

[23] Pablo San Segundo, Fernando Matia, Diego Rodriguez-Losada, and Miguel Hernando. 2011. An improved bit parallel exact maximum clique algorithm. *Optimization Letters* 7, 3 (2011), 467–479.

[24] Leonard Soicher. 2016. *The GRAPE package for GAP, Version 4.7.* http://www.maths.qmul.ac.uk/~leonard/grape/

[25] The GAP Group 2017. *GAP – Groups, Algorithms, and Programming, Version 4.8.7.* The GAP Group. http://www.gap-system.org

[26] Qinghua Wu and Jin-Kao Hao. 2015. A review on algorithms for maximum clique problems. *European Journal of Operational Research* 242, 3 (2015), 693–709. DOI: http://dx.doi.org/10.1016/j.ejor.2014.09.064