J. Parallel Distrib. Comput. 113 (2018) 92–114

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Replicable parallel branch and bound search

Blair Archibald [a,*], Patrick Maier [a], Ciaran McCreesh [a], Robert Stewart [b], Phil Trinder [a]

[a] School Of Computing Science, University of Glasgow, Scotland, G12 8QQ, United Kingdom
[b] Heriot-Watt University, Edinburgh, Scotland, EH14 4AS, United Kingdom

HIGHLIGHTS

- We consider how to gain replicable performance for parallel branch and bound searches.
- We provide a reduction-oriented formal model of parallel branch and bound.
- We present a generic branch and bound API based around higher order functions.
- We design two parallel skeletons each with different performance characteristics.
- Evaluation shows that the Ordered skeleton achieves both good and replicable parallel performance.

ARTICLE INFO

Article history:
Received 20 February 2017
Received in revised form 17 July 2017
Accepted 15 October 2017

Keywords:
Algorithmic skeletons
Branch-and-bound
Parallel algorithms
Combinatorial optimisation
Distributed computing
Repeatability

ABSTRACT

Combinatorial branch and bound searches are a common technique for solving global optimisation and decision problems. Their performance often depends on good search order heuristics, refined over decades of algorithms research. Parallel search necessarily deviates from the sequential search order, sometimes dramatically and unpredictably, e.g. by distributing work at random. This can disrupt effective search order heuristics and lead to unexpected and highly variable parallel performance. The variability makes it hard to reason about the parallel performance of combinatorial searches.

This paper presents a generic parallel branch and bound skeleton, implemented in Haskell, with replicable parallel performance. The skeleton aims to preserve the search order heuristic by distributing work in an ordered fashion, closely following the sequential search order. We demonstrate the generality of the approach by applying the skeleton to 40 instances of three combinatorial problems: Maximum Clique, 0/1 Knapsack and Travelling Salesperson. The overheads of our Haskell skeleton are reasonable: giving slowdown factors of between 1.9 and 6.2 compared with a class-leading, dedicated, and highly optimised C++ Maximum Clique solver. We demonstrate scaling up to 200 cores of a Beowulf cluster, achieving speedups of 100x for several Maximum Clique instances. We demonstrate low variance of parallel performance across all instances of the three combinatorial problems and at all scales up to 200 cores, with median Relative Standard Deviation (RSD) below 2%. Parallel solvers that do not follow the sequential search order exhibit far higher variance, with median RSD exceeding 85% for Knapsack.

1. Introduction

Branch and bound backtracking searches are a widely used class of algorithms. They are often applied to solve a range of NP-hard optimisation problems such as integer and non-linear programming problems; important applications include frequency planning in cellular networks and resource scheduling, e.g. assigning deliveries to routes [26].

Branch and bound systematically explores a *search tree* by subdividing the search space and *branching* recursively into each sub-space. The advantage of branch and bound over exhaustive enumeration stems from the way branch and bound *prunes* branches that cannot better the *incumbent*, i.e. the current best solution, potentially drastically reducing the number of branches to be explored.

The effectiveness of pruning depends on two factors: (1) the accuracy of the problem-specific heuristic to compute *bounds* (2) the value of optimal solutions in each branch, and on the quality of the incumbent; the closer to optimal the incumbent, the more can be pruned. As a result, branch and bound is sensitive to *search order*, i.e. to the order in which branches are explored.

* Corresponding author.
E-mail addresses: b.archibald.1@research.gla.ac.uk (B. Archibald), Patrick.Maier@glasgow.ac.uk (P. Maier), Ciaran.McCreesh@glasgow.ac.uk (C. McCreesh), r.stewart@hw.ac.uk (R. Stewart), Phil.Trinder@glasgow.ac.uk (P. Trinder).

https://doi.org/10.1016/j.jpdc.2017.10.010

A good search order can improve the performance of branch and bound dramatically by finding a good incumbent early on, and highly optimised sequential algorithms following the branch and bound paradigm often rely on very specific orders for performance.

Branch and bound algorithms are hard to parallelise for a number of reasons. Firstly, while branching creates opportunities for speculative parallelism where multiple *workers* i.e threads/processors search particular branches in parallel, pruning counteracts this, limiting potential parallelism. Secondly, parallel pruning requires that processors share access to the incumbent, which limits scalability. Thirdly, parallel exploration of irregularly shaped search trees generates unpredictable numbers of parallel tasks, of highly variable duration, posing challenges for task scheduling. Finally, and most importantly, parallel exploration alters the search order, potentially impacting the effectiveness of pruning.

As a result of the last point in particular, parallel branch and bound searches can exhibit unusual performance characteristics. For instance, slowdowns can arise when the sequential search finds an optimal incumbent quickly but the parallel search delays exploring the optimal branch. Alternately, super-linear speedups are possible in case the parallel search happens on an optimal branch that the sequential search does not explore until much later. In short, the perturbation of the search order caused by adding processors makes it impossible to *predict* parallel performance.

These unusual performance characteristics make reproducible algorithmic research into combinatorial search difficult: was it the new heuristic that improved performance, or were we just lucky with the search ordering in this instance? As the instances we wish to tackle become larger, parallelism is becoming central to algorithmic research, and it is essential to be able to reason about parallel performance.

This paper aims to develop a generic parallel branch and bound search for distributed memory architectures such as clusters. Crucially, the objective is *predictable parallel performance*, and the key to achieving this is careful control of the parallel search order.

The paper starts by illustrating performance anomalies with parallel branch and bound by using a Maximum Clique graph search. The paper then makes the following research contributions:

- To address search order related performance anomalies, Section 2 postulates three *parallel search properties* for replicable performance as follows.

  Sequential Bound: Parallel runtime is never higher than sequential (one worker) runtime.
  Non-increasing Runtimes: Parallel runtime does not increase as the number of workers increases.
  Repeatability: Parallel runtimes of repeated searches on the same parallel configuration have low variance.
- We define a novel formal model for general parallel branch and bound backtracking search problems (BBM) that specifies both search order and parallel reduction (Section 3). We show the generality of BBM by using it to define three different benchmarks with a range of application areas: Maximum Clique (Section 3), 0/1 Knapsack (Appendix B) and Travelling Salesperson (Appendix D).
- We define a new Generic Branch and Bound (GBB) search API that conforms to the BBM (Section 4). The generality of the GBB is shown by using it to implement Maximum Clique (Section 2),[1] 0/1 Knapsack (Appendix C) and Travelling Salesperson (Appendix E).

[1] This implementation being the first *distributed-memory* parallel implementation of San Segundo's bit parallel Maximum Clique algorithm (BBMC) [52].

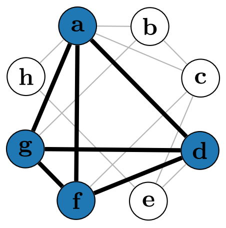

Fig. 1. A graph, with its Maximum Clique $\{a, d, f, g\}$ shown.

- To avoid the significant engineering effort required to produce a parallel implementation for each search algorithm we encapsulate the search behaviours as a pair of *algorithmic skeletons*, that is, as generic polymorphic computation patterns [12], providing distributed memory implementations for the skeletons (Section 5). Both skeletons share the same API yet differ in how they schedule parallel tasks. The *Unordered skeleton* relies on random work stealing, a tried and tested way to scale irregular task-parallel computations. In contrast, the *Ordered skeleton* schedules tasks in an ordered fashion, closely following the sequential search order, so as to guarantee the parallel search properties.
- We compare the sequential performance of the skeletons with a class leading hand tuned C++ search implementation, seeing slowdown factors of between 1.9 and 6.2. We then assess whether the Ordered skeleton preserves the parallel search properties using 40 instances of the three benchmark searches on a cluster with 17 hosts and 200 workers (Section 7). The Ordered skeleton preserves all three properties and produces replicable results. The key results are summarised and discussed in Section 8.

2. The challenges of parallel branch and bound search

We start by considering a branch and bound search application, namely finding the largest clique within a graph. The Maximum Clique problem appears as part of many applications such as in bioinformatics [16], in biochemistry [9,15,18,24], for community detection [66], for document clustering [41], in computer vision, electrical engineering and communications [8], for image comparison [53], as an intermediate step in maximum common subgraph and graph edit distance problems [34], and for controlling flying robots [48].

To illustrate the Maximum Clique problem we use the example graph in Fig. 1. In practice the graphs searched are much larger, having hundreds or thousands of vertices. A clique within a graph is a set of vertices where each vertex in the set is adjacent to every other vertex in the set. For example, in Fig. 1 the set $V = \{a, b, c\}$ is a clique as all vertices are adjacent to one another. $\{a, b, h\}$ is not a clique as there is no edge between b and h. In the Maximum Clique problem we wish to find a largest clique (there may be multiple of the same size) in the graph. Here we are interested in the *exact* solution requiring the full search space to be explored.

One approach to solving this problem would be to enumerate the power set of vertices and check the clique property on each (ordering by largest set). While this approach can work for smaller graphs, the number of combinations grows exponentially with the number of nodes in the graph making it computationally unfeasible for large graphs.

A better approach, particularly for larger graphs, is to only *generate* sets of vertices that maintain the clique property. This is the essence of the *branching* function. In the case of clique search, given any set of vertices, the set of candidate choices is the set of