J. Parallel Distrib. Comput. 84 (2015) 65–75

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

On scalable parallel recursive backtracking

Faisal N. Abu-Khzam[a,*], Khuzaima Daudjee[b], Amer E. Mouawad[b], Naomi Nishimura[b]

[a] Department of Computer Science and Mathematics, Lebanese American University, Beirut, Lebanon

[b] David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario, N2L 3G1, Canada

HIGHLIGHTS

- A simple framework for parallelizing exact search-tree algorithms.
- An indexing scheme for simple task transmission and efficient communication.
- Efficient and effective extraction of heavy tasks for dynamic load balancing.
- The presented method scales almost linearly to a record number of processing elements.

ARTICLE INFO

Article history:
Received 16 March 2014
Received in revised form
3 March 2015
Accepted 7 July 2015
Available online 22 July 2015

Keywords:
Parallel algorithms
Recursive backtracking
Load balancing
Vertex cover
Dominating set

ABSTRACT

Supercomputers are equipped with an increasingly large number of cores to use computational power as a way of solving problems that are otherwise intractable. Unfortunately, getting serial algorithms to run in parallel to take advantage of these computational resources remains a challenge for several application domains. Many parallel algorithms can scale to only hundreds of cores. The limiting factors of such algorithms are usually communication overhead and poor load balancing. Solving NP-hard graph problems to optimality using exact algorithms is an example of an area in which there has so far been limited success in obtaining large scale parallelism. Many of these algorithms use recursive backtracking as their core solution paradigm. In this paper, we propose a lightweight, easy-to-use, scalable approach for transforming almost any recursive backtracking algorithm into a parallel one. Our approach incurs minimal communication overhead and guarantees a load-balancing strategy that is implicit, i.e., does not require any problem-specific knowledge. The key idea behind our approach is the use of efficient traversal operations on an indexed search tree that is oblivious to the problem being solved. We test our approach with parallel implementations of algorithms for the well-known Vertex Cover and Dominating Set problems. On sufficiently hard instances, experimental results show nearly linear speedups for thousands of cores, reducing running times from days to just a few minutes.

1. Introduction

Parallel computation is becoming increasingly important as performance levels out in terms of delivering parallelism within a single processor due to Moore's law. This paradigm shift means that to attain speedup, software that implements algorithms that can run in parallel on multiple processors/cores is required. Today we have a growing list of supercomputers with tremendous processing power. Some of these systems include more than a million computing cores and can achieve up to 30 Petaflop/s. The constant increase in the number of processors/cores per supercomputer motivates the development of parallel algorithms that can efficiently utilize such processing infrastructures. Unfortunately, migrating known serial algorithms to exploit parallelism while maintaining scalability is not straightforward. The overheads introduced by parallelism are very often hard to evaluate, and fair load balancing is possible only when accurate estimates of task "hardness" or "weight" can be calculated on-the-fly. Providing such estimates usually requires problem-specific knowledge, rendering the techniques developed for a certain problem useless when trying to parallelize an algorithm for another.

As it is not likely that polynomial-time algorithms can be found for NP-hard problems, the search for fast deterministic algorithms could benefit greatly from the processing capabilities of supercomputers. Researchers working in the area of exact algorithms have developed algorithms yielding lower and lower running times

* Corresponding author.
E-mail addresses: faisal.abukhzam@lau.edu.lb (F.N. Abu-Khzam), kdaudjee@uwaterloo.ca (K. Daudjee), aabdomou@uwaterloo.ca (A.E. Mouawad), nishi@uwaterloo.ca (N. Nishimura).

http://dx.doi.org/10.1016/j.jpdc.2015.07.006
0743-7315/© 2015 Elsevier Inc. All rights reserved.

```
1:  procedure SERIAL-RB($N_{d,p}$)
2:      if (ISSOLUTION($N_{d,p}$)) then
3:          $best\_so\_far \leftarrow N_{d,p}$;
4:      if (ISLEAF($N_{d,p}$)) then
5:          Backtrack;                                  ▷ undo operations
6:      $p' \leftarrow 0$;
7:      while HASNEXTCHILD($N_{d,p}$) do
8:          $N_{d+1,p'} \leftarrow$ GETNEXTCHILD($N_{d,p}$);
9:          SERIAL-RB($N_{d+1,p'}$);
10:         $p' \leftarrow p' + 1$;
```

Fig. 1. The SERIAL-RB algorithm (here p' denotes the position of a search node in the left-to-right ordering of the node and its siblings).

[5,15,6,14,21]. However the major focus has been on improving the asymptotic worst-case behavior of algorithms. The practical aspects of the possibility of exploiting parallel infrastructures have received much less attention.

Most existing exact algorithms for NP-hard graph problems follow the well-known branch-and-reduce paradigm. A branch-and-reduce algorithm searches the complete solution space of a given problem for an optimal solution. Simple enumeration is usually prohibitively expensive due to the exponentially increasing number of potential solutions. To prune parts of the solution space, an algorithm uses reduction rules derived from bounds on the function to be optimized and the value of the current best solution. The reader is referred to Woeginger's excellent survey paper on exact algorithms for further details [25]. At the implementation level, branch-and-reduce algorithms translate to search-tree-based recursive backtracking algorithms. The search tree size usually grows exponentially with either the size of the input instance n or some integer parameter k when the problem is fixed-parameter tractable [11].

Nevertheless, search trees are good candidates for parallel decomposition. While most divide-and-conquer methods for parallel algorithms aim at partitioning a problem instance among the cores, we partition the search space of the problem instead. Given c cores or processing elements, a brute-force parallel solution would divide a search tree into c subtrees and assign each subtree to a separate core for sequential processing. One might hope to thus reduce the overall running time by a factor of c. However, this intuitive approach suffers from several drawbacks, including the obvious lack of load balancing.

Even though our focus is on NP-hard graph problems, we note that recursive backtracking is a widely-used technique for solving a very long list of practical problems. This justifies the need for a general strategy to simplify the migration from serial to parallel algorithms. One example of a successful parallel framework for solving different types of problems is MapReduce [8]. The success of the MapReduce model can be attributed to its simplicity, transparency, and scalability, all of which are properties essential for any efficient parallel algorithm. In this paper, we propose a simple, lightweight, scalable approach for transforming almost any recursive backtracking algorithm into a parallel one with minimal communication overhead and a load balancing strategy that is implicit, i.e., does not require any problem-specific knowledge. The key idea behind our approach is the use of efficient traversal operations on an indexed search tree that is oblivious to the problem being solved. To test our approach, we implement parallel exact algorithms for the well-known VERTEX COVER and DOMINATING SET problems. Experimental results show that for sufficiently hard instances, we obtain nearly linear speedups on at least 32,768 cores.

2. Preliminaries

Typically, a recursive backtracking algorithm exhaustively explores a search tree T using depth-first search traversal. Each node of T (a *search node*) maintains some data structures required for completing the search. We denote a search node by $N_{d,p}$, where d is the depth of $N_{d,p}$ in T and p is the position of $N_{d,p}$ in the left-to-right ordering of all search nodes at depth d. The root of T is thus $N_{0,0}$. We use $T(N_{d,p})$ to denote the subtree rooted at node $N_{d,p}$. We say T has *branching factor* b if every search node has at most b children. A generic serial recursive backtracking algorithm, SERIAL-RB, is given in Fig. 1.

As an example, consider the problem of finding a minimum set of vertices $S \subset V$ of a graph $G = (V, E)$ such that the graph induced by $V \setminus S$ is a forest, i.e. a graph with no cycles. A possible implementation of SERIAL-RB which solves this problem, also known as the MINIMUM FEEDBACK VERTEX SET problem, is as follows. Every search node maintains a graph $G' = (V', E')$ and a solution set S'. We use $N_{d,p}(G')$ and $N_{d,p}(S')$ to denote the graph and the solution set at node $N_{d,p}$, respectively. At $N_{0,0}$, we have $N_{0,0}(G') = G$ and $N_{0,0}(S') = \emptyset$. The ISSOLUTION ($N_{d,p}$) function returns true whenever the graph induced by $N_{d,p}(V') \setminus N_{d,p}(S')$ is a forest and $|N_{d,p}(S')| < |best\_so\_far(S')|$, i.e the size of the smallest solution found so far. The ISLEAF ($N_{d,p}$) function returns true when the current branch cannot lead to any better solutions (e.g., whenever $|N_{d,p}(S')| \geq |best\_so\_far(S')|$). Finally, to generate the children of a search node, we simply find a cycle in $N_{d,p}(G')$ and for each vertex v in that cycle we get a new search node $N_{d+1,p'}$, where $N_{d+1,p'}(S') = N_{d,p}(S') \cup \{v\}$ and $N_{d+1,p'}(G')$ is obtained by deleting v and all the edges incident on v from $N_{d,p}(G')$. In terms of exact algorithms [25], GETNEXTCHILD corresponds to the implementation of *branching rules* and ISLEAF implements *pruning rules*. If we let G be a graph consisting of two triangles sharing an edge, then Fig. 2 shows one possible search tree generated by the described algorithm. Even though $N_{1,1}(G')$ is not acyclic, the children of $N_{1,1}$ will be pruned. This follows from the fact that, in a serial execution, $N_{1,0}(S')$ is a solution of size one and hence ISLEAF ($N_{1,1}$) would return true.

The goal of this paper is to transform SERIAL-RB into a scalable parallel algorithm with as little effort as possible. For ease of presentation, we make the following assumptions:

- SERIAL-RB solves an NP-hard optimization problem (i.e. minimization or maximization) where each solution appears in a leaf of the search tree.
- The global variable *best_so_far* stores the best solution found so far.
- The ISSOLUTION ($N_{d,p}$) function returns true only if $N_{d,p}$ contains a solution which is "better" than *best_so_far*.
- The search tree explored by SERIAL-RB is binary (i.e. every search node has at most two children).

In Section 4.4, we discuss how the same techniques can be easily adapted to any search tree with arbitrary branching factor. The only (minor) requirement we impose is that the number of children of a search node can be calculated on-the-fly and that generating those children (using GETNEXTCHILD ($N_{d,p}$)) follows a deterministic