J. Parallel Distrib. Comput. ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

Solving combinatorial problems using a parallel framework

Tarek Menouer

Paris-Nanterre University, LIP6 Laboratory, CNRS UMR 7606, Paris, France

HIGHLIGHTS

- Multiple pools of nodes shared between threads.
- A new computation model based on combining node and tree oriented parallelization.
- Hybridization of the two solutions.
- Good results are obtained for solving combinatorial problems using IBobpp framework.

ARTICLE INFO

Article history:
Received 19 September 2016
Received in revised form 17 May 2017
Accepted 24 May 2017
Available online xxxx

Keywords:
Combinatorial problems
Search algorithms
Parallelism
Cluster

ABSTRACT

This paper presents a new IBobpp framework which is an improvement of a high level parallel programming framework called Bobpp to optimize the performance of solving combinatorial problems. The Bobpp parallel computation model is as the majority of parallel models proposed in the context of tree search algorithms with *node oriented parallelization*, meaning that at every step of the algorithm, each thread gets one node from a unique global pool of non-explored nodes, generates the child nodes, and reinserts them into the pool to be explored later. This classical model has two drawbacks. First, the use of many threads creates a bottleneck problem. Second, grabbing a node causes memory contention problem when many nodes are generated and inserted into the same pool. To solve these problems, IBobpp framework proposes three solutions. The first consists of using multiple pools of nodes shared between all threads. The second solution consists of using a new computation model. The third solution consists of hybridization of the two previous solutions. Preliminary result shows that IBobpp gives a good result using the third solution.

1. Introduction

The parallelization of exact methods used to solve combinatorial problems have been widely studied in the literature, including but not limited to the work presented in [13,21,22,24,38,39,45]. These problems belong to the *NP-Hard* complexity class of problems. In worst case scenarios, they require exponential time to be exactly solved. Therefore, it is logical to reduce the computation time using parallel computers.

Several parallel versions of the family of tree search algorithms are proposed in the literature, such as the parallelization of Divide and Conquer (D&C), Branch and Bound (B&B), Branch and Price (B&P), Branch and Cut (B&C) and Constraint programming (CP).[1] There also exist many software frameworks which have been proposed to support the rapid development of tree search algorithms. Some of them are purely sequential, whereas others are parallel. For instance, the following works represent some of this software family: *BCP* [43], *PEBBL* [18], *PICO* [17], *ALPS* [47], *Bobpp* [23], *PUBB* [44], *PPBB* [46]. For parallel CP solvers, several works are presented in [9,25,40]. We refer as example Gecode [37] and ILOG Parallel Solver [40].

All these frameworks propose an application programming interface based on an abstract type for a tree search node and an abstract type for a solution. The tree search algorithm implemented in each framework is based on these abstract node and solution types. The application side defines, on the one hand, the concrete node and solution type and, on the other hand, a function (or method) that performs the search on one level. This means that the function is used in each level of the search tree to expand one node and generate new other nodes or solution nodes.

In the context of parallel machines, the biggest advantage of this type of application interface is that it generates as many tasks as possible. All generated nodes are inserted into a centralized or distributed global pool. The different threads or processes that execute the generation function can grab nodes from the pool when needed. This computation model is called *node oriented parallelization* that yields fine-grained parallelization. The global

E-mail address: tarek.menouer@lip6.fr.

[1] Constraint programming is sometimes called Branch and Infer.

http://dx.doi.org/10.1016/j.jpdc.2017.05.019
0743-7315/

pool can be considered as a search scheduler, and thus it has the responsibility of load balancing between threads. The *node oriented parallelization* model is used by several frameworks which support the rapid development of tree search algorithm, as Bobpp [23] framework. However, this computation model, and specially in Bobpp framework, has the following two disadvantages [5]. First, the use of many threads creates a bottleneck problem when all threads want to take nodes at the same time. Second, grabbing a node causes memory contention problem when many nodes are generated and inserted into the same pool stored in the shared memory. These two behaviors stress the memory bus and reduce the overall performances of the parallel algorithm.

In this paper we propose a new framework called IBobpp which is an improvement of Bobpp framework. The code of IBobpp framework is based on Bobpp. However, the goal of IBobpp is to minimize the Bobpp disadvantages by proposing three solutions. The first one is to use several pools of nodes shared between all threads rather than one pool. The second solution consists in proposing a new computation model with coarse-grained parallelization. This new model locally performs a large part of the search using recursive search, in order to reduce the number of accesses to the global memory due to node deletion or memory allocation. The principle is to start the search by applying the *node oriented parallelization model* for the first level of the search tree to generate new child nodes and insert them in the global pool of nodes, while the rest of the search is performed on each CPU using a previous *tree oriented parallelization model*. This previous model is implemented in Bobpp [33], the search is done locally by recursive search. Bobpp is used as a master tool that coordinates the sequential solvers to complete the parallel search and to perform the load balancing using work stealing. This latter is a popular load balancing technique where each parallel thread keeps track of its own work and occasionally steals work from the other threads to keep itself busy [10]. The third proposed solution by IBobpp is another computation model which combines the two previous solutions. The principle of this model is very simple; It consists of performing the *combining node and tree oriented parallelization model* with several pools of nodes.

The rest of this document is structured as follows. Section 2 presents related work and some combinatorial optimization frameworks proposed in the literature. Section 3 describes the Bobpp framework design and the methods used to parallelize the exploration of the search space using shared and distributed memory architectures. Section 4 presents the new IBobpp framework. Section 5 shows some experiments using the IBobpp framework. Finally, Section 6 contains the concluding remarks and provides directions for future work.

2. Related work

Several frameworks for solving combinatorial problems have been proposed in the literature, including as example *BCP* [43], *PEBBL* [18], *PICO* [17], *ALPS* [47], *PUBB* [44], *PPBB* [46]. All of these frameworks can be classified according to two major criteria:

1. The **node search algorithm** involved in the search process. These algorithms include Divide and Conquer (D&C), Branch and Bound (B&B) and its derived algorithms, Branch and Price (B&P), Branch and Cut (B&C), etc.
2. The **parallel programming environment** used to implement the parallelization: PThreads, OpenMP, MPI, PVM, etc.

For instance, the framework *PPBB* [46] proposes a Branch and Bound interface parallelized on a distributed architecture using PVM. *BCP* [43] is an implementation of the Branch-and-Price-and-Cut algorithm, which runs only with MPI.

Some other frameworks diversify their proposed functionalities. For example, *SYMPHONY* [42] solves Mixed-Integer Programming (MIP) problems using PVM for distributed memory machines or OpenMP for shared memory machines. *ALPS* [47], which is in some ways a successor to *SYMPHONY* and *BCP*, it generalizes the node search to any search tree including the Branch and Bound search among others. However, the only available programming environment for *ALPS* is MPI [47]. In a similar manner, *PEBBL* [18] integrates the Branch and Bound search from *PICO* [17], allowing the implementation of a larger variety of solvers than MIP solvers. For information, *PEBBL* and *PICO* are part of the ACRO project [1], while *ALPS*, *BCP*, *SYMPHONY* are part of the COIN-OR project [12].

As presented in Table 1, many of the available parallel search algorithm frameworks are specialized with respect to the implemented algorithm and the parallel programming environment. However, the novelty of the parallel IBobpp and Bobpp frameworks compared to other frameworks is that it provides several search algorithms classes while being able to use different parallelization methods. The aim is to propose a single framework for the most of classes of combinatorial problems, which can be solved on as many different parallel architectures as possible.

In the literature, the search space generated by Branch and Bound, Branch and Cut, Branch and Price and Divide and Conquer algorithms is similar to the search space of the algorithms used in Constraint Programming (CP). CP algorithms are used to solve combinatorial problems of great complexity, such as scheduling problems [3]. Several studies have been realized to parallelize CP search space, including the studies presented in [9,25,40]. In the literature, there are also several parallel CP solvers, such as Gecode [37], Parallel COMET [35] and ILOG Parallel Solver [40].

In the parallelization context, the majority of parallel frameworks or CP solvers are based on a unique global pool of nodes used as a search scheduling with a work stealing technique [10]. The use of a unique pool has two problems: bottleneck which can be created using many threads and memory contention which can be created by grabbing nodes. To solve these problems, we mention the following studies.

Michel et al. [34] propose to use a static parallelization without communication between threads. The principle is to generate and insert a fixed number of tasks in a unique global pool, then assign each task to one thread. The goal is that each thread works in its task locally without looking for new tasks in the global pool. The drawback of this static parallelization is that when a thread finishes its work, it waits until a solution is found. To address this problem, Evtushenko et al. [20] propose to use two pools of tasks, one global shared between all threads and a second local inside each thread. During N iterations, each thread generates new tasks and saves them in the local pool. At the end of the N iterations, threads transfer part of their tasks to the global pool. When the local pool is empty, the threads pick up tasks from the global pool. To limit the number of generated nodes, authors propose to insert a new node in the global pool only if there exists at last one thread blocked and waiting for a new node. The search ends when all threads are blocked and the global pool is empty. Furthermore, Fischetti et al. [36] propose to save a set of tasks in a global pool, then assign each task to one core according to a deterministic function. To limit the number of generated tasks, authors propose to generate a new task only if the thread detects that it works on a difficult task using an estimation function. There are other studies based on work stealing to assure a good load balancing [8,10]. We cite as example the study proposed by Anderson et al. [2]. It is based on a master–slaves approach. The master consists to distribute tasks to all slaves. When a slave completes its computation, it sends the results to the central master which provides it with a new task. The master has also the responsibility of work units sharing, communication of the best solution and termination detection.

Please cite this article in press as: T. Menouer, Solving combinatorial problems using a parallel framework, J. Parallel Distrib. Comput. (2017), http://dx.doi.org/10.1016/j.jpdc.2017.05.019.