ALGORITHMIC SKELETONS FOR BRANCH & BOUND

Michael Poldner
University of Münster, Department of Information Systems
Leonardo Campus 3, D-48149 Münster, Germany
Email: poldner@wi.uni-muenster.de

Herbert Kuchen
University of Münster, Department of Information Systems
Leonardo Campus 3, D-48149 Münster, Germany
Email: kuchen@uni-muenster.de

Keywords: Parallel Computing, Algorithmic Skeletons, Branch & Bound, Load Distribution, Termination Detection.

Abstract: Algorithmic skeletons are predefined components for parallel programming. We will present a skeleton for branch & bound problems for MIMD machines with distributed memory. This skeleton is based on a distributed work pool. We discuss two variants, one with supply-driven work distribution and one with demand-driven work distribution. This approach is compared to a simple branch & bound skeleton with a centralized work pool, which has been used in a previous version of our skeleton library Muesli. Based on experimental results for two example applications, namely the n-puzzle and the traveling salesman problem, we show that the distributed work pool is clearly better and enables good runtimes and in particular scalability. Moreover, we discuss some implementation aspects such as termination detection as well as overlapping computation and communication.

1 INTRODUCTION

Today, parallel programming of MIMD machines with distributed memory is mostly based on message-passing libraries such as MPI (W. Gropp, 1999; MPI, 2006). The resulting low programming level is error-prone and time consuming. Thus, many approaches have been suggested, which provide a higher level of abstraction and an easier program development. One such approach is based on so-called *algorithmic skeletons* (Cole, 1989; Cole, 2006), i.e. typical patterns for parallel programming which are often offered to the user as higher-order functions. By providing application-specific parameters to these functions, the user can adapt an application independent skeleton to the considered parallel application. (S)he does not have to worry about low-level implementation details such as sending and receiving messages. Since the skeletons are efficiently implemented, the resulting parallel application can be almost as efficient as one based on low-level message passing.

Algorithmic skeletons can be roughly divided into data parallel and task parallel ones. Data-parallel skeletons (see e.g. (R. Bisseling, 2005; G. H. Botorog, 1996; G. H. Botorog, 1998; H. Kuchen, 1994; Kuchen, 2002; Kuchen, 2004)) process a distributed data structure such as a distributed array or matrix as a whole, e.g. by applying a function to every element or by rotating or permuting its elements. Task-parallel skeletons (A. Benoit, 2005; Cole, 2004; Hofstedt, 1998; H. Kuchen, 2002; Kuchen, 2002; Kuchen, 2004; Pelagatti, 2003) construct a system of processes communicating via streams of data. Such a system is mostly generated by nesting typical building blocks such as farms and pipelines. In the present paper, we will focus on a particular task-parallel skeleton, namely a branch & bound skeleton.

Branch & bound (G.L. Nemhauser, 1999) is a well-known and frequently applied approach to solve certain optimization problems, among them integer and mixed-integer linear optimization problems (G.L. Nemhauser, 1999) and the well-known traveling salesman problem (J.D.C. Little, 1963). Many practically important but NP-hard planning problems can be formulated as (mixed) integer optimization problems, e.g. production planning, crew scheduling, and vehicle routing. Branch & bound is often the only practically successful approach to solve these problems exactly. In the sequel we will assume without loss of generality that an optimization problem consists of finding a solution value which minimizes an objective function while observing a system of constraints. The main idea of branch & bound is the fol-

lowing. A problem is recursively divided into subproblems and lower bounds for the optimal solution of each subproblem are computed. If a solution of a (sub)problem is found, it is also a solution of the overall problem. Then, all other subproblems can be discarded, whose corresponding lower bounds are greater than the value of the solution. Subproblems with smaller lower bounds still have to considered recursively.

Only little related work on algorithmic skeletons for branch & bound can be found in the literature (E. Alba, 2002; F. Almeida, 2001; I. Dorta, 2003; Hofstedt, 1998). However, in the corresponding literature there is no discussion of different designs. The MaLLBa implementation is based on a master/worker scheme and it uses a central queue (rather than a heap) for storing problems. The master distributes problems to workers and receives their solutions and generated subproblems. On a shared memory machine this approach can work well. We will show in the sequel that a master/worker approach is less suited to handle branch & bound problems on distributed memory machines. In a previous version of the Muesli skeleton library, a branch & bound skeleton with a centralized work pool has bee used, too (H. Kuchen, 2002). Hofstedt outlines a B&B skeleton with a distributed work pool. Here, work is only shared, if a local work pool is empty. Thus, worthwhile problems are not propagated quickly and their investigation is concentrated on a few workers only.

The rest of this paper is structured as follows. In Section 2, we recall, how branch & bound algorithms can be used to solve optimization problems. In Section 3, we introduce different designs of branch & bound skeletons in the framework of the skeleton library Muesli (Kuchen, 2002; Kuchen, 2004; Kuchen, 2006). After describing the simple centralized design considered in (H. Kuchen, 2002), we will focus on a design with a distributed work pool. Section 4 contains experimental results demonstrating the strengths and weaknesses of the different designs. In Section 5, we conclude and point out future work.

2 BRANCH & BOUND

Branch & bound algorithms are general methods used for solving difficult combinatorial optimization problems. In this section, we illustrate the main principles of branch & bound algorithms using the 8-puzzle, a simplified version of the well-known 15-puzzle (Quinn, 1994), as example. A branch & bound algorithm searches the complete solution space of a given problem for the best solution. Due to the exponentially increasing number of feasible solutions, their explicit enumeration is often impossible in practice. However, the knowledge about the currently best solution, which is called *incumbent*, and the use of *bounds* for the function to be optimized enables the algorithm to search parts of the solution space only implicitly. During the solution process, a pool of yet unexplored subsets of the solution space, called the *work pool*, describes the current status of the search. Initially there is only one subset, namely the complete solution space, and the best solution found so far is infinity. The unexplored subsets are represented as nodes in a dynamically generated search tree, which initially only contains the root, and each iteration of the branch & bound algorithm processes one such node. This tree is called the *state-space tree*. Each node in the state-space tree has associated data, called its *description*, which can be used to determine, whether it represents a *solution* and whether it has any successors. A branch & bound problem is solved by applying a small set of basic rules. While the signature of these rules is always the same, the concrete formulation of the rules is problem dependent. Starting from a given initial problem, subproblems with pairwise disjoint state spaces are generated using an appropriate *branching rule*. A generated subproblem can be estimated applying a *bounding rule*. Using a *selection rule*, the subproblem to be branched from next is chosen from the work pool. Last but not least subproblems with non-optimal or inadmissible solutions can be eliminated during the computation using an *elimination rule*. The sequence of the application of these rules may vary according to the strategy chosen for selecting the next node to process (J. Clausen, 1999). As an example of the branch and bound technique, consider the 8-puzzle (Quinn, 1994). Figure 1 illustrates the goal state of the 8-puzzle and the first three levels of the state-space tree.

The 8-puzzle consists of eight tiles, numbered 1 through 8, arranged on a 3×3 board. Eight positions on the board contain exactly one tile and the remaining position is empty. The objective of the puzzle is to repeatedly fill the hole with a tile adjacent to it in horizontal or vertical direction, until the tiles are in row major order. The aim is to solve the puzzle in the least number of moves.

The branching rule describes, how to split a problem represented by a given initial board into subproblems represented by the boards resulting after all valid moves. A minimum number of tile moves needed to solve the puzzle can be estimated by adding the number of tile moves made so far to the Manhattan distance between the current position of each tile and its goal position. The computation of this lower bound is described by the bounding rule.

The state-space tree represents all possible boards that can be reached from the initial board. One way to solve this puzzle is to pursue a breadth first search or a depth first search of the state-space tree until the

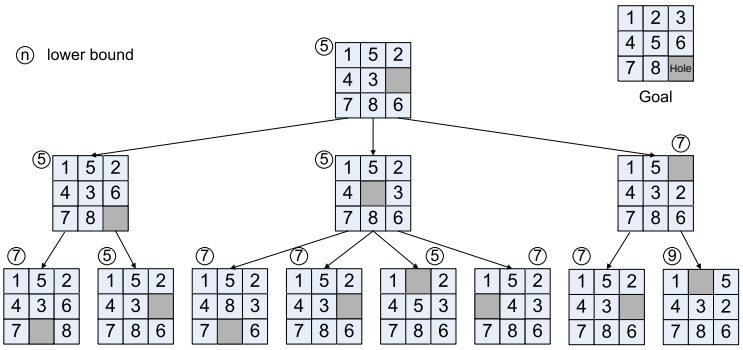

Figure 1: Upper part of the state-space tree corresponding to an instance of the 8-puzzle and its goal board.

sorted board is discovered. However, we can often reach the goal faster by selecting the node with the best lower bound to branch from. This selection rule corresponds to a best-first search strategy. Other selection rules such as a variant of depth-first search are discussed in (J. Clausen, 1999; Y. Shinano, 1995; Y. Shinano, 1997).

Branch & bound algorithms can be parallelized at a low or at a high level. In case of a low-level parallelization, the sequential algorithm is taken as a starting point and just the computation of the lower bound, the selection of the subproblem to branch from next, and/or the application of the elimination rule are performed by several processes in a data parallel way. The overall behavior of such a parallel algorithm resembles of the sequential algorithm.

In case of a high-level parallelization, the effects and consequences of the parallelism are not restricted to a particular part of the algorithm, but influence the algorithm as a whole. Several iterations of the main loop are performed in a task-parallel way, such that the state-space tree is explored in a different (non-deterministic!) order than in the sequential algorithm.

3 BRANCH & BOUND SKELETONS

In this section, we will consider different implementation and design issues of branch & bound skeletons. For the most interesting distributed design, several work distribution strategies are discussed and compared with respect to scalability, overhead, and performance. Moreover, a corresponding termination detection algorithm is presented.

A B&B skeleton is based on one or more branch & bound algorithms and offers them to the user as predefined parallel components. Parallel branch & bound algorithms can be classified depending on the organization of the work pool. A central, distributed, and hybrid organization can be distinguished. In the MaLLBa project, a central work pool is used (F. Almeida, 2001; I. Dorta, 2003). Hofstedt (Hofstedt, 1998) sketches a distributed scheme, where work is only delegated, if a local work pool is empty. Shinano et al. (Y. Shinano, 1995; Y. Shinano, 1997) and Xu et al. (Y. Xu, 2005) describe hybrid approaches. A more detailed classification can be found in (Trienekens, 1990), where also complete and partial knowledge bases, different strategies for the use of knowledge and the division of work as well as the chosen synchronicity of processes are distinguished.

Moreover, different selection rules can be fixed. Here, we use the classical best-first strategy. Let us mention that this can be used to simulate other strategies such as the depth-first approach suggested by Clausen and Perregaard (J. Clausen, 1999). The bounding function just has to depend on the depth in the state-space tree.

We will consider the skeletons in the context of the skeleton library Muesli (Kuchen, 2002; Kuchen, 2004; Kuchen, 2006). Muesli is based on MPI (W. Gropp, 1999; MPI, 2006) internally in order to inherit its platform independence.

3.1 Design with a Centralized Work Pool Manager

The simplest approach is a kind of the master/worker design as depicted in Figure 2. The work pool is maintained by the master, which distributes problems to the workers and receives solutions and subproblems from them. The approach taken in a previous version of the skeleton library Muesli is based on this centralized design. When a worker receives a problem, it either solves it or decomposes it into subproblems and computes a lower bound for each of the subproblems. The work pool is organized as a heap, and the subproblem with the best lower bound at the time is stored in its root. Idle workers are served with new problems taken from the root. This selection