Parallel Branch-and-Bound: Design and Performance Understanding *

Wagner Meira, Jr.
Dept. Computer Science
University of Rochester
Rochester, NY, USA

Annibal Sodero Andréa Tavares Márcio Carvalho
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, MG, Brazil

{meira,annibal,iabrudi,mlbc}@dcc.ufmg.br

Abstract

Branch-and-Bound techniques have been successfully used to solve combinatorial optimization problems. One common approach to improve the effectiveness of these techniques is via parallelization. The parallelization of Branch-and-Bound computations, however, is not trivial and programmers may experience difficulties both in terms of correctness and efficiency of the parallelized applications. In this paper we present an environment that helps programmers in developing efficient parallel Branch-and-bound applications. This environment integrates two tools: (1) Sabor, which aids in designing those applications, and (2) Carnival, which is a performance measurement and analysis tool that helps the programmer in understanding the performance of those applications. We also present the Carnival user interface and illustrate its usefulness and functionality by identifying and explaining sources of overhead in example applications.

Resumo

Técnicas de *Branch-and-Bound* têm sido usadas com sucesso para a solução de problemas de otimização combinatória. Essas técnicas podem se tornar ainda mais eficientes quando paralelizadas. A paralelização da computação associada a técnicas *Branch-and-Bound*, entretanto, não é trivial e programadores podem ter dificuldades tanto em termos de correção quanto eficiência das paralelizações resultantes. Neste trabalho apresentamos um ambiente que auxilia programadores no desenvolvimento de aplicações paralelas de *Branch-and-Bound* que sejam eficientes. Esse ambiente integra duas ferramentas: (1) Sabor, que auxilia no desenvolvimento daquelas aplicações e (2) Carnival, uma ferramenta de análise e medição de desempenho que provê ao programador recursos para o entendimento do desempenho daquelas aplicações. Também apresentamos a interface da Carnival e ilustramos sua utilidade e funcionalidade através da identificação e análise das fontes de degradação de desempenho em aplicações.

*This work is partially supported by CNPq-PERT Grant No. 490.039/95-2 and Grant No. 200.862/93-6

1 Introduction

There are many combinatorial optimization problems that cannot be solved in polynomial time. In order to solve these problems within reasonable time, we use techniques that aim to find optimal solutions while minimizing the number of solutions investigated.

Branch-and-Bound (B&B) [4] is the most successfull of these techniques and has been applied to solve problems such as the Traveling Salesman, Knapsack, Vertex Covering, and Integer Programming. A B&B algorithm partitions the search space recursively into smaller sub-spaces until it is possible to determine a solution or the unfeasibility of a possible solution. A B&B algorithm is characterized by three rules: (1) branching, (2) walking, and (3) bounding. The branching rule specifies how a problem is partitioned into subproblems. During this branching process, the bounding rule determines the subproblems that will not generate an optimal solution and can be discarded. The walking rule determines the order of expansion of subproblems. The algorithm ends when there is no more subproblems to expand. Note that branching and bounding rules are problem dependent, while walking rule is algorithm dependent. B&B computations are usually very intensive and there are two basic ways of reducing the execution time of B&B algorithms [3, 6]: (1) by improving the effectiveness of bounding rules, what requires deep knowledge about the problem and (2) by parallelizing efficiently the computation, what requires the user to have expertise in parallel programming.

In this paper we present an environment that facilitates the implementation of efficient parallel B&B programs. This environment results from the integration of two tools: Sabor [8] and Carnival [7]. Sabor is a tool that provides infrastructure for the implementation of parallel B&B applications. Carnival is a performance measurement and analysis tool that automates the process of understanding the causes of idle times in parallel programs, which are called *waiting times*. This integrated environment helps a programmer in all phases of the development of parallel B&B applications, from the design of the algorithm to the analysis and understanding the performance of the parallelizations.

We describe Sabor in the next section. Section 3 describes Carnival and Waiting Time Analysis. We then present the integration details and describe the Carnival user interface in Sections 4 and 5, respectively. Section 6 presents some examples of the utilization of the environment in understanding parallel B&B implementations for the TSP.

2 Sabor

Sabor [8] is a system that aids in designing and analyzing of the distributed B&B algorithms. It is motivated by the fact that the development of optimization applications is a task complex by itself, and the parallelization of these applications can be overwhelming to a programmer. Sabor aims at releasing programmers from the difficulties that arise in the process of parallelizing an application, both in terms of efficiency and correctness. In Sabor, the optimization software developer is not aware of the specific parallelization mechanisms. He implements application-dependent subroutines (i.e., branching and bounding rules) and the system provides to him

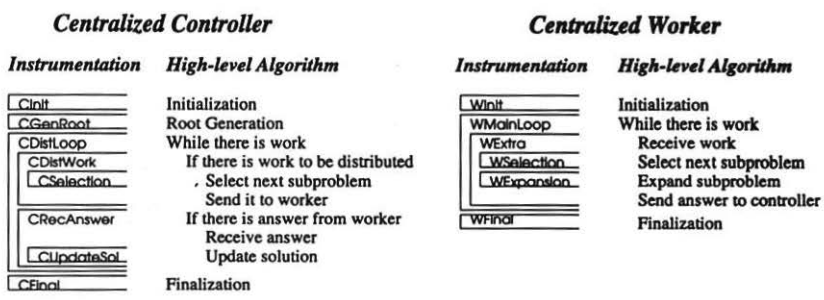

Figure 1: High-level algorithms of Centralized mode

various approaches (i.e., walking rules) to execute the algorithm in parallel.

Sabor is composed by two major subsystems: optimization and visualization. The optimization system is concerned with the implementation of the algorithms. It is composed by a collection of classes, which define the framework needed to generate Branch-and-Bound algorithms. These classes implement parallelization modes (distributed and centralized), walking strategies, and also provide a template for application specific classes, which are the only piece of code that must be implemented by the user. The visualization subsystem provides an interface with facilities for configuring, compiling, and running applications. It also provides some basic performance visualization that includes graphical and alphanumerical information about the amount of work performed by each processor and the coverage of the search space.

Sabor parallelizes B&B computation by adopting a *controller–worker* model, where the controller starts the workers and coordinates the execution of the algorithm. Sabor provides two parallel operation modes: centralized and distributed. In the centralized mode, the controller process implements a version of the sequential B&B algorithm, which is modified so that the expansion tasks are divided among the workers. During each iteration of the algorithm, the controller, instead of expansion node on its own, sends it to a worker and continues its execution, keeping track of worker's responses. Each worker task consists of receiving work (i.e., nodes to be expanded), performing the expansion, and returning the resulting *children* to the controller. The centralized mode is characterized by an even distribution of work among the workers, since they rely on a central queue of subproblems to be expanded, but at a high communication cost, caused by the successive synchronizations. The high-level algorithms of the controller and worker of the centralized mode are presented in Figure 1. Sabor provides three selection schemes that define the order of work distribution in centralized implementations: (1) best-first, (2) depth-first, and (3) breadth-first. Best-first selection chooses subproblems that have the best bounding values, what usually reduces the number of expansions necessary to find an optimal solution. Depth-first approaches select for expansion the problem-state of greatest depth in the tree. On the other hand, breadth-first strategies cause the expansion to be performed on a level by level basis.

In the distributed operation mode, the controller is responsible for the initial distribution of workload between the workers and for the termination control. Each worker implements a modified version of the sequential B&B, which is augmented

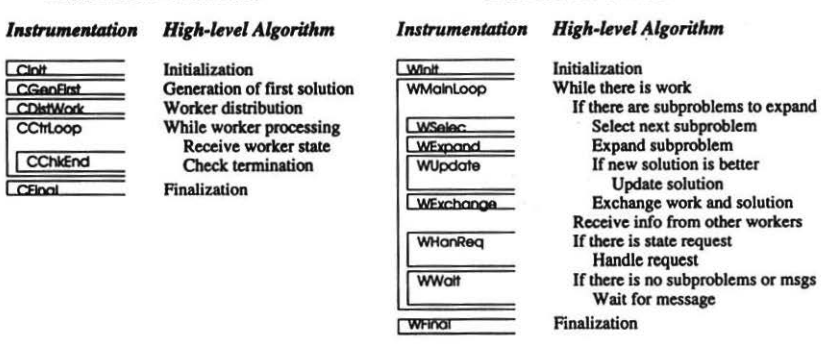

Figure 2: High-level algorithms of Distributed mode

with procedures to exchange work and new solutions. The amount of communication in distributed implementations is smaller than in centralized implementations, but the overall program performance is usually affected by load imbalance among the workers. The high-level algorithms of the controller and worker of the distributed mode are presented in Figure 2. Sabor provides some balancing strategies for distributed implementations such as static distribution (i.e., each worker receives a set of subproblems generated by the first expansion) and randomized distribution [1], which redistributes subproblems generated in the last expansion by sending them to randomly-chosen workers.

In choosing a parallelization scheme, the programmer must consider the granularity of the application in the execution environment. We define granularity as the ratio between computation and communication costs in the target execution environment. Thus, in execution environments with relatively low communication costs such as shared-memory machines, the granularity of the applications tends to increase (i.e., the computation performed by B&B computations is more significant when compared to the cost of communication operations). Fine-grain applications are more suitable for distributed approaches, where the occurrence of load imbalance affects less the application and communication costs are kept low. On the other hand, coarse-grain applications are more suitable for centralized approaches, since synchronizations are less frequent and the computation is well-divided among the processors. Note that variations in either the execution environment or the application affect this notion of granularity. Thus, in determining the best parallelization scheme the programmer needs tools and techniques that help him in assessing the granularity of the application.

3 Waiting Time Analysis and Carnival

Many of the overheads associated with parallelization ultimately manifest themselves as *waiting time* (WT); a processor is idle while it waits for another or more. Waiting time can be introduced at any synchronization point, including locks, barriers, and message exchanges. We can define (both symbolically and quantitatively) the *cause* of waiting time between two processors to be the differences between the execution

paths followed by the processors since the last time the two processors synchronized and one waited for the other, hence both processors were known to be at the same place at the same time. [1] We assume that all processors execute instructions at the same rate, and therefore attribute the difference in time required to reach the synchronization point to differences in the instructions that were executed.

In order to understand the cause of waiting time between two processors at a particular synchronization point in the program, we compare the execution paths of the two processors between that synchronization point and the last point at which those processors were known to be at the same place at the same time (as recorded in an event trace), and determine why one path is longer than the other (thereby causing the waiting time) [7]. Anything the two paths have in common is removed as a potential cause of waiting time, leaving only the differences between the two paths as an explanation for waiting time, which is called a *characterization*. Since a synchronization statement may be executed multiple times, there may be several such characterizations for each source of waiting time, corresponding to alternative execution paths. Taken together, these characterizations are the *cause* of waiting time at one particular synchronization point in the program.

Carnival is a performance measurement and visualization tool for message-passing programs that automates the cause-and-effect inference process for waiting time. The tool uses detailed event traces to gather performance information, which it presents both as global summary statistics and as localized performance profiles, facilitating top-down performance analysis. The user interface presents performance information together with the source code, creating a link between the observed phenomena and the code. Most important, Carnival supports *waiting time analysis*, an automatic inference process that *explains* each source of waiting time, instead of simply identifying where it occurs. Carnival is under development at the University of Rochester and the initial implementation is targeted at data-parallel applications running on distributed-memory machines using message-passing for communication. Carnival runs on IBM SP2, SGI Challenge [2], and networks of workstations. As described in Section 4, Carnival was integrated with Sabor without major changes, what illustrates the generality and applicability of the tool.

4 Integrating Sabor and Carnival

The integration requires changes in both tools. Sabor must be instrumented to generate static (i.e., program structure) and dynamic (i.e., execution events) information about the programs. Carnival must present information about the program in a more abstract level than source code [3], since the presentation of all source code embedded in a program generated by Sabor can be very confusing to the programmer, who is not aware of implementation details and of the whole code structure.

[1] Our discussion is in terms of pair-wise synchronization, but the techniques extend to the analysis of waiting time for synchronization operations between many processors. In particular, we characterize waiting time in barriers as a set of pair-wise synchronization operations between each waiting processor and the last processor to enter the barrier.

[2] Although the SGI Challenge is a DSM, we can use it as a message-passing machine by employing packages such as PVM.

[3] The initial implementation of Carnival handles only real source code.

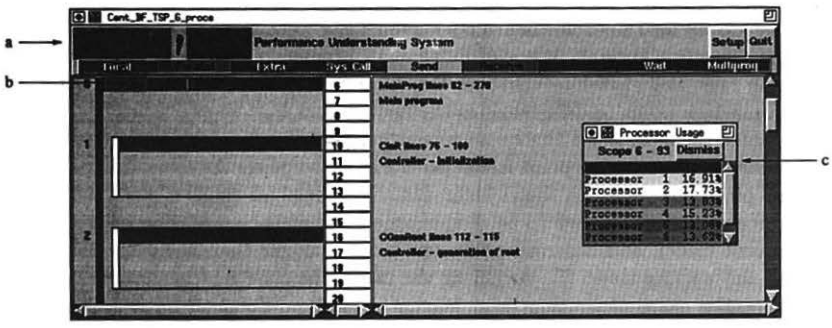

Figure 3: Carnival visualization of B&B TSP

We instrumented Sabor by inserting *instrumentation directives*, which are applied right before the code compilation. These directives are inserted in pairs, creating scopes where the specified action happens. Also, these instrumentation scopes can be nested and are distinguished by unique identifiers. Note that this approach facilitates the extraction and handling of static information about code that is shared among implementations. Figures 1 and 2 show the scopes generated by the instrumentation at the left of each high-level algorithm. These scopes and their description are presented in the visual interface of Carnival.

5 Visualizing Performance with Carnival

In this section we present the visual interface of Carnival and discuss how it can be used in understanding the performance of parallel B&B applications generated by Sabor.

The primary Carnival display window (Figure 3a) is divided into two parts. The structure of the source code is presented on the right; information about each instrumentation scope appears on the left. The line numbers are presented in a grey scale, where the intensity of the scale represents the percentage of execution time (summed up across all processors) spent on a given portion of source code. Users can quickly identify places in the code where most of the time is spent by scrolling down the line numbers and looking for the darkest portion of the scale. The left side of the display identifies the scopes in the program. As described in Section 4, scopes are delimited by instrumentation directives, and have unique identifiers[4]. A grey scale bar is drawn from the beginning to the end of every scope; the intensity of the scale indicates the cumulative execution time (across all processors) spent in that scope. Unlike the grey scale used for line numbers, this scale includes the time spent in nested scopes; therefore, the outermost scope always has the darkest bar. Clicking on the vertical bar for a scope yields the per-processor percentage of the scope's execution time in a pop-up window (Figure 3c). The colors in the horizontal bar at the top of each scope describe a breakdown of the execution time spent in the scope into categories. Clicking on the bar produces a histogram of the overhead

[4]The names inside the scopes in Figures 1 and 2 are identifiers.

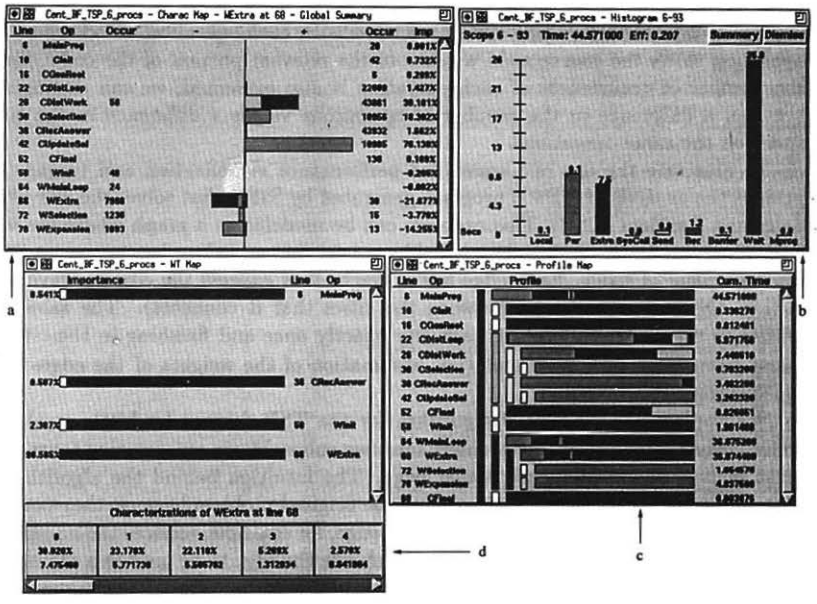

Figure 4: Carnival visualization of B&B TSP

categories in a pop-up window (Figure 4b). In order to facilitate the navigation in the main window, a Profile Map (Figure 4c) is also provided. This map presents the horizontal colored bar and grey scale vertical bars in a condensed form with cumulative time spent in each scope, its location, and its processing category. By using this Profile Map, the user can easily identify the important scopes of the application, and access them by clicking on the corresponding horizontal bar of the map, what causes the main window to be scrolled to that scope.

Two additional pop-up windows are used for waiting time analysis. The **WT Map** (Figure 4d) provides a global perspective of all sources of waiting time; the **Characterization Map** (Figure 4a) presents an explanation for a single source of waiting time in terms of the two execution paths involved. The WT Map lists each source of waiting time, the line number where the waiting occurred, the scope identifier, and the percentage of the total waiting time associated with that operation. This map is used to navigate within the source code window and to initiate waiting time analysis. Clicking on an entry in the WT Map causes the main display window to be shifted to the relevant portion of the source code, and the WT Map presents statistics about each cause of that waiting time. These statistics include the percentage contribution of each cause to the total waiting time experienced at that statement, as well as the total waiting time explained by each cause. Clicking on a characterization in the WT Map produces an explanation for that waiting time in the **Characterization Map**. Color-coded operations for the longer of the two paths are presented on the right side of the window, operations for the shorter path are on the left. The number of occurrences of each operation is given, as is

the percentage of the waiting time associated with each operation. Clicking on an operation shifts the source code window to the relevant portion of the code. Since the number of occurrences of each operation is also presented, we can distinguish between a difference in the number of operations versus a difference in the time spent on the same operations.

To illustrate the use of Carnival for performance visualization and tuning, we present the analysis of a B&B program generated by Sabor that solves the Traveling Salesman Problem (TSP). This problem can be modeled as a graph where the vertices represent the n cities to be visited by a salesman and the edges represent links between pairs of cities. Each edge has a weight that express the cost of traversing it (i.e., the cost of traveling between the cities that it connects). The salesman wishes to make a tour, visiting each city exactly once and finishing in the city he starts from, with minimum cost (i.e., summation of the weights of the edges that are traversed in the tour).

We implemented the B&B algorithm for the TSP devised by Little et al. [5], which breaks each unsolvable problem into two subproblems representing tours that must include or exclude a particular edge. The intuition behind the algorithm is that subproblems are easier to solve than the original problem because they contain additional constraints. The exclusion of an edge, for example, reduces the number of edges that may be added to a solution. We choose the edge to be used as a constraint so that the lower bound on the cost of the solution of the subproblem excluding that edge is maximized. We executed a centralized best-first implementation on a 35-city problem using seven processors (six workers and one controller) on the SGI Challenge. The Carnival visualization of the execution is shown in Figures 3 and 4.

The primary display window (Figure 3a) shows the structure of the source code on the right and execution time profiles for each nested scope on the left. In this example, the colored bar associated with the outermost scope (Figure 3b) shows a significant amount of waiting time (the dark blue portion of the bar). Clicking on the colored bar for that scope produces the pop-up window with a color histogram for each category of execution time, and the processor efficiency for that scope (Figure 4b). As seen in the figure, efficiency is only 20%, and waiting time accounts for more than half of the total cumulative execution time.

The WT Map (Figure 4d) shows that there are four sources of waiting time, but most of WT (96%) occurs while receiving work (identified by WExtra). Each of these sources of waiting time is explained by one or more characterizations. We can iterate over the characterizations for a specific source of waiting time (in the order defined by their relative contribution) by clicking on the colored bar in the WT Map, which produces an explanation in the Characterization Map. Also, we can obtain a global summary of the characterizations by clicking on the grey-scale square at the left of the color bar. In our example, the global summary of the characterizations of WExtra (Figure 4a) shows that the WT is caused by cost differences between code executed in the controller and the worker. More specifically, the selection (WSelection) and expansion (WExpansion) costs are smaller than the costs of distributing work (CDistWork) and updating solutions (CUpdateSol). Thus, we can conclude that there is contention in obtaining work from the controller, since workers spend significant time waiting (57% of the cumulative execution time) while the controller is distributing work for other processes and doing local work.

B&B Imp.	Cent BF			Cent DF			Dist Static			Dist Rand		
Workers	2	4	6	2	4	6	2	4	6	2	4	6
RunTime	9.1	6.8	6.4	9.4	6.9	7.6	20.7	18.7	19.1	6.5	7.3	6.4
CRunTime	27.2	33.9	43.7	28.1	34.6	53.5	62.0	93.6	133.9	19.4	36.6	45.0
CComp.	11.4	8.8	9.4	11.5	10.7	11.1	16.1	29.9	42.8	5.8	7.3	8.6
CExtra	8.4	9.7	8.9	8.4	9.7	11.6	11.8	22.5	33.5	7.0	11.4	14.0
CWaiting	7.3	15.4	25.4	8.2	14.2	30.7	34.1	41.2	57.6	6.6	17.9	22.5

Table 1: Execution profiles of B&B TSP – 35 cities (Seconds)

6 Understanding the performance of TSP

In this section, we compare and understand four parallel B&B implementations that solve the TSP, with two executing in centralized mode and two executing in distributed mode. In our example, the centralized implementations adopt different selection strategies, namely: best-first and depth-first. In the distributed implementations we adopted two different balancing strategies: static and randomized.

These four implementations were executed on a SGI Challenge with 2, 4, and 6 workers. Execution time results[5] are presented in the Table 1. We can see that the distributed static implementation is always slower than the other implementations, and the centralized best-first implementation outperforms the centralized depth-first in all cases (by checking RunTime). Also, some configurations presented a detrimental anomaly [2], such as the the distributed randomized implementation that took longer to execute using four workers instead of two. The same table also presents the breakdown of cumulative execution times (CRunTime) into computation (CComp), extra computation (CExtra) introduced by the parallelization, and waiting time (CWaiting). The amount of computation performed in centralized approaches is roughly the same, despite the number of workers employed, which confirms the intuition of a better work balancing in these approaches. On the other hand, we can observe an increase in the timings of all categories in the distributed approaches, that clearly shows waste of computation because of lack of global knowledge about partial solutions. Also, waiting time is significant in all implementations, accounting from 27% to 58% of the cumulative execution time.

By checking the characterizations provided by Carnival, we first noticed that the characterizations for implementations that have the same operation mode (i.e., centralized, distributed) are very similar. In the centralized approaches, WT arises in the workers because of contention in getting work from the controller, as described in Section 5. On the other hand, WT in distributed implementations is explained by work imbalance among workers. For instance, the distributed randomized implementation with 4 workers, where waiting time accounts for about half of the cumulative execution time, has five main sources of WT, but two of them account for more than 96% of the total WT in the program (CCtrLoop and WWait). The characterizations for both sources are similar and show that waiting time arises because of differences in the execution times for expanding (WExpand), selecting (WSelec), and exchanging work (WExchange). Note that the Waiting Time Analysis not only determines the causes of waiting time but also quantify the importance of these causes, and thus guide the user in improving his applications and using parallel resources efficiently.

[5]Note that these timings may vary significantly for different problems, although the observed behaviors will be similar for the various configurations.

7 Conclusions and Future Work

In this paper we presented an environment that helps programmers in implementing, analyzing, and understanding the performance of parallel Branch-and-Bound applications. The environment has been shown to be a valuable resource, since it not only relieves the user from the burden of parallelizing B&B computations, but also helps him in identifying and understanding the performance characteristics of implementations, as illustrated by the examples presented.

There are many future directions of work. We plan to continue the development of the environment by designing techniques that link characterizations to implementation decisions (e.g., selection or balancing strategy). Also, we are implementing other B&B applications that will be used to validate the integrated environment. This experience will serve as a basis for the implementation of similar environments targeted to other classes of applications, such as neural networks.

Finally, we would like to thank Tom LeBlanc, Alex Poulos, and Cláudio Amorim for many critiques, discussions, insights, and suggestions on the Carnival work. Also, we wish to thank the Computer Science Department of the University of Rochester for the use of the SGI Challenge.

References

[1] R. M. Karp and Y. Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the Association for Computing Machinery*, 40(3):765–789, July 1993.

[2] T. Lai and S. Sahni. Anomalies in parallel branch-and-bound algorithms. *Cmmunications of the ACM*, 27(6):594–602, June 1984.

[3] P. S. Laursen. Simple approaches to parallel branch and bound. *Parallel Computing*, 19:143–152, 1993.

[4] E. L. Lawler and D. E. Wood. Branch-and-bound methods: a survey. *Operations Research*, 14:699–719, 1966.

[5] J. D. C. Little, K. G. Murty, D. W. Sweeney, and C. Karel. An algorithm to the traveling salesman problem. *Operations Research*, 11(6):972–989, November–December 1963.

[6] G. P. McKeown, V. J. Rayward-Smith, and S. A. Rush. *Advances in Parallel Algorithms*, chapter 5 – Parallel Branch-and-bound, pages 111–150. John Willey and Sons, Inc., 1992.

[7] W. Meira Jr., T. LeBlanc, and A. Poulos. Waiting time analysis and performance visualization in carnival. In *Proc. of SIGMETRICS Symposium on Parallel and Distributed Tools*, Philadelphia,PA, May 1996. ACM.

[8] A. I. Tavares. Um sistema para implementação e análise de técnicas de branch-and-bound em redes de estações de trabalho. Master's thesis, Universidade Federal de Minas Gerais, Belo Horizonte, MG, Brazil, Fevereiro 1995.