
ELSEVIER

Parallel Computing 26 (2000) 27–46

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Branch-and-bound parallelization strategies applied to a depot location and container fleet management problem

Benoît Bourbeau[a], Teodor Gabriel Crainic[a,b], Bernard Gendron[a,c,*]

[a] *Centre de Recherche sur les Transports, Université de Montréal, Succursale Centre-Ville, C.P. 6128, Montréal, Québec, Canada H3C 3J7*

[b] *Département des sciences administratives, Université du Québec à Montréal, Québec, Canada*

[c] *Département d'informatique et de recherche opérationnelle, Québec Canada*

Received 1 June 1998; received in revised form 1 May 1999

Abstract

This paper presents branch-and-bound parallelization strategies applied to the location/allocation problem with balancing requirements. This formulation is representative of a larger class of discrete network design and location problems arising in many transportation logistics and telecommunications applications: it displays a multicommodity network flow structure and a complex objective function comprising fixed and variable flow costs. As for many problems of this class, the bounding procedure embedded in the branch-and-bound algorithm is complex and time-consuming. The parallelization strategies that we describe are designed to exploit this feature. Parallelism is obtained by dividing the search tree among processors and performing operations on several subproblems simultaneously. The strategies differ in the way they manage the list of subproblems and control the search. We report and analyze experimental results, on a distributed network of workstations, which aim to compare different implementations of these strategies.

Keywords: Parallel optimization; Branch-and-bound; Multicommodity location with balancing requirements

* Corresponding author.
E-mail address: bernard@crt.umontreal.ca (B. Gendron).

0167-8191/00/$ - see front matter
PII: S0167-8191(99)00094-0

1. Introduction

Network design and location models are often called upon to represent complex issues arising in transportation, logistics, telecommunications, and production planning applications. These models are usually formulated as large-scale, combinatorial mixed integer programs with complex constraint structures [15,18,20]. Branch-and-bound (B&B) is currently the only general tool available for finding optimal solutions to these difficult formulations. Yet, even with the help of efficient specialized algorithms that compute tight bounds on their optimum values, realistically formulated and dimensioned models require the exploration of such huge search trees that optimal solutions cannot be found but for the simplest instances. Parallel computing increasingly appears as an efficient and practical way to address this issue.

The objective of this paper is to present and experimentally compare general parallelization strategies adapted to typical B&B algorithms for this class of problems. Parallelism is obtained by dividing the search tree among processors and performing operations on several subproblems simultaneously. The strategies differ in the way they manage the list of subproblems and control the search. We apply the strategies to a representative formulation, the location/allocation problem with balancing requirements [7]. Similar to many other discrete network design problems, this formulation exhibits a multicommodity network flow structure and a complex objective function comprising fixed and variable flow costs. Furthermore, the bounding procedures embedded into the currently best known B&B algorithm for this problem [13] are complex and time-consuming.

Parallel B&B algorithms have been the object of an abundant literature; see, e.g., the 1994 survey by Gendron and Crainic [12]. In this survey, the authors mention only a few references that implement and compare various strategies of storage and control (notable examples are [19,21,22]). Yet, such studies are essential for our understanding of parallel B&B and its behavior on various problem classes. The present paper adds to the growing body of literature dedicated to such comparisons (recent examples are Refs. [3,9]), with a particular focus on location/network design formulations.

We present a framework for implementing three general strategies, which can be used to parallelize most B&B algorithms specialized to location/network design problems. The strategies are implemented on a distributed network of workstations and computational experiments are performed on several instances of the location/allocation problem with balancing requirements. Previous attempts at exploiting parallelism in B&B algorithms to solve this problem can be found in [11,14] and two of the strategies presented in this paper are indeed inspired by these developments. We introduce, however, a number of improvements and refinements over the methods described in these papers. Moreover, it is the first time that these parallelization strategies are compared.

The paper is organized as follows. Section 2 presents the parallelization strategies, while Section 3 briefly recalls the location/allocation problem with balancing requirements and the main features of the sequential B&B algorithm designed to solve

it. The implementation details and the computational experiments are the subject of Section 4. The conclusion summarizes our work and proposes extensions.

2. Parallelization strategies

In the following, we give a general outline of each of the parallel B&B strategies analyzed in this paper. The presentation is general and independent of the particular problem class. For simplicity of exposition, we assume a message-passing architecture, but the strategies can also be adapted to shared-memory machines. The actual implementations are detailed in Section 4. To fix the vocabulary, the section starts with a brief presentation of the B&B paradigm.

B&B algorithms may be seen as implicit enumeration methods for solving optimization problems P: $Z(P) = \min_{x \in S} f(x)$, where $f : S \to R$, $S \subseteq R^n$, and it is assumed P can be solved by enumerating a finite number of points (not necessarily known in advance) in S. The B&B strategy is often cast as the construction of a *tree* (variously called B&B, enumeration or exploration tree). The root of the tree is the original problem, while the sons of any given node (subproblem) Q are the subproblems obtained by decomposition of Q. The decomposition operation, called *branching*, partitions the feasible domain of a given problem into a number of smaller subsets on which the same optimization problem is defined. Problems are recursively decomposed until either a solution may be easily found, or one determines that further partitioning is unnecessary because the original problem has been solved, or the subproblems resulting from decomposition are infeasible, or one may prove that further branching cannot improve the current best known solution. To determine these conditions, a *bounding* operation is performed that yields a *lower bound* on the optimal value of the subproblem and, sometimes, an *upper bound* on the optimal value of P (when a feasible solution to the original problem is identified). Elimination, or *fathoming*, rules use these bounds to determine when further decomposition of a subproblem is unnecessary (when it is the case, such a subproblem is called a leaf). A subproblem Q may thus be *generated* when it has been obtained from another subproblem by branching, *evaluated* when a bounding operation has been applied to it, or *examined* when it has been either decomposed by branching or eliminated by fathoming. Sequential B&B algorithms thus consist of performing branching and bounding operations, as well as testing the fathoming rules. The order according to which subproblems are examined is determined by *selection* criteria that thus determines how the tree is explored; *best-first* (select the subproblem with the lowest lower bound) and *depth-first* (select the most recently generated subproblem) are the most used paradigms.

Parallelization strategies for B&B algorithms may be classified according to a large number of computer architecture, algorithm design, and implementation characteristics. In their survey, Gendron and Crainic [12] identify three main classes of parallel B&B algorithms according to the degree of parallelization of the search tree. Type 1 strategies correspond to the decomposition of node operations (e.g., bounding computations). In type 2 approaches, the search tree is built in parallel

(e.g., processes work on several subproblems simultaneously). When type 3 strategies are applied, several trees are explored concurrently. The strategies presented and analyzed in this paper belong to the type 2 class, as do most other parallel B&B methods found in the literature.

Implementations of type 2 may be further classified according to their *synchronization of communications* and *number of work pools*. Synchronous algorithms impose strict rules on communications, while in asynchronous algorithms, communications may occur at any time and depend only on the local algorithmic logic of the interacting processes. The strategies presented in this paper belong to the asynchronous type. We favor asynchronous approaches because, in our opinion, they offer more efficient parallelizations: allocation of work is not delayed while waiting for the next synchronization moment, and essential information, such as upper bound values, is distributed in a timely manner.

A *work pool* is a memory location where processes pick up and store their units of work. Typically, a process picks up a subproblem in a work pool and examines it. When it finishes its action, the process stores the subproblems not yet examined in the same or in a different work pool. *Single* pool algorithms make use of only one memory location, whereas in *multiple* pool implementations there are several memory locations where processes find and store subproblems yet to be examined. In this paper, we analyze three strategies: (1) *centralized*: all subproblems are kept in a single pool; (2) *decentralized*: a collegial multiple pool strategy where each process that participates in the exploration of the tree has one dedicated pool; (3) *hybrid*: a mixed multiple pool approach where each process has an associated work pool, but there is also a global work pool, shared by all processes.

The three strategies share a number of features. All are built for a fixed number of processes created when the algorithm is initiated. Although the allocation of tasks to processes is different for each approach, there is always a process, identified as *coordinator*, which facilitates the inter-process exchanges of information, particularly regarding the equitable distribution of work loads, and is in charge of detecting the end of the computations. The three strategies also share the same initialization procedure: a sequential exploration of the tree, that starts at the root node and proceeds until either K nodes are evaluated, or up to the identification of a first leaf. In Section 4, we show that such an initialization phase, if well calibrated, helps decrease the so-called *search penalty* (defined as the number of nodes treated by the parallel algorithm in excess of the sequential one).

The centralized strategy is implemented as a classical *master–slave* approach. The master process, which is also the coordinator, executes the B&B search, specifies the tasks to be executed by the other processes, controls the pool of subproblems to be examined, and determines the end of the computations (an empty pool and all processes idle). Worker processes perform bounding and branching operations. Upon completion of this task, workers return the results – the new subproblems or a fathoming message – to the master process. When a worker improves the upper bound, it broadcasts it to the master and all the other workers. The master selects subproblems from the pool and allocates them to idle workers according to a simple round-robin strategy. The selection of subproblems may be based on one of the

usual sequential criteria (e.g., depth-first, best-first), or on a hierarchical combination thereof (e.g., among depth-first solutions with equal values, select the one with the lowest lower bound) expressed as the subproblem *priority*.

In this single-pool strategy, the control of the search is centralized at the level of the master process. In the decentralized and hybrid strategies, however, the control of the search is distributed among the individual processes, each executing a B&B on the subproblems in its own pool. Consequently, differences may appear among the amount of work each process has to undertake. One then has to introduce mechanisms to detect this situation and correct it by transferring work between the appropriate processes. These *load balancing strategies*, are based on the notion of *workload*, $L(\Omega)$, defined as an estimation of the effort required to examine the subproblems in pool Ω (see [9], and the references therein, for further discussion on this notion).

The decentralized approach implements a totally distributed memory organization: each process manages its own pool of unexamined subproblems and executes a complete B&B on these subproblems. The control of the search is thus collectively assumed by the individual processes that exchange upper bound information as well as subproblems when the workloads are unbalanced. To facilitate the load balancing operations, relevant information is kept by the coordinator. Load balancing is performed according to a dynamic, on-request strategy that aims to match processes with almost empty pools to highly loaded ones, while avoiding that processes become idle waiting for work. Processes with workloads superior to a $L_{\min}$ threshold are said to be highly charged and potential suppliers of work to other processes; we identify them as *granting* processes.

The load balancing strategy proceeds as follows. Each process evaluates its workload $L(\Omega)$ each time a subproblem is inserted or deleted from the pool and either *informs* the coordinator of the status of its workload, or *requests* work if it has reached or fallen under an alert threshold L_a. It is thus hoped that work will be transferred before all the remaining subproblems in the pool are examined and disposed of. Following the reception of a request for work, the coordinator selects a granting process and issues a transfer proposal. This selection is based on the workload values stored in the coordinator memory and may either pick up the most heavily loaded process (rule HL), or proceed according to a round-robin scheme on the heavily loaded processes (rule RR). On receiving the transfer proposal, the process verifies that its workload is superior to $L_{\min}$ indicating that it may still satisfy it; otherwise, it sends back a refusal message. In the latter case, another process has to be selected by the coordinator. When a process accepts the transfer proposal, it selects a number N of subproblems from its pool to send to the requesting process (this load balancing strategy is similar to the one proposed by Gendron and Crainic [14], but it introduces two new parameters, L_a and N, as well as a new rule, HL, to identify the granting process). The search is completed when all processes have work requests pending.

The hybrid strategy is close to the decentralized approach, but it confers a more important role to the coordinator process. The fundamental idea is that looking for a suitable process to satisfy a work request might be too time consuming, especially

when the first selected process cannot accept the transfer proposal. It would then be more efficient to endow the coordinator with its own pool of subproblems, out of which it could directly satisfy incoming work requests. In a hybrid strategy, subproblem exchanges thus proceed between processes, each implementing the sequential B&B, and the coordinator. On the one hand, on request, the coordinator transfers a number of subproblems to the requesting process. Similar to the centralized strategy, the selection is performed according to the subproblem priorities. On the other hand, heavily loaded processes transfer some of their unexamined subproblems to the coordinator pool. The selection of these subproblems is guided by a dual principle that states that one should not send "bad" problems (to avoid generate unnecessary work) and that the exploration of the tree should aim to mimic that of the original sequential method. Thus, one identifies and transfers so-called *second quality* subproblems, e.g., subproblems that show promise but which would have been examined at a later moment in a sequential exploration (recall that these subproblems are sent to the coordinator work pool and that they are to be examined later). Thus, for dichotomous branching and depth-first exploration, e.g., second quality subproblems could be the one not selected for immediate examination. The search is completed when all processes have work requests pending and the coordinator pool is empty.

3. Problem formulation and sequential algorithm

To test our parallelization strategies, we chose a representative location/network design formulation, the multicommodity location/allocation problem with balancing requirements (MLB). This class of models is motivated by applications related to the management of heterogeneous fleets of empty containers by international maritime shipping companies. The general goal is to locate depots for empty containers in order to collect the supplies available at customers' sites and to satisfy customer requests, while minimizing the total operating costs. These costs comprise fixed costs for opening and operating the depots, and transportation costs generated by customer-depot traffic and by inter-depot movements. These inter-depot balancing flows differentiate the problem from classical location/allocation applications. For further details on the problem description, the reader is referred to Crainic et al. [7].

Consider a directed network $G = (N, A)$, where N is the set of nodes and A is the set of arcs. There are several *commodities* (*types of containers*), represented by set P, which move through the network. The set of nodes may be partitioned into three subsets: O, the set of *origin* nodes (*supply customers*); D, the set of *destination* nodes (*demand customers*); and T, the set of *transshipment* nodes (*depots*). For each depot $j \in T$, we define $O(j) = \{i \in O : (i, j) \in A\}$ and $D(j) = \{i \in D : (j, i) \in A\}$, the sets of customers adjacent to this depot, and assume that $O(j) \cup D(j) \neq \emptyset$. For each node $i \in N$, we define the sets of depots adjacent to this node in both directions: $T^+(i) = \{j \in T : (i, j) \in A\}$, and $T^-(i) = \{j \in T : (j, i) \in A\}$. Since it is assumed that there are no arcs between customers, the set of arcs may be partitioned into three subsets: customer-to-depot, $A_{OT} = \{(i, j) \in A : i \in O,\ j \in T\}$; depot-to-customer,

$A_{TD} = \{(i,j) \in A : i \in T,\ j \in D\}$; and depot-to-depot, $A_{TT} = \{(i,j) \in A : i \in T,\ j \in T\}$. For each customer $i \in O$, the supply of commodity p is noted o_i^p, while for each customer $i \in D$, the demand for commodity p is noted d_i^p. All supplies and demands are assumed to be non-negative and deterministic. A non-negative cost c_{ij}^p is incurred for each unit of flow of commodity p moving on arc (i,j). In addition, for each depot $j \in T$, a non-negative fixed cost f_j is incurred if the depot is opened.

Let x_{ij}^p represent the amount of flow of commodity p moving on arc (i,j), and y_j be the binary location variable that assumes value 1 if depot j is opened and 0 otherwise. The problem is then formulated as

$$Z = \min\left\{\sum_{j\in T} f_j y_j + \sum_{p\in P}\left(\sum_{(i,j)\in A_{OT}} c_{ij}^p x_{ij}^p + \sum_{(j,i)\in A_{TD}} c_{ji}^p x_{ji}^p + \sum_{(j,k)\in A_{TT}} c_{jk}^p x_{jk}^p\right)\right\}, \tag{1}$$

subject to

$$\sum_{j\in T^+(i)} x_{ij}^p = o_i^p \quad \forall i \in O,\ p \in P, \tag{2}$$

$$\sum_{j\in T^-(i)} x_{ji}^p = d_i^p \quad \forall i \in D,\ p \in P, \tag{3}$$

$$\sum_{i\in D(j)} x_{ji}^p + \sum_{k\in T^+(j)} x_{jk}^p - \sum_{i\in O(j)} x_{ij}^p - \sum_{k\in T^-(j)} x_{kj}^p = 0 \quad \forall j \in T,\ p \in P, \tag{4}$$

$$x_{ij}^p \leqslant o_i^p y_j \quad \forall j \in T,\ i \in O(j),\ p \in P, \tag{5}$$

$$x_{ji}^p \leqslant d_i^p y_j \quad \forall j \in T,\ i \in D(j),\ p \in P, \tag{6}$$

$$x_{ij}^p \geqslant 0 \quad \forall (i,j) \in A,\ p \in P, \tag{7}$$

$$y_j \in \{0,1\} \quad \forall j \in T. \tag{8}$$

Eqs. (2) and (3) ensure that supply and demand requirements are met, relations (4) correspond to flow conservation constraints at depot sites, while constraints (5) and (6) forbid customer-related movements through closed depots. Note that analogous constraints for the inter-depot flows are redundant since inter-depot costs satisfy the triangle inequality [7].

Two Lagrangian relaxations of the MLB may be used in order to efficiently compute tight lower bounds on Z. In general, a Lagrangian relaxation approach removes some constraints from the formulation and introduces them into the objective function by weighting their violations with so-called Lagrangian multipliers. One Lagrangian relaxation can be obtained by removing constraints (5) and (6). This yields a *multicommodity uncapacitated minimum cost network flow problem* (MCNF), which decomposes into $|P|$ single-commodity uncapacitated minimum cost network flow problems, for which efficient solution techniques exist [1]. Another Lagrangian relaxation can be derived by removing constraints (4). One then obtains an

uncapacitated location problem (ULP) [6,17], for which one of the most efficient methods is the DUALOC algorithm proposed by Erlenkotter [10]. Embedded into DUALOC is a dual-ascent procedure that solves approximately the linear relaxation of the ULP and also derives primal solutions satisfying the integrality constraints. When solving the MCNF relaxation, values of the dual variables associated to constraints (4) can be obtained and then used to adjust the Lagrangian multipliers for the ULP relaxation. Similarly, when the linear relaxation of the ULP is solved approximately by the dual-ascent procedure of DUALOC, values for the dual variables associated to constraints (5) and (6) are obtained, which can then be used to adjust the Lagrangian multipliers for the MCNF relaxation. By iteratively solving the two relaxations, adjusting the Lagrangian multipliers of one relaxation to the values of the dual variables obtained when solving the other, one obtains a dual-ascent scheme that approximates the linear relaxation of the MLB. Within this iterative scheme, upper bounds can be easily computed by using the primal information generated when solving the relaxations. An upper bound is derived from the optimal solution of the MCNF relaxation by setting y_j to 1 whenever there is flow moving through depot j (to 0 otherwise). When the ULP relaxation is solved, an upper bound is computed by solving the MCNF obtained by fixing the y variables to the values of the best primal integral solution identified by the dual-ascent procedure of DUALOC.

Gendron and Crainic [13] present a bounding procedure based on this iterative scheme, which stops either when the relative gap between the lower and upper bounds is less than a user-supplied value ϵ_1, or when the relative difference between two successive lower bounds is less than a user-supplied value ϵ_2, or, finally, when the number of iterations has reached a limit $t_{\max}$. This procedure is embedded into a B&B algorithm that uses efficient branching rules and preprocessing tests to reduce the size of the tree. A dichotomous branching rule is designed to generate new subproblems. The *depth-first rule* is used to decide which generated subproblem should be examined in priority. This rule minimizes computer storage requirements and can be implemented efficiently using a last-in-first-out stack, although the total number of subproblems it generates might be large [16]. However, when, as in the present case, a good heuristic is used to compute effective upper bounds and smart branching rules are implemented to efficiently explore the tree, this disadvantage may be significantly reduced. Further details on the bounding procedure, as well as on the branching and preprocessing rules, can be found in [13].

4. Implementation and computational experiments

A number of choices were made in order to implement the three general strategies presented in Section 2. Some of these choices were motivated by the computer environment used – a message-passing, distributed architecture – while others seemed reasonable given the particular class of formulations for which the methods were intended.

The centralized strategy is implemented rather straightforwardly. The pool uses a heap data structure and the lower bound of the parent problem is used to try to

eliminate newly generated subproblems before their insertion in the pool. The priority used to select the next subproblem favors the deepest nodes, with ties broken by using, first, the lower bound value of the parent node and, second, the value of the variable fixed to generate the subproblem, in a way that imitates the order of exploration of the sequential tree. A number of other possible priority rules based on different orderings of the same criteria have been tested (see [4]), but clearly shown to be inferior because they increase the search penalty.

The sequential B&B of Section 3 makes use of a dichotomous branching operation. Thus in both decentralized and hybrid approaches, a process that performs a branching operation keeps one subproblem for immediate examination. The other one is inserted in the pool in the first strategy, while it is passed to the coordinator pool in the latter. Therefore, in the current implementation of the hybrid strategy, the working processes do not have to manage pools. Indeed, these are always empty, since the subproblems that are not immediately examined are considered of second quality and are sent to the coordinator. As for the management of the coordinator pool of the hybrid approach, it is similar to that used for the centralized strategy.

The workload associated with a pool in the decentralized strategy is computed as the number of subproblems in the pool. As mentioned above, two rules have been implemented to select the granting process in this approach. Rule HL selects the most heavily loaded process. In this case, a process informs the coordinator of its charge every time it has varied by more than a quantity ΔL compared to the last charge value sent. Rule RR selects the granting process through a round-robin scheme on the heavy loaded processes. For this approach, a process sends the value of its workload to the coordinator only when the new value modifies its status: from heavily to lightly charged or vice versa. The pools are implemented using stacks to ensure that the depth-first paradigm of the sequential procedure is followed.

The strategies have been calibrated and tested on a network of 16 Sun Ultra 1 workstations, each equipped with 64 Mb of RAM. The stations were linked by Fast-Ethernet (100 Mbits/s) technology and the network was dedicated to the tests (i.e., at the same time, no other jobs were running on any processor of the network). The bounding and branching procedures are programmed in *FORTRAN* and compiled with the *f77* compiler (*O4* optimization option). The parallel B&B strategies are written in *C* and compiled with the *gcc* compiler (*O3* optimization option). The PVM library was used to manage the message exchanges among the processes.

The objective of the computational experiments was to compare the relative performances of the three strategies with respect to:

1. the number of processors;
2. the problem *granularity*, which corresponds to the time spent in examining one subproblem (as there can be significant variations of the granularity for the same problem, average and standard deviation can be used to measure it more precisely);
3. the problem *size*, which is the number of examined subproblems in the sequential tree.

Our series of experiments were performed on 11 problem instances. Table 1 indicates for each problem instance the number of container types ($|P|$), customers

Table 1
Dimensions of test problems and sequential results

Problem	$\|P\|$	$\|O\|$	$\|T\|$	$\|A\|$	n	T_s
P1	1	219	44	7152	221	185
P2	2	219	44	7150	565	281
P3	2	219	44	7150	145	339
P4	1	219	44	7154	145	38
P5	2	220	43	7100	247	125
P6	1	220	43	7100	99	121
P7	12	289	130	4718	3681	3013
P8	12	289	130	4718	10 825	8146
P9	12	289	130	4718	3909	2690
P10	12	289	130	4718	9523	6794
P11	12	289	130	4718	3421	2395

($|O|$; note that $O = D$ for all instances), depots ($|T|$), arcs ($|A|$), the size of the sequential tree (n), and the elapsed time (in CPU seconds) required to explore this tree (T_s). The first six problems are randomly generated (the generator is described in [8]). These instances are relatively easy to solve, and therefore we did not expect significant savings by solving them in parallel. However, we have included them in our series of tests because similar instances have been used as testbeds in previous studies on the MLB. Problem P7 is from an actual application, while P8–P11 are obtained from P7 by perturbing all costs by a random factor chosen in the interval [0.8, 1.8]. In all experiments with these instances, the parameters of the bounding procedure were set to the following values: $\epsilon_1 = \epsilon_2 = 0.01$ and $t_{max} = 10$. Typically, between 1 and 6 iterations of the bounding procedure were performed for each subproblem (some two iterations, on average).

First, we have calibrated the parameters of the load balancing method of the decentralized strategy. In these experiments, we have used an initialization procedure that consists of a sequential exploration up to the identification of a first leaf, and have tested both the HL and RR rules. When testing the HL rule, the parameter ΔL is set to 1, so that the coordinator retains the most recently updated information about the workloads of the processes. We have used three measures to compare the various parameter settings: (1) *search penalty* P_S: ratio of the number of nodes generated during a parallel execution to the number of nodes explored by the sequential algorithm; (2) *load balancing factor* L_F: ratio of the minimum useful time to the maximum useful time, where the minimum and the maximum are taken over all working processes (here, the useful time of a working process is the total elapsed time minus the waiting and I/O times of the process during the parallel execution); (3) *speedup* S: ratio of the total elapsed time required by the sequential algorithm to the total elapsed time of the parallel execution.

Our experiments allowed us to conclude that $L_{min} = 1$ (i.e., a worker can share part of its work when its pool contains at least two subproblems) performs uniformly better than other values, irrespective of the particular settings of the other parameters. This can be easily explained. On the one hand, if $L_{min} = 0$, a worker with only

one node left in its pool might send it to answer a request, therefore becoming idle. On the other hand, if $L_{\min} \geqslant 2$, a worker might refuse to share work, while, in fact, at least one node would be available for immediate transfer. This conclusion is consistent with the observations reported by Gendron and Crainic [14], who also identified 1 as the best value for parameter $L_{\min}$.

Since the randomly generated problems (P1–P6) and the most realistic instances (P7–P11) are quite different with respect to both size and granularity, we have calibrated these two classes of instances separately. For each instance, every parameter setting was tested three times to account for variations due to the asynchronous nature of the communications (all experiments reported in the paper obey the same protocol). Performance measures averaged over the three runs have been used to compare the different parameter settings.

The calibration performed on problems P1–P6, reported in [4], allowed us to select $L_a = 2$ (i.e., a worker sends a request when it has two subproblems left to examine) and $N = 1$ (i.e., a process sends one subproblem each time it shares part of its pool) combined with the HL rule. In Table 2, we report the results of the calibration on problems P7–P11, averaged over all five instances. The table displays the three performance measures for each parameter setting tested, each rule, HL or RR, and for $p = 16$ processors (similar conclusions are obtained for other values of p).

We first observe that the load balancing method is quite robust, since its performances do not significantly vary when different parameter values are used (a similar observation also applies to problems P1–P6). We note that, as N increases, the load balancing factor generally increases. This is to be expected, since transferring as many nodes as possible (note that we always ensure that the load of the granting process never falls below $L_{\min}$) clearly improves load balancing. However, if N is too large, the chance of experiencing a search penalty increases, as illustrated by the values of P_S for $N = 4$. This is so because when early exchanges occur, the granting process will examine nodes closer to the root earlier than it would do in a

Table 2
Calibration of the load balancing strategy (P7–P11, $p = 16$)

L_a	N	HL			RR		
		P_S	L_F	S	P_S	L_F	S
1	1	1.019	0.908	12.14	1.029	0.886	11.93
1	2	1.016	0.914	11.89	1.046	0.920	11.91
1	3	1.020	0.914	12.22	1.019	0.924	12.34
1	4	1.459	0.941	10.81	1.247	0.947	11.31
2	1	1.016	0.861	11.86	1.028	0.796	11.16
2	2	1.070	0.948	12.04	1.056	0.927	12.00
2	3	1.018	0.937	12.45	1.024	0.911	12.24
2	4	1.457	0.943	10.44	1.210	0.934	11.36
3	1	1.017	0.875	12.21	1.034	0.769	11.13
3	2	1.063	0.912	11.98	1.064	0.867	11.62
3	3	1.143	0.941	11.58	1.008	0.920	12.62
3	4	1.433	0.920	10.81	1.194	0.942	11.39

sequential exploration; if the optimal value has not been found by then, this would result in a search penalty (assuming the sequential exploration is effective, as in the present case). We observe that $N = 2$ and $N = 3$ generally perform best, irrespective of the values of L_a and of the rule chosen to identify the granting process. The value of L_a is not particularly critical, although it is reasonable to have $L_a \simeq L_{min} + 1$, otherwise a processor might send a request for work too early, when in fact, it should be considered as a candidate for granting requests (this is why we have only tested $L_a = 1, 2, 3$). Rules HL and RR are competitive; indeed, significant differences in the speedups attained by the two rules often can only be explained by differences in the search penalty. After these tests, the following values were selected for the remaining experiments on problems P7–P11: $L_a = 3$ and $N = 3$, combined with rule RR.

Next, we have experimented with the most realistic instances (P7–P11) to calibrate the initialization procedure. Specifically, we have run the three strategies (C: centralized; D: decentralized; H: hybrid) by using four initialization approaches: $K = 1$, which corresponds to evaluate only the root node in the initialization phase; $K = p$ and $K = 2p$, for which the sequential initialization stops after p and $2p$ nodes have been evaluated; $K = \infty$, which is used to denote the initialization that stops once a first leaf is found. Table 3 shows the results obtained for $p = 16$ processors, where in addition to P_S and S, we display *Amdahl's speedup*, S_A, which is an estimation of the speedup derived from Amdahl's law [2]. More specifically, if p is the number of processors and $\widetilde{T}$ is the time spent in the sequential part of the algorithm, an estimate of the speedup is then obtained from the expression

$$S_A = p\left(\frac{T_s}{T_s + (p-1)\widetilde{T}}\right).$$

In our case, $\widetilde{T}$ corresponds to the time spent in the initialization phase, which, in the current implementation, is performed sequentially. The table reports values of S_A based on averaging over all runs.

These results show that it is not productive to start the parallel exploration too early because this increases the chance of experiencing a search penalty. This phenomenon occurs because the optimal value is not known in advance (although generally a good upper bound is computed at the root), but discovered after a while, once the depth-first search has explored deep regions of the tree. If the parallel search starts just after the root node has been evaluated, it will explore the tree in a breadth-first fashion, possibly expanding nodes that would have been fathomed in sequential;

Table 3
Comparisons of the initialization strategies (P7–P11, $p = 16$)

K	C		D		H		S_A
	P_S	S	P_S	S	P_S	S	
1	1.930	10.06	1.276	11.50	1.104	12.92	15.96
p	1.058	13.10	1.243	11.49	1.002	12.57	15.37
$2p$	0.990	13.11	1.125	12.02	1.006	12.47	14.81
∞	1.004	12.51	1.008	12.62	1.002	12.65	13.94

this is so because at the moment these nodes are examined by the sequential algorithm, the optimal value has already been found deep into the tree. Of course, such search penalties occur because the sequential search is effective, otherwise the parallel algorithm, by exploring the tree in a breadth-first fashion, could actually find the optimal value earlier than the sequential one, and consequently generate a smaller tree. This phenomenon is observed for some runs and some instances, especially when the centralized approach is used with $K = 2p$ (in this case, the average value of P_S is less than 1). However, the same setting of K is not as good when the decentralized strategy is used, in which case we observe a significant search penalty. The most conservative rule, which tends to imitate the sequential search, is to stop the initialization after the search has gone as deep as possible into the tree, i.e., up to the identification of a first leaf. In this case, we observe that the sequential and parallel trees have, on average, the same size, irrespective of the strategy used. Hence, this rule was selected for the remaining experiments.

The drawback of this approach is that the exploration is performed sequentially for a while, only one processor being used during the initialization phase. This disadvantage is clearly illustrated by the values of S_A, especially when $K = \infty$. The following parallel implementation of the initialization phase would remedy this problem: while one processor explores the tree up to K nodes or until a first leaf is found, other processors compute the bounds for the new nodes. Although it would clearly improve the efficiency of the initialization phase, we have not included such an implementation in our study because it would not impact the efficiency of the parallel search. In fact, our analysis, focused on the relative performances of the three parallelization strategies, is not affected by whether the initialization is performed in parallel or not.

We now present the main results of our experiments, which aim to compare the relative performances of the three strategies. In order to facilitate the analysis, we first show the results obtained for a fixed number of processors ($p = 16$). Table 4 displays for each problem instance and each strategy the four measures defined above, to which we add the *utilization factor*, U, which is the ratio of the minimum useful time to the total elapsed time.

We first note that the search penalties for problems P1–P6 are very close to 1, except for problem P5, for which P_S is still reasonably close to 1. This confirms the validity of our earlier observations concerning the initialization strategy. We also note that the sequential trees for these instances are small and thus the initial sequential exploration takes up a significant part of the parallel computation time, as illustrated by the values of S_A. Therefore, the maximum speedups are small. The situation is much better for larger instances (P7–P11), but still the serial time is significant.

To better understand the results displayed in Table 4, we observe that the implementation of each strategy may be characterized according to its particular *task decomposition*, where a task corresponds to a computation phase between two consecutive requests for work: for the centralized strategy, a task corresponds to the examination of one node; for the decentralized strategy, a task corresponds to the exploration of a partial subtree (partial, since some nodes can be sent to other

Table 4
Comparisons of the parallelization strategies ($p = 16$)

Problem		C	D	H
	P_S	1.109	1.145	1.086
P1	L_F (U)	0.871 (0.661)	0.521 (0.520)	0.601 (0.573)
	S (S_A)	3.86 (4.74)	3.69 (4.59)	3.48 (4.49)
	P_S	0.979	0.986	0.972
P2	L_F (U)	0.932 (0.841)	0.673 (0.670)	0.875 (0.802)
	S (S_A)	7.00 (7.95)	6.13 (7.32)	6.57 (7.73)
	P_S	0.945	1.000	0.945
P3	L_F (U)	0.544 (0.366)	0.163 (0.163)	0.350 (0.331)
	S (S_A)	1.87 (2.09)	1.82 (2.07)	1.74 (1.97)
	P_S	0.966	1.000	0.966
P4	L_F (U)	0.835 (0.563)	0.344 (0.335)	0.520 (0.509)
	S (S_A)	2.48 (3.14)	2.32 (3.03)	2.24 (2.87)
	P_S	1.287	1.235	1.283
P5	L_F (U)	0.890 (0.687)	0.379 (0.348)	0.819 (0.670)
	S (S_A)	3.62 (4.81)	3.35 (4.69)	3.42 (4.60)
	P_S	1.000	1.000	1.000
P6	L_F (U)	0.647 (0.629)	0.366 (0.353)	0.399 (0.388)
	S (S_A)	1.90 (2.17)	1.87 (2.13)	1.73 (1.99)
	P_S	1.024	1.028	1.019
P7	L_F (U)	0.977 (0.910)	0.896 (0.896)	0.986 (0.973)
	S (S_A)	12.90 (13.38)	13.03 (13.43)	13.55 (13.61)
	P_S	0.997	1.023	0.997
P8	L_F (U)	0.983 (0.965)	0.979 (0.972)	0.992 (0.991)
	S (S_A)	14.14 (14.95)	13.53 (14.92)	13.67 (14.88)
	P_S	0.994	0.987	0.986
P9	L_F (U)	0.972 (0.927)	0.922 (0.916)	0.953 (0.945)
	S (S_A)	11.58 (13.38)	12.15 (13.39)	11.85 (13.59)
	P_S	1.000	1.002	1.000
P10	L_F (U)	0.986 (0.969)	0.938 (0.936)	0.987 (0.979)
	S (S_A)	13.15 (14.75)	13.38 (14.76)	13.19 (14.82)
	P_S	1.004	1.010	1.007
P11	L_F (U)	0.974 (0.898)	0.865 (0.865)	0.913 (0.905)
	S (S_A)	10.77 (13.08)	11.02 (13.11)	10.99 (13.25)

processors); for the hybrid strategy, a task corresponds to the exploration of a branch of the tree (i.e., a simple path starting at one arbitrary node and ending at a leaf).

For randomly generated problems (P1–P6), we observe that a fine task granularity contributes to better balance the loads among processors. Indeed, the C strategy shows much better load balancing and utilization factors than the two others. This is so because, for problems of such a small size, not enough tasks are generated for the H and D strategies to be efficient, especially for the decentralized strategy which has the coarsest task granularity of the three. As a result, the speedups obtained by the C strategy are better than those displayed by the other strategies, but only slightly so. We also observe that the centralized strategy shows, for each instance, significant differences between the load balancing and utilization factors, which means that the

useful time corresponding to the most busy processor is significantly smaller that the total elapsed time. This indicates important waiting times resulting from the bottleneck created by simultaneous accesses to the pool. A similar tendency may be observed for the hybrid approach as well, albeit significantly less marked. For the D strategy, however, load balancing and utilization factors almost coincide.

For instances derived from the actual application (P7–P11), all strategies show much better load balancing and utilization factors compared to the other problems. This is so because problems P7–P11 generate larger trees; therefore, more tasks are available, which improves load balancing. This is especially true for the D and H strategies, the last one even showing better statistics in this respect than the C strategy. Still, the load balancing and utilization factors differ for the centralized approach, but these differences are much less significant than for the other problems. Finally, note that the speedups are comparable for the three strategies.

To analyze the evolution of performances with respect to the number of processors, the three strategies were run on 2, 4, 8 and 16 processors. We illustrate the results for the most difficult of the problems, P8. Figs. 1 and 2 show the variation of the load balancing factor and of the speedup, respectively. As illustrated in Fig. 1, the load balancing factor slowly decreases as p increases for the C strategy, as no discernible pattern emerges for the D and H approaches. The speedups for the three strategies are quasi-linear up to 16 processors (only the serial fraction imposes a penalty).

Although overall, the results of the three approaches are comparable, we can identify two factors that have a clear impact on the relative performances of the three strategies: (1) *the number of generated tasks*: when a limited number of tasks are created, the decentralized approach does not utilize the resources well (this follows

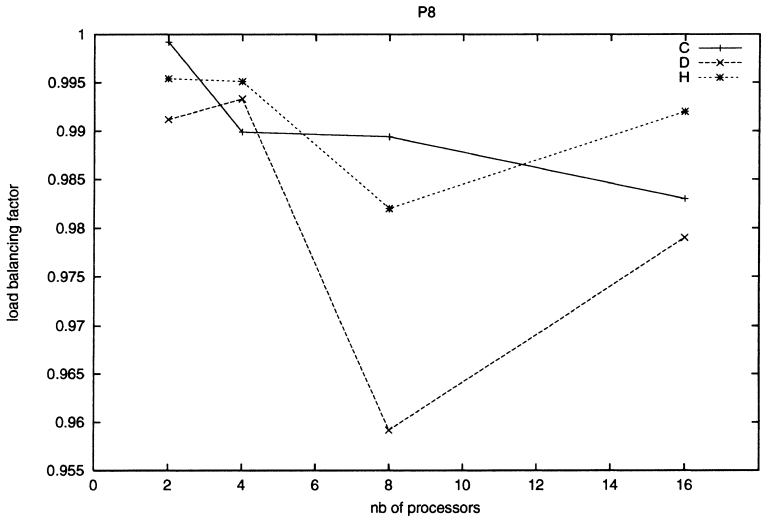

Fig. 1. Load balancing factor for coarse grain – medium size problem.

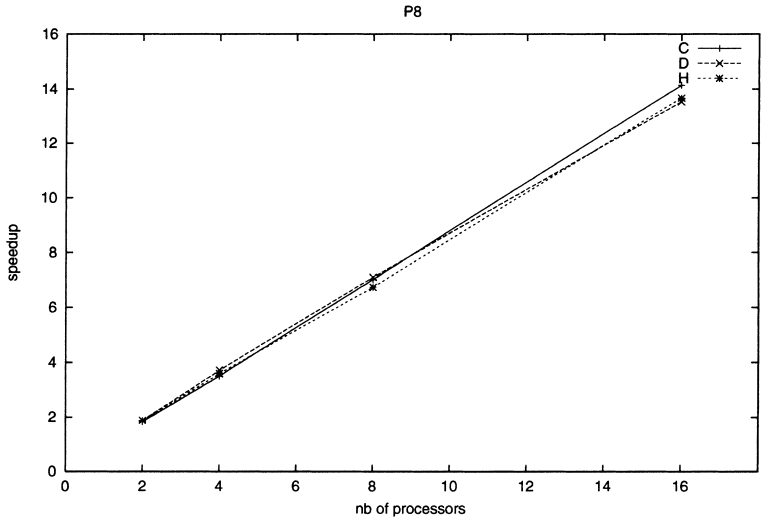

Fig. 2. Speedup for coarse grain – medium size problem.

clearly from the results on problems P1–P6); (2) *the ratio of the number of processors over the time per task*: if this ratio is high, the bottleneck generated by simultaneous accesses to the pool significantly penalizes the centralized approach.

To better analyze the effect of these two factors, we ran the algorithm limiting the maximum number of iterations in the bounding procedure to 1 ($t_{\max} = 1$). This is equivalent to solving only the MCNF relaxation at each node of the tree. This also dramatically increases the size of the tree. With this new parameter setting, none of the problems could be solved sequentially in a reasonable amount of time. Therefore, we selected a randomly generated problem of smaller dimensions to run these tests. This problem, called P12, has 26 depots, 124 customers, 3 types of containers, and 2392 arcs. Using the new parameter setting, the sequential algorithm generated 300 563 nodes and took 19 141 s on one Ultra 1 workstation. Thus this resulting problem is *fine grain* (only one iteration of the bounding procedure is performed at each node for a problem of small dimensions) and *large size* (the tree is one order of magnitude larger than the one generated for problem P8). In contrast, problem P8 is *coarse grain* (several iterations of the bounding procedure are executed for a large-scale problem) and *medium size*.

The three parallel strategies have been tested on problem P12. Again, the search penalties were very close to 1. Figs. 3 and 4 show the load balancing factors and the speedups obtained by the three strategies for $p = 2, 4, 8, 16$, respectively. When compared to Figs. 1 and 2, one notices significant differences: collegial control approaches are more efficient than the centralized strategy, with the decentralized displaying a slight edge over the hybrid. Moreover, the centralized approach exhibits difficulties in balancing the loads and the situation deteriorates rapidly as the number of processors increases. This is a consequence of the bottleneck created by frequent simultaneous accesses to the pool. The current implementation of the hybrid

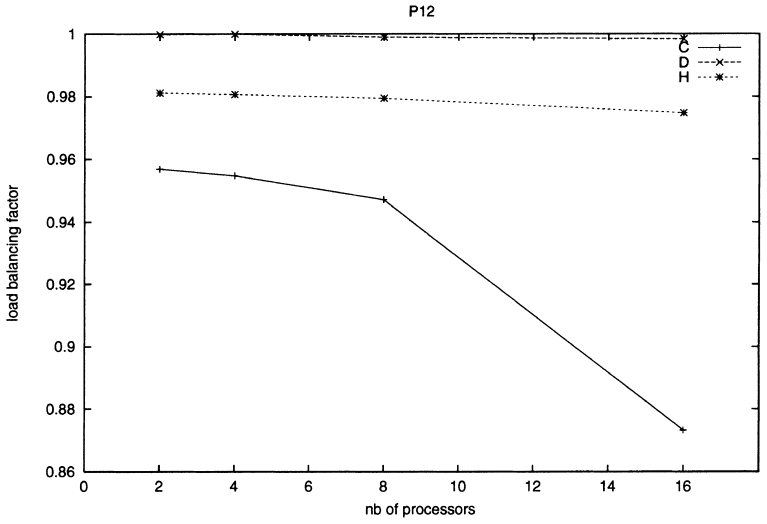

Fig. 3. Load balancing factor for fine grain – large size problem.

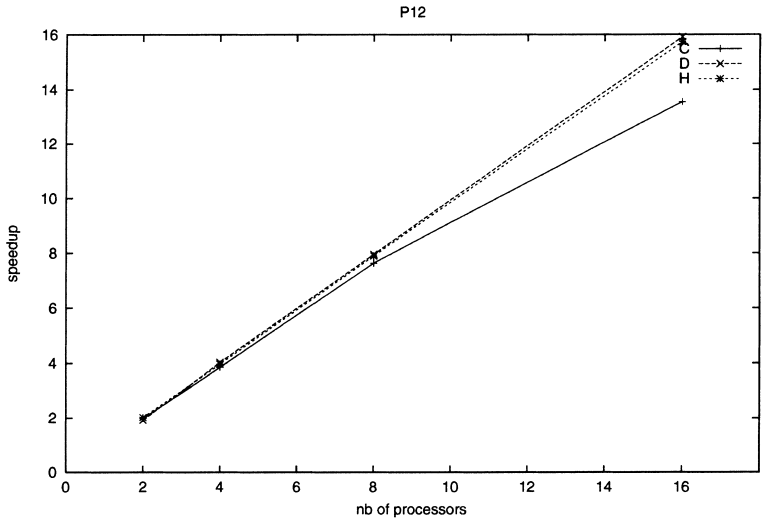

Fig. 4. Speedup for fine grain – large size problem.

approach could suffer from the same drawback, but the problem is much less critical than for the centralized strategy, since each task corresponds to the exploration of a branch in a large tree. If deemed necessary, this could be corrected by actually using the distributed pools and decreasing the number of nodes sent to the coordinator pool. One notices that, since the tree is very large, the decentralized strategy creates a sufficiently large number of tasks to keep processors busy most of the time. Indeed,

the load balancing factor is very close to 1, irrespective of the number of processors (although it decreases very slowly with p). As a result, the speedup of the centralized strategy drops to 13.55 when $p = 16$, while for both the H and D strategies, it is (almost) ideal: 15.74 and 15.92, respectively. Note that for such large trees, the initialization phase is negligible and Amdahl's law is no longer a concern.

5. Conclusion

We have presented three general strategies that may be used to parallelize most B&B algorithms specialized to location/network design formulations. An experimental comparison of these strategies has been performed by using the best known sequential B&B algorithm for the location/allocation problem with balancing requirements.

The three approaches are obtained by dividing the search tree among processors and performing operations on several subproblems simultaneously. The strategies differ in the way they manage the list of subproblems and control the search. While the presentation is general, the implementations made use of a message-passing environment, a distributed network of workstations.

Relative to the particular problem studied, the experimental results indicate that our implementations are very effective. No search penalty is observed, indicating that the parallel search tree is no larger than the sequential one, and near-linear speedups have been obtained for actual large-scale applications, with up to 16 processors. From a broader point of view, the results indicate that while a centralized approach, where one process controls the search, is appropriate for problems of relatively coarse granularity and small trees, its performances degrade as soon as the granularity gets finer and the size of the tree increases. In these circumstances, strategies that implement a collegial control of the list of subproblems perform better. In our experiments, a purely decentralized approach performed best, with a hybrid organization a close second.

Few studies have appeared that compare strategies to control the distribution of subproblems and the search path of parallel B&B algorithms (see Section 1). Our paper thus contributes to broaden our understanding of parallel B&B algorithm behavior, particularly when applied to location/network design formulations on message-passing architectures. The study presented in this paper has also pointed out a number of interesting research directions, including: initialization procedures that make a better use of the parallel environment without increasing the search penalty; implementation strategies of hybrid approaches that exploit more intensively the local pools; a more comprehensive study of the various parameters that fashion the decentralized and hybrid searches; the development of adaptive control and load balancing strategies; the study of the relative performances of the strategies on various architectures; the integration of different selection rules (depth-first, best-first) within the parallelization strategies (see [5] for a recent comparative study of selections rules in the context of parallel B&B algorithms); the adaptation of the strategies to other location/network design formulations.

Acknowledgements

Financial support for this project was provided by NSERC (Canada) and the Fonds FCAR (Québec). Special thanks are due to Serge Bisaillon for his help in testing our code. We are also indebted to four anonymous referees whose comments have helped us to improve the quality of our paper.

References

[1] R.K. Ahuja, T.L. Magnanti, J.B. Orlin, Network Flows: Theory, Algorithms, and Applications, Prentice-Hall, Englewood Cliffs, MA, 1993.
[2] G.M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: AFIPS Conference Proceedings, 1967, pp. 483–485.
[3] G. Authié, M. Elkihel, F. Viader, A parallel best-first branch-and-bound algorithm for 0–1 problems: single pool and multiple pool implementations, Report LAAS 96155, Laboratoire d'analyse et d'architecture des systèmes, CNRS, Toulouse, 1996.
[4] B. Bourbeau, Approches de parallélisation basées sur l'organisation de la mémoire pour des méthodes de séparation et évaluation progressive, M.Sc. Thesis, Département d'informatique et de recherche opérationnelle, Université de Montréal, publication CRT-98-18, Centre de recherche sur les transports, Montréal, 1998.
[5] J. Clausen, M. Perregaard, On the best search strategy in parallel branch-and-bound – best-first search vs. lazy depth-first-search, Annals of Operations Research 90 (1999) 1–17.
[6] G. Cornuéjols, G.L. Nemhauser, L.A. Wolsey, The uncapacitated facility location problem, in: R.L. Francis, P.B. Mirchandani (Eds.), Discrete Location Theory, Wiley/Interscience, New York, 1990, pp. 119–168.
[7] T.G. Crainic, P.J. Dejax, L. Delorme, Models for multimode multicommodity location problems with interdepot balancing requirements, Annals of Operations Research 18 (1989) 279–302.
[8] T.G. Crainic, L. Delorme, P.J. Dejax, A branch-and-bound method for multicommodity location with balancing requirements, European Journal of Operational Research 65 (3) (1993) 368–382.
[9] J. Eckstein, Distributed versus centralized storage and control for parallel branch and bound: mixed integer programming on the CM-5, Computational Optimization and Applications 7 (2) (1997) 199–220.
[10] D. Erlenkotter, A dual-based procedure for uncapacitated facility location, Operations Research 26 (6) (1978) 992–1009.
[11] B. Gendron, T.G. Crainic, Parallel implementations of a branch-and-bound algorithm for multicommodity location with balancing requirements, INFOR 31 (3) (1993) 151–165.
[12] B. Gendron, T.G. Crainic, Parallel branch-and-bound algorithms: survey and synthesis, Operations Research 42 (6) (1994) 1042–1066.
[13] B. Gendron, T.G. Crainic, A branch-and-bound algorithm for depot location and container fleet management, Location Science 3 (1) (1995) 39–53.
[14] B. Gendron, T.G. Crainic, A parallel branch-and-bound algorithm for multicommodity location with balancing requirements, Computers and Operations Research 24 (9) (1997) 829–847.
[15] B. Gendron, T.G. Crainic, A. Frangioni, Multicommodity capacitated network design, in: B. Sansó, P. Soriano (Eds.), Telecommunications Network Planning, Kluwer Academics Publishers, Dordrecht, 1998, pp. 1–19.
[16] T. Ibaraki, Enumerative approaches to combinatorial optimization, Annals of Operations Research (1987) 10–11.
[17] J. Krarup, P.M. Pruzan, The simple plant location problem: survey and synthesis, European Journal of Operational Research 12 (1983) 36–81.
[18] T.L. Magnanti, R.T. Wong, Network design and transportation planning: models and algorithms, Transportation Science 18 (1) (1984) 1–55.

[19] G.P. McKeown, V.J. Rayward-Smith, H.J. Turpin, Branch-and-bound as a higher-order function, Annals of Operations Research 33 (1991) 379–402.
[20] M. Minoux, Network synthesis and optimum network design problems: models, solution methods and applications, Networks 19 (1989) 313–360.
[21] J. Mohan, A study in parallel computation: the traveling salesman problem, Report CMU-CS-82-136(R), Computer Science Department, Carnegie-Mellon University, Pittsburgh, 1982.
[22] M.J. Quinn, Analysis and implementation of branch-and-bound algorithms on a hypercube multicomputer, IEEE Transactions on Computers 39 (3) (1990) 384–387.