Pergamon

PII: S0305-0548(96)00094-9

Computers Ops Res. Vol. 24, No. 9, pp. 829–847, 1997

0305-0548/97 $17.00+0.00

A PARALLEL BRANCH-AND-BOUND ALGORITHM FOR MULTICOMMODITY LOCATION WITH BALANCING REQUIREMENTS

Bernard Gendron[1]†‡ and Teodor Gabriel Crainic[2]§

[1] Département d'informatique et de recherche opérationnelle and Centre de recherche sur les transports, Université de Montréal, C.P. 6128, Succursale Centre-Ville, Montreal, Quebec, H3C 3J7, Canada

[2] Département des sciences administratives, Université du Québec à Montréal and Centre de recherche sur les transports, Université de Montréal, Montreal, Quebec, Canada

(Received May 1995; in revised form December 1996)

Scope and Purpose—The management of a fleet of vehicles over a medium to long-term planning horizon constitutes one of the major logistics issues faced by distribution and transportation firms. In the particular context of the management of heterogeneous fleets of containers by international maritime shipping companies, important strategic and tactical decisions have to be taken relative to the location of the depots for the empty containers and the forecast and management of the empty movements. This article reviews a formulation and an algorithm to tackle this problem that, in practice, may have to be solved repeatedly because often, container shipping companies do not build their own depots, but rather use existing facilities from other modes (ports and railyards, mostly). As a result of this characteristic, it is desirable that the algorithm be solved in a reasonable amount of time (within a few minutes) on computers which are widely available. However, experiments with the best known algorithm on an actual application have shown computing times of approximately 3 h on a powerful workstation. Therefore, parallel computing emerges as an attractive way to improve the performance of the algorithm. This article presents and analyzes an efficient parallel branch-and-bound algorithm and applies many new ideas for implementing branch-and-bound algorithms on parallel architectures.

Abstract—This article presents a parallel branch-and-bound algorithm for solving the multicommodity location problem with balancing requirements, that is based on the best known sequential method for solving the problem. The algorithm aims to exploit parallelism by dividing the search tree among processes and by performing operations on several subproblems simultaneously. The algorithm is divided into two phases: synchronous initialization and asynchronous exploration. Experimental results on a distributed network of workstations are reported and analyzed. © 1997 Elsevier Science Ltd

1. INTRODUCTION

The multicommodity location problem with balancing requirements (MLB) was first introduced by Crainic *et al.* [1]. The problem is motivated by the following industrial application, related to the management of a heterogeneous fleet of containers by an international maritime shipping company. Once a ship arrives at port, the company has to deliver loaded containers, which may come in several types and sizes, to designated in-land destinations. Following their unloading by the importing customer, empty containers are moved to a depot. From there, later on, they may be delivered to customers which request containers for subsequent shipping of their own products. Further, empty containers often have to be repositioned to other depots. These interdepot movements are a consequence of the regional unbalances in empty container availabilities and needs throughout the network: some areas lack containers of certain types, while others have surpluses of them. This requires balancing movements of empty containers among depots and thus differentiates this problem from classical location-allocation applications. The general problem is therefore to locate depots in order to collect the supply of empty containers available at customers' sites and to satisfy the customer requests for empty containers, while minimizing the total operating costs: the costs of opening and operating the depots, and the costs generated by customer-depot and interdepot movements.

† To whom all correspondence should be addressed.

‡ Bernard Gendron is Assistant Professor at the department of Computer Science and Operations Research, University of Montreal. His Ph.D. Thesis, centered on sequential and parallel algorithms for solving network design problems, was awarded the first prize at the Transportation Science Section (INFORMS) Dissertation competition of 1995. His research interests include integer and combinatorial optimization, large-scale optimization, location and network design problems and parallel computing.

§ Teodor Gabriel Crainic is Professor of Operations Research at the Department of Administrative Sciences, University of Quebec at Montreal, and Director of the Centre for Research on Transportation, University of Montreal. His research interests are in OR models, methods, and planning tools applied to transportation, as well as the study of parallel computing and its impact on the design of OR models and algorithms.

In practice, any formulation of this problem may have to be solved repeatedly because often, container shipping companies do not build their own depots, but rather use facilities from other modes (ports and railyards, mostly). Note also that, when planning these operations for a medium to long term time horizon (typically, one month to a year), shipping companies need to test several scenarios, corresponding to variations in patterns of demand, transportation costs, space availability and costs for container warehousing, etc. Therefore, both algorithmic and solution efficiencies are of prime importance for this class of applications.

Among the solution procedures that have been proposed for solving the problem [2–5], the branch-and-bound algorithm presented by Gendron and Crainic [5], based on a dual-ascent bounding procedure proposed by Crainic and Delorme, [2] proved to be the most efficient. However, there are still hard data instances for which the algorithm generates rather large search trees. As an example, solving an actual application generates 3845 nodes in the branch-and-bound tree and, because of the complexity of the bounding procedure, requires more than 3 h of computing time on a powerful workstation. Therefore, parallel computing emerges as an attractive way to improve the performance of the algorithm. The objective of this article is to present and analyze an efficient parallel branch-and-bound algorithm for the MLB, where operations are performed on several subproblems simultaneously.

Previous attempts at exploiting parallelism to solve the MLB can be found in [6–8]. In particular, Gendron and Crainic [8] implement a parallel version of the branch-and-bound method developed by Crainic *et al.* [3]. This algorithm implements a master-slave approach to perform bounding procedures for several nodes simultaneously. The master process manages the list of generated subproblems and assigns subproblems to slave processes according to a depth-first criterion. The algorithm proposed in the present article is significantly different, not only because it is based on the most efficient sequential algorithm known up to date, but also, and foremost, because it proposes a multiple-list implementation where each process has its own local pool of subproblems.

To the best of our knowledge, the present article proposes many new ideas for implementing branch-and-bound algorithms on parallel architectures (see Gendron and Crainic [9] for a detailed survey of the field). It presents a two-phase algorithm starting with a synchronous initialization phase that can be seen as a generalization of the sequential best-first search strategy. Two variants of the synchronous procedure are introduced and compared. The second phase consists of an asynchronous procedure where each process performs its own depth-first search of a subtree. When its local pool of subproblems is empty, an idle process sends a request for work to a coordinator process that schedules the load balancing activities based on information received from the working processes.

The article is organized as follows. In Section 2, we give a general network formulation of the MLB, which is independent of the original application. Section 3 presents the sequential branch-and-bound algorithm that forms the basis of the parallel one; this last is the subject of Section 4. Computational experiments on a distributed network of SUN workstations are presented in Section 5. The Conclusion summarizes our work, in particular the results obtained from the experiments.

2. PROBLEM FORMULATION AND RELAXATIONS

To formulate the problem, we 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) which move through the network and which are represented by set P. 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 we assume that there exists at least one origin or destination adjacent to each depot $j (O(j) \cup D(j) \neq \emptyset)$. 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 arcs, $A_{OT}=\{(i,j) \in A: i \in O, j \in T\}$; depot-to-customer arcs, $A_{TD}=\{(i,j) \in A: i \in T, j \in D\}$; and depot-to-depot arcs, $A_{TT}=\{(i,j) \in A: i \in T, j \in T\}$.

The problem consists in minimizing costs incurred by moving flows through the network in order to satisfy supplies at origins and demands at destinations. For each supply customer $i \in O$, the supply of commodity p is noted o_i^p, while for each demand 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 takes value 1 if depot j is opened, and value 0 otherwise. The problem is then formulated as:

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

subject to

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

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

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

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

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

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

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

Constraints (2) and (3) ensure that supply and demand requirements are met, relations (4) correspond to flow conservation constraints at depot sites, while (5) and (6) forbid customer-related movements through closed depots. Note that analogous constraints for the interdepot flows are redundant if interdepot costs satisfy the triangle inequality [1] an assumption that we follow throughout this text.

Lower bounds on the optimal value of this problem may be derived by considering the *strong relaxation*, obtained by replacing the integrality constraints (8) with $y_j\ge 0$, $\forall j\in T$. (Note that the constraints $y_j\le 1, \forall j\in T$, are redundant because of constraints (5) and (6) and the fact that the fixed costs are non-negative.) The dual of the resulting linear program, noted $\mathcal{D}$, may be formulated as;

$$Z_{\mathcal{D}} = \max \sum_{p\in P}\left(\sum_{i\in O} o_i^p \mu_i^p + \sum_{i\in(\mathcal{D})} d_i^p \eta_i^p \right), \tag{9}$$

subject to;

$$\mu_i^p - \lambda_j^p - \gamma_{ij}^p, \le c_{ij}^p\ \forall (i,j)\in A_{OT}, p\in P, \tag{10}$$

$$\eta_i^p + \lambda_j^p - \gamma_{ji}^p \le c_{ji}^p,\ \forall (j,i)\in A_{TD}, p\in P, \tag{11}$$

$$\lambda_j^p - \lambda_k^p \le c_{jk}^p,\ \forall (j,k)\in A_{TT}, p\in P, \tag{12}$$

$$\sum_{p\in P}\left\{ \sum_{i\in O(j)} o_i^p \gamma_{ij}^p + \sum_{i\in D(j)} d_i^p \gamma_{ji}^p \right\} \le f_j,\ \forall j\in T, \tag{13}$$

$$\gamma_{ij}^p \ge 0,\ \forall (i,j)\in A_{OT}, p\in P, \tag{14}$$

$$\gamma_{ji}^p \ge 0,\ \forall (j,i)\in A_{TD}, p\in P. \tag{15}$$

Two relaxations of the MLB, that may be derived directly either from the dual of the strong relaxation [2] or from Lagrangean relaxations [5], may be used in order to efficiently compute tight lower bounds on $Z_{\mathcal{D}}$.

The first relaxation can be obtained by fixing the γ variables to values satisfying constraints (13), or equivalently, by relaxing constraints (5) and (6) and introducing them into the objective function with non-negative γ multipliers. We then obtain the following problem, called *FLIP relaxation*:

$$Z(\gamma) = \min \left\{ \sum_{p\in P}\left(\sum_{(i,j)\in A_{OT}} (c_{ij}^p + \gamma_{ij}^p) x_{ij}^p + \sum_{(j,i)\in A_{TD}} (c_{ji}^p + \gamma_{ji}^p) x_{ji}^p + \sum_{(j,k)\in A_{TT}} c_{jk}^p x_{jk}^p \right) \right\}, \tag{16}$$

subject to constraints (2) to (4) and (7). This problem is a *multicommodity incapacitated minimum cost network flow problem* (MCNF), and thus decomposes into $|P|$ single-commodity incapacitated minimum

cost network flow problems.

The second relaxation can be derived by relaxing constraints (4) and introducing them into the objective function with λ multipliers. Restricting the multipliers to values satisfying constraints (12), one then obtains the following problem, called *FLOP relaxation*:

$$Z(\lambda) = \min \left\{ \sum_{j \in T} f_j y_j + \sum_{p \in P} \left(\sum_{(i,j) \in A_{OT}} (c_{ij}^p + \lambda_j^p) x_{ij}^p + \sum_{(j,i) \in A_{T\mathcal{D}}} (c_{ji}^p - \lambda_j^p) x_{ji}^p \right) \right\}, \tag{17}$$

subject to constraints (2), (3) and (5) to (8). This problem is an *uncapacitated location problem* (ULP), also called simple plant location problem [10] and uncapacitated facility location problem [11]. One of the most efficient methods for solving the ULP is the DU ALOC algorithm proposed by Erlenkotter [12]. The algorithm is based on a dual-ascent procedure that provides a lower bound of good quality with a rather limited computational effort, and also derives primal solutions satisfying the integrality constraints.

3. SEQUENTIAL BRANCH-AND-BOUND ALGORITHM

The depth-first sequential branch-and-bound algorithm [5] makes use of the two relaxations defined previously to compute tight bounds, and of efficient branching rules and preprocessing tests to reduce the size of the branch-and-bound tree. To further curtail the enumeration, upper bounds may be easily obtained by using the primal information generated when solving the relaxations. After solving the FLIP relaxation, an upper bound may be computed from its optimal solution by setting y_j to 1 whenever there is flow moving through depot j, and to 0 otherwise. When solving the FLOP relaxation, one may use the best primal solution identified by DUALOC, and then solve an MCNF obtained by fixing the y variables to the values of this primal solution.

To represent location variables that are fixed through branching and preprocessing rules, we define the sets $T_{01} = \{j \in T : y_j \in \{0,1\}\}$, $T_0 = \{j \in T : y_j = 0\}$, and $T_1 = \{j \in T : y_j = 1\}$ of *free*, *closed*, and *open* depots, respectively. To generate subproblems from a given subproblem S, we use a dichotomic branching rule: a depot $j^* \in T_{01}$ is chosen according to some criterion, and $S_0^{j^*}$ is obtained by transferring j^* to T_0, while $S_1^{j^*}$ results from transferring j^* to T_1. According to the terminology of trees, $S_0^{j^*}$ and $S_1^{j^*}$ are the *0-son node* and the *1-son node*, respectively, of the *father node* S, and the original problem, where all depots are free, is the *root node*. To decide which generated subproblem should be examined in priority, we use the *depth-first rule*: choose one of the subproblems that was generated most recently. Since it can be implemented efficiently using a last-in-first-out stack, this rule minimizes computer storage requirements, though the total number of subproblems it generates might be large [13]. 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 branch-and-bound tree, this disadvantage may be significantly reduced [5].

Formally then, the *BB algorithm* keeps a stack Λ of generated subproblems, as well as the value Z^u of the best solution identified thus far, and proceeds as follows.

(1) (*Initialization*) S is the original problem: $T_{01} \leftarrow T$, $T_0 \leftarrow \emptyset$, $T_1 \leftarrow \emptyset$. $\Lambda \leftarrow \emptyset$. $Z^u \leftarrow +\infty$.
(2) (*Preprocessing rule*) Attempt to fix some variables (T_{01}, T_0 and T_1 may be modified).
(3) (*Bounding procedure*) Perform the bounding procedure on S (Z^u may be updated); if S may be fathomed, goto 5.
(4) (*Branching rule*) Choose $j^* \in T_{01}$ and generate $S_o^{j^*}$ and $S_1^{j^*}$; select one of them to examine next, as subproblem S, and add the other to Λ. Goto 2.
(5) (*Stopping test*) If $\Lambda = \emptyset$, STOP; Z^u is the optimal value of the original problem.
(6) (*Backtracking*) Select the subproblem S on top of Λ. If it may be fathomed goto 5, otherwise, goto 2.

Experimental results, reported in [5], have shown the superiority of the following branching rule that makes use of the *slack variables* associated to constraints 13 of the dual $\mathcal{D}$ (defined for each $j \in T$ as $s_j = f_j - \sum_{p \in P} \left\{ \sum_{i \in o(j)} o_i^p \gamma_{ij}^p + \sum_{i \in \mathcal{D}(j)} d_i^p \gamma_{ji}^p \right\}$):

Slack branching rule: Choose $j^* = arg\ \max_{j \in T_{01}} \{s_j\}$, and select first subproblem $S_0^{j^*}$.

3.1. *Bounding procedure*

The following dual-ascent bounding procedure is executed at step 3 of the BB algorithm. Lower bounds are computed on the optimal value $Z_{\tilde{\mathcal{D}}}$ of the modified dual $\tilde{\mathcal{D}}$, which is obtained from $\mathcal{D}$ by adding the constant term $\sum_{j \in T_1} f_j$ to the objective function, and by replacing the fixed costs f_j $(j \in T)$ in constraints 13 with the modified fixed costs $\tilde{f}_j (j \in T)$, defined as: $\tilde{f}_j = f_j$, if $j \in T_{01}$; $\tilde{f}_j = +\infty$, if $j \in T_0$; $\tilde{f}_j = 0$, if $j \in T_1$. The *FLIP-FLOP procedure* may then be formally stated as follows.

(1) (*Initialization*) Initialize γ, Z_0^l, a lower bound on Z_D, and Z^u, an upper bound on Z. Set the iteration counter t to 1.
(2) (*Integrality test*) If $T_{01} = \emptyset$, compute an upper bound Z_t^u on Z by solving an MCNF; if $Z_t^u < Z^u$, $Z^u \leftarrow Z_t^u$; STOP.
(3) (*Lower bound*) Compute a lower bound Z_t^l on Z_D either by solving the FLIP relaxation (if iteration t mod $2 = 1$), or by applying DUALOC to the FLOP relaxation (if iteration t mod $2 = 0$).
(4) (*Lower bound test*) If $Z_t^l \geq Z^u$, STOP.
(5) (*Upper bound*) Compute an upper bound Z_t^u on Z either from the optimal solution of the FLIP relaxation (if iteration t mod $2 = 1$), or by solving an MCNF derived from the best primal solution to the FLOP relaxation identified by DUALOC (if iteration t mod $2 = 0$).
(6) (*Upper bound update*) If $Z_t^u < Z^u$, $Z^u \leftarrow Z_t^u$.
(7) (*Stopping test*) If $Z^u - Z_t^l < \epsilon_1 Z_t^l$, $Z_t^l - Z_{t-1}^l < \epsilon_2 Z_{t-1}^l$ or $t = t_{\max}$, STOP.
(8) (*Preprocessing rule*) Attempt to fix some variables (T_{01}, T_0 and T_1 may be modified).
(9) $t \leftarrow t+1$. Goto 2.

The initialization step depends on the status of the current subproblem: either it was just generated by the branching rule, or it was obtained after backtracking (or is the root of the tree). In the first case, the values computed at the father node are used to initialize γ and Z_0^l, while in the second case these variables are initialized to 0. In all cases, Z^u is initialized to the value of the best feasible solution identified thus far by the BB algorithm.

The procedure starts with a FLIP, a choice experimentally proven to be superior [2]. Indeed, if a FLOP is first solved, one does not take into account the influence of the balancing flows. In particular, some depots may be given very large values for their associated γ multipliers, and consequently become "unattractive", though they might subsequently be required in order to satisfy the balancing constraints.

Note that the lower bound test performed at Step 4 includes the usual feasibility test that stops computations when the relaxation is determined to be infeasible. Indeed, we assume in our description that any infeasible subproblem takes an infinite optimal value. The stopping test uses three parameters ϵ_1, ϵ_2 and t_{max} that can be adjusted by the user. The first stops the procedure when the relative gap between the lower and upper bounds is sufficiently small, the second comes into play when the lower bound has not sufficiently increased from one iteration to the next, while the third limits the number of iterations.

3.2. *Preprocessing*

Two properties can be used to implement preprocessing tests in step 8 of the FLIP-FLOP procedure. The first property gives a condition, based on the slack variables, that indicates when a binary variable must take value 0 in any optimal solution to the MLB. It may be formally stated as follows.

Slack property: let Z^l be a lower bound on Z_D corresponding to a feasible solution $(\mu, \eta, \lambda, \gamma)$ of the dual $\tilde{D}$. Let Z^u be an upper bound on Z. If $(Z^l + s_j) \geq Z^u$, then $y_j = 0$ in any optimal solution to the MLB $(j \in T_{01})$.

The second property determines when a binary variable must be set to 1 in order to satisfy supply and demand requirements:

OD property: if, for a given commodity p, there exists an origin (destination) i with $o_i^p > 0 (d_i^p > 0)$ such that only one depot $j \in T_{01}$ is adjacent to i, then $y_j = 1$ in any feasible solution to the MLB.

4. PARALLEL BRANCH-AND-BOUND ALGORITHM

We now present a parallel branch-and-bound algorithm intended to be executed on coarse-grained asynchronous message-passing systems. Since our objective is to speedup the time required to solve hard data instances for which the sequential algorithm generates large search trees, our parallelization scheme performs operations on several subproblems simultaneously. Note that other parallelization strategies

would accelerate tedious computation phases, especially the bounding procedure, without changing the exploration of the tree [9]. In the present case, for example, the decomposition of the MCNF into $|P|$ single-commodity minimum cost network flow problems could be performed in parallel. However, the efficiency of a parallelization strategy based only on such a decomposition would be questionable, since actual applications of the MLB have relatively few commodities (typically, in the order of 10 to 20). In any case, when a sufficient number of processors is available, this strategy complements the tree decomposition approach.

The proposed algorithm consists of two phases: a synchronous initialization phase and an asynchronous exploration phase. The first phase can be seen as a generalization of the sequential best-first search strategy, while in the second phase, the tree is divided into several subtrees explored concurrently by a set of working processes. Each working process performs a modified sequential depth-first procedure that includes communications. These communications mainly serve two purposes: inform all processes when a new upper bound has been found, and balance the workload among processes. To achieve this last objective, we use a coordinator process that schedules the load balancing activities based on information received from the working processes.

4.1. First phase: synchronous initialization

At the beginning of the parallel execution of a branch-and-bound algorithm, only one subproblem, the root node of the tree, is available to all processes. As a consequence, a start-up phase where parallelism is not fully utilized seems difficult to avoid. Several authors have proposed initialization strategies to overcome this problem (see Gendron and Crainic [9] for a detailed review of these strategies). Here, we suggest a new approach which requires synchronization among processes and that can be seen as a generalization of the sequential best-first search procedure.

We assume there are $p=2^d$ processes, where $d>0$. The method is based on two procedures, executed by each process simultaneously with the others, that perform communications by assuming that process $n(1\leq n\leq p)$ and process $n+2^i(\lceil\log_2 n\rceil\leq i\leq\log_2 p-1)$ are connected to each other (when processes are mapped onto a hypercube topology, for example, this organization scheme may be represented by a binary tree that spans the hypercube).

The first procedure, called *p-decomposition*, when performed concurrently by all processes, results in the generation of one subproblem per process, from an initial one located on process 1. There are two variants of this procedure. The first one, called *symmetric p-decomposition*, expands the tree in a breadth-first fashion (all newly generated sub-problems are at the same level of the tree). The second variant, called *asymmetric p-decomposition*, expands the tree in a depth-first fashion (all newly generated subproblems, except two, are at different levels of the tree).

Assuming that the initial subproblem, located on process 1, is entirely characterized by the three sets T_{01}, T_0 and T_1, the symmetric p-decomposition procedure, executed by each process n, can be stated as follows:

(1) (*Initialization*) If $n\neq 1$, receive a subproblem. $t\leftarrow\lceil\log_2 n\rceil$.
(2) (*Stopping test*) If $t=\log_2 p$, STOP.
(3) (*Integrality test*) If $T_{01}=\emptyset$, send the subproblem to process $n+2^t$. $t\leftarrow t+1$. Goto 2.
(4) (*Branching rule*) Perform the branching rule, but instead of storing the alternative in a stack, send it to process $n+2^t$. $t\leftarrow t+1$. Goto 2.

In a straightforward implementation of the asymmetric variant, process 1 applies the branching rule up to $p-1$ times, each time sending one subproblem to a process that has not yet received one (for simplicity, we assume here that process 1 can reach all other processes directly):

(1) (*Initialization*) If $n\neq 1$, receive a subproblem and STOP. $t\leftarrow 1$.
(2) (*Stopping test*) If $t=p$, STOP.
(3) (*Integrality test*) If $T_{01}=\emptyset$, send the subproblem to process $n+t$. $t\leftarrow t+1$. Goto 2.
(4) (*Branching rule*) Perform the branching rule, but instead of storing the alternative in a stack, send it to process $n+t$. $t\leftarrow t+1$. Goto 2.

The second procedure, called p-best-first, assumes that each process manages a list Π of evaluated subproblems as a heap, which allows to identify the subproblem in Π with the smallest lower bound. It also assumes that Z^u is a variable used by each process to identify the best upper bound known by this process. The method, defined by the concurrent execution of the procedure by all processes, proceeds in

three steps. During the first step, process 1 obtains the smallest lower bound of all evaluated subproblems along with the identification n^* of the process that stores this subproblem in its local heap, as well as the best upper bound obtained so far by all processes. In Step 2, process 1 verifies a termination condition (namely, that the two bounds just obtained are equal), and notifies all processes when the condition is true. If the condition is not verified, process 1 initiates a communication phase at the end of which all processes know the best upper bound (as well as the process that stores the subproblem with the smallest lower bound), while process n^* knows that it stores the subproblem with the smallest lower bound. The third step is a simple communication phase between processes 1 and n^* (if $n^*=1$, process 1 simply selects the best subproblem from its heap), after which process 1 has obtained the subproblem with the smallest lower bound of all evaluated subproblems. The p-best-first procedure, run by each process n, may be stated as follows.

(1) (*First step*) Find the smallest lower bound Z^l of all subproblems in Π. If $Z^l \geq Z^u$, remove all subproblems from Π. $t \leftarrow \log_2 p$. $n^* \leftarrow n$.
(2) If $n > 2^{t-1}$, send n^*, Z^l and Z^u to process $n - 2^{t-1}$.
(3) If $t = \lceil \log_2 n \rceil$, goto 5.
(4) Receive n_+, Z^l_+ and Z^u_+. If $Z^l_+ < Z^l$, $Z^l \leftarrow Z^l_+$ and $n^* \leftarrow n_+$. If $Z^u_+ < Z^u$, $Z^u \leftarrow Z^u_+$. $t \leftarrow t-1$. If $t>0$, goto 2.
(5) (*Second step*) If $n=1$ and $Z^l = Z^u$, $n^* \leftarrow -1$ (a code to notify all processes that the problem has been solved).
(6) If $n \neq 1$, receive n^* and Z^u.
(7) $t \leftarrow \lceil \log_2 n \rceil$.
(8) If $t = \log_2 p$, goto 10.
(9) Send n^* and Z^u to process $n + 2^t$. $t \leftarrow t+1$. Goto 8.
(10) (*Third step*) If $n^* = n$, remove the best subproblem from Π.
(11) If $n^* = n$ and $n \neq 1$, send the best subproblem to process 1; if $n^* > 1$ and $n = 1$, receive the best subproblem.

Using these two procedures, it is easy to define a synchronous algorithm which, at each step, generates p subproblems from an initial one with the smallest lower bound among all generated subproblems. These p subproblems are then evaluated in parallel, and the subproblem with the smallest lower bound of all evaluated subproblems is determined by the p-best-first procedure. The algorithm is thus defined by the following procedure, executed by each process concurrently with the others.

(1) (*Initialization*) $A \leftarrow \emptyset$. $T_{01} \leftarrow T$, $T_0 \leftarrow \emptyset$, $T_1 \leftarrow \emptyset$. $Z^u \leftarrow +\infty$. $t \leftarrow 0$.
(2) (*p-decomposition*) Obtain one subproblem by applying either the symmetric or the asymmetric p-decomposition procedure.
(3) (*p-evaluation*) Perform the FLIP-FLOP procedure on the subproblem obtained at step 2. Store the subproblem in Π, unless it can be fathomed.
(4) (*Stopping test*) If $t = t_{sync}$, STOP.
(5) (*p-best-first*) Perform the p-best-first procedure. If the problem is solved, STOP. Otherwise, process 1 has obtained the subproblem with the smallest lower bound. $t \leftarrow t+1$. Goto 2.

The procedure makes use of t_{sync}, a parameter adjusted appropriately, specified by the user. If $t_{sync}=0$, the procedure is equivalent to an initial phase where each process obtains one subproblem generated from the root node (this was been used in the past by some authors [9]). If $t_{sync}=+\infty$, the procedure defines a completely synchronous algorithm, similar to the one proposed by Mohan [14]. Mohan's algorithm is also a p-best-first method, but is implemented with a single-list managed by a master process that generates the p subproblems at each step; hence, apart from synchronization, additional overheads are incurred as a result of the central control. The synchronous algorithm is of course not efficient for solving the MLB, since the time required to compute bounds by the FLIP-FLOP procedure may vary greatly from one subproblem to another. However, it may be efficient for solving problems for which bounds can be obtained almost in constant time or in a synchronous fashion. We introduce it here mainly to balance the load among processes as soon as possible (this is the role of the p-decomposition procedure), and to avoid generating unpromising subproblems (this is the role of the p-best-first procedure). In Section 5, we compare the performance of the method to an initialization which consists in giving the root node to one process.

4.2. Second phase: asynchronous exploration

After the initialization phase, each process starts with a heap of evaluated nodes and runs a depth-first branch-and-bound procedure starting from a subproblem taken from its heap. Later, when a process empties its stack, it first attempts to obtain a subproblem from its heap or, if its heap is empty, it tries to obtain work from other processes through a dynamic load balancing procedure.

A description of the various load balancing methods used in parallel branch-and-bound algorithms can be found in Gendron and Crainic [9]. Typically, load balancing methods designed for asynchronous multiple-list branch-and-bound algorithms, where each process manages its own list, can be divided into three categories (this classification can also be found in Kröger and Vornberger [15]).

Strategy on request: a process which has not enough work (or no work at all) sends a request to another process.

Strategy without request: processes transfer work units to other processes without being asked to.

Combined strategy: this strategy combines the two previous ones.

To preserve the efficiency of the sequential depth-first algorithm, we use a strategy on request. Several important issues have to be addressed, such as to which process to send a request and how this process reacts to the request (e.g., how to select the work units to transfer if it decides to grant the request). In our implementation, processes ask for work when they become idle, while the process granting a request sends the subproblem on top of its stack (it may also send a subproblem stored in its heap, as we will see shortly). In this way, local stacks can be managed similarly to the sequential version.

To decide on which process to send a request to, we use a *coordinator* that schedules the load balancing activities. The coordinator is periodically informed of the current load L_n of each process n (measured as the total number of subproblems stored in the memory of the process) and receives requests from idle processes. It identifies a candidate process that is to grant the request, by using a *round robin strategy* that eliminates processes that have an insufficient load by the yardstick of a *filter parameter* L_{min}. It proceeds by assigning the candidate identification to a variable *cand* which is incremented modulo p when the request is sent to the candidate or when it is eliminated because of an insufficient load. If no candidate is found following a complete tour, the request is kept in a queue Q. The round robin strategy, which uses a global variable to decide on the process which must receive the next request, is well-known in dynamic load balancing algorithms (see Kumar, Grama and Nageshwara Rao [16], who also propose and analyze a distributed version of this strategy that avoids contention of access to the global variable). However, to our best knowledge, a strategy that uses a coordinator to eliminate as possible candidates processes that have a poor load is new in parallel branch-and-bound algorithms.

A nice feature of this coordinator-based load balancing method is that it allows for an easy detection of the termination of the algorithm. Indeed, our implementation ensures that a process cannot receive a request from the coordinator while it is waiting for an answer to its own request, since the two types of messages (request and answer to a request) travel through different channels. Hence, the coordinator only needs to count the number n_{req} of requests received, to receive messages from all processes that are about to answer a request, and to count the number n_{send} of such messages. Then, if the difference between n_{req} and n_{send} is equal to the number of processes p, there are no more active processes and no subproblems are being exchanged. Thus, a termination message can be sent to all processes.

To summarize the activities performed by the coordinator, we state the following *coordinator procedure*:

(1) $Q \leftarrow \emptyset$, $L_n \leftarrow 0$, $1 \le n \le p$. $cand \leftarrow 1$. $n_{req} \leftarrow 0$. $n_{send} \leftarrow 0$.
(2) Check for arriving message. If no message has arrived, goto 7.
(3) If the message consists of a new load sent by process n, update L_n.
(4) If the message consists of a request from process n, insert n in Q, $L_n \leftarrow 0$ and $n_{req} \leftarrow n_{req} + 1$.
(5) If the message comes from a process that is about to answer a request, then $n_{send} \leftarrow n_{send} + 1$.
(6) If $n_{req} - n_{send} = p$, send a termination message to all processes and STOP.
(7) If $Q = \emptyset$, goto 2.
(8) Let n be the first element in Q. $cand_{old} \leftarrow cand$.
(9) If $cand \neq n$ and $L_{cand} > L_{min}$, send the request of process n to process *cand*; remove n from Q; $L_{cand} \leftarrow L_{cand} - 1$ (the coordinator guesses that the candidate will grant the request); $cand \leftarrow cand \bmod p + 1$; goto 7.
(10) $cand \leftarrow cand \bmod p + 1$. If $cand = cand_{old}$, goto 2. Otherwise, goto 9.

A working process about to perform the bounding procedure sends its current load to the coordinator if

Table 1. Dimensions of test problems and sequential results

Problem	$\|P\|$	$\|O\|$	$\|D\|$	$\|T\|$	$\|A_{OT}\|$	$\|A_{TD}\|$	$\|A_{TT}\|$	Nodes	Time (s)
P1	3	124	124	26	871	871	650	57	39
P2	6	125	125	25	875	875	600	77	141
P3	4	124	124	26	869	869	650	41	85
P4	2	219	219	44	2630	2630	1892	355	982
P5	2	219	219	44	2629	2629	1892	711	1487
P6	2	219	219	44	2629	2629	1892	267	1325
P7	1	219	219	44	2631	2631	1892	275	195
P8	2	220	220	43	2647	2647	1806	467	739
P9	2	220	220	43	2647	2647	1806	141	428
P10	12	289	289	130	1914	1914	890	3845	11860

it just became either less than or equal to L_{min}, or greater than L_{min}. At this point, it also verifies if any request has been received from the coordinator and, if this is the case, it sends a message to the coordinator to notify that the request is about to be answered. Then, if its current load is sufficient to grant the request (i.e. larger than L_{min}), it sends the subproblem on top of its stack, or, if the stack is empty, the best subproblem from its heap. If all subproblems in the heap can be fathomed by applying the lower bound test, or if its load is too small, the process sends a negative answer.

A process with no more work to perform sends a request to the coordinator, and then waits for an answer which can be a termination message, a negative answer to the request, or a positive answer to it. In the first case, the process stops. In the second case, the process sends another request to the coordinator. In the third case, a subproblem is received and treated. The treatment varies: if the subproblem was selected from the stack of the granting process, a depth-first search is started by computing bounds; otherwise, it was selected from the heap and the process starts a depth-first search by performing the slack branching rule (the values of the slack variables are received from the granting process).

To ensure the efficiency of the lower bound test of the bounding procedure, we implement a simple scheme to communicate among processes newly found upper bounds. On the one hand, after performing an iteration of the bounding procedure that improves its upper bound, a process broadcasts it to all other processes. On the other hand, before performing the lower bound test, every process verifies if some other process has sent a new upper bound and, if this is the case, it updates Z^u if the received upper bound is better than the stored one. This scheme is very simple, yet efficient in our case since, for the data instances that we tested, upper bounds are not updated frequently (typically, almost optimal upper bounds are found very early during the enumeration).

5. COMPUTATIONAL EXPERIMENTS

To perform our experiments, we used a distributed network of workstations that consists of 16 SUN Sparc5, each running a working process, and one SUN SparcServer1000 that hosts the coordinator. The workstations are connected via an Ethernet cable that provides a peak communication speed of 10 Mbits/s. The code is programmed in FORTRAN/77 by using the Parallel Virtual Machine (PVM) library of functions. To implement the FLIP-FLOP bounding procedure, we used the primal simplex code RNET [17] to solve minimum cost network flow problems. We also coded our own version of DUALOC, based on the method originally described by Erlenkotter [12], and later on refined by Van Roy and Erlenkotter [18]. In all experiments, the parameters of the FLIP-FLOP procedure are set to the following values; $\epsilon_1 = \epsilon_2 = 0.0001$ and $t_{max} = 10$.

To analyze the behavior of the parallel algorithm, we show the results obtained with ten problem instances. The first nine, noted P1 to P9, are generated randomly (the generator is described in [3]), while the last one, noted P10, is derived from data of an actual application to the planning of the container land transportation operations of a European-based maritime shipping company. We can further divide the randomly generated test problems into two classes: medium-size (P1 to P3) and large-size (P4 to P9). The characteristics of the problems are summarized in Table 1, which also presents basic statistics relative to the corresponding performance of the sequential algorithm: the number of nodes generated, and the elapsed time in seconds on one SUN Sparc5 workstation.

The objectives of our computational experiments are to assess the impact of the initialization phase, the load balancing method and the number of processes on the performance of the parallel algorithm. We also want to compare the parallel algorithm with the best implementation of the sequential method. For these purposes, we use four measures.

Load balancing factor. This is the 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 represents the total elapsed time minus the waiting and I/O times of the process during the parallel execution.

Search overhead factor. This is the ratio of the number of nodes generated during a parallel execution to the number of nodes explored by the sequential algorithm.

Speedup. This is the ratio of the total elapsed time required by the sequential algorithm to the total elapsed time ("wall clock") of the parallel execution.

Adjusted speedup. This is the speedup multiplied by the search overhead factor. For each problem instance, the reported measures of time and number of generated nodes represent averages over three runs. We also compare five initialization strategies:

A0: the asymmetric initialization with $t_{sync}=0$;
A5: the asymmetric initialization with $t_{sync}=5$;
S0: the symmetric initialization with $t_{sync}=0$;
S5: the symmetric initialization with $t_{sync}=5$;
R: the root initialization, which consists in giving the root node to one working process.

5.1. *Effect of the filter parameter*

First, we analyze the effect of the filter parameter, L_{min}, of the load balancing procedure, on the performance of the parallel algorithm, through the evolution of the load balancing factor and the adjusted speedup as functions of the filter parameter for different values of p (the number of working processes). Note that, by examining the adjusted speedup instead of the speedup, we ignore the effect of the search overhead and rather focus on the efficiency of the parallel implementation with respect to communication, synchronization and idle times (for a detailed analysis of search overhead, see Section 5.3 below). Figures 1–3 show the load balancing factor as a function of the filter parameter for, respectively, the medium-size problems (average), the large-size problems (average) and the actual application. Figures 4–6 display the adjusted speedup as a function of the filter parameter for the same three classes of problems. Figures are shown only when the root initialization is used, since the other initialization strategies display similar results.

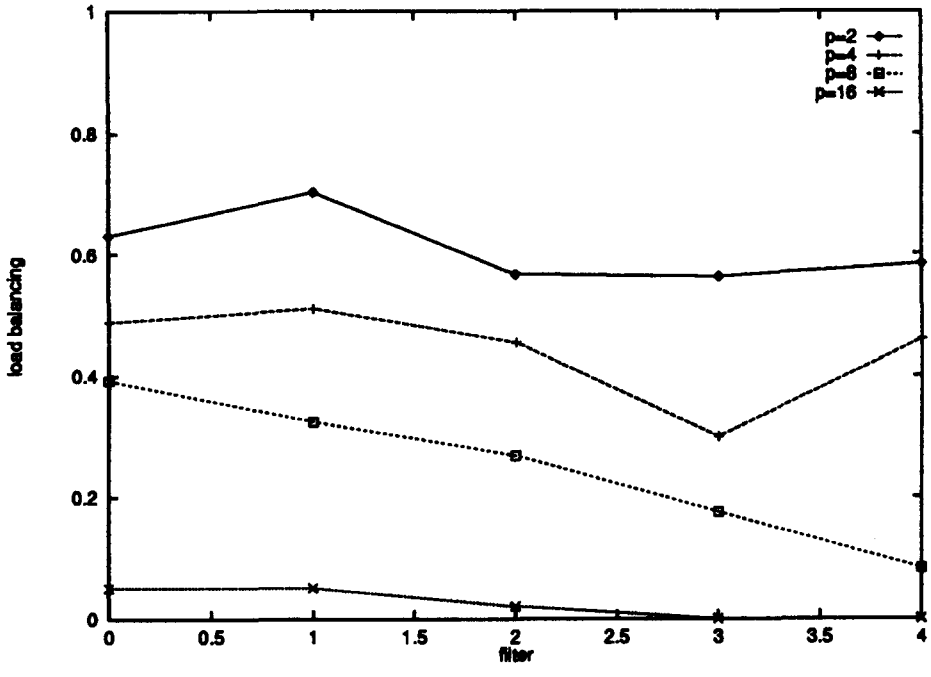

Fig. 1. Load balancing factor versus filter parameter for medium-size problems.

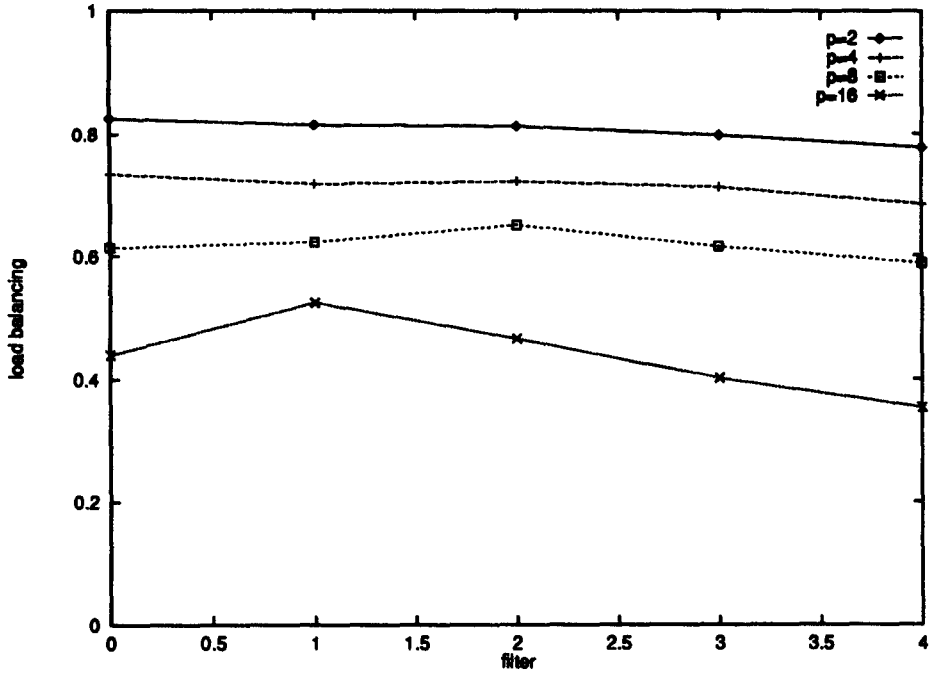

Fig. 2. Load balancing factor versus filter parameter for large-size problems.

The figures indicate that the maximum of both quantities is usually attained at $L_{min}=1$, irrespective of the problem instance and the number of working processes used. This means that best performances are obtained when granting processes have at least two subproblems in their pool (thus, being able to keep at least one for themselves). Therefore, in the remainder of this section, we only present results obtained when $L_{min}=1$.

The figures also reveal interesting characteristics of the parallel method and help to qualify the impact of the problem type on its behavior. For example, they illustrate that, for a given problem size, it is not always interesting to increase the number of processes. In particular, it is counterproductive to use 16

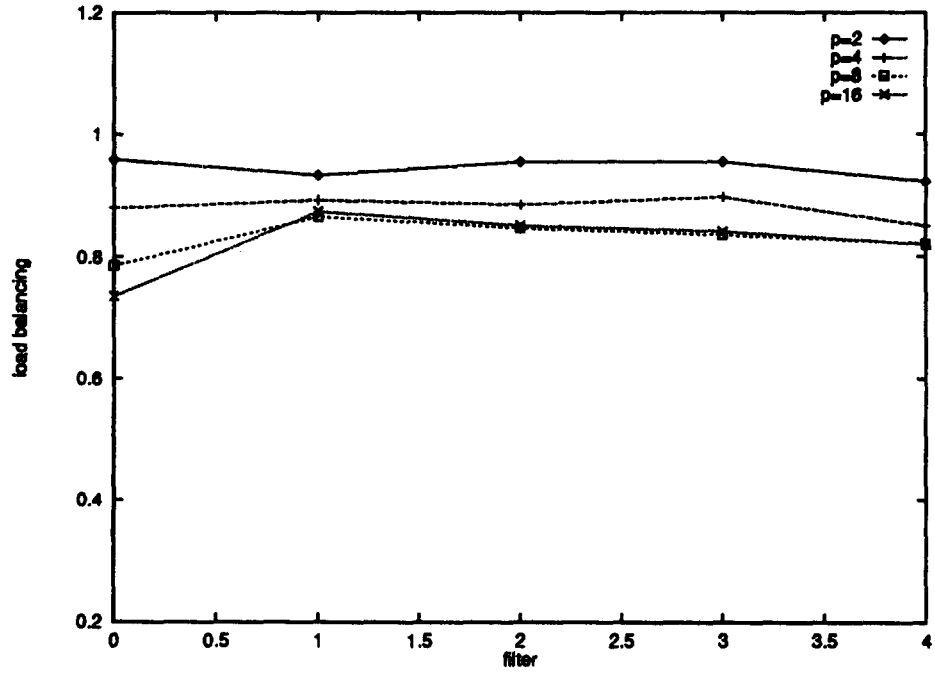

Fig. 3. Load balancing factor versus filter parameter for actual application.

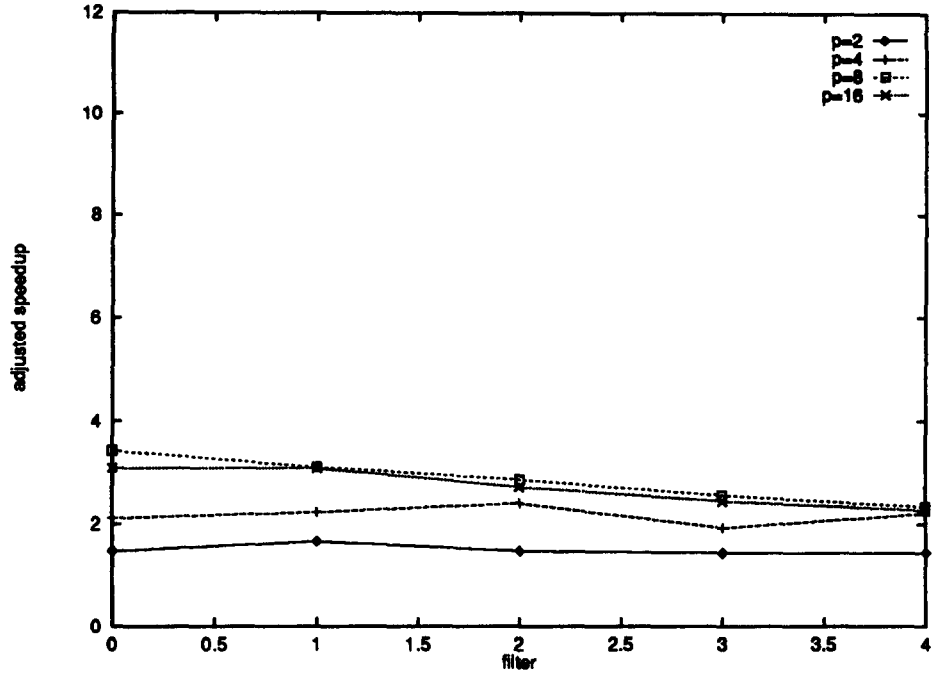

Fig. 4. Adjusted speedup versus filter parameter for medium-size problems.

working processes for medium-size problems since the results obtained are better when $p=8$. The poor values of the load balancing factor explain this loss of performance: many processes remain idle most of the time because of the small size of the branch-and-bound trees (less than 100 nodes). For larger problems, the impact is different. In particular, when $p=16$ and $L_{min}=1$, the large-size problems display, on average, a load balancing factor close to 0.5, which results in an adjusted speedup larger than 10. For the actual application, even higher load balancing factors are achieved, all greater than 0.8, and, as a result, an adjusted speedup of almost 8 is obtained for $p=8$ and of more than 12 for $p=16$. One may then conclude that 8 processes represents an "ideal" number for actual industrial problems of such dimensions,

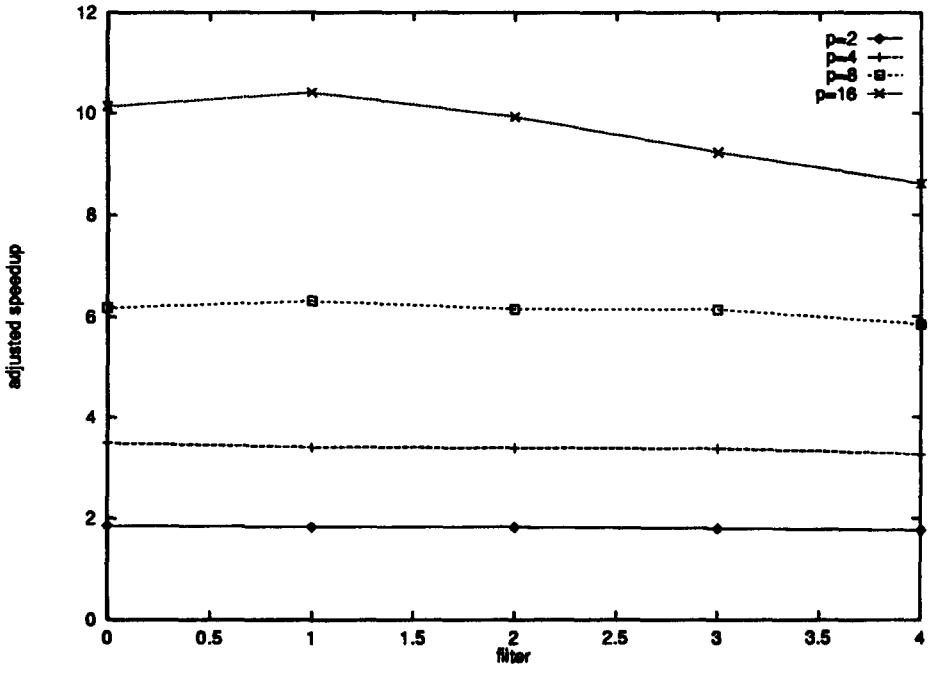

Fig. 5. Adjusted speedup versus filter parameter for large-size problems.

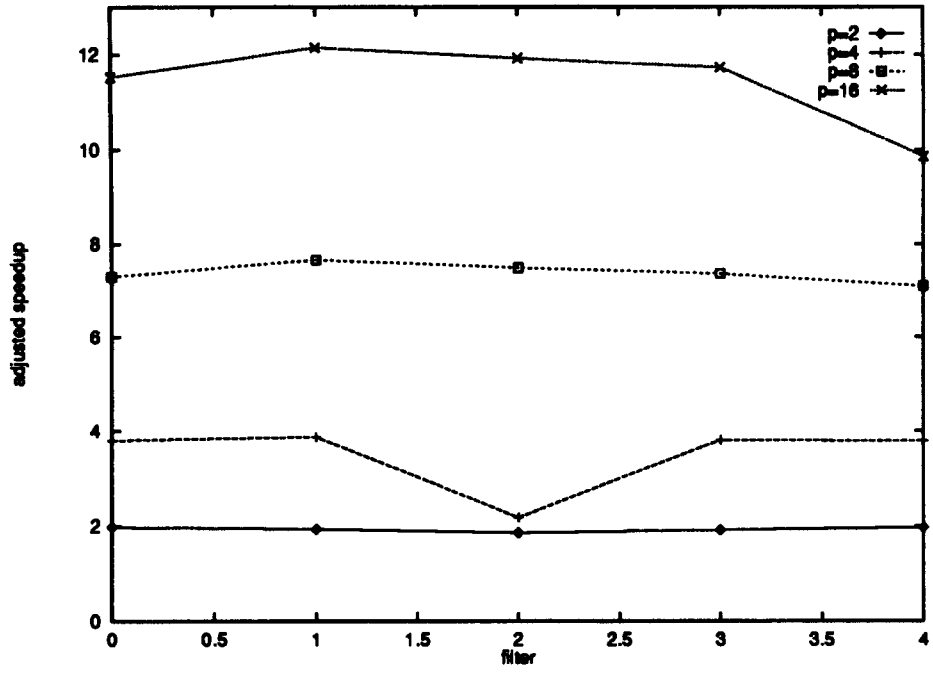

Fig. 6. Adjusted speedup versus filter parameter for actual application.

which is interesting from a practical point of view. (For a more detailed analysis of speedup results for each class of problems, see Section 5.4).

5.2. *Comparison of initialization strategies*

To compare the initialization strategies, we examine the evolution of the load balancing factor and of the adjusted speedup as functions of the number of working processes used. This is illustrated in Figs 7 and 8, which display results averaged over all 10 problems. Note that similar tendencies are observed for each problem class (medium-size, large-size and actual application), although significant differences are

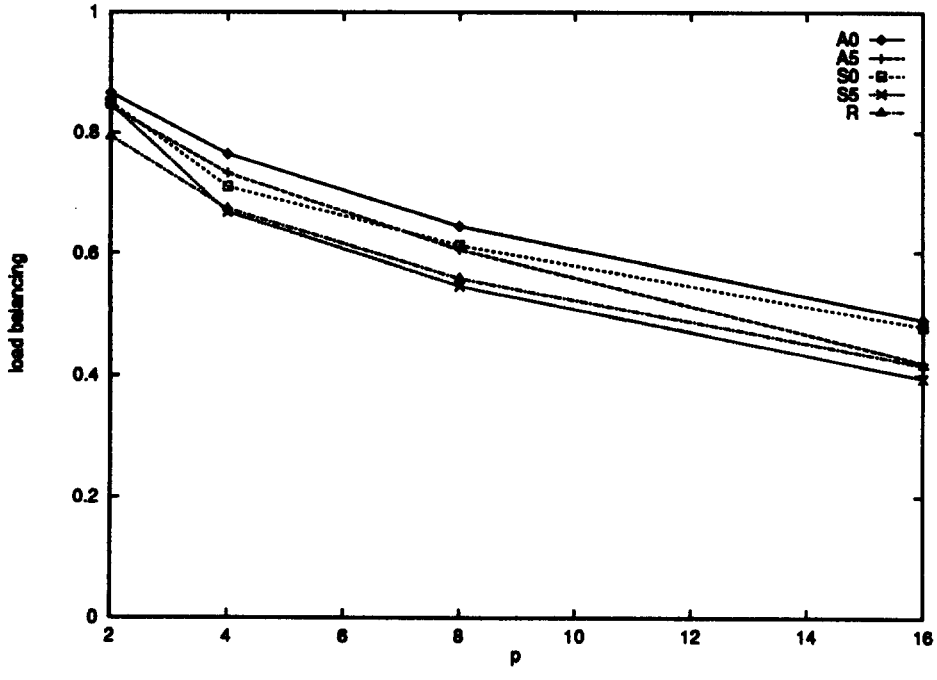

Fig. 7. Load balancing factor versus number of processes for all problems.

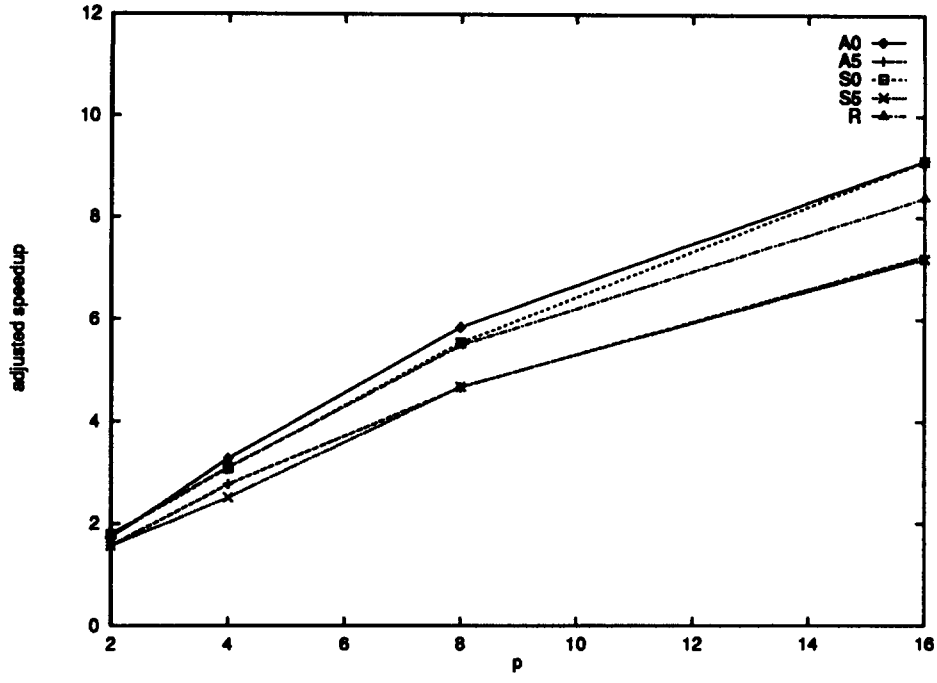

Fig. 8. Adjusted speedup versus number of processes for all problems.

displayed in the range of values of both measures obtained for each class.

Three main conclusions emerge from these figures.

- An efficient load balancing appears easier to achieve when the synchronous initialization phase is used. This is especially true when the asymmetric variant is used, which generally outperforms the symmetric one.
- A good load balancing factor does not necessarily lead to an efficient implementation. In particular, compared to the others, the A5 strategy shows competitive load balancing factors but poor adjusted speedups. This may be attributed to the significant synchronization delays incurred by this method, especially when implemented within a distributed system such as the one used here.
- Overall, the R, A0 and S0 strategies are competitive and give similar results with respect to the adjusted speedups, with A0 and SO dominating, especially when the number of processes increases.

5.3. *Search overhead*

Next, we analyze the search overhead, hereafter noted *SO*, incurred by the parallel algorithm. Since this value is clearly dependent upon the instance being solved, we first display, in Tables 2–6, the search overhead obtained by each initialization strategy for each problem instance and four values of p.

Note that, when the root initialization procedure is used, the sequential and the parallel algorithms perform the same branching and bounding operations, but in a different order. Therefore, the parallel

Table 2. Search overhead for strategy R

Problem	$p=2$	$p=4$	$p=8$	$p=16$
P1	1.40	1.50	1.64	1.87
P2	1.03	0.77	0.64	0.64
P3	0.53	0.53	0.52	0.53
P4	1.05	1.19	1.31	1.47
P5	1.01	1.01	1.05	1.11
P6	1.01	1.00	1.12	1.31
P7	1.12	1.41	1.72	1.86
P8	0.84	1.07	1.25	1.47
P9	1.00	1.00	1.06	1.08
P10	1.07	1.20	1.44	1.84

Table 3. Search overhead for strategy A0

Problem	$p=2$	$p=4$	$p=8$	$p=16$
P1	1.58	2.54	3.05	2.42
P2	0.79	1.02	1.08	1.25
P3	0.65	1.04	0.81	0.99
P4	1.11	1.38	1.20	1.54
P5	1.04	1.06	1.10	1.13
P6	1.00	0.87	1.04	1.28
P7	1.28	1.40	1.40	1.56
P8	0.93	0.93	0.94	1.26
P9	0.99	0.94	1.18	1.73
P10	1.18	1.39	1.45	1.82

algorithm may discover improving solutions faster (in terms of the number of nodes explored to this point) than the sequential method, thus leading to *positive anomalies* ($SO<1$). The contrary is also likely to occur and the parallel execution then exhibits *negative anomalies* ($SO>1$). (For a complete survey of past studies on anomalies in parallel branch-and-bound algorithms, see [9].) In this respect, the results of Table 2 are especially interesting, since they show that, for each problem, the amplitude of the anomalies observed (either positive or negative) increases as p increases. The other tables show that this behavior is exclusive to the root initialization. In particular, both the A5 and S5 strategies do not exhibit any trends with respect to p. However, the A0 and SO strategies show a similar tendency to have their search overhead increased, on average, as p increases.

We also note that the amplitude of the anomalies is generally higher when the A5 and S5 strategies are

Table 4. Search overhead for strategy A5

Problem	$p=2$	$p=4$	$p=8$	$p=16$
P1	1.44	1.12	1.16	1.96
P2	1.17	0.57	0.78	1.38
P3	0.68	0.55	0.75	1.35
P4	0.95	2.20	1.64	1.29
P5	1.14	1.04	1.13	1.04
P6	0.94	1.20	1.06	1.09
P7	1.25	1.51	3.92	1.25
P8	0.96	1.29	0.61	0.74
P9	1.93	1.48	1.25	1.22
P10	1.05	1.14	1.46	2.73

Table 5. Search overhead for strategy SO

Problem	$p=2$	$p=4$	$p=8$	$p=16$
P1	1.68	2.06	2.23	1.89
P2	0.80	0.83	0.85	0.99
P3	0.65	0.94	0.96	0.90
P4	1.11	1.30	1.74	1.50
P5	1.03	1.04	0.97	1.18
P6	1.00	0.84	1.07	1.37
P7	1.29	1.51	1.56	1.94
P8	0.94	0.87	1.12	1.39
P9	0.99	0.79	1.08	1.39
P10	1.17	1.37	1.44	1.47

Table 6. Search overhead for strategy S5

Problem	$p=2$	$p=4$	$p=8$	$p=16$
P1	1.44	1.12	1.33	2.11
P2	1.17	0.60	0.78	1.40
P3	0.68	0.78	0.86	1.35
P4	0.95	1.12	1.44	1.77
P5	1.14	0.89	1.27	1.34
P6	0.94	1.07	1.32	1.45
P7	1.25	1.56	1.52	1.56
P8	0.96	0.81	1.41	0.73
P9	1.93	1.19	1.42	2.17
P10	1.05	1.25	1.21	1.45

used. This shows that these strategies lead to follow significantly different paths than the sequential algorithm. Moreover, the trees generated are usually larger than the sequential one, especially for the actual application and the large-size problems. The same conclusion holds for all other initialization strategies, irrespective of the number of processes used. We interpret this result as an indication of the efficiency of the sequential search strategy, rather than a deficiency of the parallel one.

5.4. Speedup analysis

We now turn to analyze the speedups obtained from the parallel algorithm for each class of problems and, by using this metric, further compare the initialization strategies. Figures 9–11 show the speedup as a function of the number of working processes for, respectively, the medium-size problems (average), the large-size problems (average) and the actual application.

The following conclusions emerge from these figures.

- Very limited speedups are obtained for medium-size problems, due to the small size of the branch-and-bound trees. Also, for this class of problems, the better results obtained from the root initialization may be explained by positive anomalies.
- For large-size problems, the A5 and S5 strategies are outperformed by the others, because of the significant synchronization delays they incur. Note that the clear superiority of A5 over S5 for $p=16$ can be attributed to important differences in the size of the trees generated by each strategy. For this class of problems, it is also clear that search overhead causes a significant decrease in speedups. Nevertheless, and despite the relatively small size of the trees generated (less than 1000 nodes), interesting speedups are observed.
- For the actual application, the S5 strategy outperforms the others because it generates a smaller tree. This observation is confirmed by Fig. 12, which shows the adjusted speedup as a function of the number of processes for all initialization strategies. Here, the situation is reversed: the S5 strategy is outperformed by the others. The two figures also show that, for this problem, the speedup is severely limited by the presence of negative anomalies. Nevertheless, interesting speedups are obtained, showing the problem to be solved in less than 30 min for $p=16$.

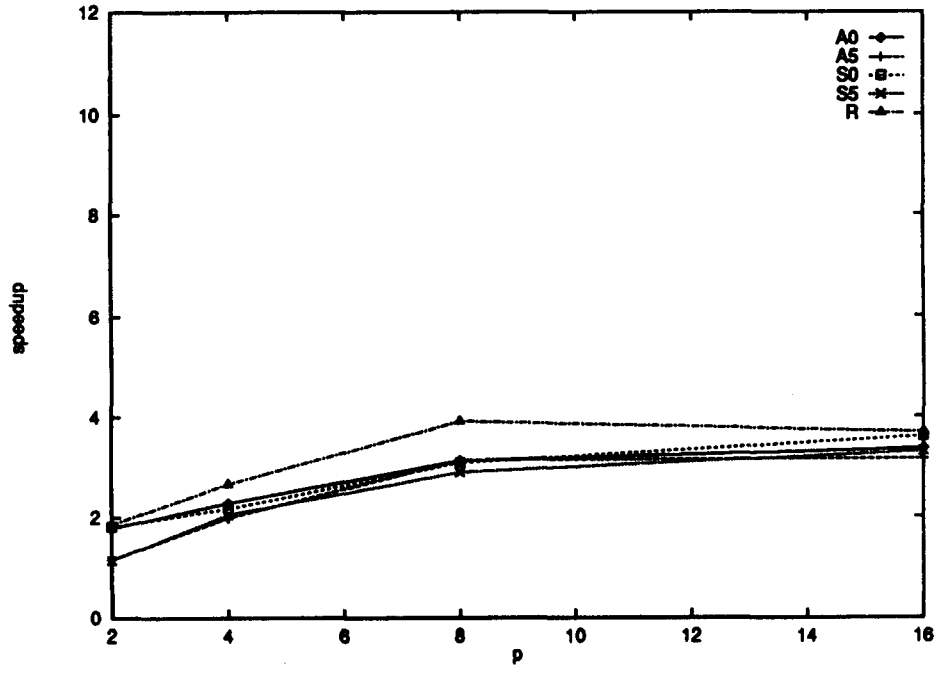

Fig. 9. Speedup versus number of processes for medium-size problems.

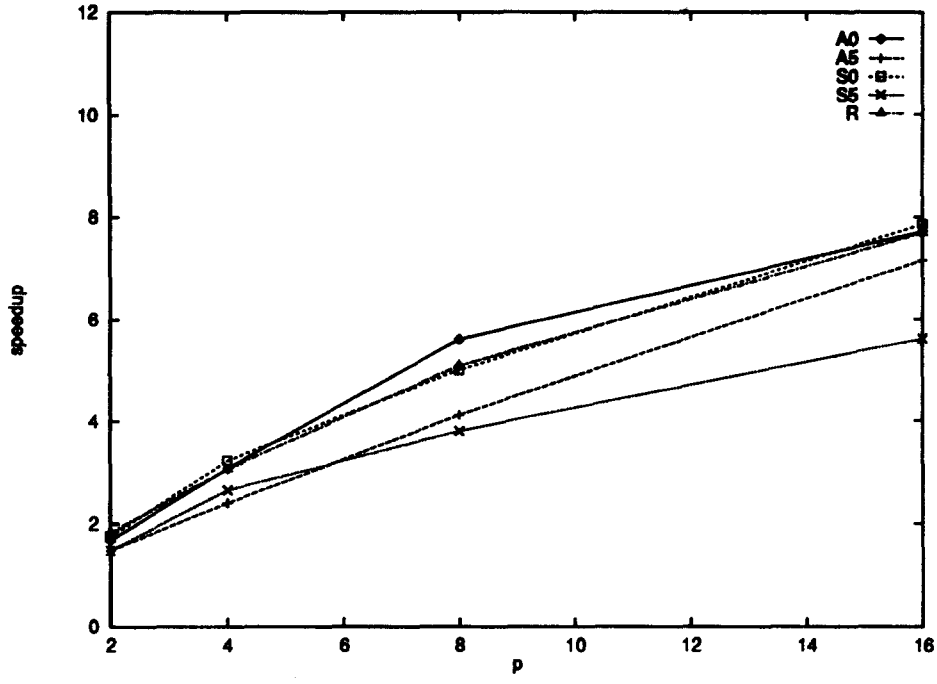

Fig. 10. Speedup versus number of processes for large-size problems.

6. CONCLUSION

We have presented a parallel branch-and-bound algorithm for solving the multicommodity location problem with balancing requirements. The algorithm is based on a depth-first branch-and-bound procedure, which is currently the most efficient sequential method for solving the problem. The parallel algorithm proceeds in two phases: a synchronous initialization and an asynchronous exploration of the branch-and-bound tree. The synchronous initialization phase can be seen as a generalization of the sequential best-first search strategy. The exploration phase consists of an asynchronous procedure where

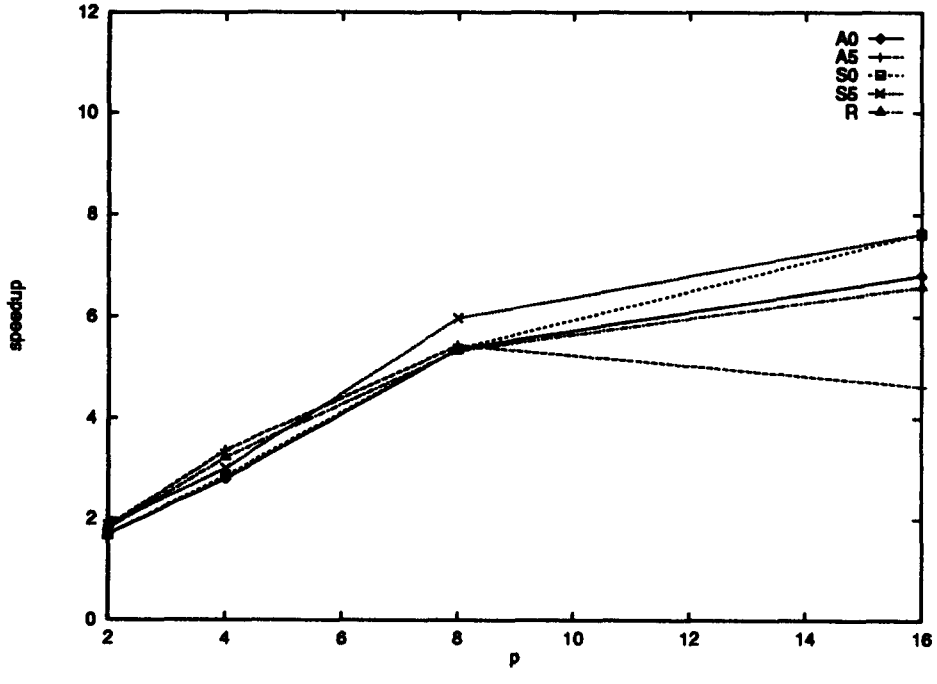

Fig. 11. Speedup versus number of processes for actual application.

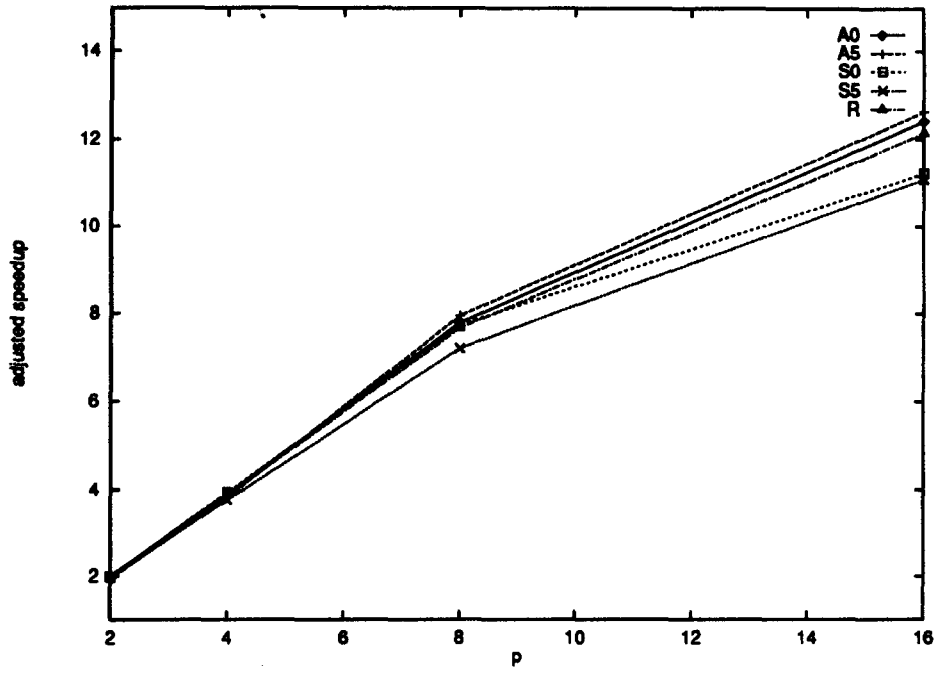

Fig. 12. Adjusted speedup versus number of processes for actual application.

each process performs its own depth-first search of a subtree. When its local pool of subproblems is empty, an idle process sends a request for work to a coordinator process that schedules the load balancing activities based on information received from the working processes.

In general, compared with the approach which initializes the algorithm by giving the root node to one process, the synchronous initialization phase yields balanced initial loads, hence decreasing the subsequent traffic of exchanged nodes. However, synchronization delays may be important, especially when the computation of bounds varies significantly from one subproblem to another, or when a distributed environment is used. Our experiments have also shown that the asynchronous exploration performs well, irrespective of the initialization phase, and achieves significant speedups on a set of ten representative test problems, despite the fact that, for many instances, the parallel algorithm generates significantly larger trees than the sequential method. We note that this is an indication of the efficiency of the sequential search strategy, since for these instances, even if different initialization strategies were tested, smaller trees than the sequential one were generally not obtained. Overall, we have demonstrated that parallel branch-and-bound techniques may help to solve more efficiently large real-size instances of the multicommodity location problem with balancing requirements. In addition, we note that the proposed parallelization approaches could prove useful in solving other problems that require time-consuming bounding procedures and which generate large search trees.

Acknowledgements—Financial support for this project was provided by N.S.E.R.C. (Canada) and the Fonds F.C.A.R. (Québec). We want to acknowledge the efforts of Nathalie Talbot and Benoît Bourbeau who helped us in carrying out the thousands of executions of the parallel algorithm, summarized in Section 5. We also want to thank two anonymous referees whose comments have helped us write a better article.

REFERENCES

1. Crainic, T. G., Dejax, P. J. and Delorme, L., Models for multimode multicommodity location problems with interdepot balancing requirements. *Annals of Operations Research* 1989, **18**, 279–302.
2. Crainic, T. G. and Delorme, L., Dual-ascent procedures for multicommodity location-allocation problems with balancing requirements. *Transportation Science* , 1993, **27**, 2 90–101.
3. Crainic, T. G., Delorme, L. and Dejax, P. J., A branch-and-bound method for multicommodity location with balancing requirements. *European Journal of Operational Research* 1993, **65**, 3 368–382.
4. Crainic, T. G., Gendreau, M., Soriano, P. and Toulouse, M., A tabu search procedure for multicommodity location/allocation with balancing requirements. *Annals of Operations Research* 1992, **41**, 359–383.
5. Gendron, B. and Crainic, T. G., A branch-and-bound algorithm for depot location and container fleet management. *Location Science* , 1995, **3**, 1 39–53.
6. Crainic, T. G., Toulouse, M. and Gendreau, M., Synchronous tabu search parallelization strategies for multicommodity location-allocation with balancing requirements. *OR Spektrum*, 1995, **17**, 2-3 113–123.
7. Crainic, T. G., Toulouse, M. and Gendreau, M., Parallel asynchronous tabu search for multicommodity location-allocation with

balancing requirements. *Annals of Operations Research* 1996, **63**, 277–299.
8. Gendron, B. and Crainic, T. G., Parallel implementations of a branch-and-bound algorithm for multicommodity location with balancing requirements. *INFOR* 1993, **31**, 3 151–165.
9. Gendron, B. and Crainic, T. G., Parallel branch-and-bound algorithms: survey and synthesis. *Operations Research* 1994, **42**, 6 1042–1066.
10. Krarup, J. and Pruzan, P. M., The simple plant location problem: survey and synthesis. *European Journal of Operational Research* 1983, **12**, 36–81.
11. Cornuéjols, G., Nemhauser, G. L. and Wolsey, L. A., The uncapacitated facility location problem, in *Discrete Location Theory*, ed. R.L. Francis and P.B. Mirchandani, Wiley-Interscience, 1990, pp. 119–168.
12. Erlenkotter, D., A dual-based procedure for uncapacitated facility location. *Operations Research* 1978, **26**, 6 992–1009.
13. Ibaraki, T., Enumerative approaches to combinatorial optimization, *Annals of Operations Research*, **10–11**, 1987.
14. Mohan, J., A study in parallel computation: the traveling salesman problem, Report CMU-CS-82-136(R), Computer Science Department, Carnegie-Mellon University, Pittsburgh, 1982.
15. Kröger, B. and Vornberger, O., A parallel branch-and-bound approach for solving a two-dimensional cutting-stock problem, Technical Report, Department of Mathematics and Computer Science, University of Osnabrück, 1990.
16. Kumar, V., Grama, A. Y. and Nageshwara Rao, V., Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing* 1994, **22**, 60–79.
17. Grigoriadis, M. D. and Hsu, T., *RNET-The Rutgers Minimum Cost Network Flow Subroutines*, Rutgers University, New Brunswick, New Jersey, 1979.
18. Van Roy, T. J. and Erlenkotter, T., A dual-based procedure for dynamic facility location. *Management Science* 1982, **28**, 10 1091–1105.