Annals of Operations Research 22(1990)271 – 292

PARALLEL BRANCH AND BOUND ALGORITHMS FOR QUADRATIC ZERO-ONE PROGRAMS ON THE HYPERCUBE ARCHITECTURE

P.M. PARDALOS and G.P. RODGERS*

Computer Science Department, The Pennsylvania State University, University Park, PA 16802, USA

Abstract

We present a parallel branch and bound algorithm for unconstrained quadratic zero-one programs on the hypercube architecture. Subproblems parallelize well without the need of a shared data structure to store expanded nodes of the search tree. Load balancing is achieved by demand splitting of neighboring subproblems. Computational results on a variety of large-scale problems are reported on an iPSC/1 32-node hypercube and an iPSC/2 16-node hypercube.

1. Introduction

A zero-one quadratic program is a problem of the form

$$\begin{aligned} &\text{minimize} \quad f(x) = c^T x + \tfrac{1}{2} x^T Q x \\ &\text{subject to} \quad x \in \{0, 1\}^n , \end{aligned} \tag{1.1}$$

where c is a rational vector of length n and Q is a rational matrix of size $n \times n$. Since $x_i^2 = x_i$ for any $x_i \in \{0, 1\}$, one can always bring problem (1.1) into the form

$$\begin{aligned} &\text{minimize} \quad f(x) = x^T A x \\ &\text{subject to} \quad x \in \{0, 1\}^n , \end{aligned} \tag{1.2}$$

where $A = \frac{1}{2} Q + D$ with $D = diagonal(c_1, \ldots, c_n)$. Problems of the form (1.1) can be easily reduced to a linear zero-one integer program [7] by introducing additional zero-one variables $y_{ij} = x_i x_j$ that satisfy the constraints

*Permanent address: IBM Corporation, General Technology Division, Burlington, Vermont 05452, USA.

$$x_i + x_j - y_{ij} \leqslant 1, \; y_{ij} \leqslant x_i, \; y_{ij} \leqslant x_j .$$

Since the matrix Q in (1.1) can be assumed to be upper triangular (with zero diagonal entries), the number of new zero-one variables introduced is $n(n-1)/2$. Similar linearization can be applied to any polynomial zero-one program. However, the number of new variables and number of constraints introduced may be high, even for small nonlinear zero-one programs.

It is well known that the general problem (1.1) is NP-hard [15]. However, many special cases have been detected that are solved in polynomial time. Such cases include problems in which the matrix Q is non-negative [17], or problems that correspond to a series parallel graph [2]. A divide and conquer algorithm in [11] is based on the use of separators of the related graphs of Q, and runs for many classes of problems in subexponential or polynomial time.

Quadratic zero-one programming has many applications in various fields [6,7]. Many algorithms are based on algebraic or enumerative techniques, and different heuristic approaches have been proposed for solving problem (1.1) [7].

In this paper, we present a branch and bound algorithm to solve (1.2) on a hypercube architecture. We assume the general case where A is indefinite. We first give an overview of branch and bound algorithms and introduce notation in section 2. In section 3, we present a sequential algorithm for quadratic zero-one programming. This is followed by an overview of the hypercube architecture. In section 5, we show how this algorithm is implemented on a distributed memory hypercube multiprocessor. In section 6, we report computational results of our implementation on two hypercube multiprocessors.

2. Branch and bound algorithms

Branch and bound is a general method that has been applied to various problems in combinatorial optimization. The main idea of a branch and bound algorithm is to decompose the given problem into several subproblems of smaller size. Each of these subproblems is also decomposed until each undecomposed problem is either solved or proved not to yield an optimal solution of the original problem. The search strategy defines the order in which partial problems are tested or decomposed. Such strategies include depth-first search, breadth-first search, best-bound search, and search strategies based on some heuristic search. Ibaraki [8] proves that depth-first search, best-bound search, and breadth-first search are special cases of heuristic search. A special variation of best-bound search (in parallel), called local best-first search, is given by Karp and Zhang [12].

Let P_0 denote an optimization problem of the form:

$$\min\{f(x) : x \in F\},$$

where the set F of feasible solutions is finite. A finite rooted tree $B = (P, E)$ is used to represent the decomposition process..Let P be the set of nodes and E the set of arcs. The root of B, denoted by P_0, corresponds to the original problem. Subproblems of P_0 are denoted by P_i for $i > 0$. Subproblem P_j is a son of P_i, denoted $(P_j, P_i) \in E$, if and only if P_j is generated from P_i by decomposition. This decomposition is known as the *branching strategy*. The terminal nodes, denoted by T, of the tree represent partial problems that can be solved without any further decomposition. Terminal nodes are also called leaves of the branch and bound search tree. The level of P_i, denoted by $L(P_i)$, is the length of the path from the root P_0 to the node P_i. For the root node, we have $L(P_0) = 0$ and for the terminal nodes, we have $L(P_0) = N$, where N is the maximum depth of the branch and bound tree.

Let $Opt(P_i)$ be the optimal solution of P_i and let $f(P_i)$ denote its value. If P_i is infeasible, then $f(P_i) = \infty$. If P_i is decomposed into subproblems $P_{i_1}, \ldots, P_{i_k}$, then

$$f(P_i) = \min\{f(P_{i_j}) : j = 1, \ldots, k\} ,$$

$$Opt(P_i) = \cup\{Opt(P_{i_j}) : f(P_i) = f(P_{i_j}),\ j = 1, \ldots, k\} .$$

Note that the above relations imply that $f(P_i) \leqslant f(P_{i_j})$ for $(P_i, P_{i_j}) \in E$. It is also clear that $f(P_i)$ is not known until P_i is solved. For pure 0-1 problems, a subproblem is usually decomposed into no more than two subproblems. For unconstrained integer programming problems, all solutions are feasible.

A branch and bound algorithm attempts to solve P_0 by examining a small number of nodes in P. This is done be eliminating those nodes P_i that are determined by a test to be incapable of yielding an optimal solution to P_0. This test uses a *lower-bounding function* $g : P \to R \cup \{\infty\}$, where R is the set of real numbers. This function satisfies three conditions:

(1) $g(P_i) \leqslant f(P_i)$, for $P_i \in P$.

(2) $g(P_i) = f(P_i)$ for the terminal nodes $P_i \in T$.

(3) $g(P_j) \geqslant g(P_i)$ if P_j is a son of P_i in B.

At any point during the execution of a branch and bound algorithm there exists a set of subproblems S, where $S \subset P$, that have been generated but not yet examined. Let $|S|$ denote the number of subproblems in S. The subproblems in S are also referred to as live nodes. The *incumbent* Z is defined as the lowest objective function value among all known feasible points. The value Z is changed to $f(P_i)$ when $P_i \in T$ and $f(P_i) < Z$. If $g(P_i) > Z$, then P_i is removed from S since further decomposition can only yield larger function values than Z. This process is called *pruning* or *fathoming*. The choice of the next subproblem to examine is determined by the *search strategy*.

The best-bound search strategy chooses the subproblem with the smallest lower bound $g(P_i)$, for $P_i \in S$. When two or more problems have the same smallest lower bound γ, we assume that the best-bound strategy chooses problem $P_i \in S$ such that $g(P_i) = \gamma$ and for all $P_j \in S$ with $g(P_j) = \gamma$, we have that $L(P_j) \leqslant L(P_i)$. The motivation for best-bound search strategy is to quickly find a low value for Z to improve the fathoming power of the algorithm since at a terminal node $g(P_i) = f(P_i)$. A limitation of the best-bound strategy is that $|S|$ may become very large.

The breadth-first search strategy chooses the subproblem from S that was first added to S. This is also known as the FIFO strategy for first-in-first-out. This strategy also suffers from the limitation that $|S|$ may become very large.

The depth-first search strategy chooses the subproblem from S that was last added to S. This is also known as the LIFO strategy for last-in-first-out. The motivation here is to limit the storage required for S. Let M_i be the maximum number of branches from node P_i and N be the maximum depth of the branch and bound tree. It is easy to see that

$$|S| \leqslant N (\max_i M_i).$$

For pure 0-1 programming, we have $M_i = 2$ (for $x_i = 1$ and $x_i = 0$) and N is the number of variables. Hence, $|S| \leqslant 2N$. An efficient implementation of depth-first search for pure 0-1 programming can be done with a stack of no more than $n + 1$ entries.

In summary, a branch and bound algorithm is characterized by three attributes:

(1) The branching strategy.
(2) The lower-bounding function.
(3) The search strategy.

The objective of an *exact* branch and bound algorithm is to find a global optimum. However, many branch and bound implementations have provisions for terminating execution after some specified time limit, since a complete search of the branch and bound tree may be impractical. When execution is terminated, the incumbent can not be guaranteed to be optimal. In this case, the branch and bound process is called a *heuristic* search, since the process was not allowed to complete.

The objective of a heuristic search is to find a good solution quickly without necessarily searching the entire space. The objective of an exact search is consistent with a heuristic search in that an early solution increases the fathoming power of the bounding function. However, for our problem, we will see that these two objectives are not completely consistent. In the next section, a branch and bound algorithm is given whose primary objective is to verify the exact solution. As a heuristic search whose measure is early discovery of the solution, the proposed algorithm performs poorly. However, in the presence of a good initial incumbent, it is an extremely good exact search whose measure is the speed at which the solution is verified [18].

3. A branch and bound algorithm for quadratic programming

A pure 0-1 program is decomposed by selecting a variable x_i from a list of free variables, creating one subproblem by setting x_i to 0, and creating another subproblem by setting x_i to 1. The variable x_i is then removed from the list of free variables, called the *free list*, and it is placed in the *fixed list*. When the free list is empty, all variables are fixed and the subproblem represents a complete assignment of values.

The branch and bound tree has a potential size of $2^{n+1} - 1$ nodes. This is prohibitively large for even moderate size problems. Hence, the pruning process is critical. There are two categories of pruning rules used: the lower bound rule and forcing rules. The lower bound rule says that if the value of a lower bound function g for a given subproblem exceeds a known optimal, then that subproblem can only yield a suboptimal solution.

For problem (1.2), an easy to compute lower bound function is chosen. Consider that for any vector x, the function value $f(x)$ is merely a sum of entries a_{ij}, where $x_i = 1$ and $x_j = 1$. Thus, a lower bound function that meets all three lower-bound criteria is the sum of all negative entries, *minus* the sum of negative entries known to be excluded from the function value by variables fixed to 0, *plus* the sum of positive entries known to be included in the function value by variables fixed to 1. Let *lev* be the level in the search tree (the number of fixed variables). Initially, $lev = 0$. Let $p_1, \ldots, p_{lev}$ be the indices of the fixed variables and $p_{lev+1}, \ldots, p_N$ be the indices of the free variables; then the lower bound g is defined as follows:

$$g = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij}^{-} - \left(2 \sum_{i=1}^{lev} \sum_{j=i+1}^{n} a_{p_i p_j}^{-}(1 - x_{p_i}) + \sum_{i=1}^{lev} a_{p_i p_i}^{-}(1 - x_{p_i})\right)$$
$$+ \sum_{i=1}^{lev} \sum_{j=1}^{lev} a_{p_i p_j}^{+} x_{p_i} x_{p_j}, \qquad (3.1)$$

where $a_{ij}^{-} = \min\{0, a_{ij}\}$ and $a_{ij}^{+} = \max\{0, a_{ij}\}$ are the negative and positive coefficients of A, respectively. The lower bound pruning rule says that if $g \geqslant OPT$, where OPT is the known incumbent objective function value, then the subproblem should not be further decomposed.

Forcing rules can be used to generate only one branch for a given variable if certain conditions exist. This is also known as *preprocessing* the subproblem. A variable may be forced if it can be shown that the alternate value can only yield suboptimal solutions. Once a variable is forced, it is considered fixed and it is handled exactly the way that a fixed variable is handled except that the alternate subproblem does not exist.

For problem (1.2), variables may be forced by examining the range of the gradient of the *continuous* objective function. In [18], it is shown that if the continuous objective function is always increasing for a free variable x_i in the *continuous* range $x_i \in [0,1]$, then x_i may be forced to 0. Likewise, if the function is always decreasing, then the variable may be forced to 1. This rule is implemented by calculating the range of continuous partial derivatives of free variables in the unit hypercube as determined by the fixed variables. The lower and upper bounds of the continuous partial derivatives of the free variables are given by lb_{p_i} and ub_{p_i}.

$$lb_{p_i} = \sum_{j=1}^{lev} a_{p_i p_j} x_{p_j} + \sum_{\substack{j=lev+1 \\ j \neq i}}^{n} a^-_{p_i p_j} + a_{p_i p_i} \quad \text{for } i = lev+1, \ldots, n \tag{3.2}$$

$$ub_{p_i} = \sum_{j=1}^{lev} a_{p_i p_j} x_{p_j} + \sum_{\substack{j=lev+1 \\ j \neq i}}^{n} a^+_{p_i p_j} + a_{p_i p_i} \quad \text{for } i = lev+1, \ldots, n\,. \tag{3.3}$$

Actually, (3.2) and (3.3) give the range for the partial derivatives of the equivalent zero-diagonal function (replace any term x_i^2 by x_i). In [18] it is shown that the zero-diagonal function has the same zero-one values but whose continuous characteristics are such that the range of the partial derivative is minimized.

After lb and ub are calculated, the two rules used to force variables are as follows:

$$\text{if } lb_{p_i} \geqslant 0, \text{ then } x^\star_{p_i} = 0 \text{ for } i = lev+1, \ldots, n\,. \qquad \text{(Rule 3.1)}$$

$$\text{if } ub_{p_i} \leqslant 0, \text{ then } x^\star_{p_i} = 1 \text{ for } i = lev+1, \ldots, n\,. \qquad \text{(Rule 3.2)}$$

Forcing variables is extremely desirable since it considerably reduces the size of the search tree. In fact, the rule that is used to select a variable to branch on when none can be forced is to choose the variable that is least likely to be forced in subsequent levels of the search tree. This leaves other variables that can be more easily forced at a lower level. By using the permutation vector p, it is possible to select variables in any order. The next variable selected is the one whose bounds for the range of the partial derivative are farthest from zero according to rule 3.3.

$$\text{Branch on } x_{p_i}, \text{ where } \delta_i = \max_k \{\min(-lb_{p_k}, ub_{p_k}), k = lev+1, \ldots, n\}. \qquad \text{(Rule 3.3)}$$

When a branch occurs, two subproblems are generated: one for $x_{p_i} = 1$ and one for $x_{p_i} = 0$. The subproblem to expand first is decided by the searching strategy. To determine this strategy, we realize that for our particular problem, a vertex of the branch and bound tree can be evaluated in $O(n)$ time, which is very fast. Since large-scale problems will have many vertices, it is impractical to store all the vertices that may be required with a best-first or breadth-first search strategy. Consequently, depth-first search is used to minimize storage requirements. Little is sacrificed by using depth-first search, since a good initial heuristic ensures that the incumbent rarely changes.

Depth-first search fixes the level of the search tree to expand first, but does not fix which branch to expand at that level. The value assigned to the variable selected by rule 3.3 defines this branch. A partial best-first strategy can be used by choosing the value that decreases g the most. However, if this branch results in no change to the incumbent, then the choice would have been irrelevant.

We are now able to discuss the details of algorithm 3.1, which solves the quadratic 0-1 problem given by equation (1.2). The key variables are summarized in table 3.1.

Table 3.1

Key variables for algorithm 3.1

a_{ij}	The elements of the input matrix A
n	The dimension of the input matrix A
OPT	The incumbent minimum which becomes the solution
x^*	The incumbent minimizer
lev	The current level of the branch and bound tree
p	The permutation vector
$p_1, \ldots, p_{lev}$	The indices of the fixed variables in the current subproblem
$p_{lev+1}, \ldots, p_n$	The indices of the free variables in the current subproblem
$x_{p_1}, x_{p_2}, \ldots, x_{p_{lev}}$	The fixed variables
$x_{p_{lev+1}}, x_{p_{lev+2}}, \ldots, x_{p_n}$	The free variables
g	The lower bound for $f(x)$
lb	The lower bound of $\nabla f(x)$ in $[0,1]^n$
ub	The upper bound of $\nabla f(x)$ in $[0,1]^n$
$NSUBP$	The number of subproblems solved
$MAXS$	The limit on the number of subproblems to solve

ALGORITHM 3.1

Depth-first branch and bound algorithm for quadratic 0-1 programming

```
Procedure Q01($A, x^\star, OPT$)
$OPT \leftarrow$ best known minimum from heuristic
$x^\star \leftarrow$ best known minimizer from heuristic
$p_{[1,n]} \leftarrow [1,n]$
push$(-1,$ stack$)$
$MAXS \leftarrow$ value based on CPU resource limit
$lev \leftarrow 0$
$NSUBP \leftarrow 0$,
while $lev \neq -1$ and $NSUBP < MAXS$ do
    Calculate lower bound $g$
    if $g \geqslant OPT$ or $lev = n$ then
        if $g < OPT$ then
            $OPT \leftarrow g$
            $x_i^\star \leftarrow x_i, i = 1, \ldots, n$
        endif
        pop $(lev,$ stack$)$
        if $(lev \neq -1)$ then $x_{p_{lev}} \leftarrow 1 - x_{p_{lev}}$
        $NSUBP \leftarrow NSUBP + 1$
    else
        $[lb_{p_i}, ub_{p_i}] \leftarrow$ range of $\partial f / \partial x_{p_i}$ over $x \in [0,1]^n$ for $i = lev+1, \ldots, n$
        if $lb_{p_i} \geqslant 0$ or $ub_{p_i} \leqslant 0$, for some $i$, $i = lev+1, \ldots, n$ then
            if $ub_{p_i} \leqslant 0$ then $x_{p_i} \leftarrow 1$ else $x_{p_i} \leftarrow 0$
        else
            $i \leftarrow j$ where $\delta_j = \max_k \{\min(-lb_{p_k}, ub_{p_k}), k = lev+1, \ldots, n\}$
            $x_{p_i} \leftarrow$ 0 or 1 depending on value that increases $g$ least
            push $(lev+1,$ stack$)$
        endif
        $lev \leftarrow lev + 1$
        $p_{lev} \leftrightarrow p_i$
    endif
endwhile
```

The expanded subproblems (S) are stored on a stack. This stack has a maximum depth of $n + 1$, where n is the dimension of the problem. Furthermore, the algorithm is written in such a way that the only value that is required to be saved on the stack is the level where a branch occurs. As the branch and bound tree is descended, *lev* is changed and indices are swapped in p to represent the order that variables are selected. Thus, the free and fixed variable lists are implicitly changed.

The first part of algorithm 3.1 is the initialization of the branch and bound procedure. As a stopping criterion, -1 is initially placed on the stack. The maximum number of subproblems to solve, *MAXS*, is initialized to some limit based on CPU resources at line 5. The size of the branch and bound tree is characterized by the number of subproblems *NSUBP*. Hence, $NSUBS > MAXS$ is an additional stopping criterion which can be used to control the amount of computer time available to the problem.

It should be assumed that a good heuristic is used to initialize the values for *OPT* and x^* at lines 1 and 2. For this implementation, two heuristics are used: the gradient midpoint method and a greedy method. The better of the two values is used to initialize *OPT*. These heuristics are discussed in [18].

Every iteration of the while-loop at line 8 represents one vertex of the branch and bound tree. The lower bound of the objective function which is denoted by the symbol g is calculated at line 9 according to equation (3.1). Line 10 states that, if pruning is required ($g \geqslant OPT$) or a leaf node has been reached ($lev = n$), then the algorithm is at a terminal node. If the test at line 11 ($g < OPT$) is true, then it is implied that $lev = n$ and a new minimizer has been discovered. The incumbent value *OPT* and the minimizer are updated in lines 12 and 13. The next node to search is obtained by popping a new level from the stack (line 15) and changing the value of the binary variable associated with that level (line 16). The change to the free and fixed variable lists in p is implicit. The statistic *NSUBP* is updated at line 17 to reflect the fact that another subproblem has been solved.

As the algorithm descends depth-first, the range of partial derivatives of free variables is calculated (line 19) according to (3.2) and (3.3). Line 20 represents the test to see if any free variables can be forced by the gradient rule, while line 21 is the selection of the forced value. If a variable can be forced by the gradient, i is set to the index into p that contains the index of the variable to be forced. If a branch is required (i.e. preprocessing is complete), the least likely variable to force is chosen by rule 3.3 (line 23). For certain classes of problems, it is important to use the implicit tie breaking rule, which is not explicitly given by line 23. After the variable has been chosen, it is then determined which branch, $x_{p_i} = 1$ or $x_{p_i} = 0$, results in a smaller lower bound. The alternate subproblem is saved by putting the level associated with the branch variable on the stack (line 25). Whether a variable is branched on or a variable is forced, subsequent levels of the current subproblem must consider that variable as fixed. Lines 27 and 28 represent a variable for the next subproblem by increasing *lev* by one and swapping the appropriate index in the permutation vector p.

The calculation of the gradient bounds and the objective lower bound dominates the computational requirements. In practice, the calculations of the bounds are done more efficiently than suggested by eqs. (3.1), (3.2), and (3.3). Those formulas require $O(n^2)$ operations per vertex. All of the bounds can be updated in less than $2n$ additions per vertex. This is done by using the bounds at the previous level. Let lb^{lev} and ub^{lev} represent the lower and upper bound gradient vectors at level *lev* in the branch and bound tree. Recall that $x_{p_{lev}}$ was the variable that was last fixed in the branch and bound tree. The calculation of the new gradient bounds at line 19 is done in $O(n)$ operations by using algorithm 3.2.

A similar procedure is used to update the function lower bound g. If the variable that was last fixed was fixed to 1, then it is necessary to increase all lower bounds by any positive coefficients that become part of the minimum. Likewise, all upper bounds need to be decreased by any negative coefficients that become part of the minimum. The opposite is done if the last variable that was fixed was fixed to 0.

ALGORITHM 3.2

Efficient update of gradient bounds

```
if $x_{p_i} = 1$ then
    $lb_{p_i}^{lev+1} \leftarrow lb_{p_i}^{lev} + a_{p_i p_{lev}}^{+}$   for $i = lev + 1, \ldots, n$
    $ub_{p_i}^{lev+1} \leftarrow ub_{p_i}^{lev} + a_{p_i p_{lev}}^{-}$   for $i = lev + 1, \ldots, n$
else
    $lb_{p_i}^{lev+1} \leftarrow lb_{p_i}^{lev} - a_{p_i p_{lev}}^{-}$   for $i = lev + 1, \ldots, n$
    $ub_{p_i}^{lev+1} \leftarrow ub_{p_i}^{lev} - a_{p_i p_{lev}}^{+}$   for $i = lev + 1, \ldots, n$
endif
```

Separate vectors for lb and ub are not required at each level. Only when a branch occurs is there a need to save separate vectors for lb^{lev} and ub^{lev}. If memory is at a premium, then lb^{lev+1} and ub^{lev+1} can be completely recalculated according to (3.1) and (3.2). Furthermore, the entire gradient range need not be calculated or stored. Only the range of the partial derivatives of $x_{p_{lev+1}}$ through x_{p_n} are needed.

Algorithm 3.1 is designed so that the loop at statement 8 can easily be restarted by saving the arguments stack, *p*, *lev*, and *x*. This is helpful if *NSUBP* exceeds the threshold *MAXS* and the user would like to apply more resources to the same problem. The termination condition $NSUBP < MAXS$ could easily be replaced with another more explicit CPU time constraint. However, the number of subproblems directly correlates to the CPU requirements. The condition $NSUBP < MAXS$ is used here to ensure portability of code between various computer systems.

The restartability feature and nonrecursive nature of the algorithm allow it to be easily tailored for execution on a parallel processor. This is accomplished by having multiple activations of the algorithm applied to separate subproblems which are represented by various states of the arguments; stack, p, lev, and x. The stack is used to simulate recursion since current software support for most parallel processing architectures does not include recursive languages. These details are discussed further in section 5.

4. Overview of the hypercube multiprocessor architecture

Processors on a hypercube architecture communicate through message passing channels. This allows practical architectures with considerably more processors than shared memory multiprocessors. The branch and bound algorithm was implemented on two separate Intel iPSC hypercube multiprocessors. The first was a 32-node (D5) first generation iPSC which used ethernet communications with Intel 80286 microprocessors. The other was a 16-node (D4) second generation iPSC which used Intel's own protocol for faster communications. The second generation iPSC uses Intel 80386 microprocessors which are substantially faster than the predecessor 80286 microprocessors. Before discussing the hypercube algorithm, an overview of hypercube multiprocessors is given.

A hypercube multiprocessor or (d-cube) has $P = 2^d$ processors, where d is the dimension that indicates the size of a particular system. Hence, the number of processors in a hypercube is always a power of 2. Each processor has a unique processor identifier consisting of d bits. Processors are numbered using a gray code so that neighboring processors differ in one, and only one, bit. Each processor has its own memory. This is why the system is called a distributed memory multiprocessor. Information is shared only by the sending and receiving of messages.

Hypercubes can be built recursively. To build a hypercube of dimension d, two hypercubes of dimension $d - 1$ are connected via their respective components with a communication channel. A 0-dimension hypercube is a uniprocessor with no neighbors. Processors that are directly connected by a communication channel are called *neighbors.* Each processor in a d-dimensional hypercube has d neighbors. A d-dimensional hypercube has a total of $d \times 2^{d-1}$ total communication channels. A *subcube* of a d-cube is a subgraph of the hypercube topology that is a hypercube itself, with dimension less than or equal to d. Figure 4.1 shows hypercube configurations for $d = 0, 1, 2, 3$.

Each processor on a hypercube may run one or more processes. However, it is most common to design multiprocessor applications with one process per processor. The term *homogeneous multiprocessing* is used to describe identical (with the

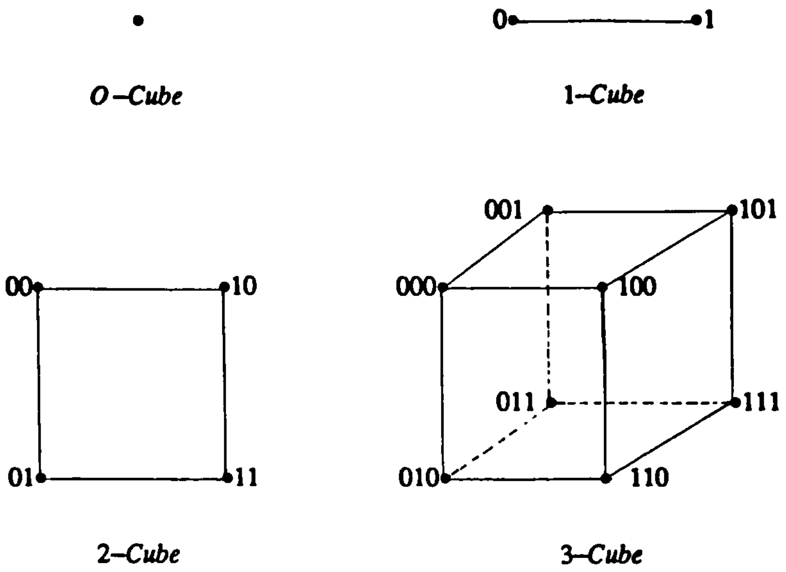

Fig. 4.1. Hypercube configurations for $d = 0, 1, 2,$ and 3.

exception of the process identifier) processes running on every processor. For the rest of the discussion, assume that each processor is running one process and that all processes are homogeneous. This is the simplest and most common use of commercial hypercube applications. Furthermore, assume that hypercube algorithms use a complete subcube of the available hypercube. That is, the algorithms are written assuming that there are P processors, where P is a power of 2.

Processors may exchange information over a communication channel via *send* and *receive* operations. A send operation transmits data to a desired processor. A receive operation is either *synchronous* or *asynchronous*. A synchronous receive blocks the process from continuing until the desired message is available on a communication channel. An asynchronous receive allows the process to continue. A process executing an asynchronous receive may continually test via a *probe* operation to see if the desired message transmission has completed. Probe operations may also be used to test if any message is waiting to be received. For more detailed information on hypercubes, see Colley et al. [4].

5. Parallel algorithm for zero-one programming

Branches of the search tree can be considered as independent subproblems and can be evaluated in parallel. Roucairol [19] parallelizes a branch and bound algorithm for the quadratic assignment problem on the CRAY XMP. The algorithm uses shared memory to store the expanded nodes of the search tree in a heap. This heap also allows for multiple searching strategies. However, this method is limited by the storage capacity for the heap. Several other authors (e.g. [3]) also use a shared data structure to store subproblems for parallel execution. Abdelrahman and Mudge [1] were the first of many ([5,14,16,20]) to propose branch and bound algorithms on hypercubes. Abdelrahman and Mudge [1] proposed two parallelization methods. The first method maintains a centralized list of subproblems and a manager. The second method outperforms the first by distributing the list of subproblems and balancing the load among neighboring processors. When all neighboring processors are idle, the algorithm "guesses" to terminate.

Our algorithm is based on the idea of splitting the branch and bound tree into exactly as many subproblems as there are processors. The subproblems are then allowed to execute for a specified number of vertices of the branch and bound search tree. If a subproblem completes its search within the allotted number of vertices, then an unfinished subproblem is split and assigned to a free processor.

ALGORITHM 5.1

General parallel algorithm for branch and bound

```
1     Assign initial problem to processor 1
2     while (any processor is not assigned a subproblem) do
2.a       Find a juncture for an active subproblem
2.b       Split the subproblem at the juncture
2.c       Assign new subproblem to a free processor
      endwhile
3     (Re)start depth-first branch and bound processes in parallel. These will
      terminate after a specified number of vertices have been visited ($MAXV$).
4     wait for all processors to finish their subproblems
      or exceed $MAXV$ vertex evaluations.
5     Obtain the best minimum of all subproblems and provide
      this value to all subproblems as the new best minimum.
6     if ALL subproblems are unfinished goto step 3
7     if ANY subproblems are unfinished goto step 2
8     STOP
```

Recall that with depth-first search subproblems are removed from the top of the stack when a terminal vertex is encountered. A location in the tree where a branch exists is called a *juncture*. Each stack location defines a juncture. Hence, depth-first search removes junctures from the bottom of the tree that are at the top of the stack. However, splitting a subproblem for parallel processing involves removing the subproblem from the *bottom* of the stack and assigning it to a free processor. If the stack is empty, the branch and bound process must be restarted to find a juncture. In summary, parallelism is applied to the breadth of the branch and bound tree while sequentially a depth-first search is performed. Algorithm 5.1 gives the general outline of our approach.

The *parallel phase* of the algorithm is asynchronous. However, the algorithm can be described as synchronous in the sense that each processor waits for all other processors to finish (step 4) before proceeding with the *synchronization phase* (steps 4, 5, 6, 7, and 2). Each pass through the parallel phase is defined as a *cycle*.

The selection of *MAXV* is a critical part of the algorithm since it is the processing *granularity*. If *MAXV* is chosen too large, the result will be poor processor utilization because of ineffective load balancing due to large variations in the size of subproblems. If *MAXV* is chosen too small, the processing required for the synchronization phase will overcome the useful work being done in step 3.

The synchronization phase is important for efficient parallelization. It must be done quickly to achieve good processor efficiency. This phase is handled quite differently for shared memory systems than it is for distributed memory systems. For a shared memory implementation of branch and bound, see [18]. If a master-slave relationship is used to implement branch and bound on the hypercube, a communications bottleneck would develop. This would occur because more splitting and subproblem transmission is required when there are more processors, resulting in excessive communication. Abdelrahman and Mudge [1] experienced a similar bottleneck when they implemented their centralized list algorithm. Another disadvantage is that subproblems would have to execute on processors that are not connected directly to the mother process, causing further communication delays. With our hypercube algorithm, the entire splitting process is done in $\log_2 p$ steps, where p is the number of processors. This is done efficiently by taking advantage of the hypercube topology. Global communication and synchronization are also accomplished at the same time, allowing for the broadcast of any new minimizer and for the detection of a correct termination condition.

Algorithm 5.2 is executed as a single process on each processor in the hypercube. Consequently, there is no mother-child relationship as there is in the shared memory implementation. Processes are independent except for the fact that a process may suspend itself while waiting to communicate with another processor. The initial problem is sent to processor 0. All processors have a busy status called *My_status*. Initially, processor 0 is marked busy (*My_status* = BUSY) and all other processors are marked free.

The call to subroutine SYNC (algorithm 5.3) in step 8 performs all synchronization and subproblem splitting. The following two assumptions can be made about subroutine SYNC to show that this algorithm will terminate correctly.

ASSUMPTION 1

SYNC may only change *My_status* from FREE to BUSY. When this happens, SYNC has acquired a subproblem from another processor.

ALGORITHM 5.2

Branch and bound on hypercube

```
1   if (My_id = 0) then
2       Receive the original problem.
3       Initialize subproblem.
4       $My\_status \leftarrow$ BUSY
5   else
6       $My\_status \leftarrow$ FREE
7   endif
8   call SYNC($My\_status$) "algorithm 5.3"
9   if ($My\_status$ = BUSY) then
10      (Re)start subproblem for $MAXV$ vertices.
11      if (Subproblem was solved) then
12          $My\_status \leftarrow$ FREE
13      endif
14      goto step 8
15  endif
16  STOP
```

ALGORITHM 5.3

Synchronization on hypercube

```
1   for $i = 1$ to $d$
2       $N\_id \leftarrow$ XOR(My_id, $2^{i-1}$)
3       if (Bit $i$ of My_id is 1) then
4           Send $My\_status$ to $N\_id$
5           Receive $N\_status$ from $N\_id$
6       else
7           Receive $N\_status$ from $N\_id$
8           Send $My\_status$ to $N\_id$
9       endif
10      if ($My\_status \neq N\_status$) then
11          if ($My\_status$ = BUSY) then
12              Split subproblem
13              Send subproblem to $N\_id$
14          else
15              Receive subproblem
16              $My\_status \rightarrow$ BUSY
17          endif
18      endif
19  endfor
```

ASSUMPTION 2

If any process has *My_status* = BUSY before calling SYNC, then all processes will have *My_status* = BUSY after calling SYNC.

By these assumptions, the initial call to SYNC will cause the initial problem in processor 0 to be split among all processors in the cube. Thus, each processor's call to SYNC will set *My_status* to BUSY. Each processor will then simultaneously start or resume its subproblem (step 10). This is equivalent to the cycle that occurs in step 3 of algorithm 5.1. After the cycle, each processor returns control (step 14) to SYNC to perform any necessary splitting or to determine if a termination condition exists. It is easy to see that if SYNC follows the above assumptions, termination will occur for all processors after they all solve a subproblem in a simultaneous cycle.

The details of subroutine SYNC are now discussed to show that assumptions 1 and 2 are valid. The objective was to make SYNC perform very quickly by exploiting the hypercube architecture. Algorithm 5.2 was implemented using 2 or 3 synchronous messages per neighbor. Each processor only communicates with its d neighbors, where $d = \log_2 p$ is the cube dimension and p is the number of processors. The exclusive OR (XOR) operation in step 2 is used to identify the id of the neighbor (differing by one bit) with whom to communicate next. Steps 3–9 are used to communicate that status with a neighbor. Although not shown here, this communication can also be used to exchange information about potential new incumbents. The lower of the two minima is kept by each processor. If one processor is free, it will initiate a split of its subproblem (step 12).

Both global communication and subproblem distribution are done in $\log_2 p$ steps. In the shared memory implementation, there are potentially $p - 1$ steps since that is how many splits might be required by the mother process to activate free processors. The tradeoff is that the shared memory algorithm is able to choose the larger subproblems to split, thus allowing for slightly better load balancing.

It is easy to see that assumption 1 holds because *My_status* is only changed in step 16 to BUSY after it receives a new subproblem. To show assumption 2 holds, consider the propagation of "BUSY" processors from just one of the processors that was "BUSY" before entering SYNC. After the first step, two processors are busy, the original and its neighbor, whose id differs in the first bit. In the next step, those two processors will propagate the busy state to two more processors whose ids differ in the second bit. After each step, 2^i processors will be busy. Therefore, after d steps, all $2^d = p$ processors will be busy. Sometimes, the process of finding a juncture will actually solve the subproblem because the branch and bound procedure must be restarted if no juncture exists. This is more likely to occur when subproblems are very small. When this happens, two *phantom* subproblems are created, one for the original subproblem and one for the process requesting a subproblem. There is no actual work to do on phantom subproblems. This condition is created only to preserve the busy status which will ensure that assumption 2 remains valid. If a process tries to split a phantom subproblem, then it creates another phantom subproblem for the requesting process.

In the next section, we show results from running this algorithm on two separate commercial hypercube multiprocessors. We also introduce a slight modification to the algorithm to allow for preemptive synchronization, which avoids unnecessary synchronizations and reduces idle time as a result of phantom subproblems.

6. Computational results

In this section, we show computational results from running the algorithm described in the previous section on two hypercube systems. The first is the iPSC/1 32-node hypercube at the University of Colorado. The second is the iPSC/2 16-node hypercube at Pennsylvania State University. The next subsection introduces a slight modification to the algorithm, followed by computational results with this algorithm.

Speedup and efficiency are calculated as follows:

$$\text{Speedup} = \frac{T_1}{T_p}, \quad \text{Efficiency} = \frac{\text{Speedup}}{p},$$

where T_p is the elapsed time to solve the problem on p processors, and T_1 is the elapsed time to solve the problem on one processor. To ensure the best sequential time (T_1), we set *MAXV*, the granularity, to infinity when solving the problem on one processor. This prevents any interruption which cause unnecessary overhead.

For accuracy purposes, the time to perform the heuristic used to obtain the initial bound is included in all of our elapsed time measurements below. However, we do not parallelize this heuristic, because the processing time for it is insignificant compared to the branch and bound processing time. One method to parallelize the heuristic would be to perform parallel local search from random starting points [18].

To show how granuality affects speedup, we choose a rather difficult problem of dimension $N = 100$ and run it with different values of *MAXV* and vary the number of processors. The problem takes about 1903 seconds on one iPSC/1 processor and 428 seconds on one iPSC/2 processor. The results of these experiments are given in figs. 6.1 and 6.2. When *MAXV* = infinity, the problem is forced into being solved in one cycle. That is, after the initial distribution of subproblems, all subproblems are solved on the first cycle regardless of how long each subproblem takes.

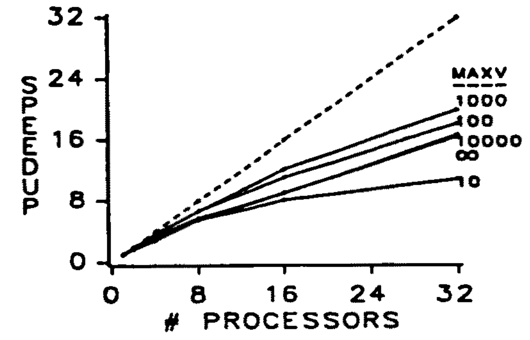

Fig. 6.1

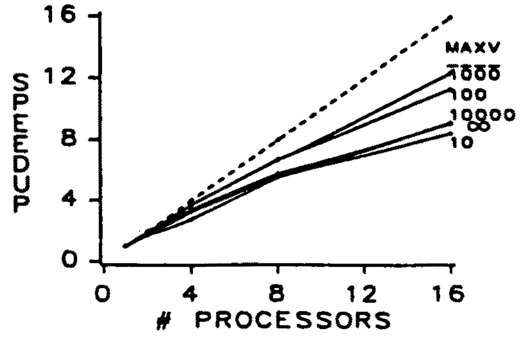

Fig. 6.2

Notice that for *MAXV* = 10, speedup is worse than *MAXV* = 100 and *MAXV* = 1000. In this case, synchronization overhead results in poor speedup. The poor curves for *MAXV* = 10000 and *MAXV* = infinity are the result of poor workload balancing. By chance, the initial splitting and distribution of subproblems may be almost perfectly matched to the number of processors.

For this particular problem, the initial heuristic does not find the global minimum. However, the heuristic finds a minimum whose function value was −54208, while the actual global minimum was −54382. Hence, our heuristic finds a solution within 0.3% of the global minimum. By using this solution, anomalies [13] in the number of vertices visited in the branch and bound tree are significantly reduced. Figure 6.3 summarizes the number of vertices found in the branch and bound tree

	Number of vertices at varying granularities				
P	10	100	1000	∞	Variation
1	183220	183220	183220	183220	0
2	183546	183548	183551	183894	348
4	183620	183622	183629	183983	368
8	183355	183361	183431	183983	628
16	183907	183045	183112	183984	939

Fig. 6.3

at the various numbers of processors and granularities for the iPSC/2. This chart shows that both "good" and "bad" anomalies can occur if the heuristic fails to find the optimal. However, since the heuristic finds a "close" solution, the anomalies are very small and thus do not affect the computational requirements significantly. It is interesting to nore that variations occur, albeit small, both in the number of processors and the granularity of global data exchange.

6.1. RESULTS FROM A PREEMPTIVE MODIFICATION TO THE ALGORITHM

We observe that early in the algorithm, after the initial splitting, a lot of synchronizations have occurred with NO splittings when low values are specified for *MAXV*. This prompts us to consider a preemptive version of the algorithm. In the preemptive version, no synchronization is done until at least one processor finishes its subproblem. In other words, splitting is done on demand. We still use the control *MAXV*, but its definition now requires that at least *MAXV* vertices be visited before a processor can be preempted, unless of course the processor solves its subproblem. This requires a small change to the depth-first branch and bound algorithm in fig. 3.2. The continuation criteria in line 3 now read:

```
while $lev \neq -1$ and (NOT-PREEMPTED or $NVERTS < MAXV$) do
```

The condition NOT-PREEMPTED is merely a test via a probe to see if no preemption message has occurred. SYNC is modified to initially send a short preemption message to every neighbor (before step 2). Furthermore, it must delete these messages after step 16 so that the next cycle may continue. Hence, the first processor solving its subproblem would cause the preemption of its neighbors according to the hypercube topology. Its neighbors, in turn, would stop their work and preempt their neighbors and so on.

The drawback to this approach is that global information about the current best minimum is not shared until the first processor finishes its subproblem. Hence, this approach puts more reliance on a good initial heuristic solution. However, the number of vertices visited for our algorithm could be no worse than the ∞ case in the nonpreemptive version.

When this approach is implemented, we observe that much of the processing occurs with no synchronization. This is expected. However, termination becomes very slow if granularity ($MAXV$) is set too low. If granularity is set too high, the preemptive algorithm becomes the nonpreemptive algorithm and thus experiences the same load balancing problems. Best performance is achieved when $MAXV$ is greater than 100 or less than 1000. Variation in this range in minimal.

Figure 6.4 compares the timings obtained on the two hypercubes. The iPSC/1 uses the Intel 80286 processor and the iPSC/2 uses the 80386 processor. This partly explains the large disparity between the sequential times of the two machines. The iPSC/2 is 4.4 times faster than the iPSC/1. Also, the iPSC/1 and iPSC/2 that we use has 32 nodes and 16 nodes, respectively.

Maximum achieved speedups

Machine	T_1	T_p	Speedup	Efficiency	P
Intel iPSC/1	1903.35	95.32	19.97	0.62	32
Intel iPSC/1	1903.35	157.66	12.07	0.75	16
Intel iPSC/2	428.26	35.10	12.20	0.76	16

Fig. 6.4

Comparing the speedups and efficiencies of these machines using the same number of processors, we see a slight efficiency improvement on the iPSC/2 at 16 processors. This is a result of improved communication hardware.

Thus far, we have shown the results from one problem run many different ways. This problem is $\min f(x) = x^T A x$, where A is a 100×100 integer matrix that has off-diagonal elements generated randomly from the interval $[-50, 50]$. The diagonal entries are selected randomly from the interval $[-2500, 2500]$. In [18], we show that problems that have a dominating diagonal are easier to solve than non-diagonally dominated problems. In fact, if all elements of the matrix are chosen in a symmetric interval about zero, problems of degree larger than 35 become very difficult to solve.

We now demonstrate that problems of increasing difficulty increase the efficiency of our algorithm. That is, as the problems become larger, it becomes more efficient to use parallel processing. In the following experiment, we generate problems with all elememts in the interval $[-50, 50]$. We ran 10 different problems at dimension $N = 10, 15, 20, 25, 30$, and 35. For practical reasons, only one problem of

iPSC/1 results					
Dimension	Vertices	T_1	T_{32}	Speedup	Efficiency
10	55	0.1	0.2	0.53	0.02
15	599	1.2	0.4	3.08	0.09
20	3679	9.2	1.0	9.00	0.28
25	50417	146.4	7.7	19.03	0.59
30	584266	1934.2	74.1	26.11	0.82
35	5462093	16193.4	556.9	29.08	0.91

Fig. 6.5

iPSC/2 results					
Dimension	Vertices	T_1	T_{16}	Speedup	Efficiency
10	55	0.02	0.03	0.76	0.05
15	599	0.26	0.09	2.78	0.19
20	3679	2.01	0.28	7.11	0.44
25	50417	33.00	2.76	11.96	0.75
30	584266	449.16	31.37	14.32	0.89
35	5264634	4207.27	278.60	15.10	0.94

Fig. 6.6

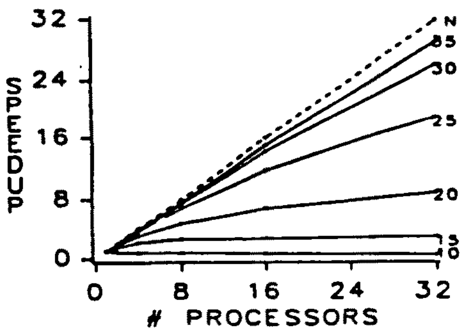

Fig. 6.7

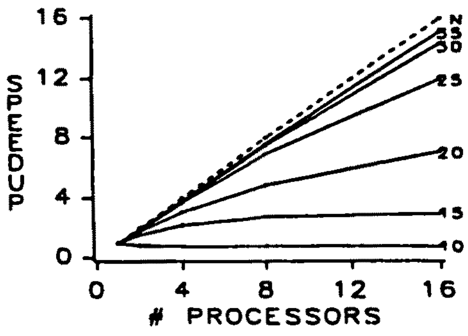

Fig. 6.8

dimension 35 was run on the iPSC/1. The average of these results from the iPSC/1 and the iPSC/2 is given in fig. 6.5 and fig. 6.6, respectively. Figures 6.7 and 6.8 plot the speedup curves for the two machines at the various problem sizes.

It is clear that as the size of the problem (in number of vertices) increases, a larger percentage of the CPU time is spent in the parallel portion of the algorithm.

This is indicative of the use of parallel systems. That is, parallelization is typically only considered for difficult large-scale problems and, as larger problems are attempted, more processors are used to solve the problem.

7. Conclusions

We have shown that a combinatorial problem can be solved efficiently using a parallel branch and bound algorithm on a distributed memory multiprocessor. We took advantage of the hypercube topology to minimize the effort to perform workload balancing. The global synchronization algorithm has no congestion and works in $\log_2 p$ steps. Efficient data structures were important in implementing fast sequential and parallel algorithms.

We introduced a minimum granularity control to reduce excessive synchronizations as the algorithm nears termination. Setting this control too high results in poor workload balancing. Setting it too low results in excessive synchronizations. A good heuristic applied before the branch and bound process is necessary to reduce parallel search anomalies. This heuristic naturally improves sequential processing time as well.

Acknowledgements

The authors are grateful to the Department of Computer Science at The University of Colorado at Boulder for providing access to their Intel iPSC/1. Instruction on the use of the Intel hypercube was given at the University of Colorado during a parallel processing workshop which was funded in part by the National Science Foundation. Research by the second author is funded by IBM through the IBM Resident Study Program.

References

[1] T.S. Abdelrahman and T.N. Mudge, Parallel branch and bound algorithms on hypercube multiprocessors, in: *3rd Conf. on Hypercube Concurrent Computers and Applications*, Vol. II (ACM Press, 1988) pp. 1492–1499.

[2] F. Barahona, A solvable case of quadratic 0-1 programming, Discr. Appl. Math. 13(1986) 23–26.

[3] R.L. Boehning, R.M. Butler and B.E. Gillett, A parallel integer linear programming algorithm, Eur. J. Oper. Res. 34(1988)393–398.

[4] S. Colley, J.P. Hayes, T.N. Mudge, J. Palmer and Q.F. Stout, Architecture of a hypercube supercomputer, in: *Int. Conf. on Parallel Processing* (1986).

[5] E.W. Felten, Best-first branch and bound on a hypercube, in: *3rd Conf. on Hypercube Concurrent Computers and Applications*, Vol. II (ACM Press, 1988) pp. 1500–1504.

[6] P.L. Hammer and S. Rudeanu, *Boolean Methods in Operations Research and Related Areas* (Springer-Verlag, 1968).

[7] P. Hansen, Methods of nonlinear 0-1 programming, Ann. Discr. Math. 5(1979)53 – 70.

[8] T. Ibaraki, *Enumerative Approaches to Combinatorial Optimization*, Ann. Oper. Res., ed. P.L. Hammer, Vols. 10 – 11 (J.C. Baltzer AG, Basel, Switzerland, 1987).

[9] *iPSC User's Guide* (Intel Corporation, CA, 1985).

[10] *iPSC Concurrent Debugger Manual*, 2nd ed. (Intel Corporation, CA, 1987).

[11] S. Jha and P.M. Pardalos, Graph separation techniques for quadratic zero-one programming, Technical Report CS-87-39, Computer Science Department, The Pennsylvania State University (1987).

[12] R.M. Karp and Y. Zhang, A randomized parallel branch and bound procedure, in: *Proc. 20th Annual ACM Symp. on Theory of Computing* (1988) pp. 290 – 300.

[13] T.-H. Lai and S. Sahni, *Anomalies in Parallel Branch and Bound Algorithms, Supercomputing*, ed. A. Lichnewsky and C. Seguez (Elsevier, 1987) pp. 111 – 129.

[14] R.P. Ma, F.-S. Tsung and M.-H. Ma, A dynamic load balancer for a parallel branch and bound algorithm, in: *3rd Conf. on Hypercube Concurrent Computers and Applications*, Vol. II (ACM Press, 1988) pp. 1505 – 1513.

[15] P.M. Pardalos and J.B. Rosen, *Constrained Global Optimization: Algorithms and Applications*, Lecture Notes in Computer Science 268 (Springer-Verlag, 1987).

[16] R.P. Pargas and D.E. Wooster, Branch and Bound algorithms on a hypercube, in: *3rd Conf. on Hypercube Concurrent Computers and Applications*, Vol. II (ACM Press, 1988) pp. 1514 – 1519.

[17] J.C. Picard and H.D. Ratliff, Minimum cuts and related problems, Networks 5(1975) 357 – 370.

[18] G. Rodgers, Algorithms for unconstrained quadratic zero-one programming on contemporary computer architectures, Ph.D. Thesis, Computer Science Department, The Pennsylvania State University (1989).

[19] C. Roucairol, A parallel branch and bound algorithm for the quadratic assignment problem, Discr. Appl. Math. 18(1987)211 – 255.

[20] K. Schwan, J. Gawkowski and B. Blake, Process and workload migration for a parallel branch and bound algorithm on a hypercube multicomputer, in: *3rd Conf. on Hypercube Concurrent Computers and Applications*, Vol. II (ACM Press, 1988) pp. 1520 – 1530.