Parallel Computing 15 (1990) 201–209
North-Holland

Parallel branch and bound on fine-grained hypercube multiprocessors *

Frank DEHNE [†], Afonso G. FERREIRA [‡] and Andrew RAU-CHAPLIN [†]

[†] Center for Parallel and Distributed Computing, School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6, [‡] Laboratoire de l'Informatique du Parallelisme - IMAG, Ecole Normale Superieure de Lyon, 69364 Lyon cedex 07, France

Received August 1989
Revised September 1989

Abstract. In this paper, we study parallel branch and bound on fine grained hypercube multiprocessors. Each processor in a fine grained system has only a very small amount of memory available. Therefore, current parallel branch and bound methods for coarse grained systems ($\leqslant$1000 nodes) cannot be applied, since all these methods assume that every processor stores the path from the node it is currently processing back to the node where the process was created (the back-up path). Furthermore, the much larger number of processors available in a fine grained system makes it imperative that global information (e.g. the current best solution) is continuously available at every processor; otherwise the amount of unnecessary search would become intolerable. We describe an efficient branch-and-bound algorithm for fine grained hypercube multiprocessors. Our method uses a global scheme where all processors collectively store all back-up paths such that each processor needs to store only a constant amount of information. At each iteration of the algorithm, all current nodes may decide whether they need to create new children, be pruned, or remain unchanged. We describe an algorithm that, based on these decisions, updates the current back-up paths and distributes global information in $O(\log m)$ steps, where m is the current number of nodes. This method also includes dynamic allocation of search processes to processors and provides optimal load balancing. Even if very drastic changes in the set of current nodes occur, our load balancing mechanism does not suffer any slow down.

Keywords. Parallel algorithms, Branch and bound, Hypercube multiprocessor, Fine grained parallel systems, Load balancing, Process allocation.

1. Introduction

Branch-and-bound (B&B) search methods such as A* search, Alpha-Beta search, depth-first and best-first search are used in areas such as VLSI design, theorem proving, linear programming, chess playing, and many other Artificial Intelligence and Operations Research applications [15,8]. Since most of the addressed problems are NP-hard, and therefore the size of the search space grows exponentially, many researchers have studied the parallelization of B&B methods (e.g. [4,7,6,14]). So far, studies have been aimed at using coarse grained multi-

* This research was conducted while the second author was visiting Carleton University. The first author's research is partially supported by the Natural Sciences and Engineering Research Council of Canada (Grant A9173). The second author is currently on leave from the University of Sao Paulo (Brazil), project BID/USP, and partially supported by CAPES/COFECUB (Grant 503/86-9).

processors for parallel B&B implementations (multiprocessors with a relatively small number – less than 1000 – of relatively powerful processors, each having a considerable amount of memory). Particular attention has recently been given to coarse-grained hypercube multiprocessors such as the FPS hypercube, NCUBE, or Intel iPSC (e.g. [5,11,1,2,3,13,10,12]).

Parallel B&B methods for coarse grained multiprocessors aim at splitting the tree searched by the B&B method into subtrees, and having subtrees searched in parallel by different processors. The essential differences among the proposed coarse grained algorithms are how they deal with the two major problems arising in such an approach: (1) *Load balancing*: When a subproblem (a subtree to be searched) is assigned to an individual processor, the size of the subtree is not known in advance. Therefore, the sizes of the problems assigned to individual processors may vary significantly and this unbalanced distribution of work load may result in performance degeneration. (2) *Global information*: For sequential B&B methods, the pruning of the search tree often depends on global information such as the best solution found so far. On a multiprocessor, the distribution of global information causes additional computational overhead or, if information is not completely distributed, the parallel method may search more nodes than the respective sequential algorithm (unnecessary search).

In this paper, we study parallel B&B on fine grained hypercube multiprocessors (hypercubes with a large number – more than 10 000 – of small processors). The Connection Machine is an example of an existing fine grained system. In the context of parallel B&B, fine grained multiprocessors obviously have the advantage of a larger number of processors and increased parallelism. However, each processor in a fine grained system has only a small constant amount of memory available. For any node p of the search tree let the *back-up path* of p be the path from p to the root of the search tree. In a fine grained multiprocessor, it is impossible for one processor to store the back-up path even for the one single node that it is currently examining. Therefore, the above mentioned coarse grained parallel B&B methods can not be applied to a fine grained system because all these methods assume that every processor stores the back-up path for its currently processed node (at least back to the tree node where the process was created).

Furthermore, the much larger number of processors (and, therefore, concurrent search processes) makes it imperative, for fine grained B&B algorithms, that global information (e.g. the current best solution) is continuously available at every processor. Otherwise, the amount of unnecessary search would become intolerable.

In the remainder of this paper we describe an efficient B&B algorithm for fine grained hypercube multiprocessors that solves the problem of storing all current back-up paths. Our method uses a global scheme where all processors collectively store all back-up paths, such that each processor needs to store only a constant amount of information. Instead of storing the individual back-up paths, we store a *current back-up tree* which is defined by the union of the back-up paths of all search tree nodes currently under examination. (This also ensures that paths in the search tree shared by several back-up paths are stored only once, thus providing optimal storage utilization.) At each iteration of the algorithm, all nodes of the current back-up tree may decide whether they need to create new children, be pruned, or remain unchanged. We describe an algorithm that, based on these decisions, updates the current back-up tree and distributes global information in $O(\log m)$ steps, where m is the size of the current back-up tree. This method also includes a dynamic allocation mechanism for allocating search processes to processors, and provides optimal load balancing. Even if very drastic changes in the current back-up tree occur, the performance of our load balancing mechanism does not deteriorate.

Note that, the $O(\log m)$ overhead mentioned above is measured by considering only constant length messages, exchanged between adjacent processors, as $O(1)$ operations. In this model, an $O(\log m)$ overhead is very small; in fact, one single "PREF" operation (parallel inter-processor read) in Connection Machine *LISP has a larger time complexity.

The remainder of this paper is organized as follows: Section 2 introduces some notation regarding B&B algorithms as well as some standard hypercube operations that will be utilized in our algorithm. In Section 3, we will then present our parallel B&B algorithm for fine-grained hypercube multiprocessors. Section 4 concludes the paper.

2. Definitions and basic hypercube operations

In this section, we will first define a generalized B&B procedure in terms of five rules. By providing different procedures to implement these rules, users will be able to implement a wide range of B&B algorithms. We then review some basic standard hypercube operations which will be used in the remainder of this paper.

2.1 Branch and bound

Branch-and-bound is a general technique for exploring search spaces; A*, Alpha-Beta, Hill-Climbing and Best-First search are well known instances of B&B algorithms.

In this paper we consider a general B&B algorithm defined by five rules; any particular B&B algorithm can be realized by providing a suitable implementation for each rule (see [1]).

- *Cost rule*: Given a leaf node in the search tree and it parent's cost, this rule defines the cost up to and including the leaf.
- *Bounding rule*: Given a node in the search tree, this rule returns 1 if the node is no longer feasible (i.e. should be deleted) else returns a 0.
- *Selection rule*: Given a node in the search tree, this rule returns an integer specifying the number of children this node should create in this iteration.
- *Expansion rule*: Given a node, an integer specifying the number of children to be created, and a pointer to where they should be created, this rule creates the new children.
- *Termination rule*: Given a set of global information, this rule returns true if a satisfactory solution has been found or all possibilities have been explored.

2.2 Basic hypercube operations

The branch-and-bound algorithm described in the next section uses slightly generalized versions of five well-defined hypercube operations. In addition to the registers listed below, implementations of these operations may require a constant number of auxiliary registers. In the following, for every register A available at every processor, $A(i)$ refers to register A at processor PE(i). We assume a hypercube consisting of $N = 2^q$ processors.

Psum(source(i),result(i)): Every processor PE(i) has some value stored in the register source(i). This operation computes result(i) := source(0) + source(1) + ... + source(i) for each PE(i). This is the standard partial sum operation in hypercubes and can be implemented in O(log N) time.

IdentifyBlock(block(i),endOfBlock(i)): A *block* of processors is defined by consecutive PE(i)'s having the same value stored in the register block(i). For each PE(i), this operation will then assign to endOfBlock(i) the largest j such that block(j) = block(i). Its time complexity is also O(log N), since it can be implemented by a "Concentrate" operation [9] followed by a RouteAndCopy operation defined below.

BlockPsum(source(i),result(i),block(i)): This operation performs a partial sum within blocks only. Every PE(i) will store in result(i) the partial sum of the value stored in the register source(i) within the block it belongs to. The BlockPsum operation can be implemented by a

Psum operation followed by a RouteAndCopy operation. Hence its time complexity is O(log N).

Route(Reg$_1$(i),Reg$_2$(i)lDest(i),Cond(i)): Every processor PE(i) has 2 data registers Reg$_1$(i), Reg$_2$(i), a destination register Dest(i), and a boolean condition register Cond(i). It is assumed that the destinations Dest(i) are monotonic; i.e. if $i<j$ then Dest(i) < Dest(j). This operation routes, for every processor PE(i) with Cond(i) = true, the contents of register Reg$_1$(i) to processor register Reg$_2$(i) of processor PE(Dest(i)); it can be implemented with an O(log N) time complexity by using a Concentrate operation followed by a Distribute operation described in [9].

RouteAndCopy(Reg$_j$(i),Reg$_2$(i),Dest(i),Cond(i)): Under the same assumptions as for the Route operation, this operation routes, for every processor PE(i) with Cond(i) = true, a copy of registers Reg$_1$(i) to registers Reg$_2$(Dest($i-1$) + 1),..., Reg$_2$(Dest(i)), each; it can be implemented with an O(log(N)) time complexity by using a Concentrate followed by a Generalize operation described in [9].

3. An algorithm for parallel branch and bound with global information on a fine-grained hypercube

A branch and bound algorithm searches in the space of all the feasible solutions for a given problem. These feasible solutions are usually seen as a search tree S over the solution space. In parallel B&B algorithms, the search in S for an optimal or satisfactory solution is performed concurrently at several nodes of S. These nodes are referred to as *active nodes*. The subtree T of S defined by the union of the back-up paths of all active nodes is the *current back-up tree* as introduced in Section 1 (an example of a current back-up tree is shown in *Fig. 1*). Note that we allow any node of the current back-up tree to be an active node, which allows e.g. the implementation of best–first search. For the remainder, m will refer to the size of the current back-up tree.

In this section, we will describe a parallel B&B algorithm for fine grained hypercube multiprocessors. Our method stores the current back-up tree such that each processor needs to store only a constant amount of information. At each iteration of the algorithm, all active nodes of the current back-up tree decide whether they need to create new children, be pruned, or remain unchanged. Based on these decisions, the algorithm described below updates the current back-up tree and distributes global information in O(log m) steps. This method also includes a dynamic allocation of search processes to processors, such that the work load balancing problem is solved optimally.

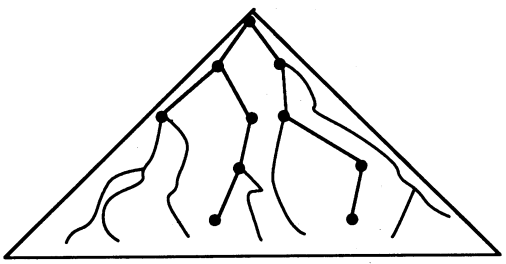

Fig. 1. A current back-up tree (bold nodes).

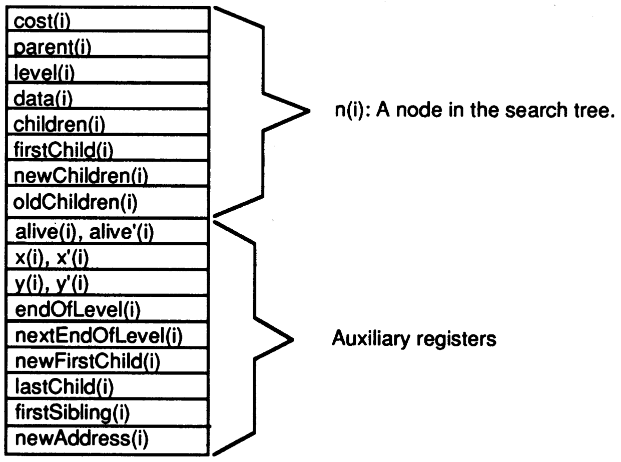

Fig. 2. The registers required at each processor PE(i).

3.1 Algorithm overview

We assume that every processor PE(i) has a constant size register $n(i)$ to store one node of the current back-up tree T. Register $n(i)$ will contain fields n.data(i), n.cost(i), n.parent(i), n.children(i), n.firstChild(i), n.level(i) and n.newChildren(i). Each of these fields stores a constant amount of data associated with the node $n(i)$, its cost, parent, number of children, position of first child, level, and number of new children, respectively. See *Fig. 2* for a list of registers required at every processor. The current back-up tree T is stored on the hypercube as follows:

Consider the level ordering of the nodes of T as shown in *Fig. 3*. Each node v of T is stored in register $n(i)$ of processor PE(i), where i is the index of v with respect to the level ordering of T.

The global structure of the B&B algorithm is described in *Fig. 4*. The back-up tree T starts as a single root node stored in register n(0) of processor PE(0). The main loop of the algorithm iterates until the termination rule returns true.

In each iteration, the algorithm updates the tree by adding new nodes in a manner prescribed by the selection and expansion rules, while at the same time pruning the tree using the cost and bound rules. A single pass through the main loop consists of five steps. First, the evaluation rule is used to calculate the cost function for all the nodes that were created in the last iteration. The evaluation rule may also maintain global knowledge concerning the progress of the search. For example, after assigning new costs to nodes, the evaluation rule may

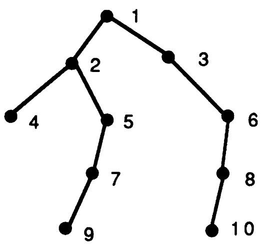

Fig. 3. Level ordering of the current back-up tree.

```
Procedure Branch-And-Bound:
  (A)  Initialize the current back-up tree to be the single root node in processor PE(0).
       Initialize n.data(0) and set all other fields of n(0) to 0.  Also create a dummy node
       at PE(1) with n.parent(1) = ∞.
  (B)  Repeat Until termination rule returns true.
       (1)  Every PE(i) storing a newly created node uses the evaluation rule to
            compute n.cost(i) and global information is distributed to all processors.
       (2)  Every PE(i) uses the bound rule to set alive(i) = 1 if the node is still feasible
            and else alive(i) = 0.
       (3)  Every PE(i) uses the selection rule to set n.newChildren(i) = number of
            children to be created.
       (4)  Procedure UpdateTree is called which creates from the old back-up tree a
            new back-up tree with all non-feasible nodes removed and new nodes
            added.
       (5)  Every PE(i) uses the expansion rule to load for each n(i) the data into its
            n.newChildren(i) new children.
  (C)  Report best node and the path from the root to the best node.
```

Fig. 4. The global structure of the branch-and-bound algorithm.

calculate a global minimum cost to be used by the bound and termination rules. The maintenance of such global knowledge does not increase the fundamental time complexity of the algorithm, since such global knowledge can be calculated by a global minimization and broadcast procedure in time O(log m), which is the same time complexity as the main body of the B&B algorithm.

The second step in the main loop of the algorithm uses the bound rule to identify nodes to be deleted in this iteration. Any particular implementation of the bound rule must never delete a parent node without also deleting its children. The parallel bounding of a large number of nodes is a particularly effective aspect of our fine-grained B&B algorithm, since with very low overhead a large number of nodes in the back-up tree may be deleted and the data structure compacted.

In the third step, the selection rule is used to assign to each node $n(i)$ an integer, n.newChildren(i), that indicates the number of children the node needs to create during this iteration. The selection rule may also use global information to instruct each node exactly which of its possible children should be created.

Step 4, i.e. procedure UpdateTree, represents the main part of our algorithm. In Steps 1 through 3 all the information needed to extend the back-up tree to its next state has been collected. Now, the back-up tree must be transformed by deleting the bounded nodes and creating space for the new additional nodes. This operation is performed by procedure UpdateTree described in detail in the following Section 3.2.

Finally, in Step 5, the data for the new leaf nodes are created using the expansion rule. At this point each node has information about how many new children should be created, and about the address of the free processors allocated for them by procedure UpdateTree. Therefore, Step 5 can be implemented with O(log m) time complexity by using a RouteAndCopy operation to copy the data of each node creating new children into the processors storing these children, and then creating for each child in parallel its actual data set locally at the respective processor.

3.2 Updating the back-up tree

The core of our parallel B&B algorithm is procedure UpdateTree which updates the back-up tree after infeasible nodes have been deleted and new nodes have been added in the previous

```
Procedure UpdateTree:
1.0) BlockPsum(alive(i),alive'(i),n.parent(i))
     Route(alive'(i), n.children(i), n.parent(i), n.parent(i) ≠ n.parent(i + 1))
     Every PE(i): n.oldChildren(i) := n.children(i)
     Every PE(i): n.children(i) := n.children(i) + n.newChildren(i)
2.0) Psum(n.children(i),x')
     Each PE(i): x'(i) := x'(i) + 1
     IdentifyBlock(n.level(i),endOfLevel(i))
     Every PE(i): nextEndOfLevel(i) := endOfLevel(i+1)
     RouteAndCopy(x'(i), x(i), nextEndOfLevel(i), i = endOfLevel(i))
3.0) BlockPsum(n.children(i), y(i), level(i))
     Each PE(i): y(i) := y(i) - n.children(i)
4.0) Each PE(i): newFirstChild(i) :=x(i) + y(i)
     Each PE(i): lastChild(i) := n.firstChild(i) + n.oldChildren(i) -1
     RouteAndCopy(newFirstChild(i), firstSibling(i), lastChild(i), n.oldChildren(i) > 0)
     Every PE(i): newAddress(i) := firstSibling(i) + alive'(i) - 1
5.0) RouteAndCopy(newAddress(i), n.parent(i), lastChild(i), n.oldChildren(i) > 0)
     Each PE(i): n.firstChild := newFirstChild(i)
6.0) Route(n(i),n(i), newAddress(i), alive(i) = 1)
```

Fig. 5. Procedure Update Tree.

step. The main problem here is that for the new back-up tree, the nodes must again be stored by level number (to allow maximum storage space utilization and performance). This makes it necessary to compute for each node of the new back-up tree its new address and relocate the nodes to obtain the correct storage scheme. This relocation of tree nodes also provides an optimal task allocation mechanism for solving the load balancing problem.

The input to this procedure is stored in the two registers alive(i) and n.newChildren(i) at each processor. Register alive(i) is set to 1, if this node is not to be deleted, or 0 otherwise; n.newChildren(i) is set to the number of new children to be created for node $n(i)$.

The procedure UpdateTree calculates for each node where its children, if any, should be located on the hypercube after the update is completed. Each parent then broadcasts this information to all of its children. Since all nodes may move, all parent nodes must also broadcast their own new address to their children so that the children can update their parent pointers. When these steps have been completed, all non-deleted nodes are routed to their new locations. This route operation will leave the necessary spaces for the new children that are to be created by the expansion rule called in the main procedure. Procedure UpdateTree is described in detail in *Fig. 5*.

In order to obtain an illustration of the idea behind procedure UpdateTree, consider the node p in *Fig. 6*. The address (processor number) of the first child of p, after the update is

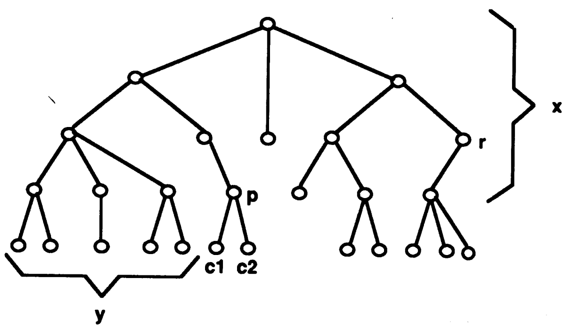

Fig. 6. Illustration of the address calculation for the new back-up tree.

completed, is the sum of two numbers x and y, defined as follows: x is the number of nodes in the updated tree up to (and including) p's level, while y is the number of children of nodes to the left of p in p's level. The sum of these two numbers indicates the position of p's first child in the new back-up tree.

Procedure UpdateTree is composed of 6 steps. In Step 1, the number of children at each node is recalculated to account for the nodes that have been marked for deletion and the new nodes that are to be added. This operation is performed by first performing a partial sum over the number of alive nodes within each block of nodes that share the same parent. The result of the partial sum is then sent to each parent by its last child. Step 1 is completed by setting register n.children(i) of each node to the sum of its undeleted children and its new children to be created.

In Step 2, the x value for each node is computed. The x value for any node at level k is one plus the sum of the number of children of all nodes up to (and including) the last node in level $k-1$. The x value for all nodes can therefore be calculated as follows. First, a partial sum on the number of children, n.children(i), of every node is computed and stored in register $x'(i)$; $x'(i)$ is then incremented by one to account for the root node. Then, the last node in each block of nodes sharing the same parent sends its x' value to its parent node.

In Step 3, the value of y for each node is calculated. This is performed by computing a partial sum on register n.children(i) for all nodes in the same level. In order not to count the children of a node in its own $y(i)$ count, n.children(i) must then be subtracted from the value provided by the partial sum operation.

In Step 4, the new address of the first child of each node is computed and stored in register newFirstChild(i), using the $x(i)$ and $y(i)$ values previously calculated. This value is then broadcast to all the children of each node; since the back-up tree is stored on the hypercube in level order, all children of a node can calculate their new addresses by adding to the new address of their first sibling the number of other siblings that are before them.

Now, every node has its new address. In Step 5, all nodes with children broadcast their new address to their children, so that the children can update their parent pointers. Step 6 completes the UpdateTree procedure by moving every node to its new address.

4. Conclusion

In this paper we have presented an efficient B&B algorithm for fine grained hypercube multiprocessors. Our method uses a global storage allocation scheme where all processors collectively store all back-up paths such that each processor needs to store only a constant amount of information. At each iteration of the algorithm, all nodes of the current back-up tree may decide whether they need to create new children, be pruned, or remain unchanged. We have described an algorithm that, based on these decisions, updates the current back-up tree and distributes global information in $O(\log m)$ steps, where m is the current number of nodes. This method also provides a dynamic allocation mechanism that obtains optimal load balancing. Another important property of our method is that even if very drastic changes in the current back-up tree occur, the performance of the load balancing mechanism remains constant.

References

[1] S. Anderson and M.C. Chen, Parallel branch and bound algorithms on the hypercube, in: M.T. Heath, ed., *Hypercube Multiprocessors* (SIAM Press, Philadelphia, PA, 1987) 309–317.

[2] T.S. Abdelrahman and T.N. Mudge, Parallel branch and bound algorithms on hypercube multiprocessors, in: *Proc. Third Conf. Hypercube Concurrent Computers and Applications*, (ACM Press, New York, 1988) 1492–1499.

[3] E.W. Felten, Best-first branch and bound on a hypercube, in: *Proc. Third Conf. on Hypercube Concurrent Computers and Applications*, (ACM Press, New York, 1988) 1500–1504.

[4] S.R. Huang and L.S. Davis, Parallel iterative A* search: an admissible distributed heuristic search algorithm, Internat. Joint Conf. on Artificial Intelligence 89, Preprint.

[5] G.J. Li and B.W. Wah, Coping with anomalies in parallel branch and bound algorithms, *IEEE Trans. on Comput.* c-35, (6) (1986) 568–573.

[6] A. Marsland and M. Campbell, Parallel search of strongly ordered game trees, *Comput. Surveys* 14 (4) (1982) 533–551.

[7] M. Newborn, Unsynchronized iteratively deepening parallel alpha–beta search, *IEEE Trans. PAMI* 10 (5) (1988) 687–694.

[8] Nils J. Nilsson, *Principles of Artificial Intelligence* (Tioga, Palo Alto, CA, 1980).

[9] D. Nassimi, and S. Sahni, Data broadcasting in SIMD computers, *IEEE Trans. Comput.* 30 (2) (1981) 101–106.

[10] R.P. Pargas and D.E. Wooster, Branch and bound algorithms on a hypercube, in: *Proc. Third Conf. Hypercube Concurrent Computers and Applications*, (ACM Press, New York, 1988) 1514–1519.

[11] M.J. Quinn, Implementing best-first branch and bound algorithms on hypercubes multicomputers, in: M.T. Heath, ed., *Hypercube Multiprocessors* (SIAM Press, Philadelphia, PA (1987) 318–326.

[12] K. Scwan, J. Gawkowski and B. Blake, Process and workload migration for a parallel branch and bound algorithm on a hypercube multicomputer, in: *Proc. Third Conf. Hypercube Concurrent Computers and Applications* (ACM Press, New York, 1988) 1520–1530.

[13] F.S. Tsung and M.H. Ma, A dynamic load balancer for a parallel branch and bound algorithm, in: *Proc. Third Conf. on Hypercube Concurrent Computers and Applications*, (ACM Press, New York, 1988) 1505–1513.

[14] H. Usui, M. Yamashita, M. Imai and T. Ibaraki, Parallel searches of game trees, *Systems Comput. in Japan* 18 (8) (1987) 97–109.

[15] Patrick H. Winston, *Artificial Intelligence* (Addison-Wesley, Reading, MA, 1984).

[16] B.W. Wah, G.J. Li and C.F. Yu, Multiprocessing of combinatorial search problems, *Computer* (June 1985) 93–108.