Multiprocessing of Combinatorial Search Problems

Benjamin W. Wah, Guo-jie Li, and Chee Fen Yu
Purdue University

Multiprocessor solutions to complex science and engineering problems require an effective representation of the problem and an efficient search. Functional requirements for search algorithms must open up a variety of architectures for any problem.

Multiprocessing refers to the concurrent execution of processes (or programs) in a computer hardware complex with more than one independent processing unit.[1] Conventionally, multiprocessing is defined as a centralized computer system with central processors, input-output processors, data channels, and special-purpose processors.

With the advent of VLSI technology, it has become cost effective to include a large number of general- and special-purpose processors in a multiprocessing system. The definition of *multiprocessing* has been extended to systems such as multiprocessors, systolic arrays, and dataflow computers. In this sense, parallel processing and multiprocessing can be considered synonymous. Examples of such systems range from the IBM 360 computers with channels, Illiac IV, Staran, Cm*, Trac, MPP, and the Cray X-MP to the latest fifth-generation computer system.

Major issues of multiprocessing

In using a multiprocessing system to solve a given problem, tremendous effort can be spent in designing a good parallel algorithm. The objective is to obtain an algorithm with a speedup proportional to the number of processors over the best available serial algorithm. This must be done within the architectural constraints, and it involves trade-offs in computation time, memory space, and communications requirements.

The applicability of multiprocessing to the problem is also an important issue. The problem should be *polynomially solvable by a serial computer*. Intractable problems with complexity that is exponential in respect to problem size cannot be solved in polynomial time* unless an exponential number of processors are used.[2] This is, of course, technologically infeasible for large problems. For example, if the best serial algorithm requires 2^N micro seconds to solve a problem of size N, it would require 2^{60} microseconds, or 366 centuries, to solve a problem of of size 60. Assuming that a linear speedup is possible, it would require 2^{50} processors to solve the problem in approximately one second, and 2^{40} processors to solve the problem in 20 minutes. For intractable problems, approximate solutions should be used in order to complete the algorithm in a reasonable time.

Multiprocessing is generally used to improve the computational efficiency of solving a given problem, *not to extend the solvable problem space of the problem*. Suppose that it takes N^k units of time to solve a problem of size N, where k is a constant (≥ 1) Assuming a linear speedup, a parallel algorithm with N processors in N^k units of time can solve problem of size $N^{(1+1/k)}$. For $k=3$, this size is $N^{1.33}$. Similarly, for a serial algorithm that takes k^N, $k>1$ units of time to solve a problem, the solvable problem size

*Saying that a problem is intractable with an exponential complexity implies that the best parallel algorithm cannot have a polynomial complexity with a polynomial number of processors. Supposing this is false, a serial simulation of this parallel algorithm can solve the problem in polynomial time, hence contradicting the fact that the problem has an exponential complexity.

0018-9162/85/0600-0093$01.00 © 1985 IEEE

with N processors in k^N units of time is $N + \log_k N$. For a large N, this size is approximately N.

Search problems. This article examines the use of multiprocessing in solving combinatorial search problems. Combinatorial search problems involve the search for one or more optimal or suboptimal solutions in a defined problem space. They can be classified as *decision* problems, which find solutions satisfying a given set of constraints, or as *optimization* problems, which seek solutions satisfying the constraints and also optimizing an objective function. Examples include proving theorems, playing games, evaluating a logic program, solving a database query, designing a computer system, assigning registers for a compiler, finding the shortest path in a graph, solving a mathematical programming problem, and searching for a permutation order to sort a set of numbers. These problems occur in a wide spectrum of engineering and science applications, including artificial intelligence and operations research.

A search problem can be represented as an acyclic graph* or as a search tree. These representations are characterized by a root node with no edge entering it and by one or more terminal nodes with no exiting edges. In a search graph, one or more edges can enter any node except the root; in a search tree, each node except the root has exactly one edge entering it.

An edge in a search graph represents an assignment of value to an unassigned parameter. This can be illustrated by the 0/1 knapsack problem, in which N objects are to be packed into a knapsack. Object i has a weight w_i, and the knapsack has a capacity M. If object i is placed in the knapsack, then a profit p_i is earned. The objective is to fill the knapsack so as to maximize profit. The unassigned parameters are the set of objects that have not been considered. In expanding a node, an object, say i, is selected, and two alternatives are created: (a) object i is included in the knapsack, and (b) object i is not included.

The nonterminal nodes in a search tree can be classified as AND nodes and OR nodes. An AND node represents a (sub)problem that is solved only when all its children have been solved. An example of an AND node is one that adds the solutions from all subtrees expanded from this node. In contrast, an OR node represents a (sub)problem that is solved if any one of its children are solved. (The definitions of AND and OR nodes are taken from Martelli and Montanari[3]; the roles of the AND and OR nodes in an AND/OR tree are reversed in Nilsson's definitions.[4]) Expanding a 0/1 knapsack problem by choosing an object to be included or excluded corresponds to transforming the problem from one state to another until the goal state is achieved. In this sense, the resulting tree contains only OR nodes.

Search problems can be represented as acyclic graphs 'or search trees.

To facilitate the design of multiprocessing systems for solving a search problem, the problem is transformed into one of the following paradigms according to the functions of the nodes.[4,5]

(1) *AND tree:* All nonterminal nodes in the search tree are AND nodes. An example is a divide-and-conquer algorithm that decomposes a problem into subproblems and solves the original problem by combining the solutions of the subproblems.

(2) *OR tree:* all nonterminal nodes in the search tree are OR nodes. Branch-and-bound algorithms that systematically prune unnecessary expansions belong to this class.

(3) *AND/OR graph:* The nonterminal nodes are either AND or OR nodes. Game trees and logic programs can be represented as AND/OR trees. Dynamic programming problems can be solved as acyclic AND/OR-graph searches.

A node is *active* if its solution value has not been found; otherwise, it is *terminated.* In a serial algorithm, the set of active nodes is maintained in a single list. A heuristic value defined by a heuristic function is computed for each node. The active node with the minimum heuristic value is always expanded first. A search is called a *depth-first search* if the negation of the level number is used as the heuristic function. In this case, the nodes in the active list are expanded in a last-in/first-out order. A search is called a *breadth-first search* if the level number is used as the heuristic function. In this case, the nodes in the active list are expanded in a first-in/first-out order. Lastly, a lower bound can be computed for each node in the active list. This represents the lower bound of the best solution that can be obtained from this node. By using the lower-bound function as the heuristic function, a *best-first search* expands the node with the minimum lower bound.

Dominance relations. To reduce the search space, unnecessary expansions can be pruned by dominance relations. When a node P_i dominates another node P_j, it implies that the subtree rooted at P_i contains a solution node with a value no more (or less) than the minimum (or maximum) solution value of the subtree rooted at P_j.

As an example, consider two assignments, P_1 and P_2, on the same subset of the objects to be packed into a knapsack in the 0/1 knapsack problem. If the total profit of the objects assigned to the knapsack for P_1 exceeds that of P_2 and the total weight of the objects assigned in P_1 is less than that of P_2, then the best solution expanded from P_1 dominates P_2.

A special case of the special class dominance tests is the class of *lower-bound tests,* which are used in branch-and-bound algorithms to solve minimization problems. If a solution with value v has already been found, then all active nodes with lower bounds greater than v can be terminated, since they would not lead to better solutions. The minimum of the solution values obtained at any time can be conve-

*An acyclic graph is one without cycles. In general, some graphs with cycles can be searched, but this topic is beyond the present scope.

·lated to the complexity of $S(N)$ and (N). In finding the sum or the max-num of N numbers, $S(N) + C(N)$ $= O(1)$; in using $N/(\log_2 N)$, processors chieve the maximum utilization.[14] In orting N numbers, $S(N) + C(N)$ $= O(N)$, and $\log_2 N$ processors should e used to maximize processor utiliza-ion. We have studied the *asymptotic processor utilization* and found that $N/\log_2 N$ is a threshold when $S(N) + C(N) = O(N)$.[15] For k (a unction of N) processors, the pro-essor utilization is one, between zero and one, or zero when the limiting atio of k and $N/\log_2 N$ is zero, greater han zero, or approaching infinity, espectively.

Since processor utilization increases with a decreasing number of pro-essors, it is not an adequate measure of the effects of parallel processing. A more appropriate measure is the KT^2 criterion, which considers both pro-essor utilization and computational ime. We have proven that the asymp-otic optimal number of processors to minimize KT^2 in parallel divide-and-conquer algorithms is $\Theta(N/\log_2 N)$, when $S(N) + C(N) = O(1)$.[15] Simu-lations have verified that the optimal number of processors is either exactly $N/(\log_2 N - 1)$ or very close to this value.

Branch-and-bound algorithms

A branch-and-bound algorithm is a systematic search of an OR tree.[16] It is characterized by four constituents: a branching rule, a selection rule, an elimination rule, and a termination condition.

The selection rule examines the list of active subproblems (nodes) and selects one for expansion based on the heuristic value. For a serial search, the minimum number of nodes is ex-panded under a best-first strategy, provided all lower bounds are distinct.[16] This is achieved at the ex-pense of increased memory space, as there are a large number of concur-rently active subproblems. The algo-rithm is terminated when all active

The elimination rule prunes un-necessary expansions by means of lower-bound and dominance tests. For lower-bound tests, the incumbent z holds the value of the best solution found so far in the search. In minimi-zation problems, a lower bound is calculated for each subproblem when it is created. A subproblem cannot lead to the optimal solution if its lower bound exceeds the incumbent; such subproblems can be eliminated from further consideration. This lower-bound test can be relaxed by defining an *allowance function*, $\epsilon(z)$. Sub-problems with lower bounds greater than $z - \epsilon(z)$ are eliminated, resulting in a suboptimal solution that deviates from the optimal solution by at most $\epsilon(z_0)$, where z_0 is the value of the op-timal solution.[17] An example of an allowance function is the relative error deviation; a subproblem is terminated if its lower bound is greater than $z/(1+\epsilon)$. An allowance function is very effective in reducing the computa-tional complexity of branch-and-bound algorithms. We have found that for some NP-hard problems under best-first searches, a linear reduction in accuracy of the solution results in an exponential reduction in the computational overhead.[18,19]

Each of the four constituents of a serial branch-and-bound algorithm can be implemented by means of parallel processing.

(1) Parallel selection of subprob-lems: Multiple subproblems with the smallest heuristic values can be se-lected for expansion.

(2) Parallel expansion of sub-problems.

(3) Parallel termination tests and update of the incumbent.

(4) Parallel elimination tests: These include the lower-bound and domi-nance tests.

We have studied the performance bounds of parallel branch-and-bound search, assuming (1) that only lower-bound tests are active, (2) that there is a single shared memory, (3) that no ap-

nously, and (5) that the heuristic func-tion is unambiguous.[20]

Let $T_b(k)$ (resp. $T_d(k)$) be the number of iterations required to ob-tain the optimal solution under a best-first (resp. depth-first) search with k processors. The following bounds have been derived:

(1) For a parallel best-first search, if the value of optimal-solution nodes differs from the lower bounds of other nodes, then

$$\left[\frac{T_b(1)-1}{k} + 1 \right] \le T_b(k) \le \left[\frac{T_b(1)}{k} + \frac{k-1}{k} h \right] \qquad (2)$$

where h is the maximum number of levels in the branch-and-bound tree.

(2) For a parallel depth-first search, if all solution nodes exist at level h, then

$$\left[\frac{T_b(1)-1}{k} + 1 \right] \le T_d(k) \le \left[\frac{T_d(1)}{k} + \frac{(k+1)\cdot(c+1)}{k} h \right] \qquad (3)$$

where c is the number of distinct in-cumbents obtained during the search. A similar equation can be also derived for parallel breadth-first searches. Equations (2) and (3) show almost a k-time reduction in the number of iterations when parallel processing is applied on the same search strategy and when $T_b(1)/k$ is large.

The best search strategy depends on the accuracy of the problem-depend-ent lower-bound function. Very inac-curate lower bounds are not useful in guiding the search; very accurate lower bounds prune most unnecessary ex-pansions. In both cases, the number of subproblems expanded by depth-first and best-first searches does not differ greatly. A depth-first search is better because it requires less memory space.

related to the complexity of $S(N)$ and $C(N)$. In finding the sum or the maximum of N numbers, $S(N) + C(N) = O(1)$; in using $N/(\log_2 N)$, processors achieve the maximum utilization.[14] In sorting N numbers, $S(N)+C(N) = O(N)$, and $\log_2 N$ processors should be used to maximize processor utilization. We have studied the *asymptotic processor utilization* and found that $N/\log_2 N$ is a threshold when $S(N)+C(N)=O(N)$.[15] For k (a function of N) processors, the processor utilization is one, between zero and one, or zero when the limiting ratio of k and $N/\log_2 N$ is zero, greater than zero, or approaching infinity, respectively.

Since processor utilization increases with a decreasing number of processors, it is not an adequate measure of the effects of parallel processing. A more appropriate measure is the KT^2 criterion, which considers both processor utilization and computational time. We have proven that the asymptotic optimal number of processors to minimize KT^2 in parallel divide-and-conquer algorithms is $\Theta(N/\log_2 N)$, when $S(N)+C(N)=O(1)$.[15] Simulations have verified that the optimal number of processors is either exactly $N/(\log_2 N-1)$ or very close to this value.

Branch-and-bound algorithms

A branch-and-bound algorithm is a systematic search of an OR tree.[16] It is characterized by four constituents: a branching rule, a selection rule, an elimination rule, and a termination condition.

The selection rule examines the list of active subproblems (nodes) and selects one for expansion based on the heuristic value. For a serial search, the minimum number of nodes is expanded under a best-first strategy, provided all lower bounds are distinct.[16] This is achieved at the expense of increased memory space, as there are a large number of concurrently active subproblems. The algorithm is terminated when all active subproblems have been either expanded or eliminated.

The elimination rule prunes unnecessary expansions by means of lower-bound and dominance tests. For lower-bound tests, the incumbent z holds the value of the best solution found so far in the search. In minimization problems, a lower bound is calculated for each subproblem when it is created. A subproblem cannot lead to the optimal solution if its lower bound exceeds the incumbent; such subproblems can be eliminated from further consideration. This lower-bound test can be relaxed by defining an *allowance function*, $\epsilon(z)$. Subproblems with lower bounds greater than $z-\epsilon(z)$ are eliminated, resulting in a suboptimal solution that deviates from the optimal solution by at most $\epsilon(z_0)$, where z_0 is the value of the optimal solution.[17] An example of an allowance function is the relative error deviation; a subproblem is terminated if its lower bound is greater than $z/(1+\epsilon)$. An allowance function is very effective in reducing the computational complexity of branch-and-bound algorithms. We have found that for some NP-hard problems under best-first searches, a linear reduction in accuracy of the solution results in an exponential reduction in the computational overhead.[18,19]

Each of the four constituents of a serial branch-and-bound algorithm can be implemented by means of parallel processing.

(1) Parallel selection of subproblems: Multiple subproblems with the smallest heuristic values can be selected for expansion.

(2) Parallel expansion of subproblems.

(3) Parallel termination tests and update of the incumbent.

(4) Parallel elimination tests: These include the lower-bound and dominance tests.

We have studied the performance bounds of parallel branch-and-bound search, assuming (1) that only lower-bound tests are active, (2) that there is a single shared memory, (3) that no approximations are allowed, (4) that the subproblems are expanded synchronously, and (5) that the heuristic function is unambiguous.[20]

Let $T_b(k)$ (resp. $T_d(k)$) be the number of iterations required to obtain the optimal solution under a best-first (resp. depth-first) search with k processors. The following bounds have been derived:

(1) For a parallel best-first search, if the value of optimal-solution nodes differs from the lower bounds of other nodes, then

$$\left\lceil \frac{T_b(1)-1}{k}+1 \right\rceil \le T_b(k) \le \left\lceil \frac{T_b(1)}{k}+\frac{k-1}{k}h \right\rceil \quad (2)$$

where h is the maximum number of levels in the branch-and-bound tree.

(2) For a parallel depth-first search, if all solution nodes exist at level h, then

$$\left\lceil \frac{T_b(1)-1}{k}+1 \right\rceil \le T_d(k) \le \left\lceil \frac{T_d(1)}{k}+\frac{(k+1)\cdot(c+1)}{k}h \right\rceil \quad (3)$$

where c is the number of distinct incumbents obtained during the search. A similar equation can be also derived for parallel breadth-first searches. Equations (2) and (3) show almost a k-time reduction in the number of iterations when parallel processing is applied on the same search strategy and when $T_b(1)/k$ is large.

The best search strategy depends on the accuracy of the problem-dependent lower-bound function. Very inaccurate lower bounds are not useful in guiding the search; very accurate lower bounds prune most unnecessary expansions. In both cases, the number of subproblems expanded by depth-first and best-first searches does not differ greatly. A depth-first search is better because it requires less memory space, in proportion to the height of the search tree. When the accuracy of the

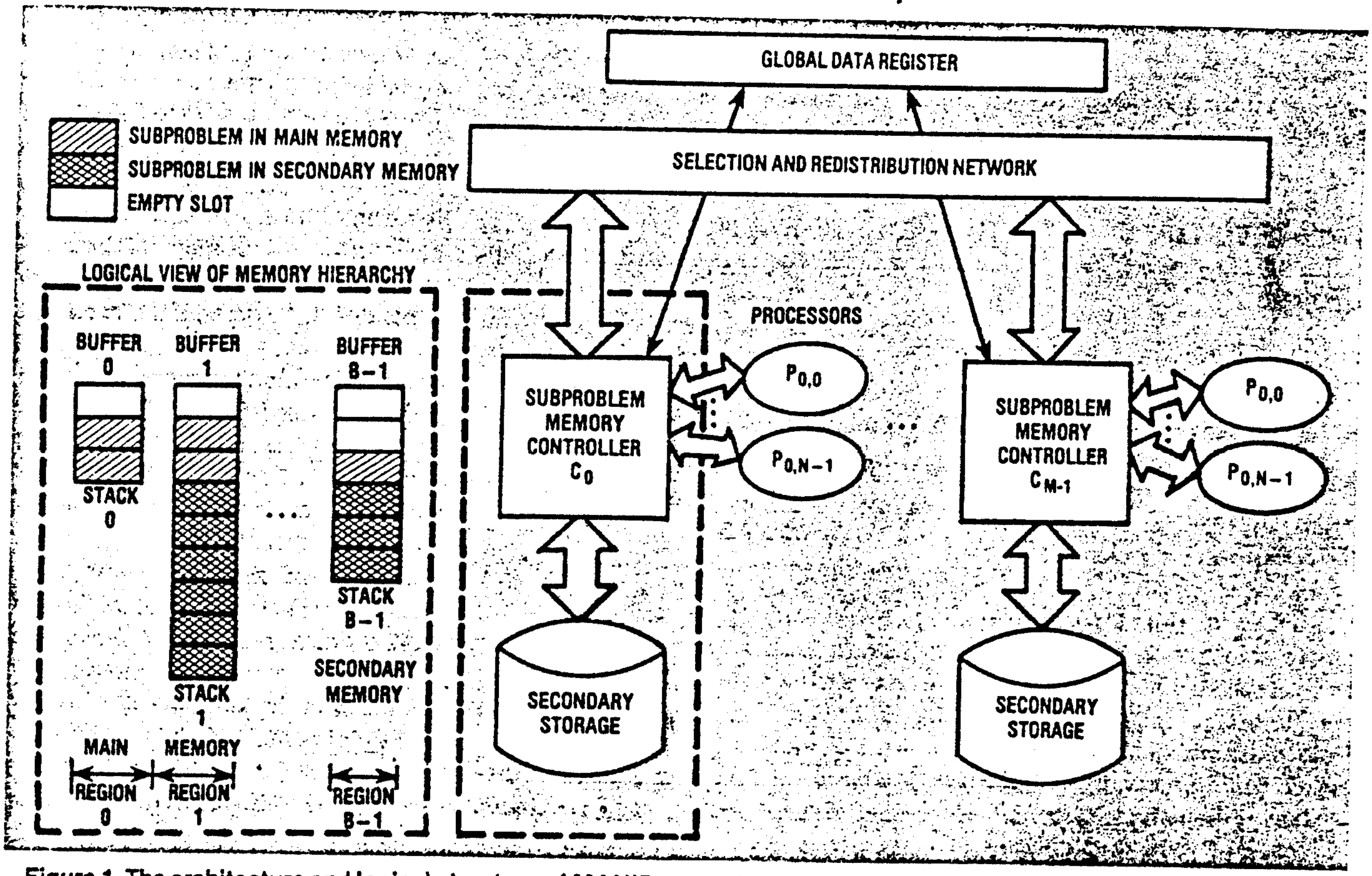

GLOBAL DATA REGISTER
SELECTION AND REDISTRIBUTION NETWORK
SUBPROBLEM IN MAIN MEMORY
SUBPROBLEM IN SECONDARY MEMORY
EMPTY SLOT
LOGICAL VIEW OF MEMORY HIERARCHY
BUFFER 0
BUFFER 1
BUFFER B−1
STACK 0
STACK 1
STACK B−1
SECONDARY MEMORY
MAIN MEMORY
REGION 0
REGION 1
REGION B−1
PROCESSORS
SUBPROBLEM MEMORY CONTROLLER C0
SUBPROBLEM MEMORY CONTROLLER CM-1
P0,0
P0,N−1
SECONDARY STORAGE

Figure 1. The architecture and logical structure of MANIP—a multiprocessor for parallel best-first search.

lower-bound function is moderate, a best-first search performs better. In this case, a good memory management system is necessary to support the memory space required.

Several architectures based on implicit enumeration have been proposed for parallel processing of branch-and-bound algorithms. These architectures delegate a subproblem to each processor, which reports to its parent processor when the evaluation is complete.[21] The limited degree of communication causes some processors to work on tasks that a better interconnection network would eliminate. Moreover, implicit enumeration is wasteful. Imai et al.[22] and El-Dessouki and Huen[23] have investigated parallel branch-and-bound algorithms based on a general-purpose network architecture with limited memory space and slow interprocessor communication. They used depth-first search, due to memory limitations.

Problems more efficiently evaluated by a parallel best-first search require more complex architectures. The design problems are a selection of subproblems with the minimum lower bounds and management of the required large memory space.

Manip – Multiprocessor for parallel best-first search with lower bound tests only. Figure 1 shows the architecture of Manip.[18,24] It consists of five major components: a selection and redistribution network, secondary storage, processors, global data register, and subproblem memory controllers.

The selection network selects subproblems with the minimum lower bounds for expansion in each iteration and connects the memory controllers for load balancing. Secondary storage holds excess subproblems that cannot be stored in the memory controllers. The memory controllers manage the local list of subproblems, maintain the secondary storage, and communicate with other controllers through the selection and redistribution network. The processors are general-purpose computers for partitioning subproblems and evaluating lower bounds. The global data register is accessible to all memory controllers and contains the value of the incumbent. To avoid contention during updates, this register can be implemented by a broadcast bus or a sequential associative memory. In the latter case, the minimum is found when the values of the feasible solutions are shifted out bit-serially and synchronously from all processors.

Two difficult issues must be solved in a parallel best-first search. First, the k subproblems with the smallest lower bounds must be selected from the N active subproblems in the system. Selection by software requires a time overhead of $O(N)$ in each iteration. A practical multistage selection network for selecting k elements from N elements requires $O(\log_2 N \log_2 k)$ time complexity and $O(N \cdot \log_2^2 k)$ hardware complexity.[25]

A single-stage selection network can also be used. One or more subproblems with the minimum lower bounds in each processor are sent to the neighboring processors and inserted into their local lists. A maximum of

$(k-1)$ shift-and-insert operations are needed to ensure that each processor has one of the k subproblems with the smallest lower bounds.[18] Assuming that insertion is implemented in software, the time overhead in each iteration is $O(k \cdot \log_2 N)$. In all these cases, selection represents significant system overhead.

No-wait policy. Selection overhead is high; furthermore, the selection rule is based on a fallible lower-bound heuristic. Therefore, it might be more efficient not to follow the selection rule strictly; We propose a no-wait policy. Instead of waiting for one of the k subproblems with the smallest lower bounds, each processor would expand the "most promising" subproblem in its local memory and initiate a fetch of the "most promising" subproblem from its neighbors. In this case, the most promising subproblem is the one with the minimum lower bound.

When the k most promising subproblems are randomly distributed among the processors, the average fraction of processors containing one or more of the most promising subproblems is at least 0.63,[18] resulting in a speedup proportional to $0.63k$. However, as expansion proceeds, the distribution might become nonrandom and require an interconnection network to randomize the distributions and balance the workload in the system. Experimental results on vertex-cover and knapsack problems have shown that the number of subproblems expanded increases by only about 10 percent when the above scheme replaces a complete selection. The performance is almost as good as that of a complete selection when the processors expand subproblems synchronously and perform one shift-and-insert operation for each subproblem expanded. The shift-and-insert operation can be overlapped with subproblem expansions and supported by a unidirectional ring network.

A second issue in implementing a best-first search lies in the management of the large memory space required. The multiprocessing model used to study this problem comprises a CPU, a main memory, a slower secondary memory, and a secondary-memory controller. The expected completion time of the branch-and-bound algorithm on this model is taken as the performance measure.

A direct implementation involving an ordered list of pointers to the subproblems results in poor locality of access, because the subproblems are not ordered by lower bounds in the virtual space. A better alternative is a special virtual memory that tailors its control strategies according to the locality of access.[26] However, this approach is inflexible, because the parameters of the control strategies are problem dependent. The inadequacies of these approaches are due, again, to strict adherence to the selection rule. We can also apply the no-wait policy here; it has resulted in the design of a modified branch-and-bound algorithm.[27]

Experimental results on integer and vertex-cover programming verify the algorithm's usefulness.

A modified algorithm. In this modified algorithm, the range of possible lower bounds is partitioned into b disjoint regions (Figure 1). The subproblems in each region are maintained in a separate list. The top portion of each list resides in the main memory, and the rest resides in the secondary memory. Due to the high overhead of secondary-storage accesses, subproblems in a list are expanded in a depth-first manner. To implement the no-wait policy, the *modified selection rule* chooses for expansion the subproblem in the main memory with the smallest lower bound. Since subproblems within a list are not sorted, the *lower-bound elimination rule* has to be modified.

Assuming that the new incumbent lies in the range of list ℓ, all lists with indices greater than ℓ are eliminated. Subproblems in list ℓ with lower bounds greater than the incumbent are eliminated only when they are moved to the main memory during the expansion of list ℓ. As a result, it is necessary to carry out the lower-bound test on each selected subproblem before it is expanded.

When one list is used, the modified algorithm is identical to a depth-first search; when infinity lists are used, it is identical to a best-first search. In general, as the number of lists increases, the number of subproblems expanded decreases and the overhead of the secondary-memory accesses increases. The number of lists should be chosen to maximize the overlap between computations and secondary-memory accesses. This overlap, in turn, depends upon the accuracy of the lower-bound function and the access times of the main and secondary memories. The accuracy of the lower-bound function is problem dependent and can be estimated from sample problems of the same type.

Experimental results on integer-programming and vertex-cover problems verify the usefulness of the modified algorithm. For vertex-cover problems, the lower-bound function is very accurate, so a depth-first search results in the best performance. For integer-programming problems, the lower-bound function is less accurate. As a result, more stacks (two to three) achieve best performance. The improvement in paging overhead over a direct implementation of the best-first search can exceed a factor of 100.

Experience with Manip and prior studies show three functional requirements for efficient evaluation of branch-and-bound algorithms with only lower-bound tests: a loosely coupled interconnection of processors with load-balancing capability, a method of concurrent update, and broadcast of the incumbent.

Parallel dominance tests. When general dominance tests are used, it is necessary to keep the set of *current dominating nodes* (denoted by N_d) in memory. These are nodes that have been generated but not yet dominated. In general, N_d can be larger than the set of active nodes. A newly generated node, P_i, has to be compared with all nodes in N_d to see whether P_i or any nodes in N_d are dominated.

If N_d is small, it can be stored in a bank of global data registers. However, centralized comparisons are inefficient when N_d is large. A large N_d should then be partitioned into k subsets, $N_d^0, \ldots, N_d^{k-1}$, and distributed among the local memories of the k processors. A subproblem, $P_{i,j}$, generated in processor i, is first compared with N_d^i; any subproblems in N_d^i dominated by $P_{i,j}$ are removed. If $P_{i,j}$ is not dominated by a subproblem in N_d^i, it is sent to a neighboring processor and the process repeats. If it has not been dominated by any node in N_d, $P_{i,j}$ eventually returns to processor i and is inserted into N_d^i.

The functional requirements for implementing parallel dominance tests depend on the size of N_d and the structure of the dominance relation. When $|N_d|$ is small, broadcast buses or global registers carry important unstructured dominance tests, in which a dominance relation can exist between any pair of nodes. For structured dominance tests, it might be possible to partition the search tree and localize the dominance tests, but this poses additional complexity on the system architecture. On the other hand, when $|N_d|$ is large, it is necessary to partition N_d into subsets and to perform the dominance tests in parallel. This results in tight coupling of the processors, because the transfer of newly generated nodes between processors must be synchronized and overlapped with computations.

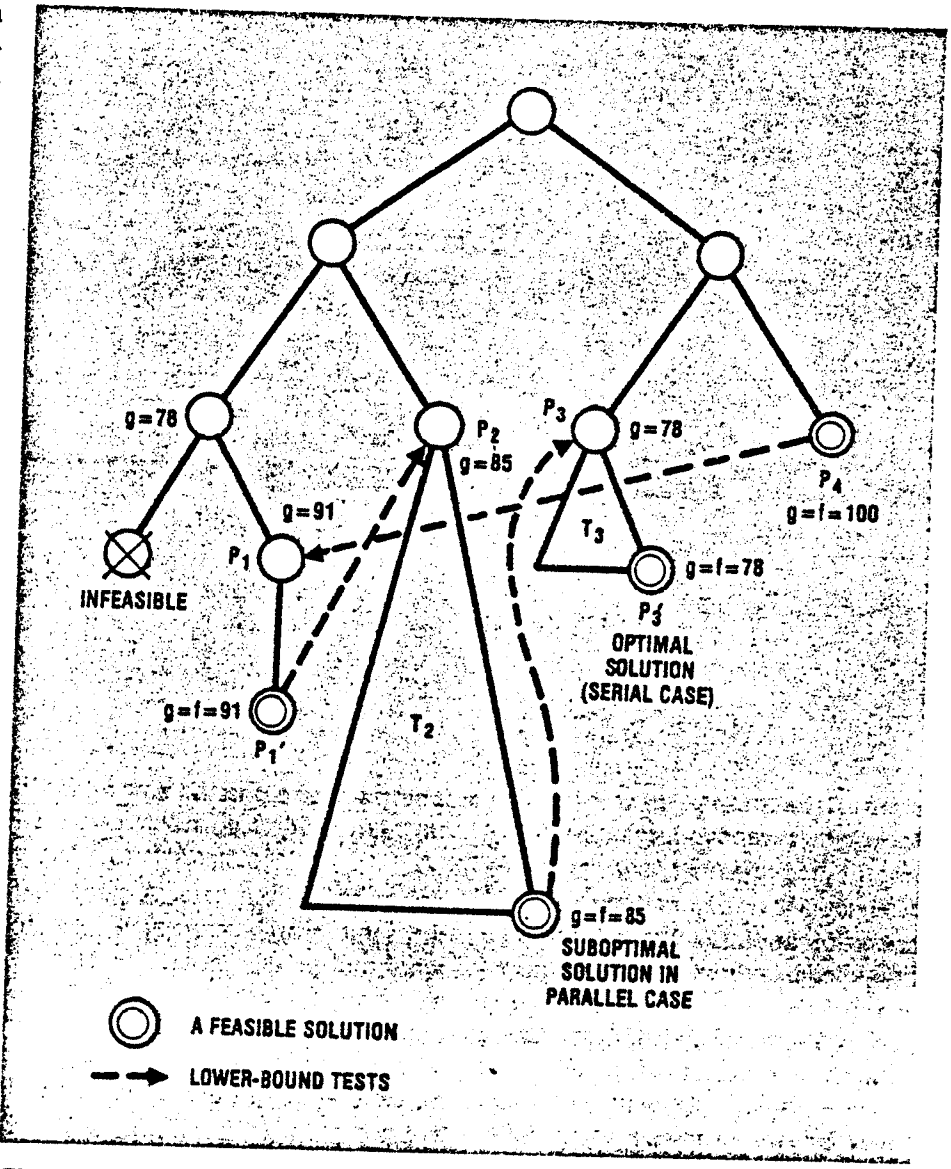


Figure 2. Example of a detrimental anomaly under a parallel depth-first search (allowance function $\epsilon = 0.1$).

Anomalies of parallelism in branch-and-bound algorithms. Since it is possible to overlap the communication overheads with computations for the various search strategies, the speedup of branch-and-bound algorithms can be measured by the ratio of the number of iterations of the best serial algorithm to that of the parallel algorithm under synchronous operations.

A k-fold speedup is expected when k processors are used. However, simulations have shown that the number of iterations for a parallel branch-and-bound algorithm using k processors can be more than the number of iterations of the best serial algorithm (this phenomenon is a *detrimental anomaly*); less than one-kth of the number of iterations of the best serial algorithm (an acceleration anomaly); or less than the number of iterations of the best serial algorithm, but more than one-kth of the number of iterations of the best serial algorithm (a deceleration anomaly).[20,28,29]

It is desirable to discover conditions that preserve acceleration anomalies, eliminate detrimental anomalies, and minimize deceleration anomalies.

Figure 2 gives an example of a detrimental anomaly. Let $g(P_i)$ be the lower bound of subproblem P_i and $f(P_i)$ be the value of the best solution that can be obtained from P_i. Suppose that the best serial algorithm for the problem is a depth-first search. In a serial depth-first search, subtree T_2 is terminated by the lower-bound test of P_1' as $f(P_1')/(1+\epsilon) \leq g(P_2)$, where $\epsilon = 0.1$. In a parallel depth-first search with two processors, a feasible solution, P_4, which terminates P_1 and P_1', is found in the second iteration. Consequently, P_2 is not eliminated, since P_1 is not generated and $f(P_4)/(1+\epsilon) > g(P_2)$. Subtree T_2 has to be expanded; this eventually terminates subtree T_3. If T_2 is much larger than T_3, the time it takes to expand T_2 by using two processors exceeds the time to expand T_3 by using one processor.

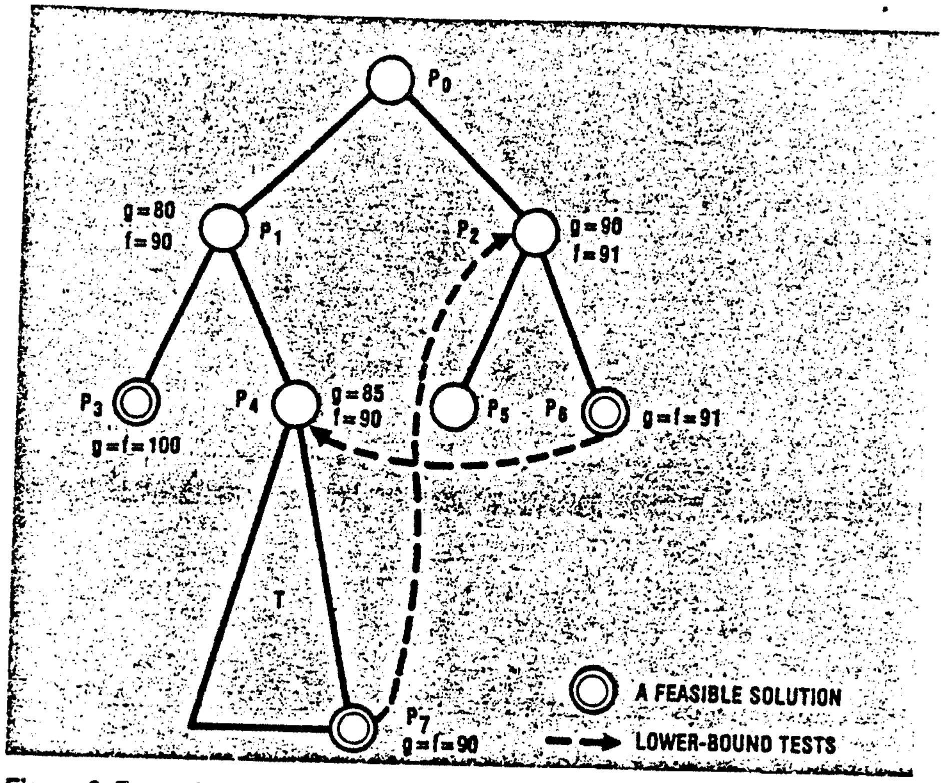


Figure 3. Example of an acceleration anomaly under a parallel depth-first search (allowance function $\epsilon = 0.1$).

Figure 3 shows an example of an acceleration anomaly. Subtree T will be expanded in a serial depth-first search, as $f(P_3)/(1+\epsilon) > g(P_4)$ when $\epsilon = 0.1$, but not in a parallel depth-first search with two processors, since P_2, and hence T, will be terminated by the lower-bound test with P_6: $(f(P_6)/(1+\epsilon) < g(P_4))$. If T is very large, an acceleration anomaly will occur.

A heuristic function is *unambiguous* if all nodes in the search tree have distinct heuristic values. An elimination rule (lower-bound or dominance tests) is said to be *consistent* with the heuristic function if the elimination of P_j by P_i implies that P_i is selected before P_j in a serial search. Anomalies are caused by a combination of reasons: (1) there are multiple solution nodes; (2) the heuristic function is ambiguous; and (3) the elimination rule is not consistent with the heuristic function.[2] These conditions cause the tree to be searched in a different order in the serial and parallel cases.

We have discussed the conditions sufficient for eliminating detrimental anomalies and the conditions necessary to preserve acceleration anomalies in a previous article[20]; a brief summary is given here. Assume that the same search strategy is used in serial and parallel cases. For branch-and-bound algorithms with dominance tests, only a best-first search with the following conditions guarantees that detrimental anomalies will not occur:

(1) The heuristic function is unambiguous.

(2) Approximations are not allowed.

(3) The dominance relation is consistent with the heuristic function.

Ambiguity in the heuristic function can be resolved by augmenting the original heuristic function with a tie-breaking rule, say, by level and left-right orientation. For most problems, dominance relations that are consistent with the heuristic function can be designed. Acceleration anomalies can occur in one of following cases: when a breadth-first or depth-first search is used; when some nodes have identical lower bounds; when the dominance relation is inconsistent with the heuristic function; when multiple lists of subproblems are used; or when a suboptimal solution is sought.

AND/OR-tree search

Searching an AND/OR tree is more complex than searching an AND tree or an OR tree. An AND/OR tree is searched in two phases. The first is a top-down expansion, as in searching an OR tree; the second is a bottom-up evaluation, as in searching an AND tree. Due to the existence of both AND and OR nodes, a parallel search algorithm should combine the features of AND- and OR-tree searches. The presence of OR nodes demands that a good selection strategy be developed. The granularity of parallelism, like that of parallel divide-and-conquer algorithms, is an important consideration. Specific restrictions on a given problem, such as pruning rules, must be considered. These rules are usually more complicated, as more information is involved in the process.

When two AND/OR subtrees are searched concurrently, more work than necessary might be performed if pruning information obtained from one processor is unavailable to the other processor. The extra work is called *information deficiency overhead.* Pruning information can be exchanged by messages or through a common memory. Increased communication overhead needed for pruning is called *information transfer overhead.* In general, a tradeoff exists between the information-deficiency and information-transfer overheads. A good parallel AND/OR-tree search should weigh the tradeoffs—the merits of parallel processing against the communications overhead of obtaining the necessary pruning information.

Parallel α-β search. A two-person game between players MAX and MIN can be represented in a game tree in which the moves of MAX and MIN are put in alternate levels of the tree. In the corresponding AND/OR tree, OR modes represent board positions resulting from MAX's moves and AND nodes represent positions resulting from MIN's moves. All nonterminal MAX nodes take the maximum score of their children, while nonterminal MIN nodes take the minimum score. This *minimax* procedure is used to find the best move for the MIN player represented as the root.[5]

A well-known technique to improve the efficiency of a minimax search is *α-β pruning*.[4] This technique uses two parameters, α and β, to define the search window. The α carries the lower bound of the MAX nodes; β represents the upper bound of the MIN nodes. The game tree has solution values defined for the terminal nodes only and is searched in a depth-first fashion.

In expanding a MIN node, if the value returned by any of its children is less than α, then this node can be pruned without further expansion. In this case, the value returned by this node to its parent—a MAX node—is less than α and another MAX node with value equal to α (according to the definition of α) already exists. The β is updated when a MIN node with a smaller value is found.

On the other hand, in expanding a MAX node, if the value returned by any of its children is greater than β, then this node can also be pruned. The α is updated when a MAX node with a larger value is found. The search is terminated when all nodes have been either pruned or expanded. The α-β search performs better when the initial search window is small.

The cost of searching a game tree depends on the distribution of values of the terminal nodes. The tree is said to have a *best-case ordering* if the first (or leftmost) branch from each node leads to the best value; it has a *worst-case ordering* if the rightmost branch from each node leads to the best value.

A number of parallel game-tree search techniques have been developed.[30] In the *parallel aspiration search*, the α-β window is divided into nonoverlapped subintervals, which are independently searched by multiple processors; Baudet reported that the maximum expected speedup is around five or six, regardless of the number of processors.[31] The speedup is limited because at least $W^{\lceil h/2 \rceil} + W^{\lfloor h/2 \rfloor} - 1$ nodes must be evaluated for a uniform tree of depth h and constant width W, even when α and β are chosen to be the optimal minimax values.[8] Acceleration anomalies can also occur when the number of processors is small, say two or three.

Finkel and Fishburn have proposed a *tree-splitting algorithm* that maps a look-ahead tree onto a processor tree with the same interconnection structure.[32] The information-transfer overhead is small, due to the close match between the communications requirements and the interconnections. However, this is a brute-force

An efficient search method must involve AND and OR pruning.

search algorithm, and pruning is not considered in process assignments. The speedup drops to $\sqrt{k}$ under the best-case ordering, where k is the number of processors.

In the *mandatory-work-first scheme*,[33] the minimum tree searched in a serial algorithm is searched in parallel during the first phase. The resulting α-β window is used in the second phase, during which the rest of the tree is searched. This scheme performs better than the tree-splitting scheme under best-case ordering, but can be worse in worst-case ordering. In the latter case, many nodes pruned in the tree-splitting scheme might be visited in the second phase.

Another approach is to use a best-first search, such as the SSS* algorithm.[34] SSS* is effective in searching a randomly or poorly ordered tree, but requires more space and is not significantly better than an α-β search on strongly ordered trees. Kumar and Kanal have shown that the SSS* algorithm can be interpreted as a branch-and-bound procedure, and they have presented two parallel implementations of SSS*.[35]

Previous approaches to parallel game-tree search have emphasized reduction of the information-transfer overhead, but paid little attention to information-deficiency overhead. We will consider the information-deficiency overhead in the illustrative context of the scheduling of parallel logic programs.

Parallel logic programs. Logic programming is a programming methodology based on Horn-clause resolution.[36] An example of a high-level language for logic programming is Prolog. Execution of a logic program can be considered as the search of an AND/OR tree.[35,37] The root represents the initial problem queried, the OR nodes represent (sub)goals, and the AND nodes represent clauses. All subgoals in the same body of a clause are children of an AND node. A (sub)goal (OR node) and its children display the choices of clauses with the same head. The terminal nodes denote clauses or subgoals that cannot be decomposed.

Searching an AND/OR tree for a logic program is quite different than searching other types of search trees. First, in contrast to extremum searches that find the best solution, solving a logic program corresponds to finding any or all solutions that satisfy the given conditions, the implicative Horn clauses, and the consistent binding of variables for the AND nodes. Second, the value of a node in the AND/OR tree for a logic program is either TRUE (success) or FALSE (failure). A node is usually selected for evaluation on the basis of a fixed order, such as the depth-first search. Third, a variable in a logic program can be bound to several values, and some subgoals might share a common variable.

An efficient search method must involve pruning. Two kinds of pruning exist here. In an AND pruning, if one of the children of an AND node is found to be FALSE, then all remaining children of this AND node can be pruned. Likewise, in an OR pruning, if one of the children of an OR node is found to be TRUE, then all remaining children of this OR node can be pruned. It should be noted that OR pruning applies only if the OR node shares no variables with its siblings.

Much research strives for parallel execution of logic programs. Conery

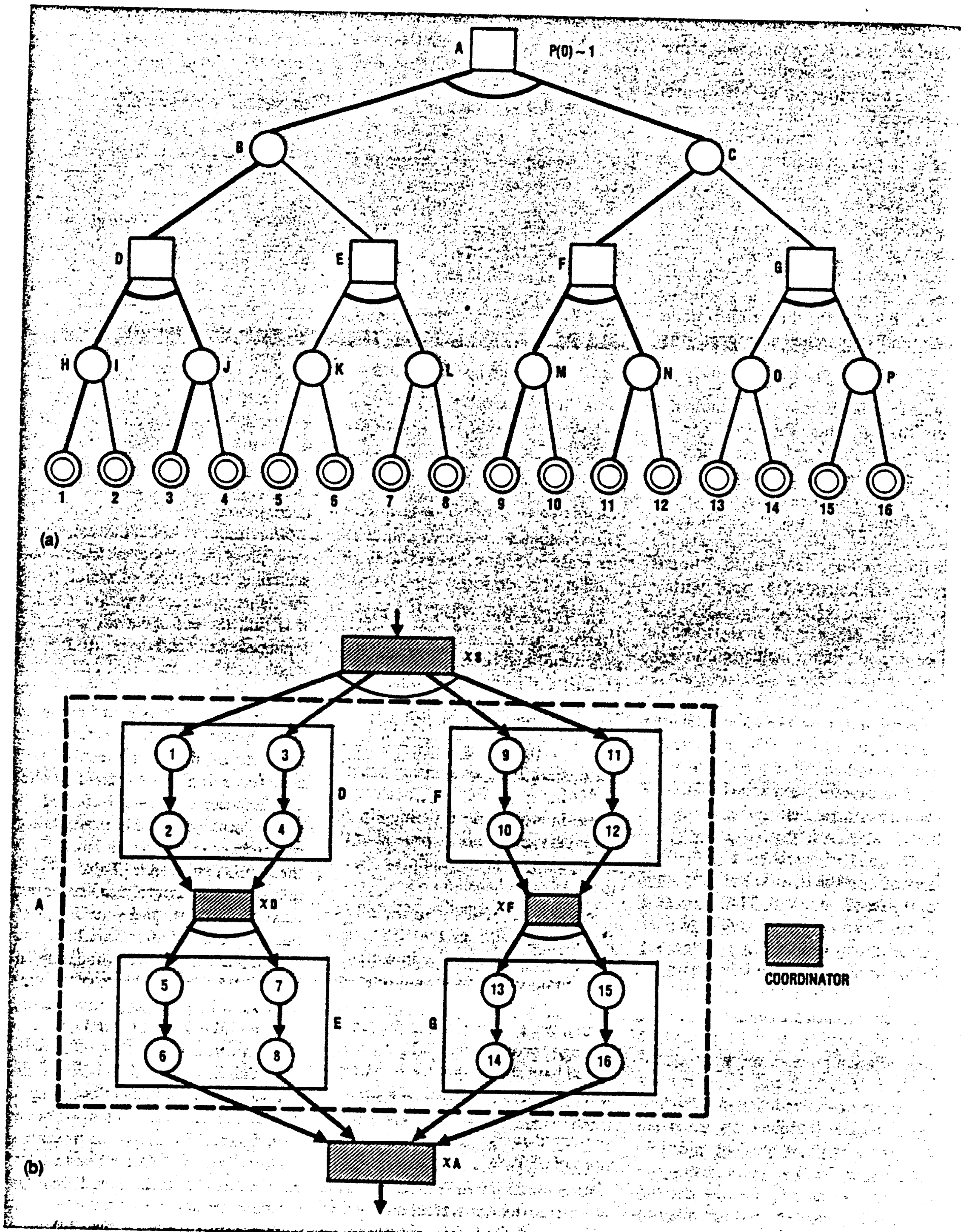


Figure 4. A binary AND/OR search tree (a) with high success probability and the corresponding fail-token-flow graph G_f. (b) AND nodes are represented as squared nodes; OR nodes are represented as circular nodes.

and Kibler[37] have classified four kinds of parallelism of logic programs: AND parallelism, OR parallelism, stream parallelism, and search parallelism; they have also investigated AND parallelism. Furukawa et al.[38] and Ciepielewski et al.[39] have discussed OR parallelism, while Lindstrom et al.[40] have addressed stream parallelism and pipelined Prolog processors.

However, very few studies have addressed processor assignment as a means to reduce information-deficiency overhead. Below, we present an algorithm that schedules searches of nodes according to estimated probabilities of a terminal node being true and does not distinguish AND and OR parallelism.

A new scheduling algorithm. Consider the case in which all terminal nodes have the value TRUE. For a binary AND/OR tree of height h (h is even and the root is at level 0), the solution tree is found after $2^{h/2}$ terminal nodes have been visited, as shown in Figure 4a. Once 1, 3, 9, and 11 have been visited, the root is determined to be true. In contrast, if all terminal nodes are FALSE, one can determine that the root is false by visiting $2^{h/2}$ terminal nodes (nodes 1, 2, 5, and 6 in Figure 4a). These observations imply that when most of the terminal nodes in a subtree are TRUE, searching the subtree by assuming that its root is TRUE is more efficient; otherwise, the subtree should be searched by assuming that its root is FALSE.

For the AND/OR tree in Figure 4a, we see that in a sequential search, if node 1 fails, then node 2 is examined; otherwise node 3 is examined next. That is, whether node 2 or node 3 is examined depends on the result of searching node 1. Similarly, the traversal of node 5 depends on the results of traversing nodes 1, 2, 3, and 4. A *fail-token-flow graph*, $\mathbf{G}_f$, as depicted in Figure 4b for the tree in Figure 4a, can be drawn, according to this dependence information. A node (circle) in the graph is active only if it receives a fail-token from an incident edge. When a terminal node in the search tree is found to be false, a fail-token is sent along the direction of the corresponding edge. The coordinator (shaded box) in the graph represents a control mechanism that coordinates the activities of the connected blocks. When a fail-token is received from any of the incident edges of a coordinator, fail-tokens are sent to *all* directly connected nodes. At the same time, any node searched in the block directly connected to this coordinator can be terminated, because it does not belong to the solution tree. For example, when node 1 is found to be false, a fail-token is sent to node 2. If node 2 is found to be false, a fail-token is sent to coordinator X_D. Any node concurrently searched in block D can then be terminated.

The token-flow graph of AND/OR tree roots is modular and can be decomposed: subgraphs correspond to nonterminal tree nodes.

A simple parallel search strategy can be derived with the aid of $\mathbf{G}_f$. To effectively search the tree, no more than $2^{h/2}$ processors are needed. A parallel depth-first search is applied in the first h steps by generating all children of a selected AND node, but only the leftmost child of a selected OR node. As an example, nodes 1, 3, 9, and 11 in the search tree are assigned to four processors in the fourth step. This corresponds to generating fail-tokens to activate these nodes in $\mathbf{G}_f$ (Figure 4b). If a node, say 3, is found to be FALSE, then a fail-token is generated and the idle processor is assigned to evaluate node 4. Close examination of Figure 4b shows that for each column of $\mathbf{G}_f$ there must be at least one node with the value TRUE if a solution tree exists. When a node is found to belong to the solution tree, all nodes on the path from the initial start node to this node in $\mathbf{G}_f$ must have failed. Processors for searching the AND/OR tree can be scheduled according to the state of execution in $\mathbf{G}_f$ at any time.

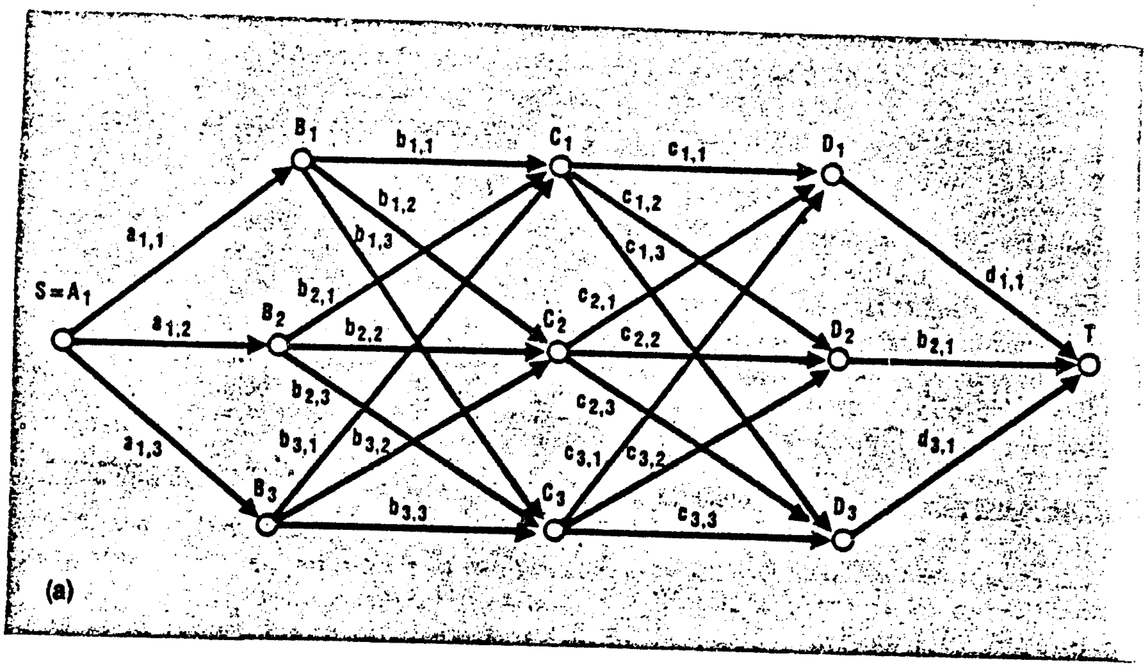
When the AND/OR tree is complete, and $Pr(h)$ (the probability that a terminal node is TRUE) is constant, $Pr(0)$ (the probability for a solution tree is to be found from the root, which is assumed to be an OR node) can be shown to be close to one for $Pr(h) > 0.618$. The threshold is 0.382 ($= 1 - 0.618$) if the root is an AND node. In both cases, a node with the value TRUE can be found quickly in each column of $\mathbf{G}_f$. As a result, the speedup is close to one.

On the other hand, if $Pr(h)$ is small, then the probability for a solution tree to exist at the root is close to zero and the above strategy is no longer suitable because a large number of nodes must be evaluated in each column of $\mathbf{G}_f$. In this case, the scheduling should be done according to the *success-token-flow graph*, $\mathbf{G}_s$. $\mathbf{G}_s$ is the dual of $\mathbf{G}_f$, in the sense that a success token replaces a fail token and the columns in $\mathbf{G}_f$ are transposed to become the rows in $\mathbf{G}_s$. Since searching for failure from an AND node is equivalent to searching for success from an OR node, the above scheduling algorithm can be extended with respect to $\mathbf{G}_s$.

The token-flow graph obtained for the root of an AND/OR tree is modular and can be decomposed into modular token-flow subgraphs corresponding to all nonterminal nodes in the tree. If the probability of leading to a solution tree for a nonterminal node can be refined as the search progresses, the corresponding token-flow subgraph can be rederived. An idle processor can be scheduled according to the token-flow subgraph derived for the root of the given subtree. We have proposed a multiprocessor architecture, MALOP, which is based on an intelligent search strategy and effective scheduling.[41]

In summary, the important issues in parallel AND/OR-tree search are the granularity of parallelism, the parallel selection of nodes for evaluation, and the intelligent pruning of unnecessary nodes. Processors should know the global state of search in order to select the nodes for expansion and should be able to tell other processors to prematurely terminate their tasks, when necessary. The architecture should support dissemination of this information.

Figure 5. A graph with five stages and three nodes in each intermediate stage (a); an AND/OR-graph representation of the reduction in finding an optimal path in a three-stage graph (b). The problem in (b) is to find $\min_{i,j,k \in \{1,2\}} \{a_{i,j} + b_{j,k}\}$; AND nodes are represented as squared nodes and indicate summations; OR nodes are represented as circular nodes and indicate comparisons.



Dynamic programming

Dynamic programming, a powerful optimization methodology, can apply to many areas, including optimal control, industrial engineering and economics.[42] In general, DP transforms the problem into a form suitable for optimization, but it is not an algorithm for optimizing the objective function. One can represent a problem solvable by DP as a multistage problem, a divide-and-conquer problem, or an acyclic AND/OR graph-search problem. Various computational approaches can be used, depending on the formulation and representation. We discuss DP problems separately here because they illustrate the effects of representation on the design of the supporting multiprocessing system.

A DP formulation is characterized by a recursive equation whose left-hand side identifies a function name and whose right-hand side is an expression containing the maximization (or minimization) of values of some monotone functions. Depending on the form of the functional equation, a DP formulation can be classified into four types: *monadic-serial, polyadic-serial, monadic-nonserial*, and *polyadic-nonserial*. Monadic and polyadic DP formulations are distinct approaches to representing various optimization problems; DP formulations can solve serial and nonserial optimization problems. *Serial optimization problems* can be decomposed into stages, and variables in one stage depend on variables in adjacent stages only. Problems such as sequential control, resource allocation, fluid flow, circuit design, and scheduling belong to this class. If variables in one stage are related to variables in other stages, the problem is a *nonserial optimization problem*. Examples include finding the optimal binary search tree and computing the minimum-cost order of multiplying a string of matrices.

To illustrate the concept of serial problems, consider the example of finding the shortest path in a multistage graph, as depicted in Figure 5a. Let $c_{i,j}$ be the cost of edge (i,j). The cost of a path from source S to sink T is the sum of costs on the edges of the path. Define $f_1(i)$ as the minimum cost of a path from I to T. The cost of getting from I to T via neighbor J is $c_{i,j} + f_1(j)$. To find $f_1(i)$, paths through all possible neighbors must be compared. Hence, the problem can be represented as

$$f_1(i) = \min_j [c_{i,j} + f_1(j)] \quad (4)$$

This is a *forward functional equation*. The formulation is *monadic*; that is, the cost function involves one recursive term only.

From Equation (4), $f(C_1)$, the minimum cost from C_1 to T is

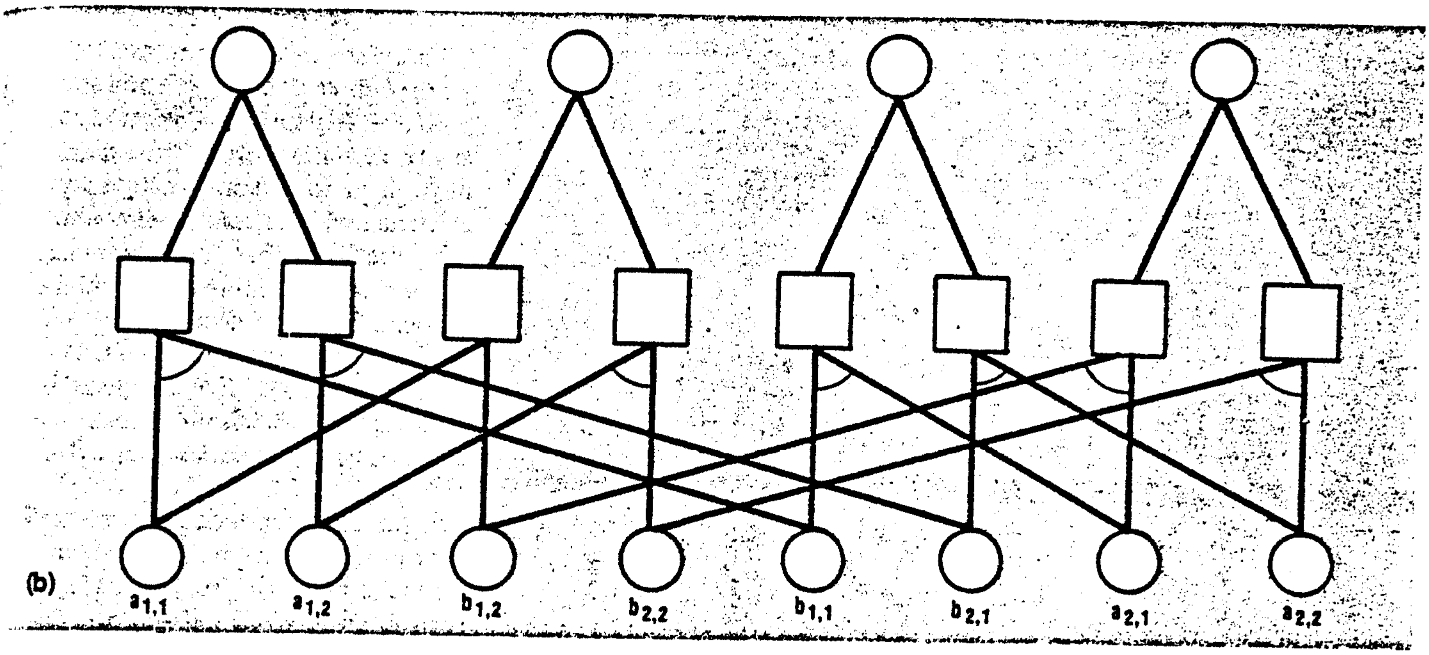
$$f(C_1) = \min\{c_{1,1} + d_{1,1},\ c_{1,2} + d_{2,1},\ c_{1,3} + d_{3,1}\} \quad (5)$$

Equation (5) can be interpreted as an inner-product operation in respect to addition and minimization. If we define matrix multiplication in terms of a closed semi-ring (R, MIN, +, +∞, 0) in which MIN corresponds to addition and + corresponds to multiplication in conventional matrix multiplications,[7] then equation (5) becomes $f(C) = \mathrm{C} \cdot \mathrm{D}$, where C is a cost matrix and D is a cost vector. It is easy to see that searching the shortest path in a multistage graph with a forward monadic DP formulation is equivalent to multiplying a string of matrices, i.e., $\mathrm{A} \cdot (\mathrm{B} \cdot (\mathrm{C} \cdot \mathrm{D}))$.

The same problem can be generalized to find the optimal path from any vertex i to any other vertex j. The functional equation is

$$f_2(i,j) = \min_k [f_2(i,k) + f_2(k,j)] \quad (6)$$

where $f_2(i,j)$ is the minimum cost of getting from I to J. This cost function is polyadic because it involves more than one recursive term. A divide-and-conquer formulation is a special case of polyadic-serial formulations.



AND/OR graphs can also be used to represent serial DP problems. Basic operations in comparisons of partial solutions over all alternatives are represented as OR nodes. AND nodes represent operations involving computations of a cost function, such as summations. Figure 5b shows an AND/OR graph for reducing the search of the shortest path in a three-stage graph with two nodes in each stage. Gensi and Montanari have shown that formulating a DP problem in terms of a polyadic functional equation is equivalent to searching for a minimal-cost solution tree in an AND/OR graph with monotone cost function.[43]

A nonserial DP problem can be represented in monadic or polyadic form.[44] A monadic-nonserial formulation is an extension of equation (4) in which the dependence of the functional term involves variables in more than one adjacent stage. A polyadic-nonserial formulation is usually represented in the form of an acyclic AND/OR graph in which edges can extend between any two arbitrary levels of the graph.

Parallel processing has been applied to DP problems. Guibas, Kung, and Thompson have proposed a VLSI algorithm for solving the optimal parenthesization problem,[45] for which linear pipelines have also been proposed recently.[46] Clarke and Dyer have designed a systolic array for curve and line detection in terms of a nonserial formulation.[47] However, these designs were directed toward implementation of a few special cases of DP formulations.

The choice of an architecture to support a serial DP problem depends on the formulation. First, if the problem is represented in a polyadic form and considered a divide-and-conquer problem, the architecture discussed above under the divide-and-conquer heading can be applied. For example, the problem of finding the shortest path in a multistage graph can be considered as the multiplication of a string of matrices, which can be decomposed into the multiplication of two or more substrings of matrices.

Second, equivalence between polyadic representations and AND/OR graphs allows various graph-search techniques to be translated into techniques for solving DP problems. Sometimes, when the AND/OR graph is regular, it can be mapped directly into a systolic array.[15]

Third, a problem can be represented in a monadic form and solved with a pipelining approach. This approach is suitable when many alternative partial solutions must be compared. Below, we illustrate this approach for evaluating the multiplication of a string of matrices.

Figure 6 depicts a scheme for computing $(A \cdot (B \cdot (C \cdot D)))$ for the multistage graph in Figure 5a. An *iteration* is defined as a shift-multiply-accumulate operation in respect to the time at which a row or column of the input matrix enters a given processor in the systolic array. Note that the same iteration number is carried out at different times in different processors (iteration numbers are indicated in Figure 6a).

In the first three iterations, $C \cdot D$ is evaluated. The control signal FIRST is one; D, the input vector is serially shifted into the systolic array; and the result vector, $\{f(C_i), i=1,2,3\}$, remains stationary. At the end of the third iteration, FIRST is set to zero. In the following three iterations, $B \cdot (C \cdot D)$ is computed. Note that matrix B is transposed, and the ith column of matrix B is fed into K_i. The input vector, $\{f(C_i), i=1,2,3\}$, remains stationary, while the result vector $\{f(B_i), i=1,2,3\}$, is shifted. At the end of the sixth iteration, the output vector $\{f(B_i), i=1,2,3\}$, is formed. In the last three iterations, input vectors A and $\{f(B_i)\ i=1,2,3\}$ are shifted into P_1 to form the final result.

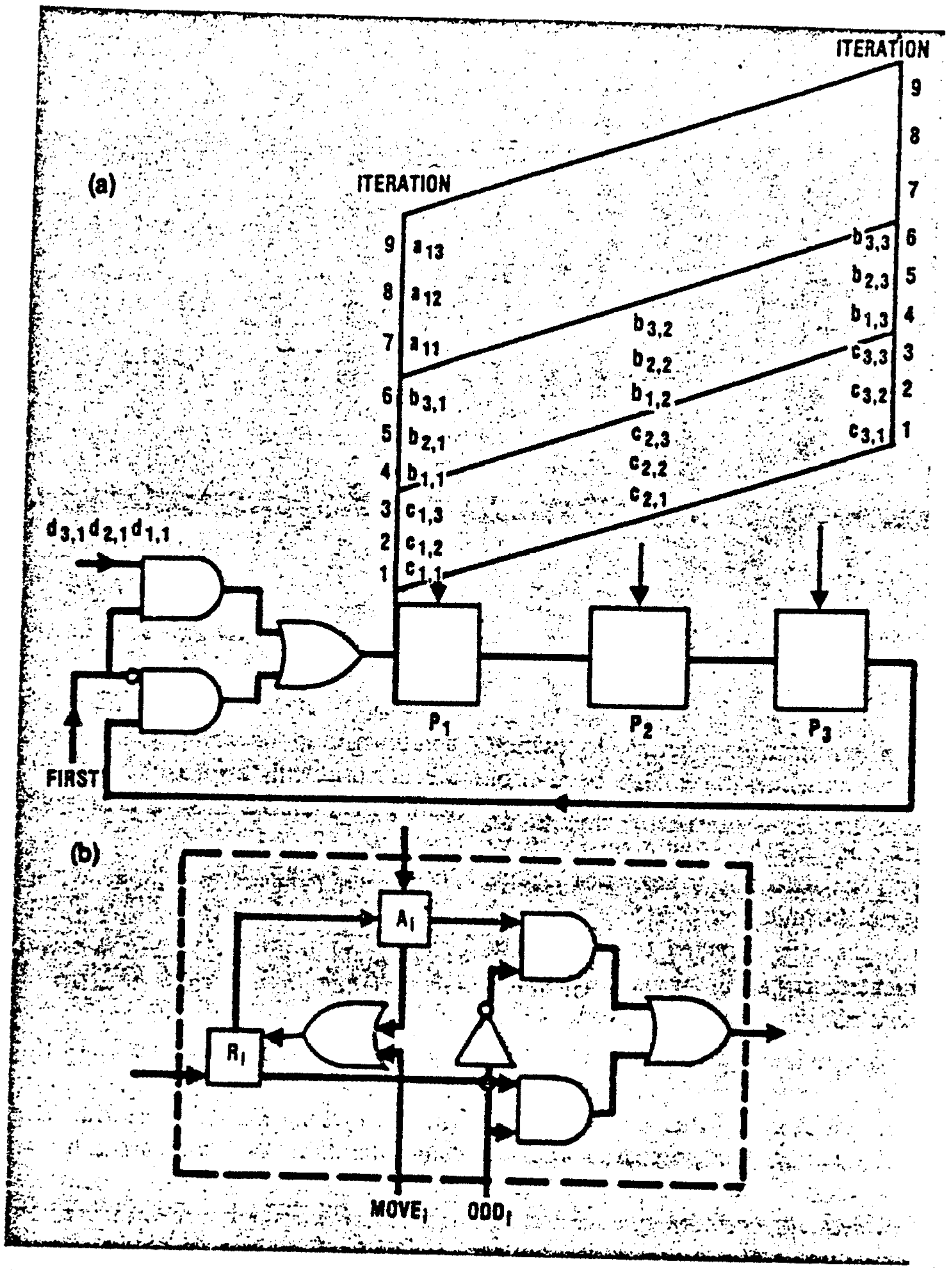


Figure 6. A pipelined version of a systolic array for multiplying a string of matrices (a); processor structure for K_i (b).

For the systolic array in Figure 6a, shifted data alternates between the input vector and the result vector every three iterations. The processor structure of K_i, depicted in Figure 6b, can control this alternation. R_i is a register that stores an element of the input vector, and A_i is the accumulator that stores the temporary result of an element of the result vector. Control signals ODD_i and $MOVE_i$ control the data paths. When the number of matrix multiplications is odd, ODD_i is one; hence, R_i is connected to the output and the input vector is shifted along the pipeline. When the number of matrix multiplications is even, ODD_i is zero, A_i is connected to the output, and the result vector is shifted. At the end of a matrix multiplication, the generated result vector becomes the input vector in the next iteration and is moved, by the control signal $MOVE_i$, from A_i to R_i.

In general, searching a multistage graph with N stages and k nodes in each stage takes $(N-1)k$ iterations with k processors. Because there are no delays between feedings of the input matrices into the systolic array, a processor utilization is very close to one when N and k are large.

Few architectures solve nonserial DP problems directly. In an AND/OR graph representation of nonserial problems, edges may connect nodes at any two arbitrary levels. These graphs might have to be searched by an architecture with a flexible interconnection, such as a dataflow computer. Another approach is to transform the nonserial problem into a serial one and solve it with approaches developed for serial problems.[15] For problems in monadic-nonserial formulations, the dependence of variables can be removed by using one variable to represent the Cartesian product of several dependent variables. For problems in a polyadic-nonserial representation, such as an AND/OR graph, the dependence can be removed by replacing each edge that connects nodes not at adjacent levels with multiple edges that connect nodes at adjacent levels. This approach has been used in designing a systolic array for finding the optimal binary search tree.[15]

Research in problem solving usually aims at developing better algorithms. Unnecessary combinatorial searches should be avoided, because they do not contribute to the quality of the solutions. Evidence of this is clear in the efforts to design optimal algorithms and to understand the reasoning process in artificial intelligence. However, searching becomes inevitable when a good algorithm has been developed and is an essential to many applications.

In this article, we have investigated the limitations of multiprocessing in solving combinatorial searches. The suitability of multiprocessing depends on problem complexity, problem representation, and the corresponding search algorithms. Problem complexity should be low enough that a serial computer can solve the problem. Problem representations are very important because they are related to the search algorithms.

However, the question of deciding which representation leads to an efficient search remains open. Moreover, efficient architectures to evaluate various search algorithms differ. That is why we have developed functional requirements for a given search algorithm, requirements that allow efficient mapping of a search algorithm on a general-purpose multiprocessor and development of special-purpose processors for searching.

In this article, we have not attempted to list all possible cases, but to illustrate the different approaches through examples (Table 1). We hope these guidelines and examples can help designers select appropriate multiprocessing systems for solving combinatorial search problems. □

Table 1. Functional requirements of different paradigms of search algorithms. (The magnitudes of large and small granularities in various algorithms differ. Special interconnections include the tree architecture.)

ALGORITHM		FUNCTIONAL REQUIREMENTS	TASKS
Divide-and-conquer	Large granularity	Loosely coupled	Balance load
	Small granularity	Tightly coupled; special interconnections	Transfer control and data
Branch-and-bound	Lower-bound test only	Loosely coupled; broadcast capability	Balance load; share incumbent
	Dominance tests	Tightly coupled; shared memory	Balance load; share dominating nodes
Serial acyclic AND/OR-graph search	Large granularity	Loosely coupled; broadcast capability	Balance load; share state of evaluation
	Small granularity	Tightly coupled; special interconnections	Transfer control and data
Nonserial acyclic AND/OR-graph search	Large granularity	Dataflow processing	Share resources; coordinate tasks
	Small granularity	Map to serial AND/OR-graph search	Transfer control and data

References

1. A. C. Shaw, *The Logical Design of Operating Systems*, Prentice-Hall, Englewood Cliffs, N.J., 1974.
2. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, Calif., 1979.
3. A. Martelli and U. Montanari, "Additive AND/OR Graphs," *Int'l Joint Conf. Artificial Intelligence*, 1973, pp. 1-11.
4. N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga, Menlo Park, Calif., 1980.
5. A. Barr and E. A. Feigenbaum, *The Handbook of Artificial Intelligence*, Vol. 1-3, Kaufmann, Los Altos, Calif., 1981.
6. E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Potomac, Md., 1978.
7. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.
8. D. E. Knuth and R. W. Moore, "An Analysis of Alpha-Beta Pruning," *Artificial Intelligence*, Vol. 6, 1975, pp. 293-326.
9. V. Kumar and L. Kanal, "A General Branch-and-Bound Formulation for Understanding and Synthesizing AND/OR Tree Search Procedures," *Artificial Intelligence*, Vol. 21, 1983, pp. 179-198.
10. F. Peters, "Tree Machine and Divide-and-Conquer Algorithms," *Conpar 81, Lecture Notes CS111*, 1981, pp. 25-35.
11. F. Burton and Huntbach, "Virtual Tree Machines," *IEEE Trans. Computers*, Vol. C-33, No. 3, Mar. 1984, pp. 278-280.
12. E. Horowitz and A. Zorat, "Divide and Conquer for Parallel Processing," *IEEE Trans. Computers*, Vol. C-32, No. 6, June 1983, pp. 582-585.
13. C. Tang and R. C. T. Lee, "Optimal Speedup of Parallel Algorithm Based on the Divide-and-Conquer Strategy," personal communication.
14. D. J. Kuck, "A Survey of Parallel Machine Organization and Programming," *ACM Computing Surveys*, Vol. 9, No. 1, Mar. 1977, pp. 29-59.
15. G.-J. Li and B. W. Wah, "Parallel Processing for Dynamic Programming," to appear *Int'l Conf. Parallel Processing*, August 1985.
16. E. L. Lawler and D. W. Wood, "Branch-and-Bound Methods: A Survey," *Operations Research*, Vol. 14, 1966, pp. 699-719.
17. T. Ibaraki, "Computational Efficiency of Approximate Branch-and-Bound Algorithms," *Mathematical Operations Research*, Vol. 1, No. 3, 1976, pp. 287-298.
18. B. W. Wah and E. Y. W. Ma, "MANIP—A Multicomputer Architecture for Solving Combinatorial Extremum Search Problems," *IEEE Trans. Computers*, Vol. C-33, No. 5, May 1984, pp. 377-390.
19. B. W. Wah and C. F. Yu, "Probabilistic Modeling of Branch-and-Bound Algorithms," *Proc. Compsac*, Nov. 1982, pp. 647-653; will also appear in *IEEE Trans. Software Engineering*, Oct. 1985.
20. G.-J. Li and B. W. Wah, "Computational Efficiency of Parallel Approximate Branch-and-Bound Algorithms," *Proc. Int'l Conf. Parallel Processing*, 1984, pp. 473-480.
21. B. C. Desai, "The BPU: A Staged Parallel Processing System to Solve the Zero-One Problem," *Proc. ICS 78*, Dec. 1978, pp. 802-817.
22. M. Imai, T. Fukumara, and Y. Yoshida, "A Parallelized Branch-and-Bound Algorithm: Implementation and Efficiency," *System Computer Controls*, Vol. 10, No. 3, 1979, pp. 62-70.

23. O. I. El-Dessouki and W. H. Huen, "Distributed Enumeration on Network Computers," *IEEE Trans. Computers*, Vol. C-29, No. 9, Sept. 1980, pp. 818-825.

24. B. W. Wah, G.-J. Li, and C. F. Yu, "The Status of MANIP—A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems," *Proc. 11th Ann. Int'l Symp. Computer Architecture*, June 1984, pp. 56-63.

25. B. W. Wah and K. L. Chen, "A Partitioning Approach to the Design of Selection Networks," *IEEE Trans. Computers*, Vol. C-33, No. 3, Mar. 1984, pp. 261-268.

26. C. F. Yu and B. W. Wah, "Virtual-Memory Support for Branch-and-Bound Algorithms," *Proc. Compsac*, Nov. 1983, pp. 618-626.

27. C. F. Yu and B. W. Wah, "Efficient Branch-and-Bound Algorithms on a Two-Level Memory Hierarchy," *Proc. Compsac*, Nov. 1984, pp. 504-514.

28. G.-J. Li and B. W. Wah, "How to Cope with Anomalies in Parallel Approximate Branch-and-Bound Algorithms," *Proc. Nat'l Conf. Artificial Intelligence*, 1984, pp. 212-215.

29. T. H. Lai and S. Sahni, "Anomalies of Parallel Branch-and-Bound Algorithms," *Comm. ACM*, Vol. 27, No. 6, June 1984, pp. 594-602.

30. T. A. Marsland and M. Campbell, "Parallel Search of Strongly Ordered Game Trees," *ACM Computing Surveys*, Vol. 14, No. 4, Dec. 1982, pp. 533-551.

31. G. Baudet, "The Design and Analysis of Algorithms for Asynchronous Multiprocessors," Tech. Rep., Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., 1978.

32. R. Finkel and J. Fishburn, "Parallelism in Alpha-Beta Search," *Artificial Intelligence*, 1982, pp. 89-106.

33. S. Akl, D. Barnard, and R. Doran, "Design, Analysis, and Implementation of a Parallel Tree Search Algorithm," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. PAMI-4, Mar. 1982, pp. 192-203.

34. G. Stockman, "A Minimax Algorithm Better than Alpha-beta?" *Artificial Intelligence*, Vol. 12, 1979, pp. 179-196.

35. V. Kumar and L. Kanal, "Parallel Branch-and-Bound Formulations for AND/OR Tree Search," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. PAMI-6, 1984.

36. R. Kowalski, *Logic for Problem Solving*, North Holland, New York, 1979.

37. J. Conery and D. Kibler, "AND Parallelism in Logic Programming," *Tutorial on Parallel Logic Programming, Int'l Conf. Parallel Processing*, D. DeGroot, ed., 1984, pp. 13-17.

38. K. Furukawa, K. Nitta, and Y. Matsumoto, "Prolog Interpreter Based on Concurrent Programming," *Proc. First Int'l Logic Programming Conf.*, 1982, pp. 38-44.

39. A. Ciepielewski and S. Haridi, "Control of Activities in OR-Parallel Token Machine," *IEEE Int'l Symp. Logic Programming*, 1984.

40. G. Lindstrom and P. Panangaden, "Stream-Based Execution of Logic Programs," *Proc. 1984 Int'l Symp. Logic Prog.*, Feb. 1984, pp. 168-176.

41. G. J. Li and B. W. Wah, "MALOP: a Multicomputer Architecture for Solving Logic Programming Problems," *Proc. Int'l Conf. Parallel Processing*, 1985.

42. R. Bellman and S. Dreyfus, *Applied Dynamic Programming*, Princeton University Press, Princeton, N.J., 1962.

43. S. Gensi, U. Montanari, and A. Martelli, "Dynamic Programming as Graph Searching: An Algebraic Approach," *J. ACM*, Vol. 28, No. 4, Apr. 1981, pp. 737-751.

44. U. Bertele and F. Brioschi, *Nonserial Dynamic Programming*, Academic Press, New York, 1972.

45. L. Guibas, H. Kung, and C. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms," *Proc. Caltech Conf. VLSI: Architecture, Design, Fabrication*, 1979, pp. 509-525.

46. P. Varman and V. Ramakrishnan, "Dynamic Programming and Transitive Closure on Linear Pipelines," *Proc. Conf. Parallel Processing*, 1984, pp. 359-364.

47. M. Clarke and C. Dyer, "Systolic Array for a Dynamic Programming Application," *Proc. 12th Workshop Applied Imagery Pattern Recognition*, 1983.

Questions about this article can be directed to Benjamin W. Wah, School of Electrical Engineering, Purdue University, West Lafayette, IN 47907.

Benjamin W. Wah is an associate professor in the School of Electrical Engineering at Purdue University. His current research interests include parallel computer architectures, distributed databases, and theory of algorithms.

Wah received the BS degree in 1974 and the MS degree in 1975 in electrical engineering and computer science from Columbia University. He received the MS degree in computer science in 1976 and the PhD degree in electrical engineering in 1979 from the University of California, Berkeley. He has been a Distinguished Visitor of the IEEE Computer Society since 1983.

Guo-jie Li is a doctoral candidate in computer science and engineering at Purdue University. His research interests include parallel processing, computer architecture, and artificial intelligence.

Li graduated from Peking University in 1968 and received the MS degree in computer science and engineering from the University of Science and Technology and the Institute of Computing Technology, Chinese Academy of Science, in 1981.

Chee Fen Yu is a doctoral candidate in electrical engineering at Purdue University. His current research interests include computer architecture and artificial intelligence.

Yu received the BE degree in electrical engineering from the University of Malaya in 1980 and the MS degree in electrical engineering from Purdue University in 1983.