To reduce the search space, unnecessary e
nance relations. When a node P_i dominates a
subtree rooted at P_i contains a solution node w
the minimum (or maximum) solution value o
example, suppose the assignment for a subset
knapsack in the 0/1 knapsack problem has b
ments, P_1 and P_2, on this subset of objects,
assigned to the knapsack for P_1 is more than tha
objects assigned in P_1 is less than that of P_2, the
P_1 will be better than the best solution expande
inates P_2. Another example of dominance tests
cussed in Section 4. A special case of dominan
that are used in branch-and-bound algorithms
Suppose a solution with value v has already b
with lower bounds greater than v can be termin
better solutions. The minimum of the solution v
conveniently kept in a single location called the

A problem can be represented in multiple f
problem may either be represented in an OR t
bound algorithm [30], or formulated in dynami
acyclic AND/OR-graph search. As another exa
from a set of numbers can be solved by either a
algorithm [1] or a decision-tree (OR-tree) sear
cedures for various representations are equivalen
the same solution(s). Kumar and Kanal have sh
procedures for state-space representations (e.g. A
searches (e.g. AO* [64]), and game-tree searche
to branch-and-bound searches with dominance te

The efficiency of solving a given problem
Although efficient search procedures for some p
the general question of deciding the representatio
is still open for many problems, especially whe
Moreover, the best computer architecture for solv
lem is very problem dependent. Our objective is,
tional requirements of various search algorith
designers in assessing whether a general-purpo
given search algorithm, and in developing the m
algorithm. Special-purpose architectures can als
tional requirements.

In presenting the performance results in th
are usually assumed, although the search algorit
nously. The performance results for synchronous
that of asynchronous models. The effects of he
tions on time and space efficiency are studied. Th
ing the nodes for expansion and defining the level
gated.

MULTIPROCESSING OF COMBINATORIAL SEARCH PROBLEMS[1]

Benjamin W. Wah
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 West Springfield Avenue
Urbana, IL 61801

Guo-Jie Li
Institute of Computing Technology
Academia Sinica
P.O. Box 2704-1, Beijing
People's Republic of China

Chee-Fen Yu
Artificial Intelligence Laboratory
Intel Corporation
Santa Clara, CA 95051

ABSTRACT

This chapter presents three paradigms of representations for combinatorial search problems. Depending on the functions of the nonterminal nodes in the graphical representation, a search problem can be represented as an AND-tree, an OR-tree, and an AND/OR graph. This classification facilitates the design of unique computer architectures for supporting efficient evaluation of combinatorial search problems. For each representation, we develop theoretical bounds, efficient algorithms, and functional requirements of multiprocessing architectures, and illustrate these results by examples.

[1] This research was supported in part by National Aeronautics and Space Administration contract NCC 2-481 and in part by National Science Foundation grant MIP 88-10584.

Part of this chapter appeared in (a) "Multiprocessing of Combinatorial Search Problems," *IEEE Computer*, vol. 18, no. 6, June 1985, pp. 93-108; (b) "Coping with Anomalies in Parallel Branch and Bound Algorithms," *IEEE Transactions on Computers*, vol. C-34, no. 6, June 1986, pp. 568-573; and (c) "Optimal Granularity of Parallel Evaluation of AND-Trees," *Proc. Fall Joint Computer Conference*, ACM/IEEE, Nov. 1986, pp. 297-306.

1. INTRODUCTION

Multiprocessing refers to the co
computer hardware complex with mor
ventionally, multiprocessing is define
processors, input-output processors, da
the advent of VLSI technology, it be
general-purpose and special-purpose
definition of multiprocessing can be e
computer systems, systolic arrays, an
cessing can be considered synonymou
range from the IBM 360 computers wi
Cray X-MP, to the latest Fifth Generat

In using a multiprocessing syste
may be spent in designing a good para
rithm with a speedup proportional to
serial algorithm. This has to be don
involves tradeoffs in computational t
ments.

The solvability of the problem b
lem should be **polynomially solvable**
an exponential complexity with respe
mial time unless an exponential numbe
technologically infeasible when the pr
algorithm requires 2^N microseconds t
microseconds (or 366 centuries) to sol
speedup is possible, it would require
mately one second, and 2^{40} processor
able problems, approximate solutions
in a reasonable amount of time.

Another important issue is on
tiprocessing. It is important to obs
improve the computational efficie
extend the solvable problem spac
size N and complexity N^k can be s
sor. Assuming that N processors a
the new problem size that can be s
following equation.

$$N*N^k = X^k$$

The left-hand-side of the above equ
N^k units of time with N processors.
operations to be performed in solv
equation yields

$$X = N^{1+1/k}$$

* By saying that a problem is intracta
the best parallel algorithm cannot h
number of processors. Suppose this
algorithm can solve the problem in po
problem has an exponential complexity

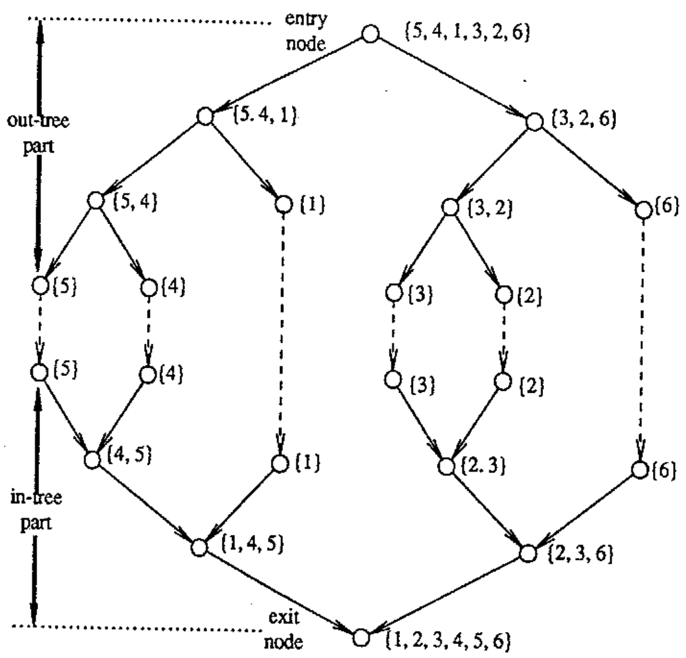


Figure 1. Outin-tree representation of the sort-merge algorithm.

processor utilization and communication overhead.

There have been contradictory views on granularity in previous studies: some researchers advocate a fine grain, while others suggest a coarse grain. For example, in designing the FFP machine [59], a small grain is chosen based on the hypothesis that appropriately designed small-grain multiprocessors will prove superior to large-grain ones in supporting ease and generality of parallel computations. In contrast, in Rediflow [36], large-grain parallelism is used to minimize communication overheads. Further, only qualitative results were derived in these studies.

A number of related studies have been carried out for scheduling tasks with tree precedence relationships in order to minimize completion time. The processing of such a task graph on a set of identical parallel processors such that the completion time is minimized have been extensively studied [12,14,27]. If preemption is allowed, then the same problem can be solved optimally either by Muntz and Coffman's Critical Path Scheduling (CPS) algorithm in O(N) time [63], or by other polynomial-time algorithms [23]. In the CPS algorithm, the next job chosen is the one with the longest length of unexecuted jobs. This longest path is called the **critical path**. If preemption is not allowed, then optimal scheduling algorithms have been obtained only for two cases: (a) all tasks have equal execution times and the precedence relationships are in the form of an intree (Hu's algorithm) [32] and (b) when two processors are used [11]. Hu's optimal scheduling algorithm is indeed a CPS algorithm. Many other cases have been proved to be NP-hard [48,71]. The CPS algorithm, besides efficient and optimal, is easy to implement and, consequently, is one of the most common scheduling algorithms [45,58].

In case that the precedence graph is a tree, that all processors are identical, and that each task requires t_i, $0<t_i \le t_{max}$, units of time to complete, the nonpreemptive CPS algorithm turns out to be almost-optimal in the sense that

$$T_p(k) \le T_{np}(k) \le T_p(k) + t_{max} \qquad (3)$$

where $T_{np}(k)$ and $T_p(k)$ are, respectively, the total time required by the nonpreemptive and preemptive CPS algorithms using k processors [35]. Some researchers have strived for nonpreemptive scheduling algorithms to solve scheduling problems with tree precedence [19,27,45]. Recently, Garey, Dolev, et al. have studied the scheduling of forests consisting of intrees and outtrees. Given a fixed number of processors, polynomial algorithms with high complexities to find an optimal schedule of these forests have been developed [19,25].

2.1. Scheduling Algorithms for Outin Trees

In studying the optimal scheduling of outin trees, an asynchronous model of parallel computation is adopted. A set of k identical parallel processors is assumed. These processors are connected by an interconnection network or a shared memory. The precedence graph of an outin tree is represented such that the entry node is at the top of the figure and the exit node is at the bottom. An arc is assumed to be always directed towards the bottom of the graph. The number inside a node is the task execution time, while the number next to a node, called its **length**, is the sum of the task execution times for nodes in the longest path from this node to the exit node. Figure 2(a) is an example of an outin tree.

The execution time of a task can be interpreted as either its maximum processing time or its expected processing time. In the former case, the worst-case time to complete the schedule is considered, while in the latter case the length of the schedule represents a rough estimate of the average time of computation. In some outin-tree problems, the execution time of each task can be predicted quite accurately. For example, in evaluating arithmetic expressions, the time to execute a primitive operation, such as a multiplication, is known. In other cases, the average execution times may have to be estimated from statistics or from previous experience. In all cases, the communication overhead is non-trivial when preemptions are allowed, and the task time should also include the overhead of preemptions.

The evaluation of an outin tree can be divided into the **splitting, all-busy,** and **combining** phases with respect to k, the fixed number of processors. In the splitting phase, the problem is decomposed, and the number of busy processors is increased from one up to at most k–1 (the number of busy processors must always be less than k if the number of available tasks at any time is less than k). In the combining phase, the subproblems are composed, and the number of busy processors is decreased from at most k–1 to one. During these two phases, some processors are idle. In contrast, in the all-busy phase, all the k processors are busy.

The interface between the all-busy and the combining phases, called a "heightline" by Schindler [68], is useful in determining the optimal schedule for outin trees. Schindler has proved that the schedule of a precedence graph is optimal if either the computations can be completed in only the all-busy or combining phase, or it can be partitioned into the all-busy and combining phases by

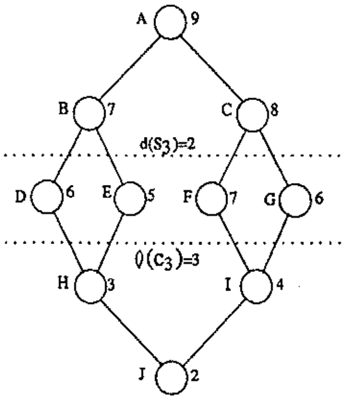


Figure 2a. Task precedence graph as outin tree.

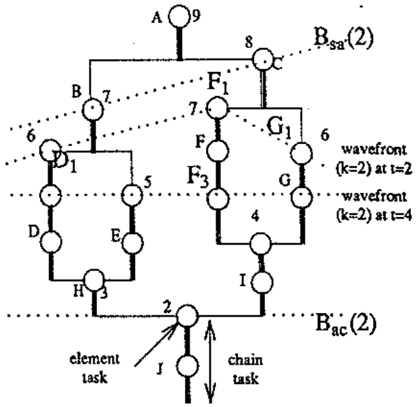


Figure 2b. Task precedence graph as outin tree using element tasks.

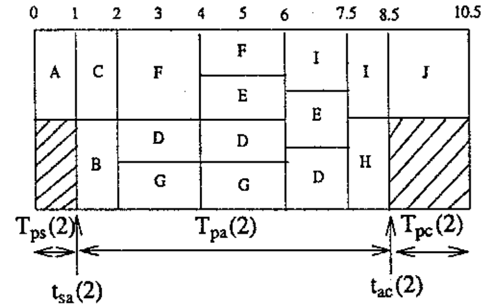


Figure 2c. Timing diagram for General Scheduling (processor sharing)

the heightline.

Three alternative scheduling algorithms for outin trees are considered in this section: the preemptive and nonpreemptive CPS algorithms, and the general scheduling algorithm.

Preemptive Critical Path Scheduling (PCPS) Algorithm. The PCPS algorithm schedules tasks according to the CPS scheduling algorithm, while allowing preemptions at specific times. A task (a node of the outin tree) are decomposed into element tasks, each of which requires one unit of execution time (see Figure 2b). Preemptions can occur between two element tasks. The PCPS algorithm has been proved to minimize the completion time of scheduling an outin tree [53].

General Scheduling (GS) Algorithm. GS algorithm [63] is a variation of preemptive scheduling algorithms in which each processor in the system is considered to have a certain amount of computing capacity rather than as a discrete unit, and this computing capacity can be assigned to tasks in any amount between zero and the equivalence of one processor. That is, processors are shared equally among all executable tasks.

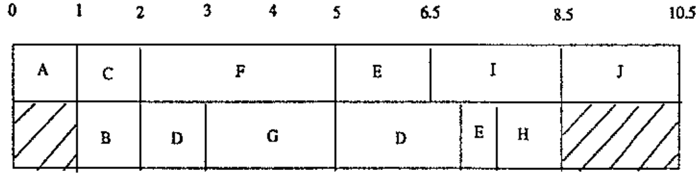


Figure 2d. Timing diagram for preemptive CPS scheduling

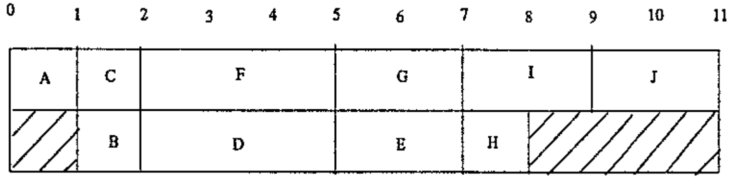


Figure 2e. Timing diagram for nonpreemptive CPS scheduling

Figure 2. Outin tree and CPS scheduling.

For example, if half of a processor is assigned to task P_i with execution time t_i, then it will take $2{\cdot}t_i$ units of time to complete P_i.

In the GS discipline, one processor is assigned to each of the k element tasks farthest from the exit node of the outin tree to be evaluated. If there is a tie in the lengths among u element tasks for the last v, u>v, processors, then v/u of a processor is assigned to each of these u element tasks. Processors are reassigned according to the CPS principle whenever a point is rearched such that, if the present assignment is continued, some element tasks will be computed at a rate faster than other element tasks that are farther from the exit node. The GS discipline is illustrated in Figure 2(c) in which the task graph in Figure 2(a) is scheduled.

Muntz and Coffman have proved the equivalence between the GS and PCPS algorithms [63]. That is, if preemptions were permitted, then the "processor-sharing" capability is not needed for optimal scheduling. To illustrate this equivalence, Figure 2(d) shows the preemptive schedule for the corresponding outin tree in Figure 2(a). Note that in the scheduling algorithms discussed in this chapter, all idle processors, if any, must be used to compute an available executable task.

At any time, a task is said to be **active** if either a processor or part of a processor is assigned to it. The total number of active tasks may be greater than the number of processors since some tasks may share processors. All active tasks form a wave-front in the outin-tree evaluation. Two particular times of the wave-fronts are of special interest when k processors are used: $t_{sa}(k)$ and $t_{ac}(k)$. The computation enters the all-busy phase at $t_{sa}(k)$ and enters the combining phase at $t_{ac}(k)$. In both times, the wave-fronts serve as phase-boundaries. We call the former phase-boundary $B_{sa}(k)$ and the latter $B_{ac}(k)$.

For the task graph in Figure 2(a), if the PCPS algorithm is employed, then $t_{sa}(2)=1$ and $t_{ac}(2)=8.5$ (see Figures 2(c) and 2(d)). The corresponding phase-boundaries $B_{sa}(2)$ and $B_{ac}(2)$ are indicated in Figure 2(b).

In practice, preemptions are usually restricted at the beginning of a time unit, so the overhead of a practical PCPS algorithm is equal to that of Hu's algorithm, which assumes that tasks have unit execution times. From Eq. (3) and the results in reference [63], we have

$$T_p(k) = T_{gs}(k) \le T_h(k) \le [T_{gs}(k)+1] = [T_p(k)+1] \qquad (4)$$

where $T_h(k)$ and $T_{gs}(k)$ are the times required by Hu's and GS algorithms, respectively. Eq. (4) shows that the behavior of GS is very close to that of any PCPS algorithm that only allows preemptions at the beginning of a time unit.

The GS algorithm is useful as a model for analyzing the properties of the PCPS algorithm in which preemptions are allowed at specific times. The granularities derived under the GS algorithm is the same as those of the PCPS algorithm in which task times are always integral.

Non-preemptive Critical Path Scheduling Algorithm. Nonpreemptive CPS algorithms are similar to the PCPS algorithm except that preemptions are not allowed. In the nonpreemptive CPS algorithm, one processor is assigned to each of the k nodes farthest from the exit node. If there is a tie in lengths among more than one node, then a left-to-right tie-breaking rule is used to assign a processor to one of these nodes. When a task of the outin tree is completed, the free processor is assigned to the node farthest from the root in the remaining outin tree to be evaluated. Figure 2(e) illustrates an example of nonpreemptive CPS scheduling.

If a preemptive (resp. nonpreemptive) CPS algorithm is applied, then the computational times required by k processors to complete the splitting, all-busy and combining phases are denoted by $T_{ps}(k)$, $T_{pa}(k)$ and $T_{pc}(k)$ (resp. $T_{nps}(k)$, $T_{npa}(k)$ and $T_{npc}(k)$). The longest path from a task to the exit node is selected as the **execution-path** through this task. For a task, if more than one such longest path exist, then a left-to-right orientation or any tie-breaking rule is applied.

For example, in Figure 2(b), the execution-path from task A_1 is (A_1, C_1, F_1, F_2, F_3, I_1, I_2, J_1, J_2) and task B_1 is the head of the execution-path (B_1, D_1, D_2, D_3, H_1, J_1, J_2). Note that when k processors are used, only the topmost k–1 path-heads are active in the splitting phase. Other path-heads are active in the all-busy phase.

In general, nonpreemptive scheduling is more useful when the task-switching overheads are high; however, it is more difficult to predict its performance and determine the optimal grain in parallel processing.

2.2. Bounds on Optimal Granularity of Scheduling Outin Trees

The criteria generally used to define the optimal granularity are the processor utilization (PU), kT^2, and AT^2, where k is the number of processors, T is the computational time, and A is the area of a VLSI implementation. The complexity

of divide-and-conquer algorithms in a SIMD model and the conditions to assure the optimal processor utilization have been studied [31]. However, processor utilization increases monotonically with decreasing number of processors, which means that PU achieves the maximum when one processor is used. Hence, PU is not an adequate measure for the effects of parallel processing. A more appropriate measure is the kT^2 criterion, which considers both PU and computational time, since

$$kT^2(k) = \frac{T(1)T(k)}{PU} \quad \text{where } PU=\frac{speedup}{k} \text{ and } T(k)=\frac{T(1)}{speedup} \qquad (5)$$

Minimizing kT^2 means reducing the computational time and maximizing the processor utilization. kT^2 is linearly related to AT^2 if the area of connection wires is proportional to the area of processing elements, as in systolic arrays. Both computational time and processor utilization are important in many applications, hence, kT^2 is a good criterion to optimize. In other applications, such as real-time processing, the completion time may be more critical and the PU is a secondary consideration. In this case, a different optimization criterion may have to be used.

In this chapter, we adopt kT^2 as a criterion of measuring processor-time efficiency; that is, given an outin-tree, we need to either choose k to minimize kT^2, or given a fixed k, determine the type of outin trees (their shapes, complexities, etc.) and its proper size that can be solved most efficiently by this system.

It is difficult to find the optimal granularity with respect to kT^2 directly because the optimal granularity depends on the execution time of each task and the shape of the outin tree. It is necessary to find parameters such that tight bounds on optimal granularities can be derived.

An important parameter in characterizing the optimal granularity is the total idle time. Let $\Phi_p(k)$ (resp. $\Phi_{np}(k)$) be the total amount of idle times when a preemptive (resp. nonpreemptive) scheduling algorithm with k processors is used. $\Phi_p(k)$ takes into account the idle times in both the splitting and combining phases. Clearly, $\Phi_p(k) = [\Phi_{ps}(k)+\Phi_{pc}(k)]$ and

$$kT(k) = T(1) + \Phi(k) \qquad (6)$$

Eq. (6) holds for both preemptive and nonpreemptive scheduling algorithms. The total idle time $\Phi_p(k)$ is related to both k and kT^2.

Due to space limitation, the following results on the bounds of optimal granularities are stated without proofs. Their proofs can be found in the references [53].

(a) When an outin tree is evaluated by the PCPS algorithm, $kT_p^2(k)$ is a concave function of k, that is, $kT^2(k)$ achieves the minimum when $k=k'$, and $kT^2(k)$ is monotonically decreasing (resp. increasing) with k when $k<k'$ (resp. $k>k'$).

(b) When an outin tree is evaluated by the PCPS algorithm and k>1, then the lower and upper bounds of k that minimizes $kT_p^2(k)$ are, respectively, $\{T_p(1)/(2h_{max})-1\}$ and $\{2T_p(1)/(h_{max})+1\}$, where h_{max} is the length of the critical path in the outin tree to be evaluated. No assumption has been made about the distribution of task times in these bounds.

(c) When an outin tree is scheduled by a nonpreemptive CPS algorithm, and $t_i>t_j$ if task i is a predecessor (resp. successor) of task j in the outtree (resp. intree) part, then k, the number of processors that minimizes $kT_{np}^2(k)$, is bounded between $[T_{np}(1)+t_{en}+t_{ex}]/(8h_{max})$ and $3T_{np}(1)\ /[d(s_k) + L(c_k) - 2t_{npa}(k)]$, where t_{en} is the task time of the entry node, t_{ex} is the task time of the exit node, $d(s_k)$ is the shortest path from the entry node to s_k, the maximum-all-busy level, $L(c_k)$ is the shortest length from c_k, the minimum all-busy level, to the exit node, and $t_{npa}(k)$ is the longest task-time among all tasks in the all-busy phase when k processors are used.

Results (a) and (b) show that optimal granularity can be found easily by a binary search, which can be completed within about $\log_2(T_p(1)/h_{max})$ steps.**** Each step in the binary search tests whether $\Delta(kT_p^2(k))$ is positive. If it is, then a smaller value of k will be checked in the next step, otherwise, a larger k will be tested.

For instance, suppose that N items need to be sorted. It is well-known that $T(1) = N{\cdot}\log N$ if a merge-sort algorithm is used. In this case, the overhead in the intree part dominates that of the outtree part. For the intree part, $h_{max} = N+N/2+ \cdots +1 = 2N{-}1$, so the lower and upper bounds of the search region can be determined from Result (b), which are close to $(\log N)/4$ and $\log N$, respectively. Since there are only $(3{\cdot}\log N)/4$ candidate values in this search region, $\log\log N$ steps of a binary search can guarantee to find the optimal grain of parallel merge sorting. For problems such as evaluating numerical or logic expressions and finding the maximum (or minimum) value, all task times are identical. Result (b) predicts that the optimal grain is between $N/(2{\cdot}\log N)$ and $2N/\log N$.

To illustrate Result (c), we can determine the region for the optimal granularity of solving the parallel merge-sort of N elements. In this problem, the computational overhead in the intree is dominant, so only the part of the intree has to be considered in the scheduling. The lower bound of the search region is $(\log N)/16$, since $T_{np}(1)=N{\cdot}\log N$ and $h_{max}<2N$. If N is large enough, then $[d(s_k)+L(c_k)-2t_{npa}(k)]$ will be larger than 1.5N, hence, the upper bound of the search region is $2{\cdot}\log N$.

The bounds on optimal granularity when the non-preemptive CPS algorithm is used are usually weaker than those when the PCPS algorithm is used. Further, since kT^2 is not monotonically decreasing or increasing with k in nonpreemptive scheduling, an exhaustive search is required to find the optimal grain.

2.3. Asymptotic Optimal Granularity of Nonpreemptive Scheduling Outin Trees

To predict the optimal order-of-magnitude granularity in general, we briefly discuss the asymptotically optimal granularity of parallel outin-tree evaluations when they are scheduled using the nonpreemptive CPS algorithm.

Let C(n) be the overhead of a node in the intree, which has n leaves rooted by this node. C(n) represents the overhead of combining the results from its immediate predecessor nodes in the intree. Likewise, let D(n) be the overhead of a node in the outtree, which has n leaves rooted by this node. D(n) represents the overhead of decomposing the given node into its immediate successor nodes in the outtree. For an outin tree with N leaves, C(N) and D(N) represent the overheads of the exit and entry nodes, respectively. Let Θ be the set of functions of the same order.

For problems such as summing a set of numbers, finding the maximum of N numbers, and returning logical values to the main goal in evaluating logic programs, $C(n)=\Theta(1)$. In quicksort and merge sort, $C(n)+D(n)=\Theta(n)$.

**** All logarithms used in this chapter have base 2. The base is not specified explicity in the rest of this chapter.

The asymptotically optimal grain depends on the complexities of C(n) and D(n). The higher the order-of-magnitude complexity of C(n) and D(n) are, the larger the granularity is. When the order-of magnitude complexity of C(n) (and/or D(n)) is large, the time spent in the combining (and/or splitting) phase is dominating the time in the all-busy phase, and the performance gain in the all-busy phase with finer grains is negligible. In other words, a small granularity may result in under-utilization of processors.

To isolate the impact of the complexities C(n) and D(n) on the optimal granularity from the shape of the outin tree, we discuss the complete binary outin tree, and assume that, for all nodes in a level of the intree (resp. outtree) part, the order-of-magnitude complexities of C(n) (resp. D(n)) are identical. This assumption enables us to estimate T(1). The following result is stated without proof [53].

(d) Suppose that a nonpreemptive CPS algorithm is applied to evaluate an outin tree of N leaves by k processors. Assume that, for all nodes in a level of outin tree, the order-of-magnitude complexities of C(n) (and D(n)) are the same and that $t_i>t_j$, if task i is a predecessor (resp. successor) of task j in the part of the outtree (resp. intree). Then the order-of-magnitude $kT_k^2(N)$ is the minimum if $\Theta(T_{npa}(k(N))) = \Theta(T_{nps}(k(N))+T_{npc}(k(N)))$.

The above result shows that if the number of leaves of an outin tree is very large, then, to achieve the minimum $kT_p^2(k)$, the number of processors should be chosen such that the times required by the all-busy phase and the total times required by the other two phases are approximately equal.

Result (d) also shows the relationship between the processor utilization and kT^2. Let N_{sc} be the amount of task-time in the splitting and combining phases, and $T_{npa}(k) = [T_{nps}(k)+T_{npc}(k)]$. Then, for arbitrary outin tree computations, an asymptotically optimal granularity is achieved when

$$PU(k) = \frac{kT_{npa}(k)+N_{sc}}{2kT_{npa}(k)} \qquad (7)$$

Since $T_{npa}(k) \le N_{sc} \le (k{-}1)T_{npa}(k)$, it can be concluded that the corresponding processor utilization is between 0.5 and 1. In other words, when a problem is solved by a parallel divide-and-conquer algorithm and there are a large number of leaves in its outin-tree representation, to pursue more than 50% processor utilization will reduce the utilization-time efficiency.

Using Result (d), the asymptotically optimal granularities with respect to various C(n)+D(n) are summarized in Table 2. The results in Table 2 show that if C(n)+D(n) is $\Theta(n^p)$ for p>1, and that a large number of processors are used, then kT^2 must be poor regardless of the capacity of the interconnection network. In this case, the time needed to evaluate a subproblem will be increased quickly during the decomposition process in the outtree and the composition process in the intree. Hence, the root and exit nodes of the tree are obvious bottlenecks.

In contrast, if C(n) and D(n) are $\Theta(1)$, then the time needed to evaluate any subproblem is bounded by a constant, and the root and exit nodes will not be bottlenecks. Examples of this kind of problems include finding the maximum and evaluating an arithmetic expression. Here, a fine-grain architecture is appropriate, and a large speedup will be obtained by using a large number of processors. Tree-structured computer architectures [29,59] and virtual-tree computers [8] are good candidates in these applications.

In cases when C(n) equals either $\Theta(n)$ or $\Theta(\log^s n)$, $s\ge0$, the time needed to evaluate a subproblem is increased slowly during the decomposition process in

Table 2. Asymptotically optimal granularities in parallel processing of outin trees with respect to order-of-magnitude kT^2 (N is the number of leaves of the outin tree).

Complexity of $C(n)+D(n)$ $1 \le n \le N$	Optimal Granularity	Architectural Requirements
$\Theta(\log^s n)$, $s \ge 0$	$\Theta\left[\frac{N}{\log^{s+1} N}\right]$	A very large number of processors; tree or other efficient interconnection
$\Theta(n^r \log^s n)$, $0<r<1$, $s \ge 0$	$\Theta\left[\frac{N^{1-r}}{\log^s N}\right]$	A large number of processors; tree or other efficient interconnection
$\Theta(n \log^s n)$ $s \ge 0$	$\Theta(\log N)$	A small number of processors; loosely coupled; simple interconnection
$\Theta(n^p)$ $p>1$	$\Theta(1)$	Single or few processors; shared memory

the outtree and the composition process in the intree. A medium-grain architecture will be more cost-effective. For example, to sort 4000 elements by a parallel merge-sort algorithm, using ten to twelve processors will be a good choice.

In many problems, the order-of-magnitude complexities of $C(n)$ and $D(n)$ may be different. For example, for the quicksort algorithm, $C(n)=\Theta(1)$ and $D(n)=\Theta(n)$, that is, most of the computational overhead is spent in the decomposition phase, and the composition operation is trivial. In contrast, for the merge-sort algorithm, $C(n)=\Theta(n)$ and $D(n)=\Theta(1)$. In many logical and functional programs, the return operation is usually simple, i.e., $C(n)=\Theta(1)$, but the complexity of a subgoal or function call depends on the number of the parameters passed and the method of copying data. For these problems, the optimal grain can be determined by the part of the tree that has dominant overhead.

The shape of the AND-tree and its task-time distribution are also important factors to be considered. Let $T_p(1)/h_{max}$ be "average width" of an AND-tree. The optimal granularity is found to depend strongly on the average width. If the AND-tree is "wide," then the degree of parallelism is high and the granularity can be small. On the other hand, if the AND-tree is "narrow," then the degree of parallelism is low and the granularity is necessarily large. Here, the tree may have to be restructured to arrive at a different representation.

In many problems involving AND-trees, the trees are usually irregular, and the workloads may be data dependent. An important functional requirement is, therefore, the ability to dynamically distribute the workload in the architecture. For a computer architecture with a small granularity, an efficient interconnection network is needed. In a loosely coupled system with a coarse grain, an effective load balancing mechanism is necessary. Here, process communications may not be well overlapped with computations, and the corresponding task-times should include the communication overhead. As a result, the optimal number of processors may be less than the theoretical bounds predicted in this section.

3. MULTIPROCESSING OF BRANCH-AND-BOUND ALGORITHMS

A branch-and-bound (B&B) algorithm is a systematic search of an OR tree [47]. 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 are expanded under a best-first strategy provided that all lower bounds are distinct [47]. However, 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 lower-bound and dominance tests. For lower-bound tests, the incumbent, z, holds the value of the best solution that has been 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, and may be eliminated from further consideration.

The way in which P_0, the initial problem, is repeatedly decomposed into smaller subproblems can be represented as a finite rooted tree **B = (P, E)**, where **P** is a set of disjunctive subproblems, and E is a set of edges. The root of the tree is P_0. If a subproblem P_j is obtained from P_i by direct decomposition, then $(P_i, P_j) \in$ **E**. The **level number** of a node is the number of edges leading from the root to this node (the root is at Level 0). Let $f(P_i)$ be the value of the best solution obtained by evaluating all subproblems decomposable from P_i, P_j be the j'th subproblem directly decomposable from P_i, and k_i be the total number of such subproblems ($k_i = |\{(P_i,x): (P_i,x) \in E\}|$). Then f satisfies:

$$f(P_i) = \min_{j=1,\ldots,k_i} \{f(P_j)\} \tag{8}$$

The lower-bound test can be relaxed by defining an **allowance function**, $\varepsilon(z)$. Subproblems with lower bounds greater than $z-\varepsilon(z)$ are eliminated resulting in a suboptimal solution that will deviate from the optimal solution by at most $\varepsilon(z_O)$, where z_O is the value of the optimal solution [33]. An example of an allowance function is the relative error deviation, and a subproblem is terminated if its lower bound is greater than $z/(1+\varepsilon)$. The use of an allowance function is very effective in reducing the computational complexity of B&B 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 [74, 77].

Each of the four constituents of a serial B&B algorithm can be implemented by parallel processing: (i) **parallel selection of subproblems:** multiple subproblems with the smallest heuristic values can be selected for expansion; (ii) **parallel expansion of subproblems;** (iii) **parallel termination tests and update of the incumbent;** (iv) **parallel elimination tests:** these include the lower-bound and dominance tests.

We have studied the performance bounds of parallel B&B search [49,54,55] assuming (i) that only lower-bound tests are active, (ii) that there is a single shared memory, (iii) that no approximations are allowed, (iv) that the subproblems are expanded synchronously, (v) that the heuristic function is unambiguous, meaning

$$h(P_i)\neq h(P_j) \quad \text{if } P_i\neq P_j,\ P_i, P_j \in \mathbf{P} \tag{9}$$

and (vi) that the heuristic function is monotone, meaning

$$h(P_i)<h(P_j) \quad \text{if } P_j \text{ is a descendant of } P_i \tag{10}$$

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

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

$$\left\lceil \frac{T_b(1)-1}{k}+1 \right\rceil \le T_b(k) \le \left\lceil \frac{T_b(1)}{k}+\frac{k-1}{k}h \right\rceil \tag{11}$$

where h is the maximum number of levels in the B&B tree.

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

$$\left\lceil \frac{T_b(1)-1}{k}+1 \right\rceil \le T_d(k) \le \left\lceil \frac{T_d(1)}{k}+\frac{(k-1)\cdot(c+1)}{k}h \right\rceil \tag{12}$$

where c is the number of distinct incumbents obtained during the search. A similar equation can be also derived for parallel breadth-first searches.

(c) For a parallel breadth-first search,

$$\left\lceil \frac{T_r(1)-1}{k}+1 \right\rceil \le T_r(k) \le \left\lceil \frac{T_r(1)}{k}+\frac{k-1}{k}(c+h) \right\rceil \tag{13}$$

Eq's (11), (12) and (13) show that there is 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, while very accurate lower bounds will prune most unnecessary expansions. In both cases, the number of subproblems expanded by depth-first and best-first searches will not differ greatly, and a depth-first search is better as it requires less memory space (proportional to the height of the search tree). When the accuracy of the lower-bound function is moderate, a best-first search gives a better performance. 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 B&B algorithms. A subproblem is delegated to each processor, which reports to its parent processor when the evaluation is completed [18]. The limited degree of communication causes some processors to work on tasks that would be eliminated if a better interconnection network was designed. Moreover, implicit enumeration is wasteful. Imai et al. [34] and El-Dessouki and Huen [20] have investigated parallel B&B algorithms based on a general-purpose network architecture with limited memory space and slow interprocessor communication. Depth-first search was used due to memory limitations.

For problems that are more efficiently evaluated by a parallel best-first search, the architecture required is more complex. The problems here are the selection of subproblems with the minimum lower bounds and the management of the large memory space required. A number of search algorithms exist for performing tradeoff between space and time in order to obtain better search efficiency. These algorithms include Korf's Iterative Deepening Search Algorithm [38] and Real-Time A* Algorithm [39], which are sequential search algorithms. In the following section, we discuss the architecture of MANIP, which represents a design when search algorithms are executed in a multiprocessing system with memory limitation.

3.1. MANIP--Multiprocessor for Parallel Best-First Search with Lower-Bound Tests only

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 multi-stage selection network for selecting k elements from N elements requires $O(\log N\cdot\log k)$ time complexity and $O(N\cdot\log^2 k)$ hardware complexity [75]. A single-stage selection network may 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 [74]. Assuming that insertion is implemented in software, the time overhead in each iteration is $O(k\cdot\log N)$. In all these cases, selection represents a significant overhead of the system.

In addition to the high overhead of selection, it is known that the selection rule is based on a fallible lower-bound heuristic. These suggest that it may be more efficient not to follow the selection rule strictly. A **No-Wait Policy** is proposed here. Instead of waiting for one of the k subproblems with the smallest lower bounds, each processor should 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 [74], resulting in a speedup proportional to 0.63k. However, as expansion proceeds, the distribution may become non-random, and an interconnection network is needed 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 about 10% when the above scheme is used in place of 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 undirectional 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 of a CPU, a main memory, a slower secondary memory, and a secondary-memory controller. The expected completion time of the B&B algorithm on this model is taken as the performance measure.

A direct implementation involving an ordered list of pointers to the subproblems results in a 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 [80]. However, this approach is inflexible as the parameters of the control strategies are problem dependent. The inadequacies of these approaches are again due to the strict adherence to the selection rule. The No-Wait Policy may also be applied here. This has resulted in the design of a modified B&B algorithm [81-83].

In this modified algorithm, the range of possible lower bounds is partitioned into b disjoint regions (Figure 3). The subproblems in each region are maintained in a separate list. The top portion of each list resides in the main memory while 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 the subproblem in the main memory with the smallest lower bound for expansion. Since subproblems within a list are not sorted, the **lower-bound elimination rule** has to be modified. Suppose that the current incumbent lies in the range of List L, then 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.

The modified algorithm is identical to a depth-first search when one list is used, and a best-first search when infinitely many lists are used. In general, as the number of lists increases, the number of subproblems expanded decreases while 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, 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 that the modified B&B algorithm is very useful. 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) are needed for the best performance. The improvement in paging overhead over a direct implementation of the best-first search can exceed one hundred times.

The architecture of MANIP as depicted in Figure 3 reflects an implementation of the solutions to the above two issues [74,76,78]. It consists of five major components: 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. A ring network is adequate, although the selection process can be mapped onto any existing interconnected multi-computer system. Excess subproblems that cannot be stored in the memory controllers are stored in the secondary storage. 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

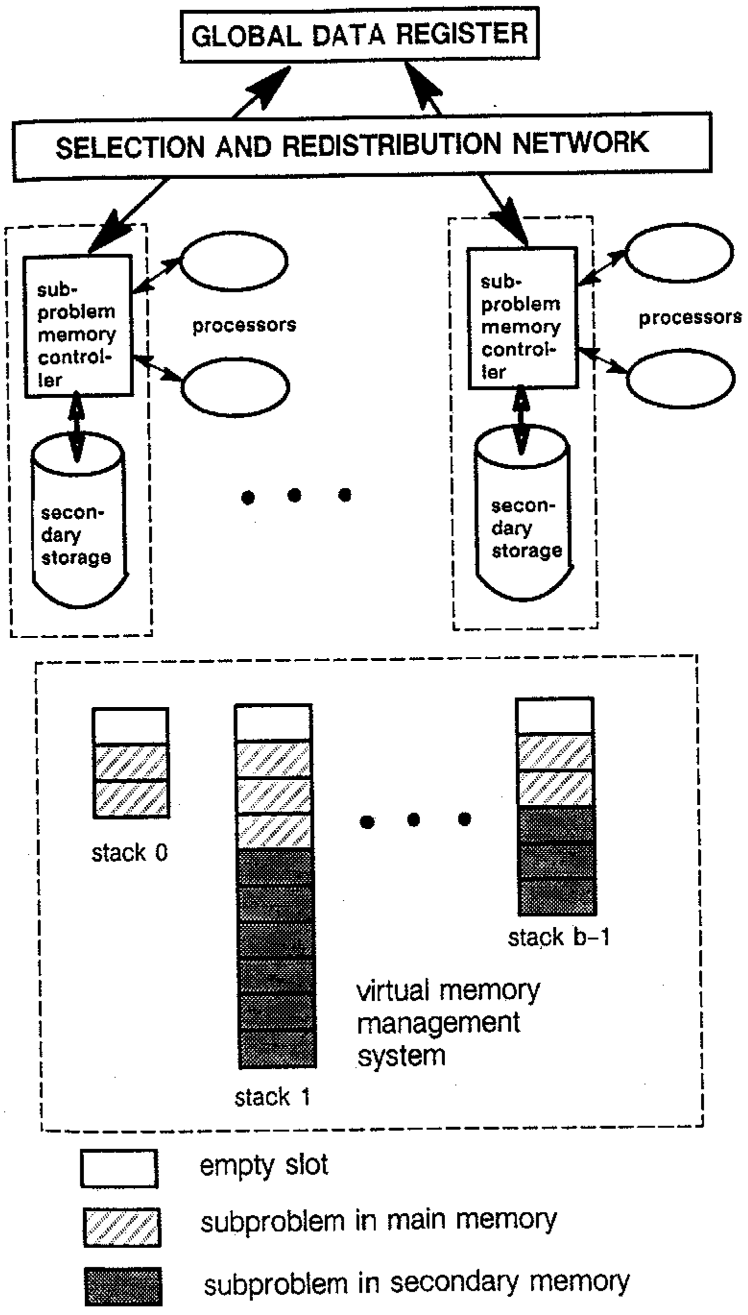

GLOBAL DATA REGISTER
SELECTION AND REDISTRIBUTION NETWORK
sub-problem memory control-ler
processors
secon-dary storage
sub-problem memory control-ler
processors
secon-dary storage
stack 0
stack 1
stack b−1
virtual memory management system
empty slot
subproblem in main memory
subproblem in secondary memory

Figure 3. Architecture and Logical Structure of MANIP

computers for partitioning subproblems and evaluating lower bounds. The global data register is a register 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.

Experience on MANIP and other previous studies [44, 62, 73] has shown that the functional requirements for efficient evaluation of B&B algorithms with lower-bound tests only are a loosely coupled interconnection of processors with load balancing capability, and a method of concurrent update and broadcast of the incumbent.

3.2. 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 have not been dominated so far. For the special case in which only lower-bound tests are used, N_d contains only one undominated solution node--the incumbent. In general, N_d could 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 if P_i or any nodes in N_d are dominated.

N_d can be stored in a bank of global data registers, if its size is small. However, centralized comparisons are inefficient when the size of N_d is large. N_d should then be partitioned into k subsets, N_d^0, ..., 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, and 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. $P_{i,j}$ will eventually return to Processor i if it has not been dominated by any node in N_d and will be inserted into N_d^i.

The functional requirements for implementing parallel dominance tests depends on the size of N_d and the structure of the dominance relation. When $|N_d|$ is small, unstructured dominance tests, in which a dominance relation can exist between any pair of nodes, can be implemented by broadcast busses or global registers. For structured dominance tests, it may be possible to partition the search tree and localize the dominance tests. This poses additional complexity on the system architecture and is investigated in Section 5. 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 has to be synchronized and overlapped with computations.

3.3. Anomalies of Parallelism in B&B Algorithms

Since it is possible to overlap the communication overheads with computations for the various search strategies, the speedup of B&B 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 B&B algorithm using k processors can be: (a) more than the number of iterations of the best serial algorithm--"**detrimental anomaly**," (b) less than one-k'th of the number of iterations of the best serial algorithm--"**acceleration anomaly**," or (c) less than the number of iterations of the best serial algorithm, but more than one-k'th of the number of iterations of the best serial algorithm--"**deceleration anomaly**." [43, 46, 49, 50, 52, 54, 55, 66, 67]

Let T(k,ε) be the number of iterations required for expanding a B&B tree to find the first optimal (or suboptimal) solution, where k is the number of processors and ε is the allowance function. Once the optimal solution is found, the time to drain the remaining subproblems from the active list is not accounted for here.

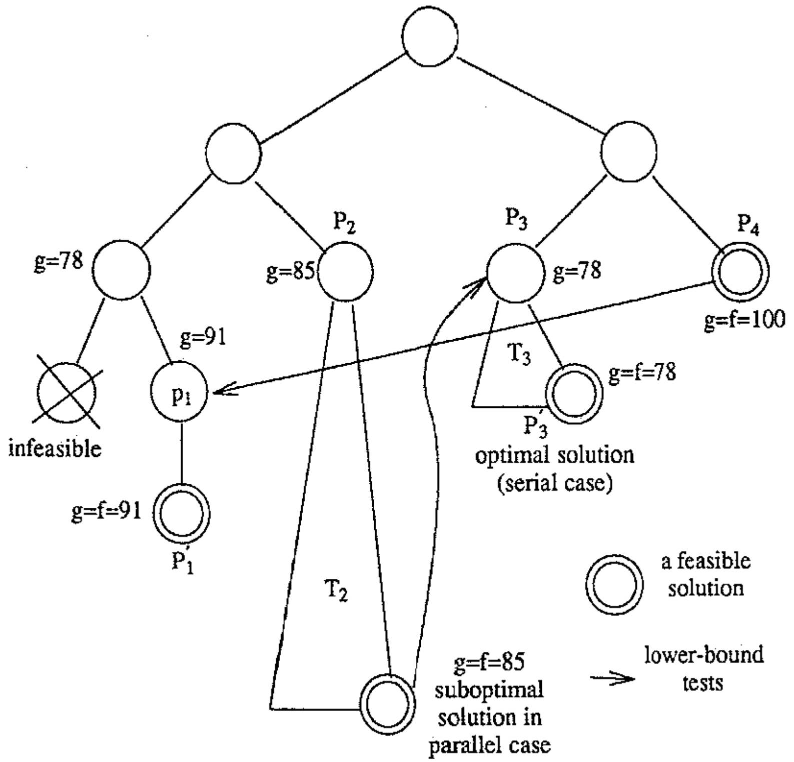


Figure 4. An example of a detrimental anomaly under a depth-first search with approximations (ε=0.1).

Figure 4 shows an example of a detrimental anomaly when approximations are allowed. In a serial depth-first search, Subtree T_2 is terminated owing to the lower-bound test of P_1': $f(P_1')/(1+\varepsilon) \le g(P_2)$, where $\varepsilon = 0.1$. In a parallel depth-first search with two processors, a feasible solution P_4 is found first, and Nodes P_1 and P_1' are terminated owing to the lower-bound test of P_4. Consequently, Subtree T_2

has to be expanded, which will eventually prune Subtree T_3. If the size of T_2 is much larger than that of T_3, the time taken to expand T_2 using two processors will be longer than that taken to expand T_3 using one processor. Strategies to handle these anomalies will be discussed in Section 3.5.

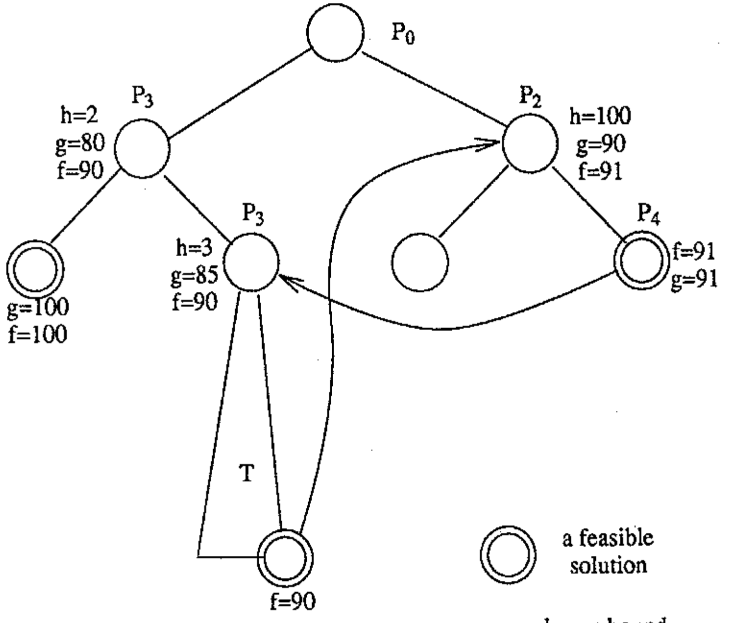


Figure 5. An example of an acceleration anomaly under a depth-first or best-first search with approximations ($\varepsilon = 0.1$).

Figure 5 shows an example of an acceleration anomaly when a depth-first search with approximations is used. When a single processor is used, Subtree T has to be expanded. However, when two processors are used, P_2 is expanded in the second iteration, and the feasible solution P_4 is found. Therefore, Node P_3 and Subtree T will be eliminated by lower-bound tests with P_4. If Subtree T is large, then the speedup of using two processors over one processor will be much greater than two. Acceleration anomalies will be discussed in Section 3.6.

Many examples to illustrate anomalies can be created for the various combinations of search strategies and allowance functions. However, the important consideration here is **not in knowing that anomalies exist, but in understanding why these anomalies occur.** Furthermore, it is desirable to preserve the acceleration anomalies and to avoid the detrimental anomalies. An important objective is to find the sufficient conditions to ensure that $T(k,\varepsilon) \le T(1,\varepsilon)$, as well as the necessary conditions for $T(k,\varepsilon) \le T(1,\varepsilon)/k$. The necessary conditions to eliminate detrimental anomalies and the sufficient conditions to preserve acceleration anomalies are not evaluated because they depend on the sequence of nodes expanded and the size of the resulting subtrees. Besides being impossible to enumerate due to the large number of possible combinations, these conditions are problem dependent and cannot be generalized to all problems.

3.4. Generalized Heuristic Function for Branch-and-Bound Search

Recall from Section 1 that the selection function uses heuristic values to define the order of node expansions. It was mentioned that breadth-first, depth-first, and best-first searches are special cases of heuristic searches. These searches are potentially anomalous when parallel expansions are allowed.

Consider the serial depth-first search. The subproblems are maintained in a last-in-first-out list, and the subproblem with the maximum level number is expanded first. When multiple subproblems have identical level numbers (or heuristic values), the node chosen by the selection function depends on the order of insertion into the stack. If the rightmost child of a parent node is inserted first, then the leftmost child will be the node inserted last and expanded first in the next iteration.

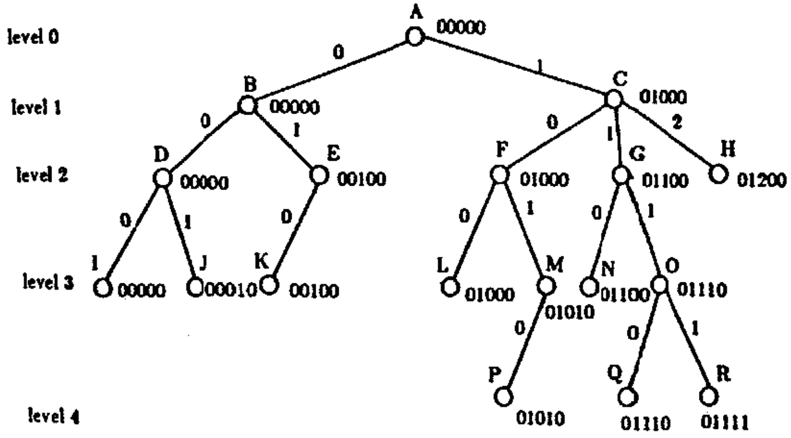


Figure 6. The path numbers of a tree.

In a parallel depth-first search, the mere extension of the serial algorithm may cause an anomalous behavior. For example, the order of expansion in a serial depth-first search for the tree in Figure 6 is A, B, D, I, J, E, etc. When two processors are used, Nodes B and C are decomposed to Nodes D, E, F, G, and H in the second iteration. Since these nodes have identical level numbers, the selection function can choose any two of these nodes in the next iteration. Suppose that they are inserted in the order E, D, H, G, and F. Then Nodes F and G will be selected and expanded in the third iteration. This may cause an unexpected behavior as compared to the serial case. A similar example can also be developed for the best-first search when the lower bounds of nodes are identical.

The ambiguous selection of nodes for expansion is exactly the reason for anomalies reported by Lai and Sahni [46]. In their proof of Theorem 1, which states that detrimental anomalies can always exist when a larger number of processors are used, the nodes selected for expansion are different when a different number of processors are used. This change of selection order in their Theorem 1 (and almost all their other theorems) is based on the implicit assumption that nodes have identical lower bounds. In this case, the lower bounds are not useful in guiding the selection of subproblems. To have an accurate comparison when different number of processors are used, a consistent selection strategy must be used.

To resolve the ambiguity of the selection of subproblems, distinct heuristic values must be defined for the nodes. In this chapter, a path number is proposed to uniquely identify a node. The **path number** of a node in a tree is a sequence of d+1 integers representing the path from the root to this node, where d is the maximum number of levels of the tree. The path number $E = e_0e_1e_2...e_d$ is defined recursively as follows. The root P_0 exists at Level 0 and has a path number of $E_0=000...0$. A node P_j on Level L, which is the j'th child (counting from the left) of P_i with path number $E_i=e_0e_1...e_{L-1}000...$, has path number $E_j=e_0e_1...e_{L-1}j00...$. As an example, the path numbers of all nodes in the tree of Figure 6 are shown outside the nodes.

To compare path numbers, the relations '>' and '=' must be defined. A path number $E_x=e_1^1e_2^1...$ is less than another path number $E_y=e_1^2e_2^2...$ $(E_x<E_y)$ if there exists $0 \leq j \leq d$ such that $e_i^1=e_i^2$, $0 \leq i<j$, and $e_j^1<e_j^2$. The path numbers are equal if $e_i^1=e_i^2$ for $0 \leq i \leq d$. For example, the path number 01000 is less than 01010. According to our definition of path numbers, nodes can have equal path numbers if they have the ancestor-descendant relationship. Since these nodes never coexist simultaneously in the active list of subproblems of a B&B algorithm, the subproblems in the active list always have unique path numbers.

The path number is now included in the heuristic function. The primary key is still the lower-bound value or the level number. The secondary or ternary key is the path number and is used to break ties in the primary key.

$$h(P_i)=\begin{cases} \text{(level number, path number)} & \text{breadth–first search} \\ \text{(path number)} & \text{depth–first search} \\ \text{(lower bound, level number, path number) or} & \\ \text{(lower bound, path number)} & \text{best–first search} \end{cases} \qquad (14)$$

where the level number, path number, and lower bound are defined for P_i. For a best-first search, nodes with identical lower bounds can be searched in a breadth-first or depth-first fashion.

The definition of the path number dictates that a partial order exists among successors of any given node in the search tree, and such an order is used consistently by both the sequential and parallel algorithms. Such a partial order implies that the same algorithm for decomposing a given node into successors is consistently used in both the sequential and parallel search. An attribute for defining this partial order may be needed if successors of a given node are generated in any random order. The lower-bound values can serve to define this partial order.

The heuristic functions defined above belong to a general class of heuristic functions satisfying Eq's (9) and (10). In general, an unambigous and monotonic heuristic function is not restricted to using path numbers. For example, the lower bound can be used as the secondary key and the path number as the ternary key in a breadth-first search.

3.5. Sufficient Conditions to Eliminate Detrimental Anomalies

3.5.1. Parallel B&B Algorithms with Lower-Bound Tests Only

In this section, we show that any heuristic search with an unambiguous heuristic function can guarantee that $T(k,\varepsilon) \leq T(1,\varepsilon)$ when only lower-bound tests with $\varepsilon=0$ are used. Assume that each subproblem is characterized by a lower-bound value, which is computed from a lower-bound function g. A **basic node** is the node with the smallest heuristic value in each iteration. Let Φ^1 and Φ^k be the sets of nodes expanded in the B&B tree using one and k processors, respectively. To show that $T(k,0) \leq T(1,0)$, it is necessary to prove (a) that at least one node belonging to Φ^1 is expanded in each iteration of the parallel search; and (b) that once all the nodes in Φ^1 are expanded or terminated, the parallel heuristic search must terminate. The proof requires the following property on basic nodes.

Lemma 1: Let P_i be a basic node, then for any node P_j such that $h(P_j)<h(P_i)$, P_j must be either expanded or terminated when P_i is expanded.

Proof: Suppose that in the current active list, U, $P_i \in U$ is a basic node. Assume that there exists a node P_j such that $h(P_j)<h(P_i)$ and that P_j has not been expanded or terminated when P_i is expanded. Since P_i has the minimum heuristic value among the active nodes in U, P_j must not be active at that time. That is, P_j is a descendant of some node P_k, $P_k \in U$, and $h(P_i)<h(P_k)$. By Eq. (10), $h(P_i)<h(P_k)<h(P_j)$, which contradicts the assumption that $h(P_j)<h(P_i)$. □

The following theorem proves that any unambiguous heuristic function satisfying Eq's (9) and (10) are sufficient to eliminate detrimental anomalies.

Theorem 1: Let $\varepsilon=0$, i.e., an exact optimal solution is sought. $T(k,0) \leq T(1,0)$ holds for any parallel heuristic search with a heuristic function satisfying Eq's (9) and (10).

Proof: The proof is by contradiction. Suppose there exists a basic node P_{i_1} in the parallel search such that $P_{i_1} \notin \Phi^1$ and that $P_{i_1} \in \Phi^k$ (see Figure 7). This means that either P_{i_1} or its an ancestor is terminated by a lower-bound test in the serial case. Hence there must exist a feasible solution $P_{i_2} \in \Phi^1$ such that $f(P_{i_2}) \leq g(P_{i_1})$ and that P_{i_2} has not been obtained when P_{i_1} is expanded in the parallel case. It implies that a proper ancestor $P_{i_3} \in \Phi^1$ of P_{i_2} exists in the serial case such that $h(P_{i_3})<h(P_{i_1})$, and that P_{i_2} is obtained before P_{i_1} and terminates P_{i_1}. Since P_{i_1} is a basic node in the parallel search, $h(P_{i_3})<h(P_{i_1})$, and P_{i_3} has not been expanded when P_{i_1} is expanded in the parallel case, P_{i_3} must be terminated according to Lemma 1. For the parallel search, there must exist a feasible solution $P_{i_4} \in \Phi^k$ such that $f(P_{i_4}) \leq g(P_{i_3})$, and that P_{i_4} has not been obtained when P_{i_3} is expanded in the serial case. Two cases are possible:

First, P_{i_4} is not generated when P_{i_3} is expanded in the serial case, i.e., a proper ancestor $P_{i_5} \in \Phi^k$ of P_{i_4} exists when P_{i_3} is active and $h(P_{i_5})>h(P_{i_3})$. According to the properties of lower-bound functions, we have $f(P_{i_4}) \leq g(P_{i_3}) \leq f(P_{i_3}) \leq f(P_{i_2}) \leq g(P_{i_1})$. Moreover, in the parallel case, P_{i_4} should be obtained before P_{i_1} is expanded (otherwise P_{i_3} would not be terminated by P_{i_4}).

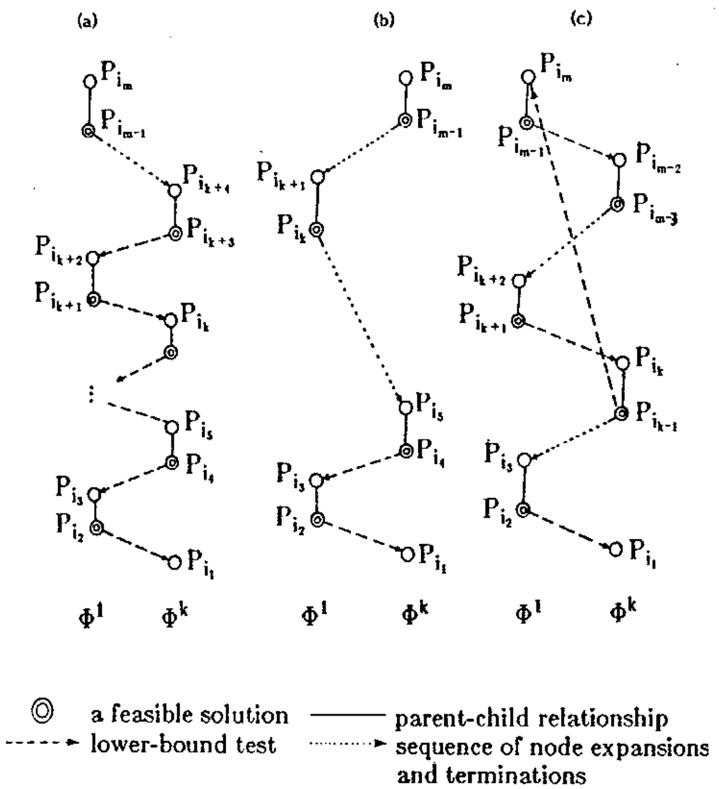


Figure 7. Proof of Theorem 1.

Hence P_{i_1} has to be terminated by P_{i_4} in the parallel algorithm, which contradicts the assumption that $P_{i_1} \in \Phi^k$.

Second, $h(P_{i_5})<h(P_{i_3})$, and P_{i_5}, as well as its descendant P_{i_4}, have been terminated in the serial case and not in the parallel case. We can then apply the above argument again to P_{i_5} and eventually obtain a sequence of nodes $P_{i_1}, P_{i_2}, ..., P_{i_m}$ as depicted in Figure 7, in which P_{i_m} is not terminated by any lower-bound test. There are three possibilities:

(a) The first node P_{i_m} occurs in the serial case (Figure 7a). Since $P_{i_{m-1}}$ is a feasible solution, we have: $g(P_{i_m}) \le f(P_{i_{m-1}}) \le g(P_{i_{m-2}}) \le f(P_{i_{m-2}}) \le g(P_{i_{m-3}}) \le \cdots \le f(P_{i_1}) \le g(P_{i_1})$. Further, since $h(P_{i_{k+2}})<h(P_{i_k})$ (otherwise, P_{i_k} could not have been terminated by $P_{i_{k+1}}$ in the serial case) and since $h(P_{i_{k+4}})<h(P_{i_{k+2}})$ (by the same argument as $h(P_{i_5})<h(P_{i_3})$), we have $h(P_{i_{k+4}})<h(P_{i_k})$. Repeating this, we get $h(P_{i_m})<h(P_{i_1})$. By Lemma 1, P_{i_m} must be expanded in the parallel case and terminates P_{i_1}, which contradicts that $P_{i_1} \in \Phi^k$.

(b) The first node P_{i_m} occurs in the parallel case (Figure 7b). Similar to the argument for P_{i_4} discussed before, we can explain that $P_{i_{m-1}}$ has been obtained when P_{i_1} is selected. Therefore, P_{i_1} must be terminated in the parallel case, which contradicts that $P_{i_1} \in \Phi^k$.

(c) There is a cycle of cutoffs such that $P_{i_{m-1}}$ L $P_{i_{m-2}}$, $P_{i_{m-3}}$ L $P_{i_{m-4}}$, ..., $P_{i_{k+1}}$ L P_{i_k}, and $P_{i_{k-1}}$ L P_{i_m} where **L** denotes a lower-bound-cutoff test (Figure 7c). By transitivity, we have $f(P_{i_m}) \le f(P_{i_{m-1}}) \le f(P_{i_{k-1}}) \le f(P_{i_m})$, which implies that $f(P_{i_m}) = f(P_{i_{m-1}}) = f(P_{i_{k-1}})$. The heuristic value of all nodes of the cycle are less than $h(P_{i_1})$, so a feasible solution has been obtained before P_{i_1} is selected. Thus P_{i_1} must be terminated in the parallel case, which contradicts that $P_{i_1} \in \Phi^k$.

So far, we have proved that at least one node in Φ^1 is expanded in each iteration of a parallel heuristic search. Since approximation is not allowed, the optimal-solution node cannot be eliminated by lower-bound tests. Hence during a parallel heuristic search, once all nodes in Φ^1 are either expanded or terminated, the optimal-solution node must be found. The remaining unexpanded nodes do not belong to Φ^1 because their lower bounds are greater than the optimal solution. The parallel heuristic search is thus completed at this time. □

The above theorem shows that detrimental anomalies can be avoided for depth-first, breadth-first, and best-first searches with ε=0 by augmenting the heuristic function with an unambiguous function. As a special case, for a best-first search in which all nodes have distinct lower bounds, the node with the smallest lower bound can always be selected from the priority queue. In this case, the path numbers do not have to be used, and no detrimental anomaly will occur.

3.5.2 Parallel B&B Algorithms with Approximations

When parallel approximate B&B algorithms are considered, Theorems 1 is no longer valid (see the example in Figure 4). The reason for the detrimental anomaly is that L, the lower-bound tests under approximation, are not transitive. That is, P_i L P_j and P_j L P_k do not imply P_i L P_k, since $f(P_i)/(1+\varepsilon) \le g(P_j)$ and $f(P_j)/(1+\varepsilon) \le g(P_k)$ implies $f(P_i)/(1+\varepsilon)^2 \le g(P_k)$ rather than $f(P_i)/(1+\varepsilon) \le g(P_k)$. Somewhat surprisingly, it is possible that Φ^1 and Φ^k are almost disjoint, and most of the nodes in Φ^1 are not expanded in the parallel case. The following theorem shows that detrimental anomalies can be avoided for a best-first search.

Theorem 2: $T(k,\varepsilon) \le T(1,\varepsilon)$, ε>0, holds for best-first searches if the heuristic function satisfies Eq. (9) and (10).

Proof: The key idea of this proof is to show that a detrimental anomaly cannot occur although transitivity of lower-bound tests is not valid here. Suppose that there exists a basic node P_{i_1} in Φ^k and not in Φ^1. There are two cases as described in the proof of Theorem 1 (see Figure 7).

First, assume that $h(P_{i_5}) > h(P_{i_3})$. Since the relations P_{i_4} L P_{i_3} and P_{i_2} L P_{i_1} exist, and P_{i_3} is selected before P_{i_1} in the serial case (for a best-first search), it is true that $g(P_{i_3}) \le g(P_{i_1})$. This implies that $f(P_{i_4})/(1+\varepsilon) \le g(P_{i_1})$. P_{i_4} can be shown to be available before P_{i_1} is expanded by the same argument as in the proof of Theorem 1. Hence P_{i_1} must be eliminated by P_{i_4} in the parallel case, a contradiction! (Note that only a best-first search has the unique property that maintains the transitivity of lower-bound tests under approximations.)

Second, assume that $h(P_{i_5}) < h(P_{i_3})$. The argument here is similar to the proof of Theorem 1 except that the lower-bound test to be used is $f(P_i)/(1+\varepsilon) \le g(P_j)$ and not $f(P_i) \le g(P_j)$. □

For depth-first searches, the condition of Theorem 2 is not sufficient. Figure 4 is an example of a detrimental anomaly caused by approximations. In breadth-first searches, since the sequences of the feasible solutions with respect to serial and parallel cases are the same, and the minimal feasible solution is selected as the new incumbent if more than one feasible solution are obtained in an iteration of a parallel search, approximations will not result in detrimental anomalies when the condition in Theorem 2 is satisfied.

In summary, detrimental anomalies can be prevented for a best-first or breadth-first search by using an unambiguous heuristic function, even when approximations are allowed. For a depth-first search, detrimental anomalies can only be prevented when an exact optimal solution is sought.

3.6. Necessary Conditions for Acceleration Anomalies

In this section, the necessary conditions for $T(k,0) < T(1,0)/k$ are developed. One condition is based on the complete consistency of heuristic functions. A heuristic function h is said to be **consistent** (resp. **completely consistent**) with the lower-bound function g if $h(P_i) < h(P_j)$ implies that $g(P_i) \le g(P_j)$ (resp. $g(P_i) < g(P_j)$) for all $P_i, P_j \in \mathbf{P}$. A heuristic function, h, is said to be **not completely consistent** with g if there exist two nodes P_i and P_j such that $h(P_i) > h(P_j)$ and $g(P_i) \le g(P_j)$. Note that if there are nodes with equal lower bounds, then the heuristic function for a serial best-first search is consistent, but not completely consistent, with the lower-bound function.

Theorem 3: The necessary condition for $T(k,0) < T(1,0)/k$ is that the heuristic function h is not completely consistent with g.

Proof: An acceleration anomaly does not exist if $\Phi^1 \subset \Phi^k$ because at least $\lceil |\Phi^1|/k \rceil$ iterations are needed to expand the nodes in Φ^1. Hence the proof is based on the assumption that a node $P_{i_1} \in \Phi^1$ exists and that $P_{i_1} \notin \Phi^k$. This means that P_{i_1} is terminated by a lower-bound test in the parallel case. That is, there is a feasible-solution node $P_{i_2} \in \Phi^k$ such that $f(P_{i_2}) \le g(P_{i_1})$, and P_{i_2} does not exist in the serial case when P_{i_1} is expanded. Let P_{i_3} be the immediate parent of P_{i_2}. Referring to Figure 8, two cases are possible.

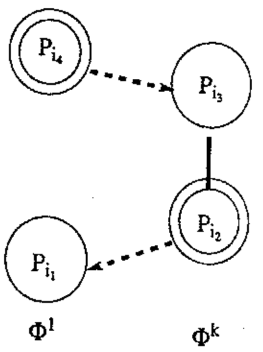


Figure 8. Proof of Theorem 3 (the notation used is similar to those of Figure 7).

First, $h(P_{i_3}) > h(P_{i_1})$. This means that P_{i_3} has not been generated when P_{i_1} is selected in the serial case. Since $P_{i_2}\,\mathbf{L}\,P_{i_1}$, thus $g(P_{i_3}) \le g(P_{i_2}) \le f(P_{i_2}) \le g(P_{i_1})$, which implies that h is not completely consistent with g. Second, $h(P_{i_3}) < h(P_{i_1})$. In order for P_{i_1} to exist in the serial case, $P_{i_4} \in \Phi^1$ must exist such that $P_{i_4}\,\mathbf{L}\,P_{i_3}$ and that $h(P_{i_4}) < h(P_{i_3})$. By the transitivity of lower-bound tests, $P_{i_4}\,\mathbf{L}\,P_{i_1}$. This contradict the assumption that $P_{i_1} \in \Phi^1$. □

The significance of Theorems 1 and 3 is in showing that acceleration anomalies may exist and that detrimental anomalies can be prevented for depth-first searches when no approximation is allowed and an unambiguous heuristic selection function is used. For a best-first search without approximation, detrimental anomalies can be prevented by using an unambiguous heuristic selection function; however, acceleration anomalies may exist when there are non-solution nodes of the B&B tree with lower bounds equal to the optimal solution value, since these nodes have heuristic values that are not completely consistent with their lower-bound values. For a breadth-first search without approximation, no acceleration anomaly occurs because at least one node belonging to Φ^1 must be expanded in each iteration of a parallel breadth-first search.

It is important to note that the conditions in Theorem 3 are not necessary when approximate solutions are sought. An example in Figure 5 shows the existence of an acceleration anomaly when ε=0.1 and h is competely consistent with g (since a best-first search is used and all lower bounds are distinct). In this case, the additional necessary condition is that the lower-bound test with approximation is inconsistent with h; that is, there exist P_i and P_j such that $h(P_i) > h(P_j)$ and that $P_i\,\mathbf{L}\,P_j$. Clearly, this condition is weak and can be satisfied easily.

In summary, acceleration anomalies may occur when one of the following conditions is true: (a) a depth-first search is used; (b) a best-first search is used, and some non-solution nodes have lower bounds equal to the optimal solution value; or (c) a suboptimal solution is sought.

Although our results have been proved with respect to a system in which all subproblems are maintained in a single list, they apply to a system in which multiple subproblem lists are used. When there are multiple lists, one for each processor, a subproblem with the minimum heuristic value is selected from each local list for decomposition. This subproblem may not belong to the global set of active subproblems with the minimum heuristic values, but the subproblem with the minimum heuristic value will always be expanded by one of the processors. Further, when multiple lists are used, it is not difficult to maintain a global incumbent in a global data register [74,76]. Hence the behavior of using multiple lists is analogous to that of a centralized list.

Due to space limitation, deceleration anomalies, anomalies due to dominance relations, and anomalies when k_1 and k_2, $1 \le k_1 < k_2$, processors are used are not discussed here. These results can be found elsewhere [49,51,55].

4. MULTIPROCESSING OF AND/OR-TREE SEARCH

Searching an AND/OR tree is more complex than that of an AND tree or an OR tree. An AND/OR-tree is searched in two phases. The first phase is a top-down expansion as in searching an OR tree, while the second phase is a bottom-up evaluation as in searching an AND tree. Due to the existence of both AND

and OR nodes, a good parallel algorithm should combine the features of AND- and OR-tree searches. A good selection strategy must be developed for selecting alternate OR nodes. The granularity of parallelism, similar to that of parallel divide-and-conquer algorithms, is an important issue to be solved. 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 may be performed if pruning information obtained from one processor is unavailable to the other processor. The extra work that must be carried out due to a lack of pruning information is defined as the **information-deficiency overhead**. Pruning information can be exchanged by messages or through a common memory. This increased communication overhead needed for pruning is defined as the **information-transfer overhead**. In general, a tradeoff exists between the information-deficiency and information-transfer overheads. A good parallel AND/OR-tree search should consider the tradeoffs between the merits of parallel processing and the communication overhead of obtaining the necessary pruning information. These tradeoffs will be discussed in this section.

4.1. 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, board positions resulting from MAX's moves are represented by OR nodes, while positions resulting from MIN's moves are represented by AND nodes. All nonterminal MAX nodes take the maximum score of their children, while all 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 [4].

A well known technique to improve the efficiency of a minimax search is the α-β **pruning** [64]. This technique uses two parameters, α and β, to define the search window. α carries the lower bound of the MAX nodes, while 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 (which is a MAX node) is less than α, and there already exists another MAX node with value equal to α (according to the definition of α). 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. 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. 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, and 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 [60]. In the **parallel aspiration search**, the α-β window is divided into non-overlapped sub-intervals, which are independently searched by multiple processors [5]. Baudet reported that the maximum expected speedup is around 5 or 6 regardless of the number of processors. 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 [37]. Acceleration anomalies may also occur when the number of processors is small, say 2 or 3.

Finkel and Fishburn have proposed a **tree-splitting algorithm** that maps a look-ahead tree onto a processor tree with the same interconnection structure [21]. The information-transfer overhead is small due to the close match between the communication requirements and the interconnections. However, this is a brute-force 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** [2], 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. Comparing this scheme with the tree-splitting scheme, it has better performance under the best-case ordering, but may be worse in the worst-case ordering. In the latter case, many nodes that are pruned in the tree-splitting scheme may be visited in the second phase.

Another approach is to use a best-first search, such as the SSS* algorithm [70]. 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 B&B procedure and have presented two parallel implementations of SSS* [42].

The previous approaches to parallel game-tree search have emphasized on the reduction of the information-transfer overhead, but have paid little attention to the information-deficiency overhead. In the next section, we will consider the information-deficiency overhead as illustrated by the scheduling of parallel logic programs.

4.2. Parallel Logic Programs

Logic programming is a programming methodology based on Horn-clause resolution [40]. An example of a high-level language for logic programming is PROLOG. Executing a logic program can be considered as the search of an AND/OR tree [13,40]. 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. More restricted processing models can be imposed such that the search can be considered as an AND-tree search [15,16,56] or as an OR-tree search.

Searching an AND/OR tree for a logic program is quite different from 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) solution(s) that satisfies 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). The selection of a node for evaluation is usually based on a fixed order such as the depth-first search. Lastly, a variable in a logic program can be bound to several values, and some subgoals may share a common variable.

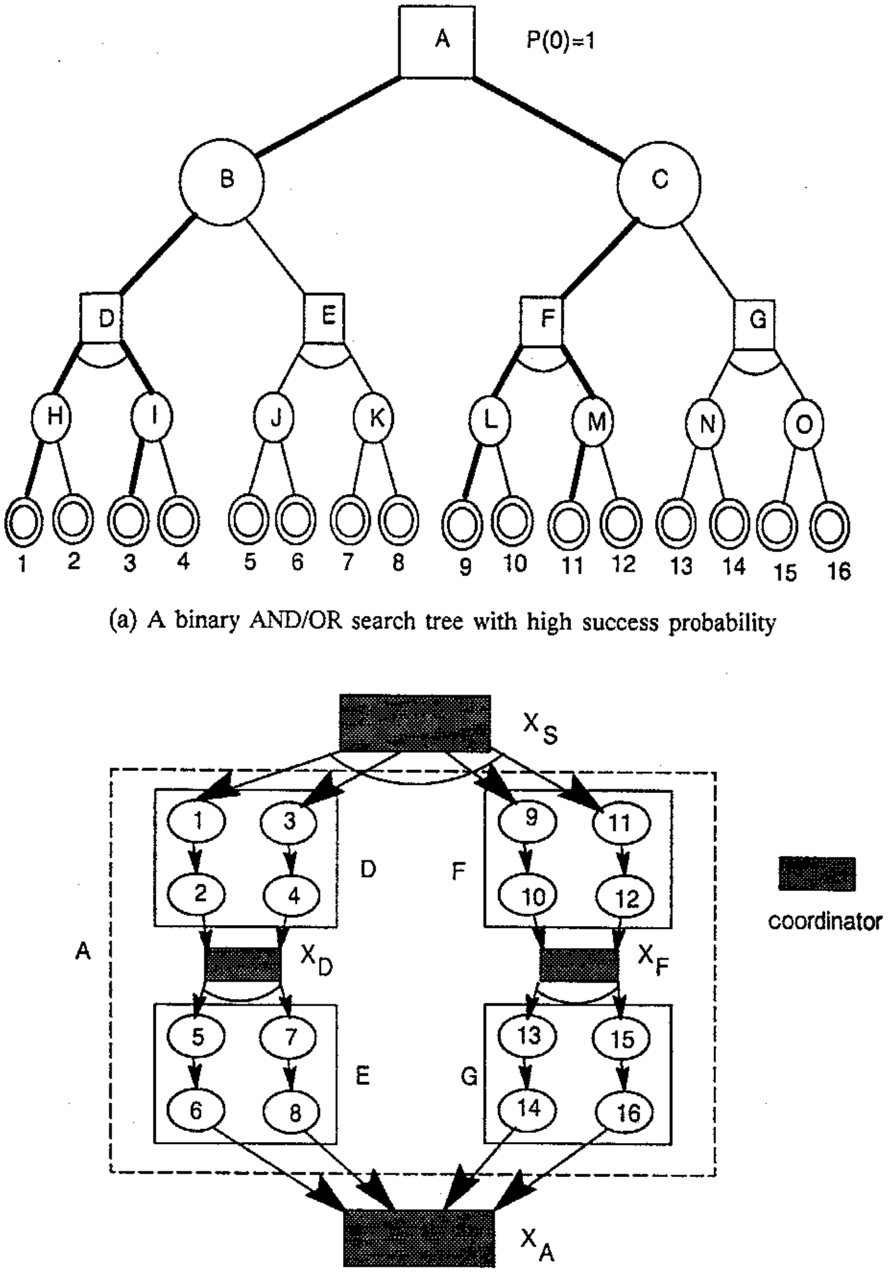
An efficient search method must involve pruning. Two kinds of pruning exist here. In an **AND-pruning**, if one of the children of an AND node is found to be false, then all remaining children of this AND node can be pruned. Likewise, in an **OR-pruning**, if one of the children of an OR node is found to be true, then all remaining children of this OR node can be pruned. It should be noted that OR-pruning is applicable only if the OR node does not share any variable with its siblings.

Much research strives for the parallel execution of logic programs. Conery and Kibler have classified four kinds of parallelism of logic programs: AND-parallelism, OR-parallelism, stream-parallelism, and search parallelism, and have investigated AND-parallelism [13]. Furukawa et al [22]. and Ciepielewski et al [9]. has discussed OR-parallelism, while Lindstrom et al. have addressed stream-parallelism and pipelined PROLOG processors [57]. However, very few studies have addressed the problem of processor assignment to reduce the information-deficiency overhead. Below, we present a scheduling algorithm that schedules the nodes to be searched according to the estimated probabilities of a terminal node being true, and does not distinguish between AND and OR parallelism.

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. An example is illustrated in Figure 9a. Once nodes 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 node (nodes 1, 2, 5, and 6 in Figure 9a). 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 9a, we see that, in a sequential search, if Node 1 fails, then Node 2 will be examined, otherwise Node 3 will be 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. According to this dependence information, a **fail-token-flow graph, G_f,** as depicted in Figure 9b for the tree in Figure 9a, can be drawn. A node (circle) in the graph will be 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, then a fail-token is sent to Node 2. If Node 2 is found to be false, then a fail-token is sent to Coordinator χ_D. At this time, any node concurrently searched in Block D can be terminated.

A simple parallel search strategy can be derived with the aid of 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



(a) A binary AND/OR search tree with high success probability

(b) The corresponding fail-token-flow graph G_f. (AND nodes are represented as squared nodes; OR nodes are represented as circular ones

Figure 9. Example illustrating AND/OR tree and fail-token-flow graph.

nodes in $\mathbf{G}_f$ (Figure 9b). 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. By examining Figure 9b closely, it is found 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. The scheduling of processors for searching the AND/OR tree can be done 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 to be found from the root (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. The above strategy is no longer suitable because a large number of nodes have to 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 fail-token is replaced by a success-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 non-terminal nodes in the tree. If the probability of leading to a solution tree for a non-terminal node can be refined as the search progresses, the corresponding token-flow subgraph can be re-derived. An idle processor can be scheduled according to the token-flow subgraph derived for the root of the given subtree.

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 to select the nodes for expansion, and be able to inform other processors to prematurely terminate their tasks when necessary. The architecture designed should support the dissemination of this information.

5. MULTIPROCESSING OF DYNAMIC PROGRAMMING ALGORITHMS

Dynamic Programming (DP) is a powerful optimization methodology that is applicable to a large number of areas including optimal control, industrial engineering and economics [6]. In general, DP is an approach that yields a transformation of the problem into a suitable form for optimization, but is not an algorithm for optimizing the objective function. A problem solvable by DP can be represented as a multistage problem, a divide-and-conquer problem, or an acyclic AND/OR-graph-search problem. Different computational approaches may be used depending on the formulation and representation. DP problems are discussed 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. According to 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 represent various optimization problems, while serial and nonserial optimization problems are problems solvable by the DP formulations. **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 said to be 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 10a. 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 a 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

$$f_1(i) = \min_j \, [c_{i,j} + f_1(j)] \tag{15}$$

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

From Eq. (15), $f(C_1)$, the minimum cost from C_1 to T, is

$$f(C_1) = \min \{c_{1,1}+d_{1,1},\ c_{1,2}+d_{2,1},\ c_{1,3}+d_{3,1}\} \tag{16}$$

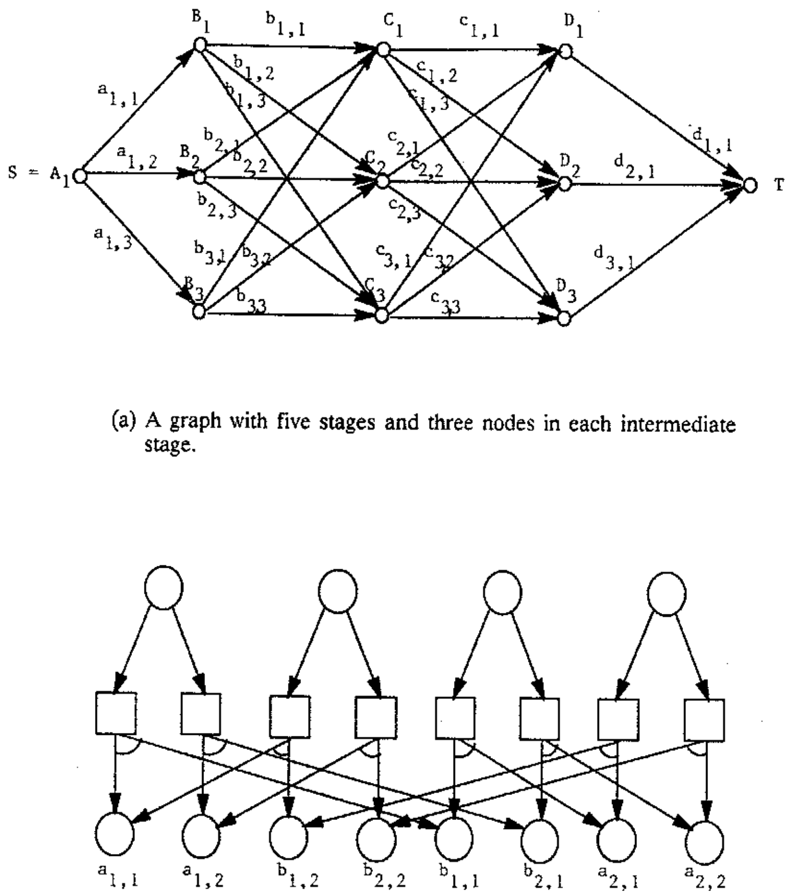
Eq. (16) can be interprered as an inner-product operation with respect to addition and minimization. If we define matrix multiplication in terms of a closed semiring (R, MIN, +, $+\infty$, 0), in which 'MIN' corresponds to addition and '+' corresponds to multiplication in conventional matrix multiplications [1], Eq. (16) becomes $\mathbf{f(C) = C \cdot D}$, where **C** is a cost matrix and **D** is a cost vector. It is easy to see that the search of the shortest path in a multistage graph with a forward monadic DP formulation is equivalent to multiplying a string of matrices, i.e., $\mathbf{A \cdot (B \cdot (C \cdot D))}$.

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

$$f_2(i,j) = \min_k \, [f_2(i,k) + f_2(k,j)] \tag{17}$$

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. Operations involving computations of a cost function, such as summations, are represented as AND nodes. An AND/OR graph for reducing the search of the shortest path in a three-stage graph with two nodes in each stage is shown in Figure 10b. 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 [26].



(a) A graph with five stages and three nodes in each intermediate stage.

(b) An AND/OR-graph representation of the reduction in finding an optimal path in a three-stage graph. AND nodes are represented as squared nodes and indicate summations; OR nodes are represented as circular nodes and indicated comparisons.

Figure 10. Example illustrating the transformation of a dynamic programming problem into an AND/OR-tree search. The problem is to find $\min_{i,j,k \in \{1,2\}} \{a_{i,j} + b_{j,k}\}$.

A nonserial DP problem can be represented in a monadic or polyadic form [7]. A monadic-nonserial formulation is an extension of Eq. (15) 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 solve DP problems. Guibas, Kung and Thompson have proposed a VLSI algorithm for solving the optimal parenthesization problem [28]. Linear pipelines for this problem have been proposed recently [72]. Clarke and Dyer have designed a systolic array for curve and line detection in terms of a nonserial formulation [10]. However, these designs were directed towards the implementation of a few special cases of DP formulations.

Architectures to support serial DP problems depend on the formulation. First, if the problem is represented in a polyadic form and considered as a divide-and-conquer problem, the architecture discussed in Section 2 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 and can be evaluated efficiently by a systolic array [79]. 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 [79]. Lastly, the problem may be represented in a monadic form and solved by a pipelining approach. This approach is suitable when many alternative partial solutions have to be compared.

There are not many architectures for solving nonserial DP problems directly. In an AND/OR-graph representation of nonserial problems, edges may connect nodes in any two arbitrary levels. These graphs may 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 to solve it by approaches developed for serial problems [79]. 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-serial representation as an AND/OR graph, the dependence can be removed by replacing each edge that connects nodes not in adjacent levels by multiple edges that connect nodes in adjacent levels. This approach has been used in designing a systolic array for finding the optimal binary search tree.

6. CONCLUSION

Research in problem solving is usually aimed at developing better algorithms. Unnecessary combinatorial searches should be avoided as much as possible because they do not contribute to the quality of the solutions. This is seen in the efforts in designing optimal algorithms and in understanding the reasoning process in artificial intelligence. However, searching becomes inevitable when a good algorithm has been developed, and is an essential part in many applications.

In this chapter, the limitations of multiprocessing in solving combinatorial searches have been investigated. The suitability of multiprocessing depends on the problem complexity, the problem representation, and the corresponding search algorithms. The problem complexity should be low enough, so the problem is solvable by a serial computer. The problem representations are very important because they are related to the search algorithms. However, the question of deciding the representation leading to an efficient search is still open for many problems. Moreover, efficient architectures to evaluate various search algorithms are different. Functional requirements for a given search algorithm are, therefore, developed in this chapter. These allow efficient mapping of a search algorithm on a general-purpose multiprocessor and the development of special-purpose processors for searching.

Table 3. Functional requirements of different paradigms of search algorithms. (Note: The magnitudes of large and small granularities in different algorithms are different. 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 tests 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

Three paradigms of problem representations are presented, and the parallel search algorithm in each class are studied. Functional requirements of search algorithms in each class are shown in Table 3. These requirements are very general and must be tailored to the special features of a given problem. We have not attempted to enumerate all possible cases in this chapter, but have illustrated the

different approaches generously with examples. These guidelines and examples can aid the designers to select the appropriate multiprocessing system in solving combinatorial search problems.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] S. G. Akl, D. T. Barnard, and R. J. Doran, "Design, Analysis and Implementation of a Parallel Tree Search Algorithm," *Trans. on Pattern Analysis and Machine Intelligence*, vol. PAMI-4, no. 2, pp. 192-203, IEEE, March 1982.

[3] J. Backus, "Can Programming be Liberated from the von Neumann Style? A Functional Style and Algebra of Programs," *Comm. of the ACM*, vol. 21, no. 8, pp. 613-641, 1978.

[4] A. Barr and E. A. Feigenbaum, *The Handbook of Artificial Intelligence*, vol. 1, 2, and 3, William Kaufmann, Los Altos, CA, 1981, 1982.

[5] G. Baudet, *The Design and Analysis of Algorithms for Asynchronous Multiprocessors*, Ph.D dissertation, Dept. Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., Apr. 1978.

[6] R. Bellman and S. Dreyfus, *Applied Dynamic Programming*, Princeton University Press, Princeton, NJ, 1962.

[7] U. Bertele and F. Brioschi, *Nonserial Dynamic Programming*, Academic Press, New York, NY, 1972.

[8] F. M. Burton and M. M. Huntbach, "Virtual Tree Machines," *IEEE Trans. on Computers*, vol. C-33, no. 3, pp. 278-280, 1984.

[9] A. Ciepielewski and S. Haridi, "Execution of Bagof on the OR-Parallel Token Machine," *Proc. Int'l Conf. Fifth Generation Computer Systems*, pp. 551-560, ICOT and North-Holland, 1984.

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

[11] E. G. Coffman, Jr. and R. H. Graham, "Optimal Scheduling for Two Processors Systems," *Acta Informatica*, vol. 1, no. 3, pp. 200-213, 1972.

[12] E. G. Coffman, Jr. (ed.), *Computer and Job-Shop Scheduling Theory*, Wiley, New York, NY, 1976.

[13] J. S. Conery and D. F. Kibler, "AND Parallelism and Nondeterminism in Logic Programs," *New Generation Computing*, vol. 3, no. 1, pp. 43-70, OHMSHA Ltd. and Springer-Verlag, 1985.

[14] R. W. Conway, W. L. Maxwell, and L. W. Miller, *Theory of Scheduling*, Addison-Wesley, Reading, MA, 1967.

[15] D. DeGroot, "Restricted AND-Parallelism," *Proc. Int'l Conf. on Fifth Generation Computers*, pp. 471-478, ICOT and North-Holland, Nov. 1984.

[16] D. DeGroot, "Restricted AND-Parallelism and Side-Effects in Logic Programming," in *Supercomputers and AI Machines*, ed. K. Hwang and D. DeGroot, McGraw-Hill, 1988.

[17] J. B. Dennis, "Data Flow Supercomputers," *Computer*, vol. 13, no. 11, pp. 48-56, IEEE, Nov. 1980.

[18] B. C. Desai, "The BPU: A Staged Parallel Processing System to Solve the Zero-One Problem," *Proc. Int'l Computer Symposium*, pp. 802-817, Taipei, Taiwan, Dec. 1978.

[19] D. Dolev and M. Warmuth, "Profile Scheduling of Opposing Forests and Level Orders," *SIAM J. of Algorithm and Discrete Mathematics*, vol. 6, no. 4, pp. 665-687, Oct. 1985.

[20] O. I. El-Dessouki and W. H. Huen, "Distributed Enumeration on Between Computers," *Trans. on Computers*, vol. C-29, no. 9, pp. 818-825, IEEE, Sept. 1980.

[21] R. A. Finkel and J. P. Fishburn, "Parallelism in Alpha-Beta Search," *Artificial Intelligence*, vol. 19, no. 1, pp. 89-106, North-Holland, 1982.

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

[23] M. R. Garey and D. S. Johnson, "Scheduling Tasks with Nonuniform Deadlines on Two Processors," *J. of ACM*, vol. 23, no. 3, pp. 461-467, 1976.

[24] M. R. Garey and D. S. Johnson, *Computers and Intractability*, Freeman, San Francisco, 1979.

[25] M. R. Garey, D. S. Johnson, R. E. Tarjan, and M. Yannakakis, "Scheduling Opposing Forests," *SIAM J. of Algorithm and Discrete Mathematics*, vol. 4, no. 1, pp. 72-93, March 1983.

[26] S. Gensi, U. Montanari, and A. Martelli, "Dynamic Programming as Graph Searching: An Algebraic Approach," *J. of the ACM*, vol. 28, no. 4, pp. 737-751, 1981.

[27] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. N. Kan, "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey," *Ann. Discrete Mathematics*, vol. 5, pp. 287-326, 1979.

[28] L. J. Guibas, H. T. Kung, and C. D. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms," *Proc. Caltech Conf. on VLSI*, pp. 509-525, Caltech, Pasadena, CA, 1979.

[29] J. A. Harris and D. R. Smith, "Simulation Experiments of a Tree Organized Multicomputer," *Proc. 6th Annual Symp. on Computer Architecture*, pp. 83-89, IEEE/ACM, April 1979.

[30] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, 1978.

[31] E. Horowitz and A. Zorat, "Divide-and-Conquer for Parallel Processing," *Trans. on Computers*, vol. C-32, no. 6, pp. 582-585, IEEE, June 1983.

[32] T. C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, vol. 9, no. 6, pp. 841-848, ORSA, 1961.

[33] T. Ibaraki, "Computational Efficiency of Approximate Branch-and-Bound Algorithms," *Math. of Oper. Research*, vol. 1, no. 3, pp. 287-298, Inst. of Management Sciences, 1976.

[34] M. Imai, T. Fukumara, and Y. Yoshida, "A Parallelized Branch-and-Bound Algorithm: Implementation and Efficiency," *Sys. Comput. Controls*, vol. 10, No. 3, pp. 62-70, 1979.

[35] M. Kaufman, "An Almost-Optimal Algorithm for the Assembly Line Scheduling Problem," *Trans. on Computers*, vol. C-23, no. 11, pp. 1169-1174, IEEE, 1974.

[36] R. M. Keller, F. C. H. Lin, and J. Tanaka, "Rediflow Multiprocessing," *Proc. COMPCON Spring*, pp. 410-417, IEEE, 1984.

[37] D. E. Knuth and R. W. Moore, "An Analysis of Alpha-Beta Pruning," *Artificial Intelligence*, vol. 6, no. 4, pp. 293-326, North-Holland, 1975.

[38] R. E. Korf, "Depth-First Iterative Deepening: An Optimal Admissible Tree Search," *Artificial Intelligence*, vol. 27, pp. 97-109, North-Holland, 1985.

[39] R. E. Korf, "Real-Time Heuristic Search: First Results," *proceedings of the Sixth National Conference on Artificial Intelligence*, pp. 133-8, Seattle, Washington, June, 1987.

[40] R. Kowalski, *Logic for Problem Solving*, North-Holland, 1979.

[41] V. Kumar and L. N. Kanal, "A General Branch and Bound Formulation for Understanding and Synthesizing And/Or Tree Search Procedures," *Artificial Intelligence*, vol. 21, no. 1-2, pp. 179-198, North-Holland, 1983.

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

[43] V. Kumar and V. N. Rao, "Parallel Depth-First Search on Multiprocessors Part II: Analysis," *Int'l Journal of Parallel Programming*, vol. 16, no. 6, 1987.

[44] V. Kumar, V. N. Rao, and K. Ramesh, "Parallel Depth-First Search on the Ring Architecture," *Proc. Int'l Conf. on Parallel Processing*, St. Charles, IL, Aug. 1988.

[45] M. Kunde, "Nonpreemptive LP-Scheduling on Homogeneous Multiprocessor Systems," *SIAM J. of Computing*, vol. 10, no. 1, pp. 151-173, Feb. 1981.

[46] T. H. Lai and S. Sahni, "Anomalies in Parallel Branch-and-Bound Algorithms," *Comm. of the ACM*, vol. 27, no. 6, pp. 594-602, June 1984.

[47] E. L. Lawler and D. W. Wood, "Branch and Bound Methods: A Survey," *Operations Research*, vol. 14, pp. 699-719, ORSA, 1966.

[48] J. K. Lenstra, A. Rinnooy Kan, and P. Brucker, "Complexity of Machine Scheduling Problems," *Proc. Discrete Mathematics*, pp. 343-362, North-Holland, 1977.

[49] G. J. Li and B. W. Wah, "Computational Efficiency of Parallel Approximate Branch-and-Bound Algorithms," *Proc. Int'l Conf. on Parallel Processing*, pp. 473-480, IEEE, Aug. 1984.

[50] G. J. Li and B. W. Wah, "How to Cope with Anomalies in Parallel Approximate Branch-and-Bound Algorithms," *Proc. National Conf. on Artificial Intelligence*, pp. 212-215, AAAI, Aug. 1984.

[51] G. J. Li and B. W. Wah, *Computational Efficiency of Parallel Approximate Branch-and-Bound Algorithms*, Tech. Rep. TR-EE-84-6, Purdue University, West Lafayette, IN, March 1984.

[52] G. J. Li, *Parallel Processing of Combinatorial Search Problems*, Ph.D. Thesis, School of Electrical Engineering, Purdue University, West Lafayette, IN, Dec. 1985.

[53] G. J. Li and B. W. Wah, "Optimal Granularity of Parallel Evaluation of AND-Trees," *Proc. Fall Joint Computer Conf.*, pp. 297-306, ACM/IEEE, Nov. 1986.

[54] G.-J. Li and B. W. Wah, "Coping with Anomalies in Parallel Branch-and-Bound Algorithms," *Trans. on Computers*, vol. C-34, no. 6, pp. 568-573, IEEE, June 1986.

[55] G. J. Li and B. W. Wah, "Computational Efficiency of Combinatorial OR-Tree Searches," *Trans. on Software Engineering*, IEEE, (accepted to appear) 1989.

[56] Y.-J. Lin and V. Kumar, "An Execution Model for Exploiting AND-Parallelism in Logic Programs," *New Generation Computing*, vol. 5, pp. 393-425, OHMSHA, Ltd. and Springer-Verlag, 1988.

[57] G. Lindstrom and P. Panangaden, "Stream-Based Execution of Logic Programs," *Proc. Int'l Symp. on Logic Programming*, pp. 168-176, IEEE, Feb. 1984.

[58] E. L. Lloyd, "Critical Path Scheduling with Resource and Processor Constrains," *J. of ACM*, vol. 29, no. 3, pp. 781-811, 1982.

[59] G. Mago, "Making Parallel Computation Simple: The FFP Machine," *Proc. COMPCON Spring*, pp. 424-428, IEEE, 1985.

[60] T. A. Marsland and M. Campbell, "Parallel Search of Strongly Ordered Game Trees," *Computing Surveys*, vol. 14, no. 4, pp. 533-551, ACM, Dec. 1982.

[61] A. Martelli and U. Montanari, "Additive AND/OR Graphs," *Proc. Int'l Joint Conf. on AI*, IJCAI, 1973.

[62] B. Monien and O. Vornberger, "The Ring Machine," *Computers and Artificial Intelligence*, vol. 3, 1987.

[63] R. Muntz and E. Coffman, Jr., "Preemptive Scheduling of Real-Time Task on Multiprocessor Systems," *J. of the ACM*, vol. 17, no. 2, pp. 324-338, April 1970.

[64] N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga, 1980.

[65] F. J. Peters, "Tree Machine and Divide-and-Conquer Algorithms," *Lecture Notes CS 111 (CONPAR81)*, pp. 25-35, Springer-Verlag, 1981.

[66] V. N. Rao and V. Kumar, "Parallel Depth-First Search on Multiprocessors Part I: Implementation," *Int'l Journal of Parallel Programming*, vol. 16, no. 6, 1987.

[67] V. N. Rao and V. Kumar, Superlinear Speedup in Parallel State-Space Search, Tech. Rep. AI88-80, Artificial Intelligence Laboratory, University of Texas, Austin, TX, 1988.

[68] S. Schindler, "On Optimal Scheduling for Multiprocessor Systems," *Proc. Princeton Conf. on Information Science and Systems*, pp. 219-223, 1972.

[69] A. C. Shaw, *The Logical Design of Operating System*, Prentice-Hall, 1974.

[70] G. Stockman, "A Minimax Algorithm Better Than Alpha-Beta?," *Artificial Intelligence*, vol. 12, pp. 179-196, North-Holland, 1979.

[71] J. D. Ullman, "NP-Complete Scheduling Problems," *J. of Computer and System Sciences*, vol. 10, pp. 384-393, 1975.

[72] P. J. Varman and I. V. Ramakrishnan, "Dynamic Programming and Transitive Closure on Linear Pipelines," *Proc. Int'l Conf. on Parallel Processing*, pp. 359-364, IEEE, 1984.

[73] O. Vornberger, *Implementing Branch-and-Bound in a Ring of Processors*, Tech. Rep. 29, University of Paderborn, FRG, 1986.

[74] B. W. Wah and Y. W. Ma, "MANIP--A Multicomputer Architecture for Solving Combinatorial Extremum Problems," *Trans. on Computers*, vol. C-33, no. 5, pp. 377-390, IEEE, May 1984.

[75] B. W. Wah and K. L. Chen, "A Partitioning Approach to the Design of Selection Networks," *Trans. on Computers*, vol. C-33, no. 3, pp. 261-268, IEEE, March 1984.

[76] B. W. Wah, G. J. Li, and C. F. Yu, "The Status of MANIP — A Multi-Computer Architecture for Solving Combinatorial Extremum-Search Problems," *Proc. 11'th Int'l Symp. on Computer Architecture*, pp. 56-63, ACM/IEEE, June 1984.

[77] B. W. Wah and C. F. Yu, "Stochastic Modeling of Branch-and-Bound Algorithms with Best-First Search," *Trans. on Software Engineering*, vol. SE-11, no. 9, pp. 922-934, IEEE, Sept. 1985.

[78] B. W. Wah, G. J. Li, and C. F. Yu, "Multiprocessing of Combinatorial Search Problems," *IEEE Computer*, vol. 18, no. 6, pp. 93-108, June 1985.

[79] B. W. Wah and G.-J. Li, "Systolic Processing for Dynamic Programming Problems," *Circuits, Systems, and Signal Processing*, vol. 7, no. 2, pp. 119-149, Springer-Verlag, 1988.

[80] C. F. Yu and B. W. Wah, "Virtual-Memory Support for Branch-and-Bound Algorithms," *Proc. Computer Software and Applications Conf.*, pp. 618-626, IEEE, Nov. 1983.

[81] C. F. Yu and B. W. Wah, "Efficient Branch-and-Bound Algorithms on a Two-Level Memory System," *Proc. Computer Software and Applications Conf.*, pp. 504-514, IEEE, Nov. 1984.

[82] C. F. Yu, *Efficient Combinatorial Search Algorithms*, Ph.D. Thesis, School of Electrical Engineering, Purdue University, West Lafayette, IN, Dec. 1986.

[83] C. F. Yu and B. W. Wah, "Efficient Branch-and-Bound Algorithms on a Two-Level Memory System," *Trans. on Software Engineering*, vol. SE-14, no. 9, pp. 1342-1356, IEEE, Sept. 1988.