

# Processor Scheduling Algorithms for Constraint-Satisfaction Search Problems

K. S. Natarajan  
Vivek Sarkar

IBM Research  
T. J. Watson Research Center  
Yorktown Heights, NY 10598

## Abstract

Constraint-satisfaction problems arise frequently in Artificial Intelligence and engineering design applications. These problems are computationally intensive and would significantly benefit from speedup through parallel processing. In this paper, we investigate parallelizations of the Forward-Checker algorithm, which is known to be an efficient sequential algorithm for constraint-satisfaction problems. We present two parallel algorithms -- the Threshold Depth-First Priority (TDFP) and the Breadth-First List Scheduling (BFLS) algorithms. Simulation results show that both algorithms are suitable for solving constraint-satisfaction problems in parallel, and yield near-linear speedup even beyond 100 processors. The best choice of algorithm depends on the amount of imbalance in the search problem and the overhead in the target multiprocessor.

## 1. Introduction

A constraint-satisfaction problem typically requires finding values for a set of variables, subject to an arbitrary set of constraints. For example, consider the problem of packing items into boxes, with the constraint that some pairs of items cannot be packed together (perhaps to avoid forming an explosive mixture). An acceptable solution to the packing problem is an assignment of items to boxes such that all constraints on pairs of items are satisfied. In a number of applications, one is interested in generating all acceptable solutions to a constraint-satisfaction problem. Many Artificial Intelligence and combinatorial search applications can be formulated as constraint-satisfaction problems.

A backtrack-search algorithm [GOL65] can be used to enumerate all solutions to a constraint-satisfaction problem. Previous work has primarily focussed on methods for improving the efficiency of backtrack search algorithms. Various authors have developed methods for improving search efficiency [BIT75] [HAR80] [NUD88]. The Iterative Deepening A\* (IDA\*) algorithm [KOR85] has been applied to find optimal cost solutions in AI applications modeled as state-space searches. Parallel execution methods for the IDA\* algorithm are described in [RAO87].

Most constraint-satisfaction problems are NP-complete and the algorithms used to solve them have worst-case exponential execution times. Using multiple processors to solve such problems cannot significantly improve the worst-case performance [KAS86]. Nevertheless, since a large number of practical applications are naturally formulated as constraint-satisfaction problems, it is important to speed up their solution using one or both of the following approaches:

1. Develop efficient sequential algorithms for the average case.
2. Use multiple processors to speed up their execution by parallel processing.

In this paper, we are interested in parallelizations of the Forward-Checker algorithm [HAR80] which has been shown to be one of the most efficient sequential algorithms for constraint-satisfaction problems. An efficient parallelization of the Forward-Checker algorithm, rather than the backtrack algorithm, gives us a true speedup over the sequential execution time.

We present two parallel algorithms that give near-linear speedups even beyond 100 processors. The first method is the Threshold Depth-First Priority (TDFP) algorithm which uses inter-node parallelism up to a specified granularity combined with a depth-first priority scheduling policy for tasks. The second method is the Breadth-First List Scheduling (BFLS) algorithm which uses inter-node parallelism to decompose the search tree into a fixed number of tasks (subproblems), and then applies the intra-node parallel algorithm from [NAT87] on each task simultaneously with a small number of processors per task. We present simulation results to support the following observations:

1. The TDFP algorithm is more robust with respect to imbalanced search trees; the algorithm can achieve near-linear speedup even when the imbalance becomes a sequential bottleneck for the BFLS algorithm.
2. The BFLS algorithm is more robust with respect to increasing multiprocessor overhead, while the performance of the TDFP algorithm degrades severely in this case.

important property of our simulation is that we accurately measure the effect of scheduling overhead in the parallel algorithm by treating a priority queue operation as a critical section. Thus, the parallel execution times measured include the effect of serialization in the task scheduler. Another important parameter studied is the effect of priority. We show that, in the presence of overhead, there is an optimal intermediate threshold size which gives the best speedup.

The lookahead search in the Forward-Checker algorithm involves making a binding decision for a free variable,  $X$ , and regarding those values for the remaining free variables that are inconsistent with the value chosen for  $X$ . If all the remaining free variables still have a feasible value, the binding for  $X$  is performed and the search procedure moves forward to attempt to find a binding for the next free variable. A binding is found when no free variables remain. However, if any of the remaining free variables has no feasible value, the binding for  $X$  is undone, and another value is chosen. When there are no more values, the search procedure backtracks to the previous variable. The reader is referred to [HAR80] for a detailed description of the sequential algorithm.

The rest of this paper is organized as follows. In Section 2, we describe two methods for parallelizing the sequential Forward-Checker algorithm. In Section 3, we present experimental results based on simulations of these algorithms. The experiments take into account the effects of scheduling, synchronization and serialization in executing the parallel algorithms. We present experimental results for two constraint-satisfaction problems, the N-Queens and Graph Coloring problems. In Section 4, we present our conclusions.

## 2. Processor Scheduling Algorithms

We assume a shared-memory multiprocessor model in this paper. Our simulation system measures the parallel execution times, according to the task scheduling algorithm being used. The primary inter-processor interaction in the TDFP algorithm is due to insertions and deletions in the priority queue, which is accurately modelled as a critical section in our simulations. The priority queue contains all the necessary synchronization, since a parent task ensures that all the data needed by its child tasks is ready before the child tasks are created. Therefore, all other inter-processor interactions are secondary effects mainly due to interference among independent shared memory accesses. These effects depend on the shared memory architecture and are ignored in our simulations.

Ideally, a good parallel algorithm for a given problem should have the following desirable properties:

1. Sufficient parallelism for the number of processors we are interested in using.

2. Low overhead component in the parallel execution time. The overhead comes from controlling the parallel execution of a program (task creation, synchronization, communication, etc.).
3. Good load balancing of tasks on processors.
4. Efficient sequential execution time.
5. Reasonable space requirement. For our purpose,  $O(\text{NumProcs} \times \text{SeqSpace})$  space is reasonable, though  $O(\text{SeqSpace})$  space is ideal, where  $\text{NumProcs}$  is the number of processors and  $\text{SeqSpace}$  is the space used by an efficient sequential algorithm.  $O(\text{NumProcs} \times \text{SeqSpace})$  space can be considered reasonable for multiprocessors in which the total memory size increases linearly with the number of processors.

Properties 1, 2, and 3 are all necessary for good speedup. Property 4 is desirable for good performance measured in absolute terms. Property 5 is a necessary resource constraint to ensure that the parallel algorithm will execute with the available amount of memory. We next present two different algorithms for scheduling multiprocessor constraint-satisfaction searches and evaluate them against the criteria outlined above.

### 2.1 Threshold Depth-First Priority Algorithm

The Threshold Depth-First Priority algorithm (TDFP) described in this section satisfies all 5 properties as follows:

1. The TDFP algorithm exploits parallelism among nodes of the search tree. Since the number of parallel nodes could potentially be exponential, the amount of parallelism in the algorithm is more than adequate for any reasonable number of processors.
2. A low overhead component is obtained by using a threshold size to decide if a search tree node should be executed as a separate task or not. If the number of tests in the node is less than the threshold value then the node and all its descendants are executed sequentially as part of the parent task. Thus, the overhead of task creation, scheduling, synchronization, etc. only applies to computations larger than the threshold size, leading to a small amortized overhead. Naturally, this use of a threshold size reduces parallelism. But for the data sizes used in practice, there still remains plenty of parallelism after the threshold size has been made large enough to reduce the overhead component to an acceptable level.
3. Good load balancing is achieved by dynamically scheduling tasks at run-time. The scheduling policy is a form of non-preemptive list scheduling with no unforced idleness. This policy is guaranteed to yield a parallel execution time within a factor of 2 of the optimal schedule [GRA69].
4. The TDFP algorithm is a parallelization of the Forward-Checker search algorithm, which is considered to be one of the most time-efficient sequential algorithms for constraint-satisfaction problems. Both the parallel

TDFP and the sequential Forward-Checker algorithms perform exactly the same number of consistency tests. Hence our parallel algorithm has an efficient sequential execution time.

5. A depth-first priority rule is used to schedule tasks in the TDFP algorithm, so that nodes at a larger depth are executed first. This rule guarantees an  $O(\text{NumProcs} \times \text{SeqSpace})$  space usage, assuming that a parent task creates at most  $O(\text{NumProcs})$  child tasks. Space usage is a very important issue for tree-structured computations, where a simple FIFO scheduling policy could incur an exponential space expansion making it impossible for the program to continue execution.

The TDFP algorithm can be conveniently described by the recursive procedure, *Parallel\_Search*, in Figure 1 which is structured like the version in [NAT87]. The main difference is in the FORK statement. The WHEN clause in the FORK statement was introduced to avoid code duplication. If the WHEN condition is true, the computation enclosed in BEGIN...END is forked as a separate task with priority = *CurVar*. If the WHEN condition is false, the same computation is just executed sequentially by the parent task.

This algorithm assumes that a scheduling mechanism exists in the multiprocessor system to assign each processor a task with the highest priority. It is necessary to maintain a run-time priority queue for scheduling tasks. Since the priority values must be in a small bounded range (up to the maximum depth of the search tree, say  $\leq 100$ ), a priority queue can be efficiently implemented as an array (indexed by priority values) of lists of tasks.

All data communication between parallel tasks in procedure *Parallel\_Search* occurs between a parent and child. There are no global shared variables. The PRIVATE declaration gives each task its own copy of the variables. The TDFP algorithm thus has locality in communication. However, the priority-based task scheduler is assumed to be a centralized structure. An interesting area of future research would be to design efficient priority-based scheduling algorithms for distributed systems, which would allow the TDFP algorithm to work efficiently on message-passing multiprocessors as well.

Finally, we observe that this algorithm has no JOIN operation corresponding to the FORK's. So the parent task is just terminated after all its child tasks have been created. Thus, the parent does not need to be suspended till its child tasks have completed and the scheduling policy can be truly non-preemptive. Program execution is completed when there are no tasks in the priority queue and no tasks being executed on any processor.

```

PROCEDURE Parallel_Search(CurVar,F,FVT);
/* CurVar = current variable = computation depth */
/* F = partial solution, defined for F(1..CurVar-1) */
/* FVT is the Feasible Value Table. FVT(i) gives the
list of feasible values for CurVar ≤ i ≤ NumVars. */
IF CurVar = NumVars THEN
  Output all solutions defined by F(1..CurVar-1)
  and FVT(CurVar)
ELSE
FOR V ← each value in FVT(CurVar) DO
/* If the number of tests in FVT(CurVar+1) ...
FVT(NumVars) exceeds ThresholdSize, then fork
BEGIN ... END as a separate task with priority=CurVar
Otherwise, execute the computation sequentially. */
FORK WITH PRIORITY=CurVar
WHEN size(FVT(CurVar+1..NumVars))
  ≥ ThresholdSize
BEGIN
  PRIVATE New_F, New_FVT;
  New_F ← F; New_F(CurVar) ← V;
  New_FVT ← Check_Forward(CurVar, V, FVT);
  IF New_FVT is not empty THEN
    Parallel_Search(CurVar+1,New_F,New_FVT)
  END
END FOR
END PROCEDURE

PROCEDURE Check_Forward(CurVar,CurVal,FVT);
/* Return a new FVT with those (Variable,Value)
pairs that are consistent with (CurVar,CurVal).
Return an empty table if some variable has no
feasible value. */
/* Initialize NewFVT to an array of empty lists. */
NewFVT ← empty table;
FOR FreeVar ← CurVar+1 TO NumVars DO
  FOR V ← each value in FVT(FreeVar) DO
    /* This is the consistency test. */
    IF (FreeVar,V) is consistent
      with (CurVar,CurVal) THEN
      Insert V in NewFVT(FreeVar)
    END FOR;
    IF NewFVT(FreeVar) is empty THEN
    BEGIN
      NewFVT ← empty table;
      RETURN NewFVT
    END
  END FOR;
RETURN NewFVT
END PROCEDURE

```

Figure 1. Outline of the Threshold Depth-First Priority Algorithm

## 2.2 Breadth-First List Scheduling Algorithm

The Breadth-First List Scheduling Algorithm (BFLS) described in this section differs from the TDFP algorithm in how the multiple processors are allocated to search the tree. The algorithm exploits intra-node parallelism available at all levels of the search tree. The average amount of intra-node parallelism is large at nodes close to the root and falls off at deeper levels of the tree. If the amount of work to be done in a node is large enough to keep all the available processors busy, then all the processors are deployed to work in parallel within the node. However, when the available work within a node is not large enough to keep all the available processors performing useful work, then the algorithm explores many nodes in parallel (inter-node parallelism) by assigning a separate group of processors to each node.

The BFLS Algorithm was motivated by our earlier study [NAT87] where we observed a fall-off in intra-node parallelism with increasing node depth. The rationale for the present approach is to divide the search problem into smaller subproblems (or tasks) and to allocate the available processors to work in parallel on the tasks so that linear speedup behavior is achieved when a medium (100) to large (500) number of processors are used.

We next comment on the BFLS Algorithm with respect to the five points listed at the beginning of Section 2.

1. The parallelism used by the algorithm is limited by intra-node parallelism and by a user definable parameter *NumTasks*, the number of search subproblems created in Phase 1 of the algorithm.
2. The overhead is proportional to *NumTasks*. We ensure that  $NumTasks < NumProcs$ , so that the overhead will be  $O(NumProcs)$ . If *NumTasks* is too small (close to 1), the algorithm will suffer from lack of parallelism. If the *NumTasks* is too large, the overhead will become significant. We have found that a reasonable choice for *NumTasks* is between 5% and 10% of *NumProcs*.
3. Good load balancing is achieved by the list scheduling algorithm used in Phase 2.
4. The BFLS algorithm performs all the consistency tests performed by the sequential Forward-Checker algorithm, and may also do some extra tests. The experimental results reported in Section 3.2 take into account any extra work that may be performed by the parallel algorithm.
5. The space requirement of the algorithm is  $O(NumTasks \times SeqSpace)$ , where *SeqSpace* is the space required by the sequential Forward-Checker algorithm. By specifying *NumTasks* to be  $O(NumProcs)$ , we ensure the space required is reasonable.

The BFLS Algorithm is outlined in Figure 2. It consists of two phases. In Phase 1, a list of tasks corresponding to the subproblems is created. This is achieved by calling *Par\_Ch\_Forward*, a procedure that performs lookahead

```

/* Main Program */
/* N = Number of variables */
/* NumProcs = total number of processors */
/* FVT = Table of feasible values */
/* Phase 1: Creates a shared list of tasks */
  Create__Tasks(NumProcs,NumTasks,L);
  ProcsPerTask = max(1,NumProcs / NumTasks)
/* Phase 2: Schedule tasks using List Scheduler */
  while (L non-empty & a processor group is idle) do
    Schedule the next search Task L(i) for execution;
    Initialize F and FVT tables corresponding to L(i);
    INITIATE (BF__Par__LAS, L(i),ProcsPerTask);
  end;

/* Phase 1 */
Procedure Create__Tasks(NumProcs,NumTasks,L);
/* Create list L : L(1), L(2), ... , L(NumTasks) */
  Par__Ch__Forward with NumProcs; Visit in
  breadth-first sequence and enqueue tasks in L until
  number of entries in L reaches NumTasks.

Procedure BF__Par__LAS (Curr,F,FVT,PGS);
/* Phase 2: Use Parallel Lookahead search within */
/* each search task (see NAT87). Use PGS procs. */
/* in parallel within a node. Array F stores */
/* values assigned to the bound variables. */
FOR F(Curr) = each element of FVT(Curr)
DO BEGIN
  IF Curr < N
  THEN BEGIN
    Par__Ch__Forward(Curr,F(Curr),FVT,New__FVT,PGS);
    IF New__FVT not empty
    THEN BF__Par__LAS (Curr+1,F,New__FVT,PGS);
  END
  ELSE Output the solution F;
  END BF__Par__LAS;

  Par__Ch__Forward(Curr,L,FVT,New__FVT,PGSize);
/* New__FVT is New Feasible__Value Table. L is bound
to Curr, FVT is revised and New__FVT is returned.
PGSize processors are simultaneously used in
the parallel execution of the nested do loops that
perform the consistency tests. */
  New__FVT := emptytable;
  FOR free__var := Curr+1 to N DOALL
  BEGIN
    FOR tvalue := each element of FVT(free__var)
    DOALL
      Perform lookahead tests in parallel (NAT87)
    ENDDO;
  ENDDO;
  return(New__FVT);
END Par__Ch__Forward;

```

Figure 2. Outline of the Breadth-First List Scheduling Algorithm

operations within a node in parallel. In this invocation, all  $NumProcs$  processors are used to work in parallel within a node. After the list is created, the algorithm calculates  $ProcsPerTask$ , the number of processors to assign each task.

Phase 2 of the main program consists of the parallel execution of tasks in list L. A scheduler accesses list L in an exclusive mode, picks up the next task waiting to be executed and initiates a group of  $ProcsPerTask$  processors to execute it. Task initiation consists of invoking  $BF\_Par\_LAS$ , and passing it appropriate data structures corresponding to the task being scheduled.  $BF\_Par\_LAS$  in turn invokes  $Par\_Ch\_Forward$  to explore the nodes using  $ProcsPerTask$  processors within each node. Note that once a task is initiated, the scheduler is free to assign another task to another idle group-of processors, provided such a group exists. If all processor groups are busy, the scheduler waits until a group becomes idle and assigns that group another task. The completion of Phase 2 occurs when all the tasks in list L have finished execution.

### 3. Experimental Results

In the following subsections, we present simulation results for the TDFP and BFLS algorithms. Only consistency tests are considered as useful work, since they dominate the computation in a search algorithm. The simulated execution times are normalized with respect to the execution time of a consistency test, so that each test has execution time = 1. Results are presented for the following constraint-satisfaction problems:

1.  $N$ -Queens problem — Find all placements of  $N$  queens on an  $N \times N$  chessboard such that no queen can attack another. Our experiments used  $N = 8$ .
2.  $m$ -Colorings of graphs — Given an undirected graph and  $m$  colors, find all possible ways in which the graph can be colored such that no two adjacent vertices in the graph are assigned the same color. Our experiments used  $m = 5$  on a graph with 10 vertices (Figure 3). Vertex 1 was chosen as the distinguished vertex (fixed color). Note that the packing problem described in Section 1 can be formulated as a graph coloring problem.

#### 3.1 TDFP Algorithm

A simulation of the Threshold Depth-First Priority algorithm (Section 2.1) was implemented with the following parameters:

1.  $NumProcs$ , the number of processors.
2.  $ThresholdSize$ , the threshold size specified as a minimum number of tests to be processed in a parallel task.
3.  $QTime$ , the overhead incurred by a processor to insert or remove a task from the priority queue. An important

feature of our simulation is that the overhead time is assumed to be a critical section, just as in a real implementation. So, the waiting time to access the priority queue is often much more significant than  $QTime$ . In our simulation, the value of  $QTime$  is normalized with respect to the execution time of a consistency test (i.e.  $QTime=1$  means that an enqueue or dequeue operation takes the same time as a test).

4. The search problem to be solved. Any constraint-satisfaction problem can be solved by this algorithm by appropriately initializing FVT and defining the consistency test predicate.

Figures 4a, 4b, 4c and 5a, 5b, 5c display results of experiments performed on the 8 Queens and Graph Coloring problems. The parallel execution time can be considered to have 4 components, corresponding to the 4 different states that a processor can be in:

$$T_{Par} = T_{Useful} + T_{WorkWait} + T_{QWait} + T_Q$$

where

1.  $T_{Useful}$  is the average time spent by a processor doing useful work (i.e. consistency tests).
2.  $T_{WorkWait}$  is the average time spent by a processor waiting for work when the priority queue is empty. This usually occurs at the start of the program when a small number of tasks is available in the queue.
3.  $T_{QWait}$  is the average time spent by a processor waiting to enter the critical section of the priority queue. A processor needs to wait before it can pick a task for execution or insert any new child tasks in the queue.
4.  $T_Q$  is the average time spent by a processor on priority queue operations. If a processor needs to insert  $t$  tasks and then dequeue one task, it spends  $(t + 1)QTime$  time as overhead after entering the critical section.

Figures 4a and 5a show the effect of threshold size on speedup for the 8 Queens problem on 100 processors and the Graph Coloring problem on 500 processors. For each value of  $ThresholdSize$  on the X-axis, the corresponding speedup was plotted on the Y-axis. The speedup is defined to be the ratio  $T_{Seq}/T_{Par}$ , where

- $T_{Seq}$  is the total number of tests performed by the sequential Forward-Checker algorithm. Its value for the 8 Queens and Graph Coloring problems was 13024 and 202857 respectively.
- $T_{Par}$  is the simulated parallel execution time.

There are 4 curves in Figures 4a and 5a corresponding to 4 different values of queue overhead,  $QTime = 0, 0.1, 1, 10$ . We define  $OptThresholdSize(NumProcs, QTime)$  to be that value of  $ThresholdSize$  which gives the largest speedup on

processors with a queue overhead of  $QTime$ .<sup>1</sup> From Figure 4a for the 8 Queens problem, we can see that  $OptThresholdSize(100, QTime) = 1, 12, 23, 25$  for  $QTime = 0, 0.1, 1, 10$  respectively. From Figure 5a for the Graph Coloring problem, we have  $OptThresholdSize(100, QTime) = 1, 13, 17, 22$  for  $QTime = 0, 0.1, 1, 10$  respectively.

$QTime = 0$  represents the ideal zero overhead situation where the smallest threshold size (= 1) is optimal. In this case, the speedup only decreases (or stays the same) as the threshold size increases. However, the speedup curves for  $QTime = 0.1, 1, 10$  all exhibit a maximum at an intermediate threshold size. This is the trade-off between overhead and parallelism. If the threshold size is too small, the overhead component increases causing the speedup to decrease. If the threshold size is too large, there is less parallelism again causing the speedup to decrease. This existence of an optimal granularity has been predicted for parallel programs in general [SAR86], [SAR87] and is observed here in a real application.

Figures 4b and 5b show how the parallel execution time from Figures 4a and 5a (for  $QTime = 0.1$ ) is split into its components. The 3 curves plotted are:

1. Useful Work Fraction =  $T_{Useful}/T_{Par}$ .
2. Work Wait Fraction =  $T_{WorkWait}/T_{Par}$ .
3. Queue Wait Fraction =  $T_{QueueWait}/T_{Par}$ .

The ratio  $T_Q/T_{Par}$  was not plotted because it was negligible ( $< 0.01$ ) compared to the other 3 components. The waiting time,  $T_{QueueWait}$ , is much more significant than the actual time spent in the critical section,  $T_Q$ . The threshold size that gives the maximum speedup can now be identified as that value which maximizes the Useful Work Fraction. The Queue Wait Fraction decreases (or remains the same) as  $ThresholdSize$  increases, which confirms that increasing the threshold size reduces the overhead component. Similarly the Work Wait Fraction increases (or remains the same) as  $ThresholdSize$  increases because of loss of parallelism.

The only exception to the monotonic behavior of the Queue Wait and Work Wait fractions occurs in Figure 5b when  $ThresholdSize$  goes from 4 to 5. We see an unexpected drop in the Work Wait Fraction with a corresponding rise in the Queue Wait Fraction. However, the actual value of  $T_{QueueWait}$  (rather than the Queue Wait Fraction) decreases monotonically through 7159.4, 2319.8, 1761.4 for  $ThresholdSize = 3, 4, 5$ . The problem is in  $T_{WorkWait}$  which goes through the values 619.9, 1253.4, 455.8. This erratic behavior in  $T_{WorkWait}$  occurs because the priority queue is a critical section. At any given time, the queue contains a subset of the tasks that are ready to execute, because the

processors must wait to enter the critical section before they can enqueue their newly created tasks. This subset can be significantly smaller than the set of ready tasks when  $ThresholdSize$  is small because of the longer waiting time to access the queue.

Finally, Figures 4c and 5c show the speedup as a function of the number of processors for  $QTime = 0, 0.1, 1, 10$ . The threshold size used for a given  $QTime$  value is that size which gave the best speedup on 100 processors for 8 Queens, or on 500 processors for Graph Coloring. We see that all 4 speedup curves begin with a linear increase and then flatten out, depending on the value of  $QTime$ . The speedup is very sensitive to  $QTime$ , which indicates that a real implementation should try and reduce the queue overhead as much as possible. There are a few points in Figure 4c where increasing the number of processors actually causes a small decrease in speedup. This occurs because our scheduling policy is not optimal. However, since we use a form of list scheduling, the theorem in [GRA69] guarantees that this anomalous decrease in speed-up cannot be by more than a factor of 2. This anomaly was only observed for 8 Queens, because the Graph Coloring problem contains more work and used a larger number of tasks which had a smoothing effect on the speedup curves.

For the 8 Queens problem on 100 processors, the speedups obtained for  $QTime = 0, 0.1, 1, 10$  were 58.9, 49.9, 27.0 and 10.4 (see Figure 4c). For the Graph Coloring problem on 500 processors, the speedups obtained for  $QTime = 0, 0.1, 1, 10$  were 364.2, 249.3, 78.3 and 35.5 (see Figure 5c). This establishes that the Threshold Depth-First Priority algorithm can attain reasonable speedups in the presence of low overhead. In fact, the simulated speedup values presented here are conservatively low. We can expect better speedups in a real implementation because:

- The problem size used in a real multiprocessor will be much larger than 8 Queens or Graph Coloring (which take around 1 CPU second on a mainframe). Therefore, a larger threshold size can be used to get the same amount of parallelism, but with a lower overhead component.
- An efficient implementation of the priority queue can reduce overhead and increase speedup in two ways:
  1. By reducing  $QTime$ , which would give a larger speedup.
  2. By further reducing  $T_{QueueWait}$  by allowing many processors to simultaneously update the queue.
 For example, if the queue is implemented as an array of lists of tasks indexed by priority values, then each queue operation takes constant time and the insertion and deletion of tasks with different priorities can proceed simultaneously.

<sup>1</sup> If there is more than one optimal value for  $ThresholdSize$ , let  $OptThresholdSize(NumProcs, QTime)$  be the smallest one.

### 3.2 BFLS Algorithm

A simulation of the Breadth-First List Scheduling Algorithm (Section 2.2) was implemented with the following parameters:

1. *NumProcs*, the number of processors.
2. *NumTasks*, the number of tasks created in the list from which tasks are assigned by the scheduler for processing by small groups of processors. For the purpose of this study the number of tasks created was controlled by an input parameter *BF\_Upto\_Level* which specified that the search tree be explored breadth-first at levels 1 through *BF\_Upto\_Level*. For depths greater than *BF\_Upto\_Level*, the search was executed according to the list scheduling algorithm.
3. *ProcsPerTask*, the number of processors per task was varied from 1 through 20.
4. The search problem to be solved.

In our simulation of the BFLS algorithm, the overhead associated with accessing the shared list of tasks was assumed to be zero (a reasonable assumption if task granularity is sufficiently large). In the next section, we estimate the effect of overhead due to list accesses on speedup.

Figure 6 displays results of experiments performed on the 8-Queens problem. We plot three curves showing Speedup vs Number of processors that reflect the effects of different combinations of breadth-first and list scheduling of search effort. Specifically, the curves correspond to the following cases:

- Using a value of *BF\_Upto\_Level* = 1, a small number of subproblems (8 tasks) were created.
- Using a value of *BF\_Upto\_Level* = 2, a medium number of tasks (42) were created.
- Using a value of *BF\_Upto\_Level* = 3, a large number of tasks (140) were created.

We make the following observations from Figure 6. When the number of tasks is small (= 8, a limited use of breadth-first search), the speedup increases upto about 100 processors and then flattens out at a limiting speedup value of 52 for larger number of processors. When a medium number of tasks (= 42, an intermediate amount of breadth-first search) is created, the speedup increases upto about 200 processors and then flattens out gradually at 125 (significantly higher than 52). When a large number of tasks (= 140, a large amount of breadth-first search) is created, the speedup tends to flatten out at about 62. This set of experiments suggests that the limiting speedup behavior of Breadth-First List Scheduling improves with increasing the number of tasks (i.e., the breadth-first component) up to some intermediate point and then starts deteriorating as the number of tasks is increased even further. The reason why the limiting speedup peaks at some intermediate number of

tasks rather than increase monotonically with increasing number of tasks is as follows. The total time in executing the search consists of two components:

1. A Breadth-First component, where nodes of the search tree are examined sequentially and tasks are created for parallel execution.
2. A List Scheduling component, different tasks are executed in parallel.

The time spent in the Breadth-First component increases with the number of tasks created. This is shown in Figure 7 as  $T_{Serial}(BF)$ . The time spent in the List-Scheduling component decreases with increasing number of tasks created because of greater potential for inter-node parallelism. This is shown in Figure 7 as  $T_{Parallel}(LS)$ . There exists an intermediate value for the number of tasks such that the sum of the two components reaches a minimum (the right most bar in each set of three bars shown in Figure 7). Since the total amount of work done by the parallel algorithm is fixed, the work performed during the creation of a shared list of tasks must be balanced against potential benefit due to parallel execution of tasks.

Figure 8 shows results corresponding to Graph Coloring problem. Note that the graph shown in Figure 3 was used as an instance of the problem. Figure 8 displays results (Speedup vs Number of processors) of experiments performed with different task decompositions. Specifically, the following cases were considered:

- Using a value of *BF\_Upto\_Level* = 3, a small number of tasks (64) were created.
- Using a value of *BF\_Upto\_Level* = 4, a medium number of tasks (204) were created.
- Using a value of *BF\_Upto\_Level* = 5, a large number of tasks (816) were created.

When the number of tasks created is 64, speedup tends to flatten out at 229. When the number of tasks created is 204, the speedup tends to flatten out at about 413 (significantly more than 229). When the number of tasks created is 816, the speedup tends to flatten out at about 174.

The experiments performed with the N-Queens and Graph Coloring problems suggest that depending on the problem and its size, an optimal mix of sequential breadth-first traversal and parallel list scheduling should be used to maximize the performance benefit due to parallelism.

### 4. Conclusions

This paper has presented two new methods for parallelizing the Forward-Checker algorithm, namely the Threshold Depth-First Priority (TDFP) and the Breadth-First List Scheduling (BFLS) algorithms. Experimental results show that these algorithms can achieve near-linear speedups beyond 100 processors for problems where the speedup

obtained by just using intra-node parallelism falls off at 10 processors.

The performance of the TDFP algorithm is very sensitive to the queue overhead,  $QTime$ , especially since a queue operation is assumed to be a critical section. It is very important to reduce the size of the critical section, and hence the value of  $QTime$ , as much as possible. For a priority queue implemented as an array (indexed by priority values) of lists, serialization is only necessary for operations with the same priority. Ideally, the basic FIFO queue operation for a given priority could just be a single Fetch-and-Add instruction (or some other equivalent instruction). While it is important to reduce the size of the critical section, the real solution to the serialization problem is to implement a distributed priority scheduler. This is an interesting area for future research.

An important parameter of the TDFP algorithm is the threshold size which defines the granularity of execution. The optimal granularity depends on the nature of the search problem, the number of processors ( $NumProcs$ ) and the queue overhead ( $QTime$ ). We observed that, in the presence of overhead, there is an optimal threshold size which gives the best speedup. This is the trade-off between overhead and parallelism. If the threshold size is too small, the overhead component increases causing the speedup to decrease. If the threshold size is too large, there is less parallelism again causing the speedup to decrease.

Similarly, the parameter which controls the granularity of the BFLS algorithm is  $NumTasks$ , the number of subproblems to be created. If the specified value is too small, the algorithm will suffer from lack of parallelizable work. If the specified value is too large, the sequential breadth-first traversal will become a bottleneck.

The TDFP and BFLS algorithms were found to perform well under different conditions. A speedup comparison is given in Table 1. The BFLS speedups were obtained assuming 42 tasks for 8 Queens and 204 tasks for Graph Coloring (i.e. the best values for  $NumTasks$ ). For the BFLS algorithm, the speedup with overhead was simply computed as  $NumTests / (T_{par} + 2 \times NumTasks \times QTime)$ , so that each task incurs an enqueue and a dequeue overhead just as in the TDFP algorithm. A \* is placed against a speedup entry in the table if the speedup is at least 10% more than the speedup due to the other method.

For low overhead ( $QTime = 0, 0.1$ ), the TDFP algorithm performed better than the BFLS algorithm for the Graph Coloring problem and they performed comparably on the N-Queens problem. This is because the search tree for Graph Coloring problem is more imbalanced than the N-Queens problem. For large overhead ( $QTime = 1, 10$ ), the BFLS algorithm performed better than the TDFP algorithm on both problems. This is because the TDFP

algorithm is more sensitive to increased overhead. Therefore, both algorithms are suitable for solving constraint-satisfaction problems. The best choice depends on the search problem and the overhead in the target multiprocessor.

## References

- [BIT75] J. R. Bitner and E. M. Reingold, "Backtrack programming techniques," *Comm. of the ACM*, vol. 18, pp. 651-656, Nov 1975.
- [GOL65] S. Golomb and L. Baumert, "Backtrack programming," *Journal of the ACM*, vol. 12, 1965, pp. 516-524.
- [GRA69] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics* 17(2), March 1969.
- [HAR80] R. M. Haralick and G. L. Elliott, "Improving tree search efficiency for constraint-satisfaction problems," *Artificial Intelligence* pp.263-313, 1980.
- [KAS86] S. Kasif, "On the parallel complexity of some constraint-satisfaction problems," *Proc. of the Fifth National Conference on Artificial Intelligence*, Philadelphia, PA, 1986, pp.349-353.
- [KOR85] R. E. Korf, "Depth-first iterative-deepening: an optimal admissible tree search," *Artificial Intelligence*, 1985.
- [NAT87] K. S. Natarajan, "An empirical study of parallel search for constraint-satisfaction problems," IBM Research Report 13320, Nov. 1987.
- [NUD88] B. A. Nudel, "Tree search and arc consistency in constraint satisfaction algorithms," in *Search in Artificial Intelligence*, edited by L. Kanal and V. Kumar, Springer-Verlag, 1988 (to appear).
- [RAO87] V. N. Rao, V. Kumar and K. Ramesh, "A parallel implementation of the IDA\* algorithm," *Proc. of the AAAI-87*, July 1987, Seattle, WA.
- [SAR86] V. Sarkar and J. L. Hennessy, "Partitioning Parallel Programs for Macro-Dataflow," *Proc. of the ACM Conference on Lisp and Functional Programming*, August 1986, pp. 202-211.
- [SAR87] V. Sarkar, "Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors," Ph.D. thesis, Stanford University, April 1987, Technical Report No. CSL-TR-87-328.

Problem	Procs	QTime	Method	Speedup
8Queens	84	0.0	TDFP	53.8
8Queens	84	0.0	BFLS	54.3
8Queens	84	0.1	TDFP	49.0
8Queens	84	0.1	BFLS	52.4
8Queens	84	1.0	TDFP	27.0
8Queens	84	1.0	BFLS	40.2 *
8Queens	84	10.0	TDFP	10.4
8Queens	84	10.0	BFLS	12.1 *
Coloring	410	0.0	TDFP	314.0 *
Coloring	408	0.0	BFLS	237.3
Coloring	410	0.1	TDFP	230.9
Coloring	408	0.1	BFLS	226.5
Coloring	410	0.1	TDFP	78.3
Coloring	408	1.0	BFLS	160.6 *
Coloring	410	10.0	TDFP	35.5 *
Coloring	408	10.0	BFLS	41.1 *

Table 1: Speedup comparison of the TDFP and BFLS algorithms

Figure 3: Graph used in Coloring Problem

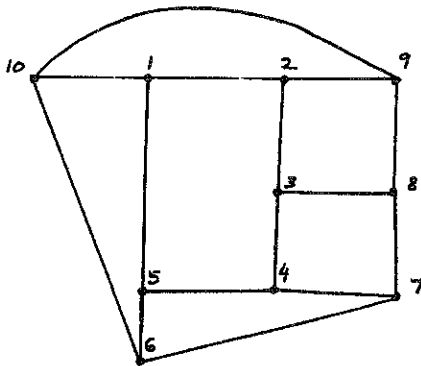


Figure 4(a): 8 Queens: Effect of ThresholdSize

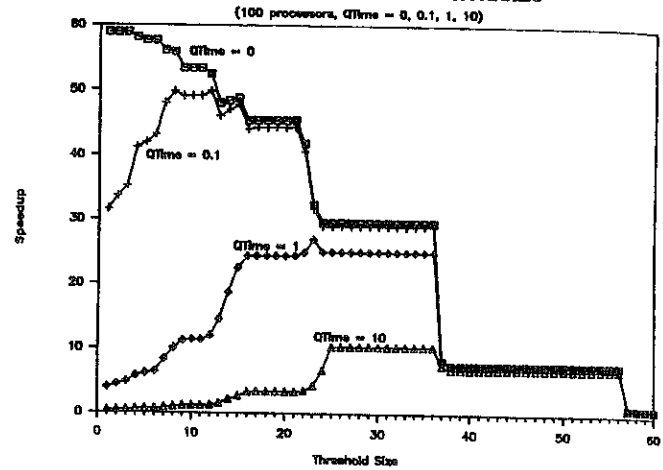


Figure 4(b): 8 Queens: Parallel time components

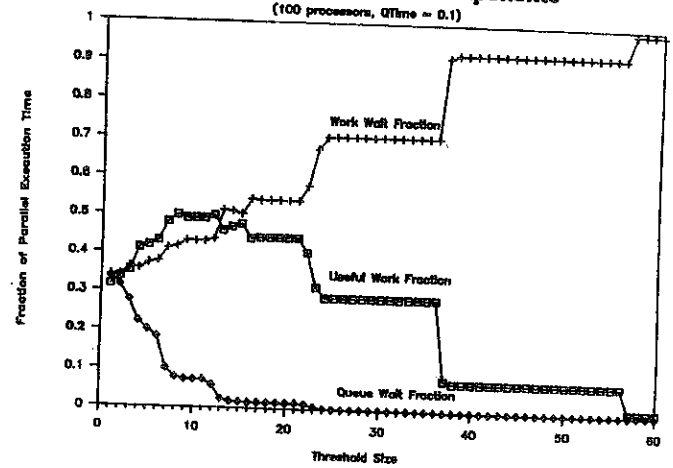


Figure 4(c): 8 Queens: Speedup vs Processors

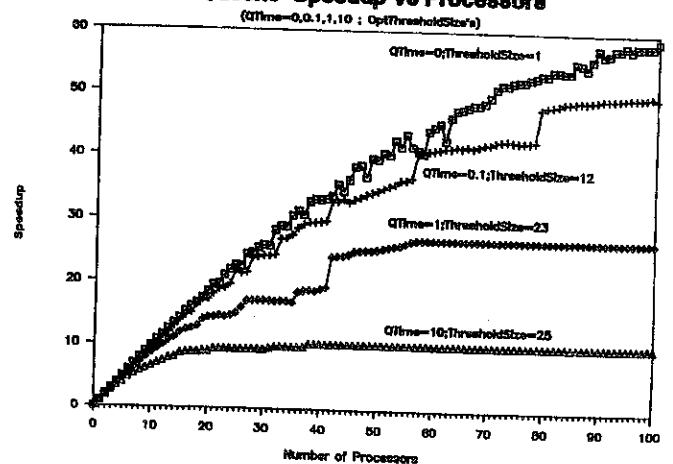


Figure 5(a): Graph Coloring: Effect of ThresholdSize  
(500 processors,  $QTime = 0, 0.1, 1, 10$ )

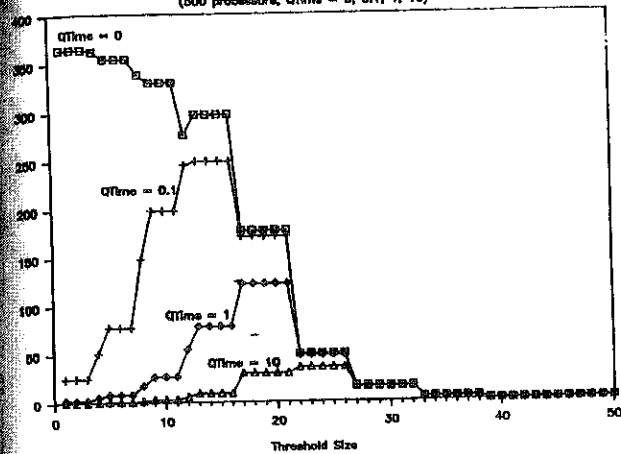


Figure 5(b): Graph Coloring: Parallel time components  
(500 processors,  $QTime = 0.1$ )

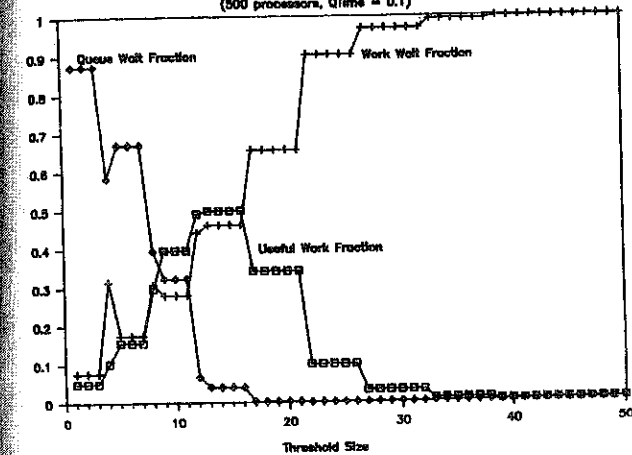


Figure 5(c): Graph Coloring: Speedup vs Processors  
( $QTime=0,0.1,1,10$ ; OptThresholdSize's)

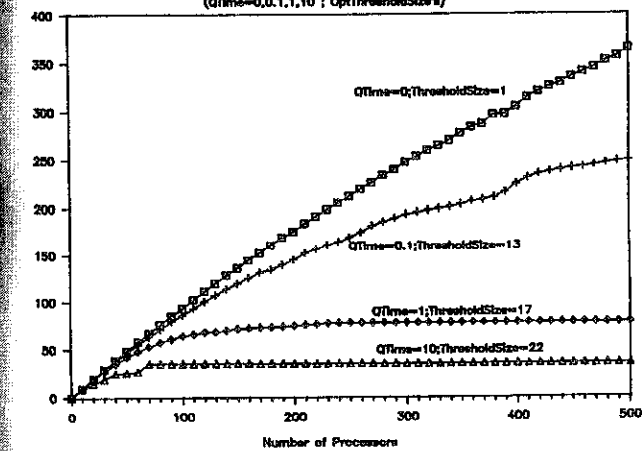


Figure 6: 8 Queens: Speedup vs Processors  
(Number of Tasks = 8, 42, 140)

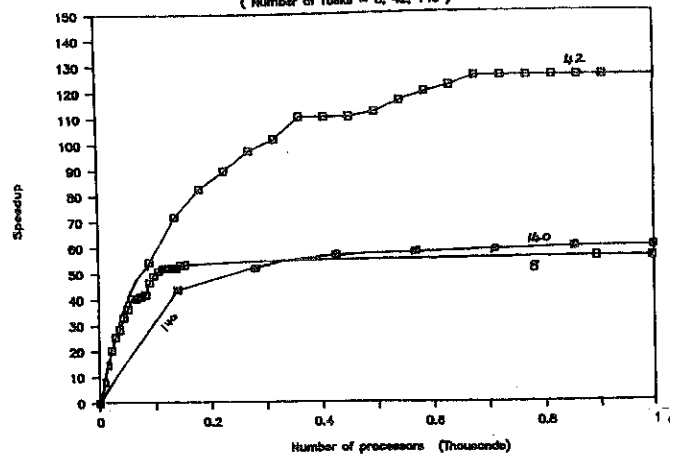


Figure 7: Components of Total Time  
(Serial and Parallel Sections)

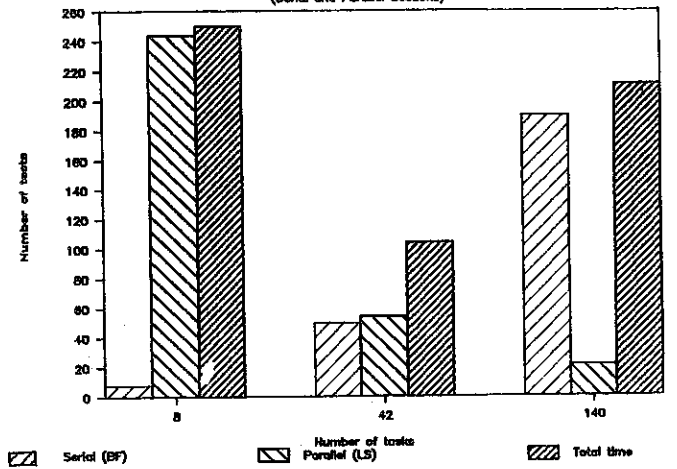


Figure 8: Graph Coloring: Speedup vs Processors  
(Number of tasks = 64, 204, 816)

