PARALLEL DEPTH FIRST SEARCH ON THE RING ARCHITECTURE

Vipin Kumar, V. Nageshwara Rao, and K. Ramesh

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-88-16 April 1988

Parallel Depth First Search on the Ring Architecture*

Vipin Kumar, V. Nageshwara Rao and K. Ramesh
Artificial Intelligence Laboratory
Department of Computer Sciences,
University of Texas at Austin,
Austin, Texas 78712

January 1988

To appear in the proc. of 1988 Int'l Conf. on Parallel Processing

Abstract

This paper presents the implementation and analysis of parallel depth-first search on the ring architecture. At the heart of the parallel formulation of depth-first search is a dynamic work distribution scheme that divides the work between different processors. The effectiveness of the parallel formulation is strongly influenced by the choice of the work distribution scheme. In particular, a commonly used work distribution scheme is found to give very poor performance on large rings(> 32 processors). We present a new work distribution scheme that is better than the work distribution scheme used by other researchers, and gives good performance even on large rings (128 processors). We introduce the concept of iso-efficiency function to characterize the effectiveness of different work distribution schemes.

1 Introduction

Depth-First Search(DFS) is a general technique used in Artificial Intelligence for solving a variety of problems in planning, decision making, theorem proving, expert systems, etc. [18,10]. It is also used under the name of backtracking to solve various combinatorial

*This work was supported by Army Research Office grant # DAAG29-84-K-0060 to the Artificial Intelligence Laboratory, and Office of Naval Research Grant N00014-86-K-0763 to the computer science department at the University of Texas at Austin.

problems[6] and constraint satisfaction problems[14]. Execution of a Prolog program can be viewed as depth-first search of a proof tree [22]. Iterative-Deepening algorithms perform cost-bounded DFS in successive iterations to solve discrete optimization problems[9] and theorem proving[21]. A major advantage of depth-first search strategy is that it requires very little memory. Since many of the problems solved by DFS are highly computation intensive, there has been a great interest in developing parallel versions of depth-first search [7,24,11,5,23,8].

This paper presents the implementation and analysis of parallel depth-first search on the ring architecture. At the heart of the parallel formulation of depth-first search is a dynamic work distribution scheme that divides the work between different processors. The effectiveness of the parallel formulation is strongly influenced by the choice of the work distribution scheme. In particular, a most commonly used work distribution scheme is found to give very poor performance on large rings($>$ 32 processors). We present a new work distribution scheme that is better than the work distribution scheme used by other researchers, and gives good performance even on large rings (128 processors). The performance is tested by solving the 15-puzzle problem[18]. The ring architecture is embedded on an 128-node Intel Hypercube. We introduce the concept of iso-efficiency function (representing the required growth in problem size with respect to number of processors to maintain the efficiency) to characterize the effectiveness of different work distribution schemes. We have also implemented parallel depth first search on Intel Hypercube and BBN Butterfly[15,16]. The ring architecture is important because it is simple to construct and is highly scalable.

Section 2 gives a brief review of sequential depth-first search. Section 3 presents a parallel formulation of DFS and a simple work distribution scheme. Section 4 presents performance results of solving the 15-puzzle by parallel depth-first search on the ring architecture. Section 5 contains an analysis of the performance of parallel depth-first search for different work distribution algorithms, and presents a new improved work distribution scheme. Section 6 reviews previous work on parallel depth-first search. Section 7 contains concluding remarks.

2 Review of Depth-First Search

Search methods are useful when a problem can be formulated in terms of finding a path in an (implicit) directed graph from an initial node to a goal node. The search begins by expanding the initial node; i.e., by generating its successors. At each next step, one of the previously generated nodes is expanded until a goal node is found. There are many ways in which a generated node can be chosen for expansion, each having its own advantages and disadvantages. In depth-first search, one of the most recently generated nodes is expanded first. A detailed treatment of depth-first search is provided in [10] and [19]. Depth-first search does not use heuristic information to eliminate the search space. It also does not find the shortest solution path.

Iterative-Deepening-A* (IDA*)[9] is a variation of depth-first search that takes care of both these drawbacks. IDA* is able to use heuristic information to speed up search, and is admissible; i.e., it always finds a least-cost solution path. IDA* performs repeated cost-bounded depth-first search (DFS) over the search space.

The unit of computation in a search algorithm is the time taken for one node expansion. The total time taken by a sequential search algorithm is roughly proportional to the total number of nodes it expands. Total number of nodes expanded by a search algorithm for a particular instance is called the **problem size** W of the instance. If the depth of the search tree is d, then the **effective-branching factor** b is defined as $log_d W$.

3 Parallel Depth-First Search

3.1 A Parallel Formulation of Depth-First Search

We parallelize depth-first search by sharing the work done among a number of processors. Each processor searches a disjoint part of the search space in a depth-first fashion. When a processor has finished searching its part of the search space, it tries to get an unsearched part of the search space from other processors. When a solution is found, all of them quit. If the search space is finite and has no solutions, then eventually all the processors would run out of work, and the (parallel) search will terminate.

Since each processor searches the space in a depth-first manner, the (part of) state-space to be searched is easily represented by a stack. The depth of the stack is the depth of the currently explored node, each level of the stack keeps track of untried alternatives. Each processor maintains its own local stack on which it executes DFS. When the local stack is empty, it sends a request for work to another processor. Each processor periodically checks for incoming work requests. If it has untried alternatives in the stack, then it sends some of them to the requesting processor;[1] otherwise it sends a null message back. See Fig. 1 for an illustration. In our implementation, at the start of each iteration, all the search space is given to one processor, and other processors are given null space (i.e., null stacks). From then on, the state-space is divided and distributed among various processors.

The basic driver routine in each of the processors is given below.

```
Parallel DFS: Processor(i)
  while (not terminated) do
    if (stack = empty) then GETWORK() ;
```

[1]It is important to make sure that the work given out is not too small (otherwise the requesting processor will be out of work soon again) or too large (otherwise the donor processor will be out of work soon). The best strategy is to try to give nearly half of the local work.

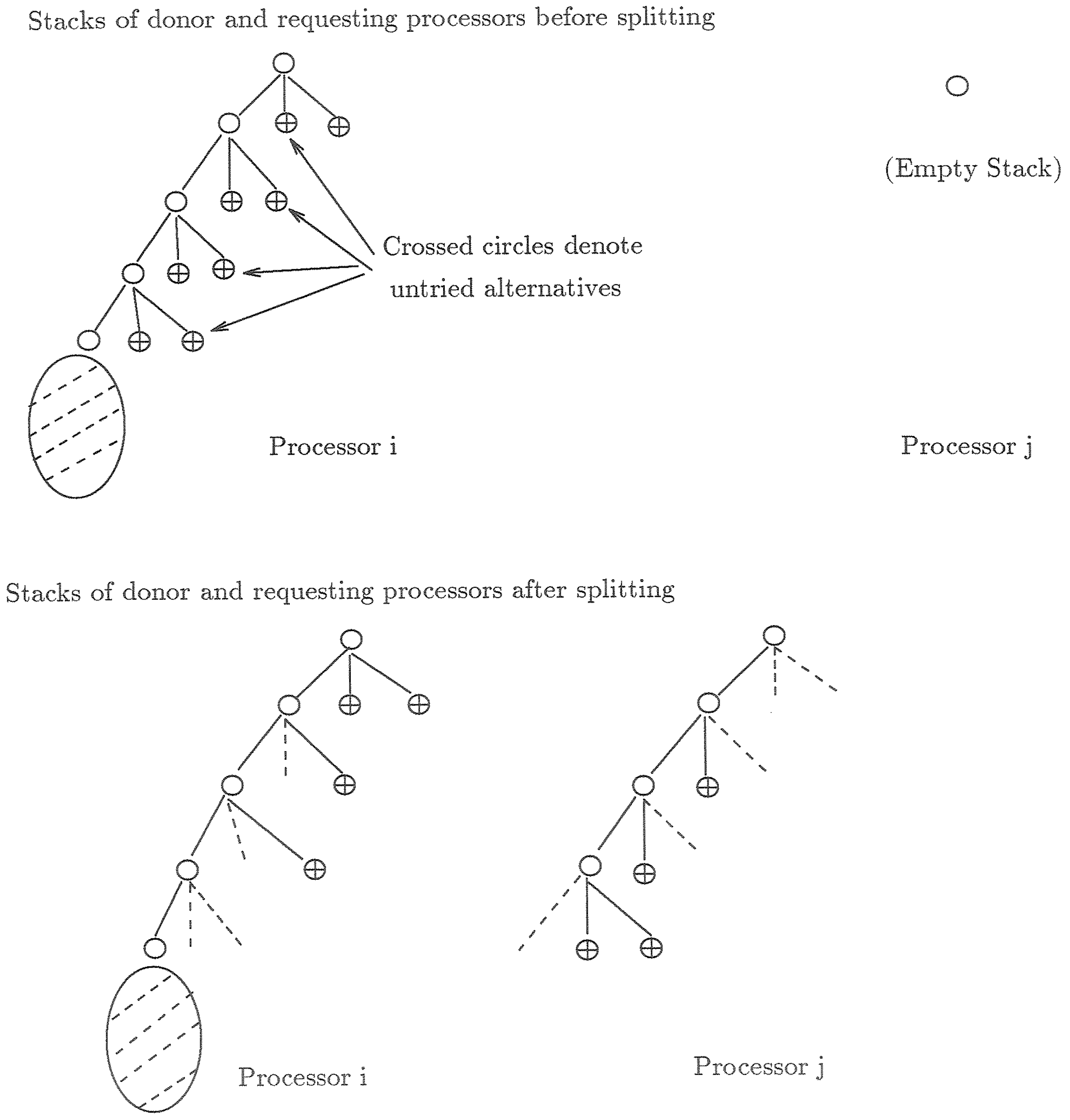

Stacks of donor and requesting processors before splitting
Crossed circles denote
untried alternatives
(Empty Stack)
Processor i
Processor j
Stacks of donor and requesting processors after splitting
Processor i
Processor j

Figure 1: Splitting work in a stack between two processors in Parallel DFS. Processor j is requesting work from Processor i. Stacks are assumed to grow downward.

```
    while (stack != empty) do
      DFS(stack) ;
      GETWORK() ;
    od
    TERMINATION-TEST() ;
  od
```

Once available space at a processor is fully searched, it calls GETWORK() to get more work. If new work is not received (from the repeated trials in GETWORK()), then a termination test is done to see if any other processor has found a solution or if every one else has finished searching it's assigned space.[2] If the termination test fails, then GETWORK() is called again to get some work. Procedure GETWORK is given below. Send(target, (WORK_REQUEST, my_id)) sends a (WORK_REQUEST, my_id) message to a processor whose id is target. Receive(WORK) waits to receive a WORK message.

```
GETWORK()
  for (j = 0;j < NUMRETRY ; j = j + 1)
    {
      target = one of the immediate neighbor;
      send( target, (WORK_REQUEST, my_id));
      receive( WORK);
      if (WORK != null) then return(SUCCESS);
    }
  return(FAIL) ;
```

As shown here, GETWORK requests for work only from an immediate neighbor. This is a simple and natural scheme and has been used by many researchers (see Section 6). Other work distribution schemes are possible, and will be considered later.

3.2 Speedup Anomalies

In Parallel DFS all the processors abort when the first solution is detected by any processor. Due to this it is possible for Parallel DFS to expand fewer or more nodes than DFS, depending upon when a solution is detected by a processor. Even on different runs for solving the same problem, Parallel DFS can expand different number of nodes, as the processors run asynchronously. If Parallel DFS expands fewer nodes than DFS, then we can observe speedup of greater than N using N processors. This phenomenon (of greater than N speedup on N processors) is referred to as acceleration anomaly [12,26].

[2]To check if every processor has run out of work, Dijkstra's ring termination detection algorithm[2] is executed in an interleaved fashion with the rest of the computation.

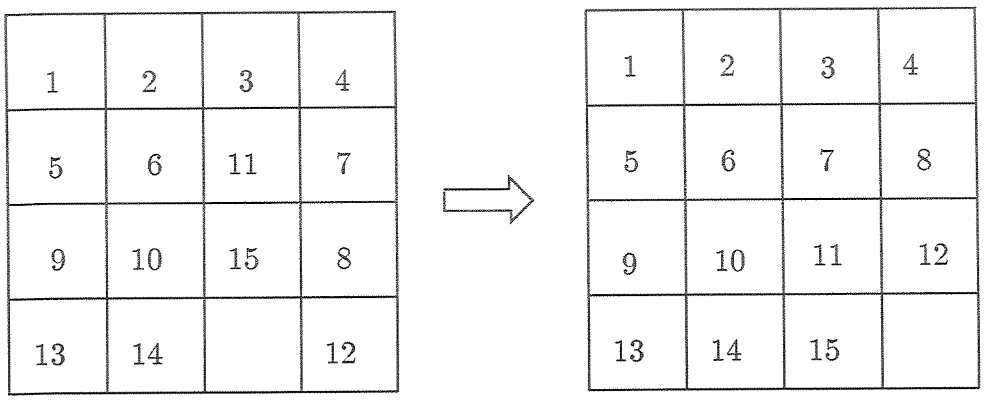
4 Performance of Parallel DFS

4.1 Experiments for Evaluating Parallel DFS

To test the effectiveness of Parallel DFS, we have used it to solve the 15-puzzle problem [18]. The 15-puzzle is a 4x4square tray in which are placed 15 square tiles. The remaining sixteenth square is uncovered. Each tile has a number on it. A tile that is adjacent to the blank space can be slid into that space. A game consists of a starting position and a specified goal position. The goal is to transform the starting position into the goal position by sliding the tiles around.(See Fig. 2). The 15-puzzle problem is particularly suited for testing the effectiveness of parallel DFS, as it is possible to create search spaces of different sizes (W) by choosing appropriate starting configurations. IDA* is the best known sequential algorithm to find optimal solutions for the 15-puzzle problem[9]. It is much faster than simple depth-first search, as it is able to use the Manhattan distance heuristic [18] to focus the search. We have parallelized IDA* to test the effectiveness of our parallel formulation of depth-first search. Since each iteration of IDA* is a cost-bounded depth-first search, a parallel formulation of IDA* is obtained by executing each iteration via parallel DFS. To isolate the overhead due to work distribution from the overhead due to termination detection in each iteration, we modified IDA* and its parallel version to execute only the last iteration (by starting IDA* with the final cost bound). To study the speedup of parallel approach in the absence of anomaly, we modified IDA* and Parallel IDA* to find all optimal solutions. This ensures that both IDA* and Parallel IDA* search all the space within the cost bound of the final iteration; hence both IDA* and Parallel IDA* explore exactly the same number of nodes.[3]

We implemented Parallel cost-bounded search (i.e., the last iteration of IDA*) to solve the 15-puzzle problem on 1-ring and 2- ring embedded on an Intel Hypercube. On 1-ring (i.e., the unidirectional ring), a processor could ask for work from only one neighbor. On a 2-ring, a processor could ask for work from both of its neighbors. We ran our algorithm on a number of problem instances given in Korf's paper [9]. As shown in Fig 3, we are able to get linear speedup up to 16 processors, but for more processors, the performance is not very good. The performance of a 2-ring is better than a 1-ring, but the maximum speedup obtained is only 25 even on 128 processors. In general, for a given number of processors, we get more speedup for bigger problems and less speedup for smaller problems. The size of a problem is determined by it's sequential execution time. The average execution time of the problems for which the speedups of Fig. 3 were obtained is roughly 200 minutes. On

[3]We have also tested the performance of parallel DFS w.r.t sequential DFS when both find only one optimal solution. For this case, the speedup varies dramatically for different executions, and different problems. On the average we obtained slightly superlinear speedups. These speedup results, from runs on a shared memory multiprocessor (Sequent Balance), are reported in [15]. The phenomenon of superlinear speedup is further analyzed in [27].



A starting configuration Desired goal configuration

Figure 2: The 15-puzzle

smaller problems (sequential execution time 16 minutes), the maximum speedup for 128 processors for 2-ring is approximately 10. But even for very large problems, we were not able to get speedups significantly higher than 25. It seems that parallel depth-first search with the simple work distribution scheme is incapable of effectively utilizing larger rings. The next section presents an analysis of this scheme which explains this poor performance.

5 Analysis of Performance

In this section we analyze the performance of parallel cost-bounded DFS. We assume that the effective branching factor[4] of the cost-bounded search space is greater then $1+e$ (where e is a positive constant). We also assume that whenever work W is split between a donor and a requester, then the smallest of the two work pieces is at least αW (for some constant α such that $0 \leq \alpha \leq 0.5$). To avoid speedup anomalies (discussed in Section 3.2), we assume that both sequential and parallel DFS search the whole cost bounded space for all solutions. All these conditions are met by the parallel formulation presented in Section 3.

5.1 Definitions and Terminology

1. Problem size W: is the size of the space searched (in number of nodes)

2. Number of processors N: is the number of processors being used to run Parallel DFS.

[4]Note that due to use of heuristic function h, the effective branching factor of a search tree could be much smaller than the average number of successors of a node. For example, in the 15-puzzle, the average number of successors of a node is 2 (not counting the parent), whereas due to Manhattan distance heuristic, the effective branching factor ranges between 1.2 and 1.3 .

1

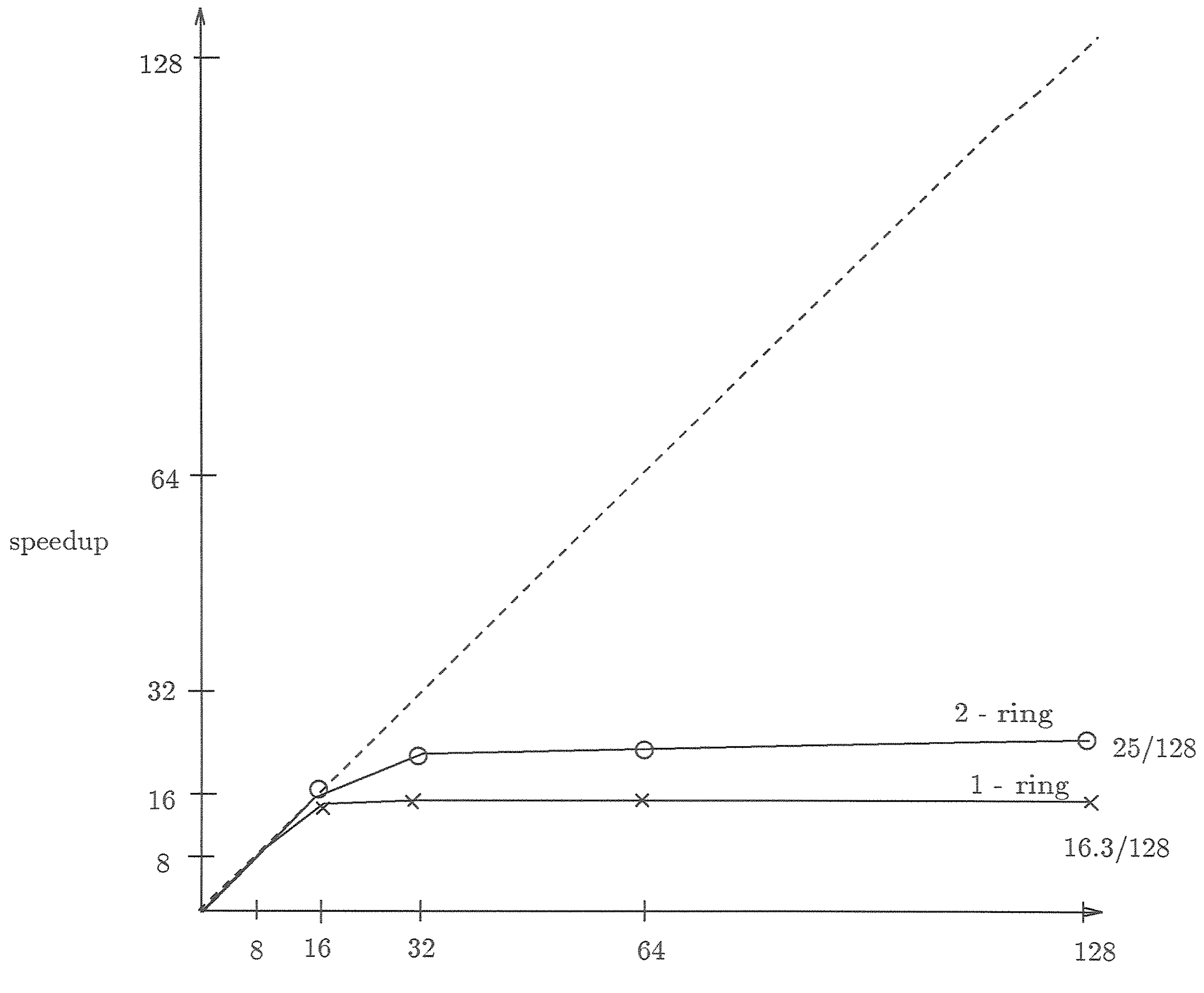


Figure 3: Average speedup vs Number of processors for parallel cost-bounded depth-first search on a ring embedded in Intel Hypercube. Sequential Exec. time $\simeq$ 10500 secs, problem size $\simeq$ 9 M nodes

3. Running time T_N: is the execution time on N processors. T_1 is the sequential execution time.

4. Computation time T_{calc}: is the sum of the time spent by all the processors in useful computation. Since, both sequential and parallel versions search exactly the same cost-bounded space (to find all optimal solutions),

$$T_{calc} \text{ on } N \text{ processors } = T_{calc} \text{ on } 1 \text{ processor } = T_1$$

5. Communication time T_{comm}: is the sum of the time spent by all processors in communicating with neighboring processors, waiting for arrival messages, time in starvation, etc. For single processor execution, $T_{comm} = 0$. Since, at any time, a processor is either communicating or computing,

$$T_{comm} + T_{calc} = N * T_N$$

6. Speedup S: is the ratio $\frac{T_1}{T_N}$.

   It is the effective gain in computation speed achieved by using N processors in parallel on a given instance of a problem.

7. Efficiency E: is the speedup divided by N. E denotes the effective utilization of computing resources.

$$\begin{aligned} E &= \frac{S}{N} \\ &= \frac{T_1}{T_N * N} \\ &= \frac{T_{calc}}{T_{calc} + T_{comm}} \\ &= \frac{1}{1 + \frac{T_{comm}}{T_{calc}}} \end{aligned}$$

8. Unit Computation time U_{calc}: is the mean time taken for 1 node expansion.

9. Unit Communication time U_{comm}: is the mean time taken for sending a work request to a processor and receiving a response (work or a null message). In the work distribution scheme of section 3, U_{comm} is a fixed constant (determined by the speed of the communication).

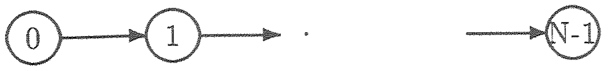


Figure 4: Linear Chain of processors

5.2 Iso-efficiency Functions

As discussed in the previous section, the efficiency obtained in parallel DFS is determined by the number of processors and the problem size.[5] For a given problem size W, increasing the number of processors N causes the efficiency to decrease because T_{comm} increases while T_{calc} remains the same. For a fixed N, increasing W improves efficiency because T_{calc} increases and the work distribution scheme with α–splitting does not cause a proportionate increase in T_{comm} . If N is increased, then we can keep the efficiency fixed by increasing W. The rate of increase of W with respect to N is dependent upon the architecture and the work distribution algorithm. The required rate of growth of W w.r.t N (to keep efficiency fixed) essentially determines the scalability of the architecture for the work distribution algorithm. For example, if W is required to grow exponentially w.r.t. N, then it would be difficult to utilize the architecture for a large number of processors. On the other hand, if W needs to grow only linearly w.r.t N, then the work distribution scheme is highly suited for the architecture. If W needs to grow as f(N) to maintain an efficiency E, then f(N) is the **iso-efficiency function** and the plot of f(N) w.r.t N is the **iso-efficiency curve**.

Next we derive the iso-efficiency function of parallel cost-bounded DFS for 1-ring. The analysis for 2-ring is similar and is left out. We present theoretical models that give us bounds on total communication time T_{comm} in terms of problem size W and number of processors N for different work distribution schemes. Predictions from our models seem to closely agree with experimental data, hence we feel that the models are reliable.

5.3 Iso-efficiency Analysis of the simple work distribution scheme

Consider a linear chain of N processors of Fig. 4. A 1–ring is a linear chain with a fold back from processor N-1 to 0. In a 1–ring a processor can get work from its left neighbor and send work to its right neighbor. Initially W work is available in processor 0. In order to

[5]It is also determined by the architecture; eg., hypercube and shared memory architectures provide better efficiencies for parallel DFS [16]. In this paper we restrict our discussion to the ring architecture only.

achieve good work distribution every processor needs to get roughly $\frac{W}{N}$ for itself[6]. Suppose that stack splitting follows α–split. Then

Maximum piece of work coming into processor 0 is W
Maximum piece of work coming into processor 1 is αW
Maximum piece of work coming into processor i is $\alpha^i W$
From the above, we can see that in order to get $\frac{W}{N}$ work
Processor i has to get at least $\frac{\frac{W}{N}}{\alpha^i W}$ transfers ($i \geq 0$).

$$Hence\ the\ total\ number\ of\ stack\ transfers\ \geq \sum_{i=0}^{N-1} \frac{1}{N\alpha^i}$$

$$= \frac{1}{N} \sum_{i=0}^{N-1} \beta^i\ (where\ \beta = \frac{1}{\alpha})$$

$$= \frac{\beta^N - 1}{\beta - 1} * \frac{1}{N}$$

$$T_{comm} = U_{comm} * \frac{\beta^N - 1}{\beta - 1} * \frac{1}{N}\ (lower\ bound)$$

$$T_{calc} = U_{calc} W$$

$$Efficiency = \frac{1}{1 + \frac{T_{comm}}{T_{calc}}}$$

$$= \frac{1}{1 + \frac{U_{comm}}{U_{calc} N W} * \frac{\beta^N - 1}{\beta - 1}}$$

For constant efficiency

$$U_{calc} N W = U_{comm} \frac{\beta^N - 1}{\beta - 1}$$

or

$$W = \Omega(\frac{\beta^N}{N})$$

(since U_{comm} and U_{calc} are constant)

Thus the iso-efficiency function is exponential.[7] The Iso-efficiency function for 2-ring can be obtained similarly, and is also exponential. This explains the poor performance of 1-ring and 2-ring. Fig. 5 shows experimentally iso-efficiency curves obtained for parallel DFS for the 15-puzzle on a 1-ring embedded in the Intel Hypercube. Clearly these curves show exponential growth.[8]

[6]This is true only if the efficiency is high. Hence the analysis given here is not valid for "low-efficiency" iso-efficiency curves

[7]Since the value of T_{comm} used in the analysis is only a lower bound, the actual iso-efficiency function can be worse than exponential.

[8]Since, N and W are plotted on a logarithmic scale, a polynomial growth of W w.r.t. N would have resulted in a linear curve.

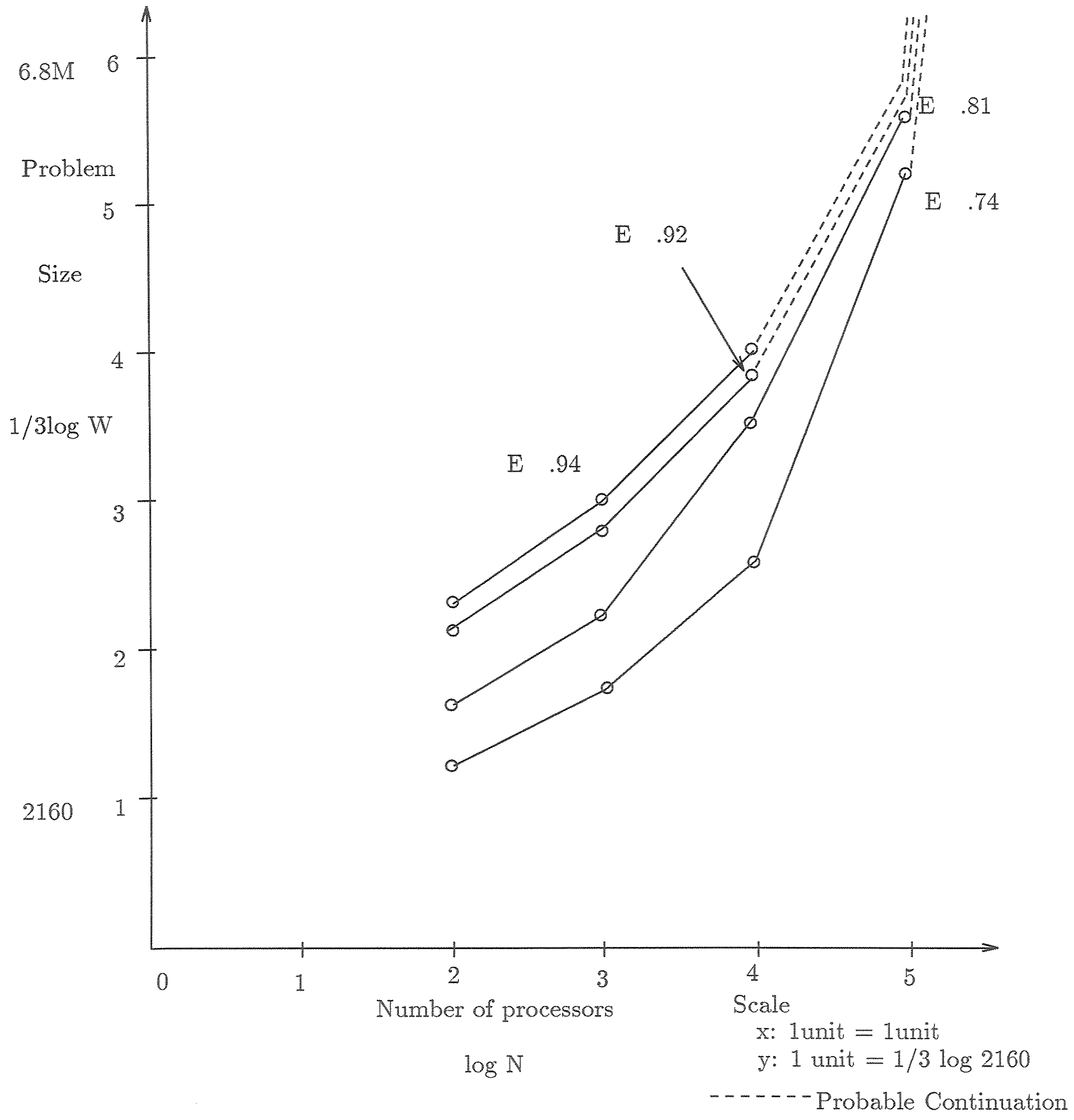


Figure 5: Isoefficiency curves for 1-ring. For each curve, E denotes the efficiency.

5.4 An Improved work distribution strategy for ring

In the previous work distribution scheme, we restricted communication to occur only between immediate neighbors of the ring. The analysis in the previous section clearly indicates a weakness due to this: the total count of stack transfers grows exponentially in a ring of processors because the size of the work pieces coming into successive processors decreases geometrically (in the ratio $1, \alpha, \alpha^2, ...$). To solve this problem, we designed the following work distribution scheme.

In this scheme, we designate a special processor that selects the target for each requesting processor. This special processor maintains a variable I whose value denotes the next donor processor. Whenever a processor needs work, it sends a message to the special processor, which returns the current value of I and also increments it. The requesting processor now sends a request for work to processor I.

The code of the special processor and GETWORK are sketched below:

```
SPECIAL PROCESSOR
begin
  I = 0;
  while (not terminated)
   receive((TARGET_REQUEST, requester_id));
   while((requester_id = I) or (I = special_processor_id))
    {
     I = I +1;
    }
   send(requester_id, I);
   I = I+1;
end

GETWORK()
begin
send(special_processor_id,(TARGET_REQUEST, my_id));
receive(target_id);
send(target_id,(WORK_REQUEST,my_id));
receive(WORK);
if(WORK != null) then return(SUCCESS);
end
```

This work distribution scheme appears to have lots of overhead, as even to decide the identity of the next donor, a processor needs to wait for O(N) time. Actual request for work and receiving a response again takes O(N) time. On the other hand, requesting work and receiving a response takes only a constant time in the simple scheme. But, as the analysis of the next section shows, the new scheme needs to make far fewer requests for work. Therefore it has a substantially better iso-efficiency function and speedup performance.

5.5 Iso-efficiency analysis of the Improved Scheme

Let ϵ be the minimum amount of work transferable. (The absolute minimum amount of work transferable is one node expansion. If we give out work only from levels that are above a CUTOFF depth, then ϵ can be increased by increasing the CUTOFF.) We now present an upper bound on the number of stack transfers.

Let us assume that in every V(N) requests made for work, every processor in the system is requested at least once. Clearly, $V(N) \geq N$. In general, V(N) depends on the load balancing algorithm. Recall that in a transfer, work (w) available in a processor is split into two parts (αw and $(1-\alpha)$w), and one part is taken away by the requesting processor. Hence after a transfer, neither of the two processors (donor and requester) have more than $(1-\alpha)$w work (assuming $\alpha \leq 0.5$). The process of work transfer continues until work available in every processor is less than ϵ. Initially processor 0 has W units of work, and all other processors have no work.

After $V(N)$ requests, maximum work available in any processor is less than $(1-\alpha)W$
After $2V(N)$ requests, maximum work available in any processor is less than $(1-\alpha)^2W$

.

.

.

After $(log_{\frac{1}{1-\alpha}}\frac{W}{\epsilon})V(N)$ requests, maximum work available in any processor is less than ϵ.

Hence, the total number of transfers $\leq V(N)log_{\frac{1}{1-\alpha}}\frac{W}{\epsilon}$

$$T_{comm} \simeq U_{comm} * V(N)log_{\frac{1}{1-\alpha}}\frac{W}{\epsilon}\ (upper\ bound)$$

$$T_{calc} = U_{calc}W$$

$$Efficiency = \frac{1}{1+\frac{T_{comm}}{T_{calc}}}$$

$$= \frac{1}{1+\frac{U_{comm}*V(N)log_{\frac{1}{1-\alpha}}\frac{W}{\epsilon}}{U_{calc}*W}}$$

For the improved work distribution scheme, V(N) = N, and $U_{comm} = O(N)$. Hence for iso-efficiency,

$$W \sim O(N^2)logW\ or\ W \sim O(N^2logN)$$

This iso-efficiency function is much better than β^N. We implemented the scheme and tested it's performance. As shown, in the Fig. 6, the speedups are substantially higher than the previous scheme.

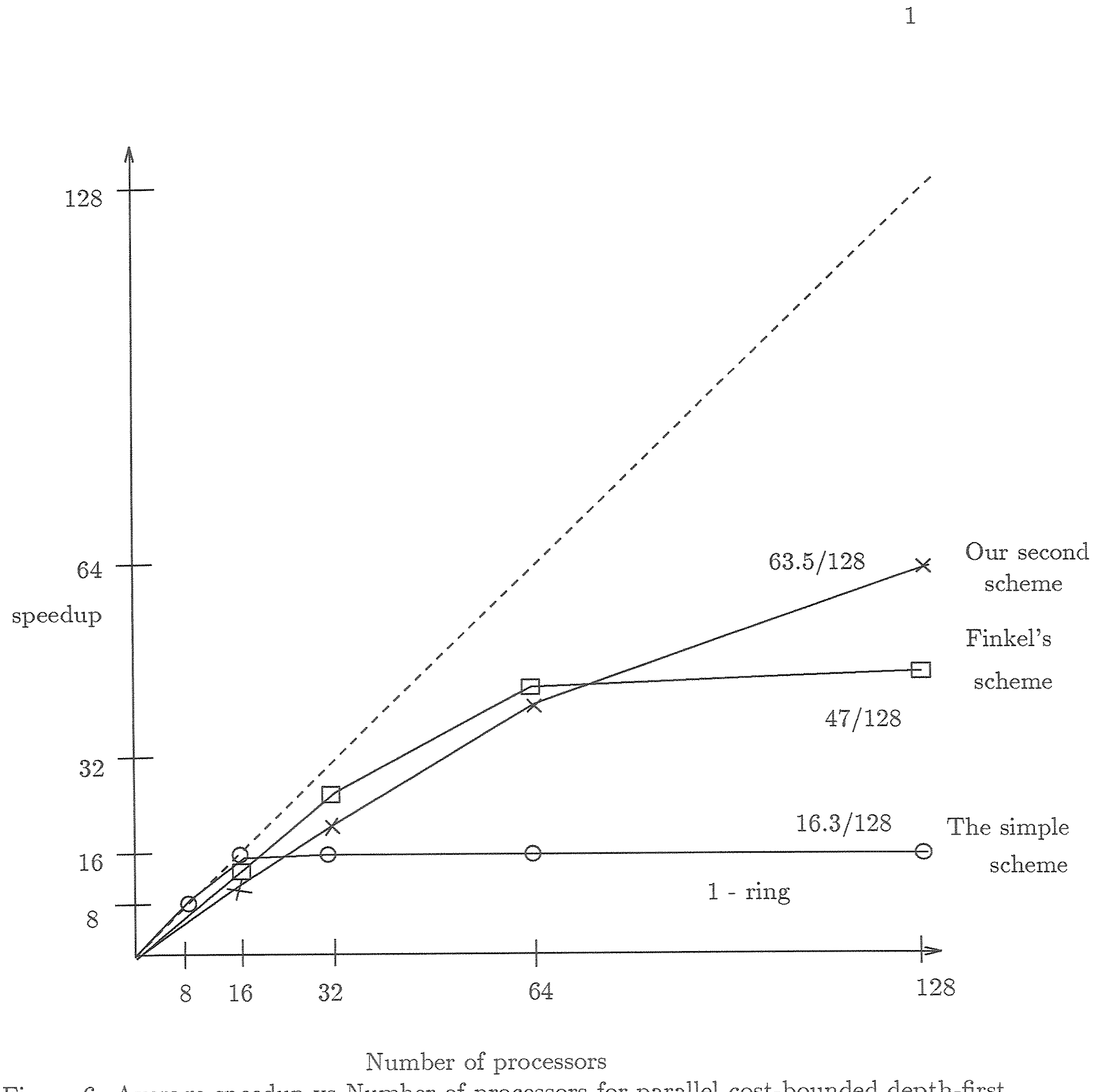


Figure 6: Average speedup vs Number of processors for parallel cost-bounded depth-first search on a ring embedded in Intel Hypercube. Sequential Exec. time $\simeq$ 10500 secs, problem size $\simeq$ 9 M nodes

5.6 Finkel and Manber's Scheme

Finkel and Manber used a different work distribution scheme in their implementation of parallel depth-first search [5]. In their scheme, each processor maintains a local variable, target, to point to a donor processor. Target is incremented (modulo N) every time the processor seeks work. We can compute the iso-efficiency function of this scheme by following the method in section 5.5. For this scheme, V(N) = N^2 in the worst case.[9] But U_{comm} is still O(N). Hence the iso-efficiency function can be as bad as $O(N^3 logN)$.

The superiority of our improved work-distribution scheme over this and the first scheme is clearly seen in the speedup curves of Fig 6. Initially our second scheme is slightly worse than the other two schemes due to the extra overhead of requesting the value of target before requesting for work. But, for larger number of processors, our second scheme makes sufficiently fewer requests than the other schemes, and hence gives higher speedups.

6 Related Research.

Many researchers have implemented parallel DFS on the ring architecture and studied its performance for around 16–20 processors. Vornberger[23], Monien[25] and Wah[24] present parallel depth-first search procedures on a ring network. The work distribution schemes in these formulations is very similar to the first scheme presented in this paper. From the analysis of Section 5.3 (and our experiments) it is clear that this work distribution scheme is not able to provide good speedup on large rings. The initialization part in Monien's[25] and Wah's[24] scheme is slightly different than the one discussed in this paper. Before starting parallel search they divide the search space into N parts, and give each part to a processor. If the initial distribution is quite good, then good speedup can be obtained even with the simple work distribution scheme. But good distribution can be difficult to obtain especially for large problems and large number of processors. To test this we modified parallel DFS to start with an initial distribution of work. We found the performance improvements to be minimal.

Manber presents an abstract model in [13] that captures the distribution of work being done in parallel depth-first search. For this model, Manber presents different work distribution schemes and computes lower bounds on the amount of interference in a shared-memory system. (Part of the analysis presented in Section 5.5 uses the same technique that Manber used for the analysis of interference). Manber's analysis served as a basis for the design of parallel depth-first search scheme presented by Finkel and Manber in [5]. This scheme has a better iso-efficiency function ($O(N^3 \log W)$ worstcase) for the ring than the simple work distribution scheme (see section 5.6). But this function is worse than the iso-efficiency

[9]This result was proved by Manber[13] while analyzing the memory interference in shared memory architectures.

function ($N^2 logW$) of the improved scheme in Section 5.4. Superiority of our improved scheme is clearly seen in the speedup curves of Fig 6.

Kumar and Kanal [11] present a parallel formulation of depth-first search in which different processors search the space with different expectations (cost bounds). At any time, at least one processor has the property that if it terminates, it returns an optimal solution; the other processors conduct a look-ahead search. This approach requires very little communication between processors, but the maximum speedup obtained is problem dependent. The work done by each processor in this scheme can be executed in parallel using our work distribution scheme to give additional speedup.

Janakiram et. al. [8] present a parallel formulation in which different processors search the space in different (random) orders. The speedup in this scheme is dependent upon the probability distribution of solutions in the search tree. A major feature of this scheme is that it is fault tolerant, as any single processor is guaranteed to find a solution. As with Kumar and Kanal's scheme, the work done by each processor in this scheme can be executed in parallel using our work distribution method.

One of the first parallel implementations of depth-first search was proposed by Imai et al [7] . In this scheme, the search tree is maintained as a shared data structure, and different processors remove and expand one node at a time in depth-first fashion. This scheme requires a shared memory and hence is not suited for a distributed-memory architecture such as the ring.

7 Conclusions.

We have presented experimental and analytical evaluation of a number of work distribution schemes used in parallel depth-first search on the ring architecture. We found that the choice of the work distribution algorithm has a significant impact on the performance of the parallel depth-first search algorithm. We have introduced the concept of iso-efficiency function to characterize the effectiveness of different work distribution schemes. Table 1 shows iso-efficiency functions of parallel depth-first search for different work distribution schemes. The development of the new work distribution scheme for the ring was motivated by the iso-efficiency analysis of the other two work distribution schemes. Even though, the new scheme appeared to have a lot of overhead, it had a better iso-efficiency function, and was expected to perform better on larger rings. We were pleased to find that the experimental results were in close agreement with our theoretical results.

The performance of parallel depth-first search is also greatly dependent upon the architecture. Our experimental results and iso-efficiency analysis shows that the hypercube and shared-memory architectures are significantly better than the ring. In[16] we also present a work distribution scheme that has almost optimal performance on shared-memory/ω-network-with- message-combining architectures (such as RP3[4], Ultracomputer[3]). The

Iso-efficiency Function	*Load balancing scheme*
β^N	Section3.1,Wah[24],Vornberger[23]
$N^3 logN$	Finkel and Manber[5]
$N^2 logN$	The improved scheme (Section5.4)

Table 1: Iso-efficiency functions of different work-distribution schemes.

iso-efficiency function has been found useful in characterizing the scalability of many other parallel algorithms as well.

Acknowledgements: We would like to thank Ralph Brickner and Randy Michelson of Los Alamos National Lab for providing access to a 128-processor Intel Hypercube. Mohamed Gouda and Dan Miranker provided useful comments on an earlier draft of the paper.

References

[1] O. Vornberger B. Monien and E. Spekenmeyer. **Superlinear Speedup for Parallel Backtracking**. Technical Report 30, Univ. of Paderborn, FRG, 1986.

[2] E.W. Dijkstra, W.H. Seijen, and A.J.M. Van Gasteren. Derivation of a termination detection algorithm for a distributed computation. **Information Processing Letters**, 16–5:217–219, 1983.

[3] A. Gottlieb et al. The NYU ultracomputer - designing a mimd, shared memory parallel computer. **IEEE Transactions on Computers**, 175–189, February 1983.

[4] G. F. Pfister et al. The IBM research parallel processor prototype (rp3). In **Proceedings of International conference on Parallel Processing**, pages 764–797, 1985.

[5] Raphael A. Finkel and Udi Manber. Dib - a distributed implementation of backtracking. **ACM Trans. of Progr. Lang. and Systems**, 9 No. 2:235–256, April 1987.

[6] Ellis Horowitz and Sartaj Sahni. **Fundamentals of Computer Algorithms**. Computer Science Press, Rockville, Maryland, 1978.

[7] M. Imai, Y. Yoshida, and T. Fukumura. A parallel searching scheme for multiprocessor systems and its application to combinatorial problems. In **IJCAI**, pages 416–418, 1979.

[8] Virendra K. Janakiram, Dharma P. Agrawal, and Ram Mehrotra. Randomized parallel algorithms for prolog programs and backtracking applications. In **Proceedings of International conference on Parallel Processing**, pages 278–281, 1987.

[9] R.E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. **Artificial Intelligence**, 27:97–109, 1985. Also a chapter in 'Search and Artificial Intelligence',Vipin Kumar and Laveen Kanal,(editors), Springer-Verlag,1987(in press).

[10] Vipin Kumar. Depth-first search. In Stuart C. Shapiro, editor, **Encyclopaedia of Artificial Intelligence: Vol 2**, pages 1004–1005, John Wiley and Sons, Inc., New York, 1987.

[11] Vipin Kumar and Laveen N. Kanal. Parallel branch-and-bound formulations for and/or tree search. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, PAMI-6, November 84.

[12] T. H. Lai and Sartaj Sahni. Anomalies in parallel branch and bound algorithms. In **Proceedings of International conference on Parallel Processing**, pages 183–190, 1983.

[13] Udi Manber. On maintaining dynamic information in a concurrent environment. **SIAM J. of Computing**, 15 No. 4:1130–1142, 1986.

[14] Bernard A. Nadel. Constraint satisfaction algorithms. In Vipin Kumar and Laveen Kanal, (editors), **Search and Artificial Intelligence**, Springer-Verlag, New York, 1988(in press).

[15] V. Nageshwara Rao, and Vipin Kumar Parallel Depth-First Search on Multiprocessors, Part I: Implementation. To appear in **International Journal of Parallel Programming**, 1988.

[16] Vipin Kumar and V. Nageshwara Rao Parallel Depth-First Search on Multiprocessors, Part II: Analysis. To appear in **International Journal of Parallel Programming**, 1988.

[17] V. Nageshwara Rao, Vipin Kumar, and K. Ramesh. A parallel implementation of iterative-deepening-a*. In **AAAI**, pages 878–882, 1987. Also AI Lab TR 87-46, University of Texas at Austin, January 87.

[18] Nils J. Nilsson. **Principles of Artificial Intelligence**. Tioga Press, 1980.

[19] Judea Pearl. **Heuristics - Intelligent Search Strategies for Computer Problem Solving**. Addison-Wesley, Reading, MA, 1984.

[20] Charles Seitz. The cosmic cube. **Commun.ACM**, 28-1:22–33, 1985.

[21] M.E. Stickel and W.M. Tyson. An analysis of consecutively bounded depth-first search with applications in automated deduction. In **IJCAI**, pages 1073–1075, 1985.

[22] M. H. van Emden. An interpreting algorithm for prolog programs. In J.A. Campbell, editor, **Implementations of Prolog**, Ellis Horwood, West Sussex, England, 1984.

[23] Olivier Vornberger. **Implementing Branch-and-Bound in a Ring of Processors**. Technical Report 29, Univ. of Paderborn, FRG, 1986.

[24] Benjamin W. Wah and Y. W. Eva Ma. Manip - a multicomputer architecture for solving combinatorial extremum-search problems. **IEEE Transactions on Computers**, c–33, May 1984.

[25] Monien B. and Vornberger O. The Ring Machine. Technical Report No. 27, Dept. of Math./Computer Science, University of Paderborn, Dec. 1985, to appear in Computers and Artificial Intelligence, 3(1987).

[26] Guo-Jie Li and Benjamin W. Wah. Coping with anomalies in parallel branch-and-bound algorithms. *IEEE Trans on Computers*, C-35, June 1986.

[27] V. Nageshwara Rao and V. Kumar. Superlinear speedup in depth-first search. In *Submitted for publication*, 1988. Also AI Lab TR, University of Texas at Austin, March 88.

[28] V. Kumar, K. Ramesh and V. Nageshwara Rao. Parallel Heuristic Search of State-Space Graphs: A Summary of Results. In *Submitted for publication*, 1988. Also AI Lab TR, University of Texas at Austin, March 88.