# Parallel Depth First Search, Part II: Analysis*

Vipin Kumar[†] and V. Nageshwara Rao

Department of Computer Sciences,
University of Texas at Austin,
Austin, Texas 78712

**Abstract**

This paper presents the analysis of a parallel formulation of depth-first search. At the heart of this parallel formulation is a dynamic work-distribution scheme that divides the work between different processors. The effectiveness of the parallel formulation is strongly influenced by the work-distribution scheme and the target architecture. We introduce the concept of isoefficiency function to characterize the effectiveness of different architectures and work-distribution schemes. Many researchers considered the ring architecture to be quite suitable for parallel depth-first search. Our analytical and experimental results show that hypercube and shared-memory architectures are significantly better. The analysis of previously known work-distribution schemes motivated the design of substantially improved schemes for ring and shared-memory architectures. In particular, we present a work-distribution algorithm which guarantees close to optimal performance on a shared-memory/$\omega$-network-with-message-combining architecture (e.g. RP3). Much of the analysis presented in this paper is applicable to other parallel algorithms in which work is dynamically shared between different processors (e.g., parallel divide-and-conquer algorithms). The concept of isoefficiency is

*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.

[†]Arpanet: kumar@sally.utexas.edu

useful in characterizing the scalability of a variety of parallel algorithms.

**Key words:** parallel algorithm, depth-first search, isoefficiency function, work distribution schemes

# 1 Introduction

This paper presents the analysis of our parallel formulation of depth-first search presented in [12]. At the heart of this parallel formulation is a dynamic work-distribution scheme that divides the work between different processors. The effectiveness of the parallel formulation is strongly influenced by the work-distribution algorithm and architectural features such as presence/absence of shared memory, the diameter of the network, relative speed of the communication network, etc.

We present the notion of isoefficiency function as a figure of merit to evaluate parallel algorithms. We analyze the isoefficiency functions of the work-distribution schemes for ring, hypercube and shared-memory architectures, and validate the analysis via experiments. We also present improved work-distribution schemes for ring and shared-memory architectures. The development of these new schemes was motivated by the analysis of the earlier schemes. From our analysis, it is clear that on suitable architectures, it is feasible to speedup depth-first search by several orders of magnitude. Experimental validation of the analysis was done by parallelizing the IDA* algorithm [3, 4] to solve the 15-puzzle problem[9] on BBN Butterfly[1], Intel Hypercube iPSC/1[2] and a ring embedded in the Intel Hypercube. Our analysis of parallel DFS is also applicable to other parallel algorithms in which work is shared dynamically among processors.

Section 2 gives a brief review of a parallel formulation of DFS. Section 3 states assumptions and definitions needed for the analysis. Section 4 introduces the isoefficiency function as a metric for the scalability of a parallel algorithm. Sections 5, 6, 7 present the isoefficiency analyses of the commonly used work-distribution schemes in parallel DFS on ring, hypercube and shared-memory architectures. In Section 7, we also present an improved

[1]Butterfly is a trademark of BBN Advanced Computers Inc.

[2]iPSC is a trademark of Intel Scientific Computers

work-distribution scheme for the shared-memory architecture. In section 8, we present an improved work-distribution scheme for the ring architecture and show that it has a better isoefficiency function as well as speedup performance than the other known schemes for the ring. Section 9 reviews previous work on the analysis of parallel depth-first search. Section 10 contains concluding remarks.

# 2 A Parallel Formulation of Depth-First Search

We parallelize depth-first search by distributing the work to be done among a number of processors. Each processor searches a disjoint part of the search space in a depth-first fashion. When a processor finishes searching its part of the search space, it tries to get an unsearched part of the search space from other processors. When a solution path (i.e., a path from the initial node to a goal node) is found, all of them quit. If the search space is finite and has no goal nodes, then eventually all the processors would run out of work, and the (parallel) search will terminate. We assume that, at the start of each iteration, all the search space is assigned to one processor, and other processors are given null spaces. From then on, the search space is dynamically divided and distributed among various processors.

Since each processor searches the space in a depth-first manner, the (part of) state-space tree 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 its local stack is empty, the processor tries to get some of the untried alternatives from the stack of another processor.

In the formulation we implemented in [12], an idle processor tries to get work (in a round-robin fashion) only from its immediate neighbors; i.e., in a 2-ring, it can get work from any of its two neighbors; in a 1-ring, it can get work from only one neighbor; in a hypercube, it can get work from $\log N$ neighbors; and in a shared-memory architecture, it can get work from any of the $N$ processors in the system. These are simple and intuitive work-distribution schemes for the respective architectures and have been used by many researchers. Other work-distribution schemes are possible, and will be considered later.

# 3 Definitions and Assumptions

## 3.1 Assumptions

We assume that the search space of the problem being solved is bounded. This is true of most practical problems solved by DFS. If the search space is not bounded (or is very deep), then simple DFS may never terminate (or take a very long time). Note that our analysis is applicable for iterative-deepening depth-first search algorithms (e.g., IDA* [3, 4]) even if the search space is not bounded. The reason is that each iteration of these algorithms performs depth-first search in a bounded part of the search space. To simplify the analysis (i.e., to avoid dealing with speedup anomalies [13, 5]) we assume that both sequential and parallel DFS search the whole bounded space for all solution paths. In the case of IDA* (sequential or parallel), it means that all optimal (i.e., least cost) solution paths need to be found. The possibility of superlinear speedup in our parallel formulation of depth-first search is discussed in [13]. We assume that the effective branching factor (defined below) of the search space is greater then $1 + e$ (where $e$ is an arbitrarily small 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 $\alpha W$ for some constant $\alpha$ such that $0 < \alpha \leq 0.5$. This assumption simply says that the splitting function is not unreasonable.

All these assumptions are satisfied by the cost-bounded DFS (i.e., the last iteration of IDA*) presented in Section 4.4 of [12]. This algorithm was used to solve the 15-puzzle problem in all the experiments discussed in this paper.

## 3.2 Definitions

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

2. Effective-Branching Factor $b$: is defined as the average number of successors of the nodes of the search tree. If the depth of the search tree is d, then the effective-branching factor $b$ is approximately $W^{\frac{1}{d}}$.

3. Number of processors $N$: is the number of processors being used to run parallel DFS. $P_i$ denotes the ith processor.

4. Running time $T_N$: is the execution time on $N$ processors. $T_1$ is the sequential execution time. We assume that $T_1$ is proportional to $W$.

5. 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 bounded space to find all solution paths (see assumptions above),

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

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

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

7. 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.

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

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

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

10. Unit Communication time $U_{comm}$: is the mean time taken for getting some work (a stack) from a neighboring processor. $U_{comm}$ depends upon the size of the message transferred (which depends upon the actual splitting strategy used [12]), the distance between the donor and the requesting processors, and the communication speed of the

underlying hardware. For simplicity, in our analysis, we assume that the message size is fixed. Even if we assume that the size of the message grows as $O(\log W)$ (which is a better approximation for the splitting strategy used in our implementation in [12]), the results change only slightly.

# 4 The Isoefficiency Function

The efficiency (and speedup) achieved in parallel DFS is determined by the architecture, the work-distribution algorithm, the number of processors and the problem size. 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 (for the work-distribution schemes used in [12]) $T_{comm}$ does not increase proportionately. (For example, see the speedup curve for the Intel Hypercube in [12]). If $N$ is increased, then we can keep the efficiency fixed (i.e., maintain the speedup to be linear) by increasing $W$. The rate of increase of $W$ with respect to (w.r.t.) $N$ is dependent upon the architecture and the work-distribution algorithm.

In many parallel algorithms (e.g., parallel DFS, parallel 0/1 knapsack[6], parallel algorithms for the shortest path problem[11], parallel quicksort[11]), it is possible to obtain linear speedup on arbitrarily many processors by simply increasing the problem size (i.e., the sequential execution time $W$). The required rate of growth of $W$ w.r.t. $N$ (to keep the efficiency fixed) essentially determines the scalability of the parallel algorithm (for a specific architecture). 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 parallel algorithm can easily deliver linear speedup for arbitrarily large $N$ (provided a large enough architecture can be constructed). Since most problems have a sequential component (in DFS, it is one node expansion), asymptotically, $W$ must grow at least linearly to maintain a particular efficiency. If $W$ needs to grow as $f(N)$ to maintain an efficiency E, then $f(N)$ is the **isoefficiency function** and the plot of $f(N)$ w.r.t. $N$ is the **isoefficiency curve**.

Next we derive isoefficiency functions of parallel bounded DFS for shared-memory architectures and distributed-memory architectures (hypercube, ring). 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 architectures and work-distribution schemes.

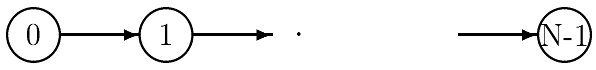


Figure 1: A Linear Chain of processors

These bounds on $T_{comm}$ are used to compute bounds on the isoefficiency functions. Predictions from our models seem to closely agree with experimental data, hence we feel that the models are reliable. Experimental isoefficiency curves were obtained by performing a large number of experiments for a range of $W$ and $N$, and collecting the points with equal efficiency.

In parallel DFS, the overhead is primarily due to dynamic work distribution. In our analysis we try to estimate the number of stack transfers that occur for each work-distribution scheme and the architecture. Since stack transfers form the most significant part of communication,

$T_{comm} \simeq U_{comm}*$number of stack transfers.

# 5 A Model for the 1–Ring Architecture.

Here we analyze the work-distribution scheme for 1-rings, in which a processor can request work from only one of its two immediate neighbors. Consider a linear chain of $N$ processors of Fig. 1. A 1–ring is a linear chain with a fold back from processor $N-1$ to 0. We assume that a processor requests work from its left neighbor (when needed), and sends work to its right neighbor (when a request comes).

Initially $W$ work is available in processor $P_0$. In order to achieve good work-distribution every processor needs to get roughly $\frac{W}{N}$ work for itself[3].

[3]This is clearly true if the efficiency is high. Even for the low-efficiency case, each processor needs to get roughly $k\frac{W}{N}$ work (for $0 < k < 1$). Hence, by following a similar analysis, we can show that the number of stack transfers will grow exponentially with $N$

Recall that when a processor requests a donor which has $w$ work, the work is split into two parts, the smallest of which is at least $\alpha w$. Hence

Maximum piece of work coming into processor $P_0$ is $W$

Maximum piece of work coming into processor $P_1$ is $(1-\alpha)W$

Maximum piece of work coming into processor $P_i$ is $(1-\alpha)^i W$

From the above, we can see that in order to get $\frac{W}{N}$ work, Processor $P_i$ has to get at least $\frac{\frac{W}{N}}{(1-\alpha)^i W}$ transfers.

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

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

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

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

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

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

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

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 constants)

Thus the isoefficiency function is exponential[4] in $N$. The isoefficiency function for 2-ring can be obtained similarly, and is also exponential. Since the value of $T_{comm}$ used in the analysis is only a lower bound, the actual isoefficiency function can be worse than exponential. This explains the poor performance of parallel DFS on large (> 16 processors) 1-ring and 2-ring in [12]. Fig. 2 shows experimentally obtained isoefficiency curves of parallel DFS for 15-puzzle on a 1-ring embedded in the Intel Hypercube. Clearly these curves show exponential growth. Since $N$ and $W$ are plotted on logarithmic scales, a polynomial growth of $W$ w.r.t. $N$ would have resulted in a linear curve.

Note that the CPU and communication speeds of the ring architecture (reflected in the values of $U_{calc}$ and $U_{comm}$) show up only as constants in the isoefficiency function. Hence, irrespective of the hardware quality (which is determined by the state-of-the-art in computer architecture), our simple work-distribution scheme on the ring architecture has (at least) an exponential isoefficiency function. Hence parallel DFS with this work-distribution scheme is not going to be effective on large rings.

# 6 A Model for the Hypercube Architecture.

Here we analyze the work-distribution scheme in which a processor polls its $\log N$ immediate neighbors (in the hypercube) for work. Assume that whenever a processor receives a request for work, it splits its work $w$ into two parts $(1-\gamma)w$ and $\gamma w$, and gives away $\gamma w$. Clearly, $\alpha \leq \gamma \leq 1-\alpha$. Following the arguments of Section 5, a processor at distance i from processor $P_0$ receives pieces of maximum size $\gamma^i W$. To get $\frac{W}{N}$ work, number of requests made by a processor at distance i from from processor $P_0$.

$$\geq \frac{\frac{W}{N}}{\gamma^i W}$$

$$= \frac{1}{N\gamma^i}$$

$$= \frac{\beta^i}{N} \; where \; \beta = \frac{1}{\gamma}$$

[4]If the stack transfer time is taken to be $O(\log N)$ (instead of $O(1)$), then the isoefficiency function is $\Omega(\frac{\beta^N \log N}{N})$, which is still exponential.

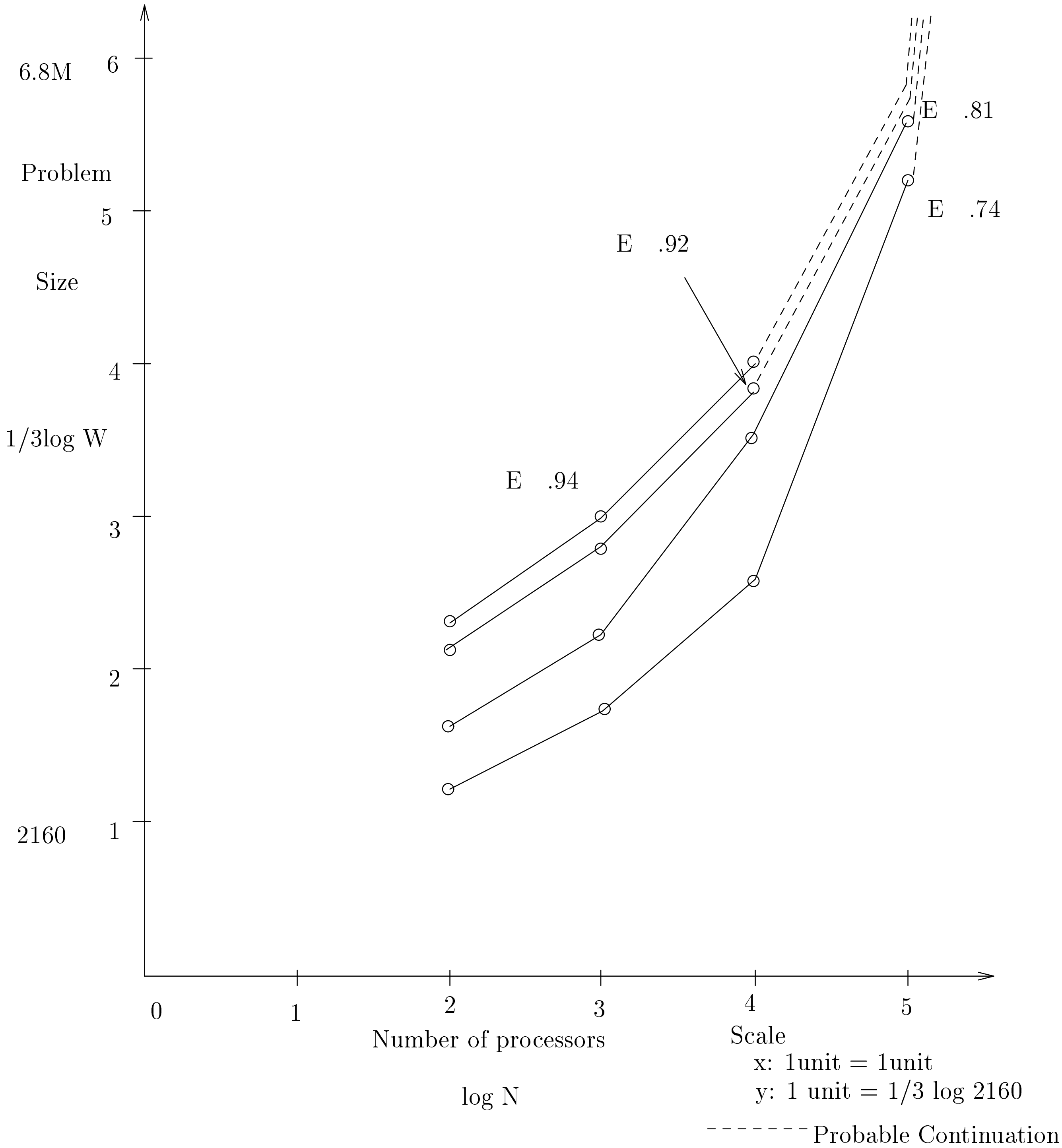


Figure 2: Experimental isoefficiency curves of parallel DFS (with the simple work-distribution scheme) on a 1-ring embedded in the Intel Hypercube.

Since there are ${}^{\log_2 N}C_i$ processors at distance i from processor 0 in a hypercube, the total number of requests in the whole system

$$\geq \sum_{i=1}^{i=\log_2 N} {}^{\log_2 N}C_i \, \frac{\beta^i}{N}$$

$$= \frac{1}{N}\,(1+\beta)^{\log_2 N}$$

$$= (\frac{1+\beta}{2})^{\log_2 N}$$

$$= N^{\log_2 \frac{1+\beta}{2}}$$

Hence

$$T_{comm} = U_{comm} N^{\log_2 \frac{1+\beta}{2}} \ (lower\ bound)$$

We know

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

Hence

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

$$= \frac{1}{1 + \frac{U_{comm} N^{\log_2 \frac{1+\beta}{2}}}{U_{calc} W}}$$

For an isoefficiency curve on the hypercube,

$$W = \Omega(N^{\log_2 \frac{1+\beta}{2}}) \tag{1}$$

Equation 1 says that for a hypercube if $\gamma \leq \frac{1}{3}$ (i.e., $\beta \geq 3$), then the problem size needs to grow polynomially with the number of processors to maintain the efficiency. For $\gamma \geq \frac{1}{3}$, Equation 1 suggests a sublinear isoefficiency curve. But note that Equation 1 provides only a lower bound on the growth of the isoefficiency function. Also, we expect peak performance when $\gamma$ is roughly equal to $\frac{1}{2}$ for the following reason. If the donor gives too much work, then the donor will be out of work too soon, if the donor gives too little work, then the requester will be out of work too soon. Except for a brief work-distribution period in the beginning, every processor is equally likely to receive requests for work, as the hypercube architecture is homogeneous. Hence every processor should try to give out nearly half of its work. Hence as $\gamma$ is increased beyond 0.5, the performance should degrade just as it would when $\gamma$ is decreased below 0.5. This is confirmed by our experiments with parallel DFS on 15-puzzle.

Fig. 3 shows experimentally obtained isoefficiency curves for parallel DFS for the 15-puzzle problem on the Intel Hypercube. $N$ and $W$ are plotted on logarithmic scales. In these experiments, the third splitting strategy given in Section 3.2.1 of [12], was used, which tries to keep $\gamma$ close to 0.5.[5] Clearly the isoefficiency function even for this case has a polynomial growth. From these, we empirically see that

$$W \sim cN^{1.59} \quad (\log_2 3 = 1.59; \beta \leq 5; \gamma \geq \frac{1}{5})$$

# 7 A Model for the Shared-Memory Architecture

Here we first derive an upper bound on the total number of work transfers and the isoefficiency function for a rather general situation . These bounds are valid for any work-distribution scheme in which (i) work is requested and transferred only when a processor is idle; (ii) the smallest of two work pieces after splitting work $w$ is $\alpha w$, and $\alpha \geq 0$; (iii) Work is split (and a part given out) only if it is greater than some minimum amount $\epsilon$.[6]

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 work-distribution algorithm. Recall that in a transfer, work (w) available in a processor is split into two parts, and one part is taken away by the requesting processor. Hence after a transfer neither of the two processors (donor and requester) has more than $(1-\alpha)$w work (because the smallest part is at least $\alpha w$). The process of work transfer continues until work available in every processor is less than $\epsilon$. Initially Processor $P_0$ 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 $(1-\alpha)^2W$

[5] Due to the nonuniform structure of the search tree, there is no guarantee that $\gamma \simeq 0.5$.

[6] As discussed in [12], untried alternatives are transferred from the stack of the donor processor to the requester processor only if they are above a user specified level called cutoff depth. This ensure that the size of the work given out by a donor is at least (roughly) $b^{cutoff}$. Even otherwise, the minimum amount of work transferred is one node.

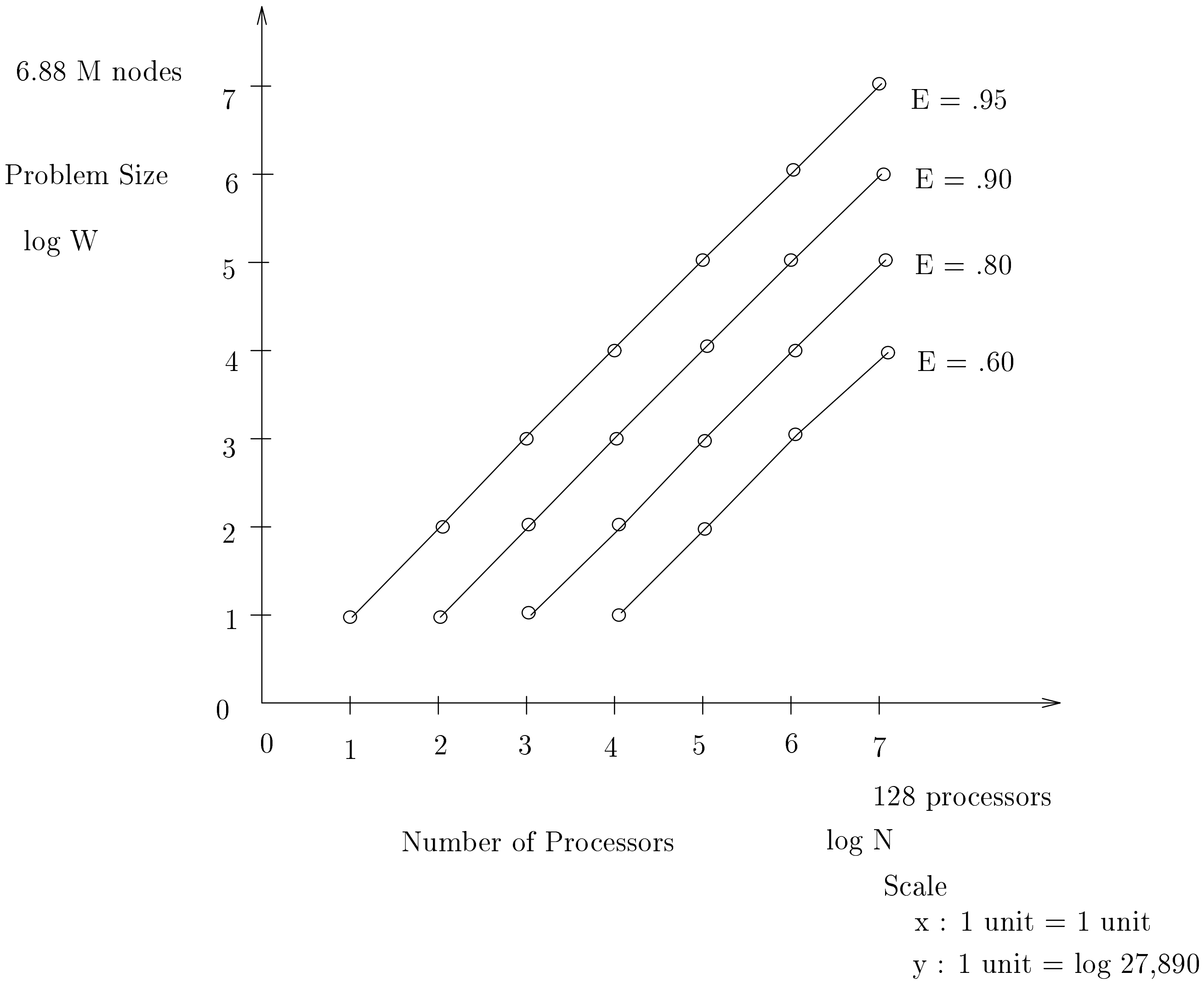


Figure 3: Experimental isoefficiency curves of parallel DFS on the Intel Hypercube

.
.
.
After $(\log_{\frac{1}{1-\alpha}} \frac{W}{\epsilon})V(N)$ requests, maximum work available in any processor is less than $\epsilon$.
Hence the total number of transfers $\leq V(N) \log_{\frac{1}{1-\alpha}} W$

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

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

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

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

Solving this for isoefficiency gives us the relation -

$$W = O(U_{comm} V(N) \log V(N)) \tag{2}$$

Note that the formula expressing $W$ in terms of $V(N)$ is an approximation.

### 7.1 Isoefficiency Function of the Simple Work-Distribution Scheme.

In the work-distribution scheme for the shared-memory architecture implemented in [12], each processor maintains a local variable 'target' to point to a donor processor. The variable target is incremented (modulo $N$) every time the processor seeks work. For this work-distribution algorithm, $V(N) = N^2$ in the worst case. (This result was proved by Manber in a somewhat different context[7]). Thus from Equation 2, the isoefficiency function is $O(N^2 \log N)$. In deriving this expression we assumed that $U_{comm} = O(1)$. If we assume that $U_{comm} = O(\log_b W)$, then the isoefficiency function is $O(N^2 \log^2 N)$.

Note that this isoefficiency function is worse than the one for the hypercube architecture (although, the function for hypercubes is a lower bound, whereas the function for shared-memory architectures is an upper bound). But, even the experimental isoefficiency curves for BBN Butterfly (which is a shared memory architecture) appear to be worse than those for the Intel Hypercube (see figures 4 and 3), and for large enough $N$, the speedup on the Intel Hypercube would perhaps be better than the speedup on BBN Butterfly. This is rather surprising, as BBN Butterfly has a much better $U_{calc}/U_{comm}$ ratio, and has a much smaller diameter than the Intel Hypercube. Clearly, the poor isoefficiency function of the shared-memory architecture is due to its work-distribution scheme (in which each processor independently polls the other processors for work in a round-robin fashion).

## 7.2 An Improved Work-Distribution Strategy for the Shared-Memory Architecture

Let us modify the work-distribution algorithm as follows. Let TARGET be a global counter maintained to point at the next donor processor. Whenever a process is hungry and needs work, it reads the value of TARGET ( to get the donor's identity) and increments TARGET (modulo $N$). Since many processors may be reading TARGET simultaneously, the read-and-increment operation should be atomic. If work is not available from the assigned donor, then it again reads the global variable TARGET and increments it[7] . This scheme guarantees that $V(N) = N$. Now, for constant efficiency, $W = O(N \log N)$.

Figure 5 shows the isoefficiency curve of the improved work-distribution scheme on BBN Butterfly. This scheme results in an isoefficiency function that is very close to $N \log N$. The isoefficiency function of the first scheme appears to be better than $N^2 \log N$ ( $V(N) = N^2$ only in the worst case), but significantly worse than $N \log N$. We have also found the speedup performance of the second scheme to be substantially better than the previous scheme.

Although BBN Butterfly allows an efficient atomic-add instruction, access to variable TARGET by all processors can create another bottleneck. In $\frac{WU_{calc}}{N}$ time, up to $N \log_{\frac{1}{1-\alpha}} W$ atomic-add requests are made to TARGET. This means that asymptotically, $W$ should grow as $O(N^2 \log N)$ to avoid contention for TARGET. But for 15-puzzle, this limitation does not take effect for the range of processors we experimented with ( $\leq 120$).

On shared-memory/$\omega$–network architectures that use message combining (e.g. RP3[10], the Ultracomputer[2]), this problem does not arise at all. In such systems, simultaneous atomic-add requests to TARGET are combined at intermediate nodes of $\omega$–network (where

[7] This new work-distribution algorithm is obtained by replacing the second line of GETWORK() in [12] by the line "TARGET = atomic-add(I,1) mod N", and by replacing TARGET for target in the rest of the procedure.

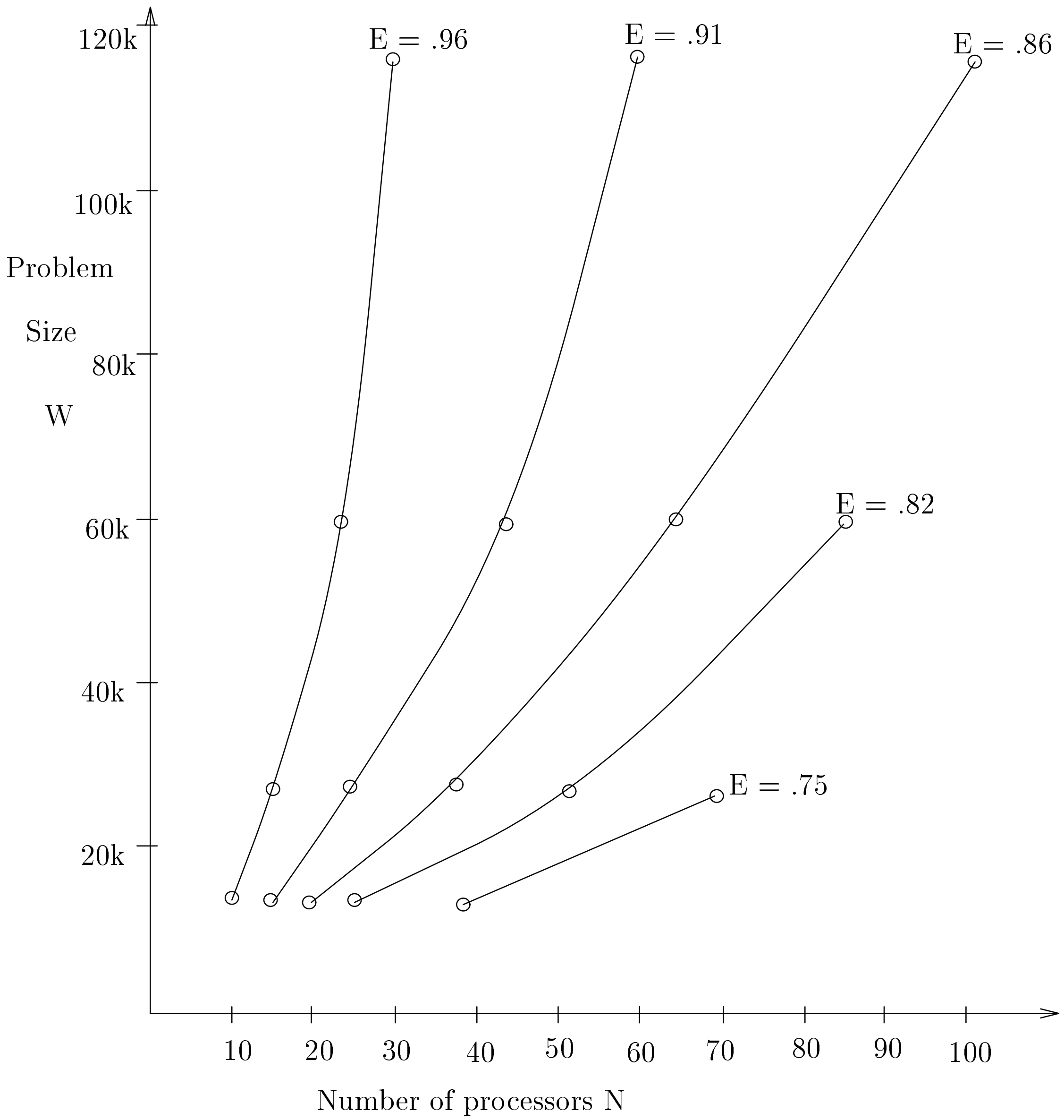


Figure 4: Experimental Isoefficiency curves of parallel DFS on BBN Butterfly for the first work-distribution scheme. E denotes efficiency.

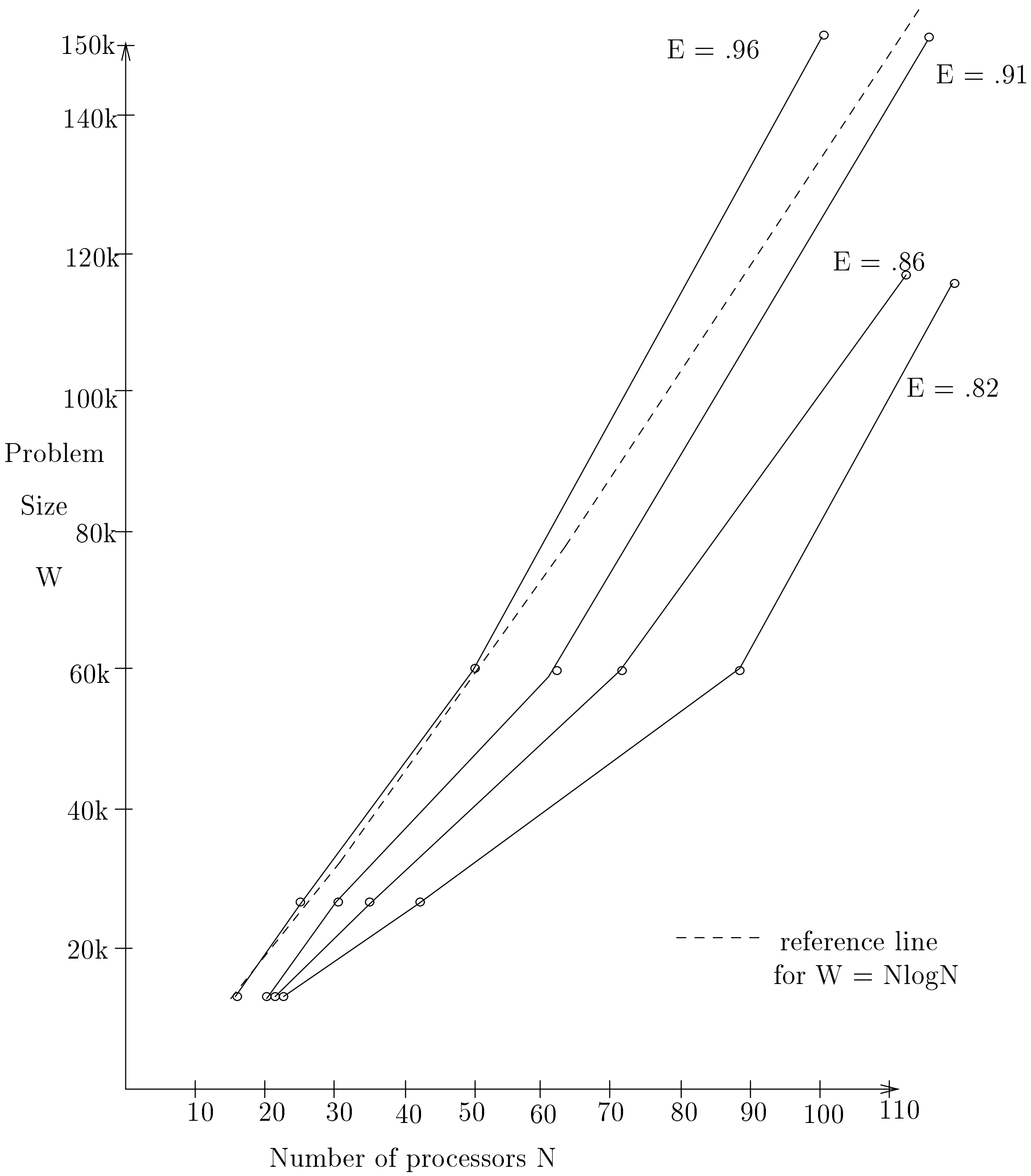


Figure 5: Experimental isoefficiency curves of parallel DFS on BBN Butterfly for the improved work-distribution scheme

they collide). Hence it is possible for all $N$ processors to simultaneously execute atomic-add instruction on the same variable in unit time [8].

Although we don't know whether this new strategy is the best strategy, it clearly has an excellent isoefficiency function, Furthermore, any other scheme can not be much better than this scheme, as the isoefficiency function has to be at least $O(N)$. So the new scheme is within a log factor of the best possible scheme.

# 8 An Improved Work-Distribution Strategy for the Ring Architecture

In the work-distribution scheme of Section 5, we restricted communication to occur only between immediate neighbors of the ring architecture. The analysis of this scheme 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 decrease geometrically ( in the ratio $1, \alpha, \alpha^2, ...$). Clearly this does not happen when work transfer is permitted between any pair of processors (as in the shared-memory architecture). We now adapt the improved work-distribution scheme of Section 7 to the ring architecture by permitting communication between any pair of processors, and analyze its performance.

Recall that communication between processors in the ring architecture involves $O(N)$ hops. Since there is no shared memory, the variable TARGET is maintained in a special processor (one of the processors in the ring architecture). Whenever a processor needs work, it sends a message to this processor, which returns the current value of TARGET and also increments it. Every communication to an arbitrary processor in the ring architecture takes $O(N)$ time (as opposed to constant time in a shared-memory multiprocessor). Hence, from Equation 2, the isoefficiency of this scheme is,

$$W = O(N^2 \log N)$$

This isoefficiency function is much better than $\beta^N$, but still worse than $N \log N$. One may wonder whether the special processor which maintains TARGET would become saturated, as it has to process so many messages. Fortunately this is not the case. The processor maintaining TARGET needs to serve $O(N \log_{\frac{1}{1-\alpha}} W)$ messages in roughly $\frac{WU_{calc}}{N}$ time. The

[8] To be precise it takes $\log N$ time.

isoefficiency term due to this communication bottleneck is also $W = O(N^2 \log N)$. Hence, the overall isoefficiency function is still $W = O(N^2 \log N)$. Note that distributed-memory systems (including hypercubes) cannot obtain better isoefficiency curve than $O(N^2 \log N)$ using this work-distribution scheme because of this communication bottleneck.

Finkel and Manber discuss a number of different work-distribution scheme in their implementation of parallel depth-first search on the ring architecture[1]. In one of their schemes, each processor maintains a **local** variable, target, to point to a donor processor. target is incremented (modulo N) every time the processor seeks work. This can be viewed as an adaptation of our simple work-distribution scheme for the shared-memory architecture to the ring architecture. We can compute the isoefficiency function of this scheme by following the method in Section 7. For this scheme, $V(N) = N^2$ in the worst case. But $U_{comm}$ is still $O(N)$. Hence the isoefficiency function is $O(N^3 \log N)$.

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 a larger number of processors, our second scheme makes substantially fewer requests than the other schemes, and hence gives higher speedups. Isoefficiency functions of other two schemes of Finkel and Manber can be computed similarly, and are also $O(N^3 \log N)$.

# 9 Related Research.

Manber [7] has designed a data structure, called "concurrent pool" that can facilitate work sharing among concurrent processes, and can be incorporated in a parallel depth-first search formulation. Manber presents many different schemes for manipulating concurrent pools and computes lower bounds on the amount of interference (defined as number of 'non local' accesses required). Part of the analysis presented in Section 7 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 in [1]. This scheme has a much better isoefficiency function ($O(N^3 \log W)$ ) for the ring architecture than the one analyzed in Section 5. But this function is significantly worse than the isoefficiency function ($O(N^2 \log W)$) of the improved scheme presented in Section 8.

For shared-memory architectures, Manber presents an algorithm for manipulating con-

1

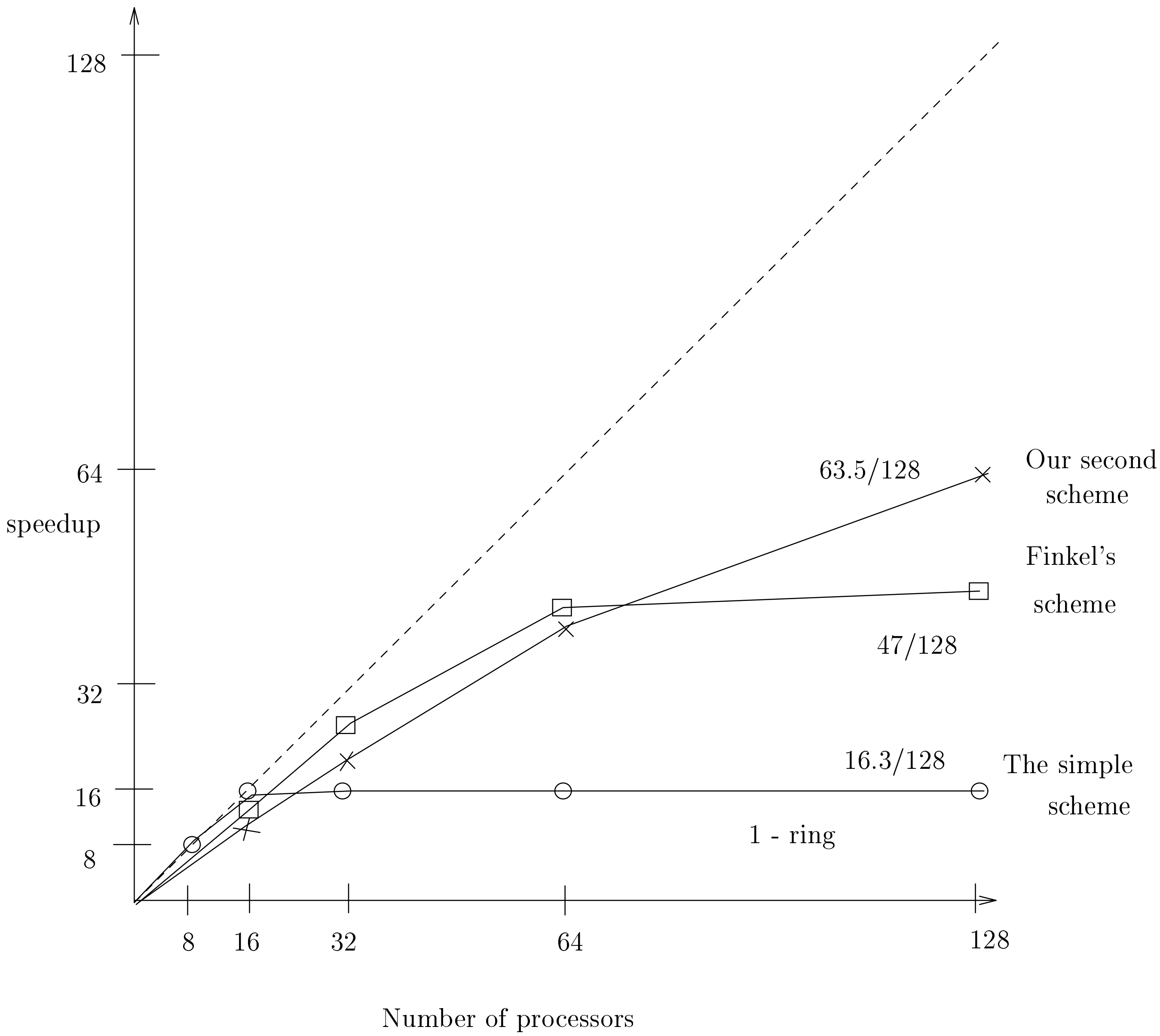


Figure 6: Speedup curves for parallel cost-bounded depth-first search on a ring embedded in the Intel Hypercube. Average problem size $\simeq$ 9 million nodes; sequential Exec. time $\simeq$ 10500 secs

current pools which makes it is possible to obtain isoefficiency function of $O(N^{1+e} \log N)$ for arbitrarily small $e$. But, as $e$ is made smaller, the constant factor in $O(N^{1+e} \log N)$ goes up. In contrast, our second work-distribution method presented in Section 7 guarantees isoefficiency function of $O(N \log N)$ for shared-memory architectures with message combining. Furthermore, the constant factor in $O(N \log N)$ is very small.

Many researchers[14, 1, 8] have considered the ring architecture to be highly suitable for parallel depth-first search. Our analysis shows that the ring architecture (even with the best known work-distribution scheme) has much worse performance than the hypercube or shared-memory architectures.

# 10 Conclusions.

This paper has presented an analysis of different work-distribution schemes used in parallel depth-first search for a variety of architectures. We have introduced the concept of isoefficiency function to characterize the effectiveness of different architectures and work-distribution schemes. The work-distribution schemes used by earlier researchers for the ring architecture were found to be substantially inferior to the one presented in this paper. Furthermore, other researchers[14, 8] considered ring to be quite suitable for parallel depth-first search. Our analytical and experimental results show that hypercube and shared-memory architectures are significantly better. We presented a work-distribution algorithm for the shared-memory/$\omega$-network-with-message-combining architecture (e.g., RP3) which has better performance than previously known algorithms. Table 1 shows isoefficiency functions for different architectures and work-distribution schemes. Much of the analysis presented in this paper is applicable to other parallel formulations in which work is shared dynamically among several processors (e.g., parallel divide and conquer algorithm).

The concept of isoefficiency is extremely useful in characterizing the scalability of parallel algorithms for which linear speedup for arbitrarily many processors can be obtained by simply increasing the problem size. For example, the isoefficiency function of the parallel algorithm for solving the 0/1 knapsack problem given [6] is $O(N \log N)$; hence it is highly scalable. On the other hand, a frequently used parallel formulation of quicksort [11] has an exponential isoefficiency function, which means that the formulation is not capable of using many processors effectively. Since the isoefficiency function has to be at least linear, we can

| *Interconnection* | Diameter | *Isoefficiency* | *Order of dependence* | *Work-distribution scheme* |
|---|---|---|---|---|
| 1–ring | $N$ | $\beta^N$ | Exponential | Section5,Wah[14],Monien[8] |
| 1–ring | $N$ | $N^3 \log N$ | Cubic Polynomial | Finkel and Manber[1] |
| 1–ring | $N$ | $N^2 \log N$ | Quad. Polynomial | Section8 |
| Hypercube | $\log N$ | $N^{1.57}$ | Small Polynomial | Section6 |
| Shared-memory $\omega$-switch | 1 | $N^2 * \log N$ | Quad. Polynomial | Section7 inferior version |
| Shared-memory Combining switch | 1 | $N * \log N$ | Almost Linear | Section7 improved version |

Table 1: Isoefficiency functions for different work-distribution schemes and architectures.

also determine whether a parallel algorithm is as good as it can be. Clearly, parallel DFS on the shared-memory architecture with the improved work-distribution scheme has an almost optimal performance, as the isoefficiency function can be improved by at most a log factor. It would be interesting to find work-distribution schemes for the ring and hypercube architectures that have better isoefficiency functions than the ones presented here, or to prove that no better schemes are possible.

**Acknowledgement**: The quality of presentation in this paper has substantially improved due to comments by anonymous referees on an earlier draft of this paper. Mohamed Gouda, Dan Miranker, Chuck Seitz, Jay Misra provided useful comments at various stages of this work.

# References

[1] 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.

[2] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer - designing a MIMD, shared memory parallel computer. *IEEE Transactions on Computers*, C–32, No. 2:175–189, February 1983.

[3] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[4] Richard Korf. Optimal path finding algorithms. In L. N. Kanal and Vipin Kumar, editors, *Search in Artificial Intelligence*. Springer-Verlag, New York, NY, 1988.

[5] T. H. Lai and Sartaj Sahni. Anomalies in parallel branch and bound algorithms. *Communications of the ACM*, pages 594–602, 1984.

[6] J. Lee, E. Shragowitz, and S. Sahni. A hypercube algorithm for the 0/1 knapsack problem. In *Proceedings of International Conference on Parallel Processing*, pages 699–706, 1987.

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

[8] B. Monien and O. Vornberger. The ring machine. Technical report, University of Paderborn, FRG, 1985. Also in *Computers and Artificial Intelligence*, 3(1987).

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

[10] G. F. Pfister et al. The IBM research parallel processor prototype (RP3). In *Proceedings of International Conference on Parallel Processing*, pages 764–797, 1985.

[11] Michael J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, New York, NY, 1987.

[12] V. Nageshwara Rao and V. Kumar. Parallel depth-first search, part I: Implementation. *International Journal of Parallel Programming*, 16 (6):479–499, December 1987.

[13] V. Nageshwara Rao and Vipin Kumar. Superlinear speedup in state-space search. In *Proceedings of the 1988 Foundation of Software Technology and Theoretical Computer Science*, number 338 in Lecture Notes in Computer Science, pages 161–174. Springer-Verlag, 1988.

[14] 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.