Proc. Scalable High Performance Computing Conf. SHPCC'94, IEEE Comp. Sc. Press (1994), 773-780

Work-Load Balancing in Highly Parallel Depth-First Search

A. Reinefeld and V. Schnecke
PC^2 – Paderborn Center for Parallel Computing
D-33095 Paderborn, Germany

Abstract

Among the various approaches for parallel depth-first search (DFS), the stack-splitting schemes are most popular. However, as shown in this paper, dynamical stack-splitting is not suitable for massively parallel systems with several hundred processors. Initial work-load imbalances and work packets of dissimilar sizes cause a high communication overhead.

We compare work-load balancing strategies of two depth-first searches and propose a scheme that uses fine-grained fixed-sized work packets. In its iterative-deepening variant (named AIDA) the global workload distribution improves from one iteration to the next. As a consequence, the communication overhead decreases with increasing search time!*

1 Introduction

Depth-first search (DFS) is an important technique for finding a solution in a state space tree (or graph) containing one or more solutions. Many applications of Operations Research, Artificial Intelligence and other areas in Computing Science use DFS as a basic solution method. Because these problems are computationally intensive, the design of efficient parallel algorithms is of great importance.

In this paper, we compare two work-load balancing methods for massively parallel DFS: a *stack-splitting scheme* [1, 7, 13] and a *fixed-packet DFS* [16]. Both employ a simple task attraction mechanism. They mainly differ by the work distribution: While the first one splits its stack on demand, the second uses fixed size work packets.

We briefly discuss basic sequential search techniques, introduce the two parallel DFS variants, and present empirical performance results obtained on a large scale 1024-node MIMD system. Our theoretical analysis gives evidence that the fixed-packet DFS scales better on large systems than the commonly used stack-splitting, thereby confirming our empirical results.

2 Basic Search Schemes

Depth-First Search (DFS) first expands the initial state by generating all successors to the root node. At each subsequent step, one of the most recently generated nodes is taken and its successors are generated. (The successors may be sorted according to some heuristic prior to expansion.) If at any instance, there are no successors to a node, or if it can be determined that this node does not lead to a solution, the search *backtracks*, that is, the expansion proceeds with one other of the most recently generated nodes.

DFS is usually implemented with a stack holding all nodes (and immediate successors) on the path to the currently explored node. The resulting space complexity of $O(d)$ is linear to the search depth d.

Backtracking is the simplest form of DFS. It terminates as soon as a solution is found. Therefore, optimality cannot be guaranteed. Moreover, backtracking might not terminate in graphs containing cycles or in trees with unbounded depth.

Depth-First Branch-and-Bound (DFBB) uses a heuristic function to eliminate parts of the search space that cannot contain an optimal solution. It continues after finding a first solution until the search space is exhausted. Whenever a better solution is found, the current solution path and value are updated. Subtrees that are known to be inferior to the current solution are cut off.

Best-First Search sorts the sequence of node expansions according to a heuristic function. The popular A* algorithm [9, 11] uses a heuristic evaluation function $f(n) = g(n) + h(n)$ to decide which successor node n to expand next. Here, $g(n)$ is the measured cost of the path from the initial state to the current node n and $h(n)$ is the estimated completion cost to a nearest goal state. If h does not overestimate the remaining cost, A* is said to be *admissible*, that is, it finds an optimal (least cost) solution path. Moreover, it does so with minimal node expansions [11]; no other search algorithm (with the same heuristic h) can do better. This is possible, because A* keeps the search graph in memory and performs a best-first search on the gathered node information.

One disadvantage of A* is its exponential space complexity of $O(w^d)$ in trees of width w and depth d. The high demands on storage space makes it infeasible for many applications.

Iterative-Deepening A* (IDA*) [6] simulates A*'s best-first node expansion by a series of depth-first searches, each with the cost-bound $f(n)$ increased by the minimal amount. The cost-bound is initially set to the heuristic estimate of the root node. Then, for each iteration, the bound is increased to the minimum value that exceeded the previous bound until a solution is found. Like A*, IDA* is admissible (finds an optimal solution) when h does not overestimate the solution cost [6]. But it does so with a much lower space complexity of $O(d)$.

At first sight, it might seem that IDA* wastes computing time by repeatedly re-examining the same nodes of the shallower tree levels. But theoretical analyses [6, 8] give evidence that IDA* has the same asymptotic branching factor as A* when the search space grows exponential with the search depth.

3 Two Approaches to Parallel DFS

DFS can be implemented on a MIMD system by partitioning the search space into subtrees that are searched in parallel. Each processor searches a disjoint subtree in a depth-first fashion, which can be done asynchronously without any communication[1]. When a processor has finished its work, it tries to get an unsearched subtree from another processor. When a goal node is found, all of them quit.

Effective work-load balancing is important to keep all processors busy. In the following, we describe two variants that are based on simple task attraction: a dynamical stack-splitting method and a scheme using fine-grained fixed work packets.

[1]This is true for IDA* and backtracking. Branch-and-bound needs some communication for broadcasting the bound values.

3.1 Stack-Splitting DFS

In the parallel *stack-splitting DFS* described by Kumar and Rao [7, 12] each processor works on its own local stack that keeps track of the untried alternatives. When the local stack is empty, a processor issues a work-request to another processor for an unsearched subtree. The donor then splits its stack and sends a part of its own work to the requester.

Initially, all work is given to one processor, P_0, which performs DFS on the root node. The other processors start with an empty stack, immediately asking for work. When P_0 has generated enough nodes, it splits its stack, donating subtrees to the requesting processors, which likewise split their stacks on demand.

This scheme works for simple DFS as well as for iterative DFS. In the latter case, the algorithm is named *PIDA**, for *Parallel IDA** [12]. PIDA* starts a new iteration with the an increased cost-bound when all processors have finished their current iteration without finding a goal node. The end of an iteration is determined by a barrier synchronization algorithm, e.g., Dijkstra's termination detection algorithm [3].

The performance results in the literature [7, 13, 15] and our own experiments indicate that stack-splitting works only on moderately parallel systems with ≤ 256 processors and a small communication diameter. It does not seem to scale up for larger system sizes because of high communication overheads and inherent work load imbalances. Two major bottlenecks make stack-splitting impractical for massively parallel systems:

- on networks with a large communication diameter it takes a long time to equally distribute the initial work-load among the processors
- recursive stack-splitting generates work packets of dissimilar sizes, resulting in vastly different (and unpredictable) processing times per packet.

Moreover, the stack-splitting method requires implementation of explicit stack handling routines, which is more error-prone and less efficient than the compiler-generated recursive program code [15].

3.2 Fixed-Packet DFS

Our second scheme, named *fixed-packet DFS*, has the two working phases shown in Figure 1:

1. In an initial *data partitioning phase*, the root node is broadcasted to all processors, which redundantly expand the first few tree levels in a

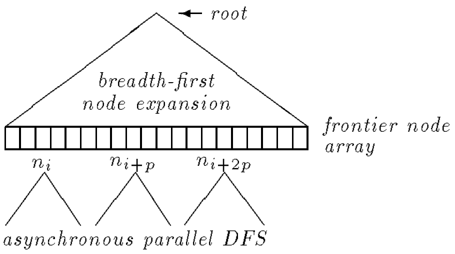

Figure 1: Fixed-Packet DFS Program Architecture

breadth-first fashion. Each processor stores the same frontier nodes in its local node array. Neither communication nor synchronization is required in this phase.

2. In the main *asynchronous search phase*, each processor P_i starts expanding its subtrees $n_i, n_{p+i}, n_{2p+i}, \ldots$ in DFBB fashion[2] When a processor gets idle, it sends a work request to obtain a new work packet (i.e., one or more unprocessed frontier nodes) from another processor. If a processor finds a new solution, all others are informed by a global broadcast.

In practice, little work-load balancing is required, because the initial distribution phase (taking only a few seconds) keeps the processors busy for more than 90% of the time without any loadbalancing.

Our fixed-packet DFS scheme ships fine-grained work packets that are not further splitted. Hence, existing sequential DFS applications can be linked to our communication code to build a highly parallel DFS program without modifying the code.

Fixed-packet DFS is especially effective in combination with iterative-deepening search [16]. We named it *AIDA** for asynchronous parallel IDA*. AIDA* expands the frontier node subtrees in an iterative-deepening manner until a solution is found.

Unlike PIDA*, AIDA* does not perform a hard barrier synchronization between the iterations. When a processor could not get further work, it is allowed to proceed asynchronously with the next iteration, first expanding its own local frontier nodes to the next larger cost-bound. On one hand, this reduces processor idle times, but on the other hand, the search cannot be stopped right after a first solution has been found. Instead, the processors working on earlier iterations must continue their search to check for better solutions.

[2] Note that the nodes assigned to a processor cover all parts of the tree, so this leads to a wide spreaded distribution of the search-frontier over the whole system.

Note, that in AIDA* the work packets change ownership when being sent to another processor. This has the effect of a self-improving global work-load balancing because subtrees tend to grow at the same rate when being expanded to the next larger search bound. Lightly loaded processors that asked for work in the last iteration will be better utilized in the next. More important, the communication overhead decreases with increasing search time [16].

4 Applications

The *15-puzzle* [6, 9, 11], a typical application from single-agent search, consists of fifteen squared tiles located in a squared tray of size 4×4. One square, the *blank square*, is kept empty so that an orthogonally adjacent tile can slide into its position – thus leaving a blank square at its origin. The problem is to re-arrange some given initial configuration with the fewest number of moves into a goal configuration without lifting one tile over another. While it would seem easy to obtain any solution, finding an optimal (=shortest) solution path is *NP*-complete [14]. The 15-puzzle spawns a search space of $16!/2 \approx 10^{13}$ states. Using IDA*, it takes some hundred million node expansions to solve a random problem instance with the popular *Manhattan distance* (the sum of the minimum displacement of each tile from its goal position) as a heuristic estimate function.

Floorplan area optimization [17, 18] is a stage in VLSI design. Here the relative placements and areas of the building blocks of a chip are known, but their exact dimensions can still be varied over a wide range. A floorplan is represented by two dual polar graphs $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and $\mathcal{H} = (\mathcal{W}, \mathcal{F})$, and a list of potential implementations for each block. As shown in Figure 2, the vertices in $\mathcal{V}$ and $\mathcal{W}$ represent the vertical and horizontal line segments of the floorplan. There exists an edge $e = (v_1, v_2)$ in the graph $\mathcal{G}$, if there is a block in the floorplan, whose left and right edges lie on the corresponding vertical line segments. For a specific configuration (i.e. a floorplan with exact block sizes), the edges are weighted with the dimensions of the blocks in this configuration. The solution of the floorplan optimization is a configuration with minimum layout area, given by the product of the longest paths in the graphs $\mathcal{G}$ and $\mathcal{H}$.

In our parallel DFBB implementation, the leaves of the search tree describe complete configurations, while the inner nodes at depth d represent partial floorplans consisting of blocks $B_1 \ldots B_d$. The algorithm back-

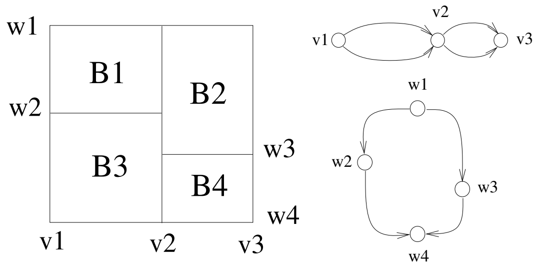

Figure 2: A floorplan and the graphs $\mathcal{G}$ and $\mathcal{H}$

tracks when the area of a new partial floorplan exceeds the area of the currently best solution.

5 Empirical Results

5.1 Speedups

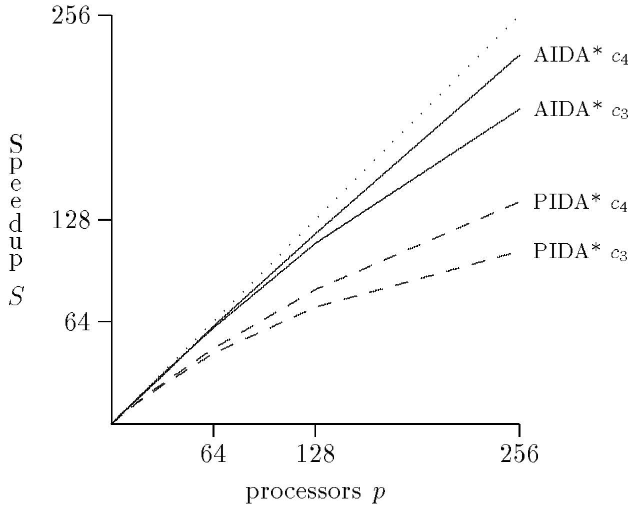

Figure 3: Speedup on a ring (15-puzzle)

Figure 3 shows speedup graphs of the 15-puzzle run on a unidirectional ring with up to 256 processors. We selected the ring topology for our experiments, because due to the large communication diameter, it is difficult to achieve good speedups. Algorithms exhibiting good performance on a relatively small ring are likely to run efficiently on much larger networks with a small diameter (torus, hypercube, etc.). As can be seen, this is true for AIDA*, which outperforms PIDA*, especially on the larger networks.

Taking Korf's [6] 100 random 15-puzzle problem instances and dividing them into four classes $c_1 \ldots c_4$, we run the 50 hardest problems of the classes c_3 and c_4 on our transputer system. Speedup anomalies are eliminated by normalizing the CPU time results to the number of node expansions. The data given is the average of the 25 single speedups achieved in the corresponding problem class.

In both algorithms, PIDA* and AIDA*, work requests are forwarded along the ring until a processor has work to share. The donor then selects an unsearched node, deletes it from the stack (for PIDA*) or frontier node array (for AIDA*) and sends it in the opposite direction to the requester[3]. When no processor has work to share, the original work-request makes a full round through the ring, indicating that no work is available.

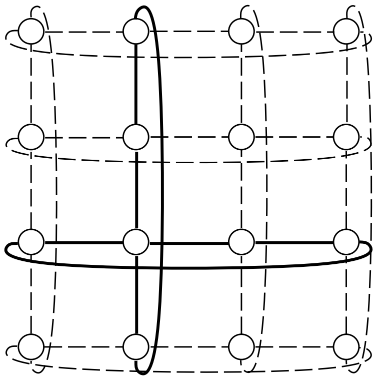

Figure 4: Work-load balancing in fixed-packet DFS

On a torus network, the work-packets are sent first along the horizontal and then along the vertical rings, see Fig. 4. This results in a widespreaded distribution of the local work-load among the whole system. Each processor consideres a different subset of $2\sqrt{p}-1$ processors. So the maximum travel-length of the work-packets doesn't increase lineary with system-size.

Figure 5 shows the AIDA* performance on torus topologies of up to $32 \times 32 = 1024$ processors. On this system, a class c_3 problem takes only 7 seconds to solve, while the larger c_4 problem instances take 43 seconds on the average.

The topmost graph in Figure 5 illustrates AIDA*'s performance on the larger $(5 \times 4) - 1 = 19$-puzzle. Thirteen problem instances have been generated and run on tori with an average CPU-time of 30 minutes on the 1024-node system.

[3]More precisely, AIDA* bundles up to five nodes in a work packet. PIDA* transfers the highest unsearched subtree; alternative stack-splitting strategies are discussed in [4, 7].

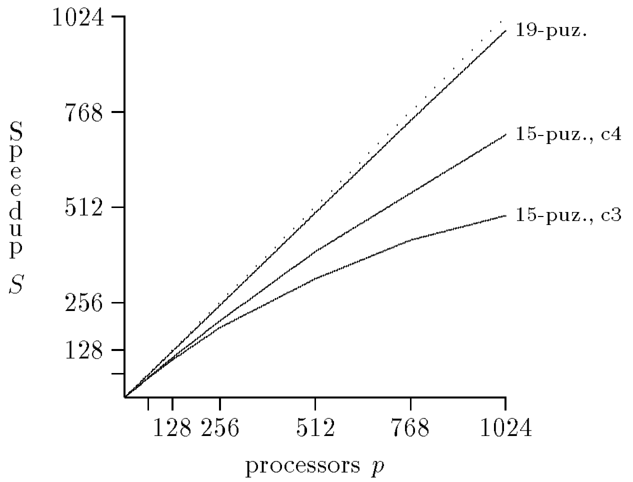

Figure 5: AIDA* speedup on torus topologies

5.2 Initial Work Distribution in Fixed-Packet DFS

In the first phase, AIDA* performs a breadth-first search to expand the same search node frontier on every processor. The breadth-first search[4] is stopped when a sufficient number of nodes is generated to give every processor its own fine-grained work packets for further asynchronous exploration. In practice, this is not done in a single step, but an intermediate search frontier is stored on every processor, so that the more fine-grained work packets can be generated asynchronously on all processors (for details see [16]).

At the end of the initial work distribution phase, each processor 'owns' $\approx$ 3000 work packets for the 15-puzzle, when being run on 1024 processors.

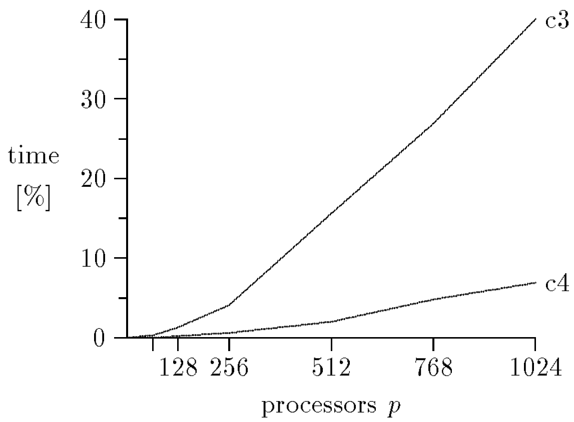

Figure 6: Time of first phase (15-puzzle)

[4] In practice, we simulate breadth-first search by iterative-deepening.

The time spent in the first phase depends on the node expansion time and the system size, because enough nodes must be generated to keep every processor busy thereafter. In the 15-puzzle, three seconds are spent in the first phase on a 1024-node system [16], resulting in a 40% overhead of the total elapsed time for the small class c_3 problems (Fig. 6) on the 1024-node system. Clearly, AIDA* – and more generally – our fixed-packet DFS, is ill suited for running small problem instances. But many other methods would not work either because they suffer the same initial work distribution problem.

5.3 Communication Overhead

bound	nodes	messages		work mess.	
		PIDA*	AIDA*	PIDA*	AIDA*
35	24	4	–	0	–
37	172	4	–	0	–
39	1,060	26	–	0	–
41	6,259	51	–	0	–
43	71,906	276	–	4	–
45	199,538	483	163	7	3
47	1,097,015	691	266	20	13
49	5,967,654	1639	329	72	15
51	32,036,451	3231	585	253	12
53	169,630,586	11287	562	1037	8
55	886,017,863	26050	285	3066	4

Table 1: Messages per processor (15-puzzle, $p = 128$)

Table 1 shows the total messages and work messages received by a single node while computing a hard 15-puzzle problem[5] on a 128-processor ring. The total message count includes all message transfers through that specific node, while the work message count represents only work packets received by the node.

As can be seen, PIDA*'s total message flow increases from one iteration to the next. AIDA*'s communication overhead, on the other hand, decreases after a couple of iterations. This is caused by our adaptive load balancing method that constantly improves the global work-load by keeping the transferred nodes at the new processor for the next iteration.

Moreover, the absolute number of messages is by an order of magnitude lower for AIDA*. Similar results have been obtained for the larger 1024-node system and also for the more time-consuming 19-puzzle. The communication overhead does not significantly in-

[5] Problem #95, h=35, g=57. AIDA* solution time: 243 secs.

crease with growing system size because most work requests are satisfied in the nearest neighborship.

5.4 Effect of Different Work Packet Sizes

Work packets of dissimilar sizes make it harder to balance the global work-load, because the amount of work a processor receives with a packet is not known *a priori*. This is a serious problem in the 15-puzzle, where the work packet sizes vary by several orders of magnitude.

Figure 7 illustrates for various work packet sizes (measured in nodes per subtree) how many of these packets exist in the system. Note that both axes are plotted logarithmical, showing packets ranging from only a handful of nodes up to 300,000 nodes for the largest subtree.

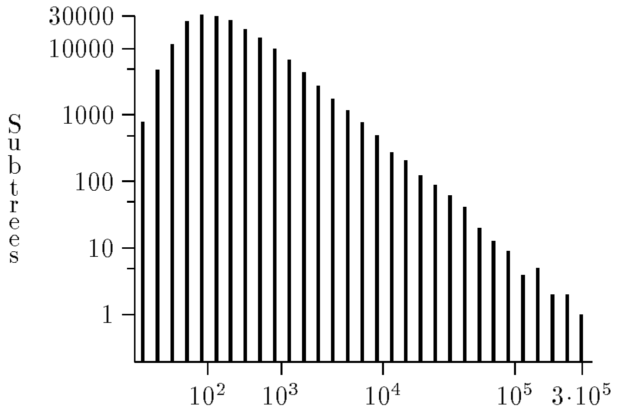

Figure 7: Size of work packets, 15-puzzle, $p = 64$

We therefore modified AIDA* to transfer only medium sized subtrees. In our implementation [16], this is done with a partial sort on the frontier nodes array. After each iteration, work packets of average size are moved to the end of the array, to be shipped to other processors during the next iteration. Further research is under way to adjust all work packets to an average size with node splitting and node contraction strategies [2].

5.5 Results on Floorplan Optimization

We implemented a DFS-Branch-and-Bound algorithm with our fixed-packet work-load distribution scheme for the floorplan optimization problem. We tested our algorithm with an instance consisting of 25 blocks and five different implementations per block, which spawns a search tree with $5^{25} \approx 3.0 \cdot 10^{17}$ nodes. In the initial distribution phase, the first levels of the tree are expanded to get ≈ 80 frontier nodes for each processor. The proportion of the first phase to the parallel execution time ranges from 0.2% ($p = 64$) to 4.6% ($p = 768$). As in the 15- and 19-puzzle, the first phase gives a good initial work distribution, which keeps the processors working on their local nodes for more than 90% of the total execution time. Note, that in this time there is no communication for load balancing required, only newly computed bound values have to be broadcasted. Parallelism is fully exploited since all processors execute the sequential search algorithm on their local nodes. This results in a high work-rate even for larger systems as shown in Table 2.

p	15-puzzle	19-puzzle	floor
1	35000	—	772
64	29586	—	726
128	29203	—	719
256	28218	—	719
512	26817	33905	709
768	25020	33817	701
1024	24165	33707	685

Table 2: Average work-rate (nodes per sec.)

At first sight, the exhaustive search of our fixed-packet DFBB might seem less efficient, because the single processors would terminate at different times due to the fixed work-packet sizes. But in practice, we found that due to the better bounds the subtrees are getting smaller to the end of the search, so that the termination times are very close to each other.

6 Scalability Analysis

As seen in our empirical results, the stack-splitting scheme does not work on the larger systems because of its large communication-overhead. The total overhead of a parallel system is given by

$$t_{over}(p) = p \cdot t_{par}(p) - t_{seq}.$$

It describes the amount of time the processors spend in addition to the execution time of the sequential algorithm t_{seq}. This overhead can be used in the speedup equation

$$S(p) = \frac{t_{seq}}{t_{par}(p)} = \frac{t_{seq} \cdot p}{t_{over}(p) + t_{seq}} = \frac{p}{1 + \frac{t_{over}(p)}{t_{seq}}}$$

and the efficiency equation

$$E(p) = \frac{1}{1 + \frac{t_{over}(p)}{t_{seq}}}.$$

Kumar and Rao [7] define a measure for the scalability of parallel systems. The *isoefficiency-function* describes the required increase of the work w to achieve a constant efficiency E with increasing system size p. Because $t_{seq} = \Theta(w)$, we can derive the following general equation for the growth rate of w:

$$w = \frac{E}{1-E} \cdot t_{over}(p)$$

To compare the scalability of the two work-distribution schemes, we must determine the overheads of both schemes. In the stack-splitting scheme, the work w must be splitted into w/p sized work-packets to get an optimal distribution among the processors. When the stack is splitted, the work w is divided into two parts, the smallest of which is at least $\alpha \cdot w$. For a processor P_d with distance d from processor P_0, which initially holds all the work, the maximum work it can get — due to the recursive splitting on the way from P_0 to P_d — is $(1-\alpha)^d \cdot w$. So P_d must receive at least $\frac{w/p}{(1-\alpha)^d \cdot w}$ work packets. This results in a total work transfer count greater than $\frac{\beta^{d(p)}}{d(p)}$ with $\beta := (1-\alpha)^{-1}$ and $d(p)$ as the diameter of the network. With this, we get a lower bound for the overhead of the stack-splitting scheme in our implementation on the ring:

$$t_{over}^{stack-split}(p) = \Omega\left(\frac{\beta^p}{p}\right)$$

Kumar and Rao [7] also describe another work-distribution method for their stack-splitting scheme where the work can be received from any processor throughout the system (not only neighboring processors). They derive an upper bound for the isoefficiency function of $O(d(p) \cdot \log p \cdot p)$.

In our fixed-packet DFS, the communication overhead and the runtime spent in the initial work distribution phase grow only linear with the system size. This results in the following upper bound for the overhead of our fixed-packet DFS:

$$t_{over}^{fixed-pack}(p) = O(p) + p \cdot t_{1st\,phase}(p) = O(p^2)$$

Figure 8 shows isoefficiency functions for the stack-splitting scheme (with both work distribution methods) and our fixed-packet scheme on the ring topology.

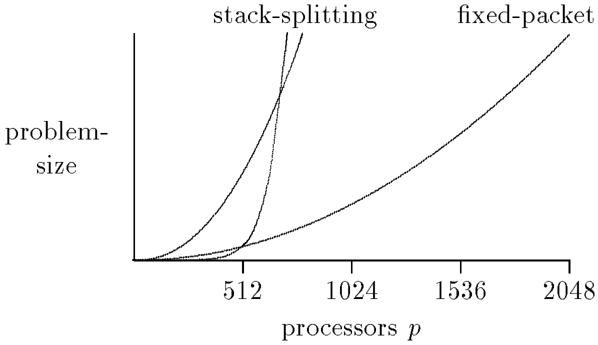

Figure 8: Isoefficiency functions for the ring

7 Conclusion

We compared the performance of two parallel DFS schemes. The *stack-splitting* scheme employs dynamical work-load balancing right from the beginning of the search, which results in high communication overheads and initial processor idle times. Recursive splitting of the work packets generates successively smaller packets that might be too small to justify shipment to another processor. In its iterative-deepening version (PIDA*) the communication overhead increases with increasing search time.

In our parallel *fixed-packet DFS* the processors start with an equal work distribution that has been generated by a redundant breadth-first search of the first few tree levels on all processors. Later on, work packets of fixed sizes are shipped to idle processors on demand.

In its iterative-deepening variant the fixed-packet DFS, named *AIDA**, automatically improves the global load balance by changing ownership of the transferred work packets. As a result, the communication overhead decreases from iteration to iteration.

Unlike PIDA*, AIDA* is portable in the sense that existing sequential search routines can be linked to AIDA*'s communication routines for implementation on a large scale MIMD system.

References

[1] S. Arvindam, V. Kumar, V. Rao. *Efficient parallel algorithms for searching problems: Applications in VLSI CAD.* 3rd Symp. Frontiers Mass. Par. Comp. 1990, 166–169.

[2] P.P. Chakrabarti, S. Ghose, A. Acharya, S.C. de Sarkar. *Heuristic search in restricted memory.* Art. Intell. 41,2(1989/90), 197 – 221.

[3] E. Dijkstra, W.H.J. Feijen and A.J.M. van Gasteren. *Derivation of a termination detection algorithm for distributed computation.* Inf. Proc. Lett. 16 (1983), 217–219.

[4] S. Farrange, T. A. Marsland. *Dynamic splitting of decision trees.* Tech. Rep. TR93.03, University of Alberta, Edmonton (March 93).

[5] A.Y. Grama, A. Gupta and V. Kumar. *Isoefficiency: Measuring the scalability of parallel algorithms and architectures.* IEEE Par. & Distr. Techn. 1,3 (1993), 12–21.

[6] R.E. Korf. *Depth-first iterative-deepening: An optimal admissible tree search.* Art. Intell. 27 (1985), 97–109.

[7] V. Kumar, V. Rao. *Scalable parallel formulations of depth-first search.* Kumar, Gopalakrishnan, Kanal, eds., Par. Alg. for Mach. Intell. and Vision, Springer 1990, 1–41.

[8] A. Mahanti, S. Ghosh, D.S. Nau, A.K. Pal and L. Kanal. *Performance of IDA* on trees and graphs.* 10th Nat. Conf. on Art. Int., AAAI-92, San Jose, (1992), 539–544.

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

[10] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, 1982.

[11] J. Pearl. *Heuristics. Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, Reading, MA, (1984).

[12] V.N. Rao, V. Kumar and K. Ramesh. *A parallel implementation of iterative-deepening A*.* AAAI-87, 878–882.

[13] V.N. Rao and V. Kumar. *Parallel depth first search. Part I. Implementation.* Intl. J. Parallel Programming 12, 6 (1987), 479–499.

[14] D. Ratner and M. Warmuth. *Finding a shortest solution for the $N \times N$ extension of the 15-puzzle is intractable.* AAAI-86, 168–172.

[15] A. Reinefeld. *Effective parallel backtracking methods for Operations Research applications.* EUROSIM Intl. Conf. Massively Par. Proc., Delft (June 1994).

[16] A. Reinefeld and V. Schnecke. *AIDA* – Asynchronous Parallel IDA*.* 10^{th} Canadian Conf. on Art. Intell. AI'94, (May 1994), Banff, Canada.

[17] L. Stockmeyer. *Optimal orientations of cells in silicon floorplan designs.* Inform. and Control 57 (1983), 97–101.

[18] S. Wimer, I. Koren and I. Cederbaum. *Optimal aspect ratios of building blocks in VLSI.* 25th ACM/IEEE Design Automation Conference, (1988), 66–72.