Unstructured Tree Search on SIMD Parallel Computers: A Summary of Results*

George Karypis
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455
karypis@cs.umn.edu

Vipin Kumar
Department of Computer Science,
University of Minnesota
Minneapolis, MN 55455
kumar@cs.umn.edu

Abstract

In this paper, we present new methods for load balancing of unstructured tree computations on large-scale SIMD machines, and analyze the scalability of these and other existing schemes. An efficient formulation of tree search on a SIMD machine comprises of two major components: (i) a triggering mechanism, which determines when the search space redistribution must occur to balance search space over processors; and (ii) a scheme to redistribute the search space. We have devised a new redistribution mechanism and a new triggering mechanism. Either of these can be used in conjunction with triggering and redistribution mechanisms developed by other researchers. We analyze the scalability of these mechanisms, and verify the results experimentally. The analysis and experiments show that our new load balancing methods are highly scalable on SIMD architectures. Their scalability is shown to be no worse than that of the best load balancing schemes on MIMD architectures. We verify our theoretical results by implementing the 15-puzzle problem on a CM-2[1] SIMD parallel computer.

1 Introduction

Tree search is central to solving a variety of problems in artificial intelligence [12, 26], combinatorial optimization [11, 21], operations research [25] and Monte-Carlo evaluations of functional integrals [31]. The trees that need to be searched for most practical problems happen to be quite large, and for many tree search algorithms, different parts can be searched relatively independently. These trees tend to be highly irregular in nature and hence, a naive scheme for partitioning the search space can result in highly uneven distribution of work among processors and lead to poor overall performance. The job of partitioning irregular search spaces is particularly difficult for SIMD parallel computers such as the CM-2, in which all processors work in lock-step to execute the same program. The reason is that in SIMD machines, work distribution needs to be done on a global scale (*i.e.* if a processor becomes idle, then it has to wait until the entire machine enters a work distribution phase). In contrast, on MIMD machines, an idle processor can request work from another busy processor without any other processor being involved. Many efficient load balancing schemes have already been developed for dynamically partitioning large irregular trees for MIMD parallel computers [2, 4, 6, 23, 29, 32, 33], whereas until recently, it was common wisdom that such irregular problems cannot be solved on large-scale SIMD parallel computers [21].

Recent research has shown that data parallel SIMD architectures can also be used to implement parallel tree search algorithms effectively. Powley, Korf and Ferguson [27, 28] and Mahanti and Daniels [3, 22] present parallel formulations of a tree search algorithm IDA*, for solving the 15 puzzle problem on CM-2. Frye and Myczkowski [5] presents an implementation of a depth-first tree search algorithm on the CM-2 for a block puzzle.

The load balancing mechanisms used in the implementations of Frye, Powley, and Mahanti are different from each other. From the experimental results presented, it is difficult to ascertain the relative merits of these different mechanisms. This is because the performance of different schemes may be impacted quite differently by changes in hardware characteristics (such as interconnection network, CPU speed, speed of communication channels etc.), number of processors, and the size of the problem instance being solved [17]. Hence any conclusions drawn on a set of experimental results may become invalid by changes in any one of the above parameters. Scalability analysis of a parallel algorithm and architecture combination is very useful in extrapolating these conclusions [9, 17, 19]. The isoefficiency metric has been found to be quite useful in characterizing scalability of a number of algorithms [8, 20]. In particular, it has helped determine optimal load balancing schemes for tree search for a variety of MIMD architectures [19, 7, 16].

In this paper, we present new methods for load balancing of unstructured tree computations on large-scale SIMD machines, and analyze the scalability of these and existing schemes. The analysis and experi-

*This work was supported by IST/SDIO through the Army Research Office grant #28408-MA-SDI and by the Army High Performance Computing Research Center at the University of Minnesota.

[1]CM-2 is a registered trademark of Thinking Machines Corporation.

ments show that our new load balancing methods are highly scalable on SIMD architectures. In particular, their scalability is no worse than that of the best load balancing schemes on MIMD architectures.

Section 2 provides a description of existing load balancing schemes and the new schemes we have developed. Section 3 describes the various terms and assumptions used in the analysis. Section 4 and 5 present the analysis of static triggering and its experimental evaluation. Section 6 and 7 present the analysis of dynamic triggering and its experimental verification. Section 8 comments on other related work in this area.

2 Dynamic Load Balancing Algorithms for Parallel Search

Specification of a tree search problem includes a description of the root node of the tree and a successor-generation-function that can be used to generate successors of any given node. Given these two, the entire tree can be generated and searched for goal nodes. Often strong heuristics are available to prune the tree at various nodes. The tree can be generated using different methods. Depth-first method is used in many important tree search algorithms such as Depth-First Branch and Bound [15], IDA* [14], Backtracking [11]. In this paper we consider parallel depth-first-search on SIMD machines.

A common method used for parallel depth-first-search of dynamically generated trees on a SIMD machine [27, 22, 5] is as follows. At any time, all the processors are either in a *search* phase or in a *load balancing* phase. In the search phase, each processor searches a disjoint part of the search space in a depth-first-search (DFS) fashion by performing node expansion cycles in lock-step. When a processor has finished searching its part of the search space, it stays idle until it gets additional work during the next load balancing phase. All processors switch from the search phase to the load balancing phase when a triggering condition is satisfied. In the load balancing phase, busy processors split their work and share it with idle processors. When a goal node is found, all processors quit. If the search space is finite and has no solutions, then eventually all the processors would run out of work, and parallel search will terminate.

Since each processor searches the space in a depth-first manner, the (part of) state space to be searched is efficiently represented by a stack. The depth of the stack is the depth of the node being currently explored; and each level of the stack keeps track of untried alternatives. Each processor maintains its own local stack on which it executes depth-first-search. The current unsearched tree space, assigned to any processor, can be partitioned into two parts by simply partitioning untried alternatives (on the current stack) into two parts. A processor is considered to be busy if it can split its work into two non empty parts, one for itself and one to give away. In the rest of this paper, a processor is considered to be busy if it has at least two nodes on its stack. We denote the number of idle processors by I, the number of busy processors by A and the total number of processors by P. Also, the terms busy and active processors will be used interchangeably.

2.1 Previous Schemes for Load Balancing

The first scheme we study is similar to one of the schemes proposed in [22]. In this algorithm, the triggering condition is computed after each node expansion cycle in the search phase. If this condition is satisfied, then a load balancing phase is initiated. In the load balancing phase, idle processors are matched one-to-one with busy processors. This is done by enumerating both the idle and the busy processors; then each busy processor is matched with the idle processor that received the same value during this enumeration. The busy processors split their work into two parts and transfer one part to their corresponding idle processors[2]. If $I > A$ then only the first A idle processors are matched to busy ones and the remaining $I-A$ processors receive no work. After each load balancing phase, at least one node expansion cycle is completed before the triggering condition is tested again.

A very simple and intuitive scheme [27, 5] is to trigger a load balancing phase when the ratio of active to the total number of processors falls below a fixed threshold. Formally, let x be a number such that $0 \leq x \leq 1$, then the triggering condition for this scheme is:

$$A \leq xP \qquad (1)$$

For the rest of this paper we will refer to this triggering scheme as the **static triggering** scheme with threshold x (in short the S^x-triggering scheme).

An alternative to static triggering is to use a trigger value that changes dynamically in order to adapt itself to the characteristics of the problem. We call this kind of triggering scheme **dynamic triggering** D. A dynamic triggering scheme was presented and analyzed by Powley, Korf and Freguson in [27]. For the rest of this paper we will refer to it as the D^P-triggering scheme. The D^P-triggering works as follows: Let w be the sum of the time spent by processors, let t be the elapsed time since the beginning of the current search phase and let L be the time required to perform the next load balancing phase. After every node expansion cycle, the ratio $\frac{w}{t+L}$ is compared against the number of active processors A, and a load balance is initiated as soon as that ratio is greater or equal to A. In other words the condition that triggers a load balance is:

$$\frac{w}{t+L} \geq A \qquad (2)$$

Because the value of L cannot be known (it requires knowledge of the future), it is approximated by the cost of the previous load balancing phase. D^P is a locally greedy approach that tries to maximize the average rate of work over a search and load balancing phase. Polwey *et. al.* also describe variants of D^P-triggering in [27].

[2]This is done using the rendezvous allocation scheme described in [10].

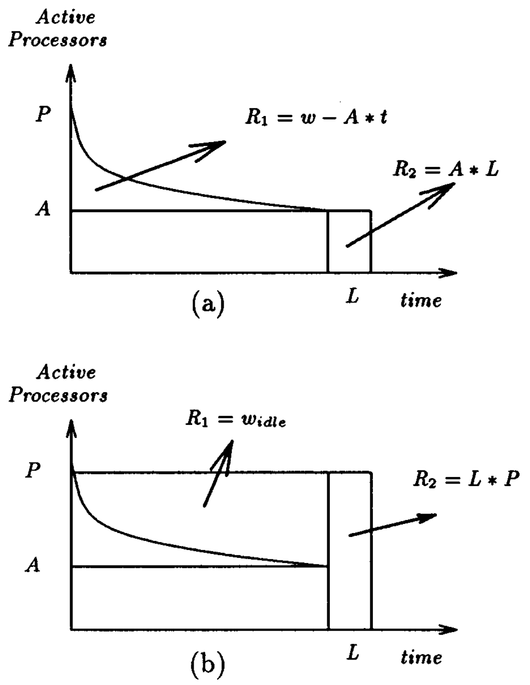

Figure 1: A graphical representation of the triggering conditions for the D^P-triggering and for the D^K-triggering schemes.

Another way of stating the triggering condition for D^P is to rewrite eqn (2) as:

$$w - A * t \geq A * L \qquad (3)$$

From this equation and Fig. 1(a) we see that the D^P-triggering scheme will trigger a load balancing phase as soon as the area R_1 is greater or equal to area R_2.

2.2 Our New Schemes for Load Balancing

We have derived a new matching scheme for mapping idle to busy processors in the load balancing phase. This method can be used with either the static or the dynamic triggering schemes. We have also derived a new dynamic triggering scheme.

The new mapping algorithm is similar to the one described earlier but with the following modification. We now keep a pointer that points to the last processor that gave work during the last load balancing phase. Every time we need to load balance, we start matching busy processors to idle processors, starting from the first busy processor after the one pointed by this pointer. When the pointer reaches the last processor, it starts again from the beginning. For the rest of this paper we will call this pointer **global pointer** and this mapping scheme *GP*. Also, due to the absence of the global pointer we will name the mapping scheme of Section 2.1, *nGP*.

Figure 2 illustrates the *GP* and the *nGP* matching schemes with an example. Assume that at the time when a load balancing phase is triggered, processors 6 and 7 are idle and the others are busy. Also, assume that the global pointer points to processor 5. Now, *nGP* will match processors 6 and 7 to processors 1 and 2 respectively, whereas *GP* will match them to processors 8 and 1 respectively and it will advance the global pointer to processor 1. If after the next search phase, processors 6 and 7 are idle again and the others remain busy, then *nGP* will match them exactly as before where *GP* will match them to processors 2 and 3. The above example also provides the motivation behind *GP*, which is to try to evenly distribute the burden of sharing work among the processors. As we will see in Section 4.1 the upper bound on the number of load balancing phases required for *GP* is much smaller than that for *nGP*. When $x \leq 0.5$ both schemes are similar.

Processors	1	2	3	4	5	6	7	8
example 1								
state	B	B	B	B	B	I	I	B
global pointer					↑			
nGP enumeration of busy processors	1	2	3	4	5			6
GP enumeration of busy processors	2	3	4	5	6			1
enumeration of idle processors						1	2	
example 2								
state	B	B	B	B	B	I	I	B
global pointer	↑							
nGP enumeration of busy processors	1	2	3	4	5			6
GP enumeration of busy processors	6	1	2	3	4			5
enumeration of idle processors						1	2	

Figure 2: Illustration of the *GP* and *nGP* matching schemes. *B* is used to denote busy processors while *I* is used to denote idle ones.

Our new dynamic triggering scheme, called D^K-triggering, takes a different approach than the D^P-triggering scheme. Formally, let w_{idle} be the sum of the idle time of all the processors since the beginning of the current search phase and let $L * P$ be the cost of the next load balancing phase, then the condition that will trigger a load balance is:

$$w_{idle} \geq L * P \qquad (4)$$

Fig. 1(b) illustrates this condition, R_1 is w_{idle} and R_2 is $L * P$. This scheme will trigger a load balancing phase as soon as $R_1 \geq R_2$. Note that if triggering takes place earlier than this point, then the load balancing overhead will be higher than the overhead due to idling and vice versa. Thus, our triggering scheme balances the idle time of the processors during the search phase and the cost of the next load balancing phase.

3 Analysis Framework

In this section we introduce some assumptions and basic terminology necessary to understand the analysis.

When a work transfer is made, work in the active processor's stack is split into two stacks one of which is given to an idle processor. Intuitively, it is ideal to split the stack into two equal pieces. If the work given out is too small, then the idle processor will soon become idle where if it is too big, then the donor processor will soon become idle. Since most practical trees are highly unstructured, it is not possible to split a stack into two parts representing roughly equal halfs of the search space. In our analysis, we make the following rather mild assumption for the splitting mechanism: if work w at one processor is split into two parts ψw and $(1-\psi)w$, then $1-\alpha > \psi > \alpha$, where α is an arbitrarily small constant. We call this splitting mechanism the **alpha-splitting mechanism**. As demonstrated by experiments on MIMD machines [23, 1, 7, 16, 22] it is possible to find alpha-splitting mechanisms for most tree search problems.

The total number of nodes expanded in parallel search can often be higher or lower than the number of nodes expanded by serial search [30, 27, 22] leading to speedup anomalies. Here we study the performance of these load balancing schemes in absence of such speedup anomalies and we assume that the number of nodes expanded by serial and parallel search are the same.

3.1 Definitions and Assumptions

- Problem size W: the number of tree nodes searched by the serial algorithm.
- Number of processors P: number of identical processors in the ensemble being used to solve the given problem.
- Unit computation time U_{calc}: the time taken for one unit of work. In our case this is the time for a single node expansion.
- Unit communication time U_{comm}: the time it takes to send a single node to neighbor processor.
- Single load balancing time t_{lb}: the average time to perform a load balancing phase. Clearly, t_{lb} depends on the size of the work transferred, the distance it travels and the speed of the communication network. For simplicity, we assume that the size of the messages containing work is constant. Also, we assume that t_{lb} doesn't depend on the number of CM-2 processors used.
- Total load balancing time T_{lb}: the total time spent in load balancing by all processors in the entire algorithm.
- Total idling time T_{idle}: this is the sum of the time spent by idle processors during node expansion phases.
- Computation time T_{calc}: is the sum of the time spent by all processors in useful computation. Useful computation is the computation required by the best sequential algorithm in order to solve the problem. Clearly, $T_{calc} = W \times U_{calc}$.
- Running time T_{par}: the execution time on P processor ensemble. Clearly, $P * T_{par} = T_{calc} + T_{idle} + T_{lb}$.
- Speedup S: the ratio $\frac{T_{calc}}{T_{par}}$.
- Efficiency E: is the speedup divided by P. E denotes the effective utilization of computing resources. $E = \frac{T_{calc}}{T_{calc}+T_{idle}+T_{lb}}$.

3.2 Scalability Analysis using the Isoefficiency function

If a parallel algorithm is used to solve a problem instance of a fixed size, then the efficiency decreases as the number of processors P increases. The reason is that the total overhead increases with P. For many parallel algorithms, for a fixed P, if the problem size W is increased, then the efficiency becomes higher, because the total overhead grows slower than W. For these parallel algorithms, the efficiency can be maintained at a desired level with increasing number of processors, provided the problem size is also increased. We call such algorithms **scalable** parallel algorithms.

For a given parallel algorithm, for different parallel architectures, the problem size may have to increase as a different function of P in order to maintain a fixed efficiency. The rate that W has to increase as a function P to keep the efficiency fixed is essentially what determines the degree of scalability of the algorithm architecture combination. If W has to increase as an exponential function of P, then the algorithm-architecture combination is poorly scalable. On the other hand if W needs to grow linearly as a function of P then the algorithm-architecture combination is highly scalable and can easily deliver linearly increasing speedup with increasing number of processors for reasonable increments of problem sizes. If W needs to grow as $f_E(P)$ to maintain an efficiency E, then $f_E(P)$ is defined to be the **isoefficiency function** for efficiency E and the plot of $f_E(P)$ with respect to P is defined to be the **isoefficiency curve** for efficiency E.

A lower bound on any isoefficiency function is that asymptotically, it should be at least linear. This follows from the fact that all problems have a sequential (*i.e.* non decomposable) component. Hence any algorithm which shows a linear isoefficiency on some architecture is optimally scalable on that architecture. Algorithms with isoefficiencies of $O(P \log^c P)$, for small constant c, are also reasonably optimal for practical purposes. For a more rigorous discussion on the isoefficiency metric and scalability analysis, the reader is referred to [19, 17].

4 Scalability Analysis of the Static Triggering Scheme

In order to analyze the scalability of a load balancing scheme, we need to compute T_{lb} and T_{idle}. Due to the dynamic nature of the load balancing algorithms being analyzed, it is very difficult to come up with a precise expression for T_{lb}. We can compute the upper bound for T_{lb} by using a technique that was originally developed in the context of Parallel Depth First Search on MIMD computers [4, 19]. In dynamic load balancing, the communication overheads are caused by work transfers. The total number of work transfers defines an upper bound on the total communication overhead. Let $V(P)$ be the number of load balancing phases needed so that each busy processor has shared its work (with some other processor) at least once. As shown in [13], in any load balancing algorithm using the alpha-splitting, the maximum number of load balancing phases required to solve a problem of size W, is $V(P)\log_{\frac{1}{1-\alpha}} W$. For the rest of this analysis, the maximum number of load balancing phases will be written as $V(P)\log W$ and this upper bound will be used as an estimate of the total number of load balancing phases (our experimental results here as well as for MIMD [16] demonstrate that it is a good approximation). Hence the load balancing overhead T_{lb} is:

$$T_{lb} = P \times V(P)\log W \times t_{lb} \tag{5}$$

The idling time T_{idle}, depends on the characteristics of the search space and the triggering threshold of the S^x-triggering scheme. In any node expansion cycle, the number of busy processors will decrease and will remain between P and xP. As the value of x increases, T_{lb} goes up and T_{idle} comes down. The overhead due to idling can be computed as follows: Assume that the average number of busy processors during node expansion cycles is $(x+\beta)P$; clearly $0 \leq \beta \leq 1-x$. The average number of idle processors during each node expansion cycle is $(1-x-\beta)P$. The total time spent during node expansion cycles is $\frac{W}{x+\beta}U_{calc}$. Hence:

$$T_{idle} = \frac{1-x-\beta}{x+\beta} W \times U_{calc} \tag{6}$$

From eqn (5) and eqn (6) we have:

$$\begin{aligned} E &= \frac{T_{calc}}{T_{calc} + T_{idle} + T_{lb}} \\ &= \frac{1}{\frac{1}{x+\beta} + \frac{P \times V(P)\log W \times t_{lb}}{W \times U_{calc}}} \end{aligned} \tag{7}$$

From eqn (7) we can see that the maximum efficiency of the algorithm is bounded by $x+\beta$. If the problem size W is fixed and P increased, then T_{lb} will increase and the efficiency will come downward approaching 0. If P is fixed and W is increased then T_{calc} will increase faster than T_{lb} and hence the efficiency will approach $x+\beta$. To maintain a fixed efficiency, T_{calc} should remain proportional to T_{lb}. Hence for isoefficiency,

$$\begin{aligned} W \times U_{calc} &\sim P \times V(P)\log W \times t_{lb} \\ W &= O(P \times V(P)\log W) \end{aligned}$$

As long as $V(P)$ is a polynomial in W, we can approximate the above equation by the following:

$$W = O(P \times V(P)\log P) \tag{8}$$

The isoefficiency defined by the above equation is the overall isoefficiency of the algorithm.

4.1 Analysis for *GP-S^x*

In order to analyze the scalability of *GP*, we have to calculate $V(P)$. When $x \leq 0.5$ it will take one load balancing phase for all the busy processors to be requested at least once, hence $V(P) = 1$. Whereas, as shown in [13], when $x > 0.5$, $V(P) = \frac{1}{1-x}$. Substituting this value of $V(P)$ in eqn (5) and eqn (7) we get:

$$T_{lb} = P\frac{1}{1-x}\log W \times t_{lb} \tag{9}$$

$$E = \frac{W \times U_{calc}}{\frac{W}{x+\beta}U_{calc} + P\frac{1}{1-x}\log W \times t_{lb}} \tag{10}$$

Now we substitute $V(P) = \frac{1}{1-x} = O(1)$ in eqn (8) to get the isoefficiency function:

$$W = O(P\log P) \tag{11}$$

4.2 Analysis for *nGP-S^x*

Similarly, in order to analyze the behavior of the *nGP* matching scheme, we have to determine the value of $V(P)$ for any value of x. When $x \leq 0.5$ it will take one load balancing phase for all the busy processors to be requested at least once, hence $V(P) = 1$. Whereas, as shown in [13], when $x > 0.5$, $V(P) = \log^{\frac{2x-1}{1-x}} W$. Substituting this value of $V(P)$ in eqn (5), eqn (7), and eqn (8) we get:

$$T_{lb} = P\log^{\frac{2x-1}{1-x}} W \log W \times t_{lb} \tag{12}$$

$$E = \frac{W \times U_{calc}}{\frac{W}{x+\beta} \times U_{calc} + P\log^{\frac{2x-1}{1-x}} W \log W \times t_{lb}} \tag{13}$$

Isoefficiency function: $W = O(P\log^{\frac{x}{1-x}} P)$ (14)

Clearly, the scalability of *nGP-S^x* becomes worse as the value of x increases. From eqn (9) and eqn (12), we see that as we try to achieve higher efficiencies by increasing x, the upper bound on load balancing overhead for *nGP* increases rapidly while for *GP* it only increases moderately. For example if x increases from 0.80 to 0.90, then T_{lb} increases by a factor of $\log^5 W$ for *nGP*, while it only increases by a factor of 2 for *GP*.

4.3 Optimal Static Trigger for *GP*

If we increase the value of x for the static triggering scheme, then the load balancing overhead increases and the idling overhead decreases. Clearly, maximum efficiency is obtained for the value of x which minimizes the sum $T_{idle} + T_{lb}$. We call such value of x the **optimal static trigger** x_o. For a given value of β we can analytically compute a good approximation of x_o. As shown in [13], for $\beta = 0$ the optimal static trigger is:

$$x_o = \frac{1}{\sqrt{\frac{P}{W}\log_{\frac{1}{1-\alpha}} W \times \frac{t_{lb}}{U_{calc}}} + 1} \tag{15}$$

From this equation we can clearly see the dependence of the optimal static trigger on the various parameters involved in dynamic load balancing. As W increases, the value of x_o also increases, meaning that higher efficiencies are possible for larger problems. As

Static Trigger		0.50		0.60		0.70		0.80		0.90		Analytical trigger, x_o
W	Metric	*nGP*	*GP*	*nGP*	*GP*	*nGP*	*GP*	*nGP*	*GP*	*nGP*	*GP*	
	N_{expand}	547	547	479	483	438	438	400	406	384	379	
2488958	N_{lb}	62	62	105	69	179	78	309	102	376	154	0.67
	E	0.41	0.41	0.37	0.43	0.29	0.43	0.21	0.41	0.19	0.35	
	N_{expand}	1957	1957	1708	1730	1520	1563	1364	1429	1320	1325	
9076121	N_{lb}	72	72	245	84	560	102	1095	134	1317	226	0.79
	E	0.51	0.51	0.43	0.55	0.32	0.58	0.22	0.59	0.19	0.54	
	N_{expand}	4629	4629	4078	4091	3588	3687	3234	3376	3104	3113	
21540929	N_{lb}	73	73	407	90	1251	107	2339	151	3044	250	0.85
	E	0.54	0.54	0.48	0.60	0.33	0.65	0.24	0.68	0.20	0.67	
	N_{expand}	9510	9510	8457	8450	7565	7637	6881	7009	6475	6494	
45584793	N_{lb}	73	73	515	84	1880	106	3636	150	5911	260	0.89
	E	0.57	0.57	0.54	0.64	0.39	0.70	0.29	0.74	0.21	0.76	

Table 1: Experimental results obtained using 8192 CM-2 processors. N_{expand} is the number of node expansion cycles, N_{lb} is the number of load balancing phases and E is the efficiency. The last column contains values for the static trigger obtained using the optimal static triggering equation.

P increases, x_o decreases, meaning that the efficiency of the algorithm decreases when P increases. Also as the ratio $\frac{t_{lb}}{U_{calc}}$ increases (*i.e.* performing a load balancing phase gets relatively more expensive), the value of x_o decreases and visa versa. Finally as α decreases (*i.e.* the work splitting scheme gets worse), the value of x_o also decreases implying that the overall efficiency drops as the alpha-splitting mechanism becomes worse.

5 Static Triggering: Experimental Results

We solved various instances of the 15-puzzle problem [24] taken from [14], on a CM-2 massively parallel SIMD computer. 15-puzzle is a 4×4 square tray containing 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. An instance of the problem consists of an initial position and a specified goal position. The goal is to transform the initial position into the goal position by sliding the tiles around. The 15-puzzle problem is particularly suited for testing the effectiveness of dynamic load balancing schemes, as it is possible to create search spaces of different sizes (W) by choosing appropriate initial positions. IDA* is the best known sequential depth-first-search algorithm to find optimal solution paths for the 15-puzzle problem [14], and generates highly irregular search trees. We have parallelized IDA* to test the effectiveness of the various load balancing algorithms. The same algorithm was also used in [27, 22]. Our parallel implementations of IDA* find all the solutions of the puzzle up to a given tree depth. This ensures that the number of nodes expanded by the serial and the parallel search is the same, and thus we avoid having to consider superlinear speedup effects [30, 27, 22].

We obtained experimental results using both the *nGP* and the *GP* matching schemes for different values of static threshold x. In our implementation, each node expansion cycle takes about 10ms while each load balancing cycle takes about 34ms. Every time work is split we transfer the node at the bottom of the stack. For the 15-puzzle problem, this appears to provide a reasonable alpha-splitting mechanism. In calculating efficiencies, we used the average node expansion cycle time of parallel IDA* as an approximation of the sequential node expansion cost. Because of higher node expansion cost associated with SIMD parallel computers [27], the actual efficiencies are lower by a constant ratio than those presented here. However, this does not affect the relative comparison of any of these schemes.

Some of these results are shown in Table 1. All the timings in this table have been taken on 8k processors. From the results shown in this table, we clearly see how *GP* and *nGP* relate to each other. When $x = 0.50$ both algorithms perform similarly, which is expected because in this case both *GP* and *nGP* have $V(P) = 1$. Where, as predicted by our theoretical analysis, the difference between the performance of *GP* and *nGP* increases as x increases. From the results shown in Table 1, we also see that the relative performance of *GP* versus *nGP* increases as W increases. The reason for that is explained in [13]. Finally, the last column in Table 1 shows the values of static trigger x_o, obtained from eqn (15) using $\alpha = 0.5$. These values for the static trigger are similar to the static trigger values where *GP* performs the best for each problem.

We constructed experimental isoefficiency graphs for both *nGP*-S^x and *GP*-S^x. These graphs are shown in figures 3, 4 and 5. These graphs were obtained by performing a large number of experiments for a range of W and P, and then collecting the points with equal efficiency. From Fig. 3, we see that the isoefficiency of *GP*-$S^{0.80}$ on the CM-2 is $O(P \log P)$. Figures 4 and 5 (for *nGP*-$S^{0.70}$ and *nGP*-$S^{0.60}$) show the dependence of the isoefficiency of *nGP* on the triggering threshold x. As x increases, the isoefficiency functions become worse. The effect is more prominent for higher efficiencies. For example, the isoefficiency graph for $E = 0.40$ for *nGP*-$S^{0.60}$ is near $O(P \log P)$, but much worse for *nGP*-$S^{0.70}$.

6 Dynamic Triggering: Analytical Results

We analyzed the behavior of both the D^P-triggering and the D^K-triggering schemes. Especially, we analyzed how they perform under a wide range of different search spaces and different load balancing costs. Due to space limitations in this section we only present a summary of our analytical results. The reader should

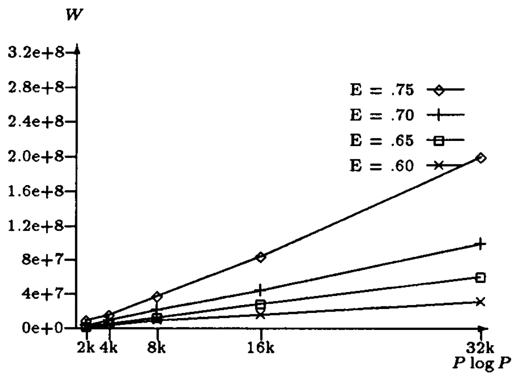

Figure 3: Isoefficiency of *GP*-$S^{0.80}$ scheme

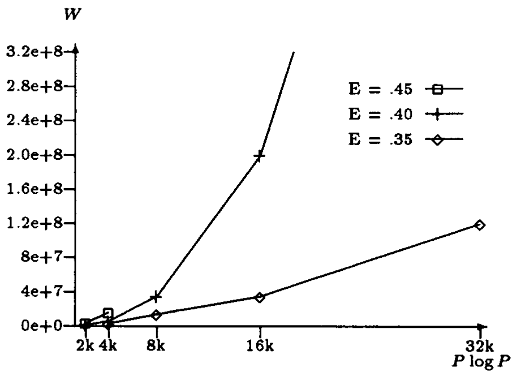

Figure 4: Isoefficiency of *nGP*-$S^{0.70}$ scheme

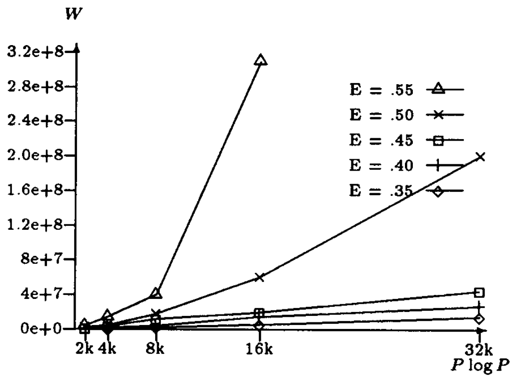

Figure 5: Isoefficiency of *nGP*-$S^{0.60}$ scheme

refer to [13] for a more detailed analysis.

The D^P-triggering schemes appears to be a reasonable dynamic triggering scheme for a large range of search spaces and load balancing costs. Nevertheless it has some serious limitations. The triggering condition, eqn (2), doesn't take into account the number of processors P, but instead is based on the change in the number of busy processors. Hence any load balancing algorithm that uses the D^P-triggering scheme must do the following: a) it has to have an initialization phase where work is distributed to most of the processors and b) during each load balancing cycle, it has to perform multiple work transfers so that most of the processors become active. Also in certain situations, the D^P-triggering scheme will trigger too late or not trigger at all and hence can yield arbitrarily poor efficiencies. We have shown [13] that for any given unstructured tree, if the load balancing cost is above some threshold, then the D^P-triggering scheme will never trigger a load balance and after some period of time, the number of busy processors will become one and no more load balancing cycles will be triggered for the rest of the search. In general, given a load balancing cost, there is a set of search spaces for which the D^P-triggering scheme will perform poorly. The size of this set increases as the load balancing cost increases.

The D^K-triggering scheme as defined from eqn (4) doesn't have the limitations of the D^P-triggering scheme. This triggering condition takes into account the total number of processors P indirectly through the idling overhead; and if the number of busy processors becomes small, the D^K-triggering scheme will soon trigger a load balancing cycle. Hence this scheme neither requires an initial work distribution phase nor multiple work transfers during each load balancing cycle. Also, the D^K-triggering scheme has the property that its overheads will never be more than twice of that for the optimal static triggering scheme S^{x_o}. This means that the efficiency obtained by using the D^K-triggering scheme cannot be much smaller than that obtained by using the S^{x_o}-triggering scheme, although it can be better. For example if the efficiency of the S^{x_o}-triggering scheme is 0.90 then the D^K-triggering scheme's efficiency will be at least 0.82 (*i.e* $\frac{1}{1+2/10}$) and could be even better than 0.90. Note that a similar bound on the overhead cannot be made for the D^P-triggering scheme.

7 Dynamic Triggering, Experimental Results

We implemented all four combinations of the two dynamic triggering schemes D^P and D^K, and the two matching schemes *nGP* and *GP*, in the parallel IDA* to solve the 15-puzzle problem on CM-2. In all cases, the root node is given to one of the processors and static triggering with $x = 0.85$ is used until 85% of the processors became active. Thus in the initial distribution phase, each node expansion cycle was followed by a work distribution cycle until 85% of the processors had work. After the initialization phase, triggering was done using the respective dynamic triggering schemes. The results are summarized in Table 2.

Dynamic Trigger		D^P-triggering		D^K-triggering	
W	Metric	nGP	GP	nGP	GP
2488958	N_{expand}	685	596	552	500
	$^*N_{lb}$	137	100	83	68
	E	0.29	0.34	0.37	0.42
9076121	N_{expand}	2002	1778	1758	1467
	$^*N_{lb}$	161	110	186	114
	E	0.45	0.52	0.46	0.60
21540929	N_{expand}	4436	3894	4002	3200
	$^*N_{lb}$	214	116	378	183
	E	0.52	0.62	0.49	0.69
45584793	N_{expand}	8682	7517	8014	6406
	$^*N_{lb}$	314	137	669	279
	E	0.57	0.70	0.54	0.77

Table 2: Experimental results obtained using 8192 CM-2 processors using various dynamic triggering schemes. N_{expand} is the number of node expansion cycles, $^*N_{lb}$ is the number of work transfers and E is the efficiency. Note that for the D^K-triggering scheme $^*N_{lb}$ is equal to the number of load balancing phases.

From the results shown in this table we can see that for the nGP matching scheme, the D^K-triggering scheme performs slightly worse than the D^P-triggering scheme for larger problems. For the GP matching scheme, D^K-triggering performs consistently better that D^P-triggering for all problems. Also the GP matching scheme constantly outperforms nGP for both dynamic triggering schemes as it does for static triggering. Comparing the two dynamic triggering schemes in Table 2, with the static triggering scheme in Table 1, we see that GP-D^K performs as good as the GP-S^x schemes using optimal trigger values for each problem.

We constructed experimental isoefficiency graphs for all four combinations of matching schemes and dynamic triggering schemes. These graphs are shown in figures 6, 7, 8 and 9. From figures 6 and 7, we see that for the GP matching scheme, the asymptotic scalability of the two dynamic triggering schemes is similar, and is $O(P \log P)$. However, GP-D^P has a somewhat higher constant than GP-D^K. In the case of the nGP matching scheme, for $E \leq 0.55$ the isoefficiency of both nGP-D^K (Fig. 8) and nGP-D^P (Fig. 9) is $O(P \log P)$ while for higher efficiencies it becomes worse. The reason is that the nGP matching scheme leads to a more uneven distribution of the search tree compared to the GP matching scheme.

8 Related Work

Powley, Korf and Ferguson [27, 28] present load balancing algorithms which have different triggering and matching schemes. Their matching scheme, min-f ordering, is identical to nGP or GP when the number of active processors is less than the number of idle ones; but when the active processors are more than the idle ones, work is given out from processors that have nodes with smaller f-value (*i.e.*, the lower bound on the cost of the node). The primary motivation behind the min-f ordering is that nodes with smaller f-values are likely to represent more work than

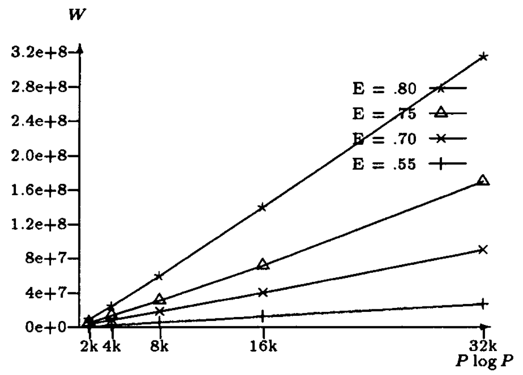

Figure 6: Isoefficiency of GP-D^K scheme

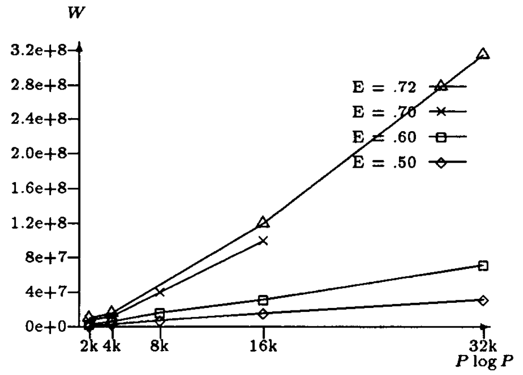

Figure 7: Isoefficiency of GP-D^P scheme

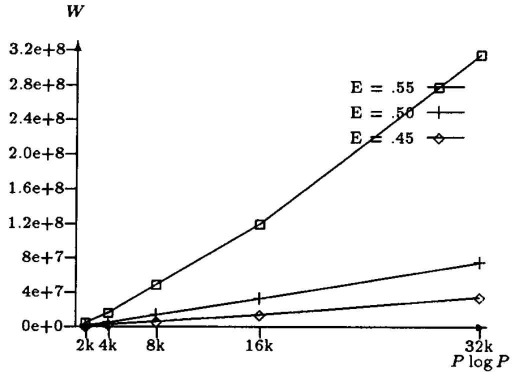

Figure 8: Isoefficiency of nGP-D^K scheme

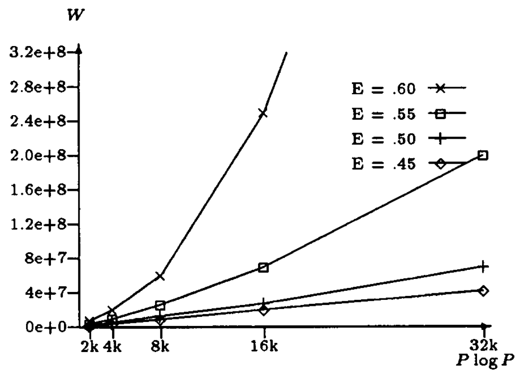

Figure 9: Isoefficiency of nGP-D^P scheme

those with larger f-values. Clearly, the min-f ordering should lead to no more load balancing phases than nGP. It may even lead to fewer number of load balancing phases than GP depending upon how good a predictor the f-value is of the overall load. But unlike GP, it is difficult to put an upper bound on the number of load balancing phases for the min-f ordering scheme. Also, implementation of min-f requires a sorting step which is more time consuming than simple enumeration required by GP. This will become important if the cost of sorting is high compared with the rest of the load balancing phase. In our experiments with the 15-puzzle problem, we found min-f and GP to require about the same number of load balancing phases. But, in our experimental setup, the load balancing cost for min-f is about 2.5 times that of GP. Hence GP leads to somewhat higher efficiencies.

Powley *et. al.* also report experimental results for a matching scheme which randomly matches idle processors to busy processors, and point out that its performance is similar to the min-f scheme. By an analysis similar to that for load balancing schemes for MIMD computers in [16], it can be shown that the upper bound on the asymptotic isoefficiency of the random scheme is worse than that for GP only by a factor of $\log P$.

Powley *et. al.* present three different triggering schemes: static triggering, D^P-triggering and a variation of D^P-triggering with the following modifications: A load balancing cycle is triggered when either the D^P-triggering condition holds or when the number of active processors is less than $P/2$ and the time spent searching is at least half of the time spent in load balancing. These modifications to D^P guarantee that at least one third of the total time is spent searching, and alleviate many of the drawbacks of the original D^P-triggering scheme. Besides load balancing within iterations of IDA*, Powley *et.al.* perform load balancing during the initial work distribution and between iterations of IDA*. These steps may not be applicable, in general, to dynamic distribution of unstructured trees on parallel computers.

Mahanti and Daniels proposed two dynamic load balancing algorithms, FESS and FEGS, in [22, 3]. In both these schemes a load balancing phase is initiated as soon as one processor becomes idle and the matching scheme used is similar to nGP. The difference between FESS and FEGS is that during each load balancing phase FESS performs a single work transfer while FEGS performs as many work transfers as required so that the total number of nodes is evenly distributed among the processors. As our analysis has shown the FESS scheme has poor scalability and because this scheme usually performs as many load balancing phases as node expansion cycles, its performance depends on the ratio $\frac{U_{calc}}{U_{comm}}$. FEGS performs better work distribution thus requires fewer number of load balancing phases. Hence it has better performance than FESS, and its scalability may be close to the GP matching scheme. The memory requirements of FEGS is unbounded, and modifications to handle this problem are discussed in [22].

Frye and Myczkowski proposed two dynamic load balancing algorithms in [5]. The first scheme is similar to nGP-S^x with the difference that each busy processor gives one piece of work to as many idle processors as many pieces of work it has. Clearly this scheme has a poor splitting mechanism. Extending this algorithm in such a way so that the total number of nodes is evenly distributed among the processors results in a scheme similar to FEGS of Mahanti *et. al.*. The second algorithm is based on nearest neighbor communication. In this scheme after each node expansion cycle the processors that have work check to see if their neighbors are idle. If this is the case then they transfer work to them. This scheme is similar to the nearest neighbor load balancing schemes for MIMD machines. As shown in [18] the isoefficiency for a hypercube is $\Omega(P^{\log_2 \frac{1+\frac{1}{\alpha}}{2}})$, while the isoefficiency for a mesh is $\Omega(c^{\sqrt{P}})$ where $c > 1$. Hence, this algorithm is sensitive to the quality of the alpha-splitting mechanism.

References

[1] S. Arvindam, Vipin Kumar, and V. Nageshwara Rao. *Efficient Parallel Algorithms for Search Problems: Applications in VLSI CAD.* In *Proc. of the Frontiers 90 Conf. on Massively Parallel Computation*, October 1990.

[2] S. Arvindam, Vipin Kumar, V. Nageshwara Rao, and Vineet Singh. *Automatic test Pattern Generation on Multiprocessors. Parallel Computing*, 17, number 12:1323–1342, December 1991.

[3] M. Evett, James Hendler, Ambujashka Mahanti, and Dana Nau. *PRA*: A Memory-Limited Heuristic Search Procedure for the Connection Machine.* In *Proc. of the third symposium on the Frontiers of Massively Parallel Computation*, pages 145–149, 1990.

[4] 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.

[5] Roger Frye and Jacek Myczkowski. *Exhaustive Search of Unstructured Trees on the Connection Machine.* Thinking Machines Corporation Tech. Rep., 1990.

[6] M. Furuichi, K. Taki, and N. Ichiyoshi. *A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI.* In *Proc. of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1990. pp.50-59.

[7] Ananth Grama, Vipin Kumar, and V. Nageshwara Rao. *Experimental Evaluation of Load Balancing Techniques for the Hypercube.* In *Proc. of the Parallel Computing 91 Conf.*, 1991.

[8] Anshul Gupta and Vipin Kumar. *On the scalability of FFT on Parallel Computers.* In *Proc. of the Frontiers 90 Conf. on Massively Parallel Computation*, October 1990. An extended version of the paper will appear in IEEE Trans. on Parallel and Distributed Systems, 1993.

[9] John L. Gustafson, Gary R. Montry, and Robert E. Benner. *Development of Parallel Methods for a 1024-Processor Hypercube. SIAM Journal on Scientific and Statistical Computing*, 9 No. 4:609–638, 1988.

[10] W. Daniel Hillis. *The Connection Machine.* MIT Press, 1991.

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

[12] Laveen Kanal and Vipin Kumar. *Search in Artificial Intelligence.* Springer-Verlag, New York, 1988.

[13] George Karypis and Vipin Kumar. *Unstructured Tree Search on SIMD Parallel Computers.* Tech. Rep. TR-92–21, Computer Science Department, University of MN, 1992.

[14] Richard E. Korf. *Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. Artificial Intelligence*, 27:97–109, 1985.

[15] 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. Revised version appears in the second edition of the encyclopedia to be published in 1992.

[16] Vipin Kumar, Ananth Grama, and V. Nageshwara Rao. *Scalable Load Balancing Techniques for Parallel Computers.* Tech. Rep., TR-91-55, Computer Science Department, University of MN, 1991.

[17] Vipin Kumar and Anshul Gupta. *Analyzing Scalability of Parallel Algorithms and Architectures.* Tech. Rep., TR-91-18, Computer Science Department, University of MN, June 1991. A short version of the paper appears in the Proc. of the 1991 Int. Conf. on Supercomputing, Germany, and as an invited paper in the Proc. of 29th Annual Allerton Conf. on Communuication, Control and Computing, Urbana,IL, October 1991.

[18] Vipin Kumar, Dana Nau, and Laveen Kanal. *General Branch-and-bound Formulation for AND/OR Graph and Game Tree Search.* In Laveen Kanal and Vipin Kumar, editors, *Search in Artificial Intelligence.* Springer-Verlag, New York, 1988.

[19] Vipin Kumar and V. Nageshwara Rao. *Parallel Depth-First Search, Part II: Analysis. Int. Journal of Parallel Programming*, 16, 1987.

[20] Vipin Kumar and Vineet Singh. *Scalability of Parallel Algorithms for the All-Pairs Shortest Path Problem: A Summary of Results.* Extended version appears in Journal of Parallel and Distributed Processing (special issue on massively parallel computation), Volume 13, 1991.

[21] Karp R. M. *Challenges in Combinatorial Computing.* To appear January 1991.

[22] A. Mahanti and C. Daniels. *SIMD Parallel Heuristic Search. To appear in Artificial Intelligence*, 1992.

[23] V. Nageshwara Rao and Vipin Kumar. *Parallel Depth-First Search, Part I: Implementation. Int. Journal of Parallel Programming*, 16, 1987.

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

[25] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization, Algorithms and Complexity.* Prentice Hall, 1982.

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

[27] C. Powley, R. Korf, and C. Ferguson. *IDA* on the Connection Machine. To appear in Artificial Intelligence*, 1992.

[28] Curt Powley and Richard E. Korf. *SIMD and MIMD Parallel Search.* In *Proc. of the AAAI Spring Symp.*, pages 49–53, 1989.

[29] Abhiram Ranade. *Optimal Speedup for Backtrack Search on a Butterfly Network.* In *Proc. of the Third ACM Symp. on Parallel Algorithms and Architectures*, 1991.

[30] V. Nageshwara Rao and Vipin Kumar. *On the Efficicency of Parallel Backtracking. IEEE Trans. on Parallel and Distributed Systems*, (to appear), 1992. available as a tech. rep. TR 90-55, Computer Science Department, University of MN.

[31] Jasec Myczkowski Roger Frye. *Load Balancing Algorithms on the Connection Machine and their Use in Monte-Carlo Methods.* In *Proc. of the Unstructured Scientific Computation on Multiprocessors Conf.*, 1992.

[32] Wei Shu and L. V. Kale. *A Dynamic Scheduling Strategy for the Chare-Kernel System.* In *Proc. of Supercomputing 89*, pages 389–398, 1989.

[33] Benjamin W. Wah and Y. W. Eva Ma. *MANIP - A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems. IEEE Trans. on Computers*, c–33, May 1984.