Unstructured Tree Search on SIMD Parallel Computers*

George Karypis and Vipin Kumar
Department of Computer Science,
University of Minnesota
Minneapolis, MN 55455

karypis@cs.umn.edu
kumar@cs.umn.edu

TR 92-21, April 1992

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.

---

*This work was supported by Army Research Office grant #28408-MA-SDI to the University of Minnesota 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.

1 Introduction

Tree search is central to solving a variety of problems in artificial intelligence [14, 29], combinatorial optimization [13, 22], operations research [27] and Monte-Carlo evaluations of functional integrals [35]. 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, 5, 7, 24, 25, 28, 31, 36, 37, 39, 40], whereas until recently, it was common wisdom that such irregular problems cannot be solved on large-scale SIMD parallel computers [22].

Recent research has shown that data parallel SIMD architectures can also be used to implement parallel tree search algorithms effectively. Frye and Myczkowski [6] presents an implementation of a depth-first tree (DFS) search algorithm on the CM-2 for a block puzzle. Powley, Korf and Ferguson [30] and Mahanti and Daniels [23] present parallel formulations of a tree search algorithm IDA*, for solving the 15 puzzle problem on CM-2.

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. The reason is that 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 [18]. 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 [10, 11, 18, 20]. The isoefficiency metric has been found to be quite useful in characterizing scalability of a number of algorithms [9, 21, 32, 38, 41, 42]. In particular, it has helped determine optimal load balancing schemes for tree search for a variety of MIMD architectures [20, 8, 17].

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 pre-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. 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. Section 9 provides a summary and concluding remarks.

2 Dynamic Load Balancing Algorithms for Parallel Search

Specification of a tree search problem includes description of the root node of the tree and a successor-generator-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 [16], IDA* [15], Backtracking [13]. In this paper we only 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 [30, 23, 34] 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 searching phase to the load balancing phase when a triggering condition is satisfied. In the load balancing phase, the busy processors split their work and share it with idle processors. When a goal node is found, all of them quit. If the search space is finite and has no solutions, then eventually all the processors would run out of work, and 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 the one proposed in [30, 23]. In this algorithm, the triggering condition is computed after each node expansion cycle in the searching phase. If this condition is satisfied, then a load balancing phase is initiated. In the load balancing phase, idle processors are matched one-on-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 [30, 34] is to trigger a load balancing phase when the

[2]This is done using the rendezvous allocation scheme described in [12].

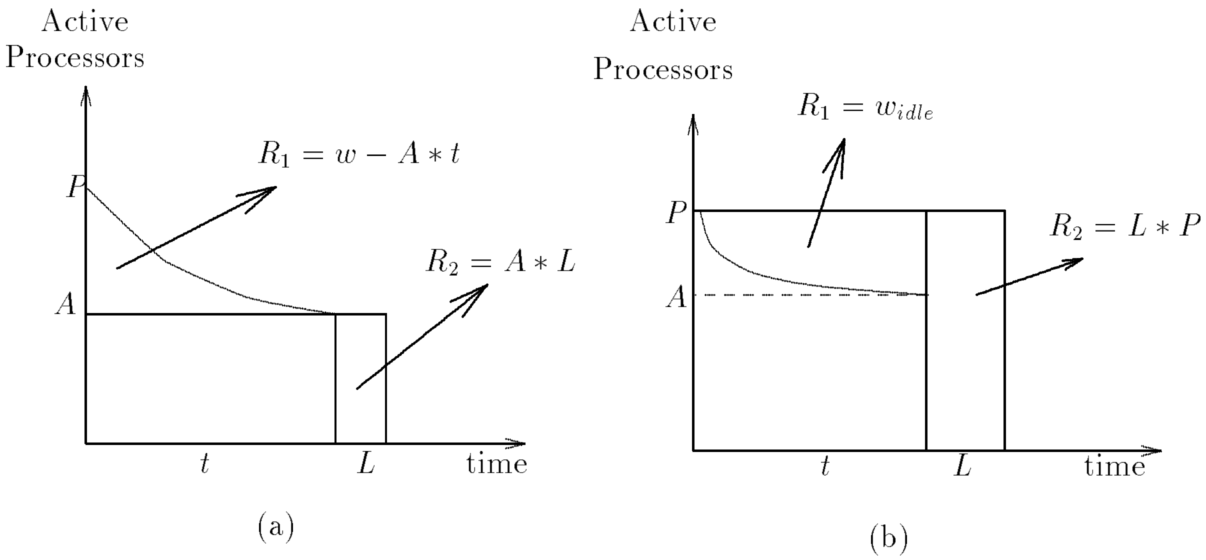

Figure 1: A graphical representation of the triggering conditions for the D^P-triggering and for the D^K-triggering schemes.

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:

$$A \leq xP \tag{1}$$

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 proposed and analyzed by Powley, Ferguson and Korf [30]. For the rest of this paper we will refer to it as the D^P-triggering scheme. D^P-triggering works as follows: Let w be the work done in processors-seconds[3], let t be the elapsed time (in seconds) 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

[3]This is the sum of the time spent in seconds by all the processors doing node expansions during the current search phase.

the condition that triggers a load balance is:

$$\frac{w}{t+L} \geq A \tag{2}$$

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. We can better understand the triggering condition for D^P if we rewrite equation (2) as:

$$w - A * t \geq A * L \tag{3}$$

From this equation and Figure 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 at 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

	Processors	1	2	3	4	5	6	7	8
example 1									
	state	B	B	B	B	B	I	I	B
	global pointer					$\uparrow$			
	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	$\uparrow$							
	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.

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.

Our new dynamic triggering scheme takes a different approach than the D^P-triggering scheme. Our triggering scheme balances the idle time of the processors during the search phase and the cost of the next load balancing phase. 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:

$$w_{idle} \geq L * P \tag{4}$$

Figure 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 $R_1 \geq R_2$. For the rest of this paper we will refer to this dynamic triggering scheme as the D^K-triggering scheme.

2.3 Summarizing the various Schemes

We studied all possible combinations of the matching and triggering schemes presented so far. For the D^P-triggering scheme to perform well, it is necessary that multiple work transfers are performed within each load balancing phase until all (or most) of the processors receive work [30] (the reason for this is given in Section 6.1). Hence every time we use the D^P-triggering scheme, we perform multiple work transfers. All load balancing schemes are listed in Table 1. These schemes differ in the matching scheme, triggering condition and whether or not we perform multiple work transfers during each load balancing phase.

Name	Comments	Number of work transfers in a single load balancing phase
nGP-S^x	This scheme is similar to [30, 23]	single
nGP-D^P	This scheme is similar to [30]	multiple
nGP-D^K	New scheme	single
GP-S^x	New scheme	single
GP-D^P	New scheme	multiple
GP-D^K	New scheme	single

Table 1: The different dynamic load balancing schemes studied.

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. In other words, some of the nodes (*i.e.* alternatives) from the active processor's stack are removed and added to the idle processor's stack. 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 again and visa versa. 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 [25, 1, 8, 17, 23] 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 [33, 30, 23] 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. This is not an unreasonable assumption, as the stack is a rather compact representation of the search space.
- Total load balancing time T_{lb}: the total time spent in load balancing by all processors in the entire algorithm. $T_{lb} = t_{lb} *$ (number of load balancing phases) $* P$.
- Total idling time T_{idle}: the total time spent idling by all processors in the entire algorithm during the search phase. 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. The reason for this is that in this case it would be difficult to obtain good speedup on the architecture for a large number of processors, unless the problem size being solved is enormously large. 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 [20, 18].

3.3 Cost of each load balancing phase

In both nGP and GP matching schemes, each load balancing phase requires a setup step and a work transfer step. During the setup step we match idle processors to busy processors by using sum-scans [3]. In the case of GP we also perform some bookkeeping calculations, involving sum-scans, in order to maintain the global pointer. The complexity of the sum-scan is $O(\log P)$ for a hypercube and $O(\sqrt{P})$ for a mesh. In computers where there is dedicated hardware for sum-scans this operation can be done in constant time. The work transfer step requires sending data from the busy to idle processors. The complexity of fixed size data transfer among any pair of processors is $O(\log^2 P)$ for a hypercube[4] and $O(\sqrt{P})$ for a mesh. Hence the cost of a load balancing phase for a hypercube is:

$$t_{lb} = O(\log^2 P) \tag{5}$$

and for a mesh is:

$$t_{lb} = O(\sqrt{P}) \tag{6}$$

All our experiments were done on our 32K-processor CM-2 SIMD parallel computer, which contains groups of 16 1-bit processors connected in a hypercube configuration. On CM-2, due to hardware optimization, the cost of sending data from busy to idle processors is a large constant and doesn't change with the number of processors (the biggest configuration of the machine contains 64K processors). The cost of performing sum-scan operations is also constant but a lot smaller than that of performing general communication. Hence, during the analysis, we assume that $t_{lb} = O(1)$. For other values of t_{lb} the isoefficiency functions are presented in Table 6 in Section 9.

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 [5, 20]. 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 Appendix A, the maximum number of load balancing phases necessary in any load balancing algorithm using the alpha-splitting mechanism 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

[4]This is the complexity of performing a general permutation. Depending on the permutation and on the network for general communication the complexity might be $O(\log P)$.

as $V(P)\log W$. In the rest of the analysis, we will use this upper bound as an estimate of the total number of load balancing phases (our experimental results here as well as for MIMD [17] demonstrate that it is a good approximation).

Hence the load balancing overhead T_{lb} is:

$$T_{lb} = P \times V(P)\log W \times t_{lb} \tag{7}$$

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:

$$T_{idle} = \frac{1-x-\beta}{x+\beta} W \times U_{calc} \tag{8}$$

From equation (7) and equation (8) we have that:

$$\begin{aligned} E &= \frac{T_{calc}}{T_{calc}+T_{idle}+T_{lb}} \\ &= \frac{W \times U_{calc}}{W \times U_{calc} + \frac{1-x-\beta}{x+\beta} W \times U_{calc} + P \times V(P)\log W \times t_{lb}} \\ &= \frac{1}{\frac{1}{x+\beta} + \frac{P \times V(P) \log W \times t_{lb}}{W \times U_{calc}}} \end{aligned} \tag{9}$$

From equation (9) 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:

$$W = O(P \times V(P) \log P) \tag{10}$$

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 $V(P)$. Let x be the static trigger. Consider the P processors as being divided into $\frac{1}{1-x}$ non overlapping blocks each containing $(1-x)P$ processors. Because we use a global pointer, during consecutive load balancing phases, the $(1-x)P$ processors that became idle will get work from a different set of $(1-x)P$ processors. Hence, in the worst case, $V(P) = \lceil \frac{1}{1-x} \rceil$, which we approximate by $V(P) = \frac{1}{1-x}$ (in the best case, $V(P) = \frac{1}{2}\frac{1}{1-x}$).

Substituting that value of $V(P)$ in equation (7) and equation (9) we get:

$$T_{lb} = P\frac{1}{1-x} \log W \times t_{lb} \tag{11}$$

$$E = \frac{W \times U_{calc}}{\frac{W}{x+\beta}U_{calc} + P\frac{1}{1-x}\log W \times t_{lb}} \tag{12}$$

Now we substitute $V(P) = \frac{1}{1-x} = O(1)$ in equation (10) to get the isoefficiency function:

$$W = O(P \log P) \tag{13}$$

4.2 Analysis for nGP-S^x

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. If $x \leq 0.5$, (*i.e.* we let half or more of the processors to go idle before we load balance), then in each load balancing phase, each busy processor (among the total P processors) is forced to share its work once with some other idle processor. Hence clearly $V(P) = 1$, and thus the performance of nGP-S^x will be similar to GP-S^x.

When $x > 0.5$, it is possible that some busy processors (those at the beginning of the enumeration sequence) will share their work many times (during successive load balancing phases) before other processors (at the end of the enumeration sequence) will share their work for the first time. As a result, $V(P)$ will become higher.

It is shown in Appendix B that for any x, $0.5 \leq x \leq 1.0$, $V(P) \leq \log^{\frac{2x-1}{1-x}} W$. If we substitute $V(P) = \log^{\frac{2x-1}{1-x}} W$ in equation (7), equation (9), and equation (10) we get:

$$T_{lb} = P \log^{\frac{2x-1}{1-x}} W \log W \times t_{lb} \tag{14}$$

$$E = \frac{W \times U_{calc}}{\frac{W}{x+\beta} \times U_{calc} + P \log^{\frac{2x-1}{1-x}} W \log W \times t_{lb}} \tag{15}$$

$$\text{Isoefficiency function:} \quad W = O(P \log^{\frac{x}{1-x}} P) \tag{16}$$

Clearly we see that the scalability of *nGP*-S^x becomes worse as the value of x increases. From equation (11) and equation (14), 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*.

In the above analysis recall that the expression for T_{lb} and the isoefficiency functions are upper bounds. In practice the isoefficiency function and T_{lb} can be better than the one derived here. In particular, the number of load balancing cycles in *nGP*-S^x or *GP*-S^x are bounded from above by the number of node expansion cycles. Hence, as x increases, the difference between the number of load balancing cycles for *nGP*-S^x and *GP*-S^x will continue to increase until the number of load balancing cycles of *nGP*-S^x approaches the upper bound mentioned above. Since the number of node expansion cycles is greater for larger problems, this "saturation" effect occurs for higher values of x for larger problems.

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. Let assume that $\beta = 0$, meaning that as soon as we load balance, $(1-x)P$ processors become idle right away. From equation (12):

$$\begin{aligned} E &= \frac{W \times U_{calc}}{\frac{W}{x} U_{calc} + P \frac{1}{1-x} \log W \times t_{lb}} \\ &= \frac{1}{\frac{1}{x} + \frac{1}{1-x} \frac{P \log W}{W} \frac{t_{lb}}{U_{calc}}} \end{aligned} \tag{17}$$

To maximize E, we just have to minimize the denominator. The denominator is a $\cup$

shaped graph; therefore it has a minimum point. To obtain that, we set the derivative equal to 0 and solve for x, giving us the optimal static trigger:

$$x_o = \frac{1}{\sqrt{\frac{P}{W} \log_{\frac{1}{1-\alpha}} W \times \frac{t_{lb}}{U_{calc}}} + 1} \tag{18}$$

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 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 balance gets relatively more expensive), the value of x_o decreases and visa versa. Finally as α decreases (*i.e.* the work splitting scheme is getting worse), the value of x_o also decreases implying that the overall efficiency drops as the alpha-splitting mechanism becomes worse.

From equation (18) we can calculate the value of the optimal static trigger if we know α, the ratio $\frac{t_{lb}}{U_{calc}}$ W and P. The equation itself is not too sensitive on α and any reasonable approximation should be acceptable. The ratio of the load balancing cost over the node expansion cost can be calculated experimentally. Given this ratio, we can calculate values for x_{opt} for any combination of P and W. As our experimental results in Section 5 show, the experimentally obtained value of x_o is close to the value obtained from equation (18). In general, when $\beta > 0$, the value of the optimal static trigger will be smaller than the one given by equation (18)[5].

5 Static Triggering: Experimental Results

We solved various instances of the 15-puzzle problem [26] taken from [15], 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

[5] We calculated the value for the optimal static trigger for the case where $\beta = \frac{1-x}{2}$ (i.e. the number of active processors decrease s following a linear function) and the difference between x_o for $\beta = 0$ and x_o for $\beta = \frac{1-x}{2}$ was relatively small.

Static Trigger		0.50		0.60		0.70		0.80		0.90		Analytical
W	Metric	nGP	GP	nGP	GP	nGP	GP	nGP	GP	nGP	GP	trigger, x_o
	N_{expand}	198	198	181	174	164	161	151	150	153	142	
941852	N_{lb}	54	54	77	59	119	69	138	88	151	122	0.82
	E	0.52	0.52	0.53	0.58	0.53	0.60	0.55	0.61	0.52	0.59	
	N_{expand}	606	606	542	535	459	486	420	445	409	417	
3055171	N_{lb}	59	59	111	62	234	76	353	98	408	152	0.89
	E	0.59	0.59	0.63	0.66	0.67	0.72	0.65	0.77	0.64	0.78	
	N_{expand}	1155	1155	1022	1029	894	936	809	863	774	805	
6073623	N_{lb}	56	56	133	63	336	78	577	104	736	170	0.92
	E	0.63	0.63	0.69	0.70	0.71	0.76	0.70	0.82	0.67	0.85	
	N_{expand}	2969	2969	2657	2652	2339	2422	2109	2240	2015	2099	
16110463	N_{lb}	52	52	177	61	655	75	1303	101	1756	172	0.95
	E	0.66	0.66	0.72	0.73	0.75	0.80	0.74	0.86	0.71	0.91	

Table 2: 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.

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 [15], 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 [30, 23]. 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 [33, 30, 23].

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 30ms while each load balancing phase takes about 13ms. Every time work is split we transfer the node at the bottom of the stack, for the 15-puzzle, 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, the actual efficiencies will be lower by a constant ratio than those presented here. However, this does not change the relative comparison of any of these schemes.

Some of these results are shown in Table 2. 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

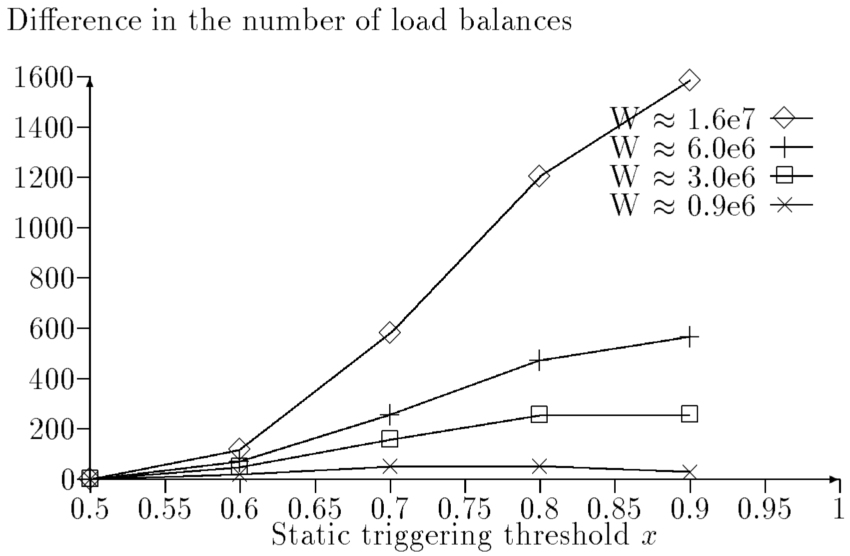

Figure 3: Graph of the difference in the number of load balancing phases performed by *nGP* and *GP* with respect to the static threshold x, for the instances of the 15-puzzle problem shown in Table 2.

in this case both *GP* and *nGP* have $V(P) = 1$. As x increases, the gap in the performance of *nGP* and *GP* increases. This gap is more prominent for larger W. The relation between the number of load balancing phases performed by *nGP* and *GP*, for increasing values of x and W, as discussed in Section 4.2, can be better seen in Figure 3. In this graph we plotted the difference in the number of load balancing phases performed by *nGP* and *GP* with respect to x for the four problems shown in Table 2.

fewer than this

We constructed experimental isoefficiency graphs for both $nGP\text{-}S^x$ and $GP\text{-}S^x$. Those graphs are shown in Figure 4. 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 Figure 4a, we see that the isoefficiency of $GP\text{-}S^{0.90}$ on the CM-2 is $O(P \log P)$. Figures 4b, 4c and 4d (for $nGP\text{-}S^{0.90}$, $nGP\text{-}S^{0.80}$ and $nGP\text{-}S^{0.70}$) 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 $E = 0.72$ for $nGP\text{-}S^{0.70}$ is near $O(P \log P)$, but much worse for $nGP\text{-}S^{0.80}$ and $nGP\text{-}S^{0.90}$. But, the isoefficiency graphs for small efficiencies such as $E = 0.50$, are near $O(P \log P)$ in all cases. The reason is that for small problems, the number of load balances

W								
941852	x	0.79	0.80	0.81	**0.82**	0.83	0.84	0.85
	E	0.60	0.61	0.61	0.61	0.60	0.60	0.59
3055171	x	0.86	0.87	0.88	**0.89**	0.90	0.91	0.92
	E	0.75	0.77	0.77	0.78	0.78	0.77	0.75
6073623	x	0.89	0.90	0.91	**0.92**	0.93	0.94	0.95
	E	0.85	0.85	0.85	0.84	0.84	0.83	0.82
16110463	x	0.92	0.93	0.94	**0.95**	0.96	0.97	0.98
	E	0.90	0.91	0.91	0.89	0.89	0.87	0.85

Table 3: Efficiencies for triggering values around the value for x calculated using the optimal static triggering equation.

is bounded by the number of node expansion cycles and is much smaller than the upper bound given by $\log_{\frac{2x-1}{1-x}} W$. This can also be seen in Figure 3.

Even though we can see a significant difference in the number of load balancing phases between nGP and GP, the actual efficiencies are relatively similar (the difference is less that 25%). The reason is that in our 15-puzzle implementation, the cost of performing a load balancing phase is considerably less than the cost of the node expansion cycle. The reason is that each CM-2 processor is a slow 1-bit processor. In architectures with more powerful CPU like MASPAR (4-bit processor) and CM-5 (32-bit processor), the relative cost of performing a load balancing phase will be substantially higher than node expansion. In such cases GP will substantially outperform nGP.

The last column in Table 2 shows the values of static trigger x_o, obtained from equation (18). In order to verify that these values are good approximations for x_o, we obtained a number of experimental results for x around the analytically computed value for the optimal static triggering threshold. These results are shown in Table 3. From this table we see that the computed values for x_o are very close to the actual optimal static triggering values.

6 Dynamic Triggering, Analysis Framework

Analyzing dynamic triggering schemes is a lot more complicated than analyzing static triggering. In order to do a precise analysis we need to know the structure of the search tree, something that is almost impossible to know. Nevertheless, we can make some observations about the relative performance of D^P, D^K and S^{x_o} triggering schemes under some reasonable assumptions for the structure of the search space.

6.1 Analysis of the D^P-triggering scheme

Even though the D^P-triggering scheme seems to be a reasonable dynamic triggering scheme, under certain circumstances, it can perform arbitrarily poorly. From equation (3) and Figure 1(a) we see that the D^P-triggering scheme is going to perform a load balancing phase as soon as $R_1 \geq R_2$. From this we can make the following observations:

1. The dynamic triggering equation (3) fails to take into account the total number of processors P, or in other words, the potential rate of work. Due to this limitation this triggering scheme works the best if it is invoked when all the processors are active and if after each load balancing phase they become active again. This is why this scheme requires multiple work transfers during each load balancing phase. We can better understand this if we consider the case where only one processor is active. In this case $R_1 = 0$ for the entire duration of the search and the D^P-triggering scheme will never trigger a load balancing phase (assuming that $L > 0$).

2. If after a load balancing phase, the distribution of work among processors is highly uneven then the number of active processors will fall sharply. In that case the area R_1 will be very small and it will not trigger a load balancing phase for a substantially long period of time. During that time the number of active processors will be very small compared to P, resulting in poor efficiencies. In the worst case the number of active processors will drop down to one and for the rest of the search, the D^P-triggering scheme will never trigger a load balancing phase.

3. If the cost of performing a load balancing phase is high then it will also take a long time before the D^P-triggering scheme triggers a load balancing phase. This is because in this case area R_2 will be quite large; it will take quite some time before R_1 exceeds R_2. For any tree there is a load balancing cost such that the D^P-triggering scheme will never trigger a load balancing phase. To see this, consider Figure 5(a). Let t_1 be the time at which the number of active processors becomes 1, and let $\mathcal{A}= \int_0^{t_1}(W(t)-1)dt$. If $L > \mathcal{A}$, then the D^P-triggering scheme will not trigger a load balancing phase for the rest of the search.

From the above we see that depending on the load balancing cost, there is a set of search spaces such that the D^P-triggering scheme will give us poor efficiencies. The size of that set increases as the load balancing cost increases. In general we expect the D^P-triggering scheme to perform well when the load balancing cost is small compared to node expansion cost and the number of active processors decrease similar to Figure 5(a), and not that well in situations where the load balancing cost is high or where the number of active processors decrease similar to Figure 5(b).

6.2 Analysis of the D^K-triggering scheme

We can analyze the behavior of the D^K-triggering scheme with respect to the optimal static triggering scheme S^{x_o}. Let $\mathcal{D}(\sqcup)$ be the number of active processors at time t when a dynamic triggering scheme triggers a load balance. Clearly $\mathcal{D}(\sqcup)$ is defined only at discrete values of t, these times at which a load balancing is triggered. We are going to assume that $D(t)$ is a non-increasing function. Let $T_{idle}^{S^{x_o}}$ and $T_{lb}^{S^{x_o}}$ be the sum of the idle time and load balancing time over all the processors for S^{x_o}, and let $T_{idle}^{D^K}$ and $T_{lb}^{D^K}$ be the sum of the idle time and load balancing time over all the processors for the D^K-triggering scheme. From the definition of the D^K-triggering scheme, equation (4), clearly:

$$T_{idle}^{D^K} = T_{lb}^{D^K} \tag{19}$$

Let $\mathcal{D}^{\mathcal{K}}(\sqcup)$ be the triggering function for the D^K-triggering scheme and let $\mathcal{S}^{\S\imath}(\sqcup)$ be the triggering function for the optimal static trigger. We are going to consider the following three cases that are shown at Figure 6.

case 1, $\mathcal{D}^{\mathcal{K}}(\sqcup) = \mathcal{D}^{\mathcal{K}}_{\infty}(\sqcup)$

From Figure 6 we see that the D^K-triggering scheme always triggers at a lower point than x_o and hence it performs fewer load balancing phases. In this case we have that $T_{lb}^{D^K} \leq T_{lb}^{S^{x_o}}$. From equation (19) we have that:

$$T_{idle}^{D^K} + T_{lb}^{D^K} \leq 2 \times T_{lb}^{S^{x_o}}$$

hence:

$$T_{idle}^{D^K} + T_{lb}^{D^K} < 2 \times (T_{idle}^{S^{x_o}} + T_{lb}^{S^{x_o}}) \tag{20}$$

case 2, $\mathcal{D}^{\mathcal{K}}(\sqcup) = \mathcal{D}^{\mathcal{K}}_{\in}(\sqcup)$

From Figure 6 we see that the D^K-triggering scheme always triggers at a higher point than x_o. At any given time, more processors are active in the D^K triggering scheme, thus $T_{idle}^{D^K} \leq T_{idle}^{S^{x_o}}$. From equation (19) we have that:

$$T_{idle}^{D^K} + T_{lb}^{D^K} \leq 2 \times T_{idle}^{S^{x_o}}$$

hence:

$$T_{idle}^{D^K} + T_{lb}^{D^K} < 2 \times (T_{idle}^{S^{x_o}} + T_{lb}^{S^{x_o}}) \tag{21}$$

case 3, $\mathcal{D}^{\mathcal{K}}(\sqcup) = \mathcal{D}^{\mathcal{K}}_{\ni}(\sqcup)$

From Figure 6, we see that the D^K-triggering scheme triggers at a point higher than x_o up to time I and at a point lower than that after time I. For the time interval before I, from equation (20), we know that the overheads of the D^K-triggering are bounded and also for the time interval after I, from equation (21), the overheads are also bounded. Hence for this case we also have that:

$$T_{idle}^{D^K} + T_{lb}^{D^K} < 2 \times (T_{idle}^{S^{x_o}} + T_{lb}^{S^{x_o}}) \tag{22}$$

From the above we see that the overheads of the D^K-triggering scheme, in the worst case are:

$$T_{idle}^{D^K} + T_{lb}^{D^K} \leq 2 \times (T_{idle}^{S^{x_o}} + T_{lb}^{S^{x_o}})$$

Due to this property, the overheads of the D^K-triggering scheme will never be more than twice of that for the S^{x_o}-triggering scheme. Hence, 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 and could be even better than 0.90.

To get an understanding of the relative performance of the D^P and D^K triggering schemes let us consider Figure 5. In situations similar to Figure 5(a), the D^P-triggering scheme will trigger a load balancing phase slightly earlier than the D^K-triggering scheme. On the other hand, in situations similar to Figure 5(b), the D^K-triggering scheme will trigger a load balancing phase significantly earlier than the D^P-triggering scheme. Hence in the worst case, the D^K-triggering scheme will perform slightly worse than the D^P-triggering scheme but in certain cases it will perform considerably better.

6.3 Summary of Dynamic Triggering Results

From the analysis presented above it is clear that reasonable dynamic triggering schemes can be developed for the class of load balancing algorithms discussed here. Particularly, it was shown that even though the D^P-triggering scheme has been shown to perform reasonably well [30], under certain circumstances it can perform arbitrarily poorly. Also, it was shown that the overheads of the D^K-triggering scheme are bounded and they can not be higher than twice the overheads of the optimal static triggering scheme.

From the analysis of the GP and nGP matching schemes for static triggering, we know their scalability depends on the value of static trigger or in other words, how frequent we perform a load balancing phase. It has been shown that an increase in the frequency we perform load balancing phases affects more the scalability of the nGP matching scheme than that of the GP matching scheme. Hence, the scalability of any of the dynamic triggering schemes depends on how frequent we perform load balancing phases. For the GP matching scheme both the D^K and the D^P triggering schemes will yield a scalable algorithm (provided that the D^P-triggering scheme doesn't perform arbitrarily poorly). For the nGP matching scheme depending on the load balancing phases frequency, both the D^P and the D^K schemes can yield either scalable or unscalable algorithms.

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. As stated in Section 6.1, for the D^P-triggering scheme to work, it is essential that most of the processors have work at the beginning of the search phase. For the D^K-triggering scheme, this initialization phase is not required, but we still used it in order to make the comparisons easier. After the initialization phase, triggering was done using the respective dynamic triggering schemes. The results are summarized in Table 4.

From the results shown in this table we can see that the two schemes have quite similar overall performance for the same matching schemes. The D^P-triggering schemes performs more load balancing cycles and fewer node expansion cycles, while the D^K-triggering

Dynamic Trigger		D^P-triggering		D^K-triggering	
W	Metric	nGP	GP	nGP	GP
941852	N_{expand}	153	149	176	164
	$*N_{lb}$	164	100	89	70
	E	0.51	0.58	0.53	0.58
3055171	N_{expand}	441	426	486	440
	$*N_{lb}$	312	143	179	104
	E	0.64	0.76	0.66	0.77
6073623	N_{expand}	842	808	905	819
	$*N_{lb}$	518	170	285	132
	E	0.68	0.83	0.72	0.84
16110463	N_{expand}	2191	2055	2293	2067
	$*N_{lb}$	935	217	598	192
	E	0.75	0.92	0.76	0.92

Table 4: 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.

scheme performs fewer load balancing phases and more node expansion cycles. For the nGP matching scheme, we see that the D^P-triggering scheme performs slightly worse than the D^K-triggering scheme because the difference in the number of load balancing phases for the two triggering schemes is much larger. The overall performance of the two schemes is similar because the cost of load balancing is very small for our problem. Comparing the two dynamic triggering schemes in Table 4, with the static triggering scheme in Table 2, we see that the dynamic triggering schemes perform as good as the optimal static triggering schemes. Also the GP matching scheme constantly outperforms nGP for both dynamic triggering schemes as it does for static triggering.

We constructed experimental isoefficiency graphs for all four combinations of matching schemes and dynamic triggering schemes. These graphs are shown in Figure 7. From Figure 7a and Figure 7b, we see that for the GP matching scheme, the scalability of the two dynamic triggering schemes is almost identical, and is $O(P \log P)$. In the case of the nGP matching scheme, for the D^K-triggering scheme, Figure 7c, the isoefficiency of the algorithm is $O(P \log P)$ while for the D^P-triggering scheme, Figure 7d, the isoefficiency of the algorithm is worse than $O(P \log P)$. As discussed in Section 6.3, the scalability of the nGP matching scheme when dynamic triggering schemes are used depends on the frequency of load balancing phases. In our experiments, the D^P-triggering scheme triggers load balancing phases more frequently that the D^K-triggering scheme, hence yielding less scalable algorithms.

To study the impact of higher load balancing cost, we simulated higher t_{lb} by sending larger than necessary messages and compared the performance of the D^P-triggering and the D^K-triggering schemes for the GP matching scheme. We increased the load balancing cost by a factor of 12 and by a factor of 16. The results are shown in Table 5. From this table we can see that when the load balancing cost was 12 times higher, the efficiency of the D^K-triggering scheme was 23% higher than that for the D^P-triggering and when the load balancing cost was 16 times higher the efficiency of the D^K-triggering scheme was 40% higher. In both cases the efficiency of the D^K-triggering scheme was similar to that of the S^{x_o}-triggering scheme (less by 10%). To better understand what actually happens, we plotted the number

	Actual Cost			12 times higher			16 times higher		
Metric	D^P	D^K	S^{x_o}	D^P	D^K	S^{x_o}	D^P	D^K	S^{x_o}
N_{expand}	310	314	307	505	487	365	615	533	410
N_{lb}	110	83	87	102	44	58	109	45	50
E	0.69	0.71	0.72	0.26	0.32	0.34	0.20	0.28	0.31

Table 5: Experimental results obtained for $W = 2067137$ using GP, for different load balancing costs. The last line shows the optimal efficiencies obtained using static triggering.

of busy processors at each node expansion cycle. These graphs for the actual and for the 16 times higher load balancing costs, are shown in Figure 8. Looking at Figure 8a and Figure 8b (those for the actual load balancing costs) we see that the two dynamic triggering schemes perform quite similar. Looking at Figure 8c and Figure 8d (those for the higher load balancing costs) we see that the D^P-triggering scheme triggers load balancing phases at a lower level than the D^K-triggering scheme does. This is consistent with our observation in Section 6.1 that for high load balancing costs, the D^P-triggering scheme might trigger too late. Also, due to the multiple work transfers in each load balancing phase, the D^P-triggering scheme performs more work transfers than the D^K-triggering scheme. Due to poorer load balancing the D^P-triggering scheme performs more node expansion cycles than the D^K-triggering scheme.

8 Related Work

Mahanti and Daniels proposed two dynamic load balancing algorithms, FESS and FEGS, in [23]. 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 than FESS and due to better work distribution the number of load balancing phases is reduced.

Frye and Myczkowski proposed two dynamic load balancing algorithms in [34]. 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. Also as shown in [23], extending this algorithm in such a way so that the total number of nodes is evenly distributed among the processors the memory requirements of this algorithm become unbounded. 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 [19] 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.

9 Summary of Results and Conclusion

From our investigation, it is clear that parallel search of unstructured trees can efficiently be implemented on SIMD parallel computers. Our new matching scheme *GP* provides substantially higher performance than the pre-existing scheme *nGP* for all triggering mechanisms. In particular, the *GP-*S^x algorithm is highly scalable for all values of the static threshold x. Also, for the *GP-*S^x algorithm we have derived the expression for the optimal threshold as a function of W and P. The isoefficiencies of the various static triggering schemes for different architectures are summarized in Table 6. Our D^K-triggering scheme is guaranteed to perform very similar to the S^{x_o} triggering scheme. This is useful, as the problem size W is not often known, thus making it hard to estimate the optimal static trigger for *GP-*S^x. We have also shown that the performance of the D^P-triggering scheme becomes substantially worse than the D^K-triggering scheme when the load balancing cost becomes relatively high.

Until now, MIMD computers were considered to be better suited for parallel search of

Scheme $\rightarrow$ Architecture $\downarrow$	$nGP\text{-}S^x$	$GP\text{-}S^x$
Hypercube	$O(P \log^{\frac{2-x}{1-x}} P)$	$O(P \log^3 P)$
Mesh	$O(P^{1.5} \log^{\frac{x}{1-x}} P)$	$O(P^{1.5} \log P)$

Table 6: Isoefficiencies for the different matching and static triggering schemes (where $x \geq 0.5$).

unstructured trees than SIMD computers. In light of the results presented in this paper, we see that there are algorithms for parallel search of unstructured trees, with similar scalability, for both MIMD and SIMD computers. The efficiency of parallel search will be lower on SIMD computers because of a) the idling overhead between load balancing phases and b) the higher node expansion cost. As we have seen, the overhead due to idling doesn't significantly hinders the efficiency of parallel search. But the higher node expansion cost, depending on the problem, will set an upper bound on the achievable efficiency. If we consider that the cost of building large scale parallel computers is substantially higher for MIMD than for SIMD, then in terms of cost/performance, SIMD computers might be a better choice.

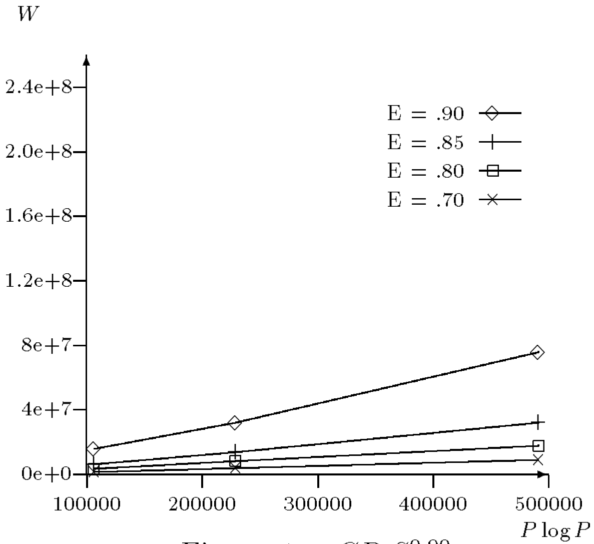

Figure 4a: *GP*-$S^{0.90}$

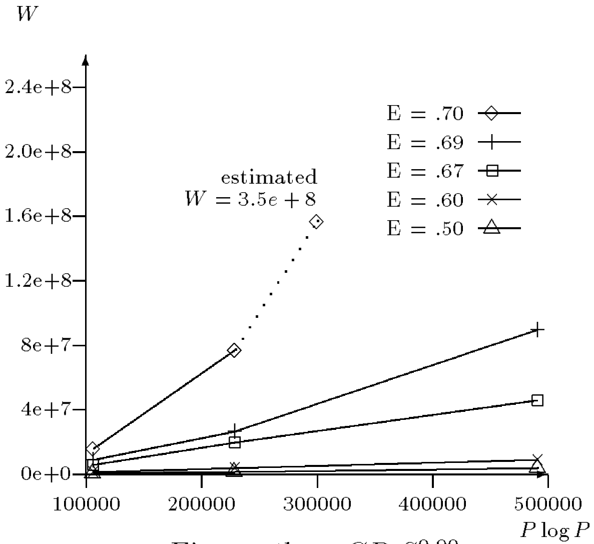

Figure 4b: *nGP*-$S^{0.90}$

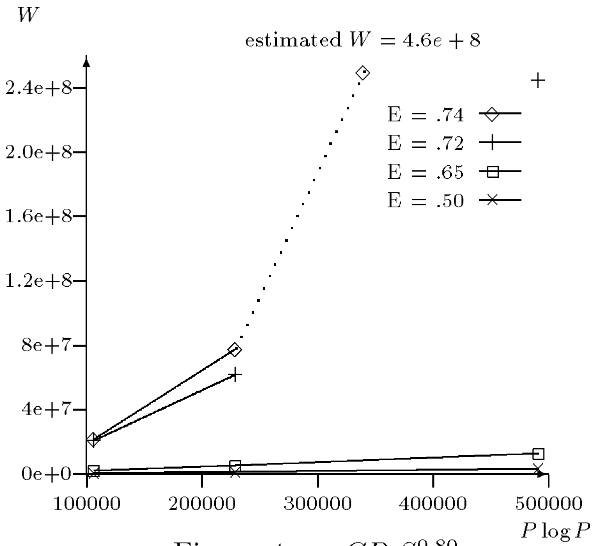

Figure 4c: *nGP*-$S^{0.80}$

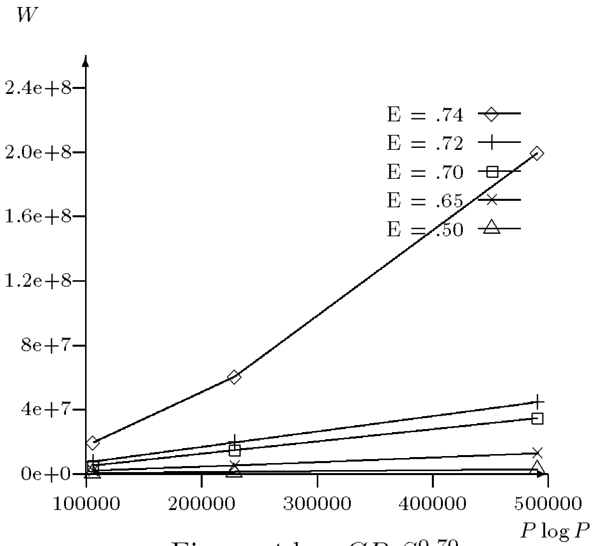

Figure 4d: *nGP*-$S^{0.70}$

Figure 4: Experimental isoefficiency curves for *nGP*-S^x and *GP*-S^x. The reader should note that labels in the graphs represent different efficiencies in different graphs.

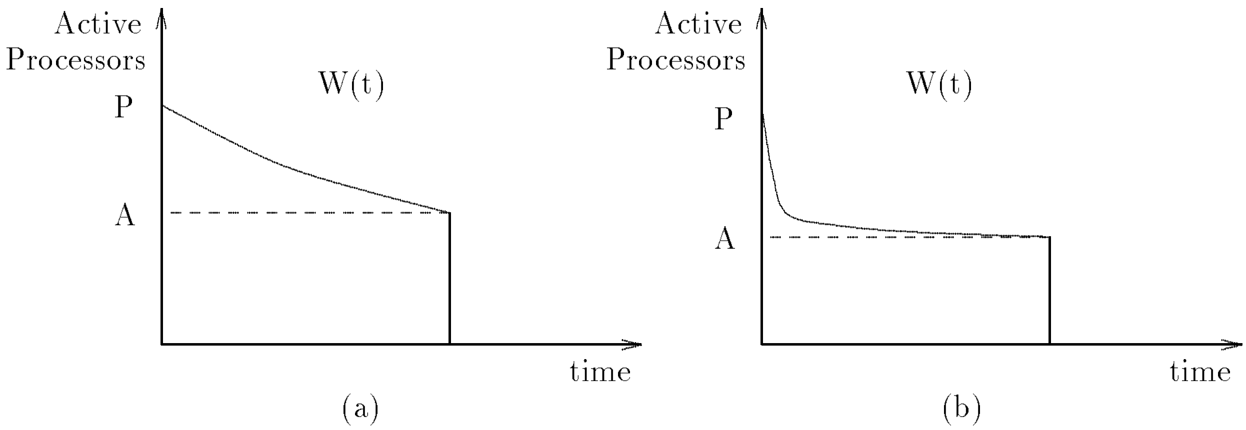

Figure 5: When the number of active processors falls similar to (a) the D^P-triggering scheme will perform well, but when it falls similar to (b) it might lead to poor efficiencies.

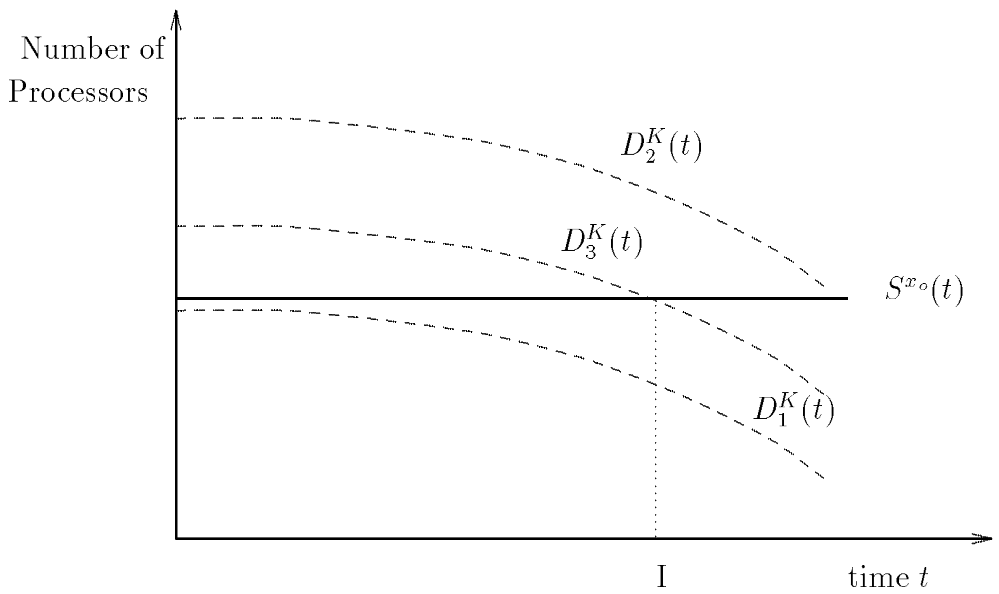

Figure 6: A graphical representation of the different graphs for the interpolated triggering functions $D^K(t)$ and $S^{x_o}(t)$.

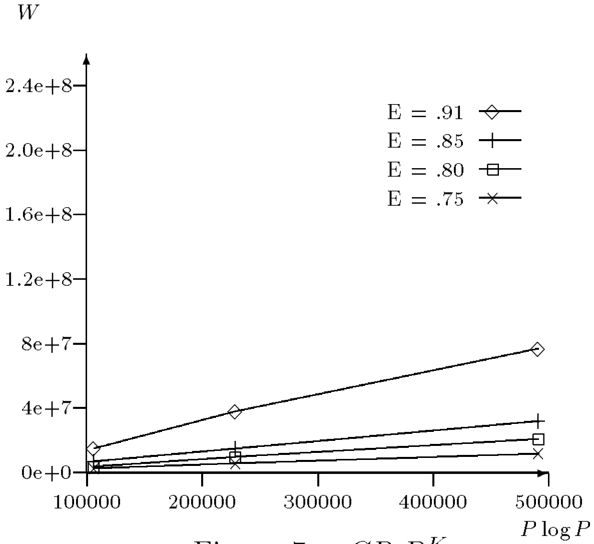

Figure 7a: *GP*-D^K

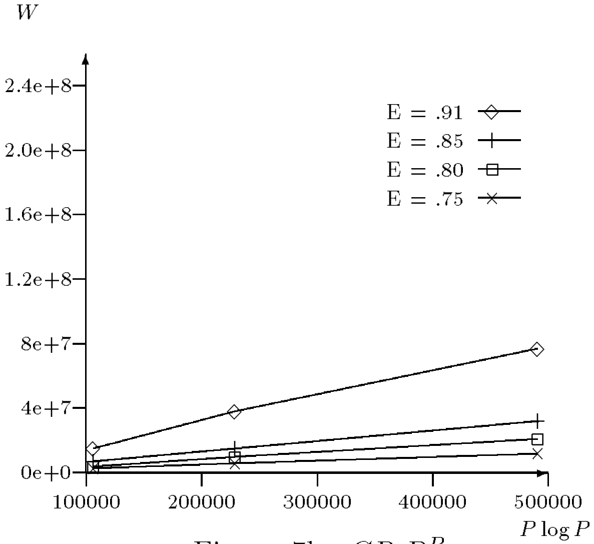

Figure 7b: *GP*-D^P

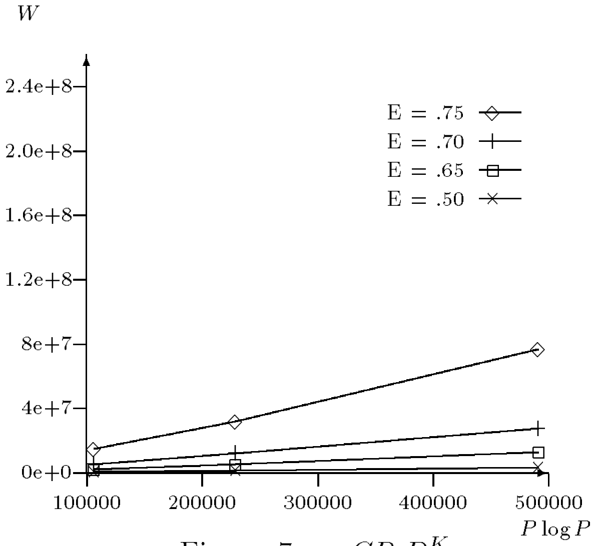

Figure 7c: *nGP*-D^K

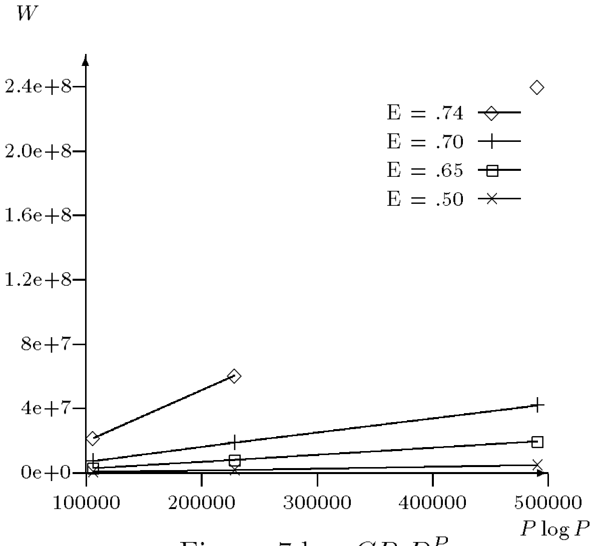

Figure 7d: *nGP*-D^P

Figure 7: Experimental isoefficiency curves for the D^P-triggering and D^K-triggering schemes. The reader should note that labels in the graphs represent different efficiencies in different graphs.

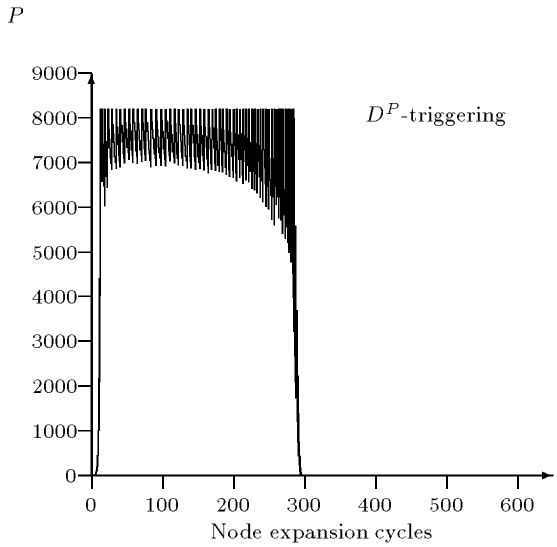

Figure 8a

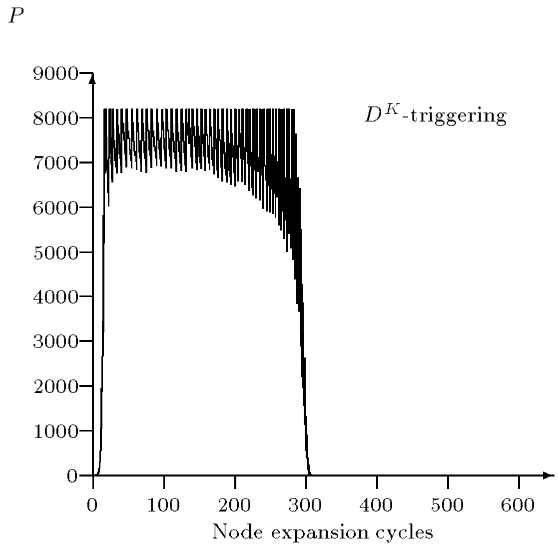

Figure 8b

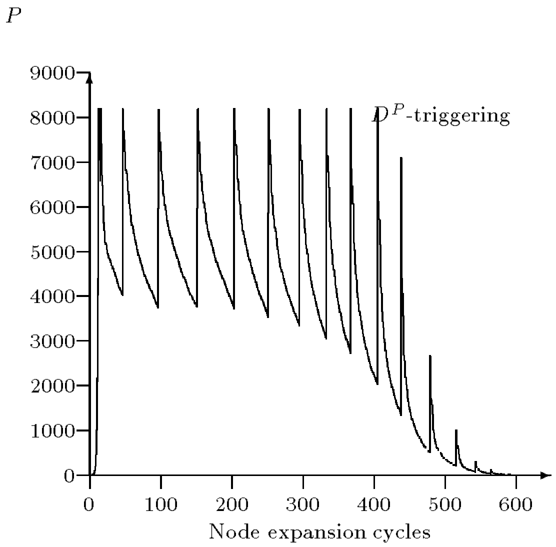

Figure 8c

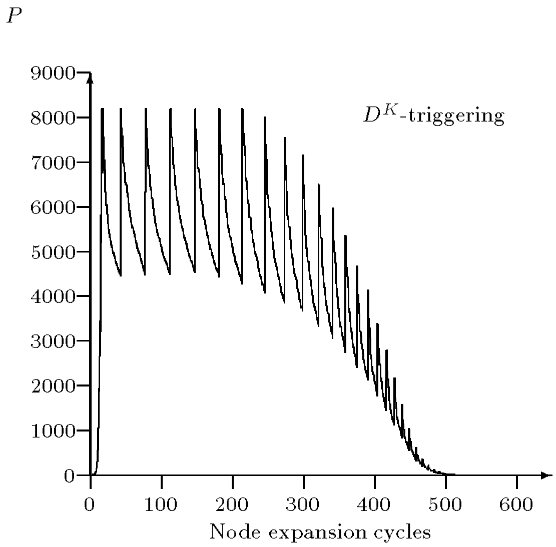

Figure 8d

Figure 8: Number of active processors with respect to node expansion cycles, for the *GP-*D^P and the *GP-*D^K algorithms, for two different load balancing costs.

References

[1] S. Arvindam, Vipin Kumar, and V. Nageshwara Rao. *Efficient Parallel Algorithms for Search Problems: Applications in VLSI CAD.* In *Proceedings of the Frontiers 90 Conference 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] Guy E. Blelloch. *Scans as Primitive Parallel Operations. IEEE Transactions on Computers*, 11:1526–1538, 1989.

[4] Chris Ferguson and Richard Korf. *Distributed Tree Search and its Application to Alpha-Beta Pruning.* In *Proceedings of the 1988 National Conference on Artificial Intelligence*, August 1988.

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

[6] Roger Frye and Jacek Myczkowski. *Exhaustive Search of Unstructured Trees on the Connection Machine.* In *Thinking Machines Corporation Technical Report*, 1990.

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

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

[9] Anshul Gupta and Vipin Kumar. *On the scalability of FFT on Parallel Computers.* In *Proceedings of the Frontiers 90 Conference on Massively Parallel Computation*, October 1990. An extended version of the paper is available as a technical report from the Department of Computer Science, and as TR 90-20 from Army High Performance Computing Research Center, University of Minnesota, Minneapolis, MN 55455.

[10] John L. Gustafson. *Reevaluating Amdahl's Law. Communications of the ACM*, 31(5):532–533, 1988.

[11] 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.

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

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

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

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

[16] 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.

[17] Vipin Kumar, Ananth Grama, and V. Nageshwara Rao. *Scalable Load Balancing Techniques for Parallel Computers.* Technical report, Tech Report 91-55, Computer Science Department, University of Minnesota, 1991.

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

[19] 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.

[20] Vipin Kumar and V. Nageshwara Rao. *Parallel Depth-First Search, Part II: Analysis. International Journal of Parallel Programming*, 16 (6):501–519, 1987.

[21] Vipin Kumar and Vineet Singh. *Scalability of Parallel Algorithms for the All-Pairs Shortest Path Problem: A Summary of Results.* In *Proceedings of the International Conference on Parallel Processing*, 1990. Extended version appears in Journal of Parallel and Distributed Processing (special issue on massively parallel computation), Volume 13, 124-138, 1991.

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

[23] A. Mahanti and C. Daniels. *SIMD Parallel Heuristic Search. To appear in Artificial Intelligence*, 1992. Also available as a technical report, University of Maryland, Computer Science Department.

[24] B. Monien and O. Vornberger. *Parallel Processing of Combinatorial Search Trees.* In *Proceedings of International Workshop on Parallel Algorithms and Architectures*, May 1987.

[25] V. Nageshwara Rao and Vipin Kumar. *Parallel Depth-First Search, Part I: Implementation. International Journal of Parallel Programming*, 16 (6):479–499, 1987.

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

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

[28] Srinivas Patil and Prithviraj Banerjee. *A Parallel Branch and Bound Algorithm for Test Generation.* In *IEEE Transactions on Computer Aided Design*, Vol. 9, No. 3, March 1990.

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

[30] C. Powley, R. Korf, and C. Ferguson. *IDA* on the Connection Machine. To appear in Artificial Intelligence*, 1992. Also available as a technical report, Department of Computer Science, UCLA.

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

[32] S. Ranka and S. Sahni. *Hypercube Algorithms for Image Processing and Pattern Recognition.* Springer-Verlag, New York, 1990.

[33] V. Nageshwara Rao and Vipin Kumar. *On the Efficicency of Parallel Depth-First Search. IEEE Transactions on Parallel and Distributed Systems*, (to appear), 1992. available as a technical report TR 90-55, Computer Science Department, University of Minnesota.

[34] Jasec Myczkowski Roger Frye. *Exhaustive Search of Unstructured Trees on the Connection Machine.* Technical report, Thinking Machines Corporation, 1990.

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

[36] Vikram Saletore and L. V. Kale. *Consistent Linear Speedup to a First Solution in Parallel State-Space Search.* In *Proceedings of the 1990 National Conference on Artificial Intelligence*, pages 227–233, August 1990.

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

[38] Vineet Singh, Vipin Kumar, Gul Agha, and Chris Tomlinson. *Scalability of parallel sorting on mesh multicomputers.* In *Proceedings of the Fifth International Parallel Processing Symposium*, March 1991. Extended version available as a technical report (number TR 90-45) from the department of computer science, University of Minnesota, Minneapolis, MN 55455, and as TR ACT-SPA-298-90 from MCC, Austin, Texas.

[39] Benjamin W. Wah, G.J. Li, and C. F. Yu. *Multiprocessing of Combinatorial Search Problems. IEEE Computers*, June 1985 1985.

[40] 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.

[41] Jinwoon Woo and Sartaj Sahni. *Hypercube Computing : connected Components. Journal of Supercomputing*, 1991.

[42] Jinwoon Woo and Sartaj Sahni. *Computing Biconnected Components on a Hypercube. Journal of Supercomputing*, June 1991.

Appendix A Upper Bound of the Work Transfers

Due to the dynamic nature of the load balancing algorithms being analyzed, it is very difficult to come up with a precise expression for the total communication overheads. In this Section, we review a framework of analysis that provides us with an upper bound on these overheads. This technique was originally developed in the context of Parallel Depth First Search in [5, 20].

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. In all the techniques being analyzed here, the total work is dynamically partitioned among the processors and processors work on disjoint parts independently, each executing the piece of work it is assigned. Initially an idle processor polls around for work and when it finds a processor with work of size W_i, the work W_i is split into disjoint pieces of size W_j and W_k. We assume that the partitioning strategy satisfies the following property:

There is a constant $\alpha > 0$ such that $W_j > \alpha W_i$ and $W_k > \alpha W_i$.

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

After	$V(P)$	requests maximum work available in any processor is less than $(1-\alpha)W$
After	$2V(P)$	requests maximum work available in any processor is less than $(1-\alpha)^2 W$
$\vdots$		
After	$(\log_{\frac{1}{1-\alpha}} \frac{W}{\epsilon})V(P)$	requests maximum work available in any processor is less than ϵ.

Hence total number of transfers $\leq V(P)\log_{\frac{1}{1-\alpha}} W$.

Appendix B Upper Bound on $V(P)$ for nGP

In this section we will calculate the upper bound of $V(P)$, that of the number of work transfers required so that every processor has shared his work at least once, for the nGP algorithm. We will do this by first considering some simple cases and then derive the general formula for $V(N)$.

If we assume that $x \leq 0.5$, (*i.e.* we let half or more of the processors to go idle before we load balance) then in each load balancing phase, each busy processor (among the total P processors) is forced to share its work once with some other idle processor. Hence clearly $V(P) = 1$.

Lets now assume that $x = 0.66$, (i.e. we let a third of the processors to go idle before we load balance). Lets b_1, b_2, b_3 be the first, second and third block of processors each containing $P/3$ non overlapping consecutive processors. Let w_i be the largest piece of work in each of the processor blocks b_i and let assume that block b_3 is the one that becomes idle all the time and requests work from block b_1. It will take roughly $\log w_1$ work transfers to consume the work of the b_1 processor block, The next work request will go to block b_2 (from either b_1 or b_3) and as soon as this happens then all the processors will have been requested at least once. Hence in this case $V(P) = O(\log w_1)$ which in turns gives us that $V(P) \leq O(\log W)$.

Lets now assume that $x = 0.75$. As before let b_i and w_i where $i = 1, 2, 3, 4$, be the processors blocks and the largest units of work they have. It will take $\log w_1$ work transfers in order to consume the work at block b_1. After that, block b_1 will get a work piece of size αw_2 from block b_2. It will take $\log \alpha w_2$ work transfers to consume that piece of work and after that block b_1 will get an other piece of work of size $\alpha(1-\alpha)w_2$ from block b_2 and so on. The number of work transfers in order to consume the work of block b_2 is:

$$\begin{aligned}
\sum_{i=1}^{\log w_2} \log(\alpha(1-\alpha)^{(i-1)} w_2) &= \sum_{i=1}^{\log w_2} (\log \alpha + (i-1)\log(1-\alpha) + \log w_2) \\
&= \sum_{i=1}^{\log w_2} \log \alpha + \sum_{i=1}^{\log w_2} (i-1)\log(1-\alpha) + \sum_{i=1}^{\log w_2} \log w_2 \\
&= \log \alpha * \log w_2 + \log(1-\alpha) * \frac{(\log w_2 - 1)\log w_2}{2} + \log^2 w_2 \\
&= O(\log^2 w_2)
\end{aligned}$$

As soon as we consume w_2 the next work request go to the b_3 block and after that all the processors will have been requested at least once. Therefore in this case:

$$\begin{aligned}
V(P) &\leq \log w_1 + \log^2 w_2 \\
&< \log W + \log^2 W \\
&< O(\log^2 W)
\end{aligned}$$

In the general case when we have $\lceil\frac{1}{1-x}\rceil$ blocks the number of work transfers to consume the w_i pieces of work at each processor block b_i, will be $O(\log^i w_i)$ and in order for all the processors to have been requested at least once we have to consume the work at the first $\lceil\frac{1}{1-x}\rceil - 2$ blocks. In this case $V(P)$ is:

$$\begin{aligned} V(P) &\leq \sum_{i=1}^{\lceil\frac{1}{1-x}\rceil-2} \log^i w_i \\ &\leq O(\log^{\lceil\frac{1}{1-x}\rceil-2} w_{\lceil\frac{1}{1-x}\rceil-2}) \\ &\leq O(\log^{\frac{2x-1}{1-x}} W) \end{aligned} \tag{23}$$

From this equations we see that as the value of x increases the number of work transfers increases substantially. Equation (23) represents the upper bound for the worst case analysis. The worst case occurs when work is transferred from block b_i to block b_{i-1} and from block b_1 to the last block. In any other case, work will be transferred from different blocks (and also overlapping blocks) and this will reduce $V(P)$.