Scalable Load Balancing Techniques for Parallel Computers*

Vipin Kumar and Ananth Y. Grama
Department of Computer Science,
University of Minnesota
Minneapolis, MN 55455
and
Vempaty Nageshwara Rao
Department of Computer Science
University of Central Florida
Orlando, Florida 32816

Abstract

In this paper we analyze the scalability of a number of load balancing algorithms which can be applied to problems that have the following characteristics : the work done by a processor can be partitioned into independent work pieces; the work pieces are of highly variable sizes; and it is not possible (or very difficult) to estimate the size of total work at a given processor. Such problems require a load balancing scheme that distributes the work dynamically among different processors.

Our goal here is to determine the most scalable load balancing schemes for different architectures such as hypercube, mesh and network of workstations. For each of these architectures, we establish lower bounds on the scalability of any possible load balancing scheme. We present the scalability analysis of a number of load balancing schemes that have not been analyzed before. This gives us valuable insights into their relative performance for different problem and architectural characteristics. For each of these architectures, we are able to determine near optimal load balancing schemes. Results obtained from implementation of these schemes in the context of the Tautology Verification problem on the Ncube/2^{TM}[1] multicomputer are used to validate our theoretical results for the hypercube architecture. These results also demonstrate the accuracy and viability of our framework for scalability analysis.

1 Introduction

Load balancing is perhaps the central aspect of parallel computing. Before a problem can be executed on a parallel computer, the work to be done has to be partitioned among different processors. Due to uneven processor utilization, load imbalance can cause poor efficiency. This paper investigates the problem of load balancing in multiprocessors for those parallel algorithms that have the following characteristics.

*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]Ncube/2 is a trademark of the Ncube Corporation.

- The work available at any processor can be partitioned into independent work pieces as long as it is more than some non-decomposable unit.
- The cost of splitting and transferring work to another processor is not excessive. (*i.e.* the cost associated with transferring a piece of work is much less than the computation cost associated with it.)
- A reasonable work splitting mechanism is available; i.e., if work w at one processor is partitioned in 2 parts ψw and $(1-\psi)w$, then $1-\alpha > \psi > \alpha$, where α is an arbitrarily small constant.
- It is not possible (or is very difficult) to estimate the size of total work at a given processor.

Although, in such parallel algorithms, it is easy to partition the work into arbitrarily many parts, these parts can be of widely differing sizes. Hence after an initial distribution of work among P processors, some processors may run out of work much sooner than others; therefore a dynamic balancing of load is needed to transfer work from processors that have work to processors that are idle. Since none of the processors (that have work) know how much work they have, load balancing schemes which require this knowledge (eg. [17, 19]) are not applicable. The performance of a load balancing scheme is dependent upon the degree of load balance achieved and the overheads due to load balancing.

Work created in the execution of many tree search algorithms used in artificial intelligence and operations research [22, 31] and many divide-and-conquer algorithms [16] satisfy all the requirements stated above. As an example, consider the problem of searching a state-space tree in depth-first fashion to find a solution. The state space tree can be easily split up into many parts and each part can be assigned to a different processor. Although it is usually possible to come up with a reasonable work splitting scheme [29], different parts can be of radically different sizes, and in general there is no way of estimating the size of a search tree.

A number of dynamic load balancing strategies that are applicable to problems with these characteristics have been developed [3, 7, 8, 10, 28, 29, 30, 33, 35, 36, 40, 41]. Many of these schemes have been experimentally tested on some physical parallel architectures. From these experimental results, it is difficult to ascertain relative merits of different schemes. 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 [21]. Hence any conclusions drawn on a set of experimental results are invalidated 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 [14, 15, 21, 23]. It may be used to select the best architecture - algorithm combination for a problem under different constraints on the growth of the problem size and the number of processors. It may be used to predict the performance of a parallel algorithm and a parallel architecture for a large number of processors from the known performance on fewer processors. Scalability analysis can also predict the impact of changing hardware technology on the performance, and thus helps in designing better parallel architectures for solving various problems.

Kumar and Rao have developed a scalability metric, called *isoefficiency* , which relates the problem size to the number of processors necessary for an increase in speedup in proportion to the number of processors used [23]. An important feature of isoefficiency analysis is that it succinctly captures the effects of characteristics of the parallel algorithm as well as the parallel architecture on which it is implemented, in a single expression. The isoefficiency metric has been found to be quite useful in characterizing scalability of a number of algorithms [13, 25, 34, 37, 42, 43]. In particular, Kumar and Rao used isoefficiency analysis to characterize the scalability of some load balancing schemes on the shared-memory, ring and hypercube architectures[23] and validated it experimentally in the context of the 15-puzzle problem.

Our goal here is to determine the most scalable load balancing schemes for different architectures such as hypercube, mesh and network of workstations. For each architecture, we establish lower bounds on the scalability of any possible load balancing scheme. We present the scalability analysis of a number of load balancing schemes that have not been analyzed before. From this we gain valuable insights about which schemes can be expected to perform better under what problem and architecture characteristics. For each of these architectures, we are able to determine near optimal load balancing schemes. In particular, some of the algorithms analyzed here for hypercubes are more scalable than those presented in [23]. Results obtained from implementation of these schemes in the context of the Tautology Verification problem on the Ncube/2^{TM} multicomputer are used to validate our theoretical results for the hypercube architecture. The paper also demonstrates the accuracy and viability of our framework for scalability analysis.

Section 2 introduces the various terms used in the analysis. Section 3 presents the isoefficiency metric used for evaluating scalability. Section 4 reviews receiver initiated load balancing schemes, which are analyzed in Section 5. Section 6 presents sender initiated schemes and discusses their scalability. Section 7 addresses the effect of variable work transfer cost on overall scalability. Section 8 presents experimental results. Section 9 contains summary of results and suggestions for future work.

Some parts of this paper have appeared in [11] and [24].

2 Definitions and Assumptions

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

1. Problem size W: the amount of essential computation (*i.e.*, the amount of computation done by the best sequential algorithm) that needs to be performed to solve a problem instance.

2. Number of processors P: number of identical processors in the ensemble being used to solve the given problem instance.

3. Unit Computation time U_{calc}: the time taken for one unit of work. In parallel depth-first search, the unit of work is a single node expansion.

4. Computation time T_{calc}: is the sum of the time spent by all processors in useful computation. (Useful computation is that computation that would also have been performed by the best sequential algorithm.) Clearly, since in solving a problem instance, we need to do W units of work, and each unit of work takes U_{calc} time,

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

5. Running time T_P: the execution time on a P processor ensemble.

6. Overhead T_o: the sum of the time spent by all processors in communicating with other processors, waiting for messages, time in starvation, etc. For a single processor, $T_o = 0$. Clearly,

$$T_{calc} + T_o = P \times T_P$$

7. Speedup S: the ratio $\frac{T_{calc}}{T_P}$.

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

8. Efficiency E: the speedup divided by P. E denotes the effective utilization of computing resources.

$$E = \frac{S}{P} = \frac{T_{calc}}{T_P * P}$$

$$= \frac{T_{calc}}{T_{calc} + T_o} = \frac{1}{1 + \frac{T_o}{T_{calc}}}$$

9. Unit Communication time U_{comm}: the mean time taken for getting some work from another processor. U_{comm} depends upon the size of the message transferred, the distance between the donor and the requesting processor, and the communication speed of the underlying hardware. For simplicity, in the analysis of Sections 5 and 6, we assume that the message size is fixed. Later, in Section 7, we demonstrate the effect of variable message sizes on the overall performance of the schemes.

3 Isoefficiency function : A measure of scalability

If a parallel algorithm is used to solve a problem instance of a fixed size, then the efficiency decreases as 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 (and approaches 1), because the total overhead grows slower than W. For these parallel algorithms, the efficiency can be maintained at a desired value (between 0 and 1) with increasing number of processors, provided the problem size is also increased. We call such algorithms **scalable** parallel algorithms.

Note that for a given parallel algorithm, for different parallel architectures, the problem size may have to increase at different rates w.r.t. P in order to maintain a fixed efficiency. The rate at which W is required to grow w.r.t. P to keep the efficiency fixed is essentially what determines the degree of scalability of the parallel algorithm for a specific architecture. For example, if W is required to grow exponentially w.r.t. 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 speedups 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 only linearly w.r.t. P, then the algorithm-architecture combination is highly scalable and can easily deliver linearly increasing speedups with increasing number of processors for reasonable increments in problem sizes. If W needs to grow as $f(P)$ to maintain an efficiency E, then $f(P)$ is defined to be the **isoefficiency function** for efficiency E and the plot of $f(P)$ w.r.t. P is defined to be the **isoefficiency curve** for efficiency E.

As shown in [21], an important property of a linear isoefficiency function, is that the problem size can be increased linearly with the number of processors while maintaining a fixed execution time if and only if the isoefficiency function is $\Theta(P)$. Also, parallel systems with near linear isoefficiencies enable us to solve increasingly difficult problem instances in only moderately greater time. This is important in many domains (*e.g.* real time systems) in which we have only finite time in which to solve problems.

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 [23, 21].

3.1 Lower bounds on isoefficiency functions for load balancing algorithms for different architectures

For some algorithm - architecture combinations, it is possible to obtain a tighter lower bound than $O(P)$ on the isoefficiency. Consider a problem whose run time on a parallel architecture (comprising of P processors) is given by T_P. By definition, speedup for this problem is given by $\frac{T_{calc}}{T_P}$ and efficiency by $\frac{T_{calc}}{P \times T_P}$. Thus for efficiency to be a constant, $\frac{T_{calc}}{P \times T_P}$ must be constant and thus T_{calc} should grow as $\Theta(P \times T_P)$. Since $T_{calc} = W \times U_{calc}$, W should also grow as $\Theta(P \times T_P)$. Thus, if T_P has a lower bound of $\Omega(G(P))$, we have a lower bound of $\Omega(P \times G(P))$ on the on the isoefficiency function.

For the hypercube architecture, it takes at least $\Omega(\log P)$ time for the work to propagate to all the processors (since the farthest processors are $\log P$ hops away). Thus the execution time for any load balancing algorithm running on a hypercube is lower bounded by $\Omega(\log P)$. Hence the isoefficiency function has a lower bound of $\Omega(P \log P)$. On a mesh, it would take $\Omega(\sqrt{P})$ time for the work to propagate to all processors and consequently the lower bound on isoefficiency is given by $\Omega(P^{1.5})$. For the network of workstations, since for all processors to get work, there have to be at least P messages, and these have to be sent sequentially over the network, the execution time is lower bounded by $\Omega(P)$ and the lower bound on isoefficiency is given by $\Omega(P^2)$.

4 Receiver Initiated Load Balancing Algorithms

In this section, we briefly review receiver initiated load balancing algorithms that are analyzed in section 5. Many of these schemes have been presented in the context of specific architectures [28, 8, 29, 30, 40]. These schemes are characterized by the fact that the work splitting is performed

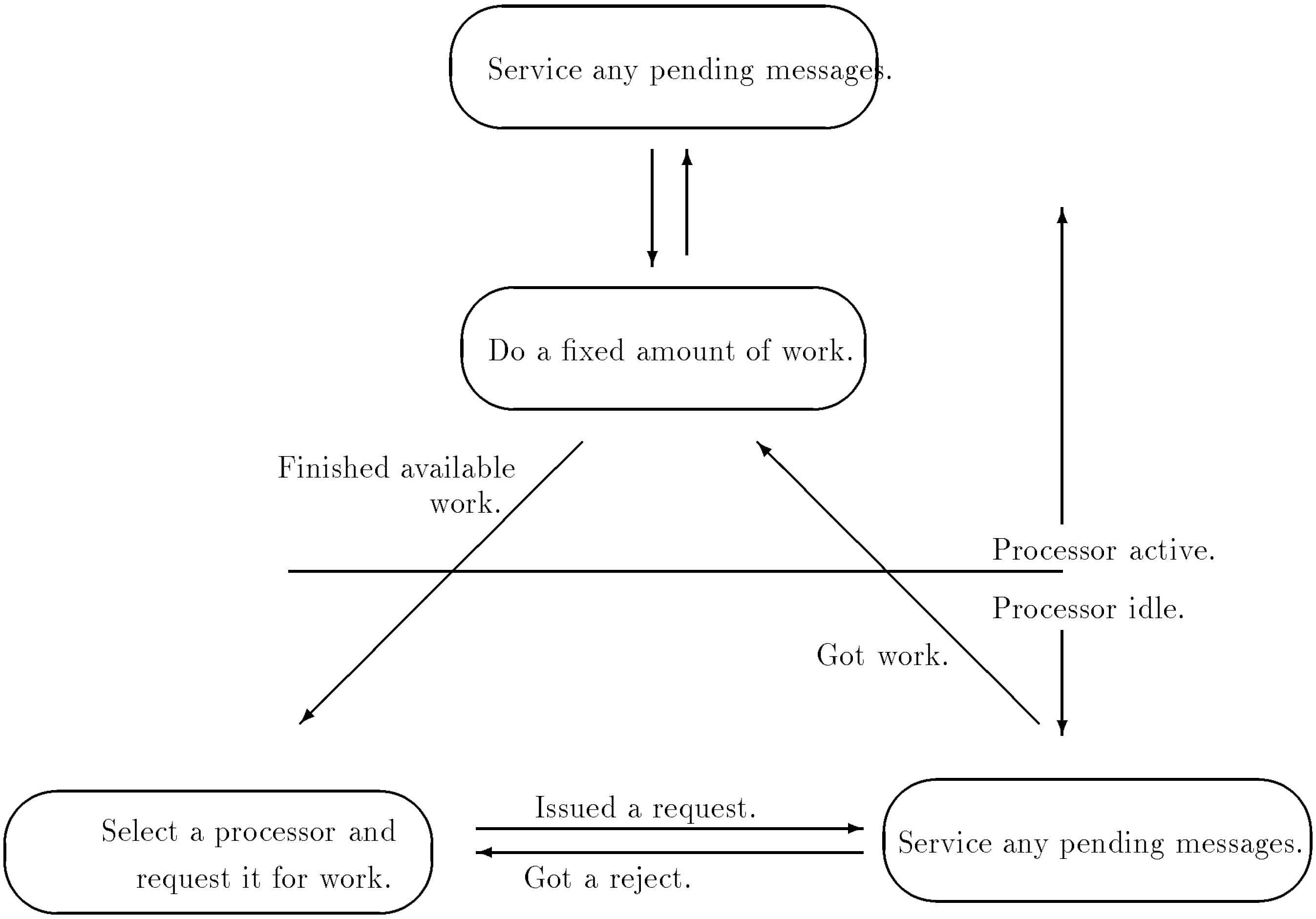


Figure 1: State diagram describing the generic parallel formulation.

only when an idle processor (receiver) requests for work. In all these schemes, when a processor runs out of work, it generates a request for work. The selection of the target of this work request is what differentiates all these different load balancing schemes. The selection of the target for a work request should be such as to minimize the total number of work requests and transfers as well as load imbalance among processors.

The basic load balancing algorithm is shown in the state diagram in Figure 4. At any given instant of time, some of the processors are busy (*i.e.* they have work) and the others are idle (*i.e.* they are trying to get work). An idle processor selects another processor as a target and sends it a work request. If the idle processor receives some work from the target processor, it becomes busy. If it receives a *reject* message (implying that there was no work available at the requested processor), it selects another target and sends it a work request. This is repeated until the processor gets work, or all the processors become idle. While in the idle state, if a processor receives a work request, it returns a *reject* message.

In the busy state, the processor does a fixed amount of work and checks for any pending

$work - request$ messages. If a $work - request$ message was received then the work available at the processor is partitioned into two parts and one of the parts is given to the requesting processor. If too little work is available, then a $reject$ message is sent. When a processor exhausts its own work, it goes into the idle state.

A termination detection algorithm runs in the background. This signals termination whenever a solution is found or all processors run out of work.

4.1 Asynchronous Round Robin (ARR)

In this scheme, each processor maintains an independent variable $target$. Whenever a processor runs out of work, it reads the value of this $target$ and sends a work request to that particular processor. The value of the $target$ is incremented (modulo P) each time its value is read and a work request sent out. Initially, for each processor, the value of $target$ is set to $((p+1) modulo P)$ where p is its processor identification number. Note that work requests can be generated by each processor independent of the other processors; hence the scheme is simple, and has no bottlenecks. However it is possible that work requests made by different processors around the same time may be sent to the same processor. This is not desirable since ideally work requests should be spread uniformly over all processors that have work.

4.2 Nearest Neighbor (NN)

In this scheme a processor, on running out of work, sends a work request to its immediate neighbors in a round robin fashion. For example, on a hypercube, a processor will send requests only to its $\log P$ neighbors. For networks in which distance between all pairs of processors is the same, this scheme is identical to the Asynchronous Round Robin scheme. This scheme ensures locality of communication for both work requests and actual work transfers. A potential drawback of the scheme is that localized concentration of work takes a longer time to get distributed to other far away processors.

4.3 Global Round Robin (GRR)

Let a global variable called TARGET be stored in processor 0. Whenever a processor needs work, it requests and gets the value of TARGET; and processor 0 increments the value of this variable by 1 (modulo P) before responding to another request. The processor needing work now sends a request to the processor whose number is given by the value read from TARGET (i.e., the one supplied by processor 0). This ensures that successive work requests originating in the system are distributed evenly over all processors. A potential drawback of this scheme is that there will be contention for accessing the value of TARGET.

4.4 GRR with Message combining. (GRR-M)

This scheme is a modified version of GRR that avoids contention over accessing TARGET. In this scheme, all the requests to read the value of TARGET at processor 0 are combined at intermediate processors. Thus the total number of requests that have to be handled by processor 0 is greatly reduced. This technique of performing atomic increment operations on a shared variable, TARGET, is essentially a software implementation of the fetch-and-add operation of [6]. To the best of our knowledge, GRR-M has not been used for load balancing by any other researcher.

We illustrate this scheme by describing its implementation for a hypercube architecture. Figure 4.5 describes the operation of GRR-M for a hypercube multicomputer with $P = 8$. Here, we embed a spanning tree on the hypercube rooted at processor zero. This can be obtained by complimenting the bits of processor identification numbers in a particular sequence. We shall use the low to high sequence. Every processor is at a leaf of the spanning tree. When a processor wants to atomically read and increment TARGET, it sends a request up the spanning tree towards processor zero. An internal node of the spanning tree holds a request from one of its children for at most δ time before it is sent to its parent. If a request comes from the other child within this time, it is combined and sent up as a single request. If i represents the number of increments combined, the resulting increment on TARGET is i and returned value for read is previous value of target. This is illustrated by the figure where the total requested increment is 5 and the original value of TARGET, *i.e.* x, percolates down the tree. The individual messages combined are stored in a table until the request is granted. When read value of TARGET is sent back to an internal node, two values are sent down to the left and right children if the value corresponds to a combined message. The two values can be determined from the entries in the table corresponding to increments by the two sons.

A similar implementation can be formulated for other architectures.

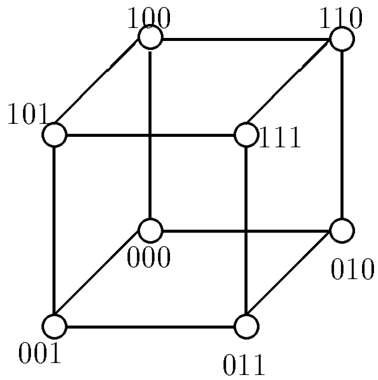
4.5 Random Polling (RP).

This is the simplest load balancing strategy where a processor requests a randomly selected processor for work each time it runs out of work. The probability of selection of any processor is the same.

4.6 Scheduler Based Load Balancing (SB)

This scheme was proposed in [30] in the context of depth first search for test generation in VLSI CAD applications. In this scheme, a processor is designated as a scheduler. The scheduler maintains a queue (*i.e.* a FIFO list) called DONOR, of all possible processors which can donate work. Initially DONOR contains just one processor which has all the work. Whenever a processor goes idle, it sends a request to the scheduler. The scheduler then deletes this processor from the DONOR queue, and polls the processors in DONOR in a round robin fashion until it gets work from one of the processors. At this point, the recipient processor is placed at the tail of the queue and the work received by the scheduler is forwarded to the requesting processor.

Like GRR, for this scheme too, successive work requests are sent to different processors. However, unlike GRR, a work request is never sent to any processor which is known to have no work. The performance of this scheme can be degraded significantly by the fact that all messages (including messages containing actual work) are routed through the scheduler. This poses an additional bottleneck for work transfer. Hence we modified this scheme slightly so that the poll was still generated by the scheduler but the work was transferred directly to the requesting processor instead of being routed through the scheduler. If the polled processor happens to be idle, it returns a 'reject' message to the scheduler indicating that it has no work to share. On receiving a 'reject', the scheduler gets the next processor from the DONOR queue and generates another poll. It should



Initially TARGET = x.
After increment operation, TARGET = x+5.

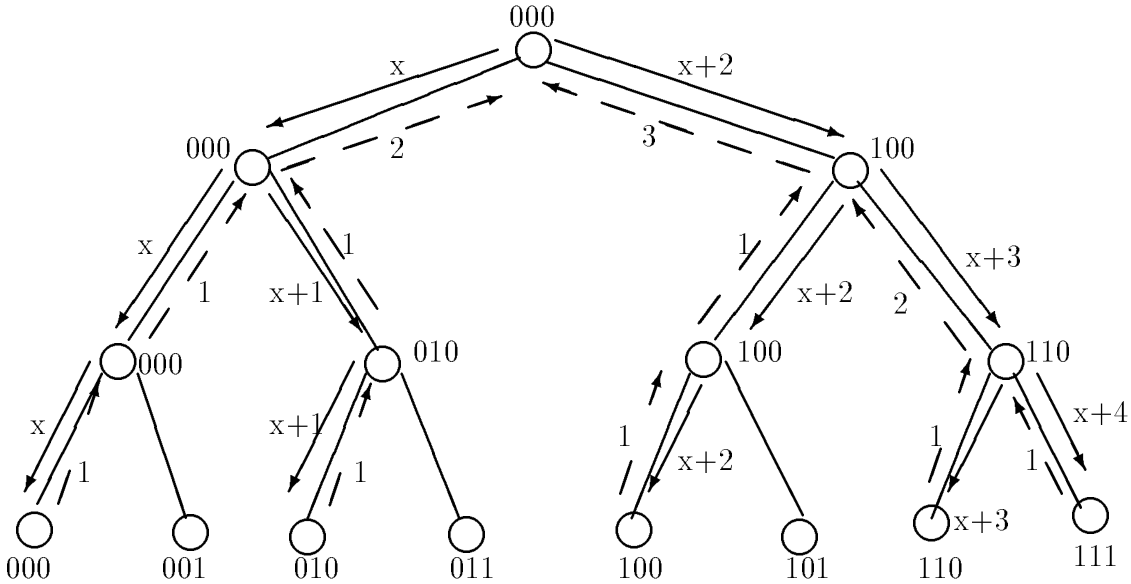


Figure 2: Illustration of message combining on an 8 processor hypercube.

be clear to the reader that the modified scheme has a strictly better performance compared to the original scheme. This was also experimentally verified by us.

5 Scalability Analysis of Receiver Initiated Load Balancing Algorithms

To analyze the scalability of a load balancing scheme, we need to compute T_o which is the extra work done by the parallel formulation. Extra work done in any load balancing is due to communication overheads (*i.e.* time spent in requesting for work and sending work), idle time (when processor is waiting for work), time spent in termination detection, contention over shared resources etc.

For the receiver initiated schemes, the idle time is subsumed by the communication overheads due to work requests and transfers. This follows from the fact that as soon as a processor becomes idle, it selects a target for a work request and issues a request. The total time that it remains idle is the time taken for this message to reach the target and for the reply to arrive. At this point, either the processor becomes busy or it generates another work request. Thus the communication overhead gives the order of time that a processor is idle.

The framework for setting upper bounds on overheads due to the communication overhead is presented in Section 5.1. For isoefficiency terms due to termination, we can show the following results. In the case of a network of workstations, we can implement a ring termination scheme, which can be implemented in $\Theta(P)$ time. This contributes an isoefficiency term of $\Theta(P^2)$. For a hypercube, we can embed a tree into the hypercube and termination tokens propagate up the tree. It can be shown that this contributes an isoefficiency term of $\Theta(P \log P)$. In the case of a mesh, we can propagate termination tokens along rows first and then along columns. This technique contributes an isoefficiency term of $\Theta(P^{1.5})$. In each of these cases, the isoefficiency term is of the same asymptotic order as the lower bound derived in Section 3.1.

5.1 A General Framework for Computing Upper Bounds on Communication

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 [8, 23].

We have seen that communication overheads are caused by work requests and work transfers. The total number of work transfers can only be less than the total number of work requests; hence the total number of work requests weighted with the total cost of one work request and corresponding work transfer defines an upper bound on the total communication overhead. For simplicity, in the rest of this section, we assume constant message sizes; hence the cost associated with all the work requests generated defines an upper bound on the total communication overheads.

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. As stated in the introduction, the partitioning strategy can be assumed to satisfy the following property -

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

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. At any given time, there is some processor that has the maximum workload of all processors. Whenever a processor receives a request for work, its local workload becomes smaller (unless it is already less than ϵ). If every processor is requested for work at least once, then we can be sure that the maximum work at any of the processors has come down at least by a factor of $(1-\alpha)$. Let us assume that in every V(P) requests made for work, 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. 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 $(1-\alpha)^2W$

$\vdots$

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 requests $\leq V(P)\log_{\frac{1}{1-\alpha}} W$

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

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

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

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

Solving this for isoefficiency term due to communication gives us the relation

$$W \sim \frac{U_{comm}}{U_{calc}} V(P)\log W$$

$$W = O(V(P)\log W) \tag{1}$$

The isoefficiency defined by this term is the overall isoefficiency of the parallel system if it asymptotically dominates all other terms (due to reasons other than the communication overhead).

5.2 Computation of V(P) for various load balancing schemes

Clearly, it can be seen that $V(P)$ is $\Theta(P)$ for GRR and GRR-M, and as shown in [8, 23] it is $O(P^2)$ for Asynchronous Round Robin techniques. The worst case for Asynchronous Round Robin techniques is realized when all processors generate successive work requests at almost identical instances of time and the initial value of counters at all processors is almost identical. Consider the following (worst case) scenario. Assume that processor $P-1$ had all the work and the local counters of all the other processors (0 to $P-2$) were pointing to processor 0. In such a case, for processor $P-1$ to receive a work request, one processor may have to request up to $P-1$

processors and the remaining $P-2$ processors in the meantime might have generated up to $P-2$ work requests (all processors except processor $P-1$ and itself). Thus $V(P)$ in this case is given by $(P-1)+(P-2)(P-2)$, i.e. $O(P^2)$. The readers must note that this is only an upper bound on the value of $V(P)$, and the actual value of $V(P)$ is between P and P^2. For NN, $V(P)$ is unbounded for architectures such as the hypercube. That is why isoefficiency functions for NN are determined in [23] using a different technique.

Random Polling

In the worst case, the value of $V(P)$ is unbounded for random polling. Here we present an average case analysis for computing the value of $V(P)$.

Consider a system of P boxes. In each trial a box is picked up at random and marked off. We are interested in mean number of trials required to mark all the boxes. In our algorithm, each trial consists of a processor requesting another processor selected at random for work.

Let $F(i,P)$ represent a state of the system where i of the P boxes have been marked and $i - P$ have not been marked. Since the next box to be marked is picked at random, there is $\frac{i}{P}$ chance that it will be a marked box and $\frac{P-i}{P}$ chance that it will be an unmarked box. Hence the system remains in $F(i,P)$ with a probability of $\frac{i}{P}$ and transits to $F(i+1,P)$ with a probability of $\frac{P-i}{P}$. Let $f(i,P)$ denote the average number of trials needed to change from state $F(i,P)$ to $F(P,P)$. Obviously, $V(P) = f(0,P)$.

We have

$$f(i,P) = \frac{i}{P}(1+f(i,P)) \; + \; \frac{P-i}{P}(1+f(i+1,P))$$

$$\frac{P-i}{P}f(i,P) = 1 + \frac{P-i}{P}f(i+1,P)$$

$$f(i,P) = \frac{P}{P-i} + f(i+1,P)$$

Hence,

$$f(0,P) = P \star \Sigma_{i=0}^{i=P-1}\frac{1}{P-i}$$

$$= P \star \Sigma_{i=1}^{i=P}\frac{1}{i}$$

$$= P \star H_P$$

where H_P denotes the harmonic mean of first P natural numbers. Since we have $H_P \simeq 1.69 ln(P)$ (where $ln(P)$ denotes natural logarithm *i.e.* base e of P), we have $V(P) = \Theta(P \log P)$. This value of V(P) will be used to compute the isoefficiency of RP on different architectures.

Scheduler Based Load Balancing

For this scheme, it can be seen that the length of the potential work donors can never exceed the total number of processors. At any given time, the processors that are not on the DONOR queue are known to have no work. Hence, the total maximum work at any processor in the system can be reduced by a factor of $(1-\alpha)$ by issuing each of these processors a work request. Since a processor, on getting work is added to the end of the queue, and since the processors already in the DONOR queue are polled in a round robin fashion, it can be seen that no processor will be requested for work twice in a time interval when another busy processor was not requested even once. Since at any given point of time, this queue can not contain more than P processor (since duplication of a processor is not possible) P forms an upper bound on the number of requests that need to be made for each processor with work to be requested at least once. Thus $V(P)$ is $O(P)$.

5.3 Analysis of Receiver Initiated Load Balancing Algorithms for Hypercube

Here we present analyses for all the schemes introduced in section 4 with the exception of NN. Scalability analysis for NN was presented in [23]. Some of the techniques analyzed here have better scalability than the NN, and have near optimal isoefficiency functions.

In this section, we assume that work can be transferred through fixed size messages. The effect of relaxing this assumption is presented in Section 7.

For a hypercube, the average distance between any randomly chosen pair of processors is $\Theta(\log P)$. On actual machines however, this asymptotic relation might not accurately describe communication characteristics for the practical range of processors, and a more precise analysis may be required. Section 5.3.6 presents such an analysis and investigates the impact of actual communication parameters for the Ncube/2^{TM} on the overall scalability of these schemes.

In this section, we perform analysis with the asymptotic value of $U_{comm} = \Theta(\log P)$.

Equation 1 thus reduces to

$$W = O(V(P) \log P \log W)$$

Substituting W into the the right hand side of the same equation, we get

$$W = O(V(P) \log P \log(V(P) \log P \log W))$$

$$W = O(V(P) \log P \log(V(P)) + V(P) \log P \log\log P + V(P) \log P \log\log W)$$

Ignoring the double log terms, we get,

$$W = O(V(P) \log P \log(V(P))) \tag{2}$$

5.3.1 Asynchronous Round Robin

This is a completely distributed algorithm, there is no contention for shared resources and the isoefficiency is determined only by the communication overhead. As discussed in section 5.2, $V(P)$ in the worst case is $O(P^2)$. Substituting this into Equation 2, this scheme has an isoefficiency function of $O(P^2 \log^2 P)$.

5.3.2 Global Round Robin

From Section 5.2, $V(P) = \Theta(P)$ for this scheme. Substituting into Equation 2, this scheme has an isoefficiency function of $O(P \log^2 P)$ because of communication overheads.

In this scheme, a global variable is accessed repeatedly. This can cause contention. Since the number of times this variable is accessed is equal to the total number of work requests, it is given by $O(V(P) \log W)$ (read and increment operations) over the entire execution. If processors are efficiently utilized, then the total time of execution is $\Theta(W/P)$. Assume that while solving some specific problem instance of size W on P processors, there is no contention. Clearly, in this case, W/P is much more than the total time over which the shared variable is accessed (which is given by $O(V(P)\log W)$). Now, as we increase the number of processors, the total time of execution (*i.e.* W/P) decreases but the number of times the shared variable is accessed increases. There is thus a crossover point beyond which the shared variable access will become a bottleneck and the overall runtime cannot be reduced further. This bottleneck can be eliminated by increasing W at a rate such that the ratio between W/P and $O(V(P)\log W)$ remains the same.

This gives the following isoefficiency term,

$$\frac{W}{P} \sim V(P)\log W$$

or

$$\frac{W}{P} \sim P\log W$$

or

$$W \sim O(P^2 \log P)$$

Thus since the isoefficiency due to contention asymptotically dominates the isoefficiency due to communication overhead, the overall isoefficiency is given by $O(P^2 \log P)$.

5.3.3 GRR with Message Combining

For this scheme, it was shown in Section 5.2 that each increment on TARGET takes $\Theta(\delta \log P)$ communication time and so, communication time for each request is $\Theta(\delta \log P)$. Also, $V(P) = \Theta(P)$, hence from Equation 2, this strategy has an isoefficiency of $O(\delta P \log^2 P)$. Increasing δ results in better message combining, but leads to larger overall latency and higher overhead in processing requests in the spanning tree. Smaller values of δ result in lower degree of message combining; consequently contention for access to TARGET will begin to dominate and in the limiting case, its isoefficiency will be the same as that for GRR. The value of δ thus has to be chosen to balance all these factors.

5.3.4 Random Polling

In Section 5.2, we had shown that for the case where an idle processor requests a random processor for work, $V(P) = \Theta(P\log P)$. Substituting values of U_{comm} and $V(P)$ into Equation 1, this scheme has an isoefficiency function of $O(P\log^3 P)$.

5.3.5 Scheduler Based Load Balancing

From Section 5.2, $V(P)$ for this scheme is $O(P)$. Plugging this value into Equation 2, we can see that isoefficiency due to communication overhead is $O(P\log^2 P)$. Also, it can be shown from an analysis similar to that for GRR that the isoefficiency due to contention is given by $O(P^2 \log P)$. The overall isoefficiency of the scheme is thus given by $O(P^2 \log P)$.

5.3.6 Effect of Specific Machine Characteristics on Isoefficiency.

Asymptotic communication costs can differ from actual communication costs in available range of processors. In such cases, it is necessary to consider a more precise expression for the communication cost.

We illustrate this for a 1024 node Ncube/2^{TM}, which we also use in our experimental validation. The Ncube/2^{TM} is a hypercube multicomputer having the cut-through message routing feature. For such a machine, the time taken to communicate a message of size m words over the communication network between a pair of processors which are d hops apart is given by $t_s + m \times t_w + d \times t_h$. Here t_s, t_w and t_h represent the message startup time, per-word transfer time, and the per-hop time respectively. For the simplified case, since we assume fixed size messages, the communication time can be written as $k + d \times t_h$, where k is a constant. For the hypercube architecture, the maximum distance between any pair of processors is $\log P$. Hence, a more precise expression for U_{comm} is given by $k + t_h \times \log P$.

For the 1024 processor Ncube/2^{TM}, t_s, t_w and t_h are approximately 100 μs,2 μs per 4 byte word and 2 μs respectively. In the experiments reported in Section 8, the size of a message carrying work is approximately 500 bytes (this figure is actually a function of the amount of work transferred). Thus, $t_s + m \times t_w \sim 350\mu$s and for communication between farthest pair of nodes, $d = \log P = 10$ (for $P = 1024$) and $d \times t_h \sim 20\mu$s. Clearly $t_s + m \times t_w$ dominates $d \times t_h$ even for communication between farthest processors. Consequently, the message transfer time is approximately given by $t_s + m \times t_w$ which is a constant if we assume a constant message size m.

Hence for practical configurations of Ncube/2^{TM}, U_{comm} is $\Theta(1)$ instead of $\Theta(\log P)$ for ARR, GRR, RP and SB. Since for both RP and ARR, the dominant isoefficiency term is due to communication overheads, reducing U_{comm} to $\Theta(1)$ has the effect of decreasing the isoefficiencies of both these schemes by a log factor. For such machines, the isoefficiencies of RP and ARR are thus given by $O(P \log^2 P)$ and $O(P^2 \log P)$, respectively. However, since the dominant isoefficiency term in both GRR and SB is due to contention, which is not affected by this machine characteristic, the isoefficiency of both of these schemes remains $O(P^2 \log P)$. U_{comm} is still $\Theta(\log P)$ for GRR-M. The reason is that in the absence of message combining hardware, we have to stop the request for reading the value of TARGET at each intermediate step and allow for latency for message combining. Consequently, the overall isoefficiency of this scheme remains unchanged at $O(P \log^2 P)$.

We can thus see that machine specific characteristics even for a given architecture have a significant effect on overall scalability of different schemes.

5.4 Analysis of Receiver Initiated Load Balancing Algorithms for a network of workstations

In this section, we analyze the scalability of these schemes on a network of workstations. The network of workstations under consideration for analysis are assumed to be connected on a standard Carrier Sense Multiple Access (CSMA) Ethernet. Here, the time to deliver a message of fixed size between any pair of processors is the same. The total bandwidth of the Ethernet is limited and so this imposes an upper limit on the number of messages that can be handled in a given period of time. As the number of processors increases, the total traffic on the network also increases causing contention over the Ethernet.

For this architecture, U_{comm} is given by $\Theta(1)$. Substituting into Equation 1 , we get,

$$W \sim O(V(P)\log W)$$

Simplifying this as in Section 5.3, we get

$$W \sim O(V(P)\log(V(P))) \tag{3}$$

5.4.1 Asynchronous Round Robin

From Section 5.2, $V(P)$ is $O(P^2)$. Substituting into Equation 3, this scheme has an isoefficiency of $O(P^2 \log P)$ due to communication overheads.

For isoefficiency term due to bus contention, we use an analysis similar to one used for analyzing contention in GRR for the hypercube. The total number of messages on the network over the entire execution is given by $O(V(P)\log W)$. If processors are efficiently utilized, the total time of execution is $\Theta(W/P)$. By an argument similar to one used for contention analysis for GRR on a hypercube, W must grow at least at a rate such that the ratio of these two terms (*i.e.* $O(V(P)\log W)$ and $\Theta(W/P)$) remains the same.

Thus for isoefficiency, we have,

$$\frac{W}{P} \sim V(P)\log W$$

or

$$\frac{W}{P} \sim P^2 \log W$$

Solving this equation for isoefficiency, we get

$$W \sim O(P^3 \log P)$$

Since this is the dominant of the two terms, it also defines the isoefficiency function for this scheme.

5.4.2 Global Round Robin

For this case, from Section 5.2, $V(P)$ is $\Theta(P)$. Substituting into Equation 3, this scheme has an isoefficiency of $O(P \log P)$ due to communication overheads. We now consider the isoefficiency due to contention at processor 0. We had seen that processor 0 has to handle $V(P)\log W$ requests in $\Theta(W/P)$ time. Equating the amount of work with the number of messages, we get an isoefficiency term of $W \sim O(P^2 \log P)$. By a similar analysis, it can be shown that the isoefficiency due to bus contention is also given by $W \sim O(P^2 \log P)$. Thus the overall isoefficiency of this scheme is given by $O(P^2 \log P)$.

5.4.3 Random Polling

Here, from Section 5.2, $V(P) = \Theta(P \log P)$. Substituting this value into Equation 3, this scheme has an isoefficiency of $O(P \log^2 P)$ due to communication overheads.

For isoefficiency term due to bus contention, as before, we equate the total number of messages that have to be sent on the bus against the time. We had seen that the total number of messages is given by $O(V(P)\log W)$ and the time (assuming efficient usage) $\Theta(W/P)$. Thus for isoefficiency,

$$\frac{W}{P} \sim V(P)\log W$$

or

$$\frac{W}{P} \sim P\log P\log W$$

Solving this equation for isoefficiency, we get

$$W \sim O(P^2\log^2 P)$$

Thus, since the isoefficiency due to bus contention asymptotically dominates the isoefficiency due to communication overhead, the overall isoefficiency is given by $O(P^2\log^2 P)$.

Effect of number of messages on U_{comm}.

The readers may contend that the assumption of U_{comm} being a constant is valid only when there are few messages in the system and network traffic is limited. In general, the time for communication of a message would depend on the amount of traffic in the network. It is true that as the number of messages generated over a network in a fixed period of time increases, the throughput decreases (and consequently U_{comm} increases). However, in the analysis, we keep the number of messages generated in the system in unit time to be a constant. We derive isoefficiency due to contention on the network by equating W/P (which is the effective time of computation) with the total number of messages ($V(P)\log W$). By doing this, we essentially force the number of messages generated per unit time to be a constant. In such a case, the message transfer time (U_{comm}) can indeed be assumed to be a constant. Higher efficiencies are obtained for sufficiently large problem sizes. For such problems, the number of messages generated per unit time of computation is lower. Consequently U_{comm} also decreases. In particular, for high enough efficiencies, U_{comm} will be close to the optimal limit imposed by the network bandwidth.

Table 1 shows the isoefficiency functions for different receiver initiated schemes for various architectures. The results in boldface were derived in [23]. Others were either derived in Section 5, Appendix A, or can be derived by a similar analysis. Table 2 presents a summary of the various overheads for load balancing over a network of workstations and their corresponding isoefficiency terms.

6 Sender Initiated Load Balancing Algorithms

In this section, we discuss load balancing schemes in which work splitting is sender initiated. In these schemes, the generation of subtasks is independent of the work requests from idle processors. These subtasks are delivered to processors needing them, either on demand (*i.e.*, when they are idle) [10] or without demand [33, 36, 35].

Scheme→ Arch↓	ARR	NN	GRR	GRR-M	RP	Lower Bound
SM	$\mathbf{O(P^2 \log P)}$	$\mathbf{O(P^2 \log P)}$	$\mathbf{O(P^2 \log P)}$	$\mathbf{O(P \log P)}$	$O(P \log^2 P)$	$O(P)$
Cube	$O(P^2 \log^2 P)$	$\mathbf{\Omega(P^{\log_2 \frac{1+\frac{1}{\alpha}}{2}})}$	$O(P^2 \log P)$	$O(P \log^2 P)$	$O(P \log^3 P)$	$O(P \log P)$
Ring	$\mathbf{O(P^3 \log P)}$	$\mathbf{\Omega(K^P)}$	$\mathbf{O(P^2 \log P)}$		$O(P^2 \log^2 P)$	$O(P^2)$
Mesh	$O(P^{2.5} \log P)$	$\Omega(K^{\sqrt{P}})$	$O(P^2 \log P)$	$O(P^{1.5} \log P)$	$O(P^{1.5} \log^2 P)$	$O(P^{1.5})$
WS	$O(P^3 \log P)$	$O(P^3 \log P)$	$O(P^2 \log P)$		$O(P^2 \log^2 P)$	$O(P^2)$

Table 1: Scalability results of receiver initiated load balancing schemes for shared memory (SM), cube, ring, mesh and a network of workstations (WS).

Arch.	Overheads→ Scheme↓	Communication	Contention (shared data)	Contention (bus)	Isoefficiency
WS	ARR	$O(P^2)$		$O(P^3 \log P)$	$O(P^3 \log P)$
	GRR	$O(P)$	$O(P^2 \log P)$	$O(P^2 \log P)$	$O(P^2 \log P)$
	RP	$O(P \log^2 P)$		$O(P^2 \log^2 P)$	$O(P^2 \log^2 P)$
H-cube	ARR	$O(P^2 \log^2 P)$			$O(P^2 \log^2 P)$
	GRR	$O(P \log^2 P)$	$O(P^2 \log P)$		$O(P^2 \log P)$
	RP	$O(P \log^3 P)$			$O(P \log^3 P)$

Table 2: Various overheads for load balancing over a network of workstations (WS) and hypercubes (H-cube) and their corresponding isoefficiency terms.

6.1 Single Level Load Balancing (SL)

This scheme balances the load among processors by dividing the task into a large number of subtasks such that each processor is responsible for processing more than one subtask. In doing so it statistically assures that the total amount of work at each processor is roughly the same. The task of subtask generation is handled by a designated processor called MANAGER. The MANAGER generates a specific number of subtasks and gives them one by one to the requesting processors on demand. Since the MANAGER has to generate subtasks fast enough to keep all the other processors busy, subtask generation forms a bottleneck and consequently this scheme does not have a good scalability.

The scalability of this scheme can be analyzed as follows. Assume that we need to generate k subtasks, and the generation of each of these subtasks takes time ν. Also, let the average subtask size be given by z. Thus, $z = W/k$. Clearly $k = \Omega(P)$ since the number of subtasks has to be at least of the order of the number of processors. It can also be seen that z is given by $\Omega(P)$. This follows from the fact that in a P processor system, on the average, P work request messages will arrive in $\Theta(z)$ time. Hence, to avoid subtask generation from being a bottleneck, we have to generate at least P subtasks in time $\Theta(z)$. Since the generation of each of these takes ν time, z has to grow at a rate higher than $\Theta(\nu P)$. Now, since $W = k \times z$, substituting lower bounds for k and z, we get the isoefficiency function to be $W = \Omega(P^2)$. Furuichi et. al. [10] present a similar analysis to predict speedup and efficiency. This analysis does not consider the idle time incurred by processors between making a work request and receiving work. Even though this time is different for different architectures such as the mesh, cube etc., it can be shown that the overall scalability is still $\Omega(P^2)$ for these architectures.

In general, subtask sizes (z) can be of widely differing sizes. Kimura and Ichiyoshi present a detailed analysis in [20] for the case in which subtasks can be of random sizes. They show that in this case, the isoefficiency of SL is given by $\Theta(P^2 \log P)$.

6.2 Multi Level Load Balancing (ML)

This scheme tries to circumvent the subtask generation bottleneck [10] of SL through multiple level subtask generation. In this scheme, all processors are arranged in the form of an m-ary tree of depth l. The task of super-subtask generation is given to the root processor. It divides the task into super-subtasks and distributes them to its successor processors on demand. These processors subdivide the super-subtasks into subtasks and distribute them to successor processors on request. The leaf processors repeatedly request work from their parents as soon as they finish previously received work. A leaf processor is allocated to another subtask generator when its designated subtask generator runs out of work. For $l = 1$, ML and SL become identical.

For ML utilizing an l level distribution scheme, it has been shown in [20] that the isoefficiency is given by $\Theta(P^{\frac{l+1}{l}}(\log P)^{\frac{l+1}{2}})$. These isoefficiency functions were derived by assuming that the cost of work transfers between any pair of processors is $\Theta(1)$. The overall efficiency and hence the isoefficiency of these schemes will be impacted adversely if the communication cost depends on the distance between communicating processors. As discussed in Section 5.3.6, for the Ncube/2^{TM}, the assumption of a constant communication cost ($\Theta(1)$) is reasonable, and hence these scalability relations hold for practical configurations of this machine.

For a two level scheme, the isoefficiency is therefore $\Theta(P^{\frac{3}{2}}(\log P)^{\frac{3}{2}})$ and for a three level distribution scheme, it is given by $\Theta(P^{\frac{4}{3}}(\log P)^2)$. Scalability analysis of these schemes indicates that isoefficiency of these schemes can be improved to a certain extent by going to higher numbers of levels but the improvement is marginal and takes effect only at very large number of processors. For instance, $P^{\frac{4}{3}}(\log P)^2 > P^{\frac{3}{2}}(\log P)^{\frac{3}{2}}$ only for $P > 1024$.

6.3 Randomized Allocation

A number of techniques using randomized allocation have been presented in the context of parallel depth first search of state space trees [33, 36, 35]. In depth first search of trees, the expansion of a node corresponds to performing a certain amount of useful computation and generation of successor nodes, which can be treated as subtasks.

In the Randomized Allocation Strategy proposed by Shu and Kale [36], every time a node is expanded, all of the newly generated successor nodes are assigned to randomly chosen processors. The random allocation of subtasks ensures a degree of load balance over the processors. There are however some practical difficulties with the implementation of this scheme. Since for each node expansion, there is a communication step, the efficiency is limited by the ratio of time for a single node expansion to the time for a single node expansion and communication to a randomly chosen processor. Hence applicability of this scheme is limited to problems for which the total computation associated with each node is much larger than the communication cost associated with transferring it. This scheme also requires a linear increase in the cross section communication bandwidth with respect to P; hence it is not practical for large number of processors on any practical architecture (*eg*. cube, mesh, networks of workstations). For practical problems, in depth first search, it is much cheaper to incrementally build the state associated with each node rather than copy and/or create the new node from scratch [39, 4]. This also introduces additional inefficiency. Further, the memory requirement at a processor is potentially unbounded, as a processor may be required to store an arbitrarily large number of work pieces during execution. In contrast, for all other load balancing schemes discussed up to this point, the memory requirement of parallel depth first search remains similar to that of serial depth first search.

Ranade [33] presents a variant of the above scheme for execution on butterfly networks or hypercubes. This scheme uses a dynamic algorithm to embed nodes of a binary search tree into a butterfly network. The algorithm works by partitioning the work at each level into two parts and sending them over to the two sons (processors) in the network. Any patterns in work splitting and distributions are randomized by introducing dummy successor nodes. This serves to ensure a degree of load balance between processors. The author shows that the time taken for parallel tree search of a tree with M nodes on P processors is given by $O(M/P + \log P)$ with a high degree of probability. This corresponds to an optimal isoefficiency of $O(P \log P)$ for a hypercube. This scheme has a number of advantages over Shu's scheme for hypercube architectures. By localizing all communications, the communication overhead is reduced by a log factor here. (Note that communicating a fixed length message between a randomly chosen pair of processors in a hypercube takes $O(\log P)$ time.) Hence, Ranade's scheme is physically realizable on arbitrarily large hypercube architectures. To maintain the depth-first nature of search, nodes are assigned priorities and are maintained in local heaps at each processor. This adds an additional overhead of managing heaps, but may help in reducing the overall memory requirement. Apart from these, all the other restrictions on applicability of this scheme are the same as those for Shu and Kale's [36] scheme.

The problem of performing a communication for each node expansion can be alleviated by enforcing a granularity control over the partitioning and transferring process [29, 36]. It is , however, not clear whether mechanisms for effective granularity control can be derived for highly irregular state space trees. One possible method [29] of granularity control works by not giving away nodes below a certain "cutoff" depth. Search below this depth is done sequentially. This clearly reduces the number of communications. The major problem with this mechanism of granularity control is that subtrees below the cutoff depth can be of widely differing sizes. If the cutoff depth is too deep, then it may not result in larger average grain size and if it is too shallow, subtrees to be searched sequentially may be too large and of widely differing sizes.

7 Effect of Variable Work Transfer Cost on Scalability

In the analysis presented in previous sections, we have assumed that the cost of transferring work is independent of the amount of work transferred. However, there are problems for which the work transfer cost is a function of the amount of work transferred. Instances of such problems are found in tree search applications for domains where strong heuristics are available [38]. For such applications, the search space is polynomial in nature and the size of the stack used to transfer work varies significantly with the amount of work transferred. In this section, we demonstrate the effect of variable work transfer costs for the case where the cost of transferring w units of work varies as $\sqrt{w}$ for the GRR load balancing scheme. We present analysis for the hypercube and network of workstations. Analysis for other architectures and load balancing schemes can be carried out similarly.

We perform the analysis in the same framework as presented in Section 5. The upper bound on the total number of work requests is computed. Since the total number of actual work transfers can only be less than the total number of requests, the upper bound also applies to the number of work transfers. Considered in conjunction with the communication cost associated with each work piece, this specifies an upper bound on the total communication overhead. Note that by using the upper bound on the number of requests to specify an upper bound on the number of work transfers, we are actually setting a loose upper bound on the total communication overhead. This is because of the fact that the total number of work transfers may actually be much less than the number of requests.

7.1 Hypercube Architecture

From our analysis in Section 5.1, we know that after the i^{th} round of $V(P)$ requests, the maximum work available at any processor is less than $(1-\alpha)^i W$. Hence, if the size of a work message varied as $\sqrt{w}$ where w was the amount of work transferred, then $w = O((1-\alpha)^i W)$ and U_{comm} at the i^{th} step is given by $O(\log P \sqrt{w})$, *i.e.* $O(\log P \sqrt{(1-\alpha)^i W})$. Since $O(\log W)$ such iterations are required, the total communication cost is given by:

$$T_o = \Sigma_{i=1}^{\log W} U_{comm} * V(P)$$

$$T_o = \Sigma_{i=1}^{\log W} \sqrt{(1-\alpha)^i W} * \log P * P$$

$$T_o = (P \log P)\Sigma_{i=1}^{\log W} \sqrt{(1-\alpha)^i W}$$

$$T_o = (P \log P)\sqrt{W}\Sigma_{i=1}^{\log W} \sqrt{(1-\alpha)^i}$$

Let $\sqrt{(1-\alpha)} = a$. Clearly $a \leq 1$.
We have,

$$T_o = (P \log P)\sqrt{W}\Sigma_{i=1}^{\log W} a^i$$

$$T_o = (P \log P)\sqrt{W}\frac{a - a^{\log W + 1}}{1 - a}$$

Since $a^{\log W+1}$ approaches 0 for large W, $\frac{a-a^{\log W+1}}{1-a}$ approaches a constant value.
Thus,

$$T_o = O((P \log P)\sqrt{W})$$

So, the isoefficiency due to communication is given by equating the total communication overhead with the total amount or work. Thus,

$$W = O(P \log P \sqrt{W})$$

or

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

The isoefficiency term due to contention is still the same, *i.e.* $O(P^2 \log P)$, and so the overall isoefficiency of the scheme in this case is now given by $W \sim O(P^2 \log^2 P)$. We can see that in this case, the overall isoefficiency function has increased due to the dependence of cost of work transferred on the amount of work transferred thus resulting in poorer scalabilities.

7.2 Network of Workstations

For a network of workstations, we know that the message transfer time for a fixed length message is $\Theta(1)$. Thus for communicating a message of size $\Theta(\sqrt{w})$, communication cost $U_{comm} = \Theta(\sqrt{w})$.
As before,

$$T_o = \Sigma_{i=1}^{\log W} U_{comm} * V(P)$$

Substituting U_{comm} and $V(P)$, we get,

$$T_o \sim \Sigma_{i=1}^{\log W} \sqrt{w} * P$$

$$T_o \sim P\Sigma_{i=1}^{\log W} \sqrt{(1-\alpha)^i W}$$

$$T_o \sim P\sqrt{W}\Sigma_{i=1}^{\log W} \sqrt{(1-\alpha)^i}$$

The summation term has been shown to approach a constant value as W increases. Therefore,

$$T_o = O(P\sqrt{W})$$

Equating T_o with W for isoefficiency, we get

$$W = O(P\sqrt{W})$$

or

$$W = O(P^2)$$

Thus isoefficiency due to communication overheads is given by $W \sim O(P^2)$. The term corresponding to accessing the shared variable TARGET remains unchanged, and is given by $O(P^2 \log P)$. For isoefficiency due to contention on shared bus, we have to balance the time for computation with the total time required to process the required number of messages on the Ethernet. Since all messages have to be sent sequentially, the total time for processing all messages is of the same order as the total communication overhead. Thus, we have, $W/P \sim O(P\sqrt{W})$ or $W \sim O(P^4)$. We can see that this term clearly dominates the communication overhead term, therefore, the overall isoefficiency for this case is given by $W \sim O(P^4)$. This should be compared with the isoefficiency value of $O(P^2 \log P)$, which we had obtained under the fixed message size assumption.

From the above sample cases, it is evident that the cost of work transfer has a significant bearing on the overall scalability of a given scheme on a given architecture. Thus, it becomes very important to analyze the application area to determine what assumptions can be made on the size of the work transfer message.

8 Experimental Results and Discussion

Here we report on the experimental evaluation of the eight schemes. All the experiments were done on a second generation Ncube^{TM} in the context of the Satisfiability problem [5]. The Satisfiability problem consists of testing the validity of boolean formulae. Such problems arise in areas such as VLSI design and theorem proving among others [2, 5]. The problem is "given a boolean formula containing binary variables in disjunctive normal form, find out if it is unsatisfiable". The Davis and Putnam algorithm [5] presents a fast and efficient way of solving this problem. The algorithm essentially works by performing a depth first search of the binary tree formed by true/false assignments to the literals. Thus the maximum depth of the tree cannot exceed the number of literals. The algorithm works as follows: select a literal, assign it a value true, remove all clauses where this literal appears in the non-negated form, remove the literal from all clauses where this appears in the negated form. This defines a single forward step. Using this step, search through all literals by assigning values true (as described) and false (invert the roles of the negated and non-negated forms) until such time as either the clause set becomes satisfiable or we have explored all possible assignments. Even if a formula is unsatisfiable, only a small subset of the 2^n combinations possible will actually be explored. For instance, for a 65 variable problem, the total number of combinations possible is 2^{65} (approximately 3.69×10^{19}) but only about 10^7 nodes are actually expanded in a typical problem instance. The search tree for this problem is highly pruned in a nonuniform fashion and any attempt to simply partition the tree statically results in extremely poor load balance.

We implemented the Davis-Putnam algorithm, and incorporated the load balancing algorithms discussed in Sections 4, and 6.1 and 6.2 into it. This program was run on several unsatisfiable instances. By choosing unsatisfiable instances, we ensured that the number of nodes expanded by the parallel formulation was exactly the same as that by the sequential one, and any speedup loss was only due to the overheads of load balancing.

In the various problem instances that the program was tested on, the total number of nodes in the tree varied between approximately 100 thousand and 10 million. The depth of the tree generated (which is equal to the number of variables in the formula) varied between 35 and 65 variables. The speedups were calculated with respect to the optimum sequential execution time for the same problems. Average speedups were calculated by taking the ratio of the cumulative time to solve all the problems in parallel using a given number of processors to the corresponding cumulative sequential time. On a given number of processors, the speedup and efficiency were largely determined by the tree size (which is roughly proportional to the sequential runtime). Thus, speedup on similar sized problems were quite similar.

All schemes were tested over a sample set of 5 problem instances in the specified size range. Tables 3 and 5 show average speedup obtained in parallel algorithm for solving the instances of the satisfiability problem using eight different load balancing techniques. Figures 8 and 8 present graphs corresponding to the speedups obtained. Graphs corresponding to NN and SB schemes have not been drawn. This is because they are nearly identical to RP and GRR schemes respectively. Table 4 presents the total number of work requests made in the case of random polling and message combining for a specific problem. Figure 8 presents the corresponding graph and compares the number of messages generated with $O(P \log^2 P)$ and $O(P \log P)$ for random polling and message combining respectively.

Our analysis had shown that GRR-M has the best isoefficiency for the hypercube architecture. However, the reader would note that even though the message combining scheme results in the smallest number of requests, the speedup obtained is similar to that for RP. This is because software message combining on Ncube/2^{TM} is quite expensive, and, this difference between the number of work requests made in load balancing schemes is not high enough. It is clear from the analysis in Section 5 and the trend in our experimental results, that this difference will widen for more processors. Hence for larger number of processors, message combining scheme would eventually outperform the other schemes. In the presence of message combining hardware, the log factor reduction in the number of messages causes a significant reduction in overheads and consequently this scheme can be expected to perform better than the others even for a moderate number of processors.

Experimental results show that NN performs slightly better than RP. Recall that the isoefficiency of NN is $\Theta(P^r)$, where r is determined by the quality of the splitting function (better splitting functions result in smaller values of r). In the context of the 15 puzzle, in [29], r was determined to be 1.57. For up to 1024 processors, NN and RP have very similar performance. However, for higher values of P, $P \log^2 P$ would become smaller than P^r and RP would outperform NN. The exact crossover point is determined by the value of r, which depends on the quality of the splitting function. Appendix B analyzes the effect of the quality of the splitting function on overall scalability of NN and RP. It is shown that the scalability of the NN scheme degrades much faster than RP as the quality of the splitting function deteriorates. Thus for domains where good splitting functions are not available, RP would be uniformly better than NN.

To clearly demonstrate the effect of quality of splitting function on the scalability of RP and NN, we test the effect of damaging the splitting function on both of these schemes. Figure 8 shows the experimental results comparing the speedup curves for RP and NN schemes with a damaged splitting function. It can be seen here that though with the original splitting function, NN marginally outperformed RP, it performs significantly worse for poorer splitting functions. This is in perfect conformity with expected theoretical results.

From our experiments, we observe that the performance of GRR and SB load balancing schemes is very similar. As shown analytically, both of these schemes have identical isoefficiency functions. Since the isoefficiency of these schemes is $O(P^2 \log P)$, performance of these schemes deteriorates very rapidly after 256 processors. Good speedups can be obtained for $P > 256$, only for very large problem instances. Neither of these schemes is thus useful for larger systems. Our experimental results also show ARR (Asynchronous Round Robin) to be more scalable than these two schemes, but significantly less scalable than RP, NN or GRR-M. The readers should note that although the isoefficiency of ARR is $O(P^2 \log^2 P)$ and that of GRR is $O(P^2 \log P)$, ARR performs better than GRR. The reason for this is that $P^2 \log^2 P$ is only an upper bound which is derived using $V(P) = O(P^2)$. This value of $V(P)$ is only a loose upper bound for ARR. In contrast, the value of $V(P)$ used for GRR ($\Theta(P)$) is a tight bound.

For the case of sender initiated load balancing, (SL and ML) the cutoff depths have been fine-tuned for optimum performance. ML is implemented for $l = 2$, as for less than 1024 processors $l = 3$ does not provide any improvements. It is clear from the speedup results presented in Table 5 that the single level load balancing scheme has a very poor scalability. Though this scheme outperforms the multilevel load balancing scheme for small number of processors, the subtask generation bottleneck sets in for larger number of processors and performance degradation of single level scheme is rapid after this point. The exact crossover point is determined by the problem size, values of parameters chosen and architecture dependent constants. Comparing the isoefficiency function for a two level load distribution, (given by $\Theta(P^{\frac{3}{2}}(\log P)^{\frac{3}{2}})$) with corresponding functions for RP and GRR-M which are given by $O(P \log^3 P)$ and $O(P \log^2 P)$, we see that, asymptotically, RP and GRR-M should both perform better than two level sender based distribution scheme. Our experimental results are thus in perfect conformity with expected analytical results.

To demonstrate the accuracy of the isoefficiency functions in Table 1, we experimentally verify the isoefficiency of the RP technique (the selection of this technique was arbitrary). As a part of this experiment, we ran 30 different problem instances varying in size from 100 thousand nodes to 10 million nodes on a range of processors. Speedups and efficiencies were computed for each of these. Data points with same efficiency for different problem sizes and number of processors were then grouped together. Where identical efficiency points were not available, the problem size was computed by averaging over points with efficiencies in the neighborhood of the required value. This data is presented in Figure 8, which plots the problem size W against $P \log^2 P$ for values of efficiency equal to 0.9, 0.85, 0.74 and 0.64. Also note the two data points for which exact values of problem sizes are not available for corresponding efficiencies. Instead we have plotted a neighboring point. We expect the points corresponding to the exact efficiency to be collinear with the others. We had seen in Section 5 that the isoefficiency of the RP scheme was $O(P \log^3 P)$. We had further seen in Section 5.3.6 that due to large message startup time for the Ncube/2^{TM}, effective isoefficiency of RP is $O(P \log^2 P)$ when P is not very large. Thus, analytically, the points corresponding to the same efficiency on the said graph must be collinear. We can see from Figure 8 that the points are indeed collinear, which clearly shows that the isoefficiency of the RP scheme

Scheme→ P↓	ARR	GRR-M	RP	NN	GRR	SB
8	7.506	7.170	7.524	7.493	7.384	7.278
16	14.936	14.356	15.000	14.945	14.734	14.518
32	29.664	28.283	29.814	29.680	29.291	28.875
64	57.721	56.310	58.857	58.535	57.729	57.025
128	103.738	107.814	114.645	114.571	110.754	109.673
256	178.92	197.011	218.255	217.127	184.828	184.969
512	259.372	361.130	397.585	397.633	155.051	162.798
1024	284.425	644.383	660.582	671.202		

Table 3: Average Speedups for Receiver Initiated Load Balancing strategies.

Scheme→ P↓	GRR-M	RP
8	260	562
16	661	2013
32	1572	5106
64	3445	15060
128	8557	46056
256	17088	136457
512	41382	382695
1024	72874	885872

Table 4: Number of requests generated for GRR-M and RP.

is indeed what was theoretically derived. This demonstrates that it is indeed possible to accurately estimate the isoefficiency of a parallel algorithm and establishes its viability as a useful tool in evaluating parallel system.

9 Summary of Results and Future Work

The scalability analysis of various load balancing schemes has provided valuable insights into the relative performance and suitability of these schemes for different architectures.

For the hypercube, our analysis shows that GRR-M, RP and NN schemes are more scalable than ARR, GRR and SB; hence, they are expected to perform better for higher number of processors. Asymptotically, GRR-M has the best isoefficiency of the three; but in the absence of message combining hardware, the high constant of proportionality can cause GRR-M to perform poorer than RP and NN for moderate number of processors. In our experiments, GRR-M performs similar to RR and NN for up to 1024 processors on the Ncube/2^{TM} which does not have message

Scheme→ P↓	RP	SL	ML
8	7.524	7.510	7.315
16	15.000	14.940	14.406
32	29.814	29.581	28.246
64	58.857	52.239	56.003
128	114.645	79.101	106.881
256	218.255		192.102
512	397.585		357.515
1024	660.582		628.801

Table 5: Speedups for Sender Initiated Load Balancing strategies in comparison to RP.

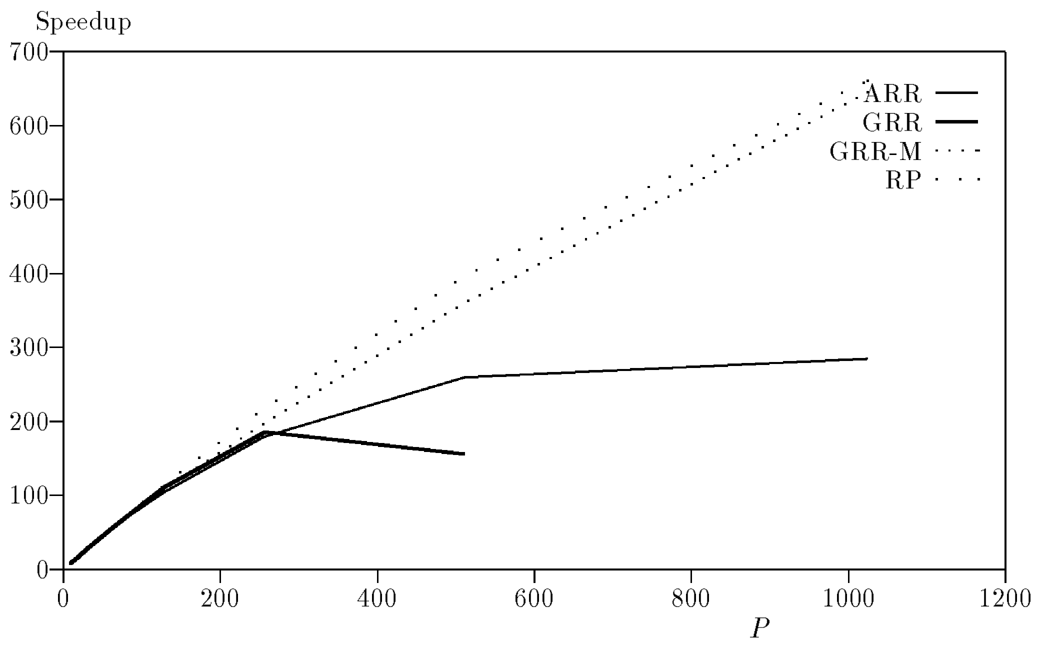


Figure 3: Average Speedups for Receiver Initiated Load Balancing Strategies.

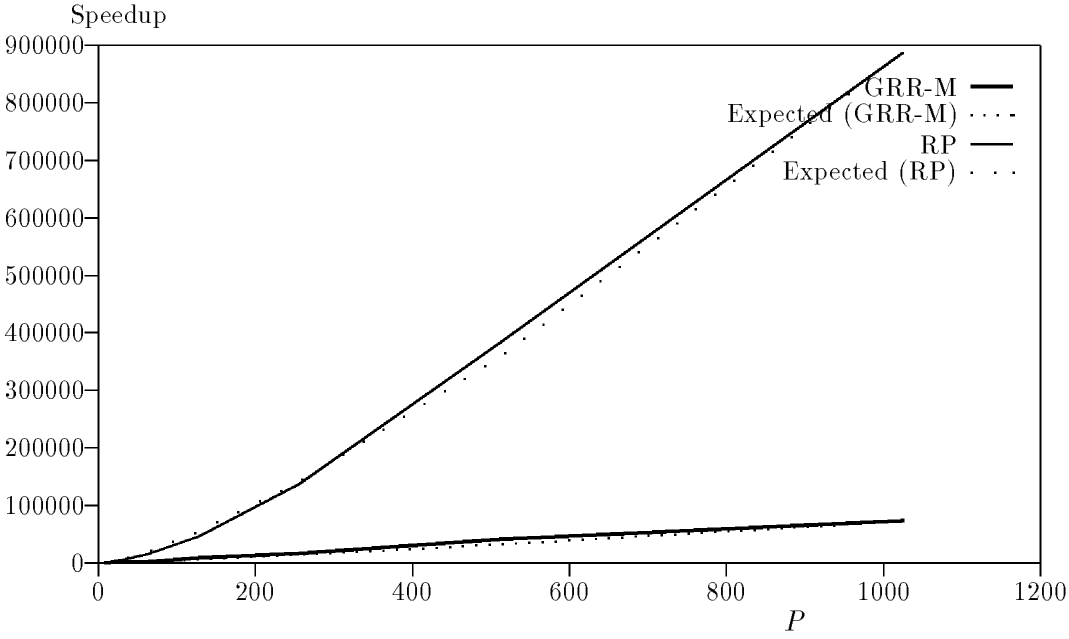


Figure 4: Number of work requests generated for RP and GRR-M and their expected values ($O(P \log P)$ and $O(P \log^2 P)$ respectively).

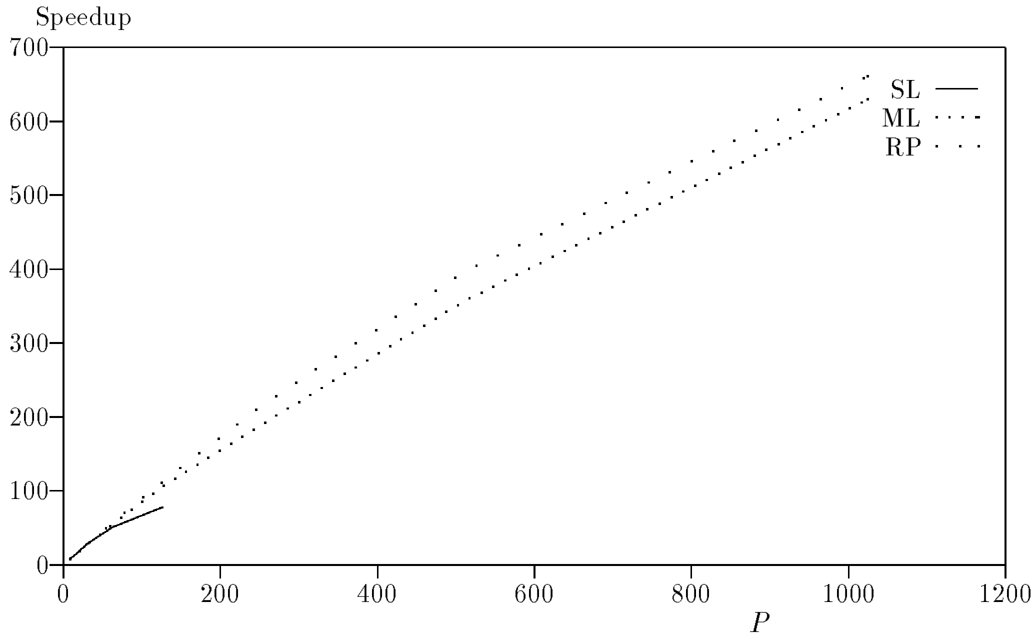


Figure 5: Average Speedups for SL and ML strategies compared to RP.

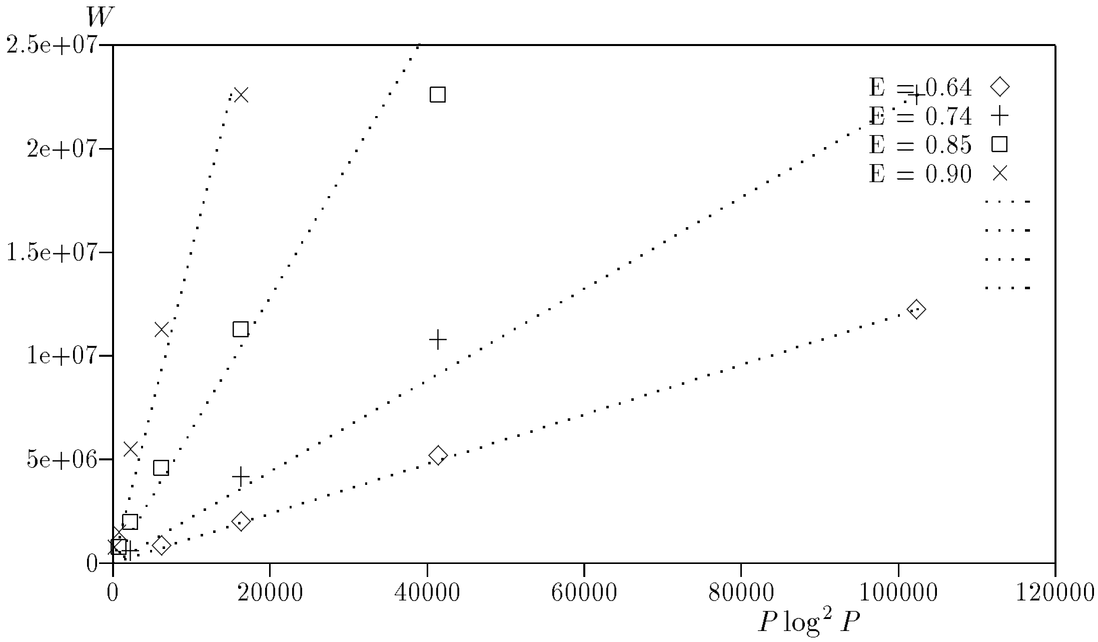


Figure 6: Experimental Isoefficiency curves for RP for different efficiencies.

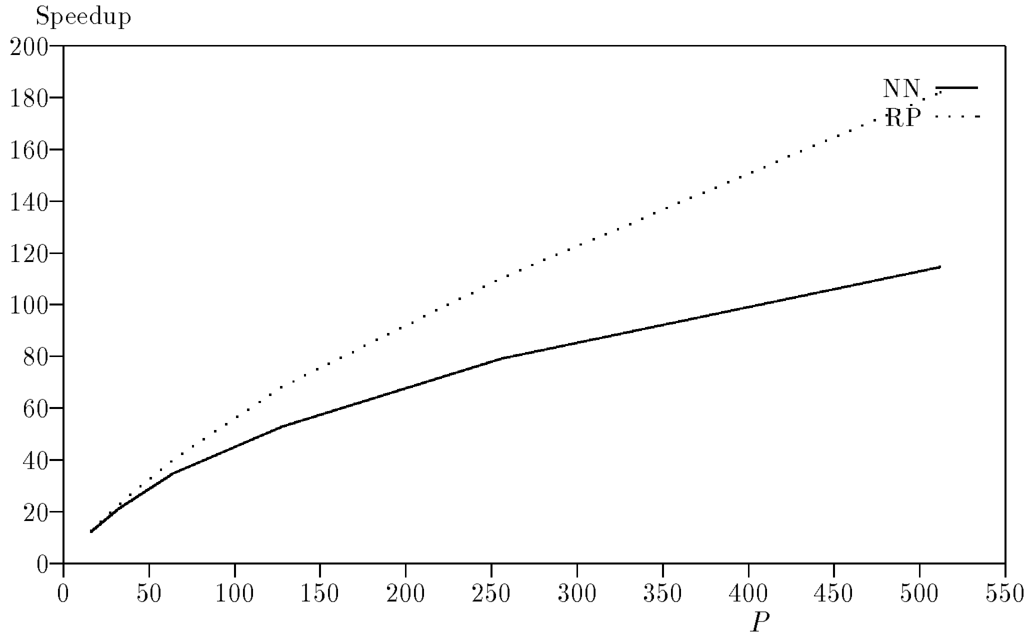


Figure 7: Effect of damaging the splitting function on speedup curves for RP and NN schemes.

combining hardware. However, from the number of communication messages, it can be inferred that, asymptotically, as P grows, GRR-M can eventually be expected to outperform RP and NN even on the Ncube/2^{TM}. Between RP and NN, RP has an asymptotically better scalability. Thus, with increasing number of processors, RP is expected to outperform NN. RP is also relatively insensitive to degradation of work splitting functions compared to NN. In our experiments, NN has been observed to perform slightly better than RP. This is attributed to the high quality of the splitting function and the moderate number of processors. Our experiments show that with a poorer splitting function, NN performs consistently poorer than RP even for very small number of processors.

Scalability analysis indicates that SB has a performance similar to that of GRR even though SB generates fewer work requests by not requesting any processors that are known to be idle. The poor scalability of both of these schemes indicates that neither of these are effective for larger number of processors. These conclusions have been experimentally validated.

The sender based scheme, ML, has been shown to have reasonable scalability, and has only a slightly worse performance compared to RP in our experiments. A major drawback of this scheme is that it requires the fine tuning of a number of parameters to obtain best possible performance. The random allocation scheme for the hypercube presented in [33] has an isoefficiency of $O(P \log P)$, which is optimal. However, for many problems, the maximum obtainable efficiency of this scheme has an upper bound much less than 1. This bound can be improved, and made closer to 1 by using effective methods for granularity control; but it is not clear if such mechanisms can be derived for practical applications. Also the memory requirements of these schemes are not well understood. In contrast, the best known receiver initiated scheme for the hypercube has an isoefficiency of $O(P \log^2 P)$, and its per-processor memory requirement is the same as that for corresponding serial implementations.

All the sender initiated schemes analyzed here use a different work transfer mechanism compared to the receiver initiated schemes. For instance, in the context of tree search, sender based schemes give the current state itself as a piece of work, whereas stack splitting and transfer is the common work transfer mechanism for receiver initiated schemes. The sender-based transfer mechanism is more efficient for problems for which the state description itself is very small but the stacks may grow very deep and stack splitting may become expensive. In addition, if the machine has a low message startup time (startup time component of the message passing time between two processors), the time required to communicate a stack may become a sensitive function of the stack size, which in turn may become large. In such domains, sender based schemes may potentially perform better than receiver based schemes.

A network of workstations provides us with a cheap and universally available platform for parallelizing applications. Several applications have been parallelized to run on a small number of workstations [1, 26]. For example, in [1] an implementation of parallel depth first branch and bound for VLSI floorplan optimization is presented. Linear speedups were obtained on up to 16 processors. The essential part of this branch-and-bound algorithm is a scalable load balancing technique. Our scalability analysis can be used to investigate the viability of using a much larger number of workstations for solving this problem. Recall that GRR has an overall isoefficiency of $O(P^2 \log P)$ for this platform. Hence, if we had 1024 workstations on the network, we can obtain the same efficiency on a problem instance which is 10240 times bigger compared to a problem instance being run on 16 processors ($10240 = \frac{1024^2 \log 1024}{16^2 \log 16}$). This result is of significance, as it indicates that it is indeed possible to obtain good efficiencies with large number of workstations. Scalability analysis also sheds light on the degree of scalability of such a system with respect to other parallel architectures such as hypercube and mesh multicomputers. For instance, the best applicable technique implemented on a hypercube has an isoefficiency function of $O(P \log^2 P)$. With this isoefficiency, we would be able to get identical efficiencies as those obtained on 16 processors by increasing the problem size 400 fold (which is $\frac{1024 \log^2 1024}{16 \log^2 16}$). We can thus see that it is possible to obtain good efficiencies even with smaller problems on the hypercube. We can thus conclude from isoefficiency functions that the hypercube offers a much more scalable platform compared to the network of workstations for this problem.

For the mesh architecture, we have analytically shown GRR-M and RP to have the best scalability. These are given by $O(P^{1.5} \log P)$ and $O(P^{1.5} \log^2 P)$ respectively. These figures indicate that these schemes are less scalable than their corresponding formulations for hypercube connected networks. However, it must be noted that GRR-M is within a log factor of the lower bound on isoefficiency for mesh architectures, given by $P^{1.5}$. Thus this scheme is near optimal for mesh architecture.

Speedup figures of individual load balancing schemes can change with technology dependent factors such as the CPU speed, the speed of communication channels etc. These performance changes can be easily predicted using isoefficiency analysis. For instance, if each of the CPUs of a parallel processor were made faster by a factor of 10, then U_{comm}/U_{calc} would increase by a factor of 10. From our analysis in Section 5.1, we can see that we would have to increase the size of our problem instance by a factor of 10 to be able to obtain the same efficiency. On the other hand, increasing communication speed by a factor of 10 would enable us to obtain the same efficiency on problem instances a tenth the size of the original problem size. This shows that the impact of changes in technology dependent factors is moderate. These can, however, be quite drastic for other algorithms such as FFT [13] and Matrix algorithms [12]. Being able to make such predictions is one of the significant advantages of isoefficiency analysis.

Two problem characteristics, communication coupling between subtasks and the ability to estimate work size, define a spectrum of application areas. Different load balancing strategies are needed for different points in this spectrum. In this paper, we have analyzed the point where there is no communication coupling and its not possible to estimate work size. It would be interesting to investigate optimal load balancing schemes for other points in this spectrum. For instance, there is another class of problems where the amount of work associated with a subtask can be determined but there is a very definite pattern of communication between subtasks. Examples can be found in scientific computations involving the solution of partial differential equations.

Dynamic load Balancing algorithms for SIMD processors are of a very different nature compared to those for MIMD architectures [9, 27, 32, 18]. Due to architectural constraints in SIMD machines, load balancing needs to be done on a global scale. In contrast, on MIMD machines, load can be balanced among a small subset of processors while the others are busy doing work. Further, in massively parallel SIMD machines, computations are of a fine granularity, hence communication to computation tradeoffs are very different compared to MIMD machines. Hence, the load balancing schemes developed for MIMD architectures may not perform well on SIMD architectures. Analysis similar to that used in this paper has been used to understand the scalability of different load balancing schemes for SIMD architectures and to determine best schemes [18].

Appendix A

Analysis of Load Balancing Algorithms for Mesh Architectures

Here we analyze the isoefficiency function for the above schemes on the mesh architecture.

Asynchronous Round Robin

As before, for this case, $V(P) = O(P^2)$. U_{comm} for this architecture is $\Theta(\sqrt{P})$. Substituting these values of U_{comm} and $V(P)$ into Equation 1, the isoefficiency for this scheme is given by $O(P^{2.5} \log P)$.

Global Round Robin

For isoefficiency due to communication overhead, $V(P) = O(P)$ and $U_{comm} = \Theta(\sqrt{P})$. Substituting these values into Equation 1, the isoefficiency due to communication overheads is $O(P^{1.5} \log P)$.

For isoefficiency due to contention, as discussed before, we need to equate the work at each processor (assuming efficient distribution) with number of messages that need to be processed by processor 0 during this time. Thus for isoefficiency,

$$\frac{W}{P} \sim V(P) \log W$$

or

$$\frac{W}{P} \sim P \log W$$

Solving this equation for isoefficiency, we get

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

Thus since the isoefficiency due to contention asymptotically dominates the isoefficiency due to communication overhead, the overall isoefficiency is given by $O(P^2 \log P)$.

GRR-M

From Section 5.2, we have $V(P) = \Theta(P)$. Also, message combining can be performed by propagating a work request left up to column 0 of the mesh and then propagating it up to processor 0. In this way, each increment on the variable TARGET takes $\Theta(\delta\sqrt{P})$, where δ is the time allowed for message combining at each intermediate processor. Thus, $U_{comm} = \Theta(\delta\sqrt{P})$.Substituting these values into Equation 1, the isoefficiency due to communication overheads is $O(P^{1.5} \log P)$. Since we have no contention this term defines the overall isoefficiency of this scheme.

Random Polling on a Mesh

As before, $V(P) = \Theta(P \log P)$ and $U_{comm} = \Theta(\sqrt{P})$. Substituting these values of U_{comm} and $V(P)$ into Equation 1, this scheme has an isoefficiency function of $O(P^{1.5} \log^2 P)$.

Appendix B

Effect of Quality of Splitting Function on Scalability.

Our theoretical analysis shows that the isoefficiency functions for RP and NN schemes are $O(P \log_{\frac{1}{1-\alpha}} P \times \log_2^2 P)$ and $\Omega(P^{\log_2 \frac{1+\frac{1}{\alpha}}{2}})$ respectively.

For RP, thus,

$$W \sim O(P \log_{\frac{1}{1-\alpha}} P \times \log_2^2 P)$$
$$\sim O(P \frac{\log_2 P}{\log_2 \frac{1}{1-\alpha}} P \times \log_2^2 P)$$

Now,

$$\frac{1}{1-\alpha} = 1 + \alpha + \alpha^2 + ...$$

and for $\alpha << 1$, $\frac{1}{1-\alpha} \sim 1 + \alpha$. Substituting in relation for W, we get,

$$W \sim O(P \frac{\log_2 P}{\log_2(1+\alpha)} \times \log_2^2 P)$$

Also,

$$\log_e(1+\alpha) = \alpha - \frac{\alpha^2}{2} + \frac{\alpha^3}{3}...$$

Again, for $\alpha << 1$, we can neglect higher powers of α, and

$$\log_e(1+\alpha) \sim \alpha$$

Substituting in relation for W, we get,

$$W \sim O(\frac{1}{\alpha} \times P \log^3 P) \tag{4}$$

Similarly, for the case of NN,

$$W \sim \Omega(P^{\log_2 \frac{1+\frac{1}{\alpha}}{2}})$$
$$\sim \Omega(P^{\log_2 2 \times (\frac{1+\frac{1}{\alpha}}{4})})$$
$$\sim \Omega(P^{1+\log_2 (\frac{1+\frac{1}{\alpha}}{4})})$$
$$\sim \Omega(P \times P^{\log_2 (\frac{1+\frac{1}{\alpha}}{4})})$$

Thus we get

$$W \sim \Omega(P \times P^{\log_2 \frac{1}{\alpha} - 2}) \tag{5}$$

Now consider Equations 4 and 5. On damaging the splitting function so as to reduce α by a factor of 2, isoefficiency for RP increases by a factor of 2 whereas that for NN increases by a factor of P. This clearly demonstrates that the scalability of NN is significantly degraded by a poor splitting function.

Acknowledgements

The authors sincerely acknowledge Dr. Robert Benner at Sandia National Labs for providing us with access to the 1024 node Ncube/2^{TM}. Vivek Sarin and Anshul Gupta provided many helpful comments on the earlier drafts of this paper. The idea of software message combining on the hypercube was suggested by Chuck Seitz. The authors also acknowledge the constructive comments and suggestions by the referees.

References

[1] S. Arvindam, Vipin Kumar, and V. Nageshwara Rao. Floorplan optimization on multiprocessors. In *Proceedings of the 1989 International Conference on Computer Design (ICCD-89)*, 1989. Also published as MCC Tech Report ACT-OODS-241-89.

[2] 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.

[3] 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.

[4] E. Charniak and D. McDermott. *Introduction to Artificial Intelligence*. Addison Wesley, 1985.

[5] M. Davis, G. Logeman, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5, number 7, 1962.

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

[7] 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.

[8] 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.

[9] Roger Frye and Jacek Myczkowski. Exhaustive search of unstructured trees on the connection machine. In *Thinking Machines Corporation Technical Report*, 1990.

[10] 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.

[11] 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.

[12] Anshul Gupta and Vipin Kumar. The scalability of Matrix Multiplication Algorithms on parallel computers. Technical Report TR 91-54, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, 1991.

[13] Anshul Gupta and Vipin Kumar. 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 will appear in IEEE Transactions on Parallel and Distributed Systems, 1993.

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

[15] 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.

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

[17] L. V. Kale. Comparing the performance of two dynamic load distribution methods. In *Proceedings of International conference on Parallel Processing*, pages 8–12, 1988.

[18] George Karypis and Vipin Kumar. Unstructured Tree Search on SIMD Parallel Computers. Technical Report 92–21, Computer Science Department, University of Minnesota, 1992. A short version of this paper appears in the Proceedings of Supercomputing 1992 Conference, November 1992.

[19] R. Keller and F. Lin. Simulated performance of a reduction based multiprocessor. *IEEE Computers*, July 1984 1984.

[20] Kouichi Kimura and Ichiyoshi Nobuyuki. Probabilistic analysis of the efficiency of the dynamic load distribution. In *Proceedings of 1991 Distributed Memory and Concurrent Computers*, 1991.

[21] Vipin Kumar and Anshul Gupta. Analyzing the scalability of parallel algorithms and architectures: A survey. In *Proceedings of the 1991 International Conference on Supercomputing*, June 1991. also appear as an invited paper in the Proc. of 29th Annual Allerton Conference on Communuication, Control and Computing, Urbana,IL, October 1991.

[22] 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.

[23] Vipin Kumar and V. Nageshwara Rao. Parallel depth-first search, part II: Analysis. *International Journal of Parallel Programming*, 16 (6):501–519, 1987.

[24] Vipin Kumar and V. Nageshwara Rao. Load balancing on the hypercube architecture. In *Proceedings of the 1989 Conference on Hypercubes, Concurrent Computers and Applications*, pages 603–608, 1989.

[25] 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.

[26] Kai Li. Ivy: A shared virtual memory system for parallel computing. In *Proceedings of International conference on Parallel Processing: Vol II*, pages 94–101, 1988.

[27] A. Mahanti and C. Daniels. Simd parallel heuristic search. *To appear in Artificial Intelligence*, 1992.

[28] B. Monien and O. Vornberger. Parallel processing of combinatorial search trees. In *Proceedings of International Workshop on Parallel Algorithms and Architectures*, May 1987.

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

[30] 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.

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

[32] C. Powley, R. Korf, and C. Ferguson. Ida* on the connection machine. *To appear in Artificial Intelligence*, 1992.

[33] Abhiram Ranade. Optimal speedup for backtrack search on a butterfly network. In *Proceedings of the Third ACM Symposium on Parallel Algorithms and Architectures*, 1991.

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

[35] 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.

[36] Wei Shu and L. V. Kale. A dynamic scheduling strategy for the chare-kernel system. In *Proceedings of Supercomputing 89*, pages 389–398, 1989.

[37] Vineet Singh, Vipin Kumar, Gul Agha, and Chris Tomlinson. Scalability of parallel sorting on mesh multicomputers. *International Journal of Parallel Programming*, 20 (2), 1991. A short version of this paper appears in the Proceedings of the Fifth International Parallel Processing Symposium, 1991.

[38] Douglas R. Smith. Random trees and the analysis of branch and bound proceedures. *Journal of the ACM*, 31 No. 1, 1984.

[39] Nageshwara Rao Vempaty, Vipin Kumar, and Richard Korf. Analysis of heuristic search algorithms. In *Proceedings of the National Conf. on Artificial Intelligenc e (AAAI-91)*, 1991.

[40] Benjamin W. Wah, G.J. Li, and C. F. Yu. Multiprocessing of combinatorial search problems. *IEEE Computers*, June 1985 1985.

[41] 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.

[42] Jinwoon Woo and Sartaj Sahni. Hypercube computing : connected components. *Journal of Supercomputing*, 1991.

[43] Jinwoon Woo and Sartaj Sahni. Computing biconnected components on a hypercube. *Journal of Supercomputing*, June 1991.