

## RANDOM SEEKING: A GENERAL, EFFICIENT, AND INFORMED RANDOMIZED SCHEME FOR DYNAMIC LOAD BALANCING

NIHAR R. MAHAPATRA

*Department of Computer Science & Engineering, State University of New York at Buffalo  
137 Bell Hall, Buffalo, NY 14260-2000, U.S.A.*

and

SHANTANU DUTT

*Department of Electrical Engineering & Computer Science, University of Illinois at Chicago  
851 South Morgan Street (M/C 154), Chicago, IL 60607-7053, U.S.A.*

Received April 1999

Revised May 2000

Communicated by Sanguthevar Rajasekaran

### ABSTRACT

We propose a completely general, informed randomized dynamic load balancing method called random seeking (RS) suitable for parallel algorithms with characteristics found in many search algorithms used in artificial intelligence and operations research and many divide-and-conquer algorithms. In it, source processors randomly seek out sink processors for load balancing by flinging “probe” messages. These probes not only locate sinks, but also collect load distribution information which is used to efficiently regulate load balancing activities. We empirically compare RS with a well-known randomized dynamic load balancing method, the random communication (RC) strategy, by using them in parallel best-first branch-and-bound algorithms on up to 512 processors of a parallel system. We find that the RC execution times are more than those of RS by 16–67% when used to perform combined dynamic quantitative and qualitative load balancing, and by 9–74% when used to perform only dynamic quantitative load balancing.

*Keywords:* Dynamic load balancing, mixed-integer programming, qualitative load balancing, quantitative load balancing, randomization.

### 1. Introduction

A load balancing strategy is used to partition a problem among the processors of a parallel machine so that all processors are usefully engaged. To maximize processor utilization, the total computational load associated with the problem must be as equitably distributed among the processors as possible. Here we are concerned with load balancing in parallel algorithms with the following characteristics.

- The work available at any processor either (1) comprises of independent work pieces or (2) can be partitioned into such pieces as long as it is more than

some non-decomposable unit; in this case, work partitioning takes much less time compared to work processing.

- Work pieces considered for transfer by a load balancing strategy are large enough so that the time to transfer a piece of work from one processor to another is small compared to its processing time.
- It is not possible or is very difficult to estimate the processing time for a piece of work.

These are characteristics of many search algorithms used in artificial intelligence and operations research and many divide-and-conquer algorithms [22, 14, 6]. Since different work pieces in these applications can be of widely differing sizes and since their sizes are unknown, any static partitioning of work pieces among processors may result in excessive processor idling. Hence some form of dynamic *quantitative load balancing* is required to transfer work from busy “source” processors to idle “sink” processors to reduce processor idling. Moreover, in some applications like best-first branch-and-bound search (BFS) [4], a commonly used method for solving a number of practical combinatorial optimization problems [22, 8, 29, 6], work pieces generated during execution are processed in a best-first manner with regard to their costs [27]. When such methods are parallelized, it is not only important to perform quantitative load balancing, but also some form of *qualitative load balancing* to transfer good quality (low cost) work from “source” processors to “sink” processors with bad quality (high cost) work to curtail *non-essential work* (work not processed by a sequential BFS algorithm).

Dynamic quantitative and qualitative load balancing are the two types of load balancing that parallel algorithms with the above characteristics may require to maximize processor utilization [14, 5]. A number of schemes to perform such load balancing have been proposed [1, 4, 5, 7, 9, 10, 12, 13, 14, 15, 17, 21, 23, 24, 25, 28, 30]. In this paper, we consider randomized load balancing schemes for parallel algorithms with the above characteristics. Randomized schemes have the advantage of being simple and easily implementable. The random communication (RC) strategy of [10, 12] has been proposed previously for performing both quantitative and qualitative load balance. In it, a processor on generating a new piece of work transfers it to a random processor. The random quantitative load balancing schemes of [23, 30] are similar to the RC strategy. Although, due to randomization, work will be sent from source to sink processors in the RC strategy, reasonable likelihood exists for useless work transfers from source to source, from sink to sink, and from sink to source processors. These unnecessary work transfers can aggravate existing load imbalances in the parallel algorithm. Also, because of them, the overhead per useful work transfer can be quite high. Due to these reasons, substantial scope exists for obtaining better performance using some type of informed randomized method that avoids these pitfalls.

In this paper, we propose a completely general, informed randomized scheme called *random seeking* (RS) for dynamic load balancing. In it, source processors fling “probe messages” to random processors. The probes seek out sink processors by visiting processors randomly, and then work transfer takes place from source to

sink processors. Moreover, while seeking sink processors, the probes also collect information about load distribution across processors. This information is used to determine whether a processor is a source or sink relative to other processors, and to regulate the rate at which probes are flung from source processors so that the load balancing overhead of flinging probes matches the need for load balance as reflected by the number of sink processors. Thus RS not only avoids the pitfall of useless work transfers in the RC strategy, but also utilizes load distribution information to minimize load balancing overhead. Random seeking is designed to take advantage of the fact that in most parallel algorithms with characteristics mentioned earlier, the load distribution across processors does not change drastically in a short time (i.e., most processors that are sources one instant do not become sinks the next, and vice versa). In case, this is not true for an application, even then RS will perform better than RC since it performs only useful work transfers from source to sink processors.

First, we apply our RS strategy to perform combined dynamic quantitative and qualitative load balancing in parallel BFS. Next, we modify parallel BFS so that it requires only dynamic quantitative load balancing, and apply the RS scheme for this purpose. In both cases, we compare the performance of RS with RC. All comparisons are made on parallel BFS algorithms used to solve mixed-integer programming (MIP) problems from the MIPLIB benchmark set [3] on up to 512 processors of a parallel machine. We find that the RC execution times are more by 16-67% for combined quantitative and qualitative load balancing, and by 9-74% for only quantitative load balancing than those of RS.

The rest of the paper is organized as follows. In Sec. 2, we describe the RS strategy in detail. First, we provide an overview of RS in Sec. 2.1, and then in Sec. 2.2 we discuss how information is gathered in RS to make intelligent load balancing decisions. Sec. 2.3 discusses the random seeking process, Sec. 2.4 explains the difference between apparent and actual work load, and Sec. 2.5 describes how load balancing activities are efficiently regulated to minimize overhead. Finally, in Sec. 2.6, we discuss an interesting extension to RS. Next, in Sec. 3, we briefly describe the sequential BFS algorithm, its application to solving MIP, and our generic approach to BFS parallelization on distributed-memory machines. In Sec. 4, we explain how RS is applied to perform both combined dynamic quantitative and qualitative load balancing (Sec. 4.1), and only dynamic quantitative load balancing (Sec. 4.2) in parallel BFS algorithms. Next, in Sec. 5, we present performance results comparing RS to RC. Conclusions are in Sec. 6.

## 2. The Random Seeking Strategy

In this section, we describe our general RS strategy for informed, randomized dynamic load balancing.

### 2.1. Overview

First, we provide an overview of RS. Every processor  $i$  has a *load attribute*  $\mathcal{L}(i)$

associated with it that characterizes its work load and is application dependent (e.g., it may be the number or cost of work pieces  $i$  has). Depending upon their load attributes, any two processors  $i$  and  $j$  may either have a *source-sink* (denoted by  $\mathcal{L}(i) \succ \mathcal{L}(j)$ ), *peer-peer* (denoted by  $\mathcal{L}(i) \approx \mathcal{L}(j)$ ), or *sink-source* (denoted by  $\mathcal{L}(i) \prec \mathcal{L}(j)$ ) *relationship*. The load attribute should be defined so that: (1)  $\mathcal{L}(i) \succ \mathcal{L}(j)$  implies  $i$  is more likely than  $j$  to have useful work load to process, and can grant some of its work load to the latter to make it equally likely to perform useful computation; (2)  $\mathcal{L}(i) \approx \mathcal{L}(j)$  indicates if one processor is performing useful computation, then the other is also likely to be doing the same; and (3)  $\mathcal{L}(i) \prec \mathcal{L}(j)$  signifies the converse of case (1). The RS strategy strives to establish peer-peer relationships between all processor pairs by probabilistically locating source-sink pairs via “probe” messages flung to random processors and transferring work from the sources to the corresponding sinks, and thereby attempts to maximize processor utilization. These probes not only locate sinks, but also collect load distribution information which is used to efficiently regulate load balancing activities. The load balancing overhead of RS is directly related to how stringent the load balancing requirement implied by peer-peer relationships is, which should therefore be chosen in any application to maximize processor utilization at the minimum load balancing overhead. In our description of RS below, we will use terms and operators (like “load attribute” and “ $\succ$ ” above) that will have application-specific meanings, and these we will point out. Later in Sec. 4, we will give two examples of how these terms and operators are defined when we apply RS to parallel BFS.

## 2.2. Information Gathering in Random Seeking

As mentioned earlier, RS performs intelligent load balancing by gathering and using information obtained during probe visits. Here we describe how this information is collected. In RS, every processor  $i$  maintains two frequently updated estimates: (1) An estimate of the fraction  $\sigma(i)$  of all processors that are sources relative to it; and (2) An estimate of the fraction  $\rho(i)$  of all processors that are sinks relative to it, *and* to which it will act as a source of work—by this we mean the following. Suppose  $i$  is a source relative to four processors, and these processors are sinks relative to two, four, four, and one processors (including  $i$ ), respectively. Then if the task of providing work to a sink processor is equally divided among its source processors, the fraction of processors that are sinks relative to  $i$ , and to which  $i$  has the burden of supplying work is  $\rho(i) = \frac{1}{P} \cdot (\frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{1}) = \frac{2}{P}$ , where  $P$  is the total number of processors. In other words, if the  $\sigma$  estimates of all processors are accurate, then  $\rho(i) = \frac{1}{P} \cdot \sum_j \frac{1}{\sigma(j) \cdot P}$ , where  $j$  is a sink relative to  $i$ . We denote the  $n$ th updated values of these estimates by  $\sigma_n(i)$  and  $\rho_n(i)$ . The estimates should be initialized to their expected values for the application under consideration. For simplicity, in our implementation we assume processors are sources, sinks, and peers relative to other processors with equal probabilities at the beginning of the parallel algorithm, and so set  $\sigma_1 := \frac{1}{3}$  in all processors. Since on the average one-third of the processors will be sinks relative to any given processor,  $\sigma_1 := \frac{1}{3}$  implies  $\rho_1 := \frac{1}{P} \cdot \sum_j \frac{1}{\sigma_1(j) \cdot P} = \frac{1}{P} \cdot \frac{P}{3} \cdot \frac{1}{\frac{1}{3} \cdot P} = \frac{1}{P}$  in all processors.

We will see that during random seeking, new information will frequently become available regarding the fraction of a random set of processors (less than  $P$ ) that are sources or sinks relative to a processor  $i$ . Suppose after the  $n$ th update of  $\sigma$  in  $i$ , information becomes available about the fraction  $r_n$  of a random set of processors that are sources relative to  $i$ . Then we incorporate this new information in the  $(n + 1)$ th update as follows.

$$\sigma_{n+1}(i) = \alpha \cdot r_n + (1 - \alpha) \cdot \sigma_n(i), \quad 0 \leq \alpha \leq 1. \quad (1)$$

The above formula gives us an exponential average of all previous information, since  $\sigma_{n+1}$  can be expressed as [31]:

$$\sigma_{n+1} = \alpha \cdot r_n + \alpha \cdot (1 - \alpha) \cdot r_{n-1} + \alpha \cdot (1 - \alpha)^2 \cdot r_{n-2} + \dots + \alpha \cdot (1 - \alpha)^{n-1} \cdot r_1 + (1 - \alpha)^n \cdot \sigma_1. \quad (2)$$

The parameter  $\alpha$  controls the relative weight of recent (1st term in Eq. 1) and past (2nd term in Eq. 1) history in the estimation. If  $\alpha$  is close to 0, then the effect of past history dominates, while if  $\alpha$  is close to 1, the effect of recent history is predominant. Past history (contained in  $\sigma_n$ ) is important because it represents a larger sample of processors, and recent history (contained in  $r_n$ ) is important because it includes recent changes in relationships between  $i$  and other processors. We chose  $\alpha = 0.25$  in our implementation so that recent history is captured well in the estimation. As more and more information becomes available and is incorporated during random seeking, the  $\sigma$  estimates become more and more accurate. Thus some time after the algorithm starts, these estimates will be reasonably accurate. We update the  $\rho$  estimates in processors as new information becomes available in the same manner as in Eq. 1 for  $\sigma$ .

### 2.3. The Random Seeking Process

We now describe the random seeking process. At regular intervals of time  $\Delta$  (or at an average time interval  $\Delta$ ), a processor  $i$  flings a certain number of *probe* messages (to be discussed later) each containing its load attribute  $\mathcal{L}(i)$  and its estimate  $\sigma(i)$  to random processors. A processor receiving a probe compares  $\mathcal{L}(i)$  with its own load attribute and determines its relationship with  $i$ . If it is not a sink relative to  $i$ , or if it has sent out a work-request message to some processor that has not yet been serviced, it reflings the probe to another random processor (that is not  $i$ ). The probe is reflung similarly from all non-sink processors and processors with unserviced work requests until it arrives at a processor  $k$  that is a sink relative to  $i$  and does not have unserviced work requests, or until the probe has been flung  $F_{max}$  times. In either case, the probe is returned to  $i$ . A probe that has not yet returned to its originating processor is said to be *active*. At all processors the probe visits, their  $\sigma$  estimates are updated using Eq. 1 with  $r_n = 1$  or 0 depending upon whether they were found to be sinks or not, respectively, relative to  $i$ . Also, their  $\rho$  estimates are updated with  $r_n = \frac{1}{\sigma(i) \cdot P}$  or 0 depending upon whether they were found to be sources or not, respectively, relative to  $i$ . During their visit to the different processors, the probes accumulate information necessary to update the  $\sigma$

and  $\rho$  estimates of  $i$ . The new information  $r_n$  used to update  $\sigma(i)$  is the fraction of all visited processors that were found to be sources relative to  $i$ . For updating  $\rho(i)$ , the new information  $r_n = \frac{1}{F} \cdot \sum_j \frac{1}{\sigma(j) \cdot P}$ , where  $F$  is the number of processors that a probe visits and  $j$  is any sink processor (relative to  $i$ ) among them. After updating its two estimates,  $i$  discards a returned probe.

If during a probe's visits, a sink processor  $k$  was found, then  $k$  sends a work-request message to  $i$  (this message may be piggy-backed onto the probe which is also destined for  $i$ ) containing its load attribute  $\mathcal{L}(k)$ . Since  $i$  is continuously processing its work load, and since there may be multiple active probes from  $i$  and other processors at a time that may cause work transfers from  $i$ , the work load of  $i$  is continuously changing. Therefore when  $i$  receives the work request from  $k$ , it transfers some of its work to  $k$  to achieve load balance (i.e., to achieve at least a peer-peer relationship with  $k$ ) only if it is a source relative to  $k$  at that time. Fig. 1 depicts an overview of the random seeking process discussed above.

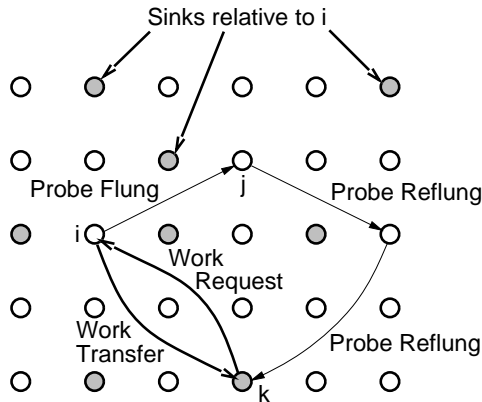


Fig. 1. Overview of the random seeking process: A source processor  $i$  flings a probe message to a random processor  $j$ . Since  $j$  is not a sink relative to  $i$ , the probe is reflung to another random processor. The probe is reflung similarly from all non-sink processors and processors with unserved work requests until it arrives at a sink processor  $k$  that does not have unserved work requests (or until the probe has been flung  $F_{max}$  times). The probe then returns to  $i$ . Processor  $k$  sends a work request message to  $i$ , and receives in response part of  $i$ 's work load so as to achieve load balance with  $i$ . During the probe's visits, information about the distribution of load across processors is collected so as to judiciously regulate probe flinging rate.

#### 2.4. Apparent vs. Actual Work Load

From the above description, we see that at any given time one or more probes may be active from each processor. Each probe  $\mathcal{P}_1$  from a processor  $i$  contains its load attribute that is used to determine relative sink processors and that represents its ability to supply work load to such processors. Suppose the probability that  $\mathcal{P}_1$  will be able to locate a sink is  $p$ —we will shortly compute  $p(i)$  for any processor  $i$  at an arbitrary time instant. Then an immediately succeeding probe  $\mathcal{P}_2$  issued from  $i$  while  $\mathcal{P}_1$  is active should represent a diminished ability to supply work with prob-

ability  $p$ , and an undiminished ability with probability  $(1 - p)$ . Thus, although the *apparent work load* with  $i$  is  $\mathcal{L}(i)$ , its *actual work load* probabilistically diminishes with each probe issued. Therefore, we maintain in each processor  $i$  a *dynamic difference attribute*  $w_d(i)$  that contains information necessary to obtain the difference between its apparent work load  $\mathcal{L}(i)$  and its actual work load which we denote by  $\mathcal{L}_d(i)$  and refer to as  $i$ 's *dynamic load attribute*; we denote the relationship between these three terms by  $\mathcal{L}_d(i) := \mathcal{L}(i) \ominus w_d(i)$ , where the operator  $\ominus$  is application dependent. The difference attribute  $w_d(i)$  may or may not be in the same units as  $\mathcal{L}(i)$  and  $\mathcal{L}_d(i)$  depending upon the application.  $\mathcal{L}(i) = 0$  or  $\mathcal{L}_d(i) = 0$  imply zero apparent and actual number of work pieces, respectively, and  $\mathcal{L}(i) > 0$  or  $\mathcal{L}_d(i) > 0$  imply one or more apparent and actual number of work pieces. Similarly,  $w_d(i) = 0$  corresponds to zero number of work pieces, and  $w_d(i) > 0$  to one or more work pieces. The operator  $\ominus$  has the following properties:  $x \ominus y$  corresponds to less amount of work than  $x$ , if  $y > 0$ , and  $x \ominus y = x$ , if  $y = 0$ . Hence, whenever a probe is sent out, processor  $i$  determines  $\mathcal{L}_d(i) := \mathcal{L}(i) \ominus w_d(i)$ , and sends  $\mathcal{L}_d(i)$  with the probe instead of  $\mathcal{L}(i)$ , since  $\mathcal{L}_d(i)$  more accurately represents the work load of  $i$ .

At the beginning of the parallel algorithm,  $w_d(i) := 0$ , meaning actual work load is the same as apparent work load or  $\mathcal{L}_d(i) := \mathcal{L}(i)$ . Upon issuing a probe, processor  $i$  *increments*  $w_d(i)$  with the probability  $p(i)$  of being able to locate a sink. The increment operation is application dependent. For example, if only quantitative load balancing needs to be performed and  $\mathcal{L}(i) =$  number of work pieces with  $i$ , and if  $i$  is supposed to give out  $m$  work pieces in response to a work request, then  $\ominus$  is the “ $-$ ” (subtraction) operator, and upon issuing a probe,  $i$  updates as  $w_d(i) := w_d(i) + m$  with probability  $p(i)$ . A queue denoted *inc\_queue<sub>i</sub>* records whether an increment operation was performed. Thus, if  $w_d(i)$  was incremented upon issuing a probe, a 1 is enqueued into it, otherwise a 0 is enqueued. Subsequently, when one of its probes returns to  $i$ , processor  $i$  dequeues a value from *inc\_queue<sub>i</sub>* and *decrements*  $w_d(i)$  if the value is 1—in the above example,  $w_d(i) := w_d(i) - m$ .<sup>a</sup> Thus, when  $i$  has no active probes,  $\mathcal{L}_d(i) = \mathcal{L}(i)$ .

## 2.5. Efficiently Regulating Load Balancing Activities

A frequently encountered problem in source-initiated schemes (e.g., that in [28]) is that multiple source processors may independently determine a common sink and send work to it, thereby converting the previously sink processor into a “super source.” We avoid this problem in RS, which is a source-initiated method, by bypassing sink processors with unserved work requests during probing.

We now compute the probability  $p(i)$  that a probe issued from a processor  $i$  will be able to locate a sink in its at most  $F_{max}$  visits. This is done as follows assuming  $\rho(i)$  (the probability estimate of the fraction of processors that  $i$  is a source relative to and will transfer work to) to be accurate, which we recall will be the case some time after the algorithm starts.

$$p(i) = \rho(i) \sum_{n=1}^{F_{max}} (1 - \rho(i))^{n-1} = 1 - (1 - \rho(i))^{F_{max}}, \quad (3)$$

---

<sup>a</sup>It is not necessary for the dequeued value to correspond to the enqueued value of the returned probe. The only purpose of the queue is to facilitate decrementing of  $w_d(i)$  in the same probabilistic manner as it was incremented, and to ensure at the same time that whenever there are no active probes, the total number of increments and decrements of  $w_d(i)$  are equal.

where the  $n$ th term in the series represents the probability that a sink is found in the  $n$ th visit. Note that ideally, only as many probes should be flung from a processor as are required to locate relative sinks. Extra probes will mean unnecessary load balancing overhead, and fewer probes will mean non-optimal processor utilization. Therefore, in every time interval  $\Delta$ , we fling from a processor  $i$  a number of probes proportional to the number of relative sinks (i.e.,  $P \cdot \rho(i)$ ) and inversely proportional to the probability  $p(i)$  of being able to locate such a sink. Hence only as much effort is invested in load balancing (in terms of issuing probes) as required to correct the prevailing load imbalance.

Next, we compute the average number of flings or visits  $\nu$  required by a probe to locate a sink as follows—the  $n$ th term in the equation below represents the probability that a sink is found in the  $n$ th visit.

$$\begin{aligned} \nu &= \rho \cdot \sum_{n=1}^{F_{max}} n \cdot (1 - \rho)^{n-1} \leq \rho \cdot \sum_{n=1}^{\infty} n \cdot (1 - \rho)^{n-1} \\ &= \rho \cdot \frac{d}{d\rho} \left[ - \sum_{n=0}^{\infty} (1 - \rho)^n \right] = \rho \cdot \frac{d}{d\rho} \left[ -\frac{1}{\rho} \right] = \frac{1}{\rho} \end{aligned} \quad (4)$$

This gives us:

**Theorem 1** *The average number of flings taken by a probe issued by any processor  $i$  in the random seeking strategy to locate a sink relative to  $i$  is at most  $\frac{1}{\rho(i)}$ .*

Therefore, more the number of sinks that a processor  $i$  needs to transfer work to, faster the sinks are located and work transferred. A formal description of the RS strategy is given in Fig. 2.

To apply the RS strategy to a given parallel algorithm with characteristics mentioned in the introduction, we need the following information. (1) The definition of static and dynamic load attributes of a processor; (2) The definition of operators “ $\succ$ ,” “ $\approx$ ,” and “ $\prec$ ” used to determine source-sink, peer-peer, and sink-source relationships between any two processors; and (3) The work transferred from a processor in response to a work request. In Sec. 4, we will discuss how to apply the RS strategy to perform dynamic quantitative and qualitative load balancing in parallel BFS algorithms.

## 2.6. Globalized vs. Localized Random Seeking

Before leaving this section, we briefly discuss an useful extension to random seeking. Random seeking as described above can be characterized as *global random seeking* (GRS), since probes are flung from a processor to all other processors with uniform probability. Since most parallel systems are not fully connected, the global nature of the communication pattern in global random seeking may cause performance degradation due to link contention and hot spots at higher probe flinging rates than schemes which require more localized communication. Completely localized communication can be achieved by using *local random seeking* (LRS), in which processors fling probes to only random neighboring processors. Clearly, better load balance will be achieved in global than local random seeking. Therefore, at low enough probe flinging rates, global random seeking will give better performance, and at high enough flinging rates, local random seeking will yield better performance. Between these extremes, one can choose to fling to nearer processors with a

**Algorithm RANDOM\_SEEKING( $i$ )**

/\* Algorithm RANDOM\_SEEKING is used in a parallel algorithm on  $P$  processors to perform dynamic load balancing. \*/

**Begin**

Processor  $i$ ,  $0 \leq i < P$ , executes the following steps:

1. **Initialization:**

$\sigma(i) := \frac{1}{3}$ ;  $\rho(i) := \frac{1}{P}$ ;  $num\_unserviced\_wk\_reqs := 0$ ;  $inc\_queue_i := NULL$ ;  $w_d(i) := 0$ ;  
Initialize  $\mathcal{L}(i)$ ,  $\alpha$ ,  $\Delta$ , and  $F_{max}$ . /\* Here  $num\_unserviced\_wk\_reqs$  stores the number of work requests from  $i$  that are remaining to be serviced at any time. \*/

**Repeat**

2. **Probe Issue:**

**After** (every  $\Delta$  time interval) **begin**

$p(i) := 1 - (1 - \rho(i))^{F_{max}}$ ;  $n :=$  number of probes to be flung  $\propto \frac{P \cdot \rho_i}{p(i)}$ ;  
 $f := r_\sigma(i) := r_\rho(i) := 0$ ; /\*  $f$  is the number of times probe has been flung;  $r_\sigma(i)$  and  $r_\rho(i)$  are the fraction of processors encountered so far by the probe that are sources and sinks, respectively, relative to  $i$ . \*/  
**for** ( $l := 0$ ;  $l < n$ ;  $l++$ ) **then begin** /\* Fling  $n$  probes, one by one to random procs. \*/  
 $\mathcal{L}_d(i) := \mathcal{L}(i) \ominus w_d(i)$ ; Fling probe  $P(i, \mathcal{L}_d(i), \sigma(i), f, r_\sigma(i), r_\rho(i))$  to a random processor  $j$ ; /\*  $P$  denotes a probe, and its arguments, the information it contains. \*/  
Generate a random number  $y$  uniformly distributed over  $[0, 1]$ ;  
**if** ( $y \in [0, p(i))$ ) **then begin** /\* With prob.  $p(i)$ , the probe will locate a sink. \*/  
Increment  $w_d(i)$ ;  $ENQUEUE(inc\_queue_i, 1)$ ; /\* Enqueue 1 into  $inc\_queue_i$ . \*/  
**endif**  
**else**  $ENQUEUE(inc\_queue_i, 0)$ ;  
**endfor**

**EndAfter**

3. **Probe Visit:**

**On** (receiving  $P(j, \mathcal{L}_d(j), \sigma(j), f, r_\sigma(j), r_\rho(j))$ ) **begin**

$f := f + 1$ ;  $\mathcal{L}_d(i) := \mathcal{L}(i) \ominus w_d(i)$ ;  
**if** ( $\mathcal{L}_d(i) \succ \mathcal{L}_d(j)$ ) **then begin** /\*  $i$  is a source relative to  $j$ . \*/  
 $r_\sigma(j) := \frac{r_\sigma(j) \cdot (f-1) + 1}{f}$ ;  $r_\rho(i) := \frac{1}{\sigma(i) \cdot P}$ ;

**endif**

**else begin**

$r_\sigma(j) := \frac{r_\sigma(j) \cdot (f-1)}{f}$ ;  $r_\rho(i) := 0$ ;

**endelse**

**if** ( $\mathcal{L}_d(i) \prec \mathcal{L}_d(j)$ ) **then begin** /\*  $i$  is a sink relative to  $j$ . \*/

$r_\rho(j) := \frac{r_\rho(j) \cdot (f-1) + \frac{1}{\sigma(i) \cdot P}}{f}$ ;  $r_\sigma(i) := 1$ ;

**endif**

**else begin**

$r_\rho(j) := \frac{r_\rho(j) \cdot (f-1)}{f}$ ;  $r_\sigma(i) := 0$ ;

**endelse**

$\sigma(i) := \alpha \cdot r_\sigma(i) + (1 - \alpha) \cdot \sigma(i)$ ;  $\rho(i) := \alpha \cdot r_\rho(i) + (1 - \alpha) \cdot \rho(i)$ ;

**if** ( $(\mathcal{L}_d(i) \prec \mathcal{L}_d(j))$  **and** ( $num\_unserviced\_wk\_reqs = 0$ )) **or** ( $f = F_{max}$ )

Return  $P(r_\sigma(j), r_\rho(j))$  to  $j$ ;

**if** ( $\mathcal{L}_d(i) \prec \mathcal{L}_d(j)$ ) **and** ( $num\_unserviced\_wk\_reqs = 0$ ) **then begin**

Send work request  $R(i, \mathcal{L}_d(i))$  to  $j$ ;

$num\_unserviced\_wk\_reqs := num\_unserviced\_wk\_reqs + 1$ ;

**endif**

**else if** ( $f < F_{max}$ ) Fling  $P(j, \mathcal{L}_d(j), \sigma(j), f, r_\sigma(j), r_\rho(j))$  to a random proc. that is not  $j$ ;

**EndOn**

4. **Probe Return:**

**On** (receiving a return probe  $P(r_\sigma(i), r_\rho(i))$ ) **begin**

$\sigma(i) := \alpha \cdot r_\sigma(i) + (1 - \alpha) \cdot \sigma(i)$ ;  $\rho(i) := \alpha \cdot r_\rho(i) + (1 - \alpha) \cdot \rho(i)$ ;

$DEQUEUE(inc\_queue_i, n)$ ; /\* Dequeue a value from  $inc\_queue_i$  into variable  $n$ . \*/

**if** ( $n = 1$ ) Decrement  $w_d(i)$ ;

**EndOn**

5. **Work Transfer:**

**On** (receiving a work request  $R(j, \mathcal{L}_d(j))$ ) **begin**

$\mathcal{L}_d(i) := \mathcal{L}(i) \ominus w_d(i)$ ; **if** ( $\mathcal{L}_d(i) \succ \mathcal{L}_d(j)$ ) Send work to  $j$  and update  $\mathcal{L}(i)$  accordingly;

**EndOn**

6. **Work Receipt:**

**On** (receiving work from  $j$ ) **begin**

Accept work and update  $\mathcal{L}(i)$  accordingly;

$num\_unserviced\_wk\_reqs := num\_unserviced\_wk\_reqs - 1$ ;

**EndOn**

**Until** (Parallel algorithm terminates)

**End** /\* Algorithm RANDOM\_SEEKING \*/

Fig. 2. Algorithm for the random seeking strategy

higher probability than those further away. A situation where non-uniform probability flinging will be useful is in a distributed setting where communication latencies between different pairs of processors may vary widely. In such a setting, for example, one pair of processors may be connected via a local area network (LAN), while another pair via a wide area network (WAN) [11]. Therefore, better performance in these situations will be obtained by having processors fling to nearer processors with higher probability, e.g., the probability of flinging between processors can be inversely varied with communication latencies between them.

### 3. Best-First Branch-and-Bound Preliminaries

In this section, we first briefly describe the sequential BFS algorithm, and then discuss a commonly used generic approach to its parallelization.

BFS performs a best-first search for a least-cost leaf node (representing an optimal solution) starting from a *root* node corresponding to the combinatorial optimization problem  $\mathcal{P}$  to be solved in a state-space tree  $\mathcal{T}$  [27]. Each node in  $\mathcal{T}$  represents a subproblem derived from  $\mathcal{P}$  and has a cost that is a lower bound on the cost of an optimal solution to that subproblem. The set of unexplored nodes is stored in an *OPEN* list. We denote by *best\_soln* the cost of the current best solution at any time during BFS’s operation. Due to its best-first search ordering, BFS explores only nodes with cost less than the optimal solution cost—these nodes are called *essential* nodes since they must be expanded by any search algorithm that guarantees optimality; all other nodes are termed *non-essential*.

BFS is most commonly parallelized on distributed-memory machines by first exploring the search space from the *root* node to generate a starting node for each of the processors, and then conducting sequential BFS on each of the processors from its starting node [5]. Processors broadcast any improvements in *best\_soln* which is maintained consistent across all processors and is used to prune all nodes with greater cost. Note that, unlike sequential BFS, in this parallel BFS algorithm, nodes are not processed in a global best-first manner, but in a local best-first manner.

We now define a few terms used in the sequel. The *OPEN* list of processor  $i$  is denoted by  $OPEN_i$ , and the number of nodes in it by  $M(i)$ . Let  $cost_i(l)$  denote the cost of the node at position  $l$  in a non-decreasing cost ordering of nodes in  $OPEN_i$ , and let  $cost_i(l) = \infty$ , for  $l > M(i)$ .

The above parallel BFS (PBFS) algorithm can be very scalable provided good load balancing methods are employed to address its following two inefficiencies [5]. (1) *Starvation*: This occurs when processors run out of work/nodes and idle. (2) *Non-essential work*: This occurs when processors processing nodes in non-global-best-first order expand non-essential nodes—this is in contrast to sequential BFS which processes nodes in global-best-first order and hence performs no non-essential work.

To tackle starvation, quantitative load balancing is needed to transfer nodes from busy processors to idle processors. Tackling non-essential work is less straightforward. Since the optimal solution cost is not known until algorithm termination, it is not known *a priori* which nodes are essential and which non-essential. Hence to address non-essential work, some type of qualitative load balancing is required to transfer good or low-cost nodes from processors that have them to those that have relatively bad nodes. The rationale being that by doing so, essential nodes



$\tau$  are parameters of RS and  $\lambda_d(i)$ ,  $\tau_d(i)$ ,  $\Delta\lambda_d(i)$ , and  $\Delta\tau_d(i)$  are integer variables explained below.

Initially,  $\Delta\lambda_d(i) = \Delta\tau_d(i) := 0$ , so that  $\tau_d(i) := \tau$  and  $\lambda_d(i) := \lambda$ . On issuing a probe, a processor  $i$  performs an *increment operation* on  $w_d(i)$  with probability  $p(i)$  given by Eq. 3 as:  $\Delta\lambda_d(i) := \Delta\lambda_d(i) - m$  and  $\Delta\tau_d(i) := \Delta\tau_d(i) - m$  (note that for qualitative load balancing, the “increment” operation is subtraction by  $m$ ). Hence the increment operation corresponds to the earlier stated fact that lead-node and threshold-node costs of processor  $i$  will become, after work transfer, the costs of nodes that were  $m$  positions after the original lead and threshold nodes. On performing an increment operation,  $i$  enqueues a 1 into  $inc\_queue_i$ . Subsequently, when a probe originating at  $i$  returns to it and  $i$  dequeues a value from  $inc\_queue_i$  and finds it to be 1, it performs a *decrement operation* on  $w_d(i)$  by incrementing both  $\Delta\lambda_d(i)$  and  $\Delta\tau_d(i)$  by  $m$ . Next, the relationship between any pair of processors  $i$  and  $j$  is determined as follows: (1)  $\mathcal{L}_d(i) \succ \mathcal{L}_d(j)$  (i.e.,  $i$ - $j$  have a source-sink relationship) iff  $cost_i(\tau_d(i)) < cost_j(\lambda_d(j))$ ; (2)  $\mathcal{L}_d(i) \prec \mathcal{L}_d(j)$  (i.e.,  $i$ - $j$  have a sink-source relationship) iff  $cost_j(\tau_d(j)) < cost_i(\lambda_d(i))$ ; and (3)  $\mathcal{L}_d(i) \approx \mathcal{L}_d(j)$  (i.e.,  $i$ - $j$  have a peer-peer relationship), otherwise. This completes the application of RS to parallel BFS.

Thus we see that RS strives to achieve qualitative load balance by probabilistically checking deterioration in lead-node costs of processors relative to the threshold-node costs of all other processors. Recall that  $cost_i(l) = \infty$  if  $i$  has less than  $l$  nodes. Therefore, in the above application of RS, if a processor has less than  $\lambda$  nodes, it will be a sink relative to all processors that have at least  $\tau$  nodes, and hence will be able to obtain nodes from such processors through the random seeking process (see Fig. 3(a)). Hence RS performs both dynamic quantitative and qualitative load balancing to minimize starvation and non-essential work.

#### 4.2. Pure Dynamic Quantitative Load Balancing

Here we modify the parallel BFS algorithm so that it requires only quantitative load balancing for good performance, and then apply RS to achieve this. The modification consists simply in initializing  $best\_soln$  in all processors to the cost of the optimal solution; this can be done by first solving the problem under consideration and finding out the optimal solution cost, and then rerunning the algorithm with the said initialization. As a result of the initialization, no non-essential nodes will be processed in any processor since such nodes will have cost equal to or greater than  $best\_soln$  and so will be pruned. Hence the total number of nodes processed will be constant and equal to the number of essential nodes, regardless of the order in which nodes are processed. Therefore, the only inefficiency to be addressed in parallel BFS is that of processor idling, for which we need only quantitative load balancing. This models other parallel branch-and-bound search methods like parallel depth-first search, and computations like parallel circuit simulation that require only quantitative load balancing. Below we describe how RS is applied for this purpose.

The application of RS to pure quantitative load balancing is similar to its application to combined quantitative and qualitative load balancing in Sec. 4.1. We use two parameters  $\lambda'$  and  $\tau'$ . The idea is to transfer work from a source processor  $j$  with  $\tau'$  or more nodes to a random sink processor  $i$  with less than  $\lambda'$

nodes. The work transferred consists of  $m'$  nodes from  $OPEN_j$ . Note that after the transfer, the number of nodes in  $j$  will reduce by  $m'$ . The above can be accomplished via RS as follows. For a processor  $i$ , we define  $\mathcal{L}(i) = M(i)$  (recall  $M(i) = |OPEN_i|$ ),  $w_d(i)$  as an integer variable, and  $\ominus$  as the “ $-$ ” (subtraction) operator. Initially,  $w_d(i) := 0$ , so that  $\mathcal{L}_d(i) := M(i)$ . On issuing a probe, a processor  $i$  performs the *increment operation* on  $w_d(i)$  with probability  $p(i)$  given by Eq. 3 as:  $w_d(i) := w_d(i) + m'$  which corresponds to  $i$  transferring  $m'$  nodes during a work transfer, since  $\mathcal{L}_d(i) := M(i) - w_d(i)$ . On performing an increment operation,  $i$  enqueues a 1 into  $inc\_queue_i$ . Subsequently, when a probe originating at  $i$  returns to it and  $i$  dequeues a value from  $inc\_queue_i$  and finds it to be 1, it performs a *decrement operation* on  $w_d(i)$  by decrementing  $w_d(i)$  by  $m'$ . Next, the relationship between any pair of processors  $i$  and  $j$  is determined as follows: (1)  $\mathcal{L}_d(i) \succ \mathcal{L}_d(j)$  (i.e.,  $i$ - $j$  have a source-sink relationship) iff  $\mathcal{L}_d(i) \geq \tau'$  and  $\mathcal{L}_d(j) < \lambda'$ ; (2)  $\mathcal{L}_d(i) \prec \mathcal{L}_d(j)$  (i.e.,  $i$ - $j$  have a sink-source relationship) iff  $\mathcal{L}_d(j) \geq \tau'$  and  $\mathcal{L}_d(i) < \lambda'$ ; and (3)  $\mathcal{L}_d(i) \approx \mathcal{L}_d(j)$  (i.e.,  $i$ - $j$  have a peer-peer relationship), otherwise. This completes the application of RS to quantitative load balancing.

Thus dynamic quantitative load balancing is achieved through RS by processors that have at least  $\tau'$  nodes randomly seeking those that have less than  $\lambda'$  nodes and transferring nodes to them. In the next section, we give performance results for RS for both combined dynamic quantitative and qualitative load balancing, and pure dynamic quantitative load balancing, and also compare it to the previous randomized scheme RC.

## 5. Performance Results

In this section, we present two sets of performance results: (1) comparing combined dynamic quantitative and qualitative load balancing capabilities of local RC (LRC) (in which generated nodes are transferred to random neighboring processors), global RC (GRC) (in which generated nodes are transferred to random processors), local random seeking (LRS) applied to quantitative and qualitative load balancing (LRSQl), and global random seeking (GRS) applied similarly (GRSQl) (Table 1(a)); and (2) comparing only dynamic quantitative load balancing capabilities of LRC, GRC, LRS applied to pure quantitative load balancing (LRSQn), and GRS applied similarly (GRSQn) (Table 1(b)). All results were collected on up to 512 processors of an nCUBE2 multicomputer. The test set consists of nine mixed-integer programming (MIP) problems from the MIPLIB benchmark set [3] with thousands of variables and constraints and a range of node-expansion (or node-processing) granularities and search-space sizes (which is equal to the number of essential nodes  $W$ ). A node expansion primarily involves solving a linear programming (LP) problem to compute a lower-bound cost for a node; for this we used the public domain simplex-based LP solver, “lp\_solve” [2].

From Table 1 we see that, as is to be expected, GRC performs better than its local counterpart LRC because of better load balance achieved through globalized work transfers and because the communication rate (of work transferred) is not high enough to cause excessive link contention or hot spots in GRC. Although GRS also performs better than LRS, the difference is not very noticeable, since unlike LRC, work transfer in LRS is not restricted to between neighboring processors due to the fact that probes flung from neighbor to neighbor in LRS may finally

Table 1. Comparison of the (a) combined dynamic quantitative and qualitative and (b) pure quantitative load balancing capabilities of different schemes for nine MIP problems corresponding to different number of essential nodes  $W$  and node-expansion times  $t_{exp}$  in terms of their execution time  $T_P$  on  $P$  processors.

Information on Problem and Number of Processors				Comb. Qnt. + Qlt. Load Bal. (All $T_P$ 's are w.r.t. $T_P(\text{GRSQl})$ )			Pure Qnt. Load Bal. (All $T_P$ 's are w.r.t. $T_P(\text{GRSQn})$ )		
Name	$W$	$t_{exp}$ (ms)	$P$	$T_P$ (LRC)	$T_P$ (GRC)	$T_P$ (LRSQl)	$T_P$ (LRC)	$T_P$ (GRC)	$T_P$ (LRSQn)
bell13a	112913	89	256	+15.18%	+11.20%	-0.45%	+8.10%	+5.76%	-1.26%
flugpl	49507	5	256	+16.32%	+9.23%	+0.91%	+9.67%	-2.49%	-3.72%
khb05250	10685	433	128	+25.49%	+11.86%	+0.08%	+16.00%	+9.12%	+5.07%
mod010	2987	6615	64	+23.85%	+22.20%	+0.80%	+22.65%	+31.27%	-3.75%
mod013	889	54	64	+31.98%	+24.74%	+18.84%	+36.82%	+20.07%	+5.17%
p0033	28083	7	512	+67.41%	+23.62%	+2.87%	+48.96%	+17.50%	+5.05%
pipex	1727	23	128	+37.74%	+19.64%	+2.66%	+43.73%	+24.86%	-0.46%
rgn	12323	118	256	+45.19%	+25.12%	+6.54%	+65.76%	+11.62%	-8.49%
stein46	219027	215	512	+45.85%	+7.86%	+1.35%	+38.92%	+3.39%	-2.08%

(a)

(b)

reach non-neighbor processors. Also, note that the execution times for LRC exceed those of LRS and GRS by 16–67% for combined quantitative and qualitative load balancing, and by 9–74% for only quantitative load balancing. The corresponding figures for GRC are 8–25% and 5–35%, except for the small granularity (i.e., small node-expansion time) problem `flugpl` in the latter case in which GRC has a 2% smaller execution time than GRSQn (but 1% larger execution time than LRSQn). The reason for the better relative performance of GRC for `flugpl` is that the time interval  $\Delta$  at which probes should be flung in GRS and LRS was not kept constant. Instead two probes were flung from a processor  $i$  after a node expansion each with probability  $p(i)$ , which means  $\Delta$  is one half the time for a single node expansion. This corresponds to a very small  $\Delta$  for `flugpl` leading to a higher flinging rate. As a result, the load balancing overhead in LRS and GRS for the very small granularity problem `flugpl` was high—about 16% of the execution time compared to about 8% of the execution time for LRC and GRC. Therefore the performance of LRS and GRS can be improved by a good margin by choosing an appropriate constant  $\Delta$ . The load balancing overhead can also be reduced by relaxing peer-peer relationships. However, even without these modifications, we see that the performance of our randomized schemes is much better than those of previous ones.

## 6. Conclusions

In this paper, we presented a completely general and informed randomized load balancing method called random seeking (RS) which we theoretically and empirically showed to be very efficient compared to previous randomized methods such as the well-known RC strategy. In future research, we will investigate the following interesting extensions to the RS strategy: (1) The first extension consists of adjusting probe flinging probabilities to different target processors based on their communication latencies from the flinging processor, and should be very useful in low bandwidth networks and in distributed settings. (2) The second extension is to consider flinging probes from processors with the objective of locating relative sources instead of relative sinks, or alternatively to locate both relative sources and relative sinks. (3) Finally, we can collect more specific information during random seeking regarding which processors or processor neighborhoods are likely to be sinks, and then fling probes to these locations with higher probability. This will help in locating sinks faster, and hence in correcting load imbalances faster.

## Acknowledgements

Sandia National Labs provided access to their 1024-processor nCUBE2 parallel computer.

## References

1. S. Anderson and M.C. Chen, "Parallel branch-and-bound algorithms on the hypercube," *Proc. Second Conference on Hypercube Multiprocessors*, pp. 309-317, 1987.
2. M. Berkelaar, "lp\_solve," from michel@es.ele.tue.nl, 1995.
3. R.E. Bixby, E.A. Boyd, and R.R. Indovina, "MIPLIB: A test set of real-world mixed-integer programming problems," *SIAM News*, Vol. 25, No. 2, pp. 16, 1992.
4. S. Dutt and N.R. Mahapatra, "Parallel A\* algorithms and their performance on hypercube multiprocessors," *Proc. Seventh International Parallel Processing Symposium (IPPS'93)*, pp. 797-803, Newport, CA, Apr. 13-16, 1993.
5. S. Dutt and N.R. Mahapatra, "Scalable load balancing strategies for parallel A\* algorithms," *Journal of Parallel and Distributed Computing*, Vol. 22, No. 3, pp. 488-505, Sep. 1994.
6. J. Eckstein, "Parallel branch-and-bound algorithms for general mixed integer-programming on the CM-5", *SIAM Journal on Optimization*, 1994.
7. C. Ferguson and R. Korf, "Distributed tree search and its application to alpha-beta pruning," *In Proc. 1988 National Conf. Artificial Intelligence*, Aug. 1988.
8. K.L. Hoffman and M. Padberg, "Solving airline crew scheduling problems by branch-and-cut," *Management Science*, Vol. 39, No. 6, pp. 657-682, June 1993.
9. S.-R. Huang and L.S. Davis, "Parallel iterative A\* search: An admissible distributed heuristic search algorithm," *Proc. Eleventh Int'l Joint Conf. on Artificial Intelligence*, pp. 23-29, 1989.
10. R.M. Karp and Y. Zhang, "A Randomized parallel branch-and-bound procedure," *J. of the ACM*, pp. 290-300, 1988.
11. G.K. Kudva and J.F. Pekny, "DCABB: A distributed control architecture for branch and bound calculations," *Computers and Chemical Engineering*, Vol. 19, No. 6/7, pp. 847-865, 1995.
12. V. Kumar, K. Ramesh and V.N. Rao, "Parallel best-first search of state-space graphs: A summary of results," *Proc. 1988 Nat'l Conf. Artificial Intell.*, 1988.
13. V. Kumar and V.N. Rao, "Load balancing on the hypercube architecture," *Proc. Hypercubes, Concurrent Comp., Appli.*, Mar. 1989.
14. V. Kumar, A. Grama, and V.N. Rao, "Scalable load balancing techniques for parallel computers," *Jour. of Par. and Distr. Computing*, Vol. 22, No. 1, pp. 60-79, 1994.
15. R. Luling and B. Monien, "Load balancing for distributed branch-and-bound algorithms," *Sixth Int'l Par. Proc. Symp.*, pp. 543-548, 1992.
16. N.R. Mahapatra and S. Dutt, "Scalable duplicate pruning strategies for parallel A\* graph search," *Proc. Fifth IEEE Symposium on Parallel and Distributed Processing (SPDP'93)*, pp. 290-297, Dallas, TX, Dec. 1-4, 1993.
17. N.R. Mahapatra and S. Dutt, "New anticipatory load balancing strategies for parallel A\* algorithms," *DIMACS Series in Discrete Mathematics and Theoretical Computer Science—Parallel Processing of Discrete Optimization Problems*, P.M. Pardalos, M.G. Resende, and K.G. Ramakrishnan (eds.), Vol. 22, pp. 197-232, 1995.
18. N.R. Mahapatra and S. Dutt, "Scalable global and local hashing strategies for du-

- licate pruning in parallel A\* graph search," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. 7, pp. 738-756, Jul. 1997.
19. N.R. Mahapatra and S. Dutt, "An efficient delay-optimal distributed termination detection algorithm," to appear in *J. of Par. and Distr. Computing*.
  20. N.R. Mahapatra and S. Dutt, "Random seeking: A general, efficient, and informed randomized scheme for dynamic load balancing," *Proc. 10th International Parallel Processing Symposium*, pp. 881-885, Honolulu, Hawaii, Apr. 15-19, 1996.
  21. S. Patil and P. Banerjee, "A parallel branch-and-bound algorithm for test generation," *IEEE Trans. Computer-Aided Design*, Vol. 9, pp. 313-322, Mar. 1990.
  22. J.F. Pekny, "Exact parallel algorithms for some members of the traveling salesman problem family," *Ph.D. Thesis*, Dept. of Chemical Engineering, Purdue University, 1989.
  23. A. Ranade, "Optimal speedup for backtrack search on a butterfly network," *Proc. Third Annual Symp. on Parallel Algorithms and Architectures*, pp. 40-48, 1991.
  24. V.N. Rao and V. Kumar, "Parallel depth-first search, part I: Implementation," *International Journal of Parallel Programming*, Vol. 16, No. 6, pp. 479-499, 1987.
  25. V.N. Rao, "Parallel processing of heuristic search," *Ph.D. Thesis*, University of Texas at Austin, July 1990.
  26. V.N. Rao and V. Kumar, "On the efficiency of parallel backtracking," Preprint 92-066, Army High-Performance Computing Research Center, University of Minnesota, Minneapolis, MN 55415.
  27. E. Rich, "Artificial Intelligence," *McGraw Hill*, New York, pp. 78-84, 1983.
  28. V.A. Saletore, "A distributed and adaptive dynamic load balancing scheme for parallel processing of medium-grain tasks," *Proc. Fifth Distributed Memory Computing Conference*, 1990.
  29. M.W.P. Savelsbergh and G.L. Nemhauser, "Functional description of MINTO, a Mixed INTegeR Optimizer," Report COC-91-03A, Georgia Institute of Technology.
  30. W. Shu and L.V. Kale, "A dynamic scheduling strategy for the chare-kernel system," *In Proc. Supercomputing 89*, pp. 389-398, 1989.
  31. A. Silberschatz and P.B. Galvin, *Addison-Wesley*, Reading, Massachusetts, pp. 138-140, 1994.