Chapter 1

COMBINATORIAL OPTIMIZATION

1.1 Search Algorithms for Discrete Optimization Problems

Search algorithms cab be used to solve discrete optimization problems (DOPs), a class of computationally expansive problems with significant theoretical and practical interest. Search algorithms solve DOPs by evaluating candidate solutions from a finite or countably infinite set of possible solutions to find one that satisfies a problem-specific criterion. DOPs are also referred to as combinatorial problems.

1.2 Definitions and Examples

A *discrete optimization problem* can be expressed as a tuple (S, f). The set S is a finite or countably infinite set of all solutions that satisfy specified constraints. This set is called the set of *feasible solutions*. The function f is the cost function that maps each element in set S onto the set of real numbers R.

$$F : S \to R$$

The objective of a DOP is to find a feasible solution x_{opt}, such that $f(x_{opt}) \leq f(x)$ for all $x \in S$.

Problems from various domains can be formulated as DOPs. Some examples are planning and scheduling, the optimal layout of VLSI chips, robot motion planning, test-pattern generation for digital circuits, and logistics and control.

1.2.1 Example The 0/1 Integer-Linear-Programming Problem

In the 0/1 integer-linear-programming problem, we are given an $m \times n$ matrix A, an $m \times 1$ vector b, and an $n \times 1$ vector c. The objective is to determine an $n \times 1$ vector $\overline{x}$ whose elements can take on only the value 0 or 1. The vector must satisfy the constraint

$$A\overline{x} \geq b$$

and the function

$$f(\overline{x}) = c^T\overline{x}$$

must be minimized. For this problem, the set S is the set of all values of the vector $\overline{x}$ that satisfy the equation $A\overline{x} \geq b$.

In most problems of practical interest, the solution set S is quite large. Consequently, it is not feasible to exhaustively enumerate the elements in S to determine the optimal element x_{opt}. Instead, a DOP can be reformulated as the problem of finding a minimum cost path in a graph from a designated initial node to one of several possible goal nodes. Each element x in S can be viewed as a path from the initial node to one of the goal nodes. There is a cost associated with each edge of the graph, and a cost function f is defined in terms of these edge costs. For many problems, the cost of a path is the sum of the edge costs. Such a graph is called a **state space**, and the nodes of the graph are called **states**. A **terminal node** is one that has no successors. All other nodes are called **nonterminal nodes**. The 8-puzzle problem can be naturally formulated as a graph search problem. In particular, the initial configuration is the initial node, and the final configuration is the goal node. The following example illustrates the process of reformulating the 0/1 Integer-linear-programming Problem as a graph search problem.

Example The 0/1 Integer-Linear-Programming. Consider an instance of the 0/1 integer-linear-programming problem defined in the previous example. Let the values of A, b, and c be given by

$$A = \begin{bmatrix} 5 & 2 & 1 & 2 \\ 1 & -1 & 2 & \\ 3 & 1 & 1 & 3 \end{bmatrix}, b \begin{bmatrix} 8 \\ 2 \\ 5 \end{bmatrix}, c = \begin{bmatrix} 2 \\ 1 \\ -1 \\ -2 \end{bmatrix}$$

The constraints corresponding to A, b, and c are as follows:

and the function $f(x)$ to be minimized is

$$f(x) = 2x_1 + x_2 - x_3 - 2x_4.$$

Each of the four elements of vector $\overline{x}$ can take the value 0 or 1. here are $2^4 = 16$ possible values for x. However, many of these values do not satisfy the problem's constraints.

The problem can be reformulated as a graph-search problem. The initial node represents the state in which none of the elements of vector x have have been assigned values. In this example, we assign to vector elements in subscript order; that is, first $x_{'}$, then x_2, and so on. The initial node generates two nodes corresponding to $x_1 = 0$ and $x_1 = 1$. After a variable x_i has been assigned a value, it is called a **fixed variable**. All variables that are not fixed are called **free variables**.

After instantiating a variable to 0 or 1, it is possible to check whether an instantiation of the remaining free variables can lead to a feasible solution. We do this by using the following condition:

$$\sum_{x_j \text{isfree}} \max\{A[i,j],0\} + \sum_{x_j \text{isfixed}} A[i,j]x_j \geq b_i, i = 1,\ldots,m$$

The left side of the equation is the maximum value of $\Sigma_{k=1}^{n} A[i,k]x_k$ that can be obtained by instantiating the free variables to either 0 or 1. If this value is grater than or equal to b_1, for $i = 1, 2, \ldots, m$, then the node may lead to a feasible solution.

For each of the nodes corresponding to $x_1 = 0$ and $x_1 = 1$, the next variable (x_2) is selected and assigned a value. The nodes are then checked for feasibility. This process continues until all the variables have been assigned and the feasible set has been generated. Figure 2 illustrates this process.

Function $f(x)$ is evaluated for each of the feasible solutions; the solution with the minimum value is the desired solution. Note that it is unnecessary to generate the entire feasible set to determine the solution. Several search algorithms can determine an optimal solution by searching only a portion of the graph.

For some problems, it is possible to estimate the cost to reach the goal state from an intermediate state. This cost is called a **heuristic estimate**. Let $h(x)$ denote the heuristic estimate of reaching the goal state from state x and $g(x)$ denote the cost of reaching state x from initial state s along the current path. The function h is called a **heuristic function**. If $h(x)$ is a lower bound on the cost of reaching the goal state from state x for all x, then h is called **admissible**. We define function $l(x)$ as the sum $h(x) + g(x)$. If h is admissible, then $l(x)$ is a lower bound on the cost of the path to a goal state that can be obtained by extending the current path between s and x. In subsequent examples we will see how an admissible heuristic can be used to determine the least-cost sequence of moves from the initial state to a goal state.

Once a DOP has been formulated as a graph search problem, it can be solved by algorithms such as branch-and-bound search and heuristic search. These technique use heuristic and the structure of the search space to solve DOPs without searching the set S exhaustively.

DOPs belong to the class of NP-hard problems. One may argue that it is pointless to apply parallel processing to these problems, since we can never reduce their worst-case run time to a polynomial without using exponentially many processors. However, the average-time complexity of heuristic search algorithms for many problems is polynomial. Furthermore, there are heuristic search algorithms that find suboptimal solutions for specific problems in polynomial time. In such cases, bigger problem instances can be solved using parallel computers. Many DOPs (such as, robot motion planning, speech understanding, and task scheduling) require real-time solutions. For these applications, parallel processing may be the only way to obtain acceptable performance. Other problems, for which optimal solutions are highly desirable, can be solved for moderate-sized instances in a reasonable amount of time by using parallel search techniques (for example, VLSI floor-plan optimization, and computer-aided design).

1.3 Sequential Search Algorithms

The most suitable sequential search algorithm to apply to a state space depends on whether the space forms a graph or a tree. In a tree, each new successor leads to an unexplored part of the space. An example of this is the 0/1 integer-programming problem. In a graph, however, a state can be reached along multiple [paths. An example of such a problem is the 8-puzzle. For such problems, whenever a state is generated, it is necessary to check if the state has already benn generated. If this check is not performed, then effectively the search graph is unfolded into a tree in which a state is repeated for every path that leads to it.

For many problems, unfolding increases the size of the search space by a small factor (for example, the 8-puzzle). For some problems, however, unfolded graphs are much larger than the original graphs. Figure 3(b) illustrates a graph whose corresponding tree has exponentially higher number of states. In this section, we present an overwiev of various sequential algorithms used to solve DOPs that are formulated as tree or graph search problems.

1.3.1 Depth-First Search Algorithms

Depth-first search (DFS) algorithms solve DOPs that can be formulated as tree-search problems. DFS begins by expanding the initial node and generating its successors. In each subsequent step, DFS expands one of the most recently generated nodes. If this node has no successors (or cannot lead to any solutions), then DFS backtracks and expands a different node. In some DFS algorithms, successors of a node are expanded in an order determined by their heuristic values. A major advantage of DFS is that its storage requirement is linear in the depth of the state space being searched. The following sections discuss three algorithms based on depth-first search.

Simple Backtracking

Simple backtracking is a depth-first search method that terminates upon finding the first solution. Thus, it is not guaranteed to find a minimum-cost solution. Simple backtracking uses no heuristic information to order the successors of an expanded node. A variant, **ordered backtracking**, does use heuristics to order the successors of an expanded node.

Depth-First Branch-and-Bound

Depth-first branch-and-bound (DFBB) exhaustively searches the state space; that is, it continues to search even after finding a solution path. Whenever it finds a new solution path, is updates the current best solution path. DFBB discards inferior partial solution paths (that is, partial solution paths whose extensions are guaranteed to be worse than the current best solution path). Upon termination, the current best solution is a globally optimal solution.

Iterative Deepening A^*

Trees corresponding to DOPs can be very deep. Thus, a DFS algorithm may get stuck searching a deep part of the search space when a solution exists higher up on another branch, For such trees, we impose a bound on the depth to which the DFS algorithm searches. If the node to be expanded is beyond the depth bound, then the node is not expanded and the algorithm backtracks. If a solution is not found, then the entire state space is searching again using a larger depth bound. This technique is called **iterative deepening depth-first search** (ID-DFS). Note that this method is guaranteed to find a solution path with the fewest edges. However, it is not guaranteed to find a least-cost path.

Iterative deepening A^* (IDA^*) is a variant of ID-DFS. IDA^* uses the l-values of nodes to bound depth (recall from previous section, that for node $x, l(x) = g(x) + h(x)$). IDA^* repeatedly performs cost-bounded DFS over the search space. In each iteration, IDA^* expands nodes depth-first. If the l-value of he node to be expanded is greater than the cost bound, then IDA^* backtracks. If a solution is not found within the current cost bound, then IDA^* repeats the entire depth-first search using a higher cost bound. In the first iteration the cost bound is set to the l-value of the initial state s. Note that since $g(s)$ is zero, $l(s)$ is equal to $h(s)$. In each subsequent iteration, the cost bound is increased. The new cost bound is equal to the minimum l-value of the nodes that were generated but could not be expanded in the previous iteration. The algorithm terminates when a goal node is expanded. IDA^* is guaranteed to find an optimal solution if the heuristic function is admissible. It may appear that IDA^* performs a lot of redundant work across iterations. However, for many problems the redundant work performed by IDA^* is minimal, because most of the work is done deep in the search space.

In each step of the DFS algorithm, untried alternatives must be stored. For example, in the 8-puzzle problem, up to three untried alternatives are stored at each step. In general, if m is the amount of storage required to store a state, and d is the maximum depth, then the total space requirement of the DFS algorithm is $\Theta(md)$. The state-space tree searched by parallel DFS can be efficiently represented as a stack. Since the depth of the stack increases linearly with the depth of the tree, the memory requirements of a stack representation are low.

There are two ways of storing untried alternatives using a stack. In the first representation, untried alternates are pushed on the stack at each step. The ancestors of a state are not represented on the stack. Figure 5(b) illustrates this representation for the three shown in Figure 5(a). In the second representation, shown in Figure 5(c), untried alternatives are stored along with their parent state. It is necessary to use the second representation if the sequence of transformations from the initial state to the goal state is required as a part of the solution. Furthermore, if the state space is a graph in which it is possible to generate an ancestor state by applying a sequence of transformations to the current state, then it is desirable to use the second representation, because it allows us to check for duplication of ancestor states and thus remove any cycles from the state-space graph. The second representation is useful for problems such as the 8-puzzle. In the previous example, using the second representation allows the algorithm to detect that nodes D and G should be discarded.

1.3.2 Best-First Search Algorithms

Best-first search (BFS) algorithms can search both graphs and trees. These algorithms use heuristics to direct the search to portions of the search space likely to yield solutions. Smaller heuristic values are assigned to more promising nodes. BFS maintains two lists: *open* and *closed*. At the beginning, the initial node is placed on the *open* list. This list is sorted according to a heuristic evaluation function that measures how likely each node is to yield a solution. In each of the search, the most promising node from the *open* list is removed. If this node is a goal node, then the algorithm terminates. Otherwise, the node is expanded. The expanded node is placed on the *closed* list. The successors of the newly expanded node are placed on the *open* list under one of the following circumstances: (1) the successor is not already on the *open* or *closed* lists, and (2) the successor is already on the *open* or *closed* list but has a lower heuristic value. In the second case, the node with the higher heuristic value is deleted.

A common BFS technique is the A^* *algorithm*. The A^* algorithm uses the lower bound function l as a heuristic evaluation function. Recall from of the first section that for each node x, $l(x)$ is the sum of $g(x)$ and $h(x)$. Nodes in the *open* list are ordered according to the value of the l function. At each step, the node with the smallest l-value (that is, the best node) is removed from the *open* list and expanded. Its successors are inserted into the *open* list at the proper positions and the node itself is inserted into the *closed* list. For an admissible heuristic function, A^* finds an optimal solution.

The main drawback of any BFS algorithm is that its memory requirement is linear in the size of the search space explored. For many problems, the size of the search space is exponential in the depth of the tree expanded. For problems with large search spaces, memory becomes a limitation.

1.4 Search Overhead Factor

Parallel search algorithms incur overhead from several sources. These include communication overhead, idle time due to load imbalance, and contention for shared data structures. Thus, if both the sequential and parallel formulations of an algorithm do the same amount of work, the speedup of parallel search on p processors is less than p. However, the amount of work done by a parallel formulation is often different from that done by the corresponding sequential formulation because they may explore different parts of the search space.

Let W be the amount of work done by a single processor, and $_p$ be the total amount of work done by p processors. The *search overhead factor* of the parallel system is defined as the ratio of the work done by the parallel formulation to that done by the sequential formulation, or W_p/W. Thus, the upper bound on speedup for the parallel system is given by $p \times (W/W_p)$. The actual speedup, however, may be less due to other parallel processing overhead. In most parallel search algorithms, the search overhead factor is greater than one. However, in some cases, it may be less than one, leading to super-linear speedup. If the search overhead factor is less than one on the average, then it indicates that the serial search algorithm is not the fastest algorithm for solving the problem.

To simplify our presentation and analysis, we assume that the time to expand each node is the same, and W and W_p are the number of nodes expanded by the serial and the parallel formulations, respectively. If the time for each expansion is t_c, then the sequential run time given by $T_S = t_c W$. In the remainder of the chapter, we assume that $t_c = 1$. Hence, the problem size W and the serial run time T_s become the same.

1.5 Parallel Depth-First Search

This section discusses parallel algorithms for depth-first search. To simplify the discussion, we focus on the case of simple backtracking. Parallel formulations of depth-first branch-and-bound and IDA^* are similar and are discussed in Section 7 and 8.

The critical issue in such algorithms is the distribution of the space among the processors. Consider the tree shown in Figure 7. Note that the left subtree (rooted at node A) can be searched in parallel with the right subtree (rooted at node B). By statically assigning a node in the tree to a processor, it is possible to expand the whole subtree rooted at that node without communicating with another processor. Thus, it seems that such a static allocation yields a good parallel search algorithm.

Let us see what happens if we try to apply this approach to the tree in Figure 7. Assume that we have two processors. The root node is expanded to generate two nodes (A and B), and each of these nodes is assigned to one of the processors. Each processor now searches the subtrees rooted at its assigned node independently. At this point, the problem with static node assignment becomes apparent. The processor exploring the subtree rooted at node A expands considerably fewer nodes than does the other processor. Due to this imbalance in the work load, one processor is idle for a significant amount of time, reducing efficiency. Using more processors worsens the imbalance.

Consider the partitioning of the tree for four processors. Nodes A and B are expanded to generate nodes C, D, E, and F. Assume that each of these nodes is assigned to one of the four processors. Now the processor searching the subtree rooted at node E does most of the work, and those searching the subtrees rooted at nodes C and D spend most of their time idle. The static partitioning of unstructured trees yields poor performance because of substantial variation in the size of partitions of the search space rooted at different nodes. Furthermore, since the search space is usually generated dynamically, it is difficult to get a good estimate of the size of the search space beforehand. Therefore, it is necessary to balance the search space among processors dynamically.

In *dynamic balancing*, when a processor runs of work, it gets more work from another processor that has work. Consider the two-processor partitioning of the tree in Figure 7(a). Assume that nodes A and B are assigned to the two processors as we just described. In this case when the processor searching the subtree rooted at node A runs out of work, it request work from the other processor. Although the dynamic distribution of work results i n communication overhead for work requests and work transfers, it reduces load imbalance among processors. This section explores several schemes for dynamically balancing the load between processors.

A parallel formulation of DFS based on dynamic load balancing is as follows. Each processor performs DFS on a disjoint part of the search space. After a processor finishes searching its part of the search space, it requests an unsearched part from other processors. Whenever any processor finds a goal node, all the processors terminate. If the search space is finite and has no solutions, then all the processors eventually run out of work, and the algorithm terminates.

Since each processor searches the state space depth-first, unexplored states can be conveniently stored as a stack. Each processor maintains its own local stack on which it executes DFS. When a processor's local stack is empty, it requests untried alternatives from another processor's stack. In the beginning, the entire search space is assigned to one processor, and other processors are assigned null search spaces (that is, empty stacks). The search space is distributed among the processors as they request work. We refer to the processor that sends work as the *donor* processor and to the processor that requests and receives work as the *recipient* processor.

As illustrated in Figure 8, each processor can be in one of two states: *active* (that is, it has work) or *idle* (that is, it is trying to get work). An idle processor select a donor processor and sends it a work request. If the idle processor receives work (part of the state space to be searched) from the donor processor, it becomes active. If it receives a *reject* message (because the and sends a work requests to that donor. This process repeats until the processor gets work or all the processors become idle. When a processor is idle and it receives a work request, that processor returns a *reject* message.

In the active state, a processor does a fixed amount of work (expands a fixed number of nodes) and then checks for pending work requests. When a work request is received, the processor partitions its work into two parts and sends one part to the requesting processor. When a processor has exhausted its own search space, it becomes idle. This process continues until a solution is found or until the entire space has been searched. If a solution is found, a message is broadcast to all processors to stop searching. A termination detection algorithm is used to detect whether all processors have become idle without finding a solution (Section 4.5).

1.6 Important Parameters of Parallel DFS

Two characteristics of parallel DFS are critical to determining its performance. First is the method for splitting work at a processor, and the second is the scheme to determine the donor processor when a processor becomes idle.

1.6.1 Work-Splitting Strategies

When work is transferred, the donor's stack is split into two stacks, one of which is sent to the recipient. In other words, some of the nodes (that is, alternatives) are removed from the donor's stack and added to the recipient's stack. If too little work is sent, the recipient quickly becomes idle; if too much, the donor becomes idle. Ideally, the stack is split into two equal pieces such that the size of the search space represented by each stack is the same. Such a split is called a *half-split*. It is difficult to get a good estimate of the size of the tree rooted at an unexpanded alternative in the stack. However, the alternatives near the bottom of the stack (that is, close to the initial node) tend to have bigger trees rooted at them, and alternatives near the top of the stack tend to have small trees rooted at them. To avoid sending very small amounts of work, nodes beyond a specified stack depth are not given away. This depth is called the *cutoff depth*.

Some possible strategies for splitting the search space are (1) send nodes near the bottom of the stack, (2) send nodes near the cutoff depth, and (3) send half the nodes between the bottom of the stack and the cutoff depth. The suitability of a splitting strategy depends on the nature of the search space. If the search space is uniform, both strategies 1 and 3 work well. If the search space is highly irregular, strategy 3 usually works well. If a strong heuristic is available (to order successors so that goal nodes move to the left of the state-space tree), strategy 2 is likely to perform better, since it tries to distribute those parts of the search space likely to contain a solution. The cost of splitting also becomes important if the stacks are deep. For such stack, strategy 1 has lower cost than strategies 2 and 3.

Figure 9 shows the partitioning of the DFS tree of Figure 5(a) into two subtrees using strategy 3. Note that the state beyond the cutoff depth are not partitioned. Figure 9 also shows the representation of the stack corresponding to the two subtrees. The stack representation used in the figure stores only the unexplored alternatives.

1.6.2 Load-Balancing Schemes

This section discusses three dynamic load-balancing schemes: asynchronous round robin, global round robin, and random polling.

Asynchronous Round Robin In asynchronous round robin (ARR), each processor maintains an independent variable, *target*. Whenever a processor runs out of work, it uses*target* as the label of a donor processor and sends it a work request. The value of the *target* is incremented (modulo p) each time a work request is sent. The initial value of *target* at each processor is set to ((*label*+ 1) m modulo p) where *label* is the local processor label. Note that work requests are generated independently by each processor. However, it is possible for two or more processors to request work from the same donor at nearly the same time.

Global Round Robin Global round robin (GRR) uses a single global variable called *target*, stored at processor O_0. Whenever a processor needs work, it requests and receives the value of *target* from P_0; then P_0 increments *target* (modulo p) before responding to another request. The recipient processor then s ends a request to the donor whose label is the value of *target* received from processor P_0. GRR ensures that successive work requests are distributed evenly over all processors. A drawback of this scheme is the contention at processor p_0 for access *target*.

Random Polling Random polling (RP) is the simplest load-balancing scheme. When a processor becomes idle, it randomly selects a donor. Each processor is selected as a donor with equal probability, ensuring that work requests are evenly distributed.

1.7 A General Framework for Analysis of Parallel DFS

To analyze the performance and scalability of parallel DFS algorithms for any load-balancing scheme, we must compute the overhead T_o

of the algorithm. Overhead in any load-balancing scheme is due to communication (requesting and sending work), idle time (waiting for work), termination detection, and contention for shared resources. If the search overhead factor is greater than one (that is, if parallel search does more work than serial search), this will add another term to T_o. In this section we assume that the search overhead factor is one. We analyze the case in which the search overhead factor is other than one in Section 6.1.

For the load-balancing schemes discussed in Section 4.1, idle time is subsumed by communication overhead due to work requests and transfers. When a processor becomes idle, it immediately selects a donor processor and sends it a work request. The total time that the processor remains idle is equal to the time for the request to reach the donor and for the reply to arrive. At that point, the idle processor either becomes busy or generates another work request. Therefore, the time spent in communication subsumes the time that a processor is idle. Since communication overhead is the dominant overhead in parallel DFS, we next consider a method to compute the communication overhead for each load-balancing scheme.

It is difficult to derive a precise expression for the communication overhead of the load-balancing schemes for DFS because they are dynamic. This section describes a technique that provides an upper bound on this overhead. We make the following assumptions in the analysis.

> (1) The work at any processor can be partitioned into independent pieces as long as its size exceeds a threshold ϵ. (2) A reasonable work-splitting mechanism is available. Assume that work ϖ at one processor is partitioned into two parts: $\psi\varpi$ and $(1-\psi)\varpi$ for $0 \leq \psi \leq 1$. Then there exists an arbitrarily small constant $\alpha(0 < \alpha \leq 0.5)$, such that $\psi\varpi > \alpha\varpi$ and$(1-\psi)\varpi > \alpha\varpi$. We call such a splitting mechanism $\alpha - splitting$. The constant α sets a lower bound on the load imbalance that results from splitting: both partitions of ϖ have at least $\alpha\varpi$ work. The first assumption is satisfied by most depth-first search algorithms. The third work-splitting strategy described in Section 4.1 results in α-splitting even for highly irregular search spaces.

In the load-balancing schemes to be analyzed, the total work is dynamically partitioned among the processors. Processors work on disjoint parts of the search space independently. An idle processor polls for work. When it finds a donor processor with work, the work is split into two pieces of size ϖ_j and ϖ_k, then assumption 2 states that there is a constant α such that $\varpi_j.\alpha\varpi_i$ and $\varpi_k.\alpha\varpi_i$. Note that α is less than 0.5. Therefore, after a work transfer, neither processor (donor and recipient) has more than $(1-\alpha)\varpi_i$ work. Suppose there are p pieces of work whose sizes are $\varpi_0, \varpi_1, \ldots, \varpi_{p-1}$. Assume that the size of the largest piece is ϖ. If all of these pieces are split, the splitting strategy yields $2p$ pieces of work whose sizes are given by $\psi_0\varpi_0, \psi_1\varpi_1, \ldots, \psi_{p-1}\varpi_{p-1}, (1-\psi_0)\varpi_0, (1-\psi_1)\varpi_1, \ldots, (1-\psi_{p-1})\varpi_{p-1}$. Among them, the size of the largest piece is given by $(1-\alpha)\varpi$. Assume that there are p processors and a single piece of work is assigned to each processor. If every processor receives a work request at least once, then each of these p pieces has been split at least once. Thus, the maximum work at any of the processors has been reduced by a factor of $(1-\alpha)$. We define $V(p)$ such that, after every $V(p)$ work request, each processor receives at least one work request. Note that $V(p) \geq p$. In general, $V(p)$ depends on the load-balancing algorithm. Initially, processor p_0 has W units of work, and all other processors have no work. After $V(p)$ requests, the maximum work remaining at any processor is less than $(1-\alpha)W$; after $2V(p)$ requests, the maximum work remaining at any processor is less than $(1-\alpha)^2W$. Similarly, after $(\log_{1/(1-\alpha)}(W/\epsilon))V(p)$ requests, the maximum work remaining at any processor is below a threshold value ϵ. Hence, the total number of work requests is $O(V(p)\log W)$. Communication overhead is caused by work requests and work transfers. /the total number of work transfers cannot exceed the total number of work requests. Therefore, the total number of work requests, weighted by the total communication cost of one work request and a corresponding work transfer, gives an upper bound on the total communication overhead. For simplicity, we assume the message size of the work request and work transfer