3rd SIAM Conf. Parallel Processing for Scientific Computing, Dec. 1987.

Extended Abstract

Introduction

Interest in parallel scientific computing has grown steadily in the past. This interest has increased considerably with the recent introduction of several commercial parallel processors. A large portion of that interest, however, has been concerned with parallel implementation of direct solution methods for linear algebra problems, differential equations and simulation techniques. There are many problems, particularly in the field of operations research, that have inefficient or no direct solution methods. Their solution methods involve the search for solution points in a large problem space. Examples include: the integer programming problem, the traveling salesman problem and the knapsack problem. An exhaustive search through the problem space to find a solution in large size search problems is usually impractical. Several techniques have been developed to improve the average search efficiency. The most general of these techniques is the branch-and-bound (BB) algorithm.

In this paper, we consider two parallel implementations of a BB algorithm for the 0–1 integer programming problem.

BB Algorithms

The BB algorithm can be best described as a partitioning algorithm. The algorithm partitions the original problem into smaller size problems and repeatedly partitions the smaller problems until a solution is found or infeasibility is determined. The state of the partitioning process can be represented as a rooted tree. The root of the tree represents the original problem. A node in the tree represents a partition of the original problem and is referred to as a *subproblem*. The set of subproblems that have been generated but not yet examined is referred to as the set of *active* subproblems. The partitioning process selects a single subproblem from the set of active subproblems and partitions it into yet smaller subproblems. The smaller subproblems are represented as the children of the node partitioned. This process of selecting subproblems and partitioning them constitutes the branching part of the algorithm. As the branching process proceeds, subproblems that cannot lead to an optimal solution are eliminated from further consideration from the partitioning process by the bounding part of the algorithm.

Hypercube Multiprocessors

A hypercube multiprocessor consists of $N = 2^n$ identical processors that are connected in an n-dimensional hypercube graph. Each processor has its own CPU and memory and a direct communication link or connection to each of n other processors in the graph. The hypercube topology for interconnecting the processors has several advantages over other topologies. It strikes a balance between the degree of each node in the graph and the communication diameter of the graph, making the topology unique. The hypercube topology is homogeneous in the sense that all the processors are identical and there are no special processors that have to be designated as the "borders" of the topology as is the case with many other regular topologies—two-dimensional grids, for example. In addition it is possible to divide a large hypercube array into smaller hypercube arrays, facilitating fault tolerance and multiprogramming. Finally, the hypercube topology can efficiently embed other regular topologies such as grids, trees and pyramids.

Several commercial parallel processors that are based on the hypercube topology have been introduced. Those include Intel's personal supercomputer, the iPSC, Ametek's Hypernet System/14, the NCUBE/ten from NCUBE and the Floating Point System T series computers. Our experiments were run on an NCUBE/ten with 64 processors.

The 0–1 Integer Programming Problem

The 0–1 integer programming problem is an optimization problem in which it is desired to maximize the value of an objective function $f(x_1, x_2, \ldots, x_n)$ subject to a set of constraints. The variables $(x_1, x_2, \ldots, x_n)$ are known as the decision variables and they are allowed to take only the values 0 or 1. The problem can be formulated as follows:

$$\begin{array}{lll} \text{Maximize} & f = \sum_{j=1}^{n} c_j x_j & \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j = b_i & i = 1, 2, \ldots, m \\ & x_j = 0 \ \ or \ \ 1 & j = 1, 2, \ldots, n \end{array}$$

A branch-and-bound algorithm referred to as *implicit enumeration* is used to solve the 0–1 integer programming problem. Optimal solutions for the problem can be found, at least in principle, by enumerating all 2^n possible solutions. However, the bounding property of the BB algorithm means that only a few of these solutions need to be enumerated explicitly, while the rest can be thought of as being enumerated implicitly— hence the name of the algorithm. The algorithm divides a problem into two smaller subproblems by assigning to a variable the value of 0 in one subproblem and 1 in the other. Each subproblem is then evaluated and a bounding value for all solutions that can be generated from each subproblem is computed. By selecting subproblems with higher bounds first and by eliminating subproblems that are known not to give better solutions than any already found, only a small number of the possible 2^n subproblems are evaluated to find the optimal solution.

The Parallel Algorithms

Two parallel algorithms are considered. In the first, referred to as Algorithm 1, a centralized list of subproblems is maintained in the host. The host sends a subproblem to a processor in the hypercube whenever that processor is free and there are subproblems on the queue. Each processor receives a subproblem, expands it and sends the results back to the host. Hence, the host acts as a manager that maintains global data structures. In our experiments, linear speedup is observed up to 16 processors. Beyond that point, the speedup curve starts to flatten and any further increase in the number of processors will result in little or no increase in the performance. This can be attributed to two factors: (1) the increased overhead of host-to-node communications relative to the amount of computation performed at each node. In our experiments, where the 0–1 integer programming problem is implemented, and the node computation time is very small compared to the overhead. When performing general integer programming the node computation time is much larger compare to the communication overhead and linear speedup is possible larger number of processors as we will show. (2) The size of the problem which at large cube sizes is not large enough to keep all the processors in the cube busy. Larger problem sizes will result in better speedups.

The second algorithm, referred to as Algorithm 2, distributes the data structure. The host generates as many subproblems as there are processors and sends a subproblem to each processor.

Each processor expands that subproblem and maintains its own list of subproblems. Hence, the global data structures is distributed. Due to the bounding property of the algorithm, some processors may run out of subproblems. Unless some load balancing mechanism is employed, the performance of the algorithm is very poor. We will demonstrate better results with a simple load balancing scheme.

Graphs illustrating the results obtained by running our experiments on an NCUBE/ten with 64 processors will be presented.