LARGE-SCALE 0-1 LINEAR PROGRAMMING ON DISTRIBUTED WORKSTATIONS

Timothy L. CANNON*

Digital Equipment Corporation, 3020 Hamaker Court, Fairfax, VA 22031, USA

and

Karla L. HOFFMAN☆

Department of Operations Research and Applied Statistics, George Mason University, 4400 University Drive, Fairfax, VA 22030, USA

Abstract

We present a methodology which uses a collection of workstations connected by an Ethernet network as a parallel processor for solving large-scale linear programming problems. On the largest problems we tested, linear and super-linear speedups have been achieved. Using the "branch-and-cut" approach of Hoffman, Padberg and Rinaldi, eight workstations connected in parallel solve problems from the test set documented in the Crowder, Johnson and Padberg 1983 *Operations Research* article. Very inexpensive, networked workstations are now solving in minutes problems which were once considered not solvable in economically feasible times. In this peer-to-peer (as opposed to master-worker) implementation, interprocess communication was accomplished by using shared files and resource locks. Effective communication between processes was accomplished with a minimum of overhead (never more than 8% of total processing time). The implementation procedures and computational results will be presented.

1. Introduction

Many combinatorial optimization problems are not solved, or not solved to optimality, because they are either too large or require too much computation time for existing sequential computers and algorithms to solve. It has been shown that a group of computers participating in the solution of a single large problem (i.e. performing parallel computation) could achieve much better results than one computer acting alone.

*Supported in part by a grant from the Digital Equipment Corporation.
☆Supported in part by grants from the Office of Naval Research and the National Science Foundation (ECS-8615438).

Furthermore, when parallel computation is achieved by connecting a number of independent small processors, problems that are "expensive" to solve on a large machine (e.g. "mainframe" or "supercomputer") may be solved at lower cost. The processors may be as small as existing computer workstations or personal computers and, while they may not achieve the equivalent performance found on larger computers, they do not require a substantial monetary investment.

The general class of combinatorial optimization problems to be discussed here is of the form of a general 0-1 decision problem:

$$\begin{aligned} &\underset{x}{\text{minimize}} \quad cx \\ &\text{subject to} \quad Ax \leqslant b; x \in \{0,1\}\,, \end{aligned} \qquad \text{Problem } Z_{IP}$$

where A is an $M \times N$ matrix, b is an M-vector and c is an N-vector. All data is assumed to be rational. We restrict our attention to the case where the integer variables can take on only the values zero or one. (Any bounded integer variable can be transformed into a set of zero-one variables.)

In this paper, we present a parallel implementation of a "branch-and-cut" algorithm (ABCOPT). This approach uses cutting planes based on the polyhedral structure of integer polytopes to tighten the linear programming relaxation of problem Z_{IP}. When it is not possible to generate any further cuts (due to our incomplete understanding of the polyhedral structure, or due to our inability *algorithmically* to generate cuts of a known form), we resort to branching on some fractional variable and then resume the generation of cuts for the corresponding linear programming problems associated with each branch.

Early work in integer cutting-plane methods was conducted by Dantzig, Fulkerson and Johnson [9], Gomory [20], and by Dantzig [8]. Although Gomory's algorithm and related research by Glover [17], Young [57], Balas [3] and Glover [18] have provided algorithms having proven finite convergence, all such algorithms have the following disadvantages (Wagner, Giglio and Glaser [53], Trauth and Wolsey [51]): (1) machine round-off errors may result in an incorrect optimal integer solution; (2) the solution of the problem remains infeasible until the optimal integer solution is obtained (meaning that no intermediate "good" solutions may be determined); (3) convergence is often too slow to allow large problems to be solved; and (4) sparse linear programs become continually more dense as the problem proceeds, forcing numerical problems and additional effort. Other disadvantages of traditional cutting planes in an algorithm that employs search trees are discussed in Padberg and Rinaldi [42].

To overcome the drawbacks of the cutting-plane methods cited above, the branch-and-cut approach employs "facial cuts". These cuts are "deep" in the sense that they cannot be pushed further into the feasible region without cutting off at

least one feasible integer point, and they belong to the class of inequalities that uniquely determines the polytope of the convex hull of all feasible integer points (see Padberg [39]).

The strengths of these cuts in assisting in the solution of zero-one programming problems are:

- the computational effort of identifying them *algorithmically* is often small;
- they are, in a mathematical sense, the "tightest" cuts possible in that they are facets of the convex hull of feasible integer points; and
- sparse, user-supplied constraints generate sparse facial cuts.

The topic of facial cuts will be addressed very briefly here; a complete treatment of polyhedral theory may be found in Nemhauser and Wolsey [38], Schrijver [47], Bachem and Grötschel [2], Rockafellar [45], Stoer and Witzgall [49], and Grünbaum [21].

In the next section of this paper, we will present a brief outline of the branch-and-cut approach. Section 3 will provide background on the previous implementations of parallel algorithms for combinatorial optimization problems. Section 4 will describe the architecture used for our parallel implementation, as well as the detailed description of the implementation. Finally, section 5 will present computational results and future research.

2. Solving zero-one integer programming problems using the branch-and-cut method

Hoffman, Padberg and Rinaldi have combined the branch-and-bound method with the generation of cutting planes based on the polyhedral structure of the integral polytope to develop a procedure known as "branch-and-cut". This general approach has been used to solve large-scale symmetric traveling salesman problems (Padberg and Rinaldi [41,42]) and general zero-one problems (Hoffman and Padberg [23]). The novelties in the branch-and-cut approach are that there is no requirement for special data structures related to node-specific cuts and that the proven solution is achieved in economically feasible times.

A complete description of how the sequential system called ABCOPT of Hoffman and Padberg [23] has been altered to allow a parallel implementation is found in Cannon [4]. To aid in understanding the parallel implementation of ABCOPT, the six main modules of the algorithm will be outlined below.

Reformulation procedure. Reformulation refers to elementary operations that can be performed automatically at any point in ABCOPT to improve or simplify a given formulation. The goal of these procedures is to obtain a solution to the linear programming relaxation of the zero-one problem which is much closer to the zero-one solution

and to remove redundant rows and columns from the matrix. A full description of the techniques is found in Hoffman and Padberg [23].

Linear program solver. The current version of the ABCOPT system uses a modified subset of the XMP software package for linear programming written by Roy Marsten of the University of Arizona (Marsten [34]).

Heuristic procedure. The heuristic procedure is guided by the linear-programming relaxation of the problem and, when successful, provides good upper bounds on the problem. A technical report which details the heuristic procedure is under preparation by Hoffman and Padberg.

Variable-fixing procedure. ABCOPT includes three methods of variable fixing: reduced-cost fixing, logical fixing, and optimality fixing. *Reduced-cost fixing* requires an analysis of the "gap" (i.e. the difference between the solution value of the linear programming relaxation to problem Z_{IP} and the best integer solution found so far). Any variable whose associated-linear programming reduced-cost at the root node of the tree is greater than the gap may be fixed permanently to the current bound values. *Logical fixing* is the process of determining that a specific variable must be fixed to either zero or one in *every* feasible solution to problem Z_{IP}. *Optimality fixing* examines a column and its associated profit value to determine when the fixing of such a variable can be done without detriment to any other variable.

Constraint (cut) generator. Once the linear-programming solution is determined, constraints based on the fractional value of that solution are automatically generated and extended (or "lifted") to include all zero-one variables of the problem. The cutting planes currently generated are extended minimal covers based on single knapsack constraints, extended minimal covers based on single knapsacks in conjunction with disjoint sets of special-ordered set constraints and the generation of disaggregated plant-location constraints. Whenever cuts are generated, the problem is augmented to include the newly generated constraints and then returned to the linear program module for re-solution. We note that all cuts generated are valid for the entire integer polytope by "lifting" any variable conditionally fixed within the branching tree. Thus, one can move freely among nodes of the branching tree without altering the data structures associated with the constraint set.

Branching procedure. The choice of the branching variable is made by evaluating the fractional variables on the basis of their largest objective function coefficients and their closeness to the value 0.5. The algorithm first chooses variables having very large objective function cost coefficients. The determination of such variables is done by statistically examining the set of cost coefficients for "outliers". If no outliers are fractional in the linear-programming solution, then one first finds the fractional variable closest to the value 0.5 and collects all fractional variables within some tolerance of this variable. The variable within this set having the largest normalized cost is chosen as the branching variable.

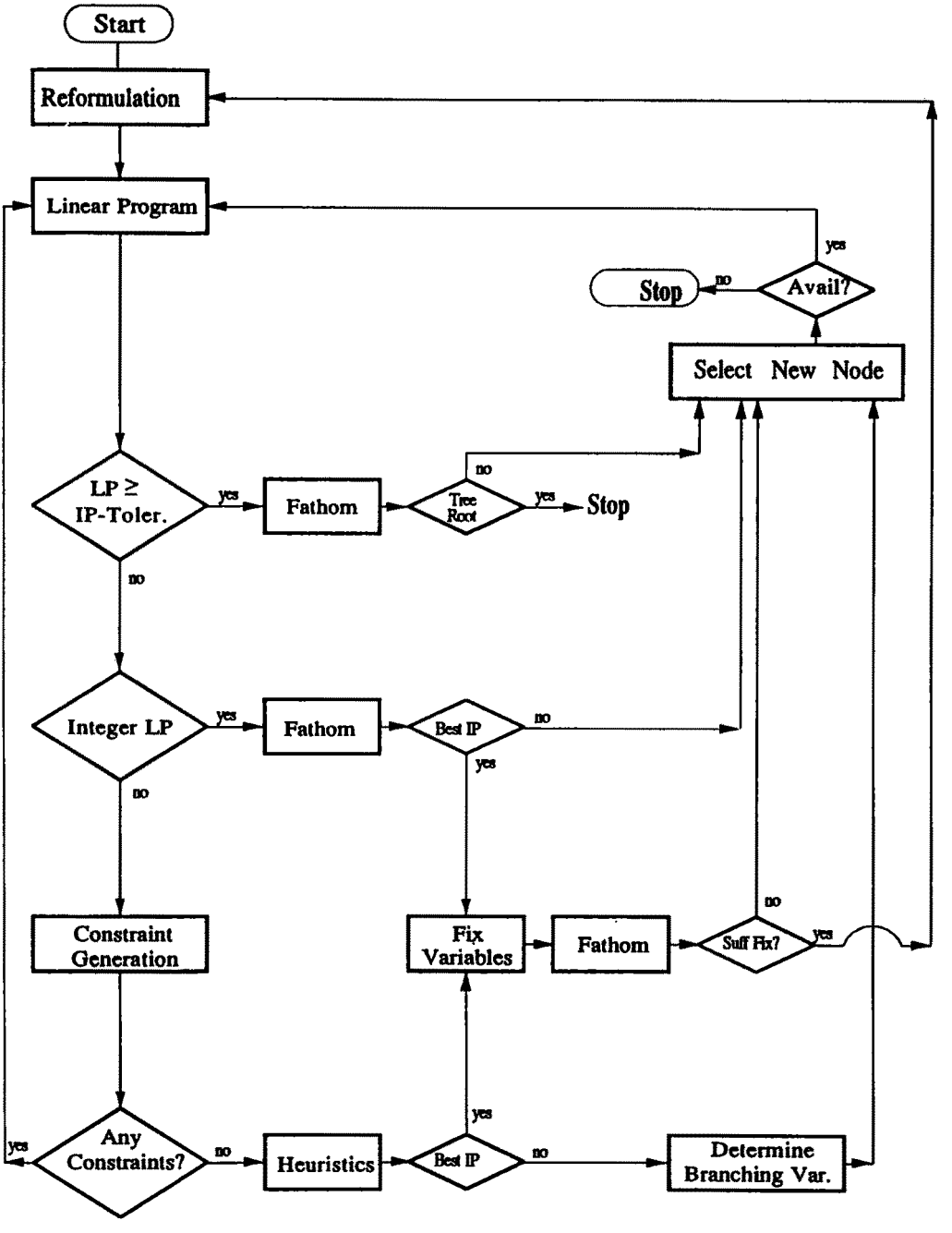

Fig. 1. Flow chart of the branch-and-cut system.

The results of the sequential branch-and-cut method are impressive: on a 48-city traveling salesman problem, a commercial branch-and-bound package required 40 minutes of CPU time (on a VAX 11/780 computer) to find an optimal solution. Padberg and Rinaldi's [42] branch-and-cut optimizer called TSPSOLVER took 25 seconds on the same machine. This traveling salesman procedure has solved the largest real-world symmetric problem – 2,392 cities – in 27 hours and 20 minutes of CPU time on a CYBER 205 computer. ABCOPT has solved problems from the test set of Crowder, Johnson and Padberg [7] with the largest problem (2,756 variables) having only ten nodes on the branching tree as compared with 2,392 nodes required by the Crowder, Johnson and Padberg algorithm. (MPSX/MIP in over 48 hours of dedicated computer time on an IBM 370 had not even found a *feasible* integer solution.) A brief flow-chart of the zero-one procedure is presented in fig. 1. Section 4 will describe the alterations made to this overall algorithm to allow a collection of machines to participate in the solution of a problem, but first we describe previous parallel branch-and-bound approaches and parallel architectures.

3. Review of parallel branch-and-bound algorithms for solving combinatorial optimization problems

This section will present an overview of related research in distributed processing of combinatorial optimization problems for the purpose of distinguishing our work. Several popular search strategies (A^*, AO^*, α–β, B^*, and SSS^*) have been shown to be special cases of the generalized branch-and-bound procedure (Nau, Kumar and Kanal [37]). Many researchers have proposed and investigated parallel implementations for several of these strategies. Cannon [4] presents a discussion of the branch-and-bound techniques which have been parallelized. Theoretical discussions of speedup and performance for parallel branch-and-bound algorithms may be found in Quinn and Deo [44], Lai and Sprague [30], Imai, Yoshida and Fukumura [25], Li and Wah [31–33]. Kindervater and Lenstra [27] present an excellent tutorial introduction to the literature on parallel computers and algorithms that is relevant for combinatorial optimization.

While it might seem that most problems could be solved faster (in terms of elapsed time) in a parallel-processing environment than in a sequential environment, it generally is not the case. Offset against the gains in potential parallelism is the overhead of creating, communicating with, and synchronizing additional processes. Sometimes, the additional impact of parallelism may warrant major changes in an algorithm when it is decomposed to run in a parallel-processing environment. For example, it may be worthwhile to perform a set of calculations on each of the processors rather than to pay a penalty to access the results on a single processor. Even though some processors may perform some work previously done by another processor, the decrease in communication cost may more than offset the redundant effort.

We define *speedup* as E_1/E_n, where E_1 is the elapsed ("wall-clock") time of an algorithm running on one processor and E_n is the elapsed time of the same algorithm running on n of the same processors. *Efficiency* or *utilization* of an algorithm running on n processors is the speedup divided by n.

Simulations (Imai, Yoshida and Fukumura [25], Imai, Fukumura and Yoshida [24], Li and Wah [31–33]. Lai and Sahni [29], and Mohan [35]) and experimental results (Wah and Ma [54,55]) have shown the effects of parallelizing branch-and-bound algorithms by expanding several nodes simultaneously. While one would expect an n-fold speedup when n processors are used (as compared with the speed for one processor), it has been shown that it is possible to experience one of three conditions:

- *Detrimental anomaly.* A detrimental anomaly occurs when the number of iterations for n processors is more than the number of iterations of the *best* serial algorithm.
- *Acceleration anomaly.* This anomaly occurs when the number of iterations for n processors is less than $1/n$ of the number of iterations of the best serial algorithm.
- *Deceleration anomaly.* A deceleration anomaly occurs when the number of iterations for n processors is less than the number of iterations of the best serial algorithm, but more than $1/n$ of the number of iterations of the best serial algorithm.

Note that in the above defintions, all comparisons are made between an algorithm on n processors and a best serial algorithm.

Much of the early research into parallel branch-and-bound algorithms concentrated on depth-first and beadth-first search because of the memory limitations of available computers. In a recently published historical note (Pruul, Nemhauser and Rushmeier [43]), Pruul showed that using a depth-first approach, the simultaneous exploration of nodes results in finding better solutions earlier, which in turn resulted in earlier fathoming and a significant reduction of the number of nodes examined. Imai, Fukumura and Yoshida [24], Imai, Yoshida and Fukumura [25], El-Dessouki and Huen [13], Finkel and Manber [14], and DeWitt, Finkel and Solomon [11], each using depth-first algorithms, showed that if the number of processors was appropriately chosen, speedups approached or sometimes exceeded the number of machines used (see Wah and Ma [54,55]; see also Wah, Li and Yu [56] for similar testing using best-first approaches).

Mohan [35,36] and Trienekens [52] showed that marked improvements could be accomplished by asynchronous rather than synchronous implementations due to the ability of each processor to work continuously. See also de Bruin, Rinnooy Kan and Trienkens [10] for a related report on a simulation tool for performance evaluation of parallel branch-and-bound algorithms.

Lai and Sahni [29] and Li and Wah [31, 32] studied the likelihood of branch-and-bound algorithms exhibiting detrimental, deceleration and acceleration anomalies. Li and Wah showed through theoretical analysis and simulation that deceleration anomalies were infrequently encountered. Although Lai and Sahni claimed that near-linear speedup for parallel branch-and-bound algorithms with best-first search could be expected for only a "small" ($\leqslant 16$) number of processors, Li and Wah showed that near-linear speedup may hold for a "large" (1000–2000) number of processors. Li and Wah also showed that a best-first branch-and-bound algorithm with dominance tests will never yield detrimental anomalies if (1) the method of selecting the next node for expansion is unambiguous, (2) approximations are not allowed, and (3) dominance relations exist and are consistent with the node-selection criteria. They also showed that acceleration anomalies could occur when (1) either a breadth-first or depth-first search was used, (2) some nodes have identical lower bounds, (3) the dominance relation is inconsistent with the node-selection functions, (4) multiple lists of subproblems are used, or (5) a suboptimal solution is sought.

We draw the following conclusions from the above research: Breadth-first asynchronous approaches to the branch-and-bound method which immediately broadcast bounding information to all processors is likely to exhibit acceleration anomalies on a large class of integer-programming problems. We have chosen to implement a best-node strategy instead of a breadth-first strategy since in our branch-and-cut environment, we expect our bounding procedures (both the cutting plane and heuristic algorithms) to limit significantly the number of nodes one will need to investigate. We have avoided the normal master-worker configuration so as to avoid the overhead of an additional processor whose major function it is to coordinate the efforts of other processors. Our implementation performs the tasks of node selection, bounding and fathoming in parallel, and relies extensively on data exchange between the processors to help better define the overall problem.

The features which distinguish this implementation from other methods are:

- The use of a local area network and distributed workstations as the model for computation.
- Peer-to-peer communication and control as opposed to a master-worker relationship.
- The use of a best-first approach based upon the value of the "tightened" linear-programming relaxation of the problem.
- Maintenance of candidates in a sorted list available to all processors. When a processor has completed a task (either fathoming the node or branching), the best candidate on the active-node list is chosen for examination.
- A change in the upper bound is communicated to all processors and is available to each processor immediately.
- Entrance to the system by a processor may occur at any time; the algorithm is adaptive.

- Implementation of a graceful shutdown phase so that all processors know when the problem has been solved to optimality.
- Ability to "pause" a node when it appears that it may not lead to an optimal solution and to "resume" processing on it at a later time. This resumption of processing may be performed by a processor other than the one that "paused" it.
- Maintenance of a pool of facial cuts from which individual processors review and select cuts prior to generating new cuts. Cutting planes generated by a processor are added (without duplication) to this pool of cuts so as to be accessible to all participating processors.
- The use of cutting planes based on the facial structure of integer polytopes to significantly tighten the lower bound.
- The use of a heuristic algorithm at each node. If the heuristic finds a new best-upper-bound, this information is communicated to all other processors immediately, even if there is additional work to perform at the node at which the bound was found.

Only the first eight topics will be addressed below. The algorithmic extensions required to implement a pool of facial cuts and the dramatic improvements that are realized will be addressed in Cannon and Hoffman [5]; for an introduction to the concept of a constraint pool, see Padberg and Rinaldi [42].

4. Implementation on distributed workstations

We have chosen to implement the extensions to ABCOPT using a collection of commercial computer workstations connected by a local area network. Major shortcomings of previous parallel-processing approaches to combinatorial optimization have been the inability to share data effectively and the inability to efficiently notify other processors about the status of computation. Both drawbacks have been overcome by our approach; data is freely shared among processors and all processors are notified immediately whenever a new pausing point or upper bound is found.

The extensions to ABCOPT which were required for distributed processing are divided into three major categories:

- interprocess communication to share critical information,
- candidate list sharing, and
- member synchronization to provide for graceful initiation and shutdown.

Each of these categories will be addressed in this section after the foundation for the decision to use a Local Area Network (LAN) architecture is established. A description of our implementation will then be presented.

4.1. WHY AN LAN ARCHITECTURE?

One of the major goals of this research was to design and implement a system for combinatorial optimization which uses readily available, relatively inexpensive technology. To that end, we have chosen an Ethernet LAN architecture as a foundation because of its increasing commercial acceptance and reasonable price. The claims made for an LAN system are similar to those made for multiprocessors with advantages typically expressed in terms such as: high-performing, available, and reliable (Ajmone Marsan, Balbo and Conte [1], and Gehringer, Siewiorek and Segall [16]). Moreover, LAN systems have the additional advantage of being separable and extremely flexible.

Because the speed of communication among the processors is substantially slower in LANs than in highly-coupled multiprocessors, distributed processing using LANs generally can be effective only if:

- The activity that is to be performed can be decomposed into smaller tasks that can be executed in parallel.
- The tasks can be conveniently allocated to processors so as to minimize the system overhead devoted to interprocessor cooperation.
- The system is designed modularly so that the addition of new elements is possible and cost effective.

An advantage in using LAN systems is the ability to quickly reconfigure the system under software control. Because each of the processors is connected to a local network, the software layers that define and communicate with all processors may be instructed to include, or to ignore, selective processors. For other optimization research using LANs, see Schnabel [46], Chang et al. [6], and Finkel and Manber [14].

4.2. THE ENVIRONMENT

4.2.1. Hardware

Our system is composed of eight Digital Equipment Corporation VAXstation 2000 systems and one MicroVAX II system, operating not in master-worker relationship (where a designated processor controls actions of the other processors), but in a true peer-to-peer relationship (where processors share the responsibility for coordinating activities). The MicroVAX II system acts merely as a file server and is called the "boot node". The interconnection network is a combination of DECnet (Digital's communication protocol operating on Ethernet) and Digital's Local Area VAXcluster software. The operating system is VAX/VMS version 4.7.

4.2.2. Software

All processors are logically joined together to form a Local Area VAXcluster (LAVC). An LAVC is a distributed system made up of computers and their associated

storage elements, all linked in a closely coupled arrangement. We will explain the LAVC concept briefly to serve as background for understanding our implementation. Cannon [4] fully describes the method of implementation for this work. A complete technical description of an LAVC may be found in Kronenberg et al. [28], Duffy [12], Fox and Ywoskus [15], and Goldstein [19].

An LAVC differs from a more tightly-coupled multiprocessor arrangement in several ways. First, the workstations communicate over a network link instead of sharing memory. Second, each processor has its own copy of the operating system in memory. Third, the members of the cluster may boot up and shut down independently. Finally, the services offered by the VAXcluster are more closely aligned with those offered by a traditional single timesharing system than with the capabilities offered by traditional networks (e.g. a VAXcluster environment includes common batch and print queues, system-wide synchronization, and a common operating system).

When a satellite member is powered on, a copy of the operating system and other necessary software is loaded over Ethernet from the central disks of the boot member. Once the satellite member joins the cluster as a member, all the resources and data are as accessible as if they were attached to the satellite system. Paging and swapping activity of the satellite node is conducted with local disks.

The collection of workstations working together to solve a single combinatorial problem will be called the **System.** An individual workstation will be called either a **member** or a **processor.**

4.3. INTERPROCESS COMMUNICATION

One of the shortcomings in other implementations of parallel search procedures has been the lack of efficient interprocess communication. In our implementation, we have accomplished efficient and effective interprocess communication by using the VAX Distributed Lock Manager and Blocking Asynchronous System Traps. While the lock manager normally is used to provide transparent, synchronized data access by members of an LAVC, we have used it as a means of passing messages between members of the system. Blocking asynchronous system traps are almost always used to support local buffer caching and have seldom, if ever, been used for interprocess communication. Used in conjunction with lock status blocks, we have found it to be not only very efficient, but a very effective means of interprocess communication. The critical information that is being shared among processors is the upper bound, ZSTAR, and the point at which a node should be paused, TARGET.

4.3.1. Resources, locks and lock value blocks

A *resource* can be any entity within the system (e.g. files, data structures, databases, and executable routines). Each resource in a cluster is represented by a

unique abstract name that is agreed upon by all the cooperating processes. This name is entered into a distributed global namespace which is maintained by the distributed lock manager. The lock management services allow processors to associate a name with a resource and to request access to that resource. Lock modes enable processes to indicate how they want to share access with other processes.

To use the lock management services, a process must request access to a resource (request a lock). There are three required arguments for the request of a new lock:

- *A resource name.* The lock management services use the resource name to look for other lock requests that use the same name.
- *The lock mode to be associated with the requested lock.* The lock mode indicates how the process wants to share the resource with other processes.
- *The address of a lock status block.* The lock status block receives the completion status for a lock request and the lock identification. The lock identification is used to refer to a lock request once it has been queued.

The lock management services compare the lock mode of the newly-requested lock to the lock modes of other locks with the same resource name. The lock manager resolves lock requests in the following manner: (1) If no other process has a lock on the resource, the new lock is granted; (2) If another process has a lock on the resource and the mode of the new request is compatible with the existing lock, the new lock is granted; and (3) If another process already has a lock on the resource and the mode of the new request is not compatible with the lock mode of the existing lock, the new request is placed in a queue where it waits until the resource becomes available.

The lock manager allows callers to specify one of six degrees of compatibility, ranging from no access to exclusive access. Once the lock is granted, the owning process can request a lock conversion to change the lock mode.

Lock conversions allow processes to change the level of locks. For example, a process can maintain a low-level lock on a resource until it wants to limit access to the resource. The process can then request a lock conversion to a higher-level lock. If the requested lock mode is compatible with the currently granted locks, the conversion request is granted immediately. If the requested lock mode is incompatible with the existing locks in the granted queue, the request is placed on the conversion queue.

When a process no longer needs a lock on a resource, the lock can be dequeued. When the last lock on a resource is dequeued, the lock management services delete the name of the resource from its data structures.

We use a lock value block as the primary means for passing critical, time-sensitive information to other members of our system. Used in conjunction with Blocking ASTs (described in the next section), we *immediately* notify other members in the system about changes in global information (e.g. the best integer answer, number of idle processors, and node cut-off point).

A lock value block is an optional 16-byte value block that functions as a small piece of global memory which is updated automatically by the operating system. The first time a process associates a lock value block with a particular resource, the lock management services create a resource lock value block for that resource. The resource lock value block is maintained by the lock management services until there are no more locks on the resource.

When a process sets the appropriate flag in a lock (or conversion) request and the lock (or conversion) request is granted, the contents of the resource lock value block are written to the process' lock value block. When a process sets the appropriate flag on a conversion from specific high-level lock modes to a lower mode, the contents of the process' lock value block are stored in the resource lock value block.

4.3.2. Blocking asynchronous system traps

An asynchronous system trap (AST) is a system service (using a combination of hardware and software interrupts) which allows a process to request that it be interrupted to perform a particular task when a specific event occurs. When the interrupt is received, control is passed to a separate procedure which is executed immediately in the context of the receiver's process. Because execution of the AST procedure occurs without respect for the process' point of execution, it is called asynchronous.

One of the services that the distributed lock manager provides is a notification mechanism whereby a process that has been granted a lock on a resource can be notified when another process has queued an incompatible lock request. The mechanism, known as a *blocking AST,* is at the heart of our interprocess communication implementation.

In our implementation, all processes establish compatible high-level locks with blocking ASTs specified on four distinct resources. Whenever a processor has new information to share, it initiates a request for an incompatible lock on the appropriate resource. Because blocking ASTs were specified, each other member will be interrupted so that new information can be obtained.

Each member begins processing by placing a Protected Read (PR)-mode lock, with a blocking AST specified, on a desired resource. When a member has a better value to share with the other processors, it places a lock conversion request for an Exclusive (EX)-mode lock on the resource. Because each member in our system has placed its lock specifying that it wants to be notified if another processor has an incompatible request (a blocking AST), each member *immediately* stops what it is doing, releases its lock on the resource and requeues a PR-mode lock request on the same resource.

When the EX-mode lock is granted to the member requesting the lock, that member supplies the new value to be shared and converts the lock back to a PR mode. As soon as the EX-mode lock is converted to a PR-mode lock, all members waiting

for a PR-mode lock on that resource are granted the lock along with access to the lock value block. As soon as each lock is granted, the lock value block is read and the value in the block is available to each processor immediately. (The lock value block may be read simultaneously by all members of the system.)

In the following discussion, the notation **LOCK** will be used to denote the resource, while *LOCK* will be used to denote the value associated with the lock.

4.3.3. The ZSTAR lock

The **ZSTAR** lock is used to notify processors of a new upper bound. Without the immediate sharing of this bound, members could be evaluating nodes long after the nodes could have been fathomed. The first member to enter the system places an EX-mode lock on **ZSTAR**, updates the lock value block with its best integer answer value, *ZSTAR* (if none is known, $ZSTAR = +\infty$), and converts the lock back to a PR mode. From that point on, as each new member enters the system and places a PR-mode lock on **ZSTAR** specifying a blocking AST, *ZSTAR* is available immediately by reading the lock status block.

When a new best integer answer is found by either the heuristic procedure or as a linear-programming solution at some node in the tree, a lock conversion request for an EX-mode lock is immediately placed for **ZSTAR**. Again, because each member has been granted a PR-mode lock on **ZSTAR**, each member is interrupted, immediately releases its lock on **ZSTAR**, and requeues a PR-mode lock request for **ZSTAR**. When the PR-mode lock is granted, the new *ZSTAR* is found in the lock status block.

At each point in the original source code where *ZSTAR* could have an effect on the flow of the algorithm, a check has been incorporated to compare the value of the most recent linear-programming objective function value with *ZSTAR*. It should be noted that these checks are only to support the distributed processing environment; they are unnecessary in the sequential case since a new integer answer can only be determined in very specific places. If a member determines that its problem can no longer lead to an optimal solution, it will fathom the node immediately.

4.3.4. The TARGET lock

We have implemented an approach to suspend the work associated with a node if it appears that that node may be going past an optimal solution. The method depends on the values of the system-wide lowest linear-programming objective function (z_{LP}) and *ZSTAR* and is calculated as follows:

$$TARGET = z_{LP} - (PERC * (z_{LP} - ZSTAR)).$$

If *TARGET* falls within (*ZSTAR*/(1 + *BETA*) of *ZSTAR*, then *TARGET* is set equal to *ZSTAR*. (*PERC* and *BETA* are user-supplied values.)

The mechanism of **TARGET** is identical to that of **ZSTAR**. Each time a member selects a new candidate or finds a better integer solution, it determines *TARGET*. If a member finds that its *TARGET* is now better than the one all the members know about, a request is placed to convert the PR-mode lock on **TARGET** to an EX mode. Each member immediately stops working, drops its lock on **TARGET**, queues a PR-mode request for **TARGET**, then receives *TARGET* from the lock status block when the lock request is granted. *TARGET* is critical in guiding the search; early suspension of unattractive nodes can lead to substantial savings in processing time.

Because *TARGET* is determined for the system based upon the largest system-wide difference between the linear-programming objective function and *ZSTAR*, *TARGET* must necessarily be dynamic. Any time *ZSTAR* changes, *TARGET* can change. Any time the member investigating that candidate with the lowest linear-programming objective function value disposes of the node (either by fathoming or branching), the value of *TARGET* may change. In this manner, *TARGET* is adaptive to the problem structure at hand.

4.4. SHARING DATA THROUGH DISK FILES

A distinguishing feature of this implementation is the ability for a member to obtain a "most promising" candidate (i.e. new node to develop) without waiting for any other processors to complete their tasks. By using shared files, members are able to insert candidates into the candidate list and to remove candidates from the list independently.

This task of retrieving and providing new nodes is accomplished by using a shared indexed file, called the *Candidate List*. The files are accessed using the Indexed Sequential Access Method (ISAM). Once a record has been accessed by an indexed-read request, sequential-read requests may then be used to retrieve records with ascending key field values, beginning with the key field value in the record retrieved by the initial read request. A record is automatically inserted into a file (whenever a WRITE command is issued) based upon its index value. VAX FORTRAN allows a single file to be shared by a number of processors.

To avoid resource contention and to allow for system shutdown, specialized extensions were implemented. Resource contention occurs when two processes each try to access the same resource at the same time. In our case, contention will be experienced when two or more processes try to simultaneously access the same record in a file. System shutdown considerations include output file creation, synchronizing file closure, and final reporting. The VAX Record Management System was used to accomplish record locking and to diagnose contention.

When a member wishes to insert candidates into the file, two write requests are initiated. The first request is for the candidate with $x_j = 1$, the second is for the candidate with $x_j = 0$. The second of these two requests is enacted almost immediately

after the first because only the sign on the index field is changed (reversed) and the record is rewritten to the file.

Because both candidate records are being written using the same key (the value of the last linear-programming objective function), duplicates have been explicitly allowed.

Because we are using a best-first search approach, the Candidate List is accessed on the basis of the lowest reference objective function. When a record is read, the record is locked automatically by the VAX Record Management Services, preventing access to that record by any other process. After a successful read operation, the record is deleted from the file. By deleting a candidate record immediately after reading it, and by having the record locked while it is being read, we can ensure that each member receives a unique subproblem to solve. If an unsuccessful read operation occurs, it is for one of two reasons: the requested record is being read by another processor or the end of the file has been reached. If the record is being read by another processor, the read request is initiated again after a delay of approximately 0.5 seconds (to allow the other processor enough time to complete the read operation and to then delete the record). If the end of the file is reached, indicating that no more subproblems are available, the member is placed in *hibernation* until awakened. Hibernation is the act of a member making itself inactive but remaining known to the system so that it can be interrupted (e.g. by an AST).

We note in closing that similar shared files are used to handle constraint information. The details of this aspect of the implementation can be found in Cannon and Hoffman [5].

4.5. MEMBER SYNCHRONIZATION

Members of the system are synchronized by using two locks: **MEMBERS** and **IDLE**. This synchronization provides the ability for each member to understand the status of the system and to be able to gracefully exit when required.

Upon startup, each member places a PR-mode lock on **MEMBERS**. One of the available pieces of information about locks is the total number of locks granted for a resource. Therefore, the total number of locks on **MEMBERS** is equivalent to the number of processes in the current system. The first member to place a lock on **MEMBERS** is responsible for creating the Candidate List file and the Constraint Pool file. Subsequent processes which are granted a lock on **MEMBERS** do not have any "managerial" responsibility.

The same first member is also responsible for populating the Candidate List with entries. The procedure adopted for this testing is to have the first member determine five initial variables on which to branch. Using these five variables, a five-level search tree is created, immediately providing 32 candidates for the Candidate List. This procedure, referred to later as a *parallel start*, was designed to ensure that sufficient nodes were available early in the process.

When a member finds that the Candidate List is empty, it will place a PR-mode lock on **IDLE**, specifying a blocking AST. By comparing the number of locks on **IDLE** with the number of locks on **MEMBERS**, the member will know if processing is complete. That is, when the number of locks on both resources is equal, all members are idle. If the number of locks on **IDLE** is less than the number on **MEMBERS**, the member notes that its "idle position" is equal to the number of locks, and then hibernates. Its idle position is used by the member to know when it should again begin processing.

The purpose of hibernation and an idle position is to provide a mechanism by which a member may be notified that it should return to being an active member of the system. Each time a member places a candidate in the Candidate List, it checks to see if it can be granted an immediate lock (i.e. granted without causing any other lock to convert) on **IDLE**. If it is granted an immediate lock, then no other members are idle. If it is not granted the lock immediately (an indication of another process holding a high-level lock on **IDLE**), it queues a PW-mode lock request for **IDLE**, followed by an immediate release of the lock. The queuing of a PW-mode lock will awaken each hibernating member (because their locks were placed with a blocking AST specified). Each awakened member will decrement its idle position by one. The member with an idle position of zero will immediately dequeue its lock, select a candidate from the Candidate List and resume processing. All other awakened members will return to a state of hibernation, but with an idle position of one less than before being awakened. If there was only one lock on **IDLE**, the resource will disappear when the lock is dequeued. However, when the next idle member is granted a lock on **IDLE**, the resource will be created and the idle count will again be equal to one.

The last member to be granted a lock on **IDLE** (i.e. when the **IDLE** count equals the **MEMBERS** count) issues a request for an EX-mode lock on **MEMBERS**. The blocking AST routine associated with **MEMBERS** initiates the shutdown procedure.

The shutdown procedure creates a report for the members and releases all locks the member holds. Each member produces a report detailing its elapsed time, its CPU time, and the number and source of identified duplicate constraints. In addition, the member that found the best answer (i.e. the one who held the last lock on **ZSTAR**) reports the final answer. Each member maintains detailed information of all its processing on a hard disk file local to that processor.

4.6. GENERAL COMMENTS ON PARALLEL IMPLEMENTATION

It should be noted from the above discussion that there is no preconceived order for member startup. In fact, any processor may begin, and others may join at any time without regard for the history of the other members, thus providing a peer-to-peer relationship among processors.

While in the current implementation we have not provided a means for a processor to leave of its own volition (i.e. when the workstation owner wants to begin

processing again), we could easily implement a mechanism that provides for notification when the user wants to regain control of his system. All that need be done when notification occurs is to have the processor stop its current computation and insert the current candidate in the Candidate List at the last linear-programming objective function. This termination procedure would mirror that of "pausing" a node for *TARGET* considerations. The user interrupt needed to accomplish this transition is estimated to require approximately four seconds. After the user need was serviced, the user could again "kick off" the optimizer without any detrimental effects to the system.

5. Computational results

In this section, we will present and discuss our computational results. Throughout this discussion, the term *Original Method* refers to the sequential computer code called ABCOPT, while the term *Extended Method* refers to the modifications made to the Original Method to support distributed processing.

On some of the small problems, the Extended Method was not significantly faster than the Original Method due to the relatively small size of the branching tree. On the larger problems, however, significant reduction in elapsed time was achieved by using the Extended Method. The Constraint Pool was responsible for a marked improvement in elapsed time on most of the problems when compared with the Original Method. Because the smaller problems were solved so quickly, gains due to the Constraint Pool were not realized. Another significant result is the fact that the Extended Method running on a single processor exhibited a pronounced improvement in run-time performance on large problems when compared with the Original Method.

The test bed (hereafter called the CJP test set) for the Extended Method contains seven of the ten problems described in Crowder, Johnson and Padberg [7] and is presented in table 1. (The other three problems were not available to us.) The test set is a collection of industry-formulated problems, three of which arose as planning applications (Johnson, Kostreva and Suhl [26]). Of the seven problems presented, only six are applicable to distributed processing; Problem P0548 is not because the Original Method solves the problem to optimality without generating a single branching node. In all tables below, **ABCOPT** denotes the performance of ABCOPT.

For the purpose of our work, the definitions of anomalies presented in section 3 will be changed slightly to better describe our environment. A *detrimental anomaly* occurs when $E_n > E_1$, where E_n denotes elapsed time when n processors are used. Another characteristic indicating a detrimental anomaly is a speedup of less than one. A *deceleration anomaly* occurs when $E_1/n < E_n < E_1$. A speedup between one and n indicates a deceleration anomaly. An *acceleration anomaly* occurs when $E_n < E_1/n$. A speedup in excess of n characterizes an acceleration anomaly and is also called

superlinear speedup. The expected behavior is that n processors will take E_1/n to complete (*linear speedup*).

Table 1

CJP Test Set with Performance Times for the Original and Extended Methods

				Original Method		Extended Method	
Name	Variables	Rows	CJP	Nodes	Elapsed (min)	Nodes	Elapsed (min)
P0033	33	16	113	8	1.53	39	1.60
P0040	40	24	11	6	0.57	32	1.03
P0201	201	134	1116	346	462.36	336	45.07
P0282	282	242	1862	8	12.86	36	5.77
P0291	291	253	87	4	1.07	32	1.55
P2756	2756	756	2392	10	648.85	137	154.58

CJP: The number of nodes as developed by Crowder, Johnson and Padberg after preprocessing and embedded constraint generation. Once no additional cuts could be generated, the augmented problem was passed to MPSX/MIP370 for re-solving.

Orig. Method: The Original Method running on one VAXstation 2000.

Ext. Method: The Extended Method running on eight VAXstation 2000s.

5.1. RUN-TIME PERFORMANCE

We have realized a significant reduction in solution times for large problems by using the Extended Method. Table 1 compares the solution times for the Original Method with those of the Extended Method running on eight processors.

All timing measurements were made after the top of the search tree had been determined. Unless specifically noted otherwise, all references to performance times relate to elapsed ("wall-clock") time. We have chosen elapsed time as a performance measure because we are interested in the amount of time taken to report the optimal solution to the user. It should be remembered that the computational model is a number of networked single-user workstations dedicated to solving these problems. Under these conditions, elapsed time is a good measurement of performance.

It should be noted that a change was made to the Original Method to facilitate a parallel-processing environment which involves the generation of nodes on the initial search tree. In the Original Method, only one new variable is generated to start the search tree. In the Extended Method, 32 nodes are generated in order to provide ample nodes to keep all processors busy. To generate 32 nodes, five vandals are selected by the branching procedure described in section 2 and a five-level balanced binary tree is created. This method was followed in all cases to provide a common benchmark.

Empirical observations indicate that generally four times the number of processors is an adequate starting point for a distributed-processing environment. A comparison of the number of nodes on the search tree is presented in table 2.

Table 2

Number of nodes on the search tree

	Number of processors								
Name	1	2	3	4	5	6	7	8	ABCOPT
P0033	39	37	36	37	39	34	36	39	8
P0040	32	32	32	32	32	32	32	32	6
P0201	307	310	316	328	320	345	349	336	346
P0282	36	36	36	36	36	36	36	36	12
P0291	32	32	32	32	32	32	32	32	4
P2756	93	90	101	89	111	115	117	137	16

The elapsed time, CPU time, speedup, and efficiency of the test set may be found in tables 3 through 6, respectively. Figures 2 through 10 present the same findings graphically. Problem P0201 has been solved with linear and, in some cases, superlinear speedup. (An efficiency range of 0.99 to 1.01 is considered linear speedup due to the inaccuracies of run-time measurements imposed by the extensive time sampling and reporting required for the testing phases of this research. Because the elapsed time is defined as the time between the starting of the first processor and the completion of the processor which found the best integer answer, the actual total system run-time may differ slightly from the reported time.) Problem P0201 is solved using the Extended Method and eight processors in 0.75 hours versus 7.7 hours for the Original Method. Using the Extended Method and eight processors, Problem P2756 is solved in 2.6 hours, while the Original Method requires 10.8 hours for solution.

In contrast to Problem P0201, however, Problem P2756 takes more time to solve using the Extended Method running on one processor and using the "parallel start". These differences will be discussed below. For problems which solve quickly (under five minutes), detrimental anomalies were both expected and observed.

Table 3

Elapsed Time of the CJP Test Set (minutes on VAXstation 2000)

		Number of processors in the system								
Name	ABCOPT	1	2	3	4	5	6	7	8	ABCOPT
P0033	1.53	1.92	1.23	1.30	1.12	1.52	1.47	1.53	1.60	1.53
P0040	0.57	0.81	0.63	0.62	0.68	0.83	0.94	1.14	1.03	0.57
P0201	462.36	370.97	184.97	123.60	96.58	75.28	61.86	52.00	45.07	462.36
P0282	12.86	6.40	5.17	5.43	5.51	6.75	6.19	6.01	5.77	12.86
P0291	1.07	1.57	1.14	1.07	1.05	1.12	1.19	1.26	1.55	1.07
P2756	648.85	872.58	365.54	290.58	185.16	191.87	171.54	154.36	154.58	648.85

Table 4

Total CPU Time of the CJP Test Set (minutes on VAXstation 2000)

		Number of processors in the system								
Name	ABCOPT	1	2	3	4	5	6	7	8	ABCOPT
P0033	1.05	1.35	1.32	0.93	1.63	2.19	1.76	1.91	2.31	1.05
P0040	0.47	0.38	0.41	0.47	0.53	0.61	0.67	0.74	0.80	0.47
P0201	453.31	318.49	343.91	341.09	349.07	336.46	330.44	321.71	313.15	453.31
P0282	12.11	5.69	5.87	6.67	6.83	8.13	7.82	7.78	7.94	12.11
P0291	0.94	1.05	1.28	1.54	1.91	2.26	2.48	2.78	2.81	0.94
P2756	631.63	844.41	683.90	808.80	648.66	695.47	762.91	937.02	1077.01	631.63

Table 5

Speedup of the CJP Test Set

	Number of processors in the system						
Name	2	3	4	5	6	7	8
P0033	1.56	1.48	1.71	1.26	1.31	1.26	1.20
P0040	1.29	1.31	1.19	0.98	0.86	0.71	0.79
P0201	2.01	3.00	3.84	4.93	6.00	7.14	8.23
P0282	1.24	1.18	1.16	0.95	1.04	1.07	0.90
P0291	1.38	1.47	1.50	1.40	1.32	1.25	1.04
P2756	2.39	3.00	4.71	4.55	5.09	5.65	5.65

Table 6

Efficiency of the CJP Test Set

	Number of processors in the system						
Name	2	3	4	5	6	7	8
P0033	0.78	0.49	0.43	0.25	0.22	0.18	0.15
P0040	0.64	0.44	0.30	0.20	0.14	0.10	0.10
P0201	1.00	1.00	0.96	0.99	1.00	1.02	1.03
P0282	0.62	0.39	0.29	0.19	0.17	0.15	0.11
P0291	0.69	0.49	0.37	0.28	0.22	0.18	0.13
P2756	1.19	1.00	1.18	0.91	0.85	0.81	0.71

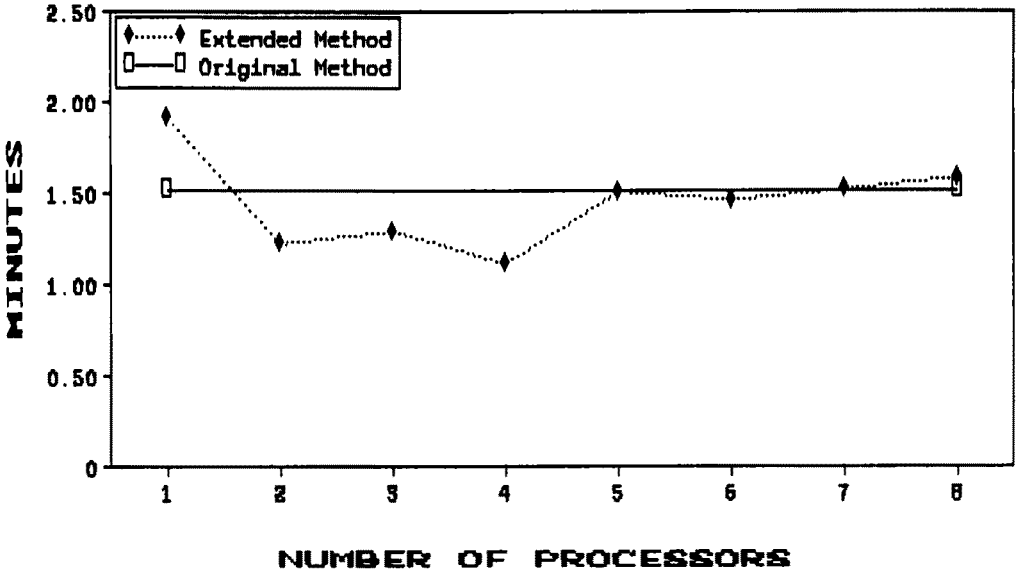

Fig. 2. Elapsed Time for Problem P0033.

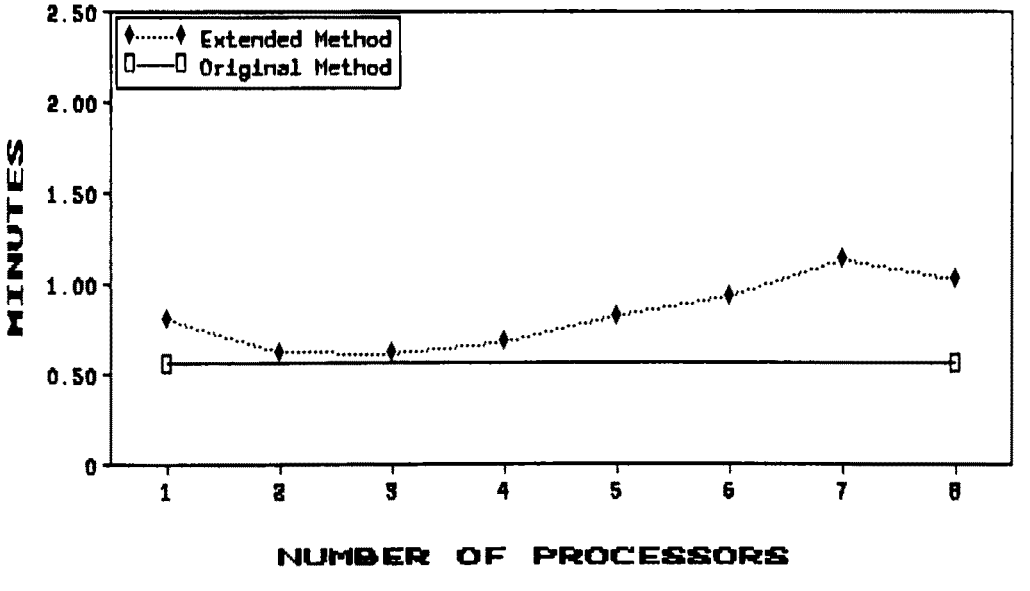

Fig. 3. Elapsed Time for Problem P0040.

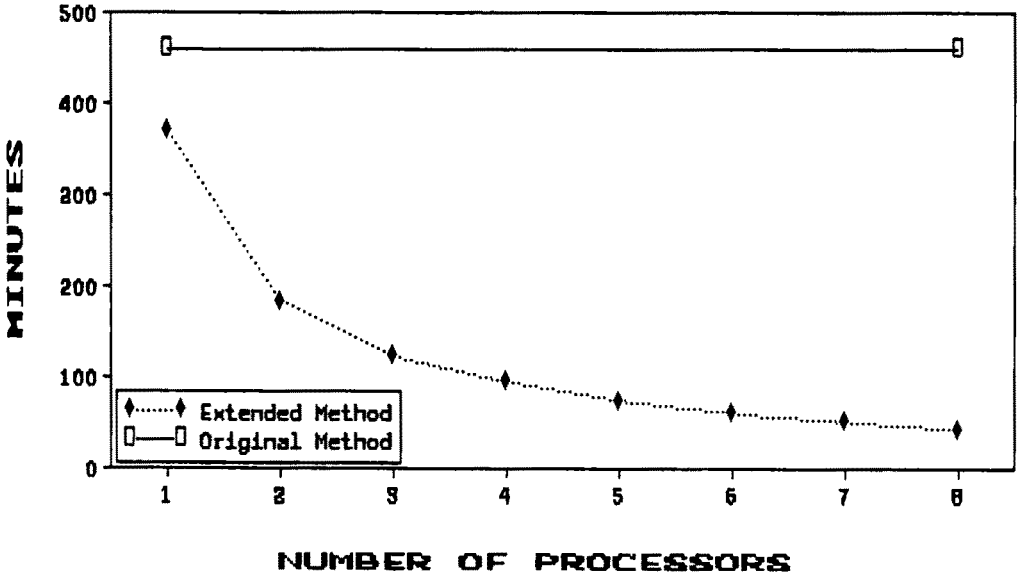

Fig. 4. Elapsed Time for Problem P0201.

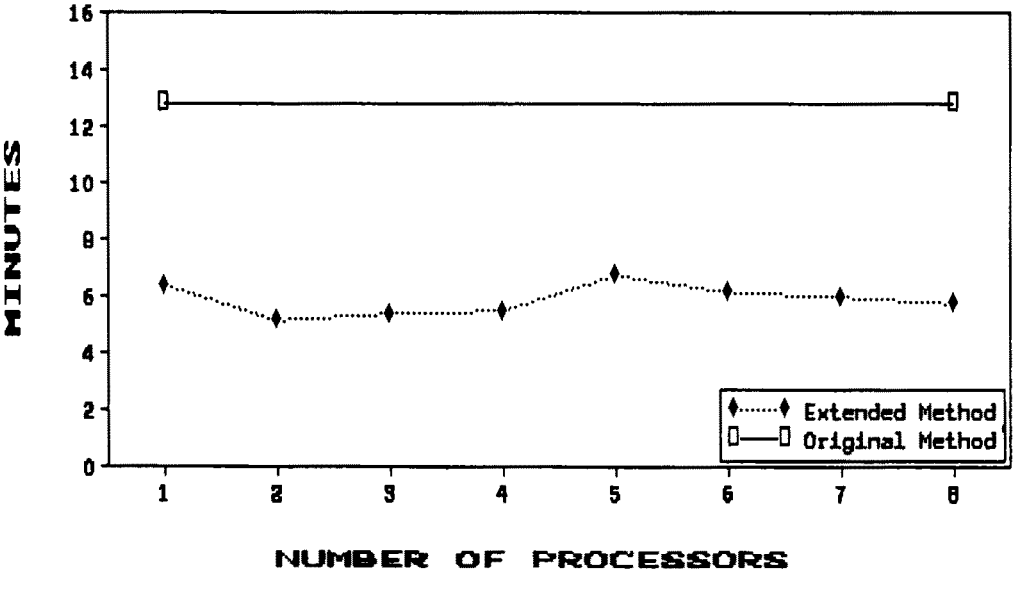

Fig. 5. Elapsed Time for Problem P0282.

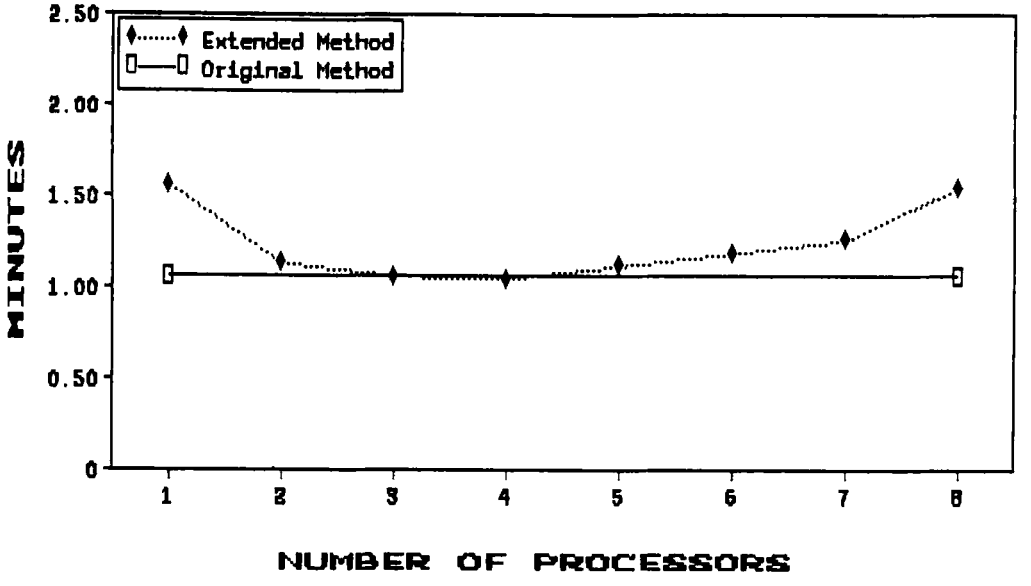

Fig. 6. Elapsed Time for Problem P0291.

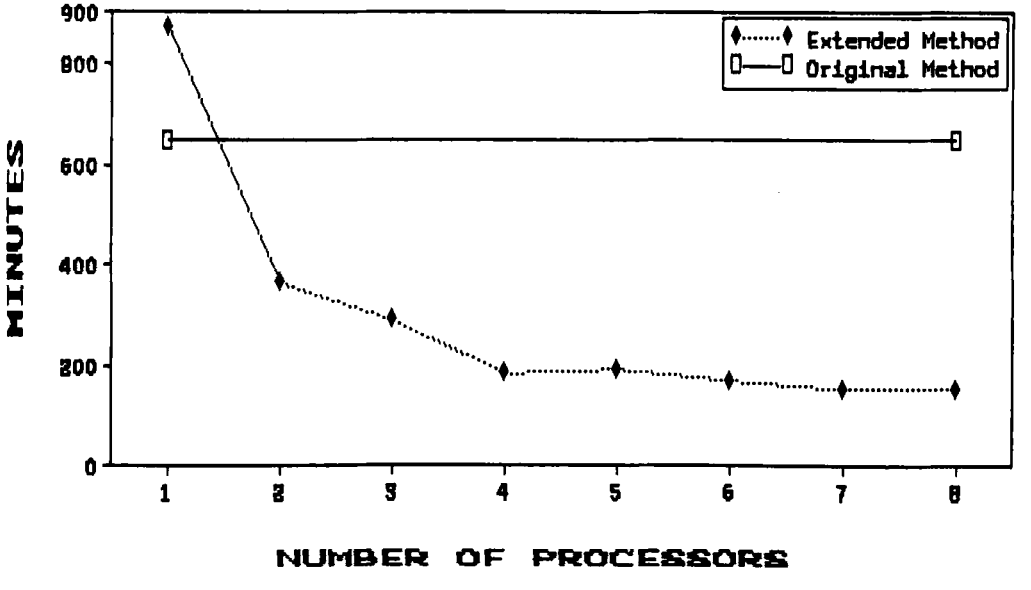

Fig. 7. Elapsed Time for Problem P2756.

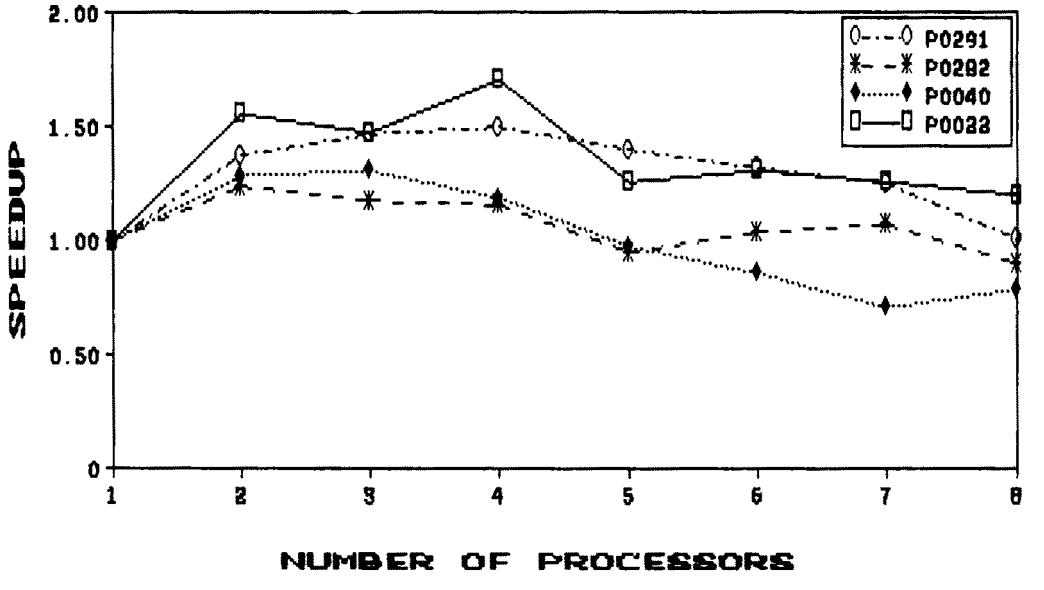

Fig. 8. Speedup of Problems P0033, P0040, P00282 and P0291

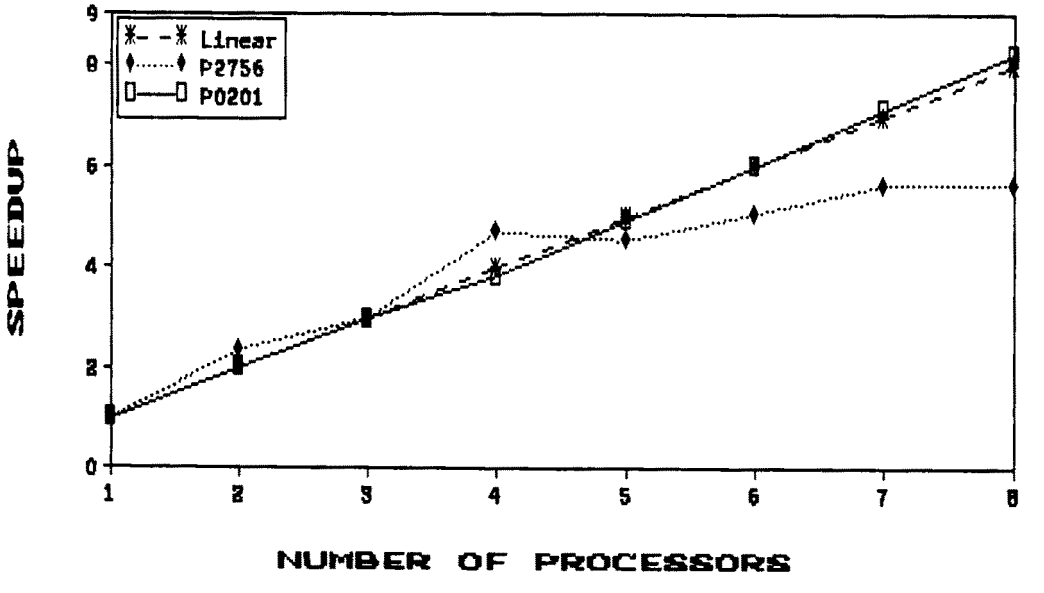

Fig. 9. Speedup of Problems P0201 and P2756.

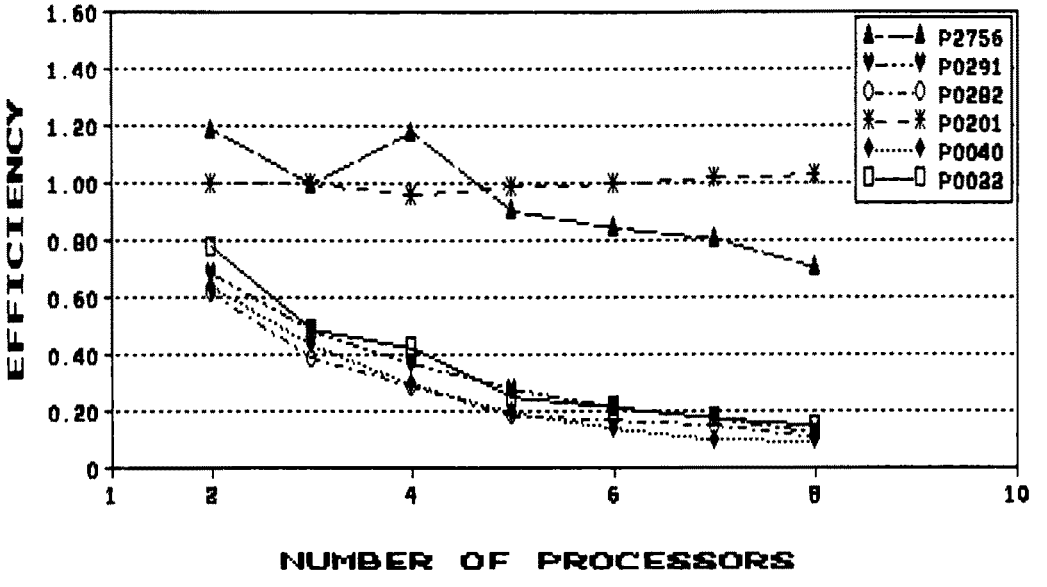

Fig. 10. Efficiency of the CJP Test Set.

5.2. ANOMALIES

Problems P0201 and P2756 have been solved *much* faster with the Extended Method than with the Original Method (with the exception of solving P2756 using a single processor in the Extended Method). In a distributed-processing environment, solutions to these problems were generally achieved with linear speedup while in a few cases, superlinear speedup was achieved. This performance is attributed to three factors: the effectiveness of the Constraint Pool, the large number of branching nodes, and the effectiveness of the heuristic procedures for obtaining bounds.

A large number of branching nodes (see table 2) means that there are always ample nodes from which to choose. Node availability is an advantage in that it supports the simultaneous investigation of several distinct paths which can lead to quicker integer solutions (either by direct solution or by the heuristic procedure). The mere presence of nodes from which to choose does not necessarily lead to any speedup. The ability to find bounds on the solution value heavily affects the performance of the Extended Method. If a good upper bound (in the case of a minimization problem) is not available, then the processors may be working well past the optimal point without knowing it. This is precisely the case with Problem P2756, which has a large number of nodes yet loses efficiency as processors are added. Problem 0201, on the other hand, does have good bounds established for it quickly. Thus, the combination of an abundance of nodes from which to choose coupled with the information necessary to decide that a path is non-optimal allows Problem P0201 to be solved very efficiently while demonstrating acceleration anomalies.

Problem P2756, like Problem P0201, benefits from substantial run-time improvements in a distributed environment when compared with the Original Method. The Extended Method running on one processor takes longer than the Original Method, but when as few as two processors are used, the elapsed times are less than those obtained with the Original Method. We note that linear and superlinear speedup was achieved for the 2-, 3-, and 4-processor configurations.

The characteristics of Problem P2756 are quite different from those of the other six problems. On P2756 we observe linear or better speedups when four or fewer processors are used, but when more processors are added we observe a deceleration in the speedup. In solving Problem P2756, a great deal of time is spent in the linear-programming solver and relatively little time branching. In addition, the heuristic procedures are not as effective in supplying an upper bound on this problem as they have been on others. Finally, the optimal solution is found high in the tree with an answer very close to the initial linear-programming solution. The combination of these characteristics caused the Extended Method to not achieve linear speedup in some cases.

The parallel implementation of the Extended Method relies on a branching strategy to be effective. That is, in the absence of a branching tree, the effects of *parallelism* in the Extended Method are extinguished. (It should be noted that the improvements of the Extended Method due to the Constraint Pool still remain and are significant.) Because the solution of Problem P2756 is achieved near the top of the tree and because of the startup procedures of the Extended Method, all 32 initial nodes must be evaluated before continuing with an informed search. In all other cases, this approach has not been detrimental because the linear-programming solution times were very short (the next largest order of magnitude is Problem P0201, with two to four minutes between branching nodes). For Problem P2756 though, with the time between branches on the order of 20 to 30 minutes, the effect of the additional work imposed by the Extended Method startup procedures becomes quite pronounced. As can be seen in table 3 and fig. 7, total elapsed time decreases rapidly with the addition of new processors. One can also see that the Extended Method with a 1-node start solved the problem to optimality in one fourth the time (see table 7).

On the other problems, the heuristic procedures have worked very well in providing an early integer answer that was near the optimal integer answer. On Problem P2756, that has not happened. Without a good upper bound on the problem, processors are devoting much effort to solving subproblems that later are shown to be past the optimal answer.

The third reason that the effectiveness of the Extended Method begins to diminish after the fourth processor in Problem P2756 is that the solution is found very close to the initial linear-programming relaxation of the problem. The combination of the length of time taken within a node (20 to 30 minutes between branches) and the absence of a good upper bound allows processors to work for relatively long periods of time on non-optimal paths before fathoming them.

Table 7

The effect of a 32-node start versus a 1-node start on solution times (VAXstation 2000 processors in minutes)

Problem	Extended Method 32-node start	Extended Method 1-node start	Minutes (shorter) longer
P0033	1.9	1.6	(0.3)
P0040	0.8	0.6	(0.2)
P0201	371.0	395.2	24.2
P0282	6.4	14.8	8.4
P0291	1.6	1.3	(0.3)
P2756	872.6	247.3	(625.3)

Extended Method: The Extended Method running on one processor using either a 32-node start or a single-node start. Reported time includes overhead associated with establishing a distributed-processing environment even though there is no parallelism used.

For the four quickly-solved problems (P0033, P0040, P0282 and P0291), detrimental and deceleration anomalies were both expected and observed. Elapsed times for these problems are presented graphically in figs. 2, 3, 5 and 6, respectively. On each of these problems, "overhead" time is spent initializing data structures, synchronizing for distributed processing, collecting performance data, and preparing final reports. For a problem which is solved very quickly, the overhead time alone can exceed the solution time. For Problems P0033, P0040 and P0291, each of which is solved in under two minutes, the solution time for one processor is worse than the Original Method. As more processors are added to the system, the solution time decreases until the fourth or fifth processor is added, at which point the solution time begins increasing. We note, however, that the Extended Method solution time occasionally drops below that of the Original Method. We conclude, therefore, that the detrimental anomaly is due to overhead time.

The solution of Problem P0282 follows the same pattern of first decreasing, then increasing, solution times, as do the other quickly-solved problems. In contrast to the other three problems, Problem P0282 using the Extended Method with only one processor is solved in one-half the time taken by the Original Method. This behavior is attributed to two factors: the effectiveness of the Constraint Pool and the start-up procedure of the Extended Method.

The start-up procedure of the Extended Method helps to solve Problem P0282 more quickly. Because 32 nodes are generated immediately, the solution space is more restricted from the beginning. This additional restriction of variables leads to faster fathoming of non-optimal branches. This conclusion is further supported by noting that when Problem P0282 was solved by the Extended Method without the 32-node start, the elapsed time was 14.8 minutes compared with 12.9 minutes for the Original Method (see table 7).

5.3. EFFECT OF A GOOD UPPER BOUND

The branch-and-cut algorithm is a "bounding" procedure whereby one iteratively tightens the upper and lower bounds until both bounds are equal. We therefore believe that if one is provided with a good upper bound, the amount of effort required to find and prove optimality will be lessened. To test how our parallel implementation might be affected by a good upper bound, we provided the system with an artificial ZSTAR that was slightly higher (1.5 units higher) than the true optimal solution. We chose to have it slightly higher than the true optimal solution so that the system would find and verify the optimal solution. Table 8 shows that computation time was reduced substantially when the system was provided with a tight bound. We conclude that the ability to determine good upper bounds is critical to achieving substantial gains in the run-time performance. Further, we believe that if heuristic procedures are employed to determine those upper bounds, then the bounds can be provided earlier in the solution process, leading to a substantially altered flow of the overall procedure.

Table 8

The effect of a good upper bound

Problem	Elapsed time of Extended Method (min)*	Elapsed time of Extended Method with a very good upper bound (min)*	% Reduction
P0201	45.07	30.44	32.5
P2756	154.58	102.54	33.7

*Problems were solved using eight VAXstation 2000s.

Another reason for providing the system with a good upper bound was to test if having such a tight bound might force I/O bottlenecks. A ZSTAR that is close to the optimal solution causes branches to be fathomed more quickly. Each time a branch is fathomed, the Candidate List must be accessed by the processor to determine the next best branch. The Constraint Pool is then accessed to determine whether cuts in the Pool violate the linear-programming solution associated with the new node. We thought that this disk activity, working across an Ethernet network, might lead to increased I/O and possible bottlenecks. The ratio I/O to total time did not increase substantially; no bottlenecks occurred and total computation time declined dramatically. The I/O considerations will be discussed in the next section.

5.4. INPUT/OUTPUT CONSIDERATIONS

A common shortcoming of previous attempts to implement a search strategy in a distributed-processing environment has been a bottleneck in interprocess communication. In the Extended Method, interprocess communication takes two forms: notification of changes in the problem (ZSTAR and TARGET) and sharing of data (a single, common Candidate List and Constraint Pool). Communication bottlenecking was not observed for either form within the Extended Method.

The notification of changes in ZSTAR and TARGET occurs very quickly. We did not conduct any measurements of resource locking for the test set for two reasons. First, generally fewer than five intermediate integer answers are found before the optimal solution is determined, while TARGET changes are on the order of 20 per run. Because our elapsed times are relatively long, we conclude that the small amount of notification would not substantively alter the elapsed time. The second reason that we did not measure the locking speed is that Snaman and Thiel [48] report that in a Local Area VAXcluster, a total of 8.1 milliseconds (elapsed) is taken to enqueue and dequeue a lock. Further, they report that 7.8 milliseconds (elapsed) are required for a lock conversion (up and down). In our several hundred runs of these problems, we have been given no reason to dispute the findings of Snaman and Thiel.

Thus, if I/O bottlenecks were to occur, they would occur in the constant accessing of the Constraint Pool and the Candidate List. Extensive measurements were taken to determine exactly how much time was being spent in accessing these files. We measured elapsed time taken to access a record when reading from the disk file and the amount of time taken to insert a record when writing to the disk file. As shown in tables 9 and 10, I/O time accounts for only a small portion of the elapsed time. (On P2756, where as many as 95,341 constraints were read and 938 distinct constraints were written, I/O time as a percent of elapsed time was never more than 5%.)

It should be remembered that disk files were implemented solely to support a distributed-processing environment. When the Constraint Pool is adopted for use in a sequential environment, or when the Extended Method is adopted for a tightly-coupled parallel-processing environment, disk files would be discarded and in-core computer memory used instead.

5.5. EFFECT OF CHANGING TARGET

The determination of when to pause a node (the TARGET value) has a pronounced effect on run-time performance. Remember that *TARGET* is calculated as follows:

$$TARGET = z_{LP} - (PERC * (z_{LP} - ZSTAR)) .$$

Table 9

Input/output time analysis for Problem P0201

Members	Constraints		Candidates		Σ of I/O (min)	Σ elapsed time (min)	I/O as % of elapsed	Run-time (min)
	Read	Wrote	Read	Wrote				
1	57,375	207	307	390	14.4	371.0	3.87	371.0
2	42,479	202	310	392	17.3	369.8	4.68	185.0
3	46,708	203	315	396	20.0	370.4	5.41	123.6
4	61,031	253	325	406	28.9	389.3	7.42	96.6
5	58,832	224	317	392	35.4	576.3	6.13	75.3
6	55,169	201	340	413	27.4	369.7	7.40	61.9
7	60,559	215	345	414	30.0	362.1	8.29	52.0
8	60,799	197	322	392	32.0	358.1	8.94	45.1

Table 10

Input/output time analysis for Problem P2756

Members	Constraints		Candidates		Σ of I/O (min)	Σ elapsed time (min)	I/O as % of elapsed	Run-time (min)
	Read	Wrote	Read	Wrote				
1	44,296	678	93	108	15.8	872.6	1.81	872.6
2	50,203	801	90	113	25.6	730.8	3.51	365.5
3	70,804	849	101	130	23.6	870.4	2.71	290.6
4	50,921	688	89	103	27.6	739.8	3.73	181.1
5	67,040	753	111	123	33.6	766.8	4.38	191.9
6	66,511	835	104	122	38.6	1,027.0	3.76	171.5
7	72,535	751	117	124	44.9	1,076.0	4.17	154.4
8	95,341	938	137	161	61.4	1,230.1	4.99	154.6

Table 11

The effect of TARGET on elapsed time for Problem P2756

PERC	BETA	Elapsed time (min)
0.10	0.002	872.58
0.20	0.002	997.00
0.30	0.002	1630.18
0.50	0.002	1859.00

If *TARGET* falls within (*ZSTAR*/(1 + *BETA*)) of *ZSTAR*, then *TARGET* is set equal to *ZSTAR*. The effect of variations in the user-supplied parameters *BETA* and *PERC* for Problem P2756 is presented in table 11.

Because progress within a node is so slow in Problem P2756, and because the gap is so large, we want to pause nodes early in the process. Therefore, for Problem P2756, BETA has been set to 0.002 and PERC has been set to 0.10. In contrast, Problem P0201 (characterized by a shorter time within a node and an optimal solution far from the initial linear-programming solution) runs with BETA equal to 0.02 and PERC equal to the default of 0.80.

Future research will address the appropriate settings of these values and the dynamic changing of BETA and PERC under software control.

6. Conclusions and future research

We have found that large-scale zero-one integer programming problems can be solved quickly by using a distributed-processing approach. On problems characterized by a large number of branching nodes, linear speedup, and sometimes superlinear speedup, can be achieved. On problems which are solved very quickly, deceleration and detrimental anomalies were both expected and observed.

A common criticism of distributed processing on local area networks is the bottleneck caused by I/O functions. We have found no I/O bottlenecks resulting from our testing. We have used shared, indexed files as a means of passing data between processors. Extensive measurements have been taken on the two largest problems (201 variables/134 constraints and 2756 variables/756 constraints) and show that total disk-file related I/O accounts for an average of approximately 5% of the total elapsed time.

It should be noted, however, that traditional approaches to solving these problems in a parallel-processing environment rely heavily on branching, which leads to an increased level of interprocess communication. Because the branch-and-cut method relies on branching strictly as a last resort, interprocess communication for the purpose of altering bounding information, or for choosing nodes, is reduced significantly.

The minimal impact of I/O activity coupled with the relatively low requirement for interprocess communication allows us to conclude that this implementation will work for much larger zero-one linear programming problems within a branch-and-cut framework. As larger problems are obtained, interprocess communication and data sharing procedures may be refined as necessary.

During the progress of this research, a number of topics were identified as either natural extensions of this work or as areas that may provide insight into the solution of this class of problems. The topics are:

Investigate the use of sensitivity analysis to establish a better bound for a branching node than the linear-programming relaxation of its parent. Currently, when a decision is made to branch on a variable, two records are inserted in the Candidate List, using the last linear-programming objective function obtained as the record key. Since effort within a node in a branch-and-cut solver includes routine solving of multiple linear programs, constraint generation, logical fixing, and heuristics, it may be worthwhile to solve the initial linear program associated with the new node (i.e. perform *all* of the work normally done in a branch-and-bound code for that node) in order to return a more accurate bound for each node, thereby ensuring true best-first ordering (i.e. making a distinction between the "up" and the "down" branch) and hastening fathoming.

Investigate the feasibility of using an idle machine to help an active processor. There are occasions in the Extended Method when processors are idle. Future research will investigate the feasibility of using idle processors to assist active processors in one of three ways: by having an active processor produce a branching variable (two candidates inserted in the Candidate List) whenever a processor is idle; by having an idle processor employ alternative heuristic approaches to achieve a better upper bound; and by having an active processor decompose its problem to allow idle processors to assist it within a node.

Implement the ability for a user to quickly regain control of his workstation without affecting the integrity of the solution. The Extended Method has been designed to allow a processor to enter the system at any point in the solution. The ability for a processor to voluntarily exit has not been provided and will be researched further. We feel that we can provide a user-activated interrupt mechanism which will allow the user to signal when the workstation is needed. That interrupt will cause the processor to immediately return the current candidate to the Candidate List at the latest linear-programming objective function. The workstation can then be returned to the user. It should be noted that this feature is not required for VAXstation 2000 workstations since they can support multi-user processing. The feature is highly desirable though, because of the compute-intensive nature of these problems and because most individual workstation applications are compute-intensive as well. The contention between the Extended Method and the user's application will cause each to perform poorly.

Much research on improving current techniques for solving combinatorial optimization problems by using parallel processing is still needed. We will continue to investigate the solution of these problems in both distributed- and shared-memory environments. One of the limiting factors in our research is the lack of available large, real, non-proprietary combinatorial optimization problems. Perhaps the ability to solve large problems will encourage the formulation and distribution of increasingly larger problems.

Trademarks

The following are trademarks of the Digital Equipment Corporation: DEC, DECnet, DECnet-VAX, Digital Network Architecture (DNA), Local Area VAXcluster, MicroVAX, MicroVAX II, RMS, VAX, VAX FORTRAN, VAXstation, VAXstation 2000, VAX/VMS.

Acknowledgements

The authors wish to thank Manfred Padberg for his helpful suggestions on improving this research and careful reviewing of an early version of this paper. We have tried to incorporate as many of his suggestions as was possible, but of course, any errors, omissions or inconsistencies are entirely our own responsibility.

References

[1] M. Ajmone Marsan, G. Balbo and G. Conte, *Performance Models of Multiprocessor Systems* (The MIT Press, Cambridge MA, 1986).

[2] A. Bachem and M. Grötschel, New aspects of polyhedral theory, in: *Modern Applied Mathematics, Optimization and Operations Research*, ed. B. Korte (North-Holland, Amsterdam, 1982) pp. 51 – 106.

[3] E. Balas, Intersection cuts – a new type of cutting planes for integer programming, Oper. Res. 19, 1(1971)19.

[4] T.L. Cannon, Large-scale zero-one linear programming on distributed workstations, Ph.D. dissertation, Department of Operations Research and Applied Statistics, George Mason University, Fairfax, VA (1988).

[5] T.L. Cannon and K.L. Hoffman, The effect of a constraint pool on large-scale zero-one linear programming problems, Technical Report (1989), in preparation.

[6] M.D. Chang, M. Enquist, R. Finkel and R.R. Meyer, A parallel algorithm for generalized networks, Technical Report 642, Department of Computer Sciences, University of Wisconsin, Madison (1987).

[7] H. Crowder, E.L. Johnson and M. Padberg, Solving large-scale zero-one linear programming problems, Oper. Res. 31, 5(1983)803.

[8] G.B. Dantzig, Notes on solving linear programs in integers, Naval Research Logistics Quarterly 6(1959)75.

[9] G.B. Dantzig, D.R. Fulkerson and S. Johnson, Solution of a large-scale traveling salesman problem, Oper. Res. 2(1954)393.

[10] A. de Bruin, A.H.G. Rinnooy Kan and H.W.J.M. Trienekens, A simulation tool for the performance evaluation of parallel branch and bound algorithms, Report 8720/A, Econometric Institute, Erasmus University, Rotterdam (1987).

[11] D. DeWitt, R. Finkel and M. Solomon, The Crystal multicomputer: Design and implementation experience, Technical Report 553, Computer Science Department, University of Wisconsin-Madison (1984).

[12] D.J. Duffy, The system communication architecture, Digital Technical Journal 5(1987)22.

[13] O. El-Dessouki and W.H. Huen, Distributed enumeration on between computers, IEEE Trans. on Computers C-29, 9(1980)818.

[14] R.A. Finkel and U. Manber, DIB-A distributed implementation of backtracking, Technical Report 588, Department of Computer Sciences, University of Wisconsin, Madison (1985).
[15] M.S. Fox and J.A. Ywoskus, Local Area VAXcluster systems, Digital Technical Journal 5(1987)56.
[16] E.F. Gehringer, D.P. Siewiorek and Z. Segall, *Parallel Processing. The Cm* Experience* (Digital Press, Rockport, MA, 1987).
[17] F. Glover, A bound escalation method for the solution of integer linear programs, Cah Cent Etud Rech Operationelle 6(1965)131.
[18] F. Glover, Convexity cut and cut search, Oper. Res. 21(1973)123.
[19] A.C. Goldstein, The design and implementation of a distributed file system, Digital Technical Journal 5(1987)45.
[20] R.E. Gomory, Outline of an algorithm for integer solutions to linear programs, Bull. Amer. Math. Soc. 64(1958)275.
[21] B. Grünbaum, *Convex Polytopes* (Wiley, London, 1967).
[22] K.L. Hoffman and M. Padberg, Techniques for improving the linear programming representation of pure zero-one linear programming problems, Technical Report, Department of Operations Research and Applied Statistics, George Mason University, Fairfax, VA (1989).
[23] K.L. Hoffman and M. Padberg, ABCOPT: A branch-and-cut optimizer for sparse zero-one linear programs, Preprint, New York University (1989).
[24] M. Imai, T. Fukumura and Y. Yoshida, A parallelized branch-and-bound algorithm implementation and efficiency, Systems Computer Controls 10, 3(1979)62.
[25] M. Imai, Y. Yoshida and T. Fukumura, A parallel searching scheme for multiprocessor systems and its application to combinatorial problems, in: *Proc. 6th Int. Joint Conf. on Artificial Intelligence* (1979) pp. 416–418.
[26] E.L. Johnson, M.W. Kostreva and U.H. Suhl, Solving 0-1 integer programming problems arising from large-scale planning models, Oper. Res. 33, 4(1985)803.
[27] G.A.P. Kindervater and J.K. Lenstra, An introduction to parallelism in combinatorial optimization, Discr. Appl. Math. 14(1986)135.
[28] N.P. Kronenberg, H.M. Levy, W.D. Strecker and R.J. Merewood, The VAXcluster concept: An overview of a distributed system, Digital Technical Journal 5(1987)7.
[29] T.-H. Lai and S. Sahni, Anomalies in parallel branch-and-bound algorithms, Commun. ACM 27, 6(1984)594.
[30] T.-H. Lai and A. Sprague, Performance of parallel branch-and-bound algorithms, IEEE Trans. on Computers C-34, 10(1985)962.
[31] G.-J. Li and B.W. Wah, Computational efficiency of parallel approximate branch-and-bound algorithms, in: *Proc. 1984 Int. Conf. on Parallel Processing* (1984) pp.473–480.
[32] G.-J. Li and B.W. Wah, Coping with anomalies in parallel branch-and-bound algorithms, IEEE Trans. on Computing 35, 6(1986)568.
[33] G.-J. Li and B.W. Wah, How good are parallel and ordered depth-first searches?, in: *Proc. 1986 Int. Conf. on Parallel Processing* (1986) pp. 992–999.
[34] R. Marsten, The design of the XMP linear programming library, ACM Trans. on Mathematical Software 7(1981)481.
[35] J. Mohan, A study in parallel computation – the traveling salesman problem, Technical Report CMU-CS-82-136, Computer Science Department, Carnegie-Mellon University (1982).
[36] J. Mohan, Experience with two parallel programs solving the traveling salesman problem, in: *Proc. 1983 Int. Conf. on Parallel Processing*, IEEE, New York (1983) pp. 191–193.
[37] D.S. Nau, V. Kumar and L. Kanal, General branch-and-bound, and its relation to A* and AO*, Artificial Intelligence 23(1984)29.

[38] G.L. Nemhauser and L.A. Wolsey, *Integer and Combinatorial Optimization* (Wiley, New York, 1988).
[39] M. Padberg, Essays in integer programming, Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, PA (1971).
[40] M. Padberg, Covering, packing and knapsack problems, Ann. Discr. Math. 4(1979)265.
[41] M. Padberg and G. Rinaldi, Optimization of a 532-city traveling salesman problem by branch-and-cut, Oper. Res. Lett. 6(1987)1.
[42] M. Padberg and G. Rinaldi, A branch-and-cut algorithm for the solution of large-scale traveling salesman problems, Technical Report, New York University (1988).
[43] E.A. Pruul, G.L. Nemhauser and R.A. Rushmeier, Branch-and-bound and parallel computation: A historical note, Oper. Res. Lett. 7, 2(1988)65.
[44] M.J. Quinn and N. Deo, An upper bound for the speedup of parallel best-bound branch-and-bound algorithms, BIT 26, 1(1986)35.
[45] R.T. Rockafellar, *Convex Analysis* (Princeton University Press, Princeton, N.J., 1970).
[46] R.B. Schnabel, Parallel computing in optimization, in: *Proc. NATO Advanced Study Institute on Computational Mathematical Programming,* Bad Windsheim, F.R.G. (1984) pp. 358–381.
[47] A. Schrijver, *Linear and Integer Programming* (Wiley, 1986).
[48] W.E. Snaman, Jr. and D.W. Thiel, The VAX/VMS distributed lock manager, Digital Technical Journal 5(1987)29.
[49] J. Stoer and C. Witzgall, *Convexity and Optimization in Finite Dimensions I* (Springer-Verlag, Berlin, 1970).
[50] H.A. Taha, *Operations Research – An Introduction,* 3rd ed. (Macmillan, New York, 1982).
[51] C.A. Trauth and R.E. Woolsey, Integer linear programming: A study in computational efficiency, Management Science 15(1969)481.
[52] H.W.J.M. Trienekens, Parallel branch-and-bound on an MIMD system, Report 8640/A, Econometric Institute, Erasmus University, Rotterdam, 1986).
[53] H.M. Wagner, R.J. Giglio and R.G. Glaser, Preventive maintenance scheduling by mathematical programming, Management Science 10(1964)316.
[54] B.W. Wah and Y.W. Ma, MANIP – a parallel computer system for implementing branch and bound algorithms, in: *Proc. 8th Annual Int. Symp. on Computer Architecture* (1982) pp. 239–262.
[55] B.W. Wah and Y.W. Ma, MANIP – a multicomputer architecture for solving combinatorial extremum-search problems, IEEE Trans. on Computers C-33, 5(1984)377.
[56] B.W. Wah, G. Li and C.F. Yu, Multiprocessing of combinatorial search problems, IEEE Trans. on Computers 18, 6(1985)93.
[57] R.D. Young, Hypercylindrically deduced cuts in 0-1 integer programming, Oper. Res. 19, 6(1971)1393.