FATCOP: A FAULT TOLERANT CONDOR-PVM MIXED INTEGER PROGRAMMING SOLVER *

QUN CHEN † AND MICHAEL C. FERRIS ‡

Abstract. We describe FATCOP, a new parallel mixed integer program solver written in PVM. The implementation uses the Condor resource management system to provide a virtual machine composed of otherwise idle computers. The solver differs from previous parallel branch-and-bound codes by implementing a general purpose parallel mixed integer programming algorithm in an opportunistic multiple processor environment, as opposed to a conventional dedicated environment. It shows how to make effective use of resources as they become available while ensuring the program tolerates resource retreat. The solver performs well on test problems arising from real applications, and is particularly useful for solving long-running hard mixed integer programming problems.

Key words. Mixed integer programming, Condor, opportunistic environment, branch-and-bound, fault tolerance

AMS subject classifications. 90C11,65K05

1. Introduction. Mixed integer programming (MIP) problems are difficult and commonplace. For many of these hard problems, only small instances can be solved in a reasonable amount of time on sequential computers, resulting in mixed integer programming being a frequently cited application of parallel computing. Most available general-purpose large-scale MIP codes use branch-and-bound to search for an optimal integer solution by solving a sequence of related linear programming (LP) relaxations that allow possible fractional values. This paper discusses a new parallel mixed integer program solver, written in PVM, that runs in the opportunistic computing environment provided by the Condor resource management system.

Parallel branch-and-bound algorithms for MIP have attracted many researchers (see [11, 15, 24] and references therein). Most parallel branch-and-bound programs were developed for large centralized mainframes or supercomputers that are typically very expensive. Users of these facilities usually only have a certain amount of time allotted to them and have to wait their turn to run their jobs. Due to the decreasing cost of lower-end workstations, large heterogeneous clusters of workstations connected through fast local networks are becoming common in work places such as universities and research institutions. In this paper we shall refer to the former resources as dedicated resources and the later as distributed ownership resources. The principal goal of the research outlined in this paper is to exploit distributed ownership resources to solve extremely difficult mathematical programming problems. While the environment is currently not "common or realistic", we believe it is an important one for the future of certain numerical computations, including computations for discrete optimization problems. In fact, in the time while this paper has been under review, several other authors have adapted the scheme first outlined in this work for other branch-and-bound applications, most notably in [1, 16, 27].

*This material is based on research supported in part by National Science Foundation Grants CDA-9726385, CCR-9619765 and CCR-9972372 and Air Force Office of Scientific Research Grant F49620-98-1-0417.

†Dept. of Industrial Engineering, University of Wisconsin, Madison, WI 53706. email: chenq@cae.wisc.edu

‡Department of Computer Sciences, University of Wisconsin, Madison, WI 53706. email: ferris@cs.wisc.edu

A parallel virtual machine (PVM) is a programming environment that allows a heterogeneous network of computers to appear as a single concurrent computational resource [14]. It provides a unified framework within which parallel programs for a heterogeneous collection of machines can be developed in an efficient manner. However PVM is not sufficient to develop an efficient parallel branch-and-bound program in a distributed ownership environment. The machines in such an environment are usually dedicated to the exclusive use of individuals. The application programming interface defined by PVM requires that users explicitly select machines on which to run their programs. Therefore, they must have permission to access the selected machines and cannot be expected to know the load on the machines in advance. Furthermore, when a machine is claimed by a PVM program, the required resources in the machine will be "occupied" during the life cycle of the program. This is not a desirable situation when the machine is owned by a person different from the user of the MIP solver.

Condor [12, 20] is a distributed resource management system that can help to overcome these problems. Condor manages large heterogeneous clusters of machines in an attempt to use the idle cycles of some users' machines to satisfy the needs of others who have computing intensive jobs. It was first developed for long running sequential batch jobs. The current version of Condor provides a framework (Condor-PVM) to run parallel programs written in PVM in a distributed ownership environment. In such programs, Condor is used to dynamically construct a PVM out of non-dedicated desktop machines on the network. Condor allows users' programs to run on any machine in the pool of machines managed by Condor, regardless of whether the user submitting the job has an account there or not, and guarantees that heavily loaded machines will not be selected for an application. To protect ownership rights, whenever a machine's owner returns, Condor immediately interrupts any job that is running on that machine, migrating the job to another idle machine. Since resources managed by Condor are competed for by owners and many other Condor users, we refer to such resources as Condor's *opportunistic resources* and the Condor-PVM parallel programming environment as the Condor-PVM *opportunistic environment*.

FATCOP represents a first attempt to develop a general purpose parallel solver for mixed integer programs in Condor's opportunistic environment. It is hoped that many of the lessons learned in developing FATCOP can be incorporated into more general branch-and-bound codes for other applications. FATCOP is implemented on top of both SOPLEX, a public available simplex object-oriented linear programming solver [28], and the CPLEX LP solver [9]. FATCOP is written in the C++ programming language with calls to PVM library. It is designed to make best use of participating resources managed by Condor while handling resource retreat carefully in order to ensure the eventual and correct completion of a FATCOP job. Key features of FATCOP include:

- parallel implementation under Condor-PVM framework;
- greedy utilization of Condor's opportunistic resources;
- powerful MIP techniques including strong branching, pseudocost estimation searching, preprocessing, and cutting plane generation;
- the ability to process both MPS [21] and GAMS [7] models;
- the use of both CPLEX and SOPLEX as its LP solver.

The remainder of this paper is organized as follows. Section 2 is a review of the standard MIP algorithm components of FATCOP that are implemented to ensure the branch-and-bound algorithm generates reasonable search trees. Section 3 introduces

Condor-PVM parallel programming framework and the parallel implementation of FATCOP. In Section 4, we present some numerical results that exhibit important features of FATCOP. A brief summary and future directions are given in Section 5.

2. Components of the sequential program. A MIP can be stated mathematically as follows:

$$\begin{array}{rll} \min & c^T x & \\ \text{s.t.} & Ax \leq b & \\ & l \leq x \leq u & \\ & x_j \in Z & \forall j \in I. \end{array}$$

Here Z denotes the integers, A is an $m \times n$ matrix, and I is a subset of the indices identifying the integer variables.

Integer programming textbooks such as [22] describe the fundamental branch-and-bound algorithm for the above MIP problem. Basically, the method explores a binary tree of subproblems. Branching refers to the process of creating refinements of the current relaxation, while bounding of the LP solution is used to eliminate exploration of parts of the tree. The remainder of this section describes refinements to this basic framework.

2.1. Preprocessing. Preprocessing refers to a set of reformulations performed on a problem instance. In linear programming this typically leads to problem size reductions. FATCOP identifies infeasibilities and redundancies, tighten bounds on variables, and improves the coefficients of constraints [25]. At the root node, FATCOP analyzes every row of the constraint matrix. If, after processing, some variables are fixed or some bounds are improved, the process is repeated until no further model reduction occurs.

In contrast to LP, preprocessing may reduce the *integrality gap*, i.e., the difference between the optimal solution value and its LP relaxation as well as the size of a MIP problem. For example, for the model *p0548* from MIPLIB [5], an electronically available library of both pure and mixed integer programs arising from real applications, the FATCOP preprocessor can only remove 12 rows, 16 columns, and modify 176 coefficients from the original model that has 176 rows, 548 columns and 1711 non zero coefficients, but pushes the optimal value of the initial LP relaxation from 315.29 up to 3125.92.

2.2. Cutting planes and reduced cost fixing. It is well known that cutting planes can strengthen MIP formulations. FATCOP generates knapsack cuts at each subproblem as described in [18]. There are about 10 models in MIPLIB for which knapsack cuts are useful. We again take *p0548* as an example; the FATCOP code can solve the model in 350 nodes with knapsack cuts applied at each node. However, it is not able to solve the problem to optimality in 100,000 nodes without knapsack cuts.

FATCOP also incorporates a standard reduced cost fixing procedure [11] that fixes integer variables to their upper or lower bounds by comparing their reduced costs to the gap between a linear programming solution value and the current problem best upper bound.

2.3. Variable and node selection. Several reasonable criteria exist for selecting branching variables. FATCOP currently provides four variable selection options: pseudocost [19], strong branching [6], maximum and minimum integer infeasibility [2]. Since the pseudocost method is widely used and known to be efficient, we set it as the default branching strategy. FATCOP can also accept user defined priorities on integer variables.

FATCOP provides five options for selecting a node from those remaining: depth-first, best-bound, best-estimation [2], a mixed strategy of depth-first and best-bound [11] (mixed strategy 1), and a mixed strategy of best-estimation and best-bound (mixed strategy 2).

Mixed strategy 1 expands the subproblems in the best-first order, but with an initial depth-first phase. FATCOP keeps track of the number of node evaluations over which the best integer solution has not been updated. It then switches searching strategy from depth-first to best-first after this number exceeds a pre-specified fixed number. Mixed strategy 2 is similar to mixed strategy 1, but starts the algorithm with best-estimation search first. Since best-estimation often finds better solutions than depth first does, mixed strategy 2 is set as default searching strategy for FATCOP.

3. Condor-PVM parallel implementation of FATCOP. In this section we first give a brief overview of Condor, PVM and the Condor-PVM parallel programming environment. Then we discuss the parallel scheme we selected for FATCOP and the differences between normal PVM and Condor-PVM programming. At the end of the section, we present a detailed implementation of FATCOP.

3.1. The Condor-PVM parallel programming environment. Heterogeneous clusters of workstations are becoming an important source of computing resources. Two approaches have been proposed to make effective use of such resources. One approach provides efficient resource management by allowing users to run their jobs on idle machines that belong to somebody else. Condor, developed at University of Wisconsin-Madison, is one such system. It monitors the activity on all participating machines, placing idle machines in the Condor pool. Machines are then allocated from the pool when users send job requests to Condor. Machines enter the pool when they become idle, and leave when they get busy, e.g. the machine owner returns. When an executing machine becomes busy, the job running on this machine is initially suspended in case the executing machine becomes idle again within a short timeout period. If the executing machine remains busy, then the job is migrated to another idle workstation in the pool or returned to the job queue. For a job to be restarted after migration to another machine a checkpoint file is generated that allows the exact state of the process to be re-created. This design feature ensures the eventual completion of a job. There are various priority orderings used by Condor for determining which jobs and machines are matched at any given instance. Based on these orderings, running jobs may sometimes be preempted to allow higher priority jobs to run instead. Condor is freely available and has been used in a wide range of production environments for more than ten years.

Another approach to exploit the power of a workstation cluster is from the perspective of parallel programming. Research in this area has developed message passing environments allowing people to solve a single problem in parallel using multiple resources. One of the most widely used message passing environments is PVM that was developed at the Oak Ridge National Laboratory. PVM's design centers around the idea of a *virtual machine*, a very general notion that can encompass a nearly arbitrary collection of computing resources, from desktop workstations to multiprocessors to massively parallel homogeneous supercomputers. The goal of PVM is to make programming for a heterogeneous collection of machines straightforward. PVM provides process control and resource management functions that allow spawning and termination of arbitrary processes and the addition and deletion of hosts at runtime.

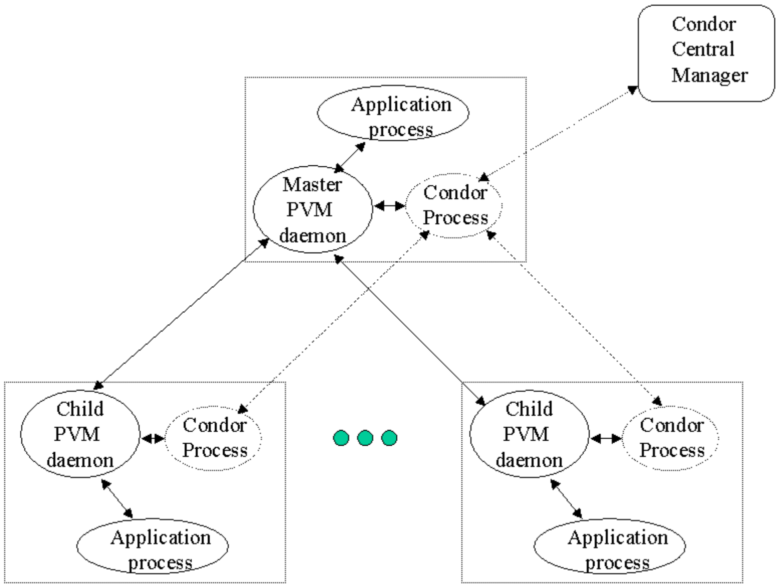

FIG. 3.1. *Architecture of Condor-PVM*

The PVM system is composed of two parts. The first part is a daemon that resides on all the computers comprising the virtual machine. The second part of the system is the PVM library. It contains user-callable routines for message passing, process spawning, virtual machine modification and task coordination. PVM transparently handles all message routing, data conversion and task scheduling across a network of incompatible computer architectures. A similar message passing environment is MPI [17]. Both systems center around a message-passing model, providing point-to-point as well as collective communication between distributed processes.

The development of resource management systems and message passing environments have been independent of each other for many years. Researchers at the University of Wisconsin have recently developed a parallel programming framework that interfaces Condor and PVM [23]. The reason to select PVM instead of MPI is that the implementation of MPI has no concept of process control, hence cannot handle resource addition and retreat in a opportunistic environment. Figure 3.1 shows the architecture of Condor-PVM. There are three processes on each machine running a Condor-PVM application: the PVM daemon, the Condor process and the user application process. The Condor-PVM framework still relies on the PVM primitives for application communication, but provides resource management in the opportunistic environment through Condor. Each PVM daemon has a Condor process associated with it, acting as the *resource manager*. The Condor process interacts with PVM daemon to start tasks, send signals to suspend, resume and kill tasks, and receive process completion information. The Condor process running on the master machine is special. It communicates with Condor processes running on the other machines, keeps information about the status of the machines and forwards resource requests to the Condor central manager. This Condor process is called the *global resource manager*. When a Condor-PVM application asks for a host (we will use host and machine interchangeably in the sequel), the global resource manager communicates with Condor central manager to schedule a new machine. After Condor grants a machine to the application, it starts a Condor process (resource manager) and a PVM daemon on the new machine. If a machine needs to leave the pool, the resource manager will send signals to the PVM daemon to suspend tasks. The master user application is notified of that via normal PVM notification mechanisms.

Compared with a conventional dedicated environment, the Condor-PVM opportunistic environment has the following characteristics:

1. There usually are a large amount of heterogeneous resources available for an application, but in each time instance, the amount of available resources is random, dependent on the status of machines managed by Condor. The resources are competed for by owners and other Condor users.
2. Resources used by an application may disappear during the life cycle of the application.
3. The execution order of components in a particular application is highly nondeterministic, leading to different solution and execution times.

Therefore a good Condor-PVM application should be tolerant to loss of resources (host suspension and deletion) and dynamically adaptive to the current status of Condor pool in order to make effective use of opportunistic resources.

3.2. The parallel scheme for FATCOP. FATCOP introduces parallelism when building the branch-and-bound tree. It performs bounding operations on several subproblems simultaneously. This approach may affect the order of subproblems generated during the expansion of the branch-and-bound tree. Hence more or less subproblems could be evaluated by the parallel program compared with its sequential version. Such phenomena are known as *search anomalies* and examples are given in Section 4.

FATCOP was designed in the master-worker paradigm. One host, called the master manages the work pool, and sends subproblems out to other hosts, called workers, that solve LPs and send the results back to the master. When using a large number of workers, this centralized parallel scheme can become a bottleneck in processing the returned information, thus keeping workers idle for large amounts of time. However this scheme can handle different kinds of resource failure well in Condor's opportunistic environment, thus achieve the best degree of fault tolerance. The basic idea is that the master keeps track of which subproblem has been sent to each worker, and does not actually remove the subproblem out of the work pool. All the subproblems that are sent out are marked as "in progress by worker i". If the master is then informed that a worker has disappeared, it simply unmarks the subproblems assigned to that worker.

The remaining design issue is how to use the opportunistic resources provided by Condor to adapt to changes in the number of available resources. The changes include newly available machines, machine suspension and resumption and machine failure. In a conventional dedicated environment, a parallel application usually is developed for running with a fixed number of processors and the solution process will not be started until the required number of processors are obtained and initialized. In Condor's opportunistic environment, doing so may cause a serious delay. In fact the time to obtain the required number of new hosts from Condor pool can be unbounded.

Therefore we implement FATCOP in such a way that the solution process starts as soon as it obtains a single host. The solver then attempts to acquire new hosts as often as possible. At the beginning of the program, FATCOP places a number of requests for new hosts from Condor. Whenever it gets a host, allocates work to this host then immediately requests a new host. Thus, in each period between when Condor assigns a machine to FATCOP and when the new host request is received by Condor, there is at least one “new host” request from FATCOP waiting to be processed by Condor. This greedy implementation makes it possible for a FATCOP job to collect a significant amount of hosts during its life cycle.

3.3. Differences between PVM and Condor-PVM programming. PVM and Condor-PVM are binary compatible with each other. However there exist some run time differences between PVM and Condor-PVM. The most important difference is the concept of machine class. In a regular PVM application, the configuration of hosts that PVM combines into a virtual machine usually is defined in a file, in which host names have to be explicitly given. Under the Condor-PVM framework, Condor selects the machines on which a job will run, so the dependency on host names must be removed from an application. Instead the applications must use class names. Machines of different architecture attributes belong to different machine classes. Machine classes are numbered 0, 1, etc. and hosts are specified through machine classes. A machine class is specified in the submit-description file submitted to Condor, that specifies the program name, input file name, requirement on machines' architecture, operating system and memory etc.

Another difference is that Condor-PVM has “host suspend” and “host resume” notifications in addition to “host add”, “host deletion” and “task exit” notifications that PVM has. When Condor detects activity of a workstation owner, it suspends all Condor processes running there rather than killing them immediately. If the owner remains for less than a pre-specified cut-off time, the suspended processes will resume. To help an application to deal with this situation, Condor-PVM makes some extensions to PVM's notification mechanism.

The last difference is that adding a host is non-blocking in Condor-PVM. When a Condor-PVM application requests a new host be added to the virtual machine, the request is sent to Condor. Condor then attempts to schedule one from the pool of idle machines. This process can take a significant amount of time, for example, if there are no machines available in Condor's pool. Therefore, Condor-PVM handles requests for new hosts asynchronously. The application can start other work immediately after it sends out a request for new host. It then uses the PVM notification mechanism to detect when the “host add” request was satisfied. This feature allows our greedy host request scheme to work well in practice.

Documentation and examples about these differences can be found at

`http://www.cs.wisc.edu/condor/.`

FATCOP was first developed as a PVM application, and modified to exploit Condor-PVM.

3.4. The parallel implementation of FATCOP. FATCOP consists of two separate programs: the master program and the worker program. The master program runs on the machine from which the job was submitted to Condor. This machine is supposed to be stable for the life of the run, so it is generally the machine owned by the user. The design of FATCOP makes the program tolerant to any type of failures for workers, but if the machine running the master program crashes due to either system reboot or power outage, the program will be terminated. To make FATCOP tolerant

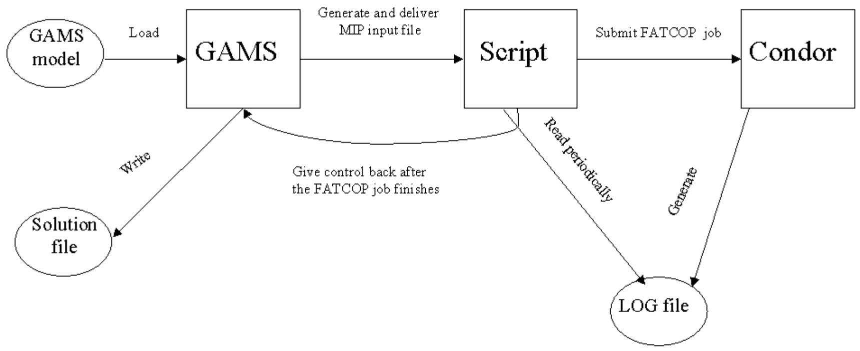

FIG. 3.2. *Interactions among Condor, FATCOP and GAMS*

even of these failures, the master program writes information about subproblems in the work pool periodically to a log file on the disk. Each time a FATCOP job is started by Condor, it reads in the MIP problem as well as the log file that stores subproblem information. If the log file does not exist, the job starts from the root of the search tree. Otherwise, it is warm started from some point in the search process. The work pool maintained by the master program has copies for all the subproblems that were sent to the workers, so the master program is able to write complete information about the branch-and-bound process to the log file.

The worker program runs on the machines selected by Condor. The number of running worker programs changes over time during the execution of a FATCOP job.

3.4.1. The master program. FATCOP can take both MPS and GAMS models as input. The interactions among Condor, FATCOP and GAMS are as follows. A user starts to solve a GAMS model in the usual way from the command line. After GAMS reads in the model, it generates an input file containing a description of the MIP model to be solved. Control is then passed to a PERL script. The script generates a Condor job description file and submits the job to Condor. After submitting the job, the script reads a log file periodically until the submitted job is finished. The log file is generated by Condor and records the status of the finished and executing jobs. After completion, control is returned to GAMS, which then reports the solution to the user. This process is depicted in Figure 3.2. The process is similar for MPS file input.

The MIP model is stored globally as an LP and integrality constraints. The master program first solves the LP relaxation. If it is infeasible or the solution satisfies the integrality constraints, the master program stops. Otherwise, it starts a sequential MIP solve process until there are N solved LP subproblems in the work pool. N is

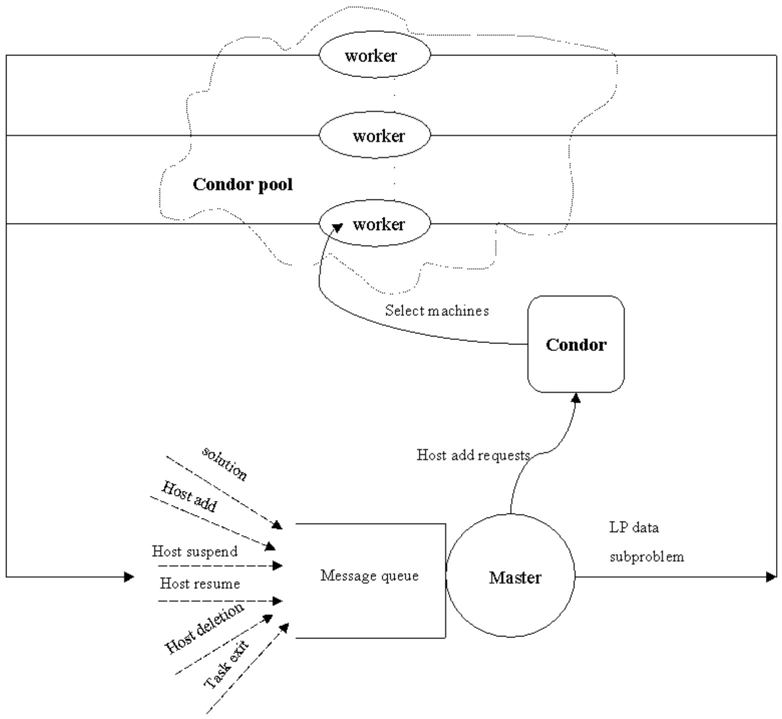

FIG. 3.3. *Message passing inside FATCOP*

a pre-defined number, that has a default value and can be modified by users. This process is based on the observation that using parallelism as soon as few subproblems become available may not be a good policy, since doing so may expand more nodes compared to a sequential algorithm. Associated with each subproblem in the work pool is the LP relaxation solution and value, modified bound information for the integer variables, pseudocosts used for searching and an optimal basis, that is used for warm starting the simplex method. The subproblems in the work pool are multi-indexed by bound, best-estimation and the order in which they entered the pool. The indices correspond to different searching rules: best-first, best-estimation and depth-first.

Following the initial subproblem generation stage, the master program sends out a number of requests for new hosts. It then sits in a loop that repeatedly does message receiving. The master accepts several types of messages from workers. The messages passing within FATCOP are depicted in Figure 3.3 and are explained further below. After all workers have sent solutions back and the work pool becomes empty, the master program kills all workers and exits itself.

Host Add Message.. After the master is notified of getting a new host, it spawns a child process on that host and sends an LP copy as well as a subproblem to the new child process. The subproblem is marked in the work pool, but not actually removed from it. Thus the master is capable of recovering from several types of failures. For example, the spawn may fail. Recall that Condor takes the responsibility to find an idle machine and starts a PVM daemon on it. During the time between when the PVM daemon was started and the message received by master program, the owner of the selected machine can possibly reclaim it. If a "host add" message was queued waiting for the master program to process other messages, a failure for spawn becomes more likely.

The master program then sends out another request for a new host if the number of remaining subproblems is at least twice as many as the number of workers. The reason for not always asking for new host is that the overhead associated with spawning processes and initializing new workers is significant. Spawning a new process is not handled asynchronously by Condor-PVM. While a spawn request is processed, the master is blocked. The time to spawn a new process usually takes several seconds. Therefore if the number of subproblems in the work pool drops to a point close to the number of workers, the master will not ask for more hosts. This implementation guarantees that only the top 50% "promising" subproblems considered by the program can be selected for evaluation. Furthermore, when the branch-and-bound algorithm eventually converges, this implementation prevents the program from asking for excess hosts. However, the program must be careful to ensure that when the ratio of number of remaining subproblems to number of hosts becomes bigger than 2, the master restarts requesting hosts.

Solution Message.. If a received message contains a solution returned by a worker, the master will permanently remove the corresponding subproblem from the work pool that was marked before. It then updates the work pool using the received LP solutions. After that, the master selects one subproblem from the work pool and sends it to the worker that sent the solution message. The subproblem is marked and stays in the work pool for failure recovery. Some worker idle time is generated here, but the above policy typically sends subproblems to workers that exploit the previously generated solution.

Host Suspend Message.. This type of messages informs the master that a particular machine has been reclaimed by its owner. If the owner leaves within 10 minutes, the Condor processes running on this machine will resume. We have two choices to deal with this situation. The master program can choose to wait for the solutions from this host or send the subproblem currently being computed in this host to another worker. Choosing to wait may save the overhead involved in solving the subproblem. However the waiting time can be as long as 10 minutes. If the execution time of a FATCOP job is not significantly longer than 10 minutes, waiting for a suspended worker may cause a serious delay for the program. Furthermore, the subproblems selected from the work pool are usually considered "promising". They should be exploited as soon as possible. Therefore, if a "host suspend" message is received, we choose to recover the corresponding subproblems in the work pool right away. This problem then has a chance to be quickly sent to another worker. If the suspended worker resumes later, the master program has to reject the solutions sent by it in order that each subproblem is considered exactly once.

Host Resume Message.. After a host resumes, the master sends a new subproblem to it. Note that the master should reject the first solution message from that worker. The resumed worker picks up in the middle of the LP solve process that was frozen when the host was suspended. After the worker finishes solving the LPs, it sends the solutions back to the master. Since the associated subproblem had been recovered when the host was suspended, these solutions are redundant, hence should be ignored by the master.

Host Delete/ Task Exit Message.. If the master is informed that a host is removed from the parallel virtual machine or a process running on a host is killed, it recovers the corresponding subproblem from the work pool and makes it available to other workers.

3.4.2. Worker program. The worker program first receives an LP model from the master, then sits in an infinite loop to receive messages from the master. The messages from the master consist of the modified bound information about the subproblem P, the optimal basis to speed up the bounding operation, and the branching variable that is used to define the "up" and "down" children $P+$ and $P-$. The worker performs two bounding operations on $P+$ and $P-$ and sends the results back to the master. The worker program is not responsible for exiting its PVM daemon. It will be killed by the master after the stopping criteria is met.

4. Computational experience. A major design goal of FATCOP is fault tolerance, that is, solving MIP problems correctly using opportunistic resources. Another design goal is to make FATCOP adaptive to changes in available resources provided by Condor in order to achieve maximum possible parallelism. Therefore the principal measures we use when evaluating FATCOP are *correctness* of solutions, and *adaptability* to changes in resources. *Execution time* is another important performance measure, but it is affected by many random factors and heavily dependent on the availability of Condor's resources. For example a FATCOP job, that can be finished in one hour at night, may take 2 hours to finish during the day because of the high competition for the resources. We first show how FATCOP uses as many resources as it is able to capture, then show how reliable it is to failures in its environment and conclude this section with numerical results on a variety of test problems from the literature.

4.1. Resource utilization. In Wisconsin's Condor pool there are more than 100 machines in our desired architecture class. Such large amounts of resources make it possible to solve MIP problems with fairly large search trees. However the available resources provided by Condor change as the status of participating machines change. Figure 4.1 demonstrates how FATCOP is able to adapt to Condor's dynamic environment. We submitted a FATCOP job in the early morning. Each time a machine was added or suspended, the program asked Condor for the number of idle machines in our desired machine class. We plot the number of machines used by the FATCOP job and the number of machines available to the job in Figure 4.1. In the figure, time goes along the horizontal axis, and the number of machines is on the vertical axis. The solid line is the number of working machines and dotted line is the number of available machines that includes idle machines and working machines used by our FATCOP job. At the start, there were some idle machines in Condor pool. The job quickly harnessed about 20 machines and eventually collected more than 40 machines with a speed of roughly one new resource every minute. At 8 a.m. it became difficult to acquire new machines and machines were steadily lost during the next four hours. There were some newly available resources at 8:30 and 10:00 (see the peaks of the dotted lines), but they became unavailable again quickly, either reclaimed by owners or scheduled to other Condor users with higher priority. At noon, another group of machines became available and stayed idle for a relatively long time. The FATCOP job acquired some of these additional machines during that time. In general, the number of idle machines in Condor pool had been kept at a very low level during life cycle of the FATCOP job except during the start-up phase. When the number of idle machines stayed high for some time, FATCOP was able to quickly increase the size of its virtual machine. We believe these observations exhibit that FATCOP can utilize

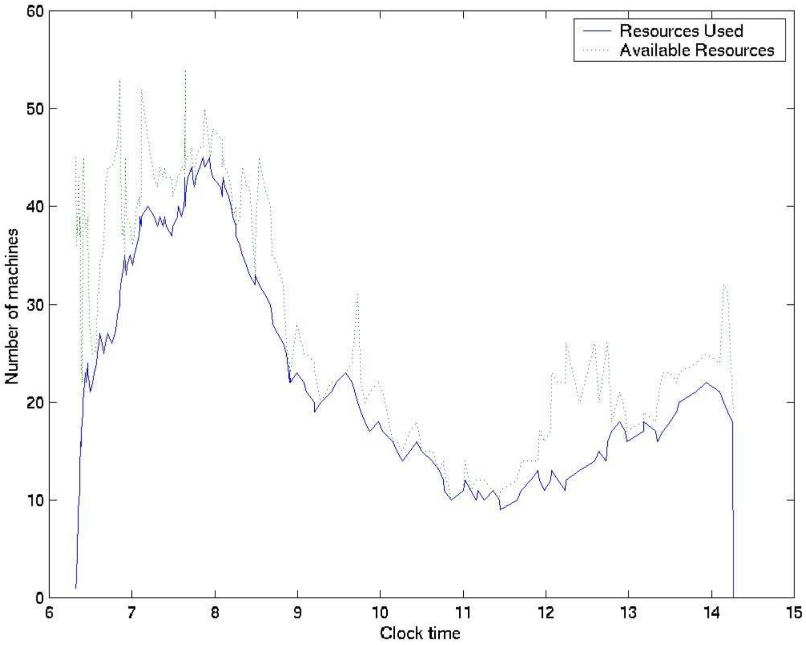

Fig. 4.1. *Resource utilization for one run of FATCOP*

Table 4.1
Average number of machine and suspensions for 4 FATCOP runs

Run	Starting time	Duration	E_{avg}	Number of suspensions
1	07:50	13.5 hrs	32	145
2	12:01	14.9 hrs	29	181
3	16:55	11.1 hrs	40	140
4	21:00	10.1 hrs	49	118

opportunistic resources very well.

We show a FATCOP daily log in Figure 4.2. The darkly shaded area in the foreground is the number of machines used and the lightly shaded area is the number of outstanding resource requests to Condor from this FATCOP job. During the entire day, the number of outstanding requests was always about 10, so Condor would consider assigning machines to the job whenever there were idle machines in Condor's pool. At night, this job was able to use up to 85 machines (the horizontal line represents 75 machines, the vertical line in the figure represent midnight). Note that the Computer Sciences Department at the University of Wisconsin reboots all instructional machines at 3 a.m. every day. This job lost almost all its machines at that time, but it quickly got back the machines after the reboot.

To get more insight about utilization of opportunistic resources by FATCOP, we

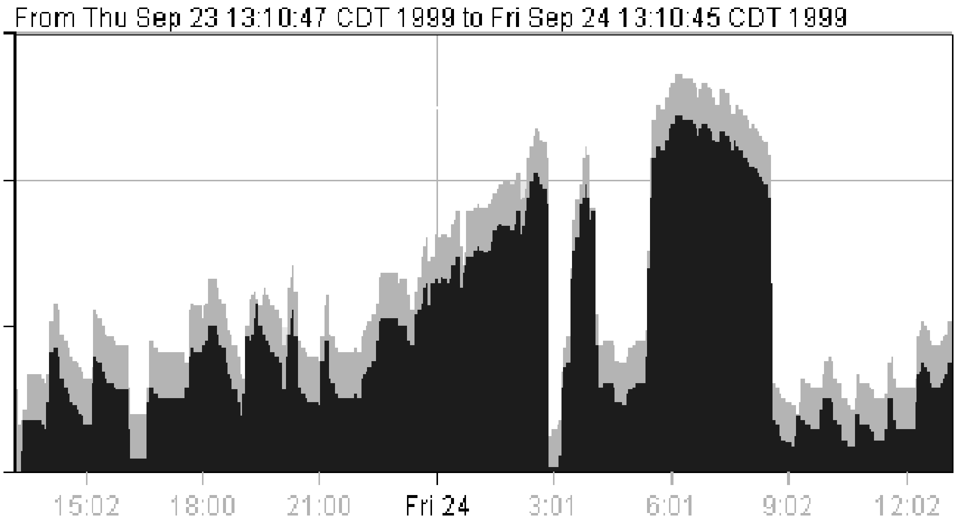

FIG. 4.2. *Daily log for a FATCOP job*

define the average number of machines used by a FATCOP job E_{avg} as:

$$E_{avg} = \frac{\sum_{k=1}^{E_{max}} k\tau_k}{T},$$

where τ_k is the total time when the FATCOP job has k workers, T is the total execution time for the job, E_{max} is the number of available machines in the desired class. We ran 4 replications of a MIP problem. The starting time of these runs was distributed over a day. In Table 4.1 we record the average number of machines the FATCOP job was able to use and number of machines suspended during each run. The first value shows how much parallelism the FATCOP job can achieve and the second value indicates how much additional work had to be done. In general the number of machines used by FATCOP is quite satisfactory. At run 4, this value is as high as 49 implying that on average FATCOP used close to 50% of the total machines in our desired class. However, the values vary greatly due to the different status of the Condor pool during different runs. In working hours it is hard to acquire machines because many of them are used by owners. After working hours and during the weekend, only other Condor users are our major competitors. As expected FATCOP lost machines frequently during the daytime. However during the runs at night FATCOP also lost many machines. It is not surprising to see this, because the more machines FATCOP was using, the more likely it would lose some of them to other Condor users.

4.2. Fault tolerance. FATCOP has been tested on the problems from MIPLIB3.0. There are 59 problems in the test set with different size and difficulty. The FATCOP sequential and parallel solver solved 41 and 44 problems respectively with default options, accounting for 70% and 75% of the total test problems. Our computational results show that for problems that can be solved in minutes by the FATCOP sequential solver, the parallel solver may take longer to solve them. In solving these problems, FATCOP spent a large portion of the total solution time on spawning and initializing workers. It suggests that it is only beneficial to use FATCOP to solve

TABLE 4.2
Summary of test problems from MIPLIB3.0

Name	#rows	#columns	#nonzeros	#integers
10TEAMS	230	2025	12150	1800
AIR04	823	8904	72965	8904
AIR05	426	7195	52121	7195
DANOINT	664	521	3232	56
FIBER	363	1298	2944	1254
L1521AV	97	1989	9922	1989
MODGLOB	291	422	968	98
PK1	45	86	915	55
PP08ACUTS	246	240	839	64
QIU	1192	840	3432	48
ROUT	291	556	2431	315
VPM2	234	378	917	168

TABLE 4.3
Results obtained by the FATCOP sequential solver

Name	Solution gap(%)	Proven optimal?	Execution time	Nodes
10TEAMS	0	yes	23.2 hrs	163,130
AIR04	0	yes	9.6 hrs	4,606
AIR05	0	yes	29.3 hrs	23,512
DANOINT	0	no	48.0 hrs	640,300
FIBER	0	yes	4.7 hrs	172,788
L1521AV	0	yes	1.9 hrs	17,846
MODGLOB	0	no	48.0 hrs	12,459,812
PK1	0	yes	1.6 hrs	475,976
PP08ACUTS	0	yes	6.4 hrs	3,469,870
QIU	0	yes	1.2 hrs	16,448
ROUT	3	no	48.0 hrs	2,582,441
VPM2	0	yes	2.4 hrs	926,740

MIPs with large search trees and/or complex LP bounding operations. In this section we report the computational results for the problems that can not be solved by the FATCOP sequential solver in an hour, but solvable by the parallel solver. The size of these problems are shown in Table 4.2.

FATCOP was configured to use default branching and node selection strategies, i.e. pseudocost branching and best-estimation based mixed searching strategy (mixed strategy 2). We let FATCOP switch to best-bound search after the best integer solution had remained unchanged for 10,000 node evaluations. MIPLIB files do not provide branching priorities, so priority branching is irrelevant to problems from MIPLIB. CPLEX was used as the primary LP solver. Due to licensing limitations, when the maximum number of CPLEX copies was reached, SOPLEX was called to perform bounding operations in the workers.

We first tried to solve the problems in Table 4.2 using the FATCOP sequential solver on a SUN Sparc SOLARIS2.6 machine. Each run was limited by 48 hours. We present the results in Table 4.3. The first column in the table shows the relative

TABLE 4.4
Average results obtained by the FATCOP parallel Condor-PVM solver for 3 replications (all instances were solved to optimality)

Name	Average execution time	Average tree size	E_{avg}	Average # of suspensions	Speedup
10TEAMS	1.9 hrs	201,553	20	256	12.2
AIR04	56.8 mins	5,464	11	94	10.1
AIR05	2.0 hrs	16,842	41	55	14.7
DANOINT	24.2 hrs	3,451,676	22	239	-
FIBER	1.1 hrs	174,489	42	25	4.3
L1521AV	24.6 mins	12,266	35	32	4.6
MODGLOB	44.8 hrs	704,056,478	18	512	-
PK1	14.5 mins	8554.886	28	12	6.6
PP08ACUTS	2.8 hrs	5,001,600	19	122	2.3
QIU	22.2 mins	16,376	27	9	3.2
ROUT	12.3 hrs	22,851,706	29	295	-
VPM2	46.7 mins	1,014,943	15	59	3.1

difference between the best solution found by the FATCOP sequential solver and the known optimal solution. If the optimal solution is found, column 2 shows whether the solution is a proven optimal solution or not. Execution time in column 3 is clock elapsed time. Tree size at the time when the program was terminated is given in column 4.

The test problems were then solved by the FATCOP parallel Condor-PVM solver. The number of problems N generated in the initial stage was set to 20. At the beginning the master sends 10 requests for new hosts to Condor. FATCOP implements an asynchronous algorithm, hence communication may occur at any time and is unpredictable. Furthermore, the number of workers in the life cycle of a FATCOP job keeps changing so that the branch-and-bound process may not follow the same path for different executions. Our experiments show that the search trees were almost never expanded in the same order for a given problem. This feature often leads FATCOP to different execution times. We ran 3 replications for each problem. For all runs, FATCOP found provable optimal solutions for the test problems. We report the average execution time, the number of evaluated nodes in the search tree, resource utilization and resource losses in Table 4.4. During all runs, FATCOP lost some workers, but the program returned correct solutions. Therefore FATCOP was tolerant to the resource retreats in our experiments.

The FATCOP parallel solver found provable optimal solutions for all the test problems. However, the sequential solver failed to prove optimality on *danoint*, *modglob* and *rout*. For the problems solved by both, the parallel solver achieved reasonable speedup over the sequential solver. Run times for these problems were reduced by factors between $2.3 - 14.7$.

We observed from Table 4.4 that many test problems exhibit strong search anomalies. *pp08aCUTS* and *pk1* have much larger search trees when solved by the parallel solver. On the other hand *10teams*, *air05* and *l1521av* have smaller search trees. While such search anomalies are well-known for parallel brand-and-bound, the highly non-deterministic nature of the Condor opportunistic environment can also lead to even more varying search patterns.

TABLE 4.5
Results for VOD and PROD problems

Name	#rows	#columns	#nonzeros	#integers	time	E_{avg}
VOD1	107	306	1207	303	7.5 mins	11
VOD2	715	1513	7316	1510	5.2 mins	18
PROD1	208	251	5350	149	1.2 hrs	25
PROD2	211	301	10501	200	10.8 hrs	35

TABLE 4.6
Comparison of with GA and without GA for PROD1

Relative Gap %	Nodes with GA	Nodes without GA
20	1,178	864,448
15	2,230	$> 1,000,000$
10	6,506	$> 1,000,000$
5	37,224	$> 1,000,000$
0	137,866	$> 1,000,000$

A remarkable example in this test set is *modglob*. The FATCOP sequential solver could not find a provable optimal solution for this problem in 48 hours, while it was solved to optimality by the parallel solver in 44.8 hours. It ran over two Computer Sciences Department daily reboot periods, used 18 machines on average and had 512 machines suspended during the run. To test fault tolerance of the master, we let FATCOP write the work pool information to disk every 100,000 node evaluations. We interrupted the job once (to simulate a master failure) and re-submitted the problem to Condor. FATCOP then started from where the work pool information was last recorded. This indicates that FATCOP is tolerant to both worker and master failures.

4.3. Application test problems. FATCOP was used to solve two classes of problems arising from marketing and electronic data delivery. One class of problems, *VOD*, are applications to video-on-demand system design [10, 13]. The other class of problems *PROD* are applications to product design [26]. Two problems from each application were formulated as GAMS models. The size of the problem instances and results found by FATCOP are reported in Table 4.5. Execution time is clock elapsed time, and does not include GAMS compilation and solution report time. User defined priorities are provided in *VOD2*. It turns out that this information is critical to solve this problem in a reasonable amount of time [13]. For PROD problems, good integer feasible solutions were found using a Genetic Algorithm (GA) [3] first, and these solutions were delivered to FATCOP as incumbent values. Provable optimal solutions were found for all the problem instances in Table 4.5.

In practice many problem specific heuristics are effective for finding near-optimal solutions quickly. Marrying the branch-and-bound algorithm with such heuristics can help both heuristic procedures and a branch-and-bound algorithm. For example, heuristics may identify good integer feasible solutions for the early stage of the branch-and-bound process, decreasing overall solution time. On the other hand, the quality of solutions found by heuristic procedures may be measured by the (lower-bounding) branch-and-bound algorithm. FATCOP can use problem specific knowledge to increase its performance. Based on interfaces defined by FATCOP, users can write their own programs to round an integer infeasible solution, improve an integer feasible solution and perform operations such as identifying good solutions or adding problem specific cutting planes at the root node. These user defined programs are dynamically linked to the solver at run time and can be invoked by turning on appropriate solver options. For product design problems, good solutions found by the GA made the problems solvable. We performed a set of experiments on *PROD1* by turning on and off the GA program at the root node. We limited the number of node evaluations to 1,000,000. The computational results are given in Table 4.6. Without the GA, FATCOP cannot reduce the optimality gap below 15% in the given number of node evaluations. However, with good solution found by the GA at the root node, FATCOP is able to prove optimality for this problem in around 1.2 hours.

4.4. Set partitioning. An encouraging side-benefit to the opportunistic environment is the fact that the parallel machine becomes much more powerful as more and newer machines are added into the pool. This is in sharp contrast to the case where a standard supercomputer is obtained, unless a new machine is purchased every six months. Thus, the computational results that we generated six months ago become much easier to obtain today. As an example of this type of progress, we cite two new set partitioning problems arising in transportation scheduling that were solved using the FATCOP solver after the original paper was written.

The first problem, t0415, was started at 18:37 and finished proving optimality two days later (around 31 hours later) at 01:11. The problem has 1518 constraints and 7254 binary variables. During this time, a maximum of 114 workers were used, and total of 1795 hours of worker CPU time. Note that many of the machines used in this run were claimed during the work day by their respective owners, but regardless of this, each of the 114 workers were processing linear programs for around 16 hours of the total run, processing 531,707 nodes in the tree in total. FATCOP used its default strategy in this run. Many of the processors used in this run were not installed when this paper was first released, but the same code was able to exploit them as soon as they became part of the Condor pool.

The second problem, t0416, seems at first inspection, very similar. It has 1771 constraints and 9345 binary variables. However, this problem proved substantially more difficult to solve, being started on Sep 16 at 19:49 and finishing on Sep 26 at 22:48. In solving this problem, three checkpoints were utilized, each time resetting FATCOP's options to switch between best-bound branching and depth-first search after each solve. Thus, the actual time the FATCOP program was running was around 97 hours, instead of the possible 242 hours between its start and finish times. This was to control the size of the unexplored tree more carefully than our default strategy allows. The maximum number of workers used was 118, average number of workers throughout the run was 78, with a total of 5409 hours of worker CPU time (out of a total worker uptime of 7544 hours). In this run, 1,081,271 nodes in the tree were processed in total. Clearly, the second run is less impressive in terms of efficiency and the need for user intervention. However, it should also be noted that we have been unable to solve either test problem using the currently available commercial solvers for mixed integer linear programs.

5. Summary and future work. In this paper, we provide a parallel branch-and-bound implementation for MIPs using distributed privately owned workstations. The solver, FATCOP, is designed in the master-worker paradigm to deal with different types of failures in an opportunistic environment with the help of Condor, a resource management system. To harness the available computing power as much as possible, FATCOP uses a greedy strategy to acquire machines. FATCOP is built upon Condor-PVM and SOPLEX, both of which are freely available.

FATCOP has successfully solved real life MIP problems such as the applications to video-on-demand system design and product design. It was also tested on a set of standard test problems from MIPLIB. Our computational results show that the solver works correctly in the opportunistic environment, and is able to utilize opportunistic resources efficiently. A reasonable speedup was achieved on long running MIPs over its sequential counterpart.

Our future work include strengthening parallel branch-and-bound procedures with more cutting planes such as flow cover cuts and disjunctive cuts and investigating how much a worker processor should do before returning results to the master. Some of these issues have already been addressed in subsequent work [8].

6. Acknowledgement. The authors are grateful to Miron Livny and Michael Yoder for advice and assistance in using Condor and the Condor-PVM environment. They also wish to thank Jeff Linderoth for his very insightful comments.

REFERENCES

[1] K. Anstreicher, N. Brixius, J.-P. Goux, and J. Linderoth. Solving large quadratic assignment problems on computational grids. Technical report, Argonne National Laboratory, Oct 2000.
[2] M. Avriel and B. Golany. *Mathematical Programming for Industrial Engineers*. Marcel Dekker, 1996.
[3] P. V. Balakrishnan and V. S. Jacob. Genetic Algorithms for Product Design. *Management Science*, 42:1105-1117, 1996.
[4] M. Benichou and J. M. Gauthier. Experiments in Mixed-Integer Linear Programming. *Management Science*, 20(5):736–773, 1974.
[5] R. E. Bixby, S. Ceria, C. M. McZeal and M.W.P. Savelsbergh. MIPLIB 3.0. *http://www.caam.rice.edu/~bixby/miplib/miplib.html*.
[6] R. E. Bixby, W. Cook, A. Cox and E. K. Lee. Parallel Mixed Integer Programming. *Center for Research on Parallel Computing Research Monograph CRPC-TR95554*, 1995.
[7] A. Brooke, D. Kendrick and A. Meeraus. *GAMS: A User's Guide*. The Scientific Press, South San Francisco, CA, 1988.
[8] Q. Chen, M. C. Ferris, and J. T. Linderoth. FATCOP 2.0: Advanced features in an opportunistic mixed integer programming solver. *Annals of Operations Research, forthcoming*, 2000.
[9] CPLEX Optimizer. *http://www.cplex.com/*
[10] D. L. Eager, M. C. Ferris and M. K. Vernon. "Optimized Regional Caching for On-Demand Data Delivery," *Multimedia Computing and Networking, Proceedings of SPIE*, 3654:301–316, Bellingham, Washington 1999.
[11] J. Eckstein. Parallel Branch-and-Bound Algorithms for General Mixed Integer Programming on the CM-5. *SIAM J. Optimization*, 4(4):794–814, 1994.
[12] D. H. Epema and M. Livny. A Worldwide Flock of Condors: Load Sharing among Workstation Clusters. *Journal on Future Generations of Computer System*, 12, 1996.
[13] M. C. Ferris and R. R. Meyer. Models and Solution for On-Demand Data Delivery Problems. To appear in *Approximation and Complexity in Numerical Optimization: Continuous and Discrete Problems*, Kluwer Academic Publishers, 1999.
[14] A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek and V. S. Sunderam. *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing*. The MIT Press, Cambridge, Massachusetts, 1994.
[15] B. Gendron and T. G. Crainic. Parallel Branch-and-Bound Algorithms: Survey and Synthesis. *Operations Research*, 42(6):1042–1060, 1994.
[16] J.-P. Goux, S. Kulkarni, J. Linderoth, and M. E. Yoder. An enabling framework for master-worker applications on the computational grid. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing*, 2000.
[17] W. Gropp, E. Lusk and A. Skjellum. *Using MPI : Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, Cambridge, Massachusetts, 1994.
[18] Z. Gu, G. L. Nemhauser and M. W. P. Savelsbergh. Lifted Cover Inequalities for 0-1 Integer Programs: Computation. *INFORMS Journal on Computing*, 10:427–437, 1995.
[19] J. Linderoth and M. W. P. Savelsbergh. A Computational Study of Branch and Bound Search

Strategies for Mixed Integer Programming. *INFORMS Journal on Computing*, 11:173–187, 1999.

[20] M. J. Litzkow and M. Livny. Condor - A Hunter of Idle Workstations. *in Proceedings of the 8th International Conference on Distributed Computing Systems, Washington, District of Columbia, IEEE Computer Society Press*, 108-111, 1988.

[21] J. L. Nazareth. *Computer Solution of Linear Programs.* Oxford University Press, 1987.

[22] G. L. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization*. Wiley Interscience, 1989.

[23] J. Pruyne and M. Livny. Providing Resource Management Services to Parallel Applications. *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, May, 1994.

[24] E. A. Pruul and G. L. Nemhauser. Branch-and-Bound and Parallel Computation: a Historical Note. *Oper. Res. Letters*, 7:65-69, 1988.

[25] M. W. P. Savelsbergh. Preprocessing and Probing for Mixed Integer Programming Problems. *ORSA J. on Computing*, 6: 445–454, 1994.

[26] L. Shi, S. Ólafsson and Q. Chen. An Optimization Framework for Product Design. submitted to *Management Science*, 1998.

[27] S. J. Wright. Solving optimization problems on computational grids. *Optima*, 2000.

[28] R. Wunderling. Documentation of the SOPLEX Library. *http://www.zib.de/Optimization/Software/Soplex/.*