PARINO – An Extendable Framework for Solving Mixed Integer Programs in Parallel

Kalyan Perumalla (kalyan@cc.gatech.edu)
Martin Savelsbergh (mwps@isye.gatech.edu)
Umakishore Ramachandran (rama@cc.gatech.edu)

College of Computing and
School of Industrial Systems and Engineering
Georgia Institute of Technology
Atlanta, GA 30332

GIT-CC-97-07
March 03, 1997

ABSTRACT

This report documents a framework called PARINO that we have developed for solving large mixed integer programs (MIPs) in parallel. PARINO combines portability and efficiency with flexibility. It is written in C++, and is portable across any message passing platforms. The flexibility of the framework is a result of mapping the MIP computation to an *entity-FSM* paradigm, where it is expressed in terms of the interactions among several distributed active objects. Thus, it is possible to incrementally enhance the functionality of PARINO by incorporating new dynamic objects into the framework. We have used this feature to augment the core parallel MIP computation in PARINO with a simple distributed cut management system, which is again expressed in terms of the entity-FSM paradigm. PARINO is currently operational. It has been tested on an 8-node IBM SP2 multicomputer, and new feature additions and enhancements are being continually made to it.

PARINO – An Extendable Framework for Solving Mixed Integer Programs in Parallel

Kalyan Perumalla* Martin Savelsbergh† Umakishore Ramachandran*

1 Introduction

Mixed integer programming deals with problems of maximizing or minimizing a function of many variables subject to linear inequality and equality constraints and integrality restrictions on some or all of the variables. A remarkably rich variety of problems can be represented by such models.

An important and widespread area of application concerns the management and efficient use of scarce resources to increase productivity. These applications include operational problems such as the distribution of goods, production scheduling and machine sequencing, but also more tactical and strategical problems such as capital budgeting, facility location and the design of communication and transportation networks.

In the past decade, research on linear programming based branch-and-bound algorithms for mixed integer programming has focused on improving the linear programming approximation. Reformulation techniques have been developed that have proven to be quite successful. In combination with the availability of faster and more robust linear programming methods (due to the development and maturation of interior point methods as well as to enhancements to the simplex method) and the availability of increasingly powerful hardware platforms providing high speed processors and large amounts of internal memory, these methods have allowed the solution of mixed integer programs of sizes and with speeds that did not seem possible ten years ago. There are many reasons to believe that parallel computing may provide the opportunity for even greater advances. Related work in parallel/distributed mixed integer program solution appears in [1, 2, 3, 4, 5, 6].

PARINO (Parallel Mixed Integer Optimizer) is a *parallel* software system that we have developed for solving mixed integer programs (MIPs). PARINO is an evolving system,

*College of Computing, Georgia Institute of Technology
†Industrial and Systems Engineering, Georgia Institute of Technology

with new features and enhancements added to it continually. The heart of PARINO is a sophisticated linear programming based branch-and-cut algorithm in a parallel/distributed setting. It is designed so that it is straight-forward to incorporate many of the state-of-the-art mixed integer programming techniques.

PARINO has been designed to work on any message-passing system, such as a network of workstations, and does not assume the availability of any sophisticated underlying inter-processor communication or shared memory system. PARINO carefully exploits the weak synchronization requirements that are inherent in a branch-and-cut algorithm. For example, in distributed active set management and in distributed cut management, the required communication is explicitly controlled, as opposed to leaving the control to a generalized native or emulated shared-memory system.

PARINO uses a special *entity-FSM* architecture as the underlying message passing system that allows the objects that can be identified in the MIP computation domain to be mapped naturally and elegantly to equivalent *entities* and their threads of computation. The use of the entity-FSM architecture also allows for easy extension of PARINO's functionality and for easy tuning of the system for optimal performance when it is ported to different computation platforms and communication subsystems. In addition, as a natural fallout, PARINO allows the incorporation of problem-dependent runtime control (*online steering*) of the computation by varying the various key parameters at runtime, thereby varying the strategies of computation and search and the appropriate use of various heuristics.

2 *Entity-FSM* Architecture

To facilitate extendability and portability without sacrificing efficiency, the PARINO system is built using an efficient and portable class library called NAYLAK (see [9] for an overview). The NAYLAK system supports a world view based on the concept of *active entities* for distributed computation. It provides a higher software interface layer above traditional process-based message passing systems. Figure 1 illustrates the NAYLAK system architecture.

Traditionally, bulky processes (usually UNIX processes) are the units which communicate by exchanging messages. Except for some relatively small or medium complexity applications, process-level messaging modularity cannot adequately scale with the application complexity. In NAYLAK, an additional layer is added over this basic process-level message-passing by introducing the concept of lighter units called *entities*. Each process, thus, consists of a set of entities, and message-passing is performed at the entity level rather than process level. Thus, entities send messages to other entities, as opposed to processes sending messages to other processes.

Another feature lacking in traditional message-passing interfaces, such as PVM, is that of *context* maintenance. If a given computation consists of several stages, with some message-passing/synchronization occuring between the stages, then either barriers or artificial message tags are used to realize the multi-stage computation. In NAYLAK, the concept of a finite state machine (FSM) is supported to directly facilitate multi-stage computation. An FSM is an arbitrary graph of *states*, where each state is a set of statements that are executed indivisibly (atomically). Every FSM is associated with a single entity, called its owner. An entity can have zero or more FSMs running for it. Each FSM, in effect, is a thread of computation, and each FSM state is a unit of computation. Each FSM state dynamically designates its next state. Since FSM states are executed indivisibly (atomically), no synchronization is required for access to common data across FSM's and FSM states[1]. FSM's can be started, paused, resumed and terminated dynamically.

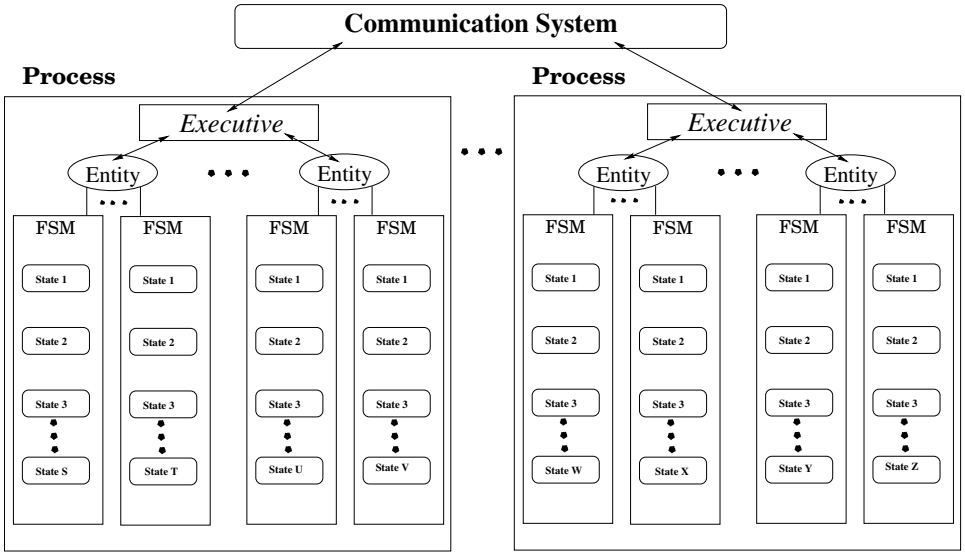

Figure 1: NAYLAK System Architecture

Entities can be created and terminated dynamically. The mapping of entities to

[1]This is one of the main differentiating features from other systems that combine conventional message passing with threads; in such threads-based systems, explicit synchronization primitives, such as mutexes or semaphores, should be used to ensure the atomic nature of the most common sequence of operations, whereas such operation is provided by default in this entity-FSM architecture. This greatly eases the burden on the programmer, and helps in a natural flow of control and data.

UNIX-level processes, and the mapping of the processes to the parallel processors can be controlled at runtime. Entities can create or terminate other entities dynamically. The creation and termination can be performed synchronously or asynchronously.

Although abstractions usually entail slight overheads, the implementation of the NAYLAK system has been carefully designed in order to minimize the overhead imposed by the layering. As compared to the execution time of the procedures involved per unit of computation in PARINO, the overhead incurred (which is mostly from extra memory-copying instructions) due to the NAYLAK runtime system is insignificant.

The NAYLAK system is portable across any message-passing system. Its software interface is in C++. Its object-oriented design facilitates its portability, with few and isolated modifications for porting. The entity-FSM architecture has been found to be very useful due to the following features:

- it facilitates development of distributed algorithms
- it allows system adjustment to improve performance on a platform-specific basis
- it is portable
- it is extendable.

3 PARINO System Architecture

The heart of PARINO is a linear programming based branch-and-bound algorithm. A brief summary of the algorithm is given below (see [7, 8] for additional information).

Branch-and-bound Node k of a branch-and-bound tree corresponds to a subproblem MIP(k) obtained by adding constraints to the original problem MIP(0). Frequently, these constraints are simply tighter bounds for the integer variables which, in the case of binary variables, implies that the variable is fixed to either 0 or 1. Any feasible solution to MIP provides a lower bound. Let z_{best} be the value of the greatest lower bound over all available feasible solutions. At node k of the tree, we solve the LP relaxation LP(k) of MIP(k). We assume that LP(k) is feasible and bounded and its optimal value is $z(k)$. If the optimal solution $x(k)$ found to LP(k) happens to satisfy the integrality constraints, then $x(k)$ is also optimal to MIP(k), in which case, if $z(k) > z_{best}$, we update z_{best}. We can forget about node k. Otherwise, if $x(k)$ does not satisfy all of the integrality constraints, there are two possibilities. If $z(k) \leq z_{best}$, an optimal solution to MIP(k) cannot improve on z_{best}. Again we can forget about node k. On the other hand, if $z(k) > z_{best}$, MIP(k) requires further exploration. This is done by branching: creating at least two new subproblems (descendants) of MIP(k), MIP($k(i)$), $i = 1, \ldots, q, q \geq 2$.

Each subproblem is formed by adding constraints to MIP(k). Each feasible solution to MIP(k) must be feasible to at least one descendant and, conversely, each feasible solution to a descendant must be feasible to MIP(k). Moreover the solution $x(k)$ must not be feasible to any of the descendants. A simple realization of these requirements is to select a variable x_j, $j \leq p$, such that $x_j(k)$ is not integral, and to create two descendants; in one of these we add the constraint $x_j \leq \lfloor x_j(k) \rfloor$ and in the other $x_j \geq \lceil x_j(k) \rceil$. Node k is now replaced by its descendants and the size of the tree grows.

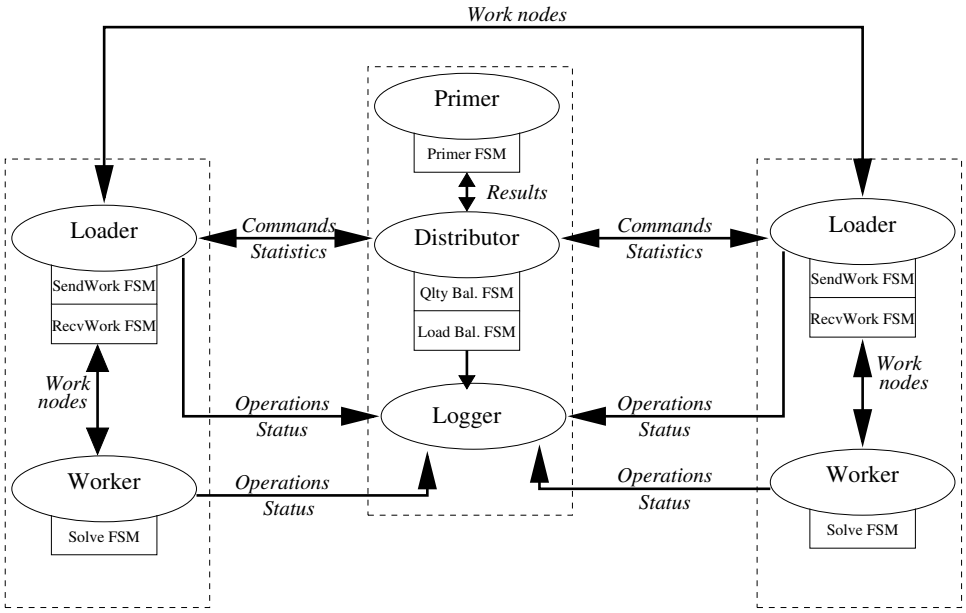

Figure 2: Illustration of the PARINO system architecture. Dotted regions represent UNIX processes, ovals represent NAYLAK entities, boxes represent FSMs and arrows represent relevant communication.

System Entities

The PARINO system consists of several entities interacting with each other. Figure 2 depicts the various entities, their FSMs and the interactions among them. Each entity is responsible for a clearly defined set of operations. The types of entities include *primer*, *distributor*, *loader*, *worker* and *logger* among others. Exactly one instance of the primer entity type and one instance of the distributor entity type are created for a given run.

Exactly one instance of the loader entity is created per processor, and exactly one worker entity is created per loader entity.

The primer entity is responsible for initially creating all the entities, reading the initial problem, waiting for results, and printing the results. The distributor is responsible for the coordination of load balancing and quality balancing across the distributed active sets (as will be explained in later sections). Each loader entity is responsible for the maintenance of the active set at each processor. Loaders act on commands from the distributors to achieve load balancing and quality balancing. The distributor, in addition, is responsible for termination-detection (end of parallel branch and bound search). The duties of a worker are simply to request a node for computation from its loader, evaluate the node, generate new nodes if necessary, report results back to the loader and repeat the entire process again. The loaders and the distributor together deal with the distributed active set management. There is no centralized active set (as will be explained shortly). As an example to illustrate the working of each entity, Figure 3 depicts a simplified version of the state transition diagram of the `SolveFSM` of a *worker* entity.

4 Active Set Management

The set of unevaluated nodes that are generated during a branch-and-bound operation is called the *active set*. Several techniques exist for the effective management of the active set of nodes in a branch-and-bound search in a parallel/distributed setting. The alternatives range from fully centralized set, through quasi-distributed set, to fully distributed set.

4.1 Shared Memory Vs. Message Passing

Several issues arise with each of the alternatives to active set management. With a centralized set implementation, the obvious concern is contention. A less obvious issue exists when the centralized set is implemented using shared-memory. Typically, the shared-memory implementations guarantee some consistency properties that are usually more generalized and stricter than that are sufficient for the management of the active set in parallel branch-and-bound. Although it is not clear how the shared memory implementation interacts in general with the access patterns of the active set, it intuitively appears that a more direct control of the required inter-process communication by the processes will result in better performance and lesser communication and latencies between computations. This effect can be more pronounced when the shared-memory is *emulated* over a message-passing environment (eg., TreadMarks), in which unnecessary communication is more expensive than in a bus-based shared-memory system. The underlying reason for the relaxed synchronization requirements is that it is algorithmically

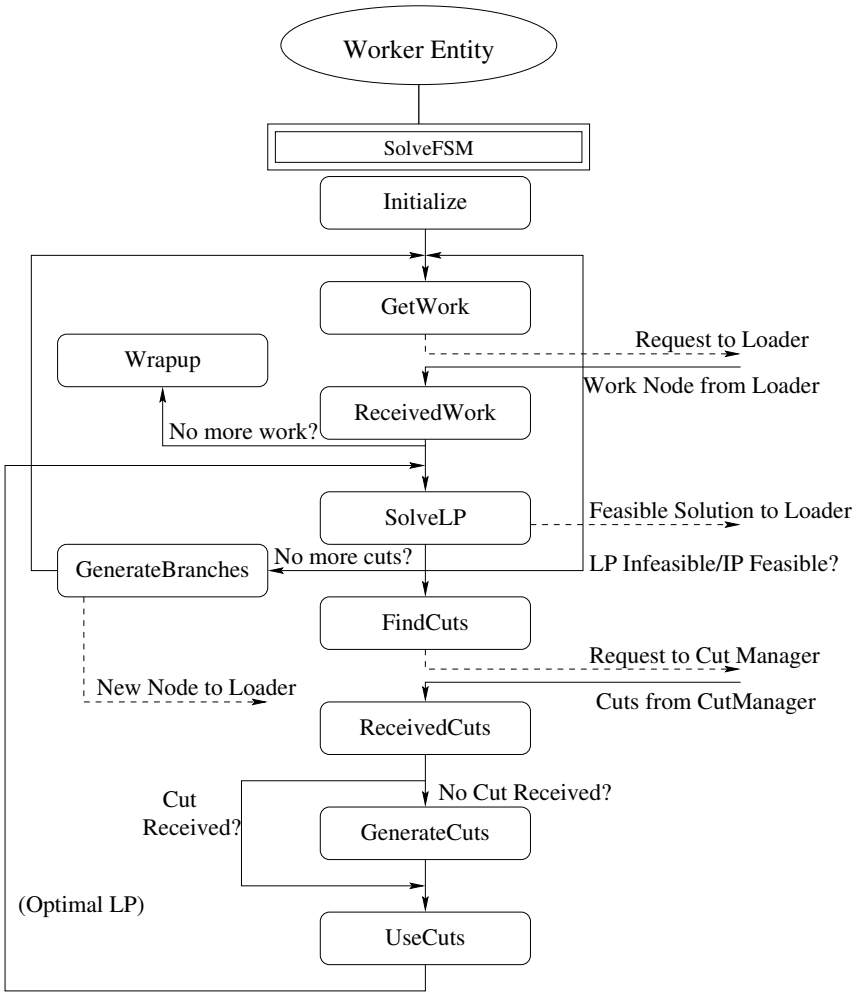

Figure 3: Illustration of a simplified version of the `SolveFSM` of a *worker* entity.

not necessary to work on the node with best bound.

In a fully distributed active set implementation, each processor holds its (partial) active set, and the union of these partial sets can be seen as equivalent to the global central active set. With such a scheme, it is well known that the issues of load imbalance and quality imbalance have to be taken into consideration.

Load imbalance arises when the number of nodes in the distributed active sets across processors varies significantly. Quality imbalance implies that the closeness of the bounds to the optimal solution varies considerably across processors, thereby resulting in some processors potentially working on unpromising nodes, which is undesirable. However, this scheme using a distributed active set does imply that explicit control of the distribution of the nodes (at appropriate time instants during the search) through explicit message passing among the processors can potentially reduce the amount of communication as compared to a highly generalized shared-memory system that works without sufficient knowledge of the application requirements.

Another important difference between the preceding two extreme alternatives is with respect to scalability. Native (say, bus-based) shared-memory systems typically cannot scale to more than a few dozens of processors. Thus, a centralized active set implementation can potentially be constrained by this limit on the number of processors that can be effectively used.

In the case of a distributed active set scheme, there is no true limit on the number of processors; hence, it is not constrained, provided that the algorithms for load and quality balancing are themselves scalable with respect to the number of processors.

The preceding observations also hold for other distributed data structures used in parallel/distributed MIP, such as cuts and pseudo-costs, that can be shared across processors, but do not have any strong consistency requirements.

With the preceding observations, and also considering the fact that message-passing systems, such as network of workstations, are ubiquitously available, and less expensive than shared-memory systems, it was decided to implement a decentralized active set scheme in the PARINO system. The quantity and quality balancing algorithms are described in the following sections. Similarly, in a separate report, we present a fully distributed *cut management* scheme that is designed and developed with parameters to provide a range of alternatives from fully centralized to fully distributed cut databases.

4.2 Implementation

Initially, the first loader's active set contains the root node, and the active sets of the other loaders are empty. The distributor asynchronously keeps track of the current levels of loads and qualities of the active sets at each loader. The load or quantity of an active set is defined as the number of nodes in its active set. The quality of an

active set is defined as the average of the bounds of the top k nodes — $k = 5$ in our current implementation. The loaders periodically — every one second in the current implementation — send updates of their current loads and qualities to the distributor. This is an example of the use of the weak synchronization and consistency requirements inherent in various aspects of parallel mixed integer programming and parallel branch and bound computations, since we can control the communication without influencing the correctness of underlying algorithm.

Quantity balancing

Whenever a worker requests its loader for a node from the active set and the loader finds its active set empty, the loader sends an *out-of-work* message to the distributor entity and waits for a response. The distributor then arranges for some unevaluated nodes to be sent from an appropriately selected loader to this loader, as explained in detail below. The distributor never actually sends or receives any nodes – it only directs the loaders to exchange nodes appropriately. The "Quantity-balancing FSM" of the distributor acts on out-of-work messages from the loaders. All out-of-work messages are collected in a FIFO queue.

Suppose the first request in the queue is from L_i. Let L_j be the loader with the maximum load (based on the most recent information available to the distributor). The distributor sends a message to L_j asking it to send part of its active set to L_i, and waits for a response from L_j. Loader L_j responds with a success status if and when it has successfully completed the transfer of part of its active set to loader L_i; if L_j could not transfer any nodes at all (which is possible if L_j had just run out of work itself), it responds with a failure status. If the response indicates success, then the distributor dequeues the out-of-work request from L_i from its queue, and sends the loader L_i a success response. Loader L_i is, at that point, guaranteed to have a non-empty active set. The distributor's quantity-balancing FSM then moves on to process the next out-of-work message, if any, from other loaders. If the response from L_j is that of failure, then the distributor does not dequeue the request of L_i, but instead continues executing the preceding steps all over again – this time using the updated information of the quantity levels (which were implied by the failure message).

The distributor uses the same request queue to detect termination of the parallel branch and bound search — this is detected if all the loaders are waiting on the queue.

Potential Enhancements

Some enhancements to the preceding distribution algorithm may be suggested. We will discuss two of them. One might suggest that a loader should not wait to fetch more

nodes until it completely runs out of nodes in its active set; instead, a threshold, k, could be used so that whenever the number of nodes in the active set drops below k, a request for more work should be sent to the distributor. Two issues arise with this approach. First, *thrashing* can occur, where, nodes get moved from one loader to another unnecessarily without ever being evaluated. Another drawback with this approach is that termination detection has to be addressed separately, where as, when $k = 1$, it can be detected as part of the load balancing algorithm. Second, there is no obvious method to find the optimal value for k, because it varies considerably with the nature of the MIP problem being solved and with the interplay of the various heuristics used. One might also suggest that a loader with an empty set should fetch new nodes from the processor with the highest quality as opposed to fetching them from the processor with the highest load. However, the quality balancing algorithm discussed in the next subsection ensures balanced qualities across the processors, so this is not an issue. The statistics collected when solving the various MIP problems demonstrates that the loaders run out of work very infrequently compared to the total number of nodes generated, and hence the computation time potentially gained by prefetching is not significant.

Another sophisticated enhancement would be to perform node-exchanges in parallel if more than one loader is waiting in the queue for work. This optimization is based on the observation that if several loaders run out of work at almost the same time, the basic quantity balancing algorithm described previously processes them sequentially, in the order in which they are added into the FIFO request queue. Instead, they could be processed in parallel. This modification entails more bookkeeping and a slight increase in the complexity in coding. It is currently unclear if such an optimization may fetch significant gains in our current experiments, which are based on relatively small number of processors. However, since such an optimization never adversely harms the performance of our basic algorithm, it is possible to incorporate this optimization when porting to configurations with large number of processors.

Quality balancing

As mentioned earlier, all loaders periodically report the load and quality levels of their respective (local) active sets to the distributor. Thus, at any given instant, the distributor possesses an approximation to the actual levels at each loader. A "quality-balancing FSM" of the distributor periodically checks if the quality levels of the active sets (as given by the approximations) are relatively close together, detects any wide disparities, and directs the loaders to exchange work units appropriately to smooth out the differences.

More precisely, the quality-balancing FSM periodically (every 2 seconds, in our current implementation) computes the minimum and maximum among the quality levels of the loaders. Suppose the quality level, q_{min} at loader L_{min}, is the minimum, and the

quality level, q_{max} at loader L_{max}, is the maximum. If the relative difference between the two levels is greater than a threshold, α, (i.e., if $\frac{q_{max}-q_{min}}{q_{max}} > \alpha$), then it sends a message to L_{max} to exchange[2] its top few nodes with those of L_{min} — alternate nodes of the top m nodes of L_{max} are exchanged with alternate nodes of the top m nodes of L_{min} ($m = 5$ in the current implementation). Loader L_{max} responds after successfully performing the exchange operation, and the FSM updates the quality and quantity levels to reflect the new values, and repeats the preceding quality-balancing process.

The quality-balancing FSM also takes into account the quantity levels of the active sets as follows. During one of the periodic checks, if the FSM finds that the quantity level of at least one loader is zero (i.e. at least one loader has run out of work), then it temporarily skips the quality-balancing process described previously. This is so that it does not interfere with the quantity-balancing algorithm. In fact, the quality-balancing is skipped temporarily whenever the quantity level of any loader drops below a threshold $\gamma \geq 0$ ($\gamma = 5$ in current implementation).

Choosing α

The parameter α varies with the MIP problem, and it directly affects the approximation of the distributed active set to a centralized active set with respect to best-bound search. A larger value of α results in lesser communication, but higher potential for degenerated search, while a lower value of α results in greater communication, but closer approximation to the global best-bound search. α can be set to any desired value during initialization, by choosing the right value based on the problems size and estimated objective value; if necessary, our implementation allows for tuning this parameter dynamically at runtime.

Our approach for choosing α is based on the following idea. Let z_{IP} denote the optimum value and let $\overline{z}_{IP}$ be an approximation of z_{IP}. Furthermore, let β be an approximation of the average absolute difference between the bound of a node and that of its parent node. This implies that, ignoring the initial and trailing phases of the search, the mutual differences of the qualities among the distributed active sets also is roughly in the order of β. Therefore, we set $\alpha = 2 \times \frac{\beta}{\overline{z}_{IP}}$. In our implementation, we take $\overline{z}_{IP}$ to be the best feasible solution found so far, we continuously update β, and periodically adjust α.

In general, α can be made lower than the value suggested by the preceding methodology; this enforces a tighter proximity of the qualities of the active sets. But, a very low value of α can result in anomolous behavior, where the processors spend more time

[2] *Shuffling* by exchanging, rather than *one-way transfer* from the loader with maximum quality to the one with minimum quality is better for proper mixture of the best bounds across the two active sets because exchanging results in a better averaging when the bounds are relatively close together.

exchanging nodes than in processing the nodes. On the other hand, a very high value of α can result in anomolous behavior as well, where processors spend most of there time working on unimportant nodes.

5 Algorithmic enhancements

Several algorithmic enhancements can be made to augment the basic LP-based branch-and-bound algorithm for mixed integer programming. The enhancements implemented in PARINO include preprocessing, reduced-cost fixing, global reduced-cost fixing and cut generation. In this section, we describe our implementation approaches in PARINO to some of these enhancements (we describe the ones in which parallel/distributed processing issues arise).

5.1 Global Reduced Cost Fixing

Whenever an improved feasible solution (new incumbent value) is found, the LP solution to the original (root) problem is used in the reduced cost fixing algorithm. The variables that are fixed by this operation, if any, can remain fixed in any solution to the MIP problem.

This global reduced cost fixing has been implemented in PARINO as follows. Whenever a feasible solution that is better than the current incumbent is found, the root node is resolved (with some of the variables fixed at known values, and with some globally valid cuts loaded into the LP). Reduced cost fixing is performed on the resulting root node solution. The information about variables that get fixed due to this operation is broadcast to all the loaders. When a loader receives such information, it *prunes* all the nodes in its active set that represent contradictory ranges of branch variable values that have been fixed by the global reduced cost fixing. In addition, all workers incorporate into their LPs the newly known values to the globally fixed variables.

5.2 Cut generation

Cut generation techniques try to tighten the linear programming relaxation of an integer program by generating strong valid inequalities.

A simple distributed cut management service is implemented using a set of entities — *local pool managers*, *shared global pool managers*, *shared pool managers* and a *global pool manager*. The entities and their relations to each other are depicted in Figure 4. The cut management entities are distinct, and complement the entities used in the branch-and-bound. The extendability through incremental addition of entities (and FSMs) using

PARINO is illustrated by this augmentation of the core entities with cut management entities.

6 Portability

PARINO is written entirely in C++. It uses the NAYLAK system for parallel processing and hence is portable with respect to the underlying communication platform of the parallel computing system. In addition, PARINO uses CPLEX, a commercially available optimization package for linear programming, for parts of its bounding operations. As CPLEX is available on a wide variety of platforms, portability on that point is also ensured. Although PARINO currently uses CPLEX as linear programming solver, it can use any other linear programming solver in place of CPLEX.

Furthermore, most features of the PARINO system can be reconfigured or fine tuned appropriately for optimal performance for different computation and communication platforms.

7 Scalability

Based on the performance data from early runs on problems from MIPLIB, we observed that the quantity and quality balancing algorithms, which are the only sections of the framework that require quasi-centralized operation, do not constitute a bottleneck. We believe from the observations that PARINO is scalable with respect to the problem size, and also scales well with respect to the number of processor available. We expect PARINO to provide linear performance even on a 100-processor configuration for large problems. We plan to perform more exhaustive and extensive study of the solver's performance data to confirm this hypothesis.

8 Future Research

We are currently working on using the extendable framework of PARINO to incorporate many advanced techniques in mixed integer program solution (both well known, as well as state-of-the-art), such as:

- primal heuristics
- improved branching
- distributed cut management

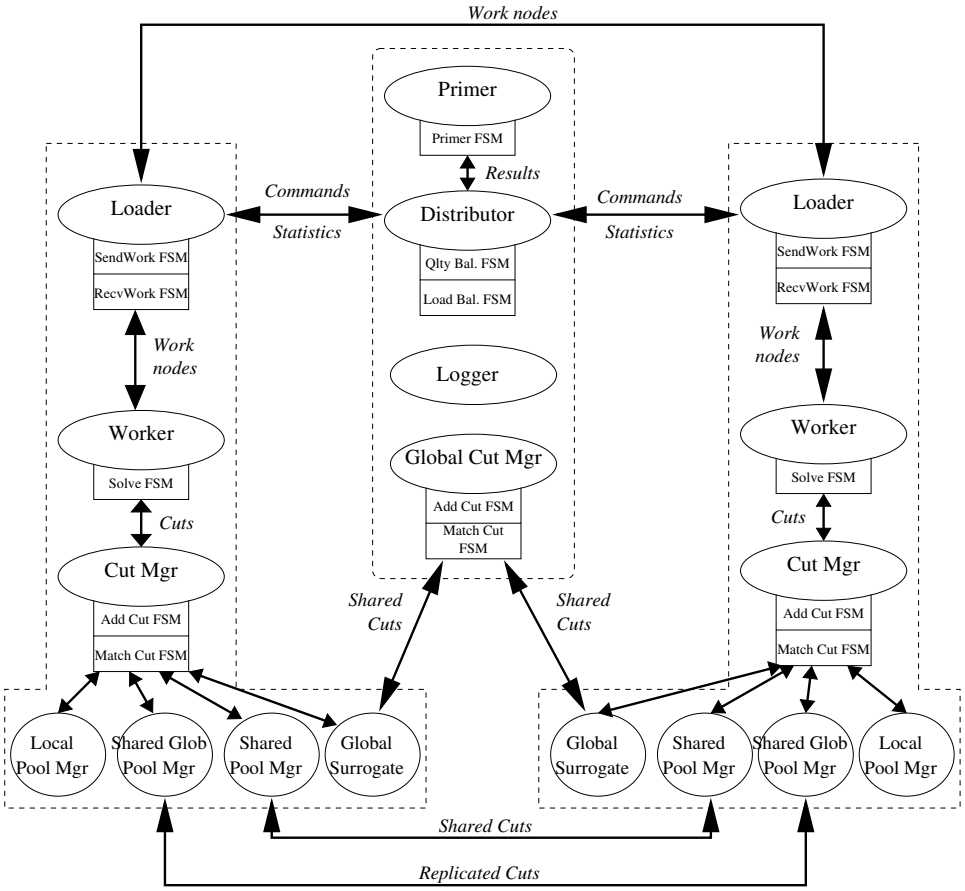

Figure 4: PARINO architecture with the addition of entities for a simple distributed cut management service.

- branch & cut
- branch & price
- Bender's decomposition.

We plan to test the performance of PARINO on the problems in MIPLIB 3.0.

References

[1] R.E. Bixby, W. Cook, A. Cox, and E. Lee, "Parallel Mixed Integer Programming," Technical Report CRPC-TR95554, Center for Parallel Computation, Rice University, 1995.

[2] R.L. Boehning, R.M. Butler, B.E. Gillett, "A Parallel Integer Linear Programming Algorithm," *European J. Operations Research 34*, 393-398, 1988.

[3] T.L. Cannon and K.L. Hoffman, "Large-scale 0-1 Programming on Distributed Workstations," *Annals of Operations Research 22*, 181-217, 1990.

[4] J. Eckstein, "Parallel Branch-and-bound Algorithms for General Mixed Integer Programming on the CM-5," *SIAM J. on Optimization 4*, 794-814, 1994.

[5] J. Eckstein, "Control strategies for Parallel Mixed Integer Branch-and-bound," *Proc. of Supercomputing 94*, IEEE Computer Society Press, Washington, DC, 1996.

[6] J. Eckstein, "Distributed versus Centralized Storage and Control for Parallel Branch-and-bound: Mixed Integer Programming on the CM-5," to appear in *Computational Optimization and Applications.*

[7] B. Gendron, T. Crainic, "Parallel Branch-and-bound Algorithms: Survey and Synthesis," *Operations Research 42*, 1042-1066, 1994.

[8] G. Nemhauser, and L. Wosley, *Integer and Combinatorial Optimization*, New York:Wiley, 1988.

[9] K. Perumalla, K. Schwan, "CoGEnt: A Distributed Active Object Framework," Internal Report, College of Computing, Georgia Institute of Technology, 1997.