2LP Meets DP:
Distributed Integer Goal Programming

David Arnow[1], Ken McAloon[2] and Carol Tretkoff[2]
Dept. of Computer and Information Science
Brooklyn College
2900 Bedford Avenue
Brooklyn, NY 11210
arnow@sci.brooklyn.cuny.edu
mcaloon@sci.brooklyn.cuny.edu
tretkoff@sci.brooklyn.cuny.edu

ABSTRACT**:** We present a vertical integration of linear programming, logic programming and distributed programming which in this paper is applied to a family of administrative problems that occur in many businesses and other organizations. These problems, which require integer goal programming, are both difficult to model and computationally demanding. 2LP is both a modeling language and a language that provides control for solving disjunctive programming problems. In 2LP integer goal programs can be modeled in an elegant manner. Most important to this work, 2LP supports a mechanism for bringing parallel computing to bear on these problems. Using the distributed programming library DP as a substratum, we have developed a scheme for parallelizing integer goal programs in an efficient way. The result is twofold: a family of applications of broad interest and a systematic way of parallelizing them on a distributed programming system.

1. Introduction

A large set of very common situations that arise in decision making can be formulated as integer goal programs. For example, in scheduling personnel many criteria must be addressed and the results must prescribe integer - as opposed to fractional - numbers of workers to fill various slots. In this situation linear programming and round-off techniques fall far short because the scale of operations is seldom large enough to absorb the effects of the round-off gracefully. Another example is portfolio optimization where there are logical conditions since securities must be purchased in sizeable lots and there are several criteria to

---

1. Research partially supported by PSC-CUNY Grant number 6-63278.

2. Research partially supported by PSC-CUNY Grant number 664184, NSF grant number IRI-9115603 and ONR grant number N00014-94-1-0035.

consider for optimization such as dollar duration and convexity. This is a family of applications which come not from the scientific world but from a more workaday world.

In previous work, we have fruitfully combined 2LP, a constraint-based programming language, with DP, a flexible distributed programming environment [McAloon,Tretkoff94; Arnow95; Arnow,McAloon,Tretkoff94] to implement distributed disjunctive programming. Here, we have extended that work and constructed a library for supporting the parallelization of integer goal programs. Our approach has several advantages, among them efficiency, automatic load-balancing and the fact that minimal modification of the sequential integer goal program is required.

We begin this paper with a brief introduction to 2LP. The reader can find more information on 2LP in [McAloon,Tretkoff95]. Linear and integer goal programming are then discussed in connection with a 24-hour scheduling application. In section 4, we describe how one may parallelize an integer goal program using the parallel integer goal programming support library. The implementation of this library is discussed in section 5. Finally, in section 6, we discuss results and future directions.

2. 2LP

The 2LP language combines a logic programming control structure together with a new built-in data type `continuous` in order to capture the modeling methods of linear and mixed integer programming (MIP). Constraints on these variables define a polyhedral set. The representation of this convex body includes a distinguished vertex, called the *witness point*. Other structural information and communication routines are also connected with the polyhedral set. This collection forms an “object” to use object-oriented programming terminology. 2LP, which stands for “linear programming and logic programming,” is a small language with C-like syntax that supports these constructs and has its own interpreter.

2LP is different from algebraic modeling languages such as GAMS [Brooke, Kendrick, Meeraus92] and AMPL [Fourer, Gay, Kernighan93] in several ways. First, it is a logical control language as well as a modeling language and so supports AI methods not available to the MIP modeler. Second, it can directly code disjunctive programs [Balas85] and avoid recoding as a MIP model. Third, 2LP supports parameter passing, procedure based modularity and other structured programming constructs while the GAMS and AMPL models basically consist of global variables and loops to generate constraints. Fourth, the data types `int` and `double` in 2LP are the same as in C/C++ and so the 2LP model can be passed pointers to data directly from C/C++

avoiding a detour through a data file. Last but not least, integer goal programs can not be formulated in GAMS and AMPL; in fact, to our knowledge, 2LP is the only system in which one can formulate a integer goal program as a single model.

In the context of parallel programming, 2LP has two unique functionalities: on the one hand, the logic programming control structure gives the programmer the ability to subdivide the application into parts of varying granularity; on the other, the fact that functions from a distributed programming library can be called and data shared from a running 2LP model means that a 2LP program can be parallelized without any change to the 2LP system itself.

3. Integer Goal Programming

Linear programming is oriented towards optimizing an objective function subject to a set of linear constraints. In practice, however, the decision maker often has several objectives to work on. The standard way of handling this is to use so called deviation variables to “measure” the gap between a feasible point and a goal and then to add these variables with appropriately weighted coefficients to the objective function. The weights on the deviation variables must correspond to the priorities given the various goals. The measure used and the assignment of weights to the deviation variables leave much to be desired both from the point of view of modeling and of results obtained. Things can be improved if goals are prioritized and an incremental approach employed. Using 2LP this can be modeled in a straightforward way. With this approach goals can be formulated and a program written to consider them in the order of their importance. The result is a sequence of successively refined linear programs. In other words, the constraints reached in satisfying one goal stay in place and their solution provides a “hot start” for working on the next goal.

When some or all of the continuous variables in an application are required to have an integer value at solution, the application becomes a mixed integer program. From the point of view of computational complexity, adding integrality requirements has a dramatic impact and can make the models extremely difficult to solve. What is needed is a branch-and-bound search done as cleverly as possible. When there are several goals, several difficult problems are created. This situation is called *integer goal programming*. Unlike the linear case where the successive goals lead to successive refinements of a sequence of constraint sets, there is a discontinuity in going from one goal to the next in the MIP situation and the previous solution can not be used as a starting point for treating the next goal. Instead, the previous solution must typically be jettisoned and the constraints brought back to their original linear state; then the result obtained for the previous goal can be added as a set of additional linear constraints and the resulting new MIP problem solved with the objective that corresponds to the new goal. Often the new goal will require additional constraints to be formulated and these constraints must also be included in order to work on this goal. In a more positive vein, when a solution is found that optimizes the current goal, that solution will provide a bound for the next goal: this information can be used to force the next round to look for strictly better solutions to its goal.

To fix ideas, let's describe an example of this type of application. Consider the problem of staffing a bank of toll booths, a 24 hour supermarket, or other round the clock operation. Let us say that these workers each work 5 consecutive days, that they can come on at any hour, work 4 hours, have a 1 hour lunch break and then work 4 more hours. For every hour of the week, there is a certain number of workers required for that time slot.

To model this situation, one can formulate linear constraints to express the fact that staffing requirements must be met. One can introduce global continuous variables **`S[i]`** to represent the number of workers that come on at each hour and global continuous variables **`X[i]`** to represent the number working at a given hour. The one can store the number of workers needed at hour **`i`** stored in **`wneeded[i]`** and define symbolic constants as follows:

```
#define DAYHOURS         24
#define FULLWEEKDAYS     7
#define WEEKHOURS        (DAYHOURS*FULLWEEKDAYS)
#define WORKWEEKDAYS     5
#define SHIFTHOURS       9
#define BT               4
```

Now we can express the personnel requirements by a call to a routine such as

```
setup()
{
   and (int i=0; i<WEEKHOURS; i++) {
      X[i] == sigma(int k=0; k<WORKWEEKDAYS; k++) (  // number working
         sigma(int j=k*DAYHOURS; j<k*DAYHOURS+SHIFTHOURS; j++)
               S[  (i+WEEKHOURS-j)%WEEKHOURS ]
                  -
                S[  (i+WEEKHOURS-(k*DAYHOURS+BT))%WEEKHOURS ] // lunch
       );
      X[i] >= wneeded[i];  // number of workers needed at hour i
   }
}
```

However, this is not a linear programming problem but an integer programming problem because fractional workers can not be hired!

Since the workers needed will vary quite a bit from hour to hour in the day, with an integer solution we will almost certainly find that the number of workers available at most time slots exceeds the number actually required.

Let's say that management's goals are to:

0. hire as few workers as possible;
1. have as few weekend workers as possible;
2. have supplementary staff available in as many hourly periods as possible;
3. spread the supplementary workers evenly among the periods that are to have supplementary workers.

Then goals 0 and 1 can be formulated as simple objective functions. For example the objective for goal 0 is

```
Z0 == sigma(int i=0; i<WEEKHOURS; i++) X[i] ;
                    // objective function for minimizing number of workers
```

Goal 2 will require additional variables and constraints to capture the slack between the number of workers present at a given hour and the number required; goal 3 requires an additional variable to serve as a uniform bound on the number of supplementary workers and constraints to express this.

We now have four integer programming problems to solve incrementally. Let us remark that for the first goal a quick and dirty initial solution can be found by rounding off the linear solution; unfortunately, this usually provides very unsatisfactory bounds. A better technique is to compute a solution by progressive round-off by which we mean re-optimizing the objective before freezing each continuous variable at the ceiling of its incumbent value. However, the progressive round-off changes the feasible region and the original region must be restored before the search for the optimal integer solution can begin. All of this can be done in 2LP by wrapping the progressive round-off in a double negation-as-failure and caching the results; double negation-as-failure tests the consistency of a statement by running it and then restoring the constraints to their state when the call was made. In this way the branch-and-bound search required to solve the problem to optimality can be given an initial bluff.

With 2LP we are able to formulate one program for the integer goal programming situation. The coding method we employ to achieve the kind of discontinuous progress required by integer goal programming is to use the **if/not** combination which is similar to double negation but which also allows for alternative action if the problem is not feasible. In the following fragment, **Z0** is the objective function for the zeroth goal, **Z1** for the first goal, etc. The variable **Z0** is to be maximized, **Z1** to be minimized, **Z2** to maximized etc.

```
2lp_main()
{
   ... // linear problem set up
   zeroth_goal();
   first_goal();
   second_goal();
   third_goal();
}
```

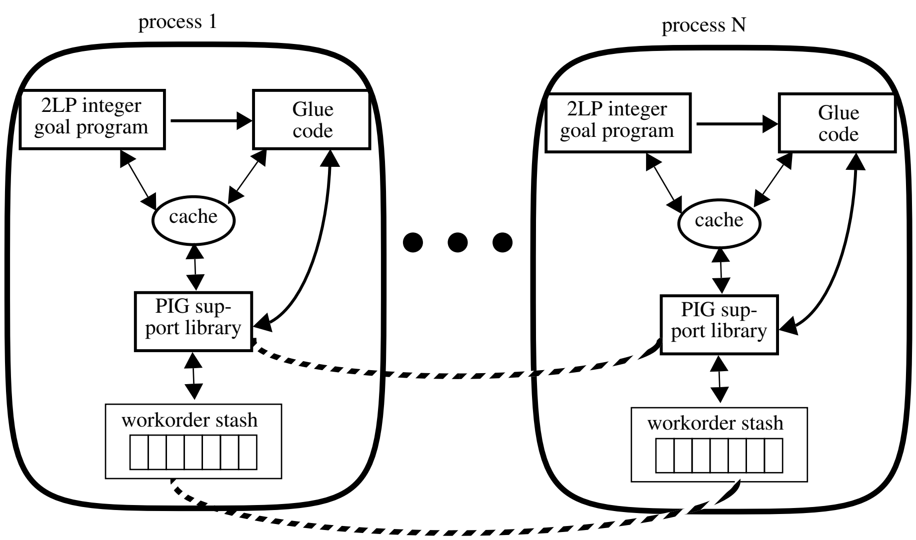
where, the goal solving routines are something like

```
zeroth_goal()
{
   if not {
      solve_goal_0();
      cache0 = wp(Z0);
      cache1 = wp(Z1);
   }
   then printf("No solution found for goal 0\n");
}



first_goal()
{
   Z0 == cache0;
   Z1 <= cache1;

   if not {
      solve_goal_1();
      cache1 = wp(Z1);
      cache2 = ...
   }
   then printf("No solution found for goal 1\n");
}

...
```

With the parallel integer goal programming support library, we can wrap the appropriate parts of the model with calls to the support library to run the program in any number of distributed environments. Thus the resulting model is independent of the number or type of processors used. Moreover, the interface machinery has been designed to apply as systematically as possible to all integer goal programming models. This support library is described in the next section.

4. Parallel Integer Goal Programming

In this section, we describe how an integer goal program, written in 2LP is parallelized in our system. The basic idea, of course, is to have N processes searching different parts of the decision tree which is created by the search for solutions; collectively, they seek to maximize or minimize an objective function. At this point in our research, for simplicity, the goals are serialized, with implicit barriers guaranteeing that all active processes are seeking the extremum of the same objective function. From the programmer's point of view, the processes are symmetric. Collectively, they maintain a *stash* of *workorders*, each of which is a complete description of a node in the current search tree. A parallel integer goal (PIG) computation has the structure shown below..



A globally distributed cache holding the provisional extrema of the objective functions associated with the different goals is accessible to all processes. The processes themselves consist of three components: a 2LP integer goal program; a library of support routines for parallelized integer goal programming; and the application-specific code that serves to provide the "glue" between them.

4.1 The PIG support library.

The parallelization of the integer goal program is facilitated by the PIG support library. This library handles process creation and synchronization, interprocess communication, and maintenance of the cache. It also manages the workorder stash, in order to carry out task decomposition and task assignment. All of these are transparent to the programmer. (In fact, the programmer barely needs to be aware of the distributed nature of the computation!) The programmer need only make a few modifications to the 2LP integer goal program and write the application-specific “glue” code, described in subsection 4.2 below.

The library provides five routines for external invocation.

- **`pig_appstart()`** is called at the outset by each process. It manages process creation and initialization.
- **`pig_stashset()`** is invoked at the outset of each goal computation to initialize the global stash of workorders.
- **`pig_getwork()`** is called at the outset of each goal computation and with the completion of each search of a sub-tree. It provides a workorder for the calling process.
- **`pig_split()`** is called when it is possible to split off a sub-tree from the calling process’s current search tree. The routine will only do so if the workorder stash has fallen below a low-water mark.
- **`pig_brag()`** is called to update the global cache when the calling process has found a new extremum.

4.2 Parallelizing the sequential code.

Although the PIG library handles all the parallelism, it requires some guidance from the integer goal programming code. Once launched the 2LP model can invoke but not be invoked by external C routines. Hence it is necessary to provide an additional C module, the glue, the purpose of which is to provide guidance to the PIG library and to provide information gleaned from that library to the 2LP model. The glue module is thus the part of the C code that is inherently application-specific (otherwise it could have been incorporated into the library).

Modifying the 2LP integer goal programming code. There are only four modifications that must be made in the sequential 2LP code— all of these involve invoking a routine in the glue code, which in turn makes a call to the PIG support library. Two of these modifications are trivial: at the outset, an initialization routine must be called. Second, the 2LP program is embedded in a loop that repeatedly calls a routine to get a workorder. The code below shows the modifications needed for the example presented in the previous section.

```
2lp_main()
{
   startup();// call glue initialization routine
   ... // linear set up
   and(;;) {
      getwork();// call glue to obtain workorder or terminate
      if goal == 0;
         then zeroth_goal();
      else if goal == 1;
         then first_goal();
      else if goal == 2;
         then second_goal();
      else if goal == 3;
         then third_goal();
   }
}
```

In addition, the programmer must identify those places in the 2LP program where a new extremum has been computed and call a routine from the glue code to update the distributed cache of provisional extrema. Second, whenever the 2LP program reaches a point in the search tree where a branching decision must be made (called a *choice point*), it must call a routine so that the alternatives *not* taken at that point are made available to the global workorder stash as a possible new task.

Writing the glue code. Thus, the lion's share of the work is writing the glue code in C. Essentially, this module serves as an intelligent interlocutor between the 2LP code and the PIG support library. It presents data made available by the support library in a form that is accessible to the 2LP code and guides the actions of the support library using information provided by the 2LP code.

There are two objects around which the code revolves: the *cache* and the *workorder*. The latter is a structure that completely defines a partially completed task. Such a structure typically includes the following information:

- the current goal
- the branching decisions made so far towards achieving the goal
- extrema of the objective functions of previously realized goals

Other information can be included as necessary. The programmer must also write any needed 2LP-callable C routines that encode and decode decision information using this structure.

Required Glue Routines

Called by Support Library	Called by 2LP Code
copy_workorder()	brag()
null_workorder()	choice()
give_workorder()	startup()
new_goal()	getwork()
get_stashsize()	
init_stash()	

The functions that manipulate these objects can be divided into those functions which are invoked by the PIG support library and those functions which are invoked by the 2LP code. Below, however, they are grouped according to purpose.

Workorder operations. The nature of the workorder is application specific, yet the PIG support library uses the objects in messages sent to distribute the overall task. The glue code must therefore provide some minimal primitives for the support library: operations to copy a workorder and to initialize it to NULL.

Cache updating. When a potentially better extremum is found in the 2LP code, it invokes a glue routine, which might be called **`brag()`**. This routine will typically invoke the **`pig_brag()`** routine of the support library.

Load balancing. There must be also be a routine to be invoked by the 2LP code when a choice point, as described above, is reached. This routine which might be called **`choice()`** is invoked by the 2LP code and will typically invoke the **`pig_split()`** routine of the support library. At the same time, there must be a routine that extracts part of the current search tree and packages it in the form of a workorder. This routine which might be called **`give_workorder()`** is invoked by the support library when the global distributed stash of workorders falls below a low-water mark.

On the other hand, when the 2LP code completes its search of its current subtree, it returns to the root of the tree, using the backtracking capability provided by the support for logic programming. Let us mention that it is basically double negation-as-failure that is used here. At the root, it will invoke a routine, which might be called **`getwork()`**, to obtain another subtree to search. At this point, the new constraints defining this subtree must be enforced. This simple recomputation accomplishes the required "context-switch." To reduce this recomputation somewhat, further optimization might be achieved using the methods in [Atay,McAloon,Tretkoff93].

Goal establishment. Because each goal involves different notions of what constitutes a workorder, a routine, which might be called **`new_goal()`**, must be provided for the support library to invoke with each new goal. This routine updates variables that define the current goal and invokes the **`pig_setstash()`** routine which creates an initial workorder stash for the new goal.

Stash setup. The size of the workorder stash is extremely important for the efficiency of the computation. The size should be large enough so that no process that has finished with a subtree remains idle for long but not so large that tree splitting and its associated message overhead is too frequent. The precise trade-off is application-dependent, and so the glue must include a function which might be called **`get_stashsize()`**, to return an appropriate size to the support library. In addition, an **`init_stash()`** routine must be provided for the support library to invoke when it needs to initialize a particular workorder in the stash at the outset of the computation of a new goal.

Startup. An initialization routine, called by the 2LP code at the outset, handles any necessary initial actions and invokes the **`pig_start()`** routine which handles process creation and initialization.

5. Implementing the PIG Support Library

5.1 Parallel strategies

The most important optimizations are algorithmic ones, which in this family of problems means tree pruning. The integer goal code can bound the extremum of a subtree that is being searched, and discard the subtree if it is known that another subtree with a better extremum exists. Therefore, the PIG support library seeks to distribute this information as rapidly as possible. It is the cache mechanism that does this. When any process discovers a candidate for the best extremum, the **`pig_cache()`** routine communicates this to one process that is singled out for cache-maintenance. If this value is, in fact, a new global extremum, this process will multicast it to all processes, increasing the possibility of pruning.

The other main strategy of the library is load-balancing. This is achieved by maintaining the stash of workorders, each a description of a search sub-tree. High-water and low-water marks are associated with this stash. One process is responsible for overseeing the stash. As processes become idle, this process provides them with workorders. When the stash falls below the low-water mark, it multicasts a request for workorders to all non-idle processes. When the stash reaches the high-water mark, it multicasts a request to cease sending workorders.

When an active process receives a request for workorders, it is generally not in a position in its computation to respond. However, at the next choice point that it reaches in the tree, it will split a sub-tree off, encode it, and ship it to the maintainer of the stash. Granularity is parameterized and under the control of the 2LP programmer, since environments will vary in communication overhead. This guarantees that even when there are great performance differences among the actual machines in the system, the load will be well-balanced.

In the current implementation, a master process maintains both cache and workorder stash. This asymmetry is invisible to the programmer and is not even essential. In the current implementation it is just a matter of convenience. In our environment (up to around 40 processes) there is no loss of efficiency. In a massively parallel environment the work of the master process would have to be distributed itself.

How effective are these strategies? In the computation associated with each goal, there are logically two phases: searching for the extremum and then verifying that it is truly the extremum. The search phase algorithmically benefits from breadth-first search. Rapid distribution of the cache is important in this phase and only in this phase. In the verification phase, a more trivial kind of parallelism is involved, because once the true extremum is known, there is a fixed number of nodes in the search tree that must be examined. Here the load balancing strategy is all-important.

5.2 Using DP

Implementing the PIG library requires the use of a distributed programming platform that efficiently supports PIG's communication requirements. Both strategies described above, i.e., obtaining and distributing the cache and the workorders have a common aspect: they are data-driven rather than need-driven. For example, it is a process's discovery of a new extremum that prompts the distribution of that information to the other processes. That information will be useful to the receiving processes *whenever* it arrives. Arrival however is completely unpredictable. To avoid the inefficiency of polling, the PIG library needs to communicate using messages that interrupt the receiver. Because of this requirement, the DP library is used. Unlike many other distributed programming support systems (for example PVM [Sunderam90]), DP provides interrupting messages.

In addition, DP allows the implementor to send messages cheaply and unreliably. This is useful when, as in the case of cache distribution the overhead incurred by guaranteeing reliable message delivery is less desirable than the occasional loss of a message. (Workorder loss on the other hand is never acceptable and it requires reliable delivery.)

Finally, DP's message-sends are asynchronous, in the sense that return after the call to send is not delayed by the system at all. This applies to both reliable and unreliable message modes. This property is important in reducing computation delays that might otherwise result from sharing information.

6. Results

Tables 1 and 2 display running times for a sample application on Sun IPCs and Sparc10s. It is interesting to note the irregularities in speedup. These are not artifacts of particular experiments but rather result from the breadth-first character of a parallelized depth-first tree search. For example, anomalies, such as superlinear speedup, can occur when, as a result of the vagaries of task distribution, a process discovers a particularly good solution early on. Such a discovery may serve as a cut-off to speed the search of the other processes. On the other hand, in the end-game, a process may not be able to readily share work because it is not generating a sufficient number of branching pointsd-dressing this problem is on our agenda for future work.

TABLE 1. SPARC 10 Running Times

processors	1	2	3	4	5
seconds	581	200	190	149	145

TABLE 2. IPC Running Times

processors	1	2	4	8	10
seconds	1380	1066	740	399	265

We also plan to explore parallelizing the pursuit of the goals themselves. That is, rather than run goal n completely prior to running goal n+1, processes would be assigned to explore later goals in order to converge more quickly in a breadth-first manner to a globally satisfactory solution taking all goals into account. This raises interesting algorithmic and communication questions.

7. References

[Arnow95] D. Arnow, DP — a library for building reliable, portable distributed programming systems. *To appear in the Winter USENIX Conference.* New Orleans, Jan. 1995.

[Arnow,McAloon,Tretkoff94] D. Arnow, K. McAloon and C. Tretkoff, Disjunctive programming and distributed programming, Sixth IASTED-ISMM Conference on Parallel and Distributed Computing and Systems, Washington DC, Oct. 94.

[Atay,McAloon,Tretkoff93] C. Atay, K. McAloon and C. Tretkoff, 2LP: a highly parallel constraint logic programming language, *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, March 1993.

[Balas85] E. Balas, Disjunctive programming and a hierarchy of relaxations for discrete optimization problems, SIAM J. Alg. Disc. Meth., **6** (1985) 466-486.

[Brooke,Kendrick,Meeraus92] A. Brooke, D. Hendrick and A. Meeraus, GAMS: A User's Guide, The Scientific Press, 1992.

[Fourer,Gay,Kernighan93] R. Fourer, D. Gay and B. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, The Scientific Press, 1993.

[McAloon,Tretkoff95] K. McAloon and C. Tretkoff, *Optimization and Computational Logic*, to appear, J. Wiley and Sons.

[McAloon,Tretkoff94] K. McAloon and C. Tretkoff, 2LP: Linear programming and logic programming, to appear in the *Proceedings of Principles and Practice of Constraint Programming '93*, MIT Press.

[Sunderam90] Sunderam, V.S.: PVM— A framework for parallel distributed computing. *Concurrency: Practice and Experience 2* (1990).