# Distributed Programming and Disjunctive 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:** A merger of two programming tools, one a library for writing portable distributed programming systems, the other a language that combines AI and OR methods for decision support software systems, has resulted in an programming environment for maximally exploiting message-passing parallel systems for solving difficult problems involving logic and optimization.

**Key terms**: *Parallel computing, optimization, languages, integer programming*

## 1. Introduction

Significant progress has been made by the AI, OR and parallel processing communities in tackling difficult combinatorial optimization problems. In addition there has been a confluence of AI and OR methods, e.g., [Balas],[Jeroslow],[Hooker *et al.*]. This has led to the extension of IP and MIP programming known as disjunctive programming. The 2LP programming language [McAloon,Tretkoff 3] is designed to make the merger of AI and OR methods available in a programming toolbox. In this paper we present the addition of the powerful technique of distributed processing to this toolbox. This is carried out by means of the DP library for message-passing based distributed processing [Arnow].

Many parallel computer systems exist, but the one that is ubiquitous is the network of workstations. In order to parallelize 2LP and these new methods in an optimal way for the network environment, certain message-passing services, which are not always available in standard systems, are desirable such as unreliable message passing and interrupts. Although there are several parallel programming environments available for distributed processing, it is the DP library [Arnow,] that provides the full set of features for this parallelization effort. This makes for a marked strengthening of the approach reported in [Atay,McAloon,Tretkoff].

In the next two sections we give a brief overview of the DP library for distributed programming and the 2LP language. We then discuss the merge of these systems, and two sample applications: a stochastic disjunctive program and the classic capacitated warehouse location problem.

---

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

## 2. DP

DP is a library of process management and communication tools for facilitating writing portable distributed programming systems on networks, internetworks, message-passing multicomputers or, in fact, on any MIMD system. Its design reflects these goals:

(1) *Flexibility and power*: The primitives should provide the power to perform most distributed programming functions. These functions involve process creation and message passing. The application programmer should not in general lose any functionality or efficiency by using DP instead of the native system primitives.

(2) *Portability*: The primitives should be implementable on most if not all MIMD parallel computing platforms. DP has run with C and often Fortran-77 interfaces on the following platforms: Sun sparcstations, IBM RS/6000, DEC riscstations, Alliant, and the KSR-1. An early version was implemented on DesqView and there are plans to implement it on Windows-NT.

(3) *Simplicity*: The DP interface is less than two pages of code and the most commonly used part is under a page. No intervention by a system administrator is needed for installation, nor does DP require any thing from its environment except for access to TCP/IP and a remote process creation service such as `rexec`.

DP supports dynamic process creation and message passing with a variety of semantics. Systems using DP can dynamically spawn processes on a given set of host machines, dynamically add or delete hosts machines from that set, invite other, ancestrally-unrelated processes to join the application. The processes in a DP-based application communicate by sending messages which, under program control, may or may not be transmitted reliably and may or may not generate a software interrupt to the recipient. DP provides the necessary primitives necessary to treat such software interrupts with integrity. It also provides primitives to facilitate data conversions across diverse architectures.

In keeping with its goal of simplicity, in comparison to other distributed programming environments and languages, DP provides a very low-level application interface. There is no typing of messages, minimal support of data format conversions, no queuing of synchronous messages, and no concept of conditional receives. That is because simplicity facilitates portability and because DP is intended to be used as a platform for implementing higher-level distributed programming support systems such as the merger with 2LP described in this paper.

Although of the same genus as environments such as PVM [Sunderam, 1990] and P4 [Butler and Lusk, 1992], DP differs from them in a number of important ways, primarily because of its intent to be a base for developing portable distributed programming environments as distinct from portable distributed applications. Functionally, the most important difference is its provision for unsolicited messages whose arrival generate a software interrupt. This provides a flexible method of sending large quantities of urgent information that cannot be easily accomplished with, say, the unix-signal transmission of PVM. It also allows a programming environment to provide a shared-memory like capability. Furthermore, DP gives the programmer the freedom to use the cheapest (and possibly unreliable) message-transmission mechanism. Other differences include DP's avoidance of Unix-specific characteristics and its provision for both non-blocking and blocking receipt of messages. DP's implementation approach also differs from the above. Except for a brief use of TCP during process creation, it makes exclusive use of UDP. This avoids the possibility of exhaustion of file descriptors, the need to route messages among processes and the cost of connection establishment and, in general, more naturally fits the message-oriented DP model.

**Process management.** Each DP process is identified by a value of type `DPID`, guaranteed to be unique among *all* possible DP processes. The function `dpgetpid()` returns the current process's id. Execution of a DP application starts by invoking a suitably compiled and linked DP executable program. This first process is termed "the primary". Process are spawned by calling `dpspawn()`, passing

- the name of the program to be executed,
- a symbolic id of the machine on which to run the new process,
- a "semantic packet" to be sent to the new process; this packet is a program-determined collection of bytes that can be used as an initial parent-to-child communication— typically it contains the DPID of the parent, along with possibly other application parameters.

The call to `dpspawn()` returns the DPID of the new process. The entry point for the newly spawned *secondary* processes is `main()`, not the instruction after the call to `dpspawn()`. These secondary processes do not have access to the original command-line arguments nor do they inherit a copy of the creator's address space— their data must come from the semantic packed they are provided or from subsequent received messages.

The first DP call any DP program makes must be to `dpinit()`, which sets up the necessary process and communication environment. If the process calling `dpinit()` was created by `dpspawn()` the caller is given access to the *semantic packet*. The contents of this packet is application defined and determined by the `dpspawn()` call that created the process.

**Communication**. DP processes communicate by sending and receiving messages. For sending messages, the `dpwrite()` routine requires the DPID of the recipient and a pointer to the start of the message body along with the message body size. Messages can be reliable or non-reliable and interrupting or non-interrupting. Reliability here means that DP will carry out an ack/timeout/retransmit protocol that will guarantee the eventual availability of the message to the target provided that the underlying network and relevant host machines do not fail and that the messages will be received by the application code in the order in which they were sent. Sending the message unreliably means that DP will send the message to the target only once and assume no further responsibility— a much cheaper method of message transmission.

Regardless of whether the message is sent reliably, return to the sender is immediate; the sending process will NOT be blocked during this time. This means that upon return from `dpwrite()`, one thing *is* certain: the target has not yet received the message!

Logically, each DP process has two receiving ports: one for receiving interrupting messages and another for receiving non-interrupting messages. Non-interrupting messages are queued upon arrival and do not affect the receiving process until it explicitly reads the message with the `dprecv()` call. In the case of the interrupting message, the message's arrival may force the invocation of a special message-catching routine if such a routine has been designated by the receiving process. Whether or not such a routine has been designated, the interrupting message must be read explicitly with the `dpgetmsg()` call, not the `dprecv()` call. Both routines return the DPID of the sender as well as the message itself.

**Critical sections.** The message handler specified in a call to `dpcatchmsg()` may be invoked at any time and may reference global objects. Thus, any other code that accesses these global objects is a one-way critical section. DP provides the necessary support routines to guarantee mutual exclusion. In addition, time-out, pause and alarm clock management routines are provided.

# 3. 2LP

The 2LP system provides an elegant and powerful programming language environment for combining AI and OR methods for decision support software systems. In the dialogue between AI and OR, there are two basic themes: (1) declarative programming and the notion of logical consequence and, (2) procedural programming and the search algorithm in its many variations. Integrating AI and OR requires an environment that combines a modeling language with a logic based language. 2LP, which stands for "linear programming and logic programming," treats the mathematical module in an object-oriented way, and has the backtracking mechanism of logic programming built in. 2LP is both an algebraic modeling language and a logical control language. By bringing these techniques together in a language which has standard C style syntax, this technology provides very powerful and usable tools for decision support programming.

2LP introduces a new built-in data type `continuous` in order to capture the modeling methods of linear programming. 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 (OOP) terminology. 2LP is a "little language" that supports these constructs and has its own interpreter. On the other hand, object-oriented programming as in C++ enables one to extend a language with new data types without writing an interpreter for the expanded language. Because of the need for logic based control, backtracking and restoring of state in 2LP, the C++/OOP method can not be used for the purposes for which 2LP is designed; a little language approach proves necessary. For the other approach to linear programming, see [Nielson].

2LP is different from algebraic modeling languages such as GAMS [Brooke, Kendrick, Meeraus] and AMPL [Fourer, Gay, Kernighan] 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 [Balas] 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.

As a programming language, 2LP is small; its ambitions are circumscribed and focused. It is designed to be embedded in the larger programming scheme of things and to be used as a set of library routines. Arrays of `double`s and `int`s and C-functions can be provided by the principal system or by ancillary C-code and addressed as `extern` by the 2LP model. The language is built on an abstract machine model called the S-CLAM which is an analog of the WAM and CLAM in the smaller 2LP context [McAloon,Tretkoff 1]. The 2LP architecture provides for a very modular linkage between the logical control and the mathematical constraint solver. As a result, 2LP supports use of optimization libraries such as Cplex, XA and OSL as well as its own simplex based mathematical code. This means that 2LP can be used for applications that require state-of-the-art optimization software and the system can move with changes in the mathematical programming world. (It is also to be hoped that by interacting with systems like 2LP, optimization software will become more responsive than at present to requirements like incremental loading and deleting of constraints.)

2LP provides language tools for programming the desired semantics for an optimization application. For a classical linear program the semantics of optimization are straightforward. When a richer mix of programming constructs and logic is introduced, there are many possible interpretations to "max: ... subject_to ... ." In particular, the programmer will want to control the state of the geometric object defined by the constraints both during and upon exiting this block of code. For that reason, 2LP supports control mechanisms and some built in optimization constructs for writing code that expresses the semantics intended by the modeler. This is especially helpful when optimization routines occur as subroutines in a larger application.

## 4. The Merge

The goal of combining two systems is to create a whole that is greater than the sum of its parts. However, it is also desirable that the effort required to work with the hybrid system not be greater than the sum of the efforts required to use each part by itself! The DP-2LP merge meets both of these goals. A programmer who understands both DP and 2LP independently can use DP-2LP with minimum effort.

Parallelization of a 2LP application can take place at several levels. At the simplest level, the application will consist of a collection of virtually independent linear, integer or disjunctive programs. For example, in the stochastic model discussed below, one makes independent runs of the same model on different data sets and the results are assembled at the end. Here the parallel control need only launch the separate processes and assemble the results at completion. Another kind of situation arises in applications which are disjunctive programs which split into a tree whose leaf nodes are linear or integer programs; this occurs, for example, in financial models when regulations permit different structurings of a model and the client can determine which is most beneficial to them. At yet another level, in applications where logical or integrality requirements lead to large search spaces, parallelism can be used to divide the search among cooperating processes. In this scenario, the search can at times be pruned dramatically as processes communicate results obtained. The full panoply of strategies for load-balancing and work distribution can be brought into play here.

From the DP point of view, the process of parallelizing an existing 2LP application proceeds in the standard way for message-based MIMD systems. Processes are identified with computational tasks along with the necessary process-to-process information transfers. The mapping of these abstractions into DP services is handled virtually identically to any DP application development effort. The invocation of DP services can be made both within the 2LP code itself as well as in ancillary C code associated with a given 2LP application. These services are transparent to the DP-2LP programmer: many of them are little more than straightforward wrappers of DP primitives; others involve special handling so as to fit seamlessly with the 2LP semantics.

Essential to the ease of parallelizing an existing 2LP application is the minimal need for modifying the application code itself to avail it of DP services. After an initial call to set up DP, all calls on DP take place at points in the 2LP code where the application has information to export explicitly. In the stochastic disjunctive programming example below, this reduces to sending the results of the various runs to a master process at the end of the worker's model solving efforts. In the capacitated warehouse example below, this takes place at three points in the model: (1) when a process has finished its work on one part of the search, additional work is obtained from the global 2LP model, mediated by DP; (2) when a process finds a solution to the logical requirements of the application, it broadcasts relevant information; and (3) when a process creates a branching point, it makes this information available.

There are two other important features of DP and 2LP that lend elegance to the merger. Because of DP's interrupt facility, the 2LP program itself does not need to incorporate any code for receiving the information exported by the other processes. This is handled by the interrupt of DP and a catcher function that is written in ancillary C code. On the other hand, because 2LP has a logic-based control, the messages that need to be passed between processes including those for assigning new tasks are very compact. This is critically important in a message-passing parallel environment.

## 5. Two Applications

**Model A: A Stochastic Multiperiod Planning Problem.** This is a stochastic version of a the multiperiod blending problem which is given in [Williams], where it is labelled as "difficult to solve" from a computational point of view. The task is to plan production of a blended cooking oil for each month over a six month period. In addition to production constraints on the blend of oils and on the amounts of oil that can be stored and held over from month to month, there are also logical requirements on the process. Thus, certain oils can not be used unless others are present, there is a threshold amount for each oil that can be used in the monthly process, there is a limit on the number of oils that can be used in the blend, and so on. These requirements change a linear program into a disjunctive program. The task is to analyze how much profit can be expected to be achieved given various pricing scenarios for the coming months. These pricing scenarios can be obtained from a random number generator. The 2LP model codes the application as a disjunctive program and uses a hill-climb to generate a good first solution, before embarking on a branch-and-bound search to find the optimal solution. The model requires 96 continuous variables and 60 constraints. If coded in classical MIP fashion, the model requires an additional 30 0-1 variables and an additional 78 (multivariable) constraints. This added machinery slows down the speed of solution since it doesn't "tighten" the model. Each such model takes about 92 seconds to solve on a Sun IPC. With the DP-2LP merged system, large numbers of these scenarios can be run with virtually linear speedup. Each worker process generates a sequence of scenarios and solves them, using 2LP's ability to communicate directly with the random number generator which is embedded in the ancillary C-code. With DP, parallelizing the 2LP application in this way is straightforward. In the `2lp_main()` routine below, the parallelization consists of a call to `startProcs()` and a call to `sendResult()`, both found in the ancillary C code, which follows.

```
... definitions ...

extern double profit, price[MONTHS][KINDS];
extern void get_prices(), startProcs(), sendResult();
continuous Revenue, Plan[MONTHS][MODES][KINDS];

2lp_main() {
      double cache;
      set_up(Plan);  // underlying linear model
      start_procs();
      or(;;) {
              sendResult();    // send profit to primary
              get_prices();      // exit if DP requests it
              objective(Revenue,Plan); // set up objective function
              if not hill_climb(Plan,cache);
              then fail;                    // No disjunctive solution possible

             Revenue >= cache;              // cache is lower bound on
                                            // profit determined by hill_climb
              solve_model();                // do branch and bound
              profit = wp(Revenue);          // record result
             fail; // undo solution and go to top of loop
        }
}
```

```
objective(continuous Revenue,Plan[][MODES][KINDS]) {
               Revenue ==                    // objective function
                sigma(int i=0;i<MONTHS;i++)
                        sigma(int j=0;j<KINDS;j++)
                                ( TONPROFT*Plan[i][PROC][j] -
                                        price[i][j]*Plan[i][PURC][j] -
                                        STORCOST*Plan[i][STOR][j]  );
}
```

The support that DP provides is shown below. The `startProcs()` routine, called by all processes, directs the primary process to call `primary()` and, for the worker processes, receives the semantic packet, the contents of which are shown in the typedef for `SEM`. This packet includes the address of the pri mary, the local sample size, and a shift for the guaranteeing independent pseudo-random number generation. The `primary()` routine spawns all the worker processes, initializing and passing the appropriate semantic packet. It then receives messages from the workers as they generate results.

Upon return from `startProcs()`, the worker processes execute the 2LP application code shown above, calling `sendResult()` (shown below) to send results to the primary. These processes get their data from `get_prices()` (shown below) which utilizes information in the semantic packet to generate stochastically independent normal distributions of price data.

```
#include        "2lplink.h"
#include        <dp/dp.h>

typedef struct  semstr {     /* The semantic packet recvd by  worker processes   */
        DPID    s_primaryAddress;/* DPID of the primary                        */
        int     s_jobsRequested;/* the local sample size                       */
        int     s_RandomSeedShift;/* guarantee stochastic independence         */
        } SEM;
static  SEM     *sp, s;
static  int     jobs_done = 0;

double  profit;
double  price[MONTHS][KINDS];    /* perturbed price data */
double  iprice[MONTHS][KINDS] = {... initial price data ... };

void
get_prices() {
        int     i, j;

        if (jobs_done++ == sp->s_jobsRequested)
                dp2lp_exit();

        for (i=0; i<1000*(sp->s_RandomSeedShift); i++)/* distribute prn sequence by  */
                (void) rannyu(0);                   /* discarding different batch  */
                                                    /* for each process            */
        ... loop over price data ...
                /* normal perturbation of initial price data */
                price[i][j] = normal(iprice[i][j]);
}

static void
primary(int nhosts, SEM *sp, int semsize, int myhid) {
        int     i, j=0, nprocs;
        DPID    dummy;
        double  profitavg=0.0;

        dp2lp_getpid(&s.s_primaryAddress);
        for (i=0; i<nprocs; i++) {
                s.s_RandomSeedShift = i;
                dp2lp_spawn(&dummy, (++myhid)%nhosts, (char *) &s, sizeof(s), 1);
        }
        for (i=0; i<nprocs*jobs_requested; i++) {
                dp2lp_recv(&dummy, (char *)&profit, sizeof(profit), DPBLOCK);
                if (profit>0.0) {
                        j++;
                        profitavg += profit;
                }
        }
        profitavg /= (double)j;
        ... open results file and write profitavg ...
        dp2lp_exit("boss done");
}
```

```
void
startProcs() {  /* CALLED by the 2lp application code    */
        int     nhosts, semsize, myhid;

        nhosts = dp2lp_init((char *) &sp, &semsize, &myhid);
        if (semsize == -1)  {
                primary(nhosts,sp,semsize,myhid);
                dp2lp_exit("boss done"); /* NEVER RETURN TO 2lp app CODE */
        } /* secondary just returns */
}

void
sendResult() {
dp2lp_write(&sp->s_primaryAddress,
              (char *) &profit, sizeof(profit), DPREL|DPRECV);
}
```

**Model B: The capacitated warehouse location problem.** Here we consider an application which is very much a classic, the Capacitated Warehouse Location Problem. The goal is to supply the needs of customers from a group of proposed warehouses and to minimize monthly cost. Using a warehouse requires a fixed monthly charge and there is the cost of supplying all or part of a customer's requirements from a given warehouse. The problem is to determine which warehouses to open so as to minimize cost while meeting demand. The solution requires logic and can not be done by a linear program because of the "all-or-nothing" character of the fixed-charge. Branch-and-Cut techniques for this kind of application are easily coded in 2LP and are very effective, but many models still require Branch-and-Bound search to solve to optimality. The merged DP-2LP system is especially suited for parallelizing the branch-and-bound search. In brief, the worker processes are given portions of the search tree to explore in response to their own requests for work. The `get_iter()` routine sends these requests to the primary and then waits until a state variable changes, indicating that work has been received:

```
void
get_iter() {          /* executed by worker when current task completed */
      MSG     m;

      ... fill in MSG text and send to primary ...
      dp2lp_write(&sp->s_primary, (char *) &m, sizeof(MSG), DPREL|DPGETMSG);
      dp2lp_block();
      while (job_status==UNEMPLOYED)        /* wait for state change    */
              dp2lp_pause(0,0);             /*    avoiding race condition */
      dp2lp_unblock();                      /*    with block/unblock      */
      if (job_status == RETIRED)
              dp2lp_exit("florida");

      ... access state information and make available to 2LP code ...
}
```

As a worker process searches its local tree, when a solution node is reached, it communicates the new bound on the objective function to the other processes via an interrupting message. This and other kinds of interrupting messages force the immediate invocation of a worker's message handling routine:

```
static void
wcatcher() {/* the worker message handler*/
      MSG     m;       /* basic application determined message structure */

      while (dp2lp_getmsg(&src, (char *)&m, sizeof(m)) != DPNOMESSAGE) {
              switch (m.msg_type) {
              case M_NEWRESULT:  updateBound(); break;
              case M_NEWJOB:     job_status = WORKING; ... create local state ... break;
              case M_RETIRED:    job_status = RETIRED; break;
              case M_SPLIT:      ... split current search tree and export to another process ...
                                 break;
              }
      }
}
```

The `M_NEWRESULT` message informs a process that a new bound has been found for the optimization problem, furnishing it with a potentially better bound with which to prune its own local search space. The `M_NEWJOB` message brings a new subtree for the process to work on. Because of the logical control in 2LP, the description of the subtree need only consist of an integer and two small integer arrays. This small message size makes 2LP especially amenable to parallelization in a message passing environment.

One further point: when the 2LP code creates a branching point, a call is made to a function `choice()` which records the fact that this branching has occurred which makes the untaken subtree available to be given to another worker in response to an `M_SPLIT` message.

It is interesting to note in these applications, a significant portion of the computation time is spent verifying that the optimal solution has indeed been found. As noted in [Atay, McAloon, Tretkoff] and [Atay], it is on this side of the search that parallelism can be expected to be most effective. In other words, while concurrency can assist in finding the optimal solution faster, still parallelism but not concurrency can serve in verifying optimality. By way of example, on problem cap71 from the Imperial College Library, sequential code on a Sun IPC processor requires 20 minutes to run to completion, while the parallelized version on 16 processors runs in less than 3 minutes.

**TABLE 1. Cap71 Running Times**

| processors | 4 | 8 | 16 |
|---|---|---|---|
| seconds | 452 | 345 | 169 |

## 6. Availability

The DP system runs on UNIX workstations.

2LP can be used as a stand-alone system or as a callable library. In stand-alone mode, a 2LP model is sent directly to the compiler/interpreter. In library mode, 2LP is called from C as a function; the 2LP function is sent the whereabouts of the data and C functions it is to use as well as the 2LP model; the model can be sent either as a file or as a pre-compiled C data structure.

The 2LP system currently runs on UNIX workstations and 386/486 PCs.

2LP, DP, and the DP-2LP merge are available via anonymous ftp to acc5.its.brooklyn.cuny.edu in /pub.

## 7. References

[Applegate93] D. Applegate, R. Bixby, V. Chvatal, and W. Cook, Solving large TSPs in parallel, DIMACS presentation, May 1993.

[Arnow91] Arnow, D.M.: Correlated Random Walks in Distributed Monte Carlo Programs. *ICIAM 91*, Washington D.C. (July 1991).

[Arnow94] Arnow, D.M.: DP: a library for building portable reliable distributed applications. *Submitted for publication.* (July, 1994).

[Atay92] C. Atay, Parallelization of the Constraint Logic Programming Language 2LP, Ph.D. Thesis, City University of New York, June 1992

[Atay93] 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.

[Bal89] Bal, H. E., Steiner, J. G., and Tanenbaum, A.S: Programming languages for distributed computing systems. *Computing Surveys 21*,3 (Sept. 1989).

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

[Butler92] Butler, R., and Lusk, E.: User's guide to the p4 programming system. *Tech. Rep. ANL-92/17*, Argonne Nat. Lab. (1992).

[Brooke92] A. Brooke, D. Hendrick and A. Meeraus, GAMS: A User's Guide, The Scientific Press, 1992.

[Clark88] Clark, K.L.: PARLOG and its applications. *IEEE Transactions on Software Engineering SE-14*, 12 (Dec. 1988).

[Douglas93] Douglas, Craig C., Mattson, Timothy G., and Schultz, Martin H.: Parallel programming systems for workstation clusters. *Yale University Dept of CS Technical Report*, (Aug., 1993).

[Cox92] J. Cox, K. McAloon, C. Tretkoff, Computational complexity and constraint logic programming, *Annals of Mathematics and Artificial Intelligence*, **5**(1992) 163-190.

[Fourer93] R. Fourer, D. Gay and B. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, The Scientific Press, 1993.

[Gelernter85] Gelernter, D.: Generative communication in Linda. *ACM Transactions on Programming Languages and Systems 7*, 1 (Jan. 1985).

[Hooker94] J. Hooker, H. Yan, I. Grossmann, R. Raman. Logic cuts for processing networks with fixed charges. *Computers and Operations Research 21*, No. 3, (1994) 265-279.

[Jeroslow89] R. G. Jeroslow, Logic-Based Decision Support, Mixed Integer Model Formulation, North-Holland, New York, 1989.

[Jeroslow89] R. G. Jeroslow, J. Wang, Dynamic programming, integral polyhedra and Horn clause knowledge bases, *ORSA Journal of Computing* **1**, No. 1(1989) 7-19.

[McAloon90] K. McAloon and C. Tretkoff, Subrecursive constraint logic programming, *Proceedings of the NACLP 1990 Workshop on Logic Programming Architectures and Implementation*, edited by J. Mills.

[McAloon94] K. McAloon and C. Tretkoff, *Logic and Optimization*, in preparation

[McAloon93] 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.

[Nielsen93] S. Nielsen, A C++ class library for mathematical programming. Technical Report, Management Science and Information Systems, University of Texas at Austin, 1993.

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