

Scalability of Integer Programming Computations for Sensor Placement in Water Networks

Jonathan W. Berry* William E. Hart* Cynthia A. Phillips*

Abstract

Integer programming (IP) is a general optimization technology capable of expressing most resource allocation decisions. More specifically, IP is the optimization of a linear objective function subject to linear constraints and additional nonlinear integrality constraints. For sensor placement problems, discrete decision variables usually represent decisions to place or not place a sensor at a particular network location. Discrete and rational derived variables compute the effect of various attacks.

For the past two years, we have investigated a number of IP models for sensor placement with varying public-health-related objectives and varying assumptions about input data, water transport, and response. In those studies, we used IP as a tool to explore tradeoffs between number of sensors and population vulnerability. In this study, we focus on IP technology itself, as applied to sensor placement problems.

In this paper, we empirically investigate the difficulty of solving some sensor-placement IPs on various computing platforms. We consider serial solution using commercial codes and parallel platforms using the PICO (Parallel Integer and Combinatorial Optimizer) parallel IP solver. We discuss algorithmic advances that might push the frontiers of tractability.

1 Introduction

In this paper we explore computational issues for solving integer programs used to compute sensor placements in municipal water networks. Within the limits of modeling assumptions and data uncertainty, these solutions are optimal when we can compute them. Our papers to date [2, 4, 6, 25, 24] focus on the results of the computations and their implications for water network security. Since these results seem to provide useful or at least provocative information, we wish to be able to solve increasingly large and more complex models as we attempt to model increasingly realistic settings.

*Algorithms and Discrete Math Dept, Sandia National Laboratories, Albuquerque, NM; PH (505)284-4021,(505)844-2217,(505)845-7296 {jberry, wehart, caphill, jwatson}@sandia.gov. Sandia is a multipurpose laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

Integer programming (IP) is the optimization of a linear function subject to linear and integrality constraints. A mixed-integer program (MIP) in standard form is:

$$\begin{aligned}
 \text{(MIP)} \quad & \text{minimize} \quad c^T x \\
 & \text{where} \quad \begin{cases} Ax = b \\ \ell \leq x \leq u \\ x_j \in \mathcal{Z} \quad \forall j \in D \subseteq \{1, \dots, n\} \end{cases}
 \end{aligned}$$

where x and c are n -vectors, A is an $m \times n$ matrix, b is an m -vector, and \mathcal{Z} is the set of integers. We can convert an inequality constraint to an equality by adding a variable to represent slack between the value of ax and its bound b . For example, a constraint of the form $ax \leq b$ becomes $ax + s = b$, $s \geq 0$. If all variables are integer variables, this is a (pure) integer program (IP). If none are integer, this is a linear program (LP). Otherwise it is a mixed-integer program (MIP). The objective can be a maximization if that is the more natural optimization direction.

The only nonlinearity in MIP is the integrality constraints. These give MIP its enormous practical modeling power, but they are also responsible for MIP's theoretical difficulty and its practical difficulty, especially for large problems. In sensor placement models, integer variables represent decisions. In particular, binary variables (those for which 0 and 1 are the only possible values) represent yes/no decisions for placing a sensor at a particular location.

Integer programs are usually solved to (near) optimality by intelligent search. Theoretically, in the worst case, IPs can require runtime that is exponential in the input size. However, in practice, even million-variable IPs can be solved efficiently. Predicting the runtime of an IP computation is still one of the biggest open questions in operations research. Practical efficiency seems to depend most critically upon problem structure and how well one can exploit that structure. Within similarly-structured problems, size becomes a limiting factor.

As IP instances increase in size, they eventually overwhelm serial solvers by requiring too much time or memory. The classic branch-and-bound search algorithm for MIPs is well suited for parallel platforms. Large-scale parallel MIP solvers are research codes which will likely never match the performance of commercial solvers for small numbers of processors. However, with a sufficient number of processors, these solvers can potentially solve instances serial solvers cannot.

Scalability is a measure of parallel performance. Let T_s be the running time of the best serial code for a problem and let T_p be the running time of a parallel solution using p processors. Then the *speed up* of the parallel computation is $s_p = T_s/T_p$. If speed up s_p increases roughly proportionally to number of processors p , the parallel program has (near) *linear speed up*. This is normally the best possible performance if the total work is stable and parallel overhead is small. Sometimes codes can achieve superlinear speed up due to cache effects or other "lucky" events that reduce the total work of the computation. Sometimes in the literature parallel performance is measured via *efficiency*, defined as $E_p = s_p/p$. A scalable code (one with near-linear speed up) has efficiency close to 1. For any

given instance, there is a limit to the amount of exploitable parallelism, but a good parallel solver scales well up to that limit.

In this paper we show that parallel MIP calculations for sensor placement problems can scale well at least with small numbers of processors and we show that one can speed the computation by exploiting problem-specific structure. The remainder of the paper is organized as follows. Section 2 discusses MIP models for sensor placement in water networks. In Section 3, we describe the basic search algorithm for solving MIP instances. This provides background for understanding our discussion of solver performance. Section 4 surveys serial and parallel codes for solving MIPs. Section 5 discusses experience with solving the most recent sensor placement models. In particular, we explain why they are not currently suitable for a parallel scalability study. Section 6 discusses serial solution of an older IP model that frequently approximates the performance of the most recent models. Section 7 discusses scalability of parallel solution of this problem. Section 8 shows how exploiting problem structure can accelerate parallel MIP solvers. Section 9 describes possible future enhancements.

2 IP Models for Sensor Placement

Our most recent IP models for sensor placement require as data detailed information about contamination transport during each of a set of possible attack scenarios. Using the EPANET simulator, we determine for each node the first time it is touched by measurable contaminant. We also record also how much contaminant would have been released to the network if a sensor at this node were the first to detect the attack. The IP then places a limited number of sensors to minimize the expected contaminant release over the set of attacks, assuming consumption stops at the moment of detection. We call these dynamic models because they track water dynamics.

The dynamic models precisely predict health effects modulo the quality of the water transport modeling. That quality depends the accuracy of demand data and other inputs to EPANET. Unfortunately, as described in Section 5, this particular model has structural properties that make it less amenable to parallel branch and bound (though it could benefit from some parallelism). Therefore, in this paper we study scalability of the static or nontemporal sensor placement problem introduced in [2]. Berry et. al. [3] show that this model makes predictions that approximately agree with those of the dynamic model roughly three quarters of the time. Therefore, we have used this clean model as a target for scalability studies. All customizations we have developed apply to any sensor-placement problem that has a simple limit on the number of sensors or has cost-based limits with a small number of cost classes.

For completeness, we include a description of the IP as defined and explained in [2]. This is a concrete example of the standard IP just described. The reader might find it easier to consider a specific IP when reading about the branch and bound algorithm for MIP in Section 3.

We model a water network as a graph $G = (V, E)$. E is a set of edges representing pipes. V is a set of vertices, or nodes, where pipes meet. Vertices can

represent sources, such as reservoirs or tanks, where water is introduced, and sinks (demand points) where water is consumed. In general, the network is represented at some scale or granularity, where nodes represent neighborhoods or regions of a city. Each pipe connects two vertices v_i and v_j and is usually denoted (v_i, v_j) .

We consider risk under a fixed number of flow patterns, where we require only the direction of the flow on each edge. Thus a flow specifies for each edge (v_i, v_j) , (connecting vertex v_i to v_j), whether the flow is i -to- j , j -to- i or essentially zero (based on a minimal threshold for the flow). We require the following input data:

- $G = (V, E)$, the network. $V = v_1, \dots, v_n$ and $E = e_1, \dots, e_m$.
- α_{ip} , the probability of an attack at node v_i during flow pattern p conditional on exactly one attack on a node during some flow pattern. We have $\sum_{v_i \in V, p \in 1 \dots P} \alpha_{ip} = 1$, where P is the number of flow patterns.
- δ_{ip} , the density (number of people) at node v_i while flow p is active. $\delta_{ip} = 0$ if node v_i is not a demand node during flow p .
- $f_{ijp} \equiv f_{ep} \in \{0, 1\}$. These parameters describe flow pattern p . $f_{ijp} = 1$ if there is positive flow along (directed) edge $e = (v_i, v_j)$ during flow pattern p and are 0 otherwise.
We have $f_{ijp}f_{jip} = 0$. That is, water cannot flow in both directions of a pipe.
- S_{\max} , the maximum number of sensors we can place.

Given a single attack on node v_i during flow pattern p , a node $v_j \neq v_i$ is contaminated if there is a path from v_i to v_j where all edges have positive flow during flow p and no sensor. More specifically, v_j is contaminated if there is a path $v_i \equiv v_1, v_2, \dots, v_j \equiv v_l$ such that $(v_k, v_{k+1}) \in E$ and $f_{k(k+1)p} = 1$ for all $k = 1 \dots l - 1$ and we place no sensors on any edge in the path. If a demand node v_j is contaminated during flow p , then all the people at node v_j during time p are exposed. We wish to minimize the expected number of exposed people.

The MIP formulation uses the following variables:

- Decision variable $s_{ij} = 1$ if we place a sensor on (undirected) edge (i, j) and 0 otherwise.
A sensor on edge (i, j) detects contaminants moving in either direction. For ease of exposition, we will use both variables s_{ij} and s_{ji} , but they will be equal and, as a pair, represent the placement of only one sensor.
- Derived variables $c_{ipj} = 1$ if node v_j is contaminated by an attack at node v_i during flow pattern p , and 0 otherwise.

The mathematical formulation of the MIP is:

$$\begin{aligned}
(\text{SP1}) \quad & \text{minimize} \quad \sum_{i=1}^n \sum_{p=1}^P \sum_{j=1}^n \alpha_{ip} c_{ipj} \delta_{jp} \\
& \text{where} \quad \begin{cases} c_{ipi} = 1 & \forall i = 1 \dots n, p = 1 \dots P \\ s_{ij} = s_{ji} & \forall i = 1 \dots n-1, i < j \\ c_{ipj} \geq c_{ipk} - s_{kj} & \forall (k, j) \in E \text{ s.t. } f_{kjp} = 1 \\ \sum_{(i,j) \in E, i < j} s_{ij} \leq S_{\max} & \\ s_{ij} \in \{0, 1\} & \forall (i, j) \in E \end{cases}
\end{aligned}$$

The first set of constraints ensures that when a node is directly attacked, it is contaminated. The second set indicates that a single sensor covers a pipe for flow in both directions. The third set propagates contamination from a node v_k to a node v_j if node v_k is contaminated, there is positive flow along a directed edge from v_k to v_j and there is no sensor on that edge. The next constraint enforces the limit on total number of sensors. The final set forces integrality of the sensor-placement decisions. If these variables are set integrally, then the contamination indicator variables c_{ipj} are also integral, even though they are not explicitly forced to be binary values in the MIP. The objective function exerts pressure to minimize these variables. The first and third set of constraints propagate values of 1 whenever there are no sensor to prevent the propagation.

3 Branch and Bound for Integer Programming

All general integer programming solvers are based on the branch and bound (B&B) and/or branch and cut algorithms. We'll now describe the fundamental algorithms applied to solve general MIPs. See, for example, [1] for a more general description of (parallel) branch and bound.

Basic B&B iteratively subdivides the feasible region (the set of all x 's that satisfy the linear and integrality constraints) and recursively searches each piece. B&B is often more efficient than straight enumeration because it eliminates regions that provably contain no optimal solution. For a minimization problem, it computes a lower bound on the value of the optimal solution in each subregion. If this bound is worse (higher) than the value of the *incumbent* (the best feasible solution found so far), then there is no optimal feasible solution in the subregion. We now describe bounding and splitting methods for general MIPs. One can modify these and other basic B&B pieces to exploit problem-specific structure.

Every MIP has a natural, usually-nontrivial bound. If we relax (remove) the integrality constraints, a MIP becomes a linear program (LP). The optimal value of this *LP relaxation* is a lower bound for a minimization problem and an upper bound for a maximization; we assume minimization for this discussion. If the LP solution coincidentally obeys all integrality constraints, then it is an optimal solution to the MIP as well. LPs are theoretically solvable in polynomial time [15] and are usually solved efficiently in practice with tools described below.

The B&B algorithm grows a search tree as follows. The first incumbent value is infinity. The initial problem is the root r . Compute the LP bound $z(r)$ for the root. If the LP is infeasible, then the MIP is infeasible. If the LP solution is integer feasible, this is an optimal solution for the MIP. Optionally, search for an

incumbent using either a problem-specific heuristic that exploits problem structure or a general MIP heuristic method. If the incumbent value matches the lower bound $z(r)$, then we can *fathom* (eliminate) the node; it contains nothing better than the incumbent. Otherwise *branch*, or *split* the problem. If the LP relaxation is not integer feasible, there is some $j \in D$ such that the optimal solution to the LP relaxation x^* has $x_j^* \notin \mathcal{Z}$. Create two new sub-MIPs as children of the root: one with the restriction $x_j \leq \lfloor x_j^* \rfloor$ and one with the restriction $x_j \geq \lceil x_j^* \rceil$. For binary variables, one child has $x_j = 0$ and the other has $x_j = 1$. The feasible regions of the two children are disjoint and any solution with $\lfloor x_j^* \rfloor < x_j < \lceil x_j^* \rceil$, including x^* , is no longer feasible in either child. Thus the LP relaxation of a child differs from the LP relaxation of the parent. Recursively solve each subproblem. At any point in the computation, let P be the pool of active (unresolved) subproblems. Then $L = \min_{p \in P} z(p)$ is a global lower bound on the original problem. B&B terminates when there are no active subproblems or when the relative or absolute gap between L and the incumbent value is sufficiently small.

Splitting causes exponential work explosion in cases where the B&B strategy fails. Therefore most MIP systems use *branch-and-cut*: they add general and/or problem-specific valid inequalities (cutting planes) to improve the lower bound on a subproblem to delay branching as long as possible (or avoid it altogether if the lower bound rises above the incumbent value). Given an optimal non-integer solution to the LP relaxation of a (sub)problem, x^* , a cutting plane is a constraint $ax = b$ such that $ax' = b$ for all possible (optimal) integer solutions x' but $ax^* \neq b$. Adding this constraint to the system (cutting) makes the current LP optimal infeasible. The branch-and-cut algorithm processes a subproblem by solving the LP relaxation, finding and adding cuts, resolving the LP, and iterating until cutting becomes too difficult or unproductive. Finally the node splits.

4 IP and LP Software

ILOG's commercial MIP solver CPLEX [10] is excellent for moderate-sized instances. CPLEX runs in serial or on small SMPs (symmetric multiprocessors, machines with a few processors and large shared memory). For example, CPLEX can run in parallel on an integrated dual-processor system. There are a number of free serial solvers such as MINTO [18], ABACUS [14], lp_solve, and GLPK (Gnu Linear Programming Kit), CPLEX dominates all of these in practice.

ILOG (the company that develops CPLEX) has invested in a huge research effort that includes proprietary cut management algorithms. In general, if one has paid for a serial (or small-scale SMP) CPLEX license, and CPLEX can solve a problem, then it is currently the best solution option.

There are a number of parallel MIP solvers, none of them commercial. SYMPHONY [20], and COIN/BCP [16, 17] are designed for small-scale, distributed-memory systems such as clusters. BLIS [22], under development, is designed as a more scalable version of SYMPHONY and BCP. It will be part of the optimization software available through COIN-OR (Computational Infrastructure for Operations Research [8]) once it is available. Ralphs *et. al.* discuss of all three [21].

FATCOP [7] is designed for grid systems. The grid offers vast computational resources, but it is not a suitable “platform” for sensitive computations such as those involving national security or company proprietary information.

For our parallel tests, we use our own parallel MIP solver PICO [12] (Parallel Integer and Combinatorial Optimizer). The parallel branch-and-bound search strategy in PICO is particularly well-suited for solving MIPs on tightly-coupled massively-parallel distributed-memory architectures, such as those available at the national laboratories. This is the only parallel code explicitly designed for such a platform. Because it's our own code, we can easily tune search parameters.

PICO has a number of features designed for efficient branch and bound on massively-parallel machines. See [12, 11] for more details. In particular, PICO has an initial *ramp-up phase* followed by a *parallel-subproblem phase*. In the ramp-up phase, there are few open subproblems relative to the number of processors, so the processors cooperate to solve a single subproblem at a time. This solves each subproblem faster than it would have been solved in serial. Furthermore, it ensures careful branching decisions early in the computation when they are most important. Once there is enough work to keep processors busy or when individual subproblems no longer have sufficient parallelism, PICO switches to a phase where processors solve separate subproblems in parallel.

We used the AMPL modeling language [13] to translate the mathematical description of a general integer programming problem (plus a data file customizing the problem to a particular instance) into the format integer programming solvers require. Mathematical programming languages like AMPL allow developers to express a MIP using natural (multidimensional) variables rather than the usual linear representation of general MIP solvers. AMPL models are a formalization (almost a direct translation) of a mathematical MIP representation. For example, there are classes of variables and classes of constraints. PICO provides scripts that accept an AMPL MIP model and automatically generate derived PICO classes that are aware of the AMPL names. A developer can then write incumbent heuristics and other customized procedures using the natural problem variables. AMPL is a commercial code. MathProg is a free code that can process AMPL models, but it's not as fast as AMPL.

Linear programs have no integrality constraints. They're usually solved with iterated linear algebra computations. Commercial tools such as CPLEX [10] LP solver, XPRESS [26], or OSL [19] are generally much faster than free tools such as COIN-LP [8]. The difference is most pronounced for the root problem (computing the LP relaxation of the full original problem). Computing LP relaxations for the other subproblems is generally at least an order of magnitude faster than solving the root because solvers can use information about the closely-related (solved) parent problem to re-bound a child node. Thus CLP is usually sufficient to bound these subproblems. PICO (or any highly parallel IP code) must use free solvers because there are no commercial licenses for this level of parallelism (if there were, it would be prohibitively expensive). PICO has a mechanism whereby one can solve the root problem using a serial commercial license, then pass the solution into a parallel computation that uses CLP to bound the subproblems.

A parallel interior point solver such as pPCx [9] could provide moderate speed up at the root. The core computational problem is the solution of a linear system of the form $AD^2A^T x = b$ where A is the original constraint matrix and D is a diagonal matrix that changes each iteration. Parallel direct Cholesky solvers for such systems are robust, but currently do not provide reasonable speed up beyond a few dozen processors. Sandia National Laboratories is leading a research effort to find more scalable interior-point LP solvers using iterative linear systems solvers [5]. There are open research problems in solving this linear system iteratively and in computing the information needed to effectively solve the subproblems.

5 Solving High-Fidelity MIP formulations

The dynamic IP formulation explicitly models water flow in a network and would have been our formulation of choice for this research. In practice, however, the LP relaxation of the root has an integral setting of the decision variables. Thus usually the IP “search tree” has only one node. The problem is still quite challenging to solve, but it cannot benefit from PICO-style parallelism in its simplest form. Its solution requires scalable parallel linear programming solvers such as the research code parPCx under development at Sandia National Laboratories [5].

6 Serial Computation

In the experiments we ran for this paper, serial (or dual-processor threaded) computations with CPLEX were generally about one to two orders of magnitude faster than equivalent runs with PICO (where PICO did not use cutting planes). However, as problems become larger or models become more complex, CPLEX is eventually unable to solve them.

We used three data sets for our experiments. The first set is an EPANET test set with 97 nodes and 117 pipes. The second is a local water network with 470 nodes and 621 pipes. The third is a skeletonized model of a real city with 3648 nodes and 3803 pipes. CPLEX generally solves the 97-node problem in about 35 seconds (after problem loading) on a linux workstation (3.06Ghz Xeon with Gb of RAM). It solves the 470-node problem in about 9.5 minutes. However, solving the large problem for 25 sensors requires 42.5 hours using two processors on a 64-bit linux workstation with 2.2 Ghz AMD Opteron 848 processors. If we modify the problem to place sensors on nodes rather than edges, the IP becomes even harder.

7 Parallel Speedup

A sufficiently scalable IP solver should be able to solve some problems too difficult for commercial serial solvers. To test the scalability of parallel computations for this IP, we solved the 97-node data set using PICO on a 32-node, 64-processor cluster of Xeon 2.8Ghz processors with 2Gb of RAM connected with gigabit ethernet. PICO computed bounds with the free LP solver CLP, which is considerably slower than commercial LP solvers. We ran without PICO’s built-in general-purpose incumbent heuristic because it found no incumbents and added to the processing time. This instance has unusual structure. The search tree is small and quite stable regardless of the number of processors. Thus there are frequently

not enough unsolved subproblems to keep large numbers of processors busy on individual problems. We set a solver parameter to avoid crossing into the parallel phase until there were at least two available subproblems for each processor. As a consequence, the computation never left ramp up. There is considerable work at the root and at some later subproblems computing gradients in order to make wise branching choices. This work is (almost) fully parallelizable during ramp up and is the source of the speed up we observed. But parallel speed up is limited by the constant fraction of nodes that did not require gradient initialization. We show the results in the “No Heuristic” columns of table 1. In problems with relatively easy LP solves and lots of subproblems, we normally observe higher speed up.

8 Exploiting Structure

One way to accelerate IP solvers is to use problem-specific information. In this section we describe a simple incumbent heuristic for general sensor placement problems using randomized rounding. An incumbent heuristic finds a feasible integer solution quickly. It speeds the search by allowing early pruning. Also, it gives a bound on the minimum quality of a feasible sensor placement.

Randomized rounding has been used to compute provably good approximations for some combinatorial optimization problems[23]. It’s a natural idea in its simplest form. Suppose all integer variables are binary. In the LP relaxation x^* of the IP, for each decision variable x_i , we have $0 \leq x_i^* \leq 1$. Treat each value x_i^* as a probability and round the variable x_i to 0 or 1 based on this probability. Specifically, generate a random number r_i between 0 and 1 for each variable x_i . If $r_i \leq x_i^*$, set $x_i = 1$ and otherwise, set $x_i = 0$. One must then compute the values of the other variables, for example, by resolving the LP with the new values of the decision variables. This is not a good strategy for a general MIP because the computed solution is almost never feasible for the linear constraints $Ax = b$. However, we can exploit the special structure of the fundamental sensor placement problem FSP to use randomized rounding effectively.

For the FSP integer program, any sensor placement is feasible as long as it uses no more than S_{\max} sensors. Of course, some of these sensor placements will have terrible performance. If we were to randomly select over all size- N subsets of sensor locations, we wouldn’t expect to obtain a good solution. However, we hope that the locations the LP picks fractionally have some overall merit. We wish to select a size- S_{\max} subset appropriately biased by the LP values.

Let \mathcal{P} be a probability distribution in the size- S_{\max} subsets of sensor placements defined as follows. Randomly and independently select each variable x_i with probability x_i^* . We call this a *pass*. If we select exactly S_{\max} sensor locations in the pass, choose that set. Otherwise do another pass. For a set S with $|S| = n$, $\mathcal{P}(S)$ is the probability this procedure will select S .

This rounding procedure generally requires an unacceptable amount of time. For an LP relaxation x^* , we will generally have $\sum_i x_i^* = S_{\max}$, so the expected number selected in each trial is S_{\max} . However, there is still a tiny probability of selecting a set of size exactly S_{\max} .

Our algorithm efficiently directly selects a size- S_{\max} subset according to the

Processors	Heuristic (CH)	CH Speed up	No Heuristic (NH)	NH speed up
1	4743		4587	
2	2782	1.70	3165	1.45
4	1129	4.20	1677	2.74
8	685	6.92	1010	4.54
16	366	13.0	640	7.17
32	252	18.8	446	10.2

Figure 1: Parallel runs for a 97-node data set using an incumbent heuristic for general sensor placement problems and using no heuristic. This is wallclock time in seconds, which includes time to send the problem to the parallel machines.

probability distribution \mathcal{P} without iterating. If there are L possible locations, then the procedure requires $O(LS_{\max})$ base computations (that is, roughly a constant times LS_{\max} computations) to build a data structure. Then it selects a subset using a single random number in $O(L)$ additional time. Because the decoding is so fast, it’s generally worth selecting several random sensor locations and choosing the best one. For this paper we tried only one location per node. A detailed description is beyond the scope of this paper.

We applied this customized incumbent heuristic to the 97-node data set. The problem always solved during ramp up even with default parameters. It had good speed up through 16 processors as shown in Table 1. Finding an incumbent earlier in the search allowed earlier pruning so these runs usually produced search trees with at most 13 nodes, while the no-heuristic runs had trees with about 60 nodes. Thus the runs with the heuristic were generally at least 1/3 faster than corresponding runs without the heuristic. We also saw (relative) superlinear speed up for the 470-node data set. This requires 45000 seconds with 8 processors, but only 16800 for 16 processors.

9 Future Improvements

There are a number of ways one can further customize IP computations for sensor placement problems. For specific formulations, one can find new classes of cutting planes to improve lower bounds. One can might be able to determine locations that must or cannot have sensors in an optimal solution in a preprocessing phase.

We have developed a new branching strategy for general sensor placement problems that targets the constraint on the number of sensors. Generally the number of sensors is small compared to the number of possible locations, so an LP relaxation can assign tiny fractional values to variables. Branching on a single sensor decision variable creates a weak child (the one restricted to not place a sensor in some location). The resulting IP is almost the same as the parent. We generalize an idea called special ordered sets, used in the IP community to improve branching when one can only select a single variable from a set of candidates. We branch on constraints that restrict the number of sensors allowed in a set of locations. By using specialized four-way branch we create children all of which have LP relaxations that differ significantly from the parent. In many cases, the new

constraints are special-ordered sets. PICO does not yet have the infrastructure to customize branching to this degree. However, we plan to add this.

We are actively developing PICO, continuously improving its performance. However, its niche will still largely be to solve problems we cannot solve in other ways. Thus we will soon add explicit external memory management to PICO. This allows the computation to use external disks to increase its memory and hence the size of the active subtree it can manage. Parallel platforms frequently have less memory per node than regular workstations. Because external disks are reasonably cheap and have high capacity, this feature could enable solution of problems that fail due to lack of memory even on 64-bit workstations.

References

- [1] D. Bader, W. Hart, and C. Phillips. Parallel algorithm design for branch and bound. In H. J. Greenberg, editor, *Tutorials on Emerging Methodologies and Applications in Operations Research*. Kluwer Academic Press, 2004.
- [2] J. Berry, L. Fleischer, W. E. Hart, C. A. Phillips, and J.-P. Watson. Sensor placement in municipal water networks. *J. Water Planning and Resources Management*, 2005. (to appear).
- [3] J. Berry, W. Hart, C. Phillips, J. G. Uber, and J-P Watson. Validation and assessment of integer programming sensor placement models. In *Proceedings of the World Water and Environmental Resources Conference*. ASCE, 2005.
- [4] J. Berry, W. E. Hart, C. A. Phillips, and J. Uber. A general integer-programming-based framework for sensor placement in municipal water networks. In *Proceedings of the World Water and Environment Resources Conference*, 2004.
- [5] E. G. Boman, O. Parekh, and C. Phillips. LDRD final report on massively-parallel linear programming: the parPCx system. Technical Report SAND2004-6440, Sandia National Laboratories, February 2005.
- [6] R. Carr, H. J. Greenberg, W. E. Hart, and C. A. Phillips. Addressing modeling uncertainties in sensor placement for community water systems. In *Proceedings of the World Water and Environment Resources Conference*, 2004.
- [7] Q. Chen and M. C. Ferris. FATCOP: A fault tolerant Condor-PVM mixed integer programming solver. *SIAM Journal on Optimization*, 11(4):1019–1036, 2001.
- [8] Computational INfrastructure for Operations Research home page, 2004. <http://www.coin-or.org/>.
- [9] T. Coleman, J Czyzyk, C. Sun, M. Wager, and S. Wright. pPCx: Parallel software for linear programming. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.
- [10] ILOG, CPLEX home page, 2004. <http://www.ilog.com/products/cplex/>.
- [11] J. Eckstein, W. Hart, and C. Phillips. Massively-parallel mixed-integer programming: algorithms and applications. In M. A. Heroux, P. Raghava, and H. D. Simon,

editors, *Frontiers of Parallel Processing for Scientific Computing*, Software, Environments, and Tools. SIAM Press, 2005.

- [12] J. Eckstein, W. E. Hart, and C. A. Phillips. PICO: An object-oriented framework for parallel branch-and-bound. In *Inherently Parallel Algorithms in Feasibility and Optimization and Their Applications*, Elsevier Scientific Series on Studies in Computational Mathematics, pages 219–265, 2001. PICO has evolved considerably since the time of this report. We hope to write an updated report soon.
- [13] R. Fourer, D.M. Gay, and B.W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Boyd & Fraser Publishing Company, 1993.
- [14] Michael Jünger and Stefan Thienel. The ABACUS system for branch-and-cut-and-price algorithms in integer programming and combinatorial optimization. *Software: Practice and Experience*, 30:1325–1352, 2001.
- [15] L. Khachian. A polynomial time algorithm for linear programming. *Soviet Mathematics, Doklady*, 20:191–194, 1979.
- [16] L. Ladányi. BCP (Branch, Cut, and Price). Available from <http://www-124.ibm.com/developerworks/opensource/coin/>.
- [17] F. Margot. BAC: A BCP based branch-and-cut example. Technical Report RC22799, IBM, 2003.
- [18] G. L. Nemhauser, M. W. P. Savelsbergh, and G. C. Sigismondi. Minto, a mixed integer optimizer. *Operations Research Letters*, 15:47–58, 1994.
- [19] IBM Optimization Solutions and Library, home page, 2004. <http://www-306.ibm.com/software/data/bi/osl/>.
- [20] T. K. Ralphs. Symphony 4.0 users manual, 2004. Available from www.branchandcut.org.
- [21] T. K. Ralphs, L. Ladányi, and M. J. Saltzman. Parallel branch, cut, and price for large-scale discrete optimization. *Mathematical Programming*, 98(1-3), 2003.
- [22] T. K. Ralphs, L. Ladányi, and M. J. Saltzman. A library for implementing scalable parallel search algorithms. *The Journal of SuperComputing*, 28(2):215–234, 2004.
- [23] Aravind Srinivasan. Approximation algorithms via randomized rounding: A survey. citeseer.ist.psu.edu/493290.html.
- [24] J.-P. Watson, H. J. Greenberg, and W. E. Hart. A multiple-objective analysis of sensor placement optimization in water networks. In *Proceedings of the World Water and Environment Resources Conference*, 2004.
- [25] J.-P. Watson, W. E. Hart, and J. Berry. Scalable high-performance heuristics for sensor placement in water distribution networks. In *Proceedings of the World Water and Environmental Resources Conference*. ASCE, 2005.
- [26] Dash Optimization, XPRESS-MP, 2004. <http://www.dashoptimization.com/>.