Solution of Mixed-Integer Programming Problems on the XT5

*Rebecca J. Hartman–Baker**
National Center for Computational Sciences

Ingrid K. Busch
Center for Transportation Analysis

Michael R. Hilliard
Center for Transportation Analysis

Richard S. Middleton
Center for Transportation Analysis

Michael S. Schultze
Center for Transportation Analysis

Oak Ridge National Laboratory, Oak Ridge, Tennessee

ABSTRACT: In this paper, we describe our experience with solving difficult mixed-integer linear programming problems (MILPs) on the petaflop Cray XT5 system at the National Center for Computational Sciences at Oak Ridge National Laboratory. We describe the algorithmic, software, and hardware needs for solving MILPs and present the results of using PICO, an open-source, parallel, mixed-integer linear programming solver developed at Sandia National Laboratories, to solve canonical MILPs as well as problems of interest arising from the logistics and supply chain management field.

KEYWORDS: mixed-integer linear programming, XT5

1 Introduction

Mixed-integer linear programs arise in many applications, including logistics, supply chain analysis, and data mining. For example, consider a mathematical model used to determine the optimal production level for a computer factory. While the optimal solution might recommend producing 2398.72 computers, in reality we cannot produce a fractional number of computers. In this case we can simply round the solution to the nearest integer and be satisfied, but for other problems we cannot do that. If we are modeling the placement of biorefineries, a solution indicating that we should build 0.6 refineries at one location and 0.35 refineries at another is essentially meaningless.

Mixed-integer linear programs are computationally intensive and very challenging, but these problems are seldom solved on a supercomputer. As a result, practitioners are limited to solving problems that can fit on their laptop or desktop machines. In this paper, we discuss the problem formulation, parallelization of mixed-integer programming solvers, and some preliminary results of our work on solving large mixed-integer linear programs on the XT5.

2 Problem Formulation

Linear programming is a method of solving a constrained optimization problem in which the objective function and the constraints are all linear. Mathematically, we have

$$\min_x c^T x \text{ subject to } Ax \leq b. \quad (1)$$

We seek a solution that minimizes the inner product of c and x within the polytope defined by the linear constraints, or the *feasible region*. The area outside the polytope is known as the *infeasible* region.

In a linear programming problem, the elements of x can take any real value. In a *mixed-integer linear programming* problem, some of the elements of x are constrained to be integral. The problem then takes the form

$$\min_x c^T x \text{ subject to } Ax \leq b,\ x_j \in \mathcal{Z}\ \forall j \in D, \quad (2)$$

*National Center for Computational Sciences, Oak Ridge National Laboratory, One Bethel Valley Road, P.O. Box 2008, MS–6008, Oak Ridge, TN 37831-6008, *hartmanbakrj@ornl.gov*.

where D is the set of indices of the integer variables in x.

2.1 Solution of Linear Programming Problems

The canonical method for linear programming problems is known as the *simplex method*. The simplex method is an iterative method that systematically examines the vertices of the polytope defining the feasible region (the region in which the solution must lie due to the constraints). See Figure 1 for an illustration of an example two-dimensional linear programming problem.

The optimal solution must lie at one of the vertices of the polytope. Since the objective function is linear, the solution must lie on one of the edges of the polytope. If we found an "optimal" solution inside the polytope, we could always find a better one by following the direction of steepest descent of the objective function. The only way we would be unable to do that is if our optimal solution lay on the edge. Furthermore, assuming that the problem is not degenerate, the optimal solution must be at a vertex of the polytope, because once we have traced to an edge and can no longer follow the gradient of the objective function, the projection of the steepest descent upon the edge is a feasible direction that further decreases the value of the objective function.

In the first step of the simplex method, we find an initial feasible solution, determine along which edge defining our vertex we can decrease the value of the objective function, and proceed along that direction and find the next feasible solution. We continue in this manner to traverse the polytope, until we reach a point at which all possible directions increase the value of the objective function. At this point, we have found the minimum and solved the problem.

The method of finding an initial feasible solution varies. For some problems, it is simple to find an initial feasible solution, but for those that are more complicated, the usual methods are to either solve an auxiliary linear program or to add a penalty against infeasibility to the problem [6].

The simplex method works well in practice, but in the worst case, a problem with n variables and m constraints could require the examination of $\binom{n}{m}$ vertices. The Klee–Minty problems, developed in 1972, are an example of such problems [4].

Typically, however, the simplex method performs much better than that. On average, we would expect to find a solution in an amount of time proportional to a polynomial in n and m: in fact, the average number of iterations is bounded by $O(\min\{(m-n)^2, n^2\})$ [6].

The uncertainty associated with the worst-case performance of simplex methods prompted the development of alternative linear programming solution methods. Interior-point methods, first brought into prominence by Karmarkar [3], are guaranteed polynomial time algorithms. Closely related to barrier methods, a class of nonlinear optimization algorithms, interior-point methods iterate through strictly feasible points (points within the polytope, not lying on the boundary). The computational complexity of interior-point methods is $O(n^3)$, where n is the number of variables. This is much better than the worst-case complexity of the simplex method, but on average, the simplex method outperforms interior-point methods [6].

2.2 Solution of Mixed-Integer Linear Programming Problems

Mixed-integer linear programming problems (MILPs) are much more challenging than linear programming problems with purely continuous variables. In fact, they are NP complete, so we must rely on heuristics.

Most MILP solvers rely on the *branching* concept. The idea behind branching is that we can divide up our discrete domain into pieces. For example, if an integer variable is constrained to the range $[L, U]$, then we could divide our domain into two pieces, $[L, \lfloor (L+U)/2 \rfloor]$ and $[\lceil (L+U+1)/2 \rceil, U]$, and explore these two spaces separately. Combining this branching with a means of computing lower (and optionally, upper) bounds, we can develop an optimization method known as *branch and bound*.

2.2.1 Branch and Bound

In a branch and bound method, we seek to improve the global upper and lower bounds by systematically performing bounding and branching operations on subproblems.

Typically, we use a relaxation of the MILP as the bounding function, meaning that we solve the LP resulting from relaxing the integrality constraints on the integer variables. This is a viable bounding because the integer solutions must lie within the LP polytope — integrality constraints restrict the viable

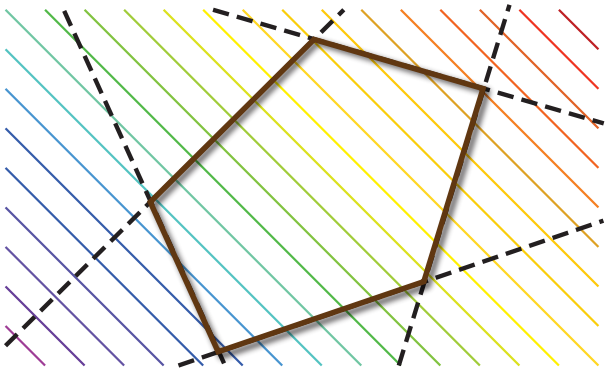

Figure 1: Example linear programming problem. Diagonal lines represent gradient of objective function. Dashed lines represent constraints. Pentagon represents polytope bounding feasible region.

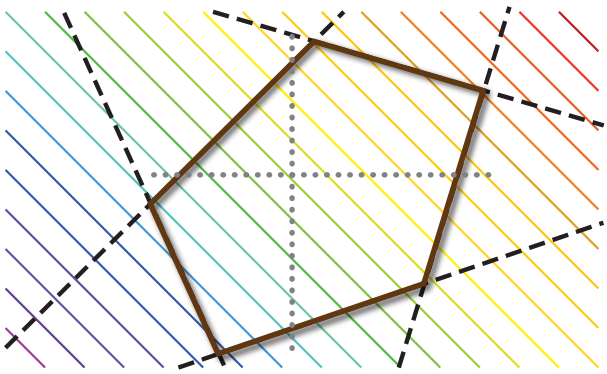

Figure 2: Example of branching. The feasible region within the pentagon is divided into four pieces (dotted lines represent cuts).

solutions to a region contained by the polytope. So if we solve the LP for its minimum, we will obtain a reasonable lower bound on the objective function.

We can use the bounding functions to eliminate nonviable pieces of the domain (nodes in the branching tree) from consideration. In other words, if the lower bound on a given node is greater than the value of the best feasible solution we have found so far, then we can eliminate that node from consideration, because there is no way that it can contain the optimal solution. We can then subdivide other viable nodes further, and continue to refine the viable regions of the problem domain recursively, stopping only when our upper and lower bounds equal one another (or are within a certain tolerance of one another).

The remaining element of the algorithm to be defined is the search strategy. We need to examine the branches in a systematic manner, which we hope will reduce the amount of time spent on futile searches. A common method used is known as *best-first search*, in which we choose to process branches with the smallest lower bounds first. This strategy reduces the number of branches that are searched, at the sacrifice of incremental improvements to the upper bound on the solution. *Depth-first search*, on the other hand, chooses the next branch to search based on its depth in the tree. Diving to the depths of the tree may result in many suboptimal solutions early in the search process, but it permits more improvement to the upper bound than does best-first search. Other search strategies have been developed that attempt to make a strategic selection of nodes, such as *best-projection* and *best-estimate* methods. These estimate-based methods select nodes based on measures of node quality, such as lower bounds, degree of infeasibility, and estimates of the optimal solution to the subproblem [7].

2.2.2 Branch and Cut

Often, the lower bound generated by the relaxation of a MILP to its corresponding LP is too loose to allow for efficient solution of the problem. In many cases, we can improve the bound by adding valid inequalities to the problem as we go, strengthening the LP relaxation [7]. This technique is known as *branch and cut*. For example, in Figure 3, we can add the red dotted-dashed line to the MILP without excluding any feasible points, and tighten the bounds.

2.3 Parallelization of MILP Solvers

A naive parallelization of the branch and bound method would subdivide the feasible region and distribute these branches across all processes. Each process can find the local optimum within its branch, and communicate at the very end of the computation, at which point the optimum is determined.

But this method results in a lot of redundant work. One process (its identity unknown at the beginning of the computation) holds the branch containing the optimal solution, so the work done by all the rest of the processes is futile. This approach results in little speedup. In fact, in a worst case scenario, it could result in slowdown: perhaps the solution of the optimum in one of the branches not containing the minimum takes a lot longer than the branch containing the optimum.

In fact, the good parallelization of MILP solvers is much more complicated than that. There are many factors to consider, and tradeoffs to balance, but the parallel algorithm can be broken into three main steps: ramp-up, search, and ramp-down.

2.3.1 Ramp-Up Step

In the ramp-up step, we break the feasible region into pieces until there is enough work to spread across all processes. As implied by the definition of this step, there is extensive idle time during this phase of the computation, at least until we break up the domain into enough pieces.

The primary difficulty is the fact that processing a single node can be large relative to the overall solution time. We can occupy the idle processes with other tasks that improve the search, such as problem preprocessing and computation of upper bounds. These tasks may not be worth the time taken to perform them, however, so ideally we would prefer to shorten the ramp-up phase rather than occupy processes with auxiliary tasks.

Unfortunately, no good ramp-up acceleration techniques have been developed. One idea is to parallelize the branching process itself by distributing pre-solve candidates across processes. Another idea is to use a branching scheme that subdivides the problems into more than two subproblems. This may increase the size of the tree, but it could result in candidate nodes that much more quickly [7].

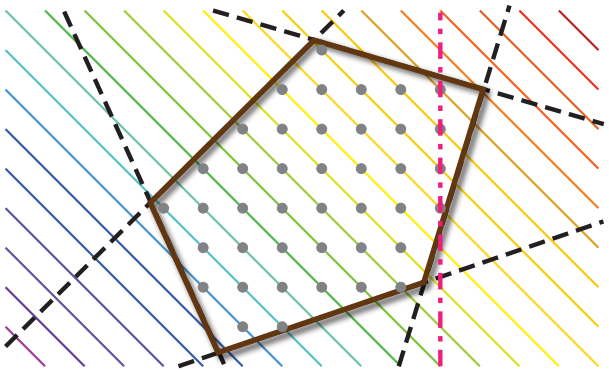

Figure 3: Example of cutting. The feasible region can be better bounded by the red dotted-dashed line, tightening the bound without removing any of the possible integer variable feasible points (gray dots).

2.3.2 Search and Process Phase

The main body of the algorithm involves searching and processing branch nodes, and using and sharing that information (also known as *knowledge management*). How can we best generate and share node descriptions and valid inequalities? There are two approaches to knowledge management: centralized and decentralized control. A single process controlling the algorithm has a much clearer global picture of the state of the solution than would corresponding decentralized hubs that each had a piece of the picture. But the manager-worker algorithm is not a scalable approach, so while more work may be performed by a decentralized algorithm, it may be the more efficient choice.

In choosing a search strategy, we now have additional considerations pertaining to parallelization. It would be difficult to execute a search strategy in parallel precisely as it is executed in serial. Global best-first search, for example, is impractical for more than a handful of processes. It might be a good local strategy to adopt, however.

Thus we now have two aspects of the search strategy to consider: global and local. The global strategy would determine how we best shift nodes between processes in order to pursue our search strategy. The local strategy would determine how to process nodes locally.

Working hand in hand with our search strategy is our load balancing strategy. In a centralized algorithm, it is relatively simple to control the movement of work and ensure balance. A decentralized algorithm, however, poses problems. Perhaps a local hub could load balance well within the context of the nodes available to it and its subset of processes, but additionally we need to implement a global policy to meet the global needs of the algorithm.

2.3.3 Ramp-Down Step

The ramp-down step, symmetric to the ramp-up step, faces similar problems with an insufficient workload to occupy all available processors. While this step's contribution to parallelization is usually ignored, it can actually lead to serious problems with scalability [7].

3 Existing MILP Solvers

3.1 CPLEX

CPLEX is the leading commercial package in the linear programming field.[1] The mixed-integer optimizer contained within CPLEX can solve mixed-integer linear programs quite rapidly in serial. It includes mixed-integer problem reduction algorithms, user-defined branching priorities, node selection algorithms, and variable selection options. A parallel version can run across many cores in a node, and while its parallelization strategy yields good speedup in many cases [5], this is of little help to those who wish to run on a distributed memory machine.

[1]ILOG CPLEX URL: `http://www.ilog.com/products/cplex/`

A distributed parallel extension to CPLEX, called ParaLEX, was developed by Shinano and Fujie [8]. Their parallel implementation, a somewhat primitive manager-worker algorithm implemented in MPI, was tested on up to thirty parallel processes, and achieved superlinear speedup on a few problems, and modest speedup on the rest.

3.2 Gurobi

A new commercial solver, Gurobi, has just been released in the spring of 2009.[2] Named after its three developers, Robert Bixby, Zonghao Gu, and Edward Rothberg, Gurobi is competitive with CPLEX on many problems [5]. Like CPLEX, Gurobi is also capable of parallelizing across multi-core processors.

3.3 COIN-OR

The purpose of the Computational Infrastructure for Operations Research (COIN-OR) project is to spur the development of open-source software for the operations research community.[3] Many packages are available, including several solvers for linear programming and mixed-integer linear programming problems. In addition, there are solvers for other classes of problems (e.g., mixed-integer nonlinear programs, sequential quadratic programming problems).

3.3.1 SYMPHONY

SYMPHONY, available in the COIN-OR project, can be used as a generic MILP solver, a callable library, or extensible framework for implementing custom MILP solvers. It has a large arsenal of solution techniques at its disposal, an OpenMP shared-memory implementation, and a distributed parallel implementation in PVM.

SYMPHONY takes a centralized approach to knowledge management.

3.3.2 CHiPPS

The COIN-OR High-Performance Parallel Search Framework (CHiPPS) is a framework for developing parallel tree-search algorithms. It is implemented in three layers:

1. The *Abstract Library for Parallel Search* (ALPS) layer implements the search handling methods needed for large-scale, data-intensive parallel search algorithms.
2. The *Branch, Constrain, and Price Software* (BiCePS) layer, built on top of ALPS, is responsible for the data-handling capabilities needed for relaxation-based branch and bound algorithms.
3. The *BiCePS Linear Integer Solver* (BLIS) layer, built on top of BiCePS, is an instantiation of BiCePS in which the relaxation method is linear programming.

BLIS takes a highly decentralized approach to knowledge management[7]. The global list of candidate nodes are spread across all processors. Every processor that processes nodes has its own local node pool from which it selects new candidates for processing. Load balancing is performed using a three-level master/hub/worker paradigm.

3.4 PICO

PICO is a distributed parallel MILP solver package that is capable of scaling up to thousands of processors [1]. It leverages the computing power of libraries from COIN-OR that perform the underlying solves. The PICO framework parallelizes by splitting the tree into its branch components, distributing the work, and eliminating dead-ends.

PICO's parallelization takes a hybrid approach, similar to BLIS. Initially, the domain is broken into branches and the branches are distributed across the processes. As branches are found to be nonviable, the idle processes are given sub-branches from processes with still-viable branches. The model is that of a manager-worker algorithm. However, because this is not scalable to thousands of processors, hubs (local managers) are spawned, which communicate to their local worker processors and to other hubs on behalf of themselves and their local worker groups.

4 Results

We tested PICO and BLIS on two sets of problems. The first set was a collection of standard MILP test problems of varying sizes and difficulty. The second set consisted of 27 problems concerning the placement of facilities in Canadian cities.

[2]Gurobi Optimization URL: `http://www.gurobi.com/`
[3]COIN-OR URL: `http://www.coin-or.org/`

All runs were performed on Jaguar, a 1.4 PF Cray XT5 with more than 100,000 processors at the National Center for Computational Sciences at Oak Ridge National Laboratory.

4.1 Standard Test Problems

These problems were standard test problems that come with the BLIS distribution. Problems vary in size with the number of unknowns ranging from roughly 100 (bell5) to nearly 7200 (air05). Properties of these problems are shown in Table 1.

4.1.1 Performance of BLIS

We obtained results, as summarized in Figure 4, with BLIS. Due to time constraints, we were unable to obtain consistent results for the last four test problems in Table 1, so they are not reproduced here.

The overall trend for the performance of BLIS on these problems is downward, so we find this promising. It is interesting to note that despite the fact that air05 is the largest problem, for some processor counts a considerably smaller problem, fc.60.20.3, actually takes longer.

4.1.2 Performance of PICO

With PICO, we were unable to solve fc.60.20.3, but otherwise successfully solved every problem. The results for PICO are summarized in Figure 5.

For the problems both PICO and BLIS solved, PICO outperformed BLIS, as depicted in Figure 6.

4.2 Canadian Cities Problems

These problems were challenging, but small enough that they can be solved in serial with CPLEX on a desktop machine. Problem sizes varied with the size of the cities, from 26,000 (in the case of Halifax) to 2.4 million (in the case of Montreal) unknowns and constraints. Integer variables made up roughly 6—55% of the total number of variables, with larger problem sizes having proportionally fewer integer variables, although the number of integer variables was larger for larger problem sizes.

4.2.1 Performance of BLIS

Two of the problems, Calgary and Montreal, failed at the point of file input when runs were attempted. For the rest of the problems, we ran them three times for each processor count and took the average of the walltimes to determine scalability. Processor counts varied from 2 to 128. The results for all the problems are summarized in Figure 7.

The overall trend is no speedup. In Figure 8, three results have been extracted. The results for the city of North York show a modest speedup. Edmonton exemplifies the trend, remaining basically constant independent of the number of processors. Vancouver results are highly irregular. It seems that at 16 processes, the worst branches were chosen, and the best performance is actually for two processors.

The reason for this dismal speedup is because the majority of the time is spent in ramp-up. Ramp-up is essentially serial. Additionally, the more processors there are, the more spawning of branches are required before we can leave the ramp-up phase.

4.2.2 Performance of PICO

PICO was less robust than BLIS. Problems PICO was unable to solve included Calgary, Edmonton, Mississauga, Montreal, North York, Toronto, and Vancouver. Its difficulties with these problems led to different behaviors: some, such as Calgary, resulted in immediate segmentation faults, while others, such as Toronto, were brought down partway through the computation by apparent memory leaks.

Furthermore, even when PICO was able to solve a problem, it did so several times slower than BLIS. PICO showed the same lack of scalability as BLIS, for the same reasons. Figure 9 shows summary results for all the city problems solved by PICO.

4.3 Analysis of Application Performance

Both applications performed poorly on the city facility placement problems. In both cases the vast majority of time was spent in ramp-up activities, which explains their miserable scaling.

On the test problems, on the other hand, there was some promise of scalability, particularly for the more difficult problems. This is because the work needed to solve the difficult problems dwarfed the work associated with ramp-up.

While these results suggest that these parallel MILP solvers show promise when applied to difficult MILPs, both solvers appear insufficiently robust to handle large inputs or large problems. Thus we conclude that parallel MILP solvers show promise in principle, but more work is needed in practice before these solvers can operate robustly without the user

Problem Name	Number of Equations	Variables	
		Total	Integer
air05	427	7196	7195
bell5	92	105	58
bienst1	577	506	28
fc.60.20.3	414	708	348
neos2	1104	2102	1040
qiu	1193	841	48
sp98ir	1531	1680	1680
vpm2	235	379	168

Table 1: Problem sizes for standard test problems.

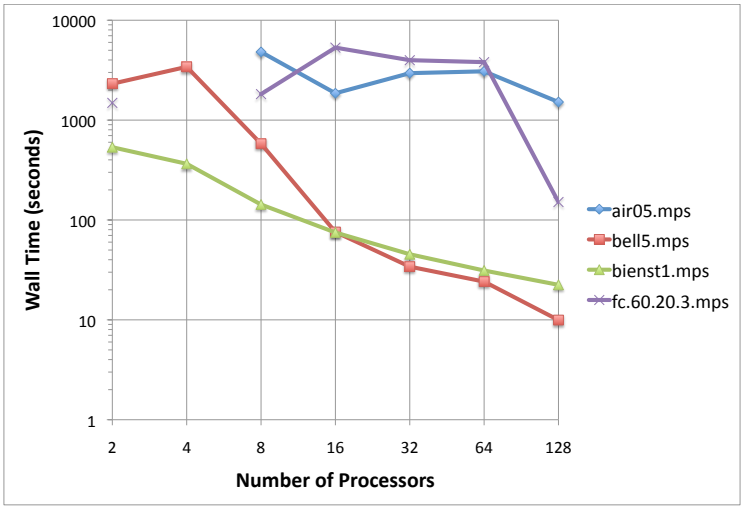

Figure 4: Wall time for solution of select test problems using BLIS, for varying numbers of processors.

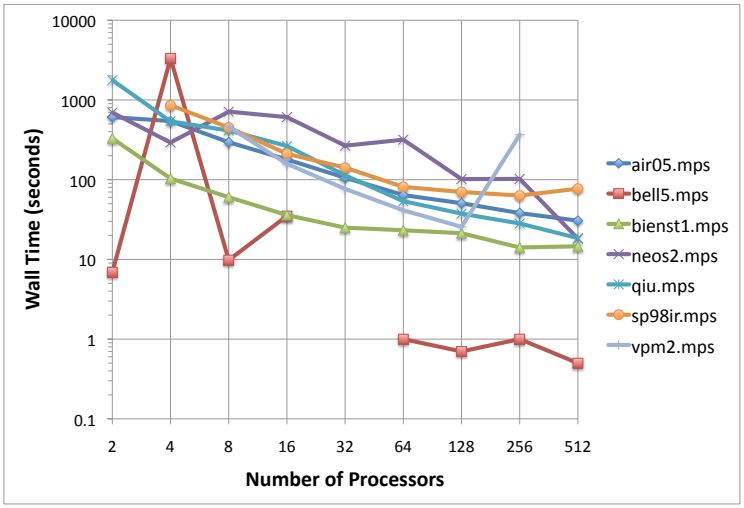

Figure 5: Wall time for solution of select test problems using PICO, for varying numbers of processors.

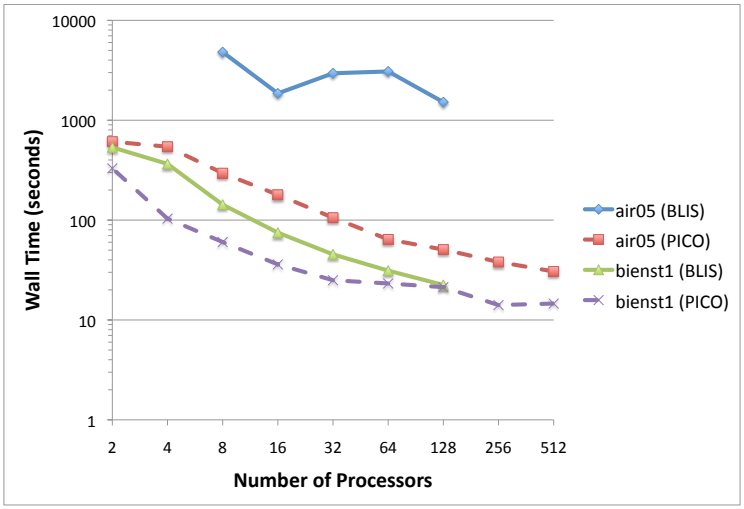

Figure 6: Comparison of wall time for solution of select test problems using BLIS (solid lines) versus PICO (dashed lines), for varying numbers of processors.

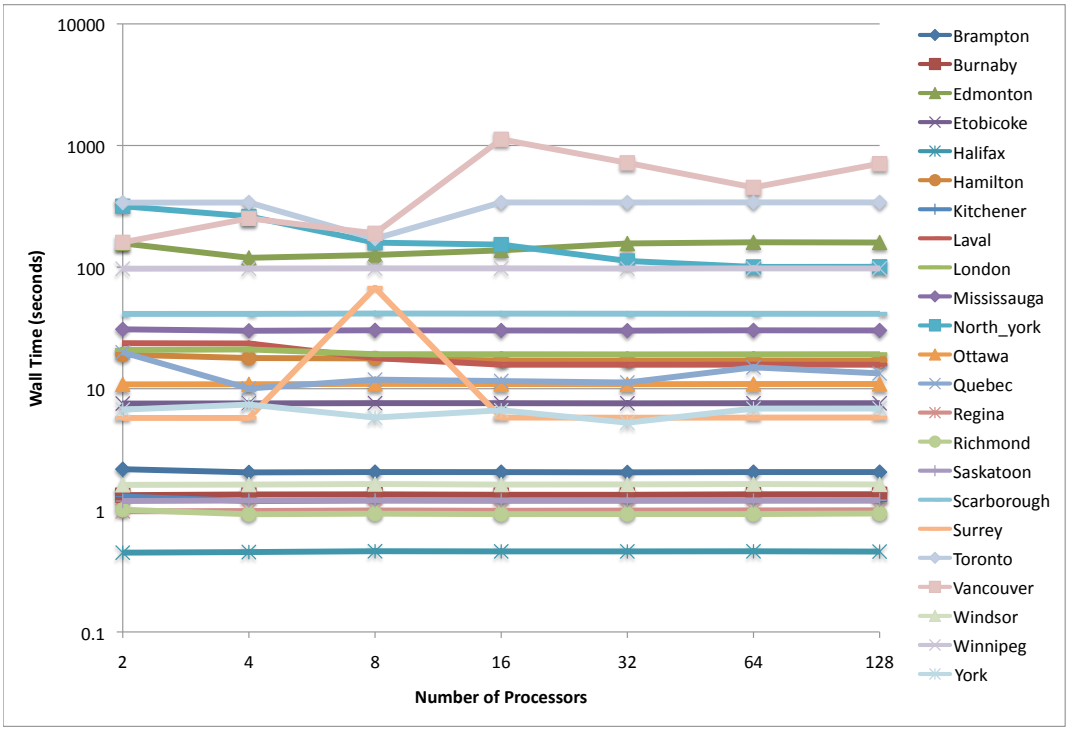

Figure 7: Wall time for solution of all city problems (except Calgary and Montreal, which did not run) using BLIS, for varying numbers of processors.

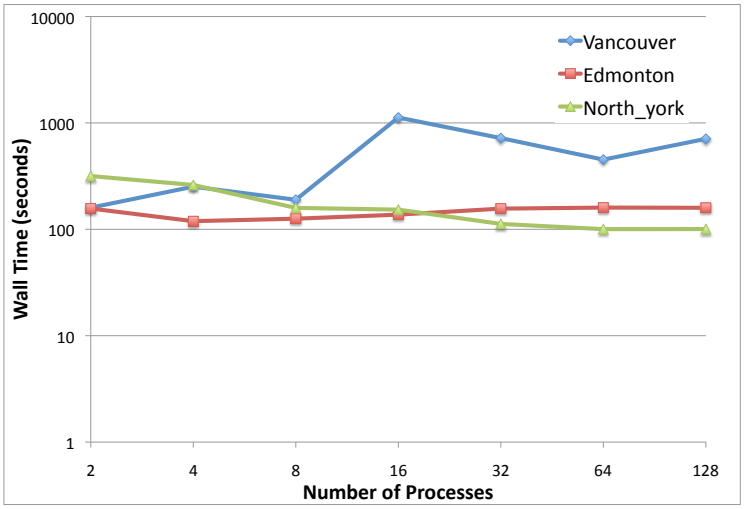

Figure 8: Wall time for solution of three city problems using BLIS, for varying numbers of processors. Observe the slight speedup for North_york, the flat performance on Edmonton, and the irregular performance on Vancouver.

worrying that the latest job submitted to the queue will only waste precious CPU time.

5 Future Work

Clearly, work is needed to before MILP solvers can be capable of using petascale resources. Existing MILP solvers are fragile and unpredictable. Thus code development is needed. We have several ideas of how to improve the performance of MILP solvers.

One idea is to leverage an existing parallel framework to parallelize the MILP solvers. MADNESS (Multiscale Adaptive Numerical Environment for Scientific Simulation) [2] is a promising candidate for the job. MADNESS was originally developed to perform multiscale computations arising from quantum chemistry applications, but the programming environment that underlies the mathematics is sufficiently general to be used for other purposes. We envision using MADNESS to distribute branches across processes, while using existing well-established methods to decompose the domain into branches and to compute the upper and lower bounds of the function.

6 Acknowledgments

This research was sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the U. S. Department of Energy.

This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

7 About the Authors

Rebecca J. Hartman–Baker is a computational scientist at the National Center for Computational Sciences at Oak Ridge National Laboratory. Her research interests include optimization, ill-posed problems, and load balancing at the petascale and beyond. She can be reached at *hartmanbakrj@ornl.gov*.

Ingrid K. Busch is an operations researcher working at the Center for Transportation Analysis at the Oak Ridge National Laboratory. Her research interests include algorithm analysis, design and development for transportation applications. She can

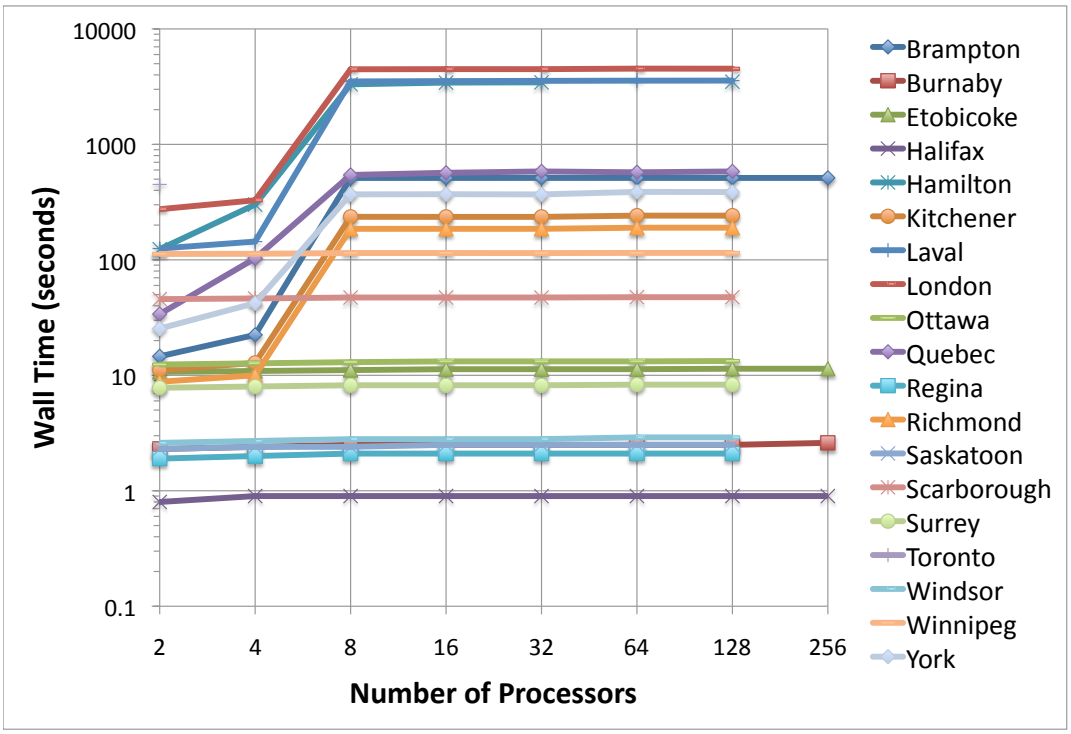

Figure 9: Wall time for solution of all city problems (except Calgary, Edmonton, Mississauga, Montreal, North York, Toronto, and Vancouver, which did not run) using PICO, for varying numbers of processors.

be reached at *buschik@ornl.gov*.

Michael R. Hilliard is an operations researcher working at the Center for Transportation Analysis at Oak Ridge National Laboratory. His research interests include the development of models for complex systems, in particular for transportation and supply chain applications. He can be reached at *hilliardmr@ornl.gov*.

Richard S. Middleton is a geospatial scientist at the Center for Transportation Analysis at Oak Ridge National Laboratory. His primary research focuses on optimizing energy infrastructure, such as pipelines and transmission lines, for applications such as carbon capture and storage and the biofuels supply chain. He can be reached at *middletonrs@ornl.gov*.

Michael S. Schultze is a geospatial scientist at the Center for Transportation Analysis at Oak Ridge National Laboratory. His research interests include large-scale database and software development for transportation-related applications. He can be reached at *schultzems@ornl.gov*.

References

[1] Jonathan Eckstein, Cynthia A. Phillips and William E. Hart, "PICO: An object-oriented framework for parallel branch-and-bound," in *Proc Inherently Parallel Algorithms in Feasibility and Optimization and Their Applications*, Elsevier Scientific Series on Studies in Computational Mathematics, pp. 219—265, 2001.

[2] R. Harrison et al., *MADNESS: Multiscale Adaptive Numerical Environment for Scientific Simulation*, `http://code.google.com/p/m-a-d-n-e-s-s/`, accessed April 24, 2009.

[3] N. Karmarkar, "A new polynomial-time algorithm for linear programming," *Combinatorica, Volume 4*, pp. 373—395, 1984.

[4] V. Klee and G. J. Minty, "How Good Is the Simplex Algorithm?" in *Inequalities III*, pp. 159—175, New York: Academic Press, 1972.

[5] H. Mittelmann, "Mixed Integer Linear Programming Benchmark," `http://plato.asu.edu/ftp/milpc.html`, accessed April 20, 2009.

[6] Steven G. Nash and Ariela Sofer, *Linear and Non-Linear Programming*, New York: McGraw–Hill, 1996.

[7] T. Ralphs, "Parallel Branch and Cut," in *Parallel Combinatorial Optimization*, Wiley, 2006.

[8] Yuji Shinano and Tetsuya Fujie, "ParaLEX: A Parallel Extension for the CPLEX Mixed Integer Optimizer," in *Lecture Notes in Computer Science, Volume 4757*, Berlin: Springer, 2007.