The Tera-Gridiron: A Natural Turf for High-Throughput Computing

Jeff Linderoth, François Margot, and Greg Thain

Abstract— Teragrid resources are often used when high-performance computing is required. We describe our experiences in using Teragrid resources in a high-throughput manner, generating a significant number of CPU cycles over a long time span. In particular, we discuss using Teragrid resources as part of a larger computational grid to perform computations in an ongoing attempt to solve an open problem in mathematical coding theory—the Football Pool Problem.

Index Terms—Football Pool Problem; High-Throughput Computing; Branch-and-Bound; Condor; Master-Worker

1 INTRODUCTION

The Football Pool Problem is one of the most famous problems in coding theory [15]. The problem derives its name from a lottery-type game where participants predict the outcome of soccer matches. A ticket is winning if the outcome of no more than d matches out of v are predicted incorrectly, where each match has three possible outcomes—win, lose, or draw. The goal of the Football Pool Problem is to determine the minimum-size set of lottery tickets to buy to guarantee at least one winning ticket, no matter the outcome of the matches. Mathematically, the problem is to determine the smallest covering code of radius d of ternary words of length v. For $d = 1$ and $v = 6$, the optimal code size is only known to be between the values of 65 and 73 [24, 26]. In this work, we report experiences on using the Teragrid as part of a large-scale computation aimed at sharpening these bounds.

A mathematical optimization problem, known as an integer program, can be formed that will determine the smallest covering code (set of tickets to purchase). Specifically, let W be the set of possible outcomes of the matches. For example, for $v = 6$ matches, the size of this set is $|W| = n = 3^6 = 729$. Define the matrix $A \in \{0,1\}^{n\times n}$ with $a_{ij} = 1$ if and only if the ticket $j \in W$ is a winning ticket for the match outcomes $i \in W$. Binary decision variables $x \in \{0,1\}^n$ are used to signify (with $x_j = 1$) that ticket j is to be purchased. The size of the smallest covering code z_v is given by the optimal solution to the integer program:

$$z_v = \min_{x\in\{0,1\}^n} \{1^T x \mid Ax \geq 1\} \qquad \text{(CCIP)}$$

Integer programs such as CCIP are NP-Hard. In practice, they are often effectively solved via a technique known as branch-and-bound, an algorithm that dates back to the work of Land and Doig [16]. By relaxing the binary constraints $x \in \{0,1\}^n$ to their linear counterparts $x \in [0,1]^n$, we obtain a problem known as the *linear programming relaxation* of CCIP. The linear programming relaxation is solvable in polynomial time, and its solution value gives a lower bound z_v. Branch-and-bound is a tree-search enumeration algorithm. At every node of the tree, the linear programming relaxation of CCIP is solved and its value is used for pruning the search. If all components of the solution to the relaxation are integer-valued, then the solution must be optimal. Otherwise, the problem is further subdivided by choosing some variable x_j whose solution value is fractional, and creating two new problems that must be solved: one with the additional constraint $x_j \leq 0$ and another with the constraint $x_j \geq 1$. See Nemhauser and Wolsey [23] for background material on Integer Programming.

In the case of the Football Pool Problem with $v = 6$ and $d = 1$, CCIP is an integer program consisting of 729 variables and 729 constraints. Integer programs of this size are *routinely* solved by state-of-the-art commercial solvers such as CPLEX [7] an XPRESS-MP [8]. However, these complex codes are unable to solve the Football Pool instance. This is demonstrated in Figure 1, which shows the improvement in lower bound value using CPLEX v9.1 as a function of the number of nodes evaluated. The figure shows that after 500,000 nodes evaluated the lower bound is improved only to 58. Improving the lower bound to 65 would appear (by simple extrapolation) to be computationally impossible in this manner.

- Jeff Linderoth, Dept. of Industrial and Systems Engineering, Lehigh University, jtl3@lehigh.edu
- François Margot, Tepper School of Business, Carnegie Mellon University, fmargot@andrew.cmu.edu
- Greg Thain, Computer Sciences Department, University of Wisconsin-Madison, gthain@cs.wisc.edu

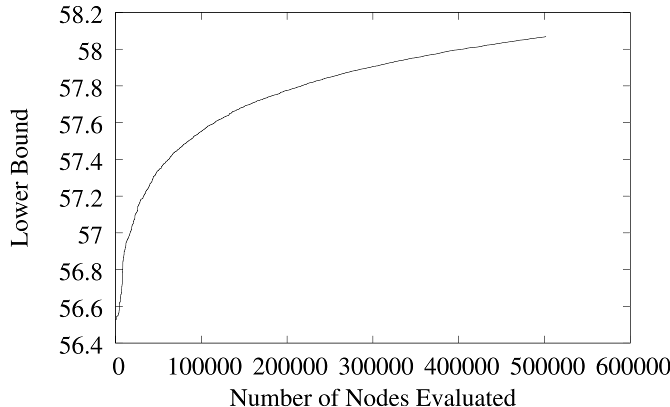

Fig. 1: CPLEX Lower Bound Improvement

2 IMPROVING BRANCH-AND-BOUND

A major factor that confounds the branch-and-bound process is that CCIP is very *symmetric*. Loosely speaking, a symmetric integer program is one in which a solution to its linear programming relaxation can be "preserved" through a permutation of its column indices. In a series of papers Margot [20, 21] demonstrated how to apply a technique known as *isomorphism pruning*, dating back to the work of Bazaraa and Kirca [4], to mitigate the undesirable effects of symmetry. Isomorphism pruning can recognizing and eliminate nodes of the branch-and-bound tree that are isomorphic with respect to the symmetry in the integer program and significantly helps a branch-and-bound algorithm make progress towards the solution of the football pool problem integer program (CCIP). However, isomorphism pruning alone is not sufficient to improve the best-known bounds on the optimal solution value z_6. Thus, in a companion work, Linderoth et al. [17] discuss how additional techniques such as the subcode inequalities of Östergård and Blass [25] have been combined with enumeration and integer programming to attack the problem. The result of this work is a set of integer programs for consecutively larger values of the best-known lower bound on z_6. In Table 1, we show the number of integer programs that must be solved in order to improve the lower bound on z_6 to M.

M	# IP
67	7
68	13
69	45
70	102
71	176
72	264
73	393
	1000

Table 1: #IP Required to Establish New Lower Bounds on z_6

The 1000 integer programs that must be solved to establish the optimality of $z_6 = 73$ are each quite computationally challenging, and in subsequent sections, we describe how we have created a powerful, computational grid for the high-throughput computations necessary to solve them.

3 THE COMPUTATIONAL GRID

Branch and bound is a natural paradigm to map to a parallel computing platform, as nodes of the enumeration tree can be evaluated independently. In fact, branch-and-bound and similar tree-search techniques have been implemented on a variety of parallel computing platforms dating back to the advent of multiprocessor machines. Gendron and Crainic [11] give a survey of parallel branch-and-bound algorithms, including references to early works.

Of particular interest to us in this work are parallel computing platforms created by harnessing CPU cycles from a *wide* variety of resources. Further, we are interested in using the CPU cycles in a *flexible* manner, using resources that would otherwise be idle. As such, we have built our computational grid using the Condor software system for high-throughput computing [19]. Ideally, we would like to aggregate many Condor clusters together to make one giant pool of resources. However, this is not possible, for both technical and administrative reasons. Thankfully, Condor is equipped with a collection of mechanisms through which CPU resources at disparate locations can be federated together, with varying degrees of transparency and overhead. We make use of many of these mechanisms to build our computational grid. In particular, we obtain resources via

- Condor Flocking [9],
- A manual version of Condor glide-in [10] sometimes known as *hobble-in*,
- A combination of hobble-in with port-forwarding, which we call *sshidle-in*,
- Direct submission of the worker executables to remote Condor pools.
- A recently-introduced mechanism to Condor called *schedd-on-the-side* [5],

TeraGrid resources are accessed for our computation via flocking, hobble-in, and remote submission, and we now briefly explain specifically how each method works in this context.

3.1 Grid-Building Mechanisms

Flocking. Condor flocking works by allowing one or more pools of execution machines to be scheduled by a single local Condor job scheduler. From the users' perspective, flocking is the most transparent way to aggregate resources. The user does not do any additional configuration, and submits

jobs as usual to the local scheduler. Jobs that the local scheduler can not run are sent as low priority jobs to unused machines in the remote pools. Flocking is best used when there is a close administrative relationship between the owners of the two Condor pools, for it must be explicitly enabled on both sides. Condor flocking requires inbound network connections on many ephemeral TCP/IP ports from the flocked-from scheduler to the flocked-to machines, so it is difficult to use where network firewalls exist. Flocked jobs are scheduled by the local scheduler, so they contribute to the load of that machine, and may limit scalability.

Glide-in. Condor glide-in is a way to construct an overlay Condor pool on top of another batch system. This overlay pool can then report to an existing pool. Typically Condor glide-in is used to access resources via a Globus gateway. Condor glide-in works in two steps: set up and execution. During set up, Condor binaries and configuration files are automatically copied to the remote resources. The execution step starts the Condor daemons running through the resource's Globus interface. Once the glide-in process is complete, the processors simply show up in the local Condor pool. Condor Glide-in is most useful when we have access to a non-Condor batch system, and want to use those resources as part of a larger Condor-aggregated computation. However, to use Condor glide-in, the user must have an X.509 certificate, access to the Globus resource, and the Globus software must be installed and properly configured.

To circumvent the dependency on Globus gateways configured for our computation, we employed a "low-weight" version of Condor Glide-in called Condor **hobble-in**. Condor hobble-in works like a manual version of Condor glide-in. First, the Condor binaries are copied to the remote resources and configured to report to an existing Condor pool. Next, batch submission requests are made to the local job schedulers. When the jobs run, the processors allocated as part of the batch request appears as workers in the local Condor pool.

When hobbling in to TeraGrid resources as part of an ongoing computation, the most effective strategy for obtaining significant resources is to make many requests for small numbers of processors and for short duration. This way, the batch requests are run more quickly, as they can be fulfilled from the backfill of local schedulers. However, the computation must be able to deal with the processors being reclaimed at the end of the batch allocation.

Remote Submit. Remote submit is the least transparent method of obtaining grid resources. In this case, we simply log into the remote system, and submit executables to the local Condor pool. Information so that the new processes can join the existing computation must be given as arguments to the executable at the time of submission.

Condor remote submission is most useful when there is a firewall in place, and the main Condor scheduler is blocked from communication with the remote pool. When performing a remote submission, we can even use ssh's port forwarding capability to forward socket connections from the remote execute machines to a master machine via a gateway. This technique, which we call *sshidle-in*, allows us to run executables on machines that are on private networks.

Schedd-on-the-side. The Schedd-on-side is a new Condor technology which takes idle jobs out of the local Condor queue, translates them into Grid jobs, and uses Condor-G to submit them to a remote Grid queue. The original submitter doesn't know that the jobs originally destined for the local queue have now been re-tasked to a Grid, and the schedd-on-the-side can do matching and scheduling of jobs to one of many remote Grid sites. This is an easy to to take advantage of large systems like the Open Science Grid.

Putting it all together. Table 2 shows the number of available number of machines at each grid site that we used in our computations. The table also lists the method used to access each class of machines and the architecture and operating system for each batch of processors. The sites that begin with OSG are processors on the Open Science Grid, and the sites that begin with TG are TeraGrid installations.

3.2 MW

The grid-building mechanisms outlined in Section 3.1 provide the underlying CPU cycles necessary for running large-scale branch-and-bound computations on grids, but we still require a mechanism for *controlling* the branch-and-bound algorithm in this dynamic and error-prone computing environment. For this, we use the MW grid-computing toolkit [14]. MW is a software tool that enables implementation of master-worker applications on computational grids. The master-worker paradigm consists of three abstractions: a master, a task, and a worker. The MW API consists of three abstract bases classes—`MWDriver`, `MWTask`, and `MWWorker`—that the user must reimplement to create an MW application.

The `MWDriver` is the master process, and as such the user must implement methods `get_userinfo()` to initialize the computation, `setup_initial_tasks()` to create initial work units, and `act_on_completed_task()` to perform necessary algorithmic action (possibly the addition of new tasks via the `addTasks()` method) once a task completes. The `MWWorker` class controls the worker processes, so the primary method to be implemented is `execute_task()`. In addition, there are required methods for marshalling and unmarshalling the data that defines the computational tasks.

MW offers advanced functionality that is often useful or required for running large, coordinated computations in a high-throughput fashion. Specifically, MW is equipped with

Site	Access Method	Arch/OS	Machines
Wisconsin - CS	Flocking	x86_32/Linux	975
Wisconsin - CS	Flocking	Windows	126
Wisconsin - CAE	Remote submit	x86_32/Linux	89
Wisconsin - CAE	Remote submit	Windows	936
Lehigh - COR@L Lab	Flocking	x86_32/Linux	57
Lehigh - Campus desktops	Remote Submit	Windows	803
Lehigh - Beowulf	ssh + Remote Submit	x86_32	184
Lehigh - Beowulf	ssh + Remote Submit	x86_64	120
OSG - Wisconsin	Schedd-on-side	x86_32/Linux	1000
OSG - Nebraska	Schedd-on-side	x86_32/Linux	200
OSG - Caltech	Schedd-on-side	x86_32/Linux	500
OSG - Arkansas	Schedd-on-side	x86_32/Linux	8
OSG - BNL	Schedd-on-side	x86_32/Linux	250
OSG - MIT	Schedd-on-side	x86_32/Linux	200
OSG - Purdue	Schedd-on-side	x86_32/Linux	500
OSG - Florida	Schedd-on-side	x86_32/Linux	100
TG - NCSA	Flocking	x86_32/Linux	494
TG - NCSA	Flocking	x86_64/Linux	406
TG - NCSA	Hobble-in	ia64-linux	1732
TG - ANL/UC	Hobble-in	ia-32/Linux	192
TG - ANL/UC	Hobble-in	ia-64/Linux	128
TG - TACC	Hobble-in	x86_64/Linux	5100
TG - SDSC	Hobble-in	ia-64/Linux	524
TG - Purdue	Remote Submit	x86_32/Linux	1099
TG - Purdue	Remote Submit	x86_64/Linux	1529
TG - Purdue	Remote Submit	Windows	1460
			19,012

Table 2: Characteristics of Computational Grid

features for user-defined checkpointing, normalized application and network performance measurements, eager task scheduling [3], and methods for the dynamic prioritization of computational tasks. This functionality is explained in greater detail in the papers [14, 12] and the MW User's Manual [18]. In particular for this (very long-running) computation, checkpointing the state of the master-process is necessary, as is the ability to dynamically prioritize the computational tasks, as discussed briefly in Section 3.3.

MW has been used to instrument branch-and-bound algorithms for the quadratic assignment problem [1], mixed integer nonlinear programs [13], and for mixed integer linear programs by Chen et al. [6]. The solver in [6], called FATCOP, was augmented with the isomorphism pruning techniques discussed in Section 2 and used in our attempt to improve the lower bounds on z_6 by solving the instances in Table 1 for consecutively larger values of M.

3.3 Scaling Master-Worker Branch-and-Bound Computations

Branch and bound is a very natural paradigm to map to run in a master-worker framework. Simply, the master processor can manage the tree of unexplored nodes that must be evaluated and pass to the workers nodes to evaluate. When running on large configurations of resources (with many workers), care must be taken to ensure that the master processor is not overwhelmed with requests from the workers. In this section, we briefly state how by tuning the algorithm and preparing the infrastructure appropriately, barriers to an efficient large-scale implementation were overcome.

Grain Size. An effective way to reduce the contention at the master in a master-worker computation is to reduce the rate at which workers report to ask for new work. Thankfully, in the branch-and-bound algorithm, there is an obvious mechanism for increasing the grain size of the worker computations. Instead of having a worker's task be the evaluating of one node, (the solution of one LP relaxation to (CCIP) at that node), the worker's task can be to evaluate the entire subtree rooted at that node. In this case, workers will perform the branching and pruning operations as well. This is precisely the strategy that we employ for our parallel algorithm. For load balancing purposes, it is necessary to stop the computation on the worker after a maximum grain size CPU time T and report unevaluated nodes from the task's subtree back to the master process. Typically, the value of $T = 20$min or $T = 30$min was chosen for our runs. Larger values of T are possible, but may result in a significant increase in the number of tasks that must be rescheduled by MW due to the worker's being recalled for another process or purpose. The value of T can be changed dynamically. In fact, whenever the number of tasks remaining to be completed at the master is less than the number of workers participating in the computation, T is changed to a much smaller value, typically $T = 10$sec. This has the effect of rapidly increasing the work pool size on the master. The implementation of the dynamic task time is accomplished by using the method `pack_driver_task_data` of the `MWDriver` class so that the (current) maximum CPU time T is sent to the worker as part of each task.

Task List Management. In MW the master class manages a list of uncompleted tasks and a list of workers. These tasks represent nodes in the branch-and-bound tree whose subtree must be completely evaluated. The default scheduling mechanism in MW is to simply assign the task at the head of the task list to the first idle worker in the worker list. However, MW gives flexibility to the users in the manner in which each of the lists are ordered. For our implementation it was advantageous to make use of the `set_task_key_function()` method of the `MWDriver` to dynamically alter the ordering of tasks during the computation. The main purpose of the reordering was to ensure that the number of remaining tasks on the master processor did not grow too large and exhaust the master's memory. Nodes deep in the branch-and-bound tree typically require less processing than do nodes high in the tree. Therefore, if the master task list was getting "too large" ($\geq \beta$), the list was ordered such that deep nodes were given as tasks. Once the size of the master task list dropped below a specified level ($\leq \alpha$), the list was again reordered so that nodes near to the root of the tree were sent out for processing. Typically, values of $\alpha = 15000$ and $\beta = 17000$ were used in our computation.

Fault Tolerance. The computation runs for weeks across thousands of machines, so failures which would be rare on a single-processor become common. Further, as discussed in Section 3.1, the primary strategy for obtaining TeraGrid resources was to make requests to the local schedulers for small amounts of CPU time. In this case, "failures" of the worker processes correspond to the processors being reclaimed by the scheduler, so in fact worker failures are extremely common. Our primary strategy for robustness is to detect failures on a worker machine and to re-run the failed task elsewhere. MW has features that automatically performing the failure detection and re-scheduling. The less common, but more catastrophic, case is when the master machine fails. To deal with this, the state of the master process is periodically checkpointed. MW performs the checkpointing automatically, as long as the user has re-implemented the `write_ckpt_info()` method of the `MWTask` class and the `write_master_state()` method of the `MWDriver` class. Should the master crash, the computation can be restarted from the state in the checkpoint file.

Infrastructure Scaling. On many of the grid sites in Table 2, our workers are run with low priority, and the scheduling policy at the sites is to simply suspend the low priority job, rather than to preempt the low priority job. This job suspension became a significant problem for our computation, as some jobs were suspended for days, blocking the entire computation waiting for the results of the suspended tasks. To work around excessively long job suspension, we used a method `reassign_tasks_timedout_workers()` of MW that will automatically reassign tasks that have not completed in a pre-specified time limit. In our case, a time limit of one hour was sufficient, as we were already limiting the grain size of the worker computation to less than $T = 30$min.

However, while running our large-scale computation, we noticed that suspension of the workers caused a more subtle and serious problem. The MW master is written as a single-threaded, event-driven program. Sometimes, the worker suspension would occur as the worker was writing results over TCP to the master. When running with a large number of workers, the suspension of a worker in the middle of an active TCP write occurred roughly twice a week. In this case, the master would block, waiting for the remainder of the results from the suspended worker. The effects could then cascade, as writes to open socket connections from other workers were initiated during the time when the master was blocked, but subsequently, the worker that initiated the socket write was itself suspended. In this case, the problem was solved by adding timeouts to each network read in MW.

Further, our recent computations on the grid resources listed in Table 2 have scaled to more than 4500 simultaneous worker processors, which necessitated re-coding MWś socket management layer to use the `epoll()` system call rather than `poll()`. Improving the scalability of MW's network performance is another instance in which running specific large-scale computations can bring benefit to a broad scientific computing community. In many cases, only by running large computations can bottlenecks that limit the scalability of existing software packages such as Condor or MW be pinpointed. Subsequent releases can address the issues in scalability, aiding the community-at-large.

4 COMPUTATIONAL RESULTS

The solution of the integer programs in Table 1 to solve the football pool problem have been ongoing since the beginning of 2006. The computation has not been continually running. It is often stopped in-between the solution of integer programs or for maintenance of the master machine. To date, we have been able to establish a new lower bound of $M = 71$ for the football pool problem, an improvement of 6 over the best bound known before this work. In Table 3, we show aggregated computational results for a portion of our computation. Specifically, we show the work required to solve the IPs to establish that $z_6 \geq M = 70$ and $z_6 \geq M = 71$. For these two portions of the computation, over *140 CPU years* were used and delivered by grid resources in roughly 92 days. The total number of nodes in the branch and bound trees for the solution of the IPs numbers in the billions, and *trillions* of LP pivots are required to evaluate these nodes. To our knowledge, this is the largest branch-and-bound computation ever run on a wide-area grid. For example, Anstreicher et al. [1] required 11 CPU years to solve the nug30 quadratic assignment problem, Mezmaz et al. [22] used 22 CPU years to solve a flow-shop problem by branch and bound, and Applegate et al. [2] used 84 CPU years (on a tightly-coupled cluster) for finding the shortest tour through 24,978 towns in Sweden. The football-pool problem computation has in fact taken more than 140 CPU years, as it is still on-going. In fact, as mention in Section 3.3, we have used simultaneously over 4,500 workers while solving IPs related to establishing $z_6 \geq 72$.

Table 3: Computation Statistics

	$M = 70$	$M = 71$
Avg. Workers	555.8	562.4
Max Workers	2038	1775
Worker Time (years)	110.1	30.3
Wall Time (days)	72.3	19.7
Worker Util.	90%	71%
Nodes	2.85×10^9	1.89×10^8
LP Pivots	2.65×10^{12}	1.82×10^{11}

Even though the computational grid that we have used has evolved considerably during the course of our quest to solve the football pool problem, it is still interesting to examine the distribution of resources used in the computation. Figure2 shows a recent snapshot of the number of workers participating from various sites in Table 2 over time. As seen in the figure, the Teragrid resource have played a significant role in this large-scale computation. (This can be seen by aggregating the clusters "TG", "NCSA", and "Purdue" in Figure 2). By our best estimation, we are roughly 50% through the computation for $M = 72$ and hope to be able to announce that $z_6 \geq 72$ soon.

ACKNOWLEDGMENTS

The preceding research is brought to you by the National Science Foundation (OCI-0330607, CMMI-0522796) and through TeraGrid resources provided by NCSA, SDSC, ANL, TACC, and Purdue University. François Margot is also supported in part by the Office of Naval Research under grant N00014-03-1-0188. Any rebroadcast, retransmission of the events, descriptions, or contents of this paper without appro-

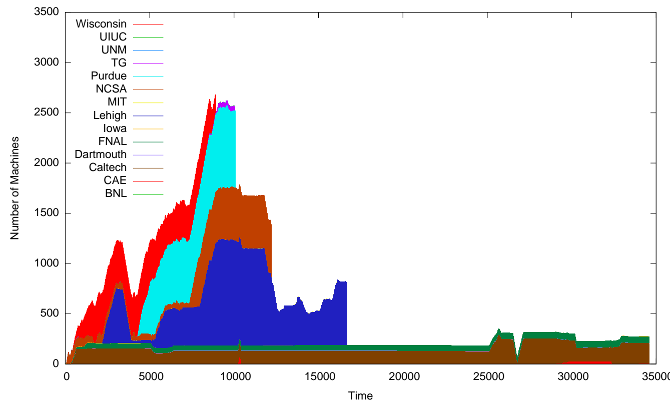

Fig. 2: Number of workers per cluster

priate citation is expressly prohibited. The authors would like to thank the entire Condor team for their tireless efforts to provide a Really Useful computing infrastructure.

References

[1] K. Anstreicher, N. Brixius, J.-P. Goux, and J. T. Linderoth. Solving large quadratic assignment problems on computational grids. *Mathematical Programming, Series B*, 91:563–588, 2002.

[2] D. Applegate, R. Bixby, W. Cook, and V. Chvátal. Personal Communication.

[3] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. *Future Generation Computer Systems*, 15:559–570, 1999.

[4] M. S. Bazaraa and O. Kirca. A branch-and-bound heuristic for solving the quadratic assignment problem. *Naval Research Logistics Quarterly*, 30:287–304, 1983.

[5] D. Bradley. Schedd on the side. In *Presentation at Condor Week 2006*, Madison, WI, 2006.

[6] Q. Chen, M. C. Ferris, and J. T. Linderoth. FATCOP 2.0: Advanced features in an opportunistic mixed integer programming solver. *Annals of Operations Research*, 103:17–32, 2001.

[7] CPLEX Optimization. *Using the CPLEX Callable Library, Version 9*. CPLEX Optimization, Inc., Incline Village, NV, 2005.

[8] *XPRESS-MP Reference Manual*. Dash Associates, 2004. Release 2004.

[9] D. H. J. Epema, M. Livny, R. v. Dantzig, X. Evers, and J. Pruyne. A worldwide flock of condors: Load sharing among workstation clusters. *Journal on Future Generation Computer Systems*, 12, 1996.

[10] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Copmuting*, 5:237–246, 2002.

[11] B. Gendron and T. G. Crainic. Parallel branch and bound algorithms: Survey and synthesis. *Operations Research*, 42:1042–1066, 1994.

[12] W. Glankwamdee and J. Linderoth. MW: A software framework for combinatorial optimization on computational grids. In E. Talbi, editor, *Parallel Combinatorial Optimization*. John Wiley & Sons, 2006. To appear.

[13] J.-P. Goux and S. Leyffer. Solving large MINLPs on computational grids. *Optimization and Engineering*, 3: 327–354, 2003.

[14] J.-P. Goux, S. Kulkarni, J. T. Linderoth, and M. Yoder. Master-Worker: An enabling framework for master-worker applications on the computational grid. *Cluster Computing*, 4:63–70, 2001.

[15] H. Hämäläinen, I. Honkala, S. Litsyn, and P. Östergård. Football pools–A game for mathematicians. *American Mathematical Monthly*, 102:579–588, 1995.

[16] A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems. *Econometrica*, 28:497–520, 1960.

[17] J. Linderoth, F. Margot, and G. Thain. The football pool problem and the computational grid. Unpublished working paper, 2007.

[18] J. Linderoth, G. Thain, and S. J. Wright. *User's Guide to MW*. University of Wisconsin Madison, 2007. `http://www.cs.wisc.edu/condor/mw`.

[19] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor—A hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, 1998.

[20] F. Margot. Pruning by isomorphism in branch-and-cut. *Mathematical Programming*, 94:71–90, 2002.

[21] F. Margot. Exploiting orbits in symmetric ILP. *Mathematical Programming, Series B*, 98:3–21, 2003.

[22] M. Mezmaz, N. Melab, and E.-G. Talbi. A grid-enabled branch and bound algorithm for solving challenging combinatorial optimization problems. Research Report 5945, INRIA, 2006.

[23] G. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley and Sons, New York, 1988.

[24] P. Ostergård and A. W. A. A new lower bound for the football pool problem for six matches. *Journal of Combinatorial Theory, Ser. A*, 99:175–179, 2002.

[25] P. Östergård and W. Blass. On the size of optimal binary codes of length 9 and covering radius 1. *IEEE Transactions on Information Theory*, 47:2556–2557, 2001.

[26] L. T. Wille. The football pool problem on six matches. *Journal of Combinatorial Theory, Ser. A*, 45:171–177, 1987.