Improving Bounds on the Football Pool Problem via Symmetry Reduction and High-Throughput Computing

Jeff Linderoth
Department of Industrial and Systems Engineering, The University of Wisconsin-Madison, 3226 Mechanical Engineering Building, 1513 University Ave., Madison, WI 53706, USA, linderoth@wisc.edu
François Margot
Tepper School of Business, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213-3890, USA, fmargot@andrew.cmu.edu
Greg Thain
Department of Computer Science, The University of Wisconsin-Madison, 1210 W. Dayton St., Madison, WI 53706, USA, gthain@cs.wisc.edu

The Football Pool Problem, which gets its name from a lottery-type game where participants predict the outcome of soccer matches, is to determine the smallest covering code of radius one of ternary words of length v. For $v = 6$, the optimal solution is not known. Using a combination of isomorphism-pruning, subcode enumeration, and linear-programming based-bounding, running on a high-throughput computational grid consisting of thousands of processors, we are able to report improved bounds on the size of the optimal code for this open problem in coding theory.

Key words: Football Pool Problem; High-Throughput Computing; Branch-and-Bound; Condor; Master-Worker
History:

---

1. Introduction

The Football Pool Problem is one of the most famous problems in coding theory (Hämäläinen et al., 1995) and concerns finding small cardinality covering codes. Before formally defining a covering code, a few definitions are necessary. Let $v \geq 1$ and $\alpha \geq 1$ be two integers. Let $W(v, \alpha)$ be the set of all words of length v using letters from the alphabet $\{0, 1, \ldots \alpha - 1\}$. To simplify notation, we use W instead of $W(v, \alpha)$ when the values of v and α are clear from the context or are irrelevant.

For any two words $a \in W, b \in W$, the *Hamming distance* between these two words is the number of components in which they are different:

$$\text{dist}(a, b) = |\{i \mid a_i \neq b_i\}|.$$

A *code* is a subset of words $C \subseteq W$. A *covering code* of radius d for the set of words W is a code $C \subseteq W$ such that every word $w \in W$ is at most a distance d away from at least one word in C, i.e. a code such that $\forall w_i \in W, \exists\ w_j \in C$ with $\text{dist}(w_i, w_j) \leq d$. The *Football Pool Problem* is to find a minimum cardinality covering code of radius $d = 1$ for the set of all ternary words of length $v, W(v, 3)$. The problem gets its name from a lottery-type game where participants predict the outcome of v soccer matches, and a prize is won if the player predicts no more than d matches incorrectly. The goal of the Football Pool problem is to determine the minimum number of tickets a player must purchase in order to ensure themselves of winning a prize no matter the outcome of the matches. For $v = 6$, the size of the optimal covering code is not known, and in fact, rather only weak bounds are known for the value that the optimal solution might take. Table 1, shows known optimal values for $1 \leq v \leq 5$. For $v = 6$, the best known feasible solution has value 73 (found by Wille (1987) using a Tabu Search algorithm) and the best published lower bound is 65 (Östergård and Wassermann (2002)).

v	1	2	3	4	5
$\lvert C^* \rvert$	1	3	5	9	27

Table 1: Optimal values for the Football Pool Problem with v matches.

An integer program is easily formulated that will determine the optimal covering code for any word set W and radius d. Specifically, for $n = |W|$, use binary decision variables $x \in \{0, 1\}^n$ with $x_j = 1$ if and only if word j is in the code C^* and define the matrix $A \in \{0, 1\}^{n \times n}$ with $a_{ij} = 1$ if and only if word $i \in W$ is at distance $\leq d$ from word $j \in W$. A smallest covering code C^* then corresponds to an optimal solution to the integer program

$$|C^*| = \min_{x \in \{0,1\}^n} \{e^T x \mid Ax \geq e\}, \tag{1.1}$$

where e is an $n-$dimensional vector of ones.

In the case of the Football Pool Problem with $W = W(6, 3)$, (1.1) 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 and XPRESS-MP . However, these software are unable to solve the Football Pool Problem for $v = 6$. This is illustrated in Figure 1, which shows the improvement in lower and upper bound values on $|C_6^*|$, the size of an optimal covering code for $W(6, 3)$, using CPLEX v9.1 as a function of the number of nodes

evaluated. After 500,000 nodes, the lower bound is improved from 56.08 (the value of the initial linear programming relaxation of (1.1)) to only to 58. Improving the lower bound even past the currently best known lower bound to 65 would appear (by simple extrapolation) to be computationally impossible in this manner. A major factor that confounds the branch-and-bound process is that (1.1) is very *symmetric*. Techniques for reducing the negative impact of the symmetry are discussed in Section 2.

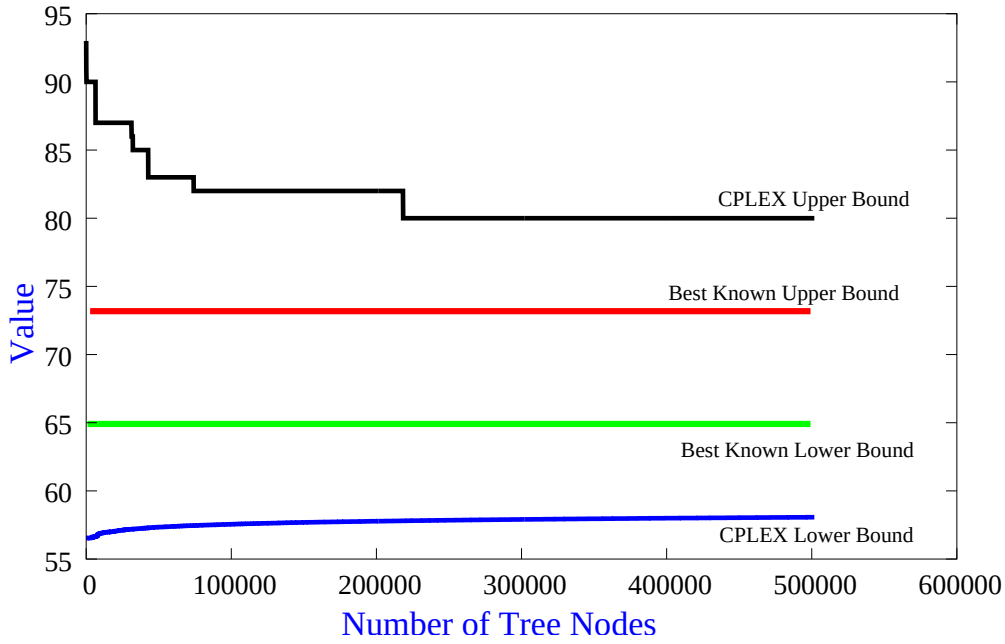


Figure 1: CPLEX Lower Bound Improvement

In this work, we focus on improving the lower bound for $|C_6^*|$. Through a variety of techniques that combine efficient isomorphism-free enumeration with integer programming, the problem of improving the lower bound on $|C_6^*|$ is transformed into a series of (simpler) integer programs. These integer programs are solved using a distributed branch-and-bound algorithm equipped with isomorphism pruning. The platform we use to solve the instances is a large-scale high-throughput computing system built mostly from CPU cycles that would have otherwise gone unused. To date, our computations have been improve the lower bound on $|C_6^*|$ from 65 to 71, and a total of more than *two CPU centuries* total have gone into the computation, making it one of the largest computations of its kind ever attempted.

The remainder of the paper is divided into five sections. In Section 2, we review some of the symmetry-reduction techniques that we employ in this work: isomorphism pruning and covering systems. In Section 3, we introduce additional techniques for processing the

results obtained from enumerating solutions to the covering systems that can further reduce the work required to improve lower bounds on $|C_6^*|$. Section 4 describes our computing platform, tools used to build that platform, and how we built a branch-and-bound algorithm to effectively run on that platform. Section 5 contains the results of our computation, and we offer conclusions of our work in Section 6.

2. Symmetry Handling Techniques

In this section, we define what is meant by a symmetric integer program, and discuss a technique called *isomorphism pruning* that can mitigate the undesirable effects of symmetry in integer linear programming. We then recount an enumerative technique based on *covering systems* that has been used to improve lower bounds on the size of covering codes in the past, and we describe how this technique can be combined with integer programming.

2.1. Isomorphism Pruning

Let Π^n be the set of all permutations of $I^n = \{1, \ldots, n\}$, and let $x \in \{0,1\}^n$. The operation of applying a permutation $\pi \in \Pi^n$ to a solution x is to permute the coordinates of x according to π. That is,

$$\pi(x) = \pi(x_1, x_2, \ldots x_n) \stackrel{\text{def}}{=} (x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)}).$$

We call a permutation π a *symmetry* of the integer program (1.1) if the permutation preserves feasibility. That is, x is feasible if and only if $\pi(x)$ is feasible. We will denote by $\Gamma \subset \Pi^n$ the set of all symmetries of (1.1). To understand why symmetries may confound the branch-and-bound algorithm, consider the following situation. Suppose that $\hat{x}$ is a (non-integral) solution to the linear programming relaxation of (1.1), with $0 < \hat{x}_j < 1$, and the decision is made to branch down on variable x_j by fixing $x_j = 0$. If $\exists \pi \in \Gamma$ such that $[\pi(\hat{x})]_j = 0$, then $\pi(\hat{x})$ is a feasible solution for this child node, and $e^T\hat{x} = e^T(\pi(\hat{x}))$, so the relaxation value for the child node will not change. If the order of Γ is large, then there are many permutations that can be applied to the present solution of the relaxation yielding a solution feasible for the child node. This results in many branches that effectively do not change the solution of the parent node. Symmetry has long been recognized as a curse for solving integer programs, and auxiliary (usually extended) formulations are often sought to reduce the amount of symmetry in an ILP formulation (Barnhart et al., 1998; Holm and Sørensen, 1993; Méndez-Díaz and Zabala, 2006). In addition, there is a body of research on valid inequalities that

can help exclude symmetric feasible solutions for specific permutation groups (Sherali and Smith, 2001; Macambira et al., 2004; Kaibel and Pfetsch, 2007).

For some permutation $\pi \in \Gamma$ and set of indices $S \subset \{1, 2, \ldots, n\}$, let $\pi(S) = \{\pi(i) \mid i \in S\}$. The set $\{\pi(S) \mid \pi \in \Gamma\}$ is an equivalence class of all equivalent "relabelings" of $S \subseteq \{1, 2, \ldots, n\}$ known as the *orbit* of S under Γ. A node a of the branch-and-bound enumeration tree can be characterized by the set of variables fixed to zero (resp. one) by branching decisions, i.e.

$$\begin{aligned} F_1^a &= \{i \mid x_i \text{ fixed to 1 by branching decisions leading to } a\} \\ F_0^a &= \{i \mid x_i \text{ fixed to 0 by branching decisions leading to } a\} \end{aligned}$$

Two nodes a and b are *isomorphic* if

$$\exists \pi \in \Gamma \text{ with } \pi(F_1^a) = F_1^b, \pi(F_0^a) = F_0^b.$$

If two nodes are isomorphic, then you may prune one of the nodes a or b, as you can be sure that if there is an optimal solution in node a, then there is an optimal solution of the same value in b. This idea was developed and used in the combinatorics community and Bazaraa and Kirca (1983) is one of the first application in the context of integer linear programming.

Unfortunately, the problem of calculating whether any two nodes of a branch-and-bound tree are isomorphic is not known to be easy. However, the concept of *isomorphism pruning*, is a mechanism that will determine if a node that is set to be evaluated will be isomorphic to another evaluated node. This idea was developed and used in many different areas. See, for example, the work of Butler and Lam (1985), Read (1998), McKay (1998), Ivanov (1985), and the book of Kreher and Stinson (1999).

The key idea is to choose one unique *representative* for each potential set F_a^1. A set S is a *representative* of its equivalence class (or orbit) if

$$S = \text{lexmin}\{\pi(S) \mid \pi \in \Gamma\}.$$

Now the isomorphism pruning rule is very simple to implement at a node a: If F_1^a is not a representative, then prune node a. It has been shown that this is a valid pruning strategy, provided that at each node of the enumeration tree, the non-fixed variable with smallest index is always selected as branching variable.

Isomorphism pruning is a powerful technique that can extend the range of symmetric integer programs that can be solved. For example, for the Football Pool problem on five

matches, branch-and-bound with isomorphism pruning can establish that optimal solution of value $|C_5^*| = 27$ in 82 seconds and 1409 nodes of the enumeration tree, while the commercial solver CPLEX (v9.1) does not solve the problem in more than 4 hours and million of nodes. However, for six matches, isomorphism pruning by itself is only able to establish that $|C_6^*| \geq 61$, even after running for days. Our ultimate goal is to solve the Football Pool problem on six matches, so we will require other strategies besides branch-and-bound with isomorphism pruning to tackle this instance.

A variant of branch-and-bound with isomorphism pruning can be used to obtain *all* non isomorphic solutions to an integer program (Margot, 2003). The method works in a similar fashion to that proposed Danna et al. (2007). Namely, branching and pruning are performed until *all* variables are fixed. All leaf nodes of the resulting tree are non-isomorphic solutions to the system. This extension is necessary in order to perform some of the tasks to be described below.

2.2. Covering Systems

An upper bound (feasible covering code) of value 73 is known for the Football Pool Problem, and it is quite likely that 73 is indeed the optimal solution value. To establish that 73 is the optimal solution value, techniques for improving the lower bounds on $|C_v^*|$ are required. One such technique was introduced by Östergård and Blass (2001) and is based on partitioning the word set into subsets and counting the words in each subset that must be covered by a given codeword. The technique is best introduced by means of a simple example. Partition the set of words $W(6,3)$ into three subsets W_0, W_1, W_2, with W_j containing all words starting with letter j for $j \in \{0,1,2\}$. Let C be a covering code with $|C| = M$. Similar to the partition of $W(6,3)$, the words of C may be partitioned into words that begin with each letter: $C = C_0 \cup C_1 \cup C_2$. Let $y_j \stackrel{\text{def}}{=} |C_j|$ be the number of code words beginning with each letter $j \in \{0,1,2\}$. Observe that a word in C_0 covers 11 words in W_0, and a word in C_1 or C_2 covers one word in W_0. Since, for $j \in \{0,1,2\}$, all 243 words in W_j must be covered by words in C, the following linear system, which we refer to as the *M-covering system* is feasible:

$$\mathcal{M}_1 = \{(y_0, y_1, y_2) \in \mathbb{Z}_+^3 \mid 11y_0 + y_1 + y_2 \geq 243, y_0 + 11y_1 + y_2 \geq 243, \\ y_0 + y_1 + 11y_2 \geq 243, y_0 + y_1 + y_2 = M\} \quad (2.1)$$

In other words, a necessary condition for there to be a covering code of size M is that $\mathcal{M}_1 \neq \emptyset$.

Naturally, this same idea applies to a different word set partitioning. For example, if the words in $W(6,3)$ are partitioned into nine subsets based on their first two letters:

$$W = W_{00} \cup W_{01} \cup W_{02} \cup W_{10} \cup W_{11} \cup W_{12} \cup W_{20} \cup W_{21} \cup W_{22},$$

the following M-covering system is obtained:

$$\begin{aligned}\mathcal{M}_2 = \{(y_{00}, y_{01}, y_{02}, y_{10}, y_{11}, y_{12}, y_{20}, y_{21}, y_{22}) \in \mathbb{Z}_+^9 \mid\ & 9y_{00} + y_{01} + y_{02} + y_{10} + y_{20} \geq 81 \\ & y_{00} + 9y_{01} + y_{02} + y_{11} + y_{21} \geq 81 \\ & y_{00} + y_{01} + 9y_{02} + y_{12} + y_{22} \geq 81 \\ & y_{00} + 9y_{10} + y_{11} + y_{12} + y_{20} \geq 81 \\ & y_{01} + y_{10} + 9y_{11} + y_{12} + y_{21} \geq 81 \\ & y_{02} + y_{10} + y_{11} + 9y_{12} + y_{22} \geq 81 \\ & y_{00} + y_{10} + 9y_{20} + y_{21} + y_{22} \geq 81 \\ & y_{01} + y_{11} + y_{20} + 9y_{21} + y_{22} \geq 81 \\ & y_{02} + y_{12} + y_{20} + y_{21} + 9y_{22} \geq 81 \\ & y_{00} + y_{01} + y_{02} + y_{10} + y_{11} + y_{12} + y_{20} + y_{21} + y_{22} = M\}.\end{aligned} \tag{2.2}$$

The M-covering system forms the basis of a method for improving the lower bound on the size of an optimal covering code $|C_v^*|$. First a potential code cardinality $M = |C_v^*|$ is chosen. Let m be the number of components fixed to define the word set partitioning. For example, we have $m = 1$ to form the M-covering system (2.1) and $m = 2$ to form (2.2). All non-isomorphic solutions to the M-covering systems for word set partitioning $m = 1, 2, \ldots, 6$ are enumerated. If for some m, there are *no* solutions to the covering system, then no covering code with exactly M words exists.

Using this idea and induction on m for the enumeration, Östergård and Blass (2001) were able to prove that $M = 62$ is an *optimal* code length for $d = 1, v = 9, \alpha = 2$, and Östergård and Wassermann (2002) were able to show $M \geq 65$ for $d = 1, v = 6, \alpha = 3$. The enumeration procedure to establish the latter required over 1 CPU year using a distributed system of twenty six 400MHz and 500 MHz computers with the batch system `autoson` (McKay, 1996). The enumeration was based on the LLL basis lattice reduction algorithm of Lenstra et al. (1982). A drawback of the method is that for intermediate values of m, the

number of non-isomorphic solutions to the M-covering system sometimes becomes extremely large, and all all nonisomorphic solutions for $m = k$ must be enumerated before moving on to $m = k + 1$.

2.3. Integer Programming and Covering Systems

The performance of the enumeration procedure described in Section 2.2 can be improved by combining the enumeration with integer programming. Specifically, the enumeration can be carried out up to a certain point of refinement m of the word set. Then, integer programming can be used to establish that no covering code exists with the specific combination of words belonging to each of the word set partitions specified by the solution to the M-covering system. In what follows, we call a solution y to the M-covering system a *code sequence*. In our work, we focus on the case $m = 2$. For each code sequence

$$y = \{y_{00}, y_{01}, y_{02}, y_{10}, y_{11}, y_{12}, y_{20}, y_{21}, y_{22}\},$$

there is an associated integer program, which we call the *y-sequence IP* (y-SIP):

$$\min_{x \in \{0,1\}^n} \{e^T x \mid Ax \geq e, x \in \mathcal{M}(y)\}, \tag{2.3}$$

where

$$\mathcal{M}(y) = \{x \in \{0,1\}^n \mid \sum_{i \in W_{jk}} x_i = y_{jk} \text{ for } j, k \in \{0,1,2\}\}.$$

If for *every* solution y to the M-covering system found by the enumeration, the corresponding y-SIP (2.3) has no feasible solution, then no covering code with exactly M words exists.

3. Additional Steps

The combination of the enumeration of non-isomorphic solutions to M-covering systems (2.2) and the solution of the corresponding y- sequence IPs (2.3) forms the basis of our method for further improving the lower bound on $|C_6^*|$. However, we perform three additional steps that are designed to reduce the number of y-SIPs that must be solved and improve the speed with which y-SIPs are solved. First, for reasons linked to the isomorphism pruning algorithm employed, by reordering the components of the y vectors before using them in the y-SIP, the solution time of the y-SIP is significantly reduced. Second, recognizing that some sequences

are very similar to each other, aggregating some components of the sequence together and solving an aggregated version of the y-SIP is advantageous. Finally, by a preprocessing operation, many sequences may be removed from consideration as potentially leading to an optimal solution. Each of these steps is discussed in detail in this section.

3.1. Sequence Reordering

Given a covering code C of $W(v,3)$, we can by symmetry arrive at another covering code by choosing a permutation σ of $\{0,1,2\}$ and applying σ_ℓ to the letter in one particular position in all the words of C. We also can choose a permutation σ_v of the v entries of the words in $W(v,3)$ and create another code by permuting the entries in the codewords of C according to the permutation σ_v. Moreover, any combination of these two types of permutations can be applied, and a covering code will result.

These permutations applied to $W(v,3)$ also will induce permutations in the entries of a code sequence y. For example, assume that

$$y = (y_{00}, y_{01}, y_{02}, y_{10}, y_{11}, y_{12}, y_{20}, y_{21}, y_{22}).$$

The cyclic permutation $\hat{\sigma_\ell}$ sending $0 \to 1, 1 \to 2$, and $2 \to 0$ of the first letter of the words in $W(v,3)$ yields the vector

$$y' = (y_{20}, y_{21}, y_{22}, y_{00}, y_{01}, y_{02}, y_{10}, y_{11}, y_{12}).$$

Then, applying the permutation $\hat{\sigma_v}$ that swaps the first two letters in each of the words of W to y' yields

$$y'' = (y_{02}, y_{12}, y_{22}, y_{00}, y_{10}, y_{20}, y_{01}, y_{11}, y_{21}).$$

Since the permutations $\hat{\sigma_\ell}$ and $\hat{\sigma_v}$ we applied can be used to create permutations in the symmetry group Γ of the y-SIP (2.3), then y-SIP, y'-SIP and y''-SIP are either all feasible or all infeasible.

Thus, entries in y can be permuted in certain ways before solving y-SIP, and the result of the computation will still reliably conclude whether the instance is feasible or infeasible. We would like to permute the entries such that the resulting y-SIP instances will require a minimum amount of computational effort. For reasons linked to the branching rule used in the isomorphism-pruning implementation, it was determined that permuting the original sequences to result in the new sequence

$$y = (y_{00}, y_{01}, y_{02}, y_{10}, y_{20}, y_{11}, y_{12}, y_{21}, y_{22})$$

works best.

This ordering matters for the efficiency of the isomorphism pruning algorithm in branch-and-bound, but is otherwise irrelevant to the presentation. In the sequel, all code sequences are represented with this reordering of the components.

3.2. Sequence Aggregation

Components of the code sequences y can be aggregated together in order to reduce the number of y-SIPs that need to be solved to establish a lower bound. For example, suppose that the following four sequences were found during the enumeration procedure:

y_{00}	y_{01}	y_{02}	y_{10}	y_{20}	y_{11}	y_{12}	y_{21}	y_{22}
20	7	5	5	8	7	5	7	8
20	7	5	5	8	7	5	6	9
20	7	5	5	7	8	5	7	8
20	7	5	5	7	8	5	6	9.

By aggregating the last four entries of of the y vector, we get the sequences

y_{00}	y_{01}	y_{02}	y_{10}	y_{20}	$\bar{y}$
20	7	5	5	8	27
20	7	5	5	8	27
20	7	5	5	7	28
20	7	5	5	7	28.

Since we have only two distinct aggregated sequences, we need to solve only two integer programs. In general, the aggregated y-SIP for an aggregated sequence y has the form

$$\min_{x\in\{0,1\}^n} \{e^T x \mid Ax \geq e, x \in \mathcal{AM}(y)\}, \tag{3.1}$$

where

$$\mathcal{AM}(y) = \{x \in \{0,1\}^n \mid \sum_{i\in W_{00}} x_i = y_{00}, \sum_{i\in W_{01}} x_i = y_{01}, \sum_{i\in W_{02}} x_i = y_{02},$$
$$\sum_{i\in W_{10}} x_i = y_{10}, \sum_{i\in W_{20}} x_i = y_{20}, \sum_{i\in W_{11}\cup W_{12}\cup W_{21}\cup W_{22}} x_i = y_{11}+y_{12}+y_{21}+y_{22}\}.$$

Note that the aggregated y-SIP (3.1) should be harder to solve than each (non aggregated) y-SIP (2.3). After all, aggregating all nine entries of the sequences would get us back to the original problem. However, in the majority of the cases, the information loss due to the aggregation of the last four entries is more than compensated by the reduction in the number of y-SIPs that need to be solved.

Margot et al. (2003) use this technique to improve the lower bound for the Football Pool Problem on six matches to $|C_6^*| \geq 67$. The code sequences are obtained by enumerating the nonisomorphic solutions to the M-covering systems (2.2) by using the algorithm of Östergård and Blass (2001). For $M = 64, 65, 66$, the number of code sequences to handle is respectively 423, 839 and 1,674. These numbers drop to 27, 40, and 65 respectively after aggregation and regrouping (aggregation and regrouping use slightly different rules than those presented above). The total CPU time (in seconds) for solving the corresponding aggregated y-SIP on an IBM Thinkpad with a clock speed of 2.0 GHz is respectively 30,932, 95,160, and 580,080.

3.3. Sequence Exclusion

Another idea to reduce the number of y-SIP that must be solved is a preprocessing operation that excludes sequences that cannot lead to a code of cardinality smaller than the best known code of size 73. Suppose for example, we could establish that in any covering code, there must be at least 19 words from W_0 in the code, i.e. $y_0 \geq 19$. If such a fact were known, then all sequences with $y_{00} + y_{01} + y_{02} \leq 18$ could be immediately discarded. Proving that $y_0 > 18$ is a matter of establishing that the following integer program has no feasible solution:

$$\min_{x \in \mathbb{B}^n} \{e^T x \mid Ax \geq e, \sum_{i \in W_0} x_i \leq 18\}. \tag{3.2}$$

By symmetry, we can assume that the code C satisfies $|C \cap W_0| \leq |C \cap W_i|$ for $i = 1, 2$, so the constraints $\sum_{i \in W_1} x_i \geq 18$ and $\sum_{i \in W_2} x_i \geq 18$ can be added to (3.2) without affecting the outcome. Solving (3.2) takes about 15 minutes for a branch-and-bound code equipped with isomorphism pruning (Margot, 2002). We can extend this approach further. Since the infeasibility of (3.2) established that at least 19 words in a covering code for $W(6, 3)$ begin with 0, we can ask if there does indeed exist a covering code such that $y_0 = 19$. In general, if we know that no covering code C with $|C \cap W_0| < p$ words exists, we can check if one exists with $|C \cap W_0| = p$ and $|C \cap W_i| \geq p$ for $i = 1, 2$ by solving the following sequence-exclusion integer program:

$$\min_{x \in \{0,1\}^n} \{e^T x \mid Ax \geq e, \sum_{i \in W_0} x_i = p, \sum_{i \in W_k} x_i \geq p \quad k \in \{1, 2\}, e^T x \leq 72\} \tag{3.3}$$

It required more than a week of CPU time to prove that (3.3) is infeasible for $p = 19$. The increased difficulty of (3.3) from $p = 18$ to $p = 19$ makes the solution of the case $p = 20$ by this method unattractive.

3.3.1. Sequence Squashing

During the computations for solving sequence-exclusion IP (3.3) for $p = 18$ and $p = 19$, it was noted that a significant portion of the CPU time was used in solving the linear programming relaxations. Therefore, another procedure, called *squashing*, was used to solve (3.3) much more quickly. The technique works by aggregating variables together, enumerating potential solutions, and then proving that none of the potential solutions leads to a feasible solution for (3.3). This squashing procedure allowed us to also solve the sequence exclusion IP (3.3) for the cases $p = 20$ and $p = 21$.

The squashing procedure begins by replacing two "similar" variables in (3.3) by one aggregated version of that variable. For example, suppose that x_{1a} and x_{2a} are variables associated with words $1a \in W_1$ and $2a \in W_2$, i.e. two words differing only in their first entry, respectively. A new continuous variable $0 \leq z_{1a} \leq 2$ is created. This variable creation is done for all words in $W_1 \cup W_2$, which results in a squashed version of sequence-exclusion IP:

$$\min_{(x,z)\in\{0,1\}^{n/3}\times[0,2]^{n/3}} \{e^T x + e^T z \mid Ax \geq e, \sum_{i\in W_0} x_i = p, \sum_{i\in W_1} z_i \geq 2p, e^T x + e^T z \leq 72\}. \quad (3.4)$$

The squashed sequence-exclusion IP (3.4) is a relaxation of the original sequence-exclusion IP (3.3).

The next step in the squashing procedure is to enumerate all non-isomorphic x that are part of a solution to (3.4). Then, each such vector x is substituted into (3.3), and the resulting, much simpler, integer program is solved via a "regular" branch-and-bound algorithm. If (3.3) is infeasible after fixing the x portion of each non-isomorphic solution to (3.4), then the original (3.3) is also infeasible. This complicated way to solve (3.3) is justified by the fact that the linear relaxation of the squashed problem is solved much faster than the one of (3.3), as well as the reoptimization required during the branch-and-bound enumeration for solving (3.3).

Enumerating all non isomorphic feasible solutions x for (3.4) for $p = 19$ requires less than 10 CPU minutes and results in 169 x vectors that need to be plugged into the sequence-exclusion IP (3.3). All but two of these integer programs are solved in less than 0.5 seconds by an unsophisticated branch-and-bound algorithm based on the COIN-OR software `BCP`. The last two instances were solved using CPLEX9.1 and each requires less than 80 seconds to be shown infeasible. Taking all the operations of this alternative squashing procedure together,

proving that (3.3) is infeasible for $p = 19$ takes less than 15 minutes CPU time, as opposed to more than one week for the straightforward application of isomorphism-pruning-based branch-and-bound.

This great increase in efficiency from the squashing procedure spurred us on to attempt to solve the sequence-exclusion IP (3.4) for $p = 20$ and $p = 21$. Enumerating the non-isomorphic solutions to (3.4) that may lead to a feasible solution to (3.3) can be a very CPU-intensive procedure for larger values of p. As such, the enumerations was done in parallel, using (a portion) of the high-throughput computing grid that we introduce in full detail in Section 4. The enumeration portion of the $p = 20$ calculation required a total of 1088 CPU hours. The total wall clock time of the computation was 6.3 hours, as the calculation ran on an average of roughly 200 machines simultaneously. There were 9451 non-isomorphic solutions to (3.4) for $p = 20$. Solving all 9451 sequence-exclusion IPs (3.3) with the corresponding x portion fixed required just over 8 CPU hours using the branch-and-bound code BCP.

The calculations required to solve (3.4) for $p = 21$ were even more extensive. There were $385,967$ non-isomorphic solutions to (3.4) for $p = 21$, and enumerating these solutions required a total of $49,023$ CPU hours (5.6 CPU years). The 385,967 instances of (3.3) were solved in parallel on a Beowulf Cluster consisting of 264 processors, and 83 CPU days was required to solve (most) of the instances. Some of the instances (6,535 of the 385,967) took more than 10 minutes for the rudimentary branch-and-bound code BCP to solve. These 6,535 difficult instances were solved with the more sophisticated IP solver MINTO, which required another 139 hours of CPU time. None of the 385,967 IP instances has a feasible solution. This establishes that any covering code of $W(6, 3)$ contains at least 22 words that begin with 0, and thus (by symmetry), proves the result $|C_6^*| \geq 66$, improving on the previously best-published result by Östergård and Wassermann (2002).

3.4. Impact of Novel Techniques

Establishing that any covering code of $W(6, 3)$ contains at least 22 codewords from W_0 was particularly important for our approach for improving the lower bound on $|C_6^*|$ and (hopefully) eventually proving the optimality of the code of size 73. Specifically, given the demonstration that there does not exist a solution of (3.3) for $p = 21$, the constraints

$$y_{00} + y_{01} + y_{02} \geq 22, \quad y_{10} + y_{11} + y_{12} \geq 22, \quad \text{and} \quad y_{20} + y_{21} + y_{22} \geq 22$$

can be added to the M-covering system (2.2). With these constraints added to the M-covering system, the enumeration of *all* non-isomorphic solutions to the M-covering system for all values $M \in \{66, 67, 68, \ldots, 72\}$ can be accomplished. In Table 2, we show the number of non-isomorphic solutions to the enhanced M-covering system for each value of M. In total, there are 91,741 solutions, and after regrouping and aggregation, there are 1000 aggregated sequence IPs of the form (3.1). Solving all IPs for a given value of M, and demonstrating that there is no feasible solution to any of them, establishes a lower bound of $|C_6|^* \geq M + 1$.

M	**# Seq.**	**# Agg. Seq.**
66	797	7
67	1,723	13
68	3,640	45
69	7,527	102
70	13,600	176
71	24,023	264
72	40,431	393
	91,741	1000

Table 2: # Sequences and Aggregated Sequences for each value of M

4. The Computational Grid

The aggregated y-SIP's from the enumeration still contain a large amount of symmetry and for many sequences y can be very difficult to solve. Therefore, we would like to employ the isomorphism-pruning-based branch-and-bound algorithm and use a powerful distributed computing platform to solve each instance.

Of particular interest to us in this work are large 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. This type of computing platform is often known as a *computational grid*, (Foster and Kesselman, 1999). Computational grids can be very powerful, but they can also be difficult to use effectively. In this section, we discuss two software toolkits that allow us to build a computational grid, Condor and MW. In addition, we discuss a variety of mechanisms for building large-scale computational grids, and we address issues in scaling the branch-and-bound algorithms to run effectively on such a platform.

4.1. Condor

Condor (`http://www.cs.wisc.edu/condor`) is a job management system for compute-intensive jobs (Litzkow et al., 1998). Condor provides a job queueing mechanism, scheduling policy, resource monitoring, and resource management. Condor runs as a collection of daemon processes that perform the necessary services. Users submit their jobs to the Condor scheduler daemon (the `schedd`), which places the jobs in a queue. Jobs are run when machines meeting the job's requirements become available. Condor carefully monitors the progress of the running job, and informs the user upon the job's completion.

Condor provides these "traditional" batch-queueing system features, but also offers additional functionality that is especially relevant for building the computational grids for our applications. Specifically, Condor has mechanisms to effectively harness wasted CPU power from otherwise idle desktop workstations. For instance, Condor can be configured run jobs on desktop machines where the keyboard and mouse are idle. Should Condor detect that a machine is no longer available (such as a key press detected), in many circumstances Condor is able to transparently checkpoint the running job, and migrate the job to a different machine without any loss of the computation. Condor does not require a shared file system across machines. Condor can transfer files on behalf of the user, or Condor may be able to transparently redirect all the job's I/O requests back to the submitting machine.

Condor is especially well-designed for completing jobs in a *high-throughput* manner. That is, jobs that require large amounts of processing over long periods of time. This is in contrast to traditional *high performance* computing, in which jobs require large computing power over a relatively short interval.

Many companies and universities are building collections of processors that run the necessary Condor daemons to create a Condor pool at that site. In our computation, we use Condor pools at the University of Wisconsin and Lehigh University that all together total over 3000 processors.

4.2. Existing Computational Grids

There are currently two large US national initiatives aimed at building large federations of computing resources. We use processors from both of these grids in our computations.

The TeraGrid (`www.teragrid.org`) is an open scientific discovery infrastructure combining large computing resources at nine partner sites: Indiana, NCAR, NCSA, ORNL, PSC,

Purdue, SDSC, TACC and UC/ANL. The Teragrid consists of over 100 teraflops of computing power, and over 3 petabytes of computing storage. The partner sites are interconnected at 10/30 gigabits/second via a dedicated national network. Access to TeraGrid is available through scientific peer review, at no cost, to any academic researcher in the United States.

The Teragrid is built using a top-down approach for federating computational resources. In contrast, the Open Science Grid (OSG) (`www.opensciencegrid.org`) uses a bottom-up approach, bringing together computing and storage resources from campuses and research communities into a common, shared grid infrastructure via a common set of middleware. OSG offers participating research communities low-threshold access to more resources than they could afford individually. At the time of this writing, there are 75 sites on the Open Science Grid, and they are organized into 30 *virtual organizations*. Users at sites in the same virtual organization can share computational resources. We use OSG resources from Wisconsin, Nebraska, Caltech, Arkansas, Brookhaven National Lab, MIT, Purdue, and Florida in our computation.

4.3. Grid-Building Mechanisms

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 (like those on the Teragrid and Open Science Grid) 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 (Epema et al., 1996),

— A manual version of Condor glide-in (Frey et al., 2002) 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* (Bradley, 2006),

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.

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 user's perspective, flocking

is the most transparent way to aggregate resources. 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 TCP/IP ports from the flocked-from scheduler to the flocked-to machines, so it is difficult to use where network firewalls exist.

Glide-in. Condor glide-in is a way to construct an overlay Condor pool on top of another batch queueing system. This overlay pool can then report to an existing pool. Typically Condor glide-in is used to access resources via a Globus gateway. Globus is an open source software toolkit used for building Grid systems and applications (Foster and Kesselman, 1997). See the URL `www.globus.org` for information on Globus. 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 and jobs are scheduled by the local Condor job scheduler. Condor glide-in is most useful when we have access to a non-Condor batch system, such as systems on the Teragrid, 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 that we call 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 batch-scheduled supercomputing sites, the most effective strategy for obtaining significant CPU resources is to make many requests for small numbers of processors and for short duration. In this way, the batch requests are run more quickly, as they can be fulfilled from the backfill of local schedulers, by "squeezing" them into empty slots on the supercomputer.

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. Required 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 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 a Globus-enhanced version of Condor called Condor-G to submit the jobs 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 different queue, and the schedd-on-the-side can do matching and scheduling of jobs to one of many remote Grid sites. This is an easy way to take advantage of large systems like the Open Science Grid.

Putting it all together. Table 3 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. In total, there are over 19,000 processors available, but we will not have access to all of them at any one time. The sites that begin with OSG are processors on the Open Science Grid, and the sites that begin with TG are TeraGrid installations.

4.4. MW

The grid-building mechanisms outlined in Section 4.3 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 (Goux et al., 2001). 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,

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 3: Characteristics of Our Computational Grid

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 features for user-defined checkpointing, normalized application and network performance measurements, and methods for the dynamic prioritization of computational tasks. This functionality is explained in greater detail in the papers Goux et al. (2001); Glankwamdee and Linderoth (2006) and the MW User's Manual Linderoth et al. (2007). 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 in Section 4.5.

MW has been used to instrument branch-and-bound algorithms for the quadratic assignment problem (Anstreicher et al., 2002), mixed integer nonlinear programs (Goux and Leyffer, 2003), and for mixed integer linear programs by Chen et al. (2001). In this work, the solver in (Chen et al., 2001), called FATCOP, was augmented with the isomorphism pruning techniques discussed in Section 2.1 and used in our attempt to improve the lower bounds on $|C_6^*|$ by solving the aggregated sequence IPs of Table 2 for consecutively larger values of M.

4.5. 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 evaluation of one 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, and many other authors have also suggested a similar strategy (Anstreicher et al., 2002; Xu et al., 2005). 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 re-ordering 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 to improve the lower bound on $|C_6^*|$ ran for months across thousands of machines, so failures which would be rare on a single-processor become common. Further, as discussed in Section 4.3, 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 `read_ckpt_info()`

and `write_ckpt_info()` methods of the `MWTask` class and the `read_master_state()` and `write_master_state()` methods 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 3, our workers are run with low priority, and the scheduling policy at the sites is to simply suspend the low priority job, do the process "hangs", rather than to preempt the low priority job, so the process "fails." 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 the 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.

A more sever problem occurred when the job suspension occurred during the middle of an active TCP write to the master process. 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.

Another way in which we needed to change the MW code to account for the scale of our computations, was to re-write MWś socket management layer. The `poll()` function can have a maximum of 4096 open sockets, and since we require one socket from the master to each workers, we could never use more than 4096 workers simultaneously. Re-writing the socket management code to use the `epoll()` function removed this limitation, and in our most recent computations, we have used more than 4500 processors from Table 3 simultaneously.

An apparent shortcoming of the simple master-worker task-distribution scheme is that the underlying architecture is not theoretically scalable. As the number of worker processes increases, the single master process may not be able to efficiently handle all incoming requests for work. In this work, by engineering the branch-and-bound algorithm properly, and by instrumenting the MW code to effectively deal with events occurring in such a large, distributed, system, our algorithm scales very effectively to over 4000 workers processors.

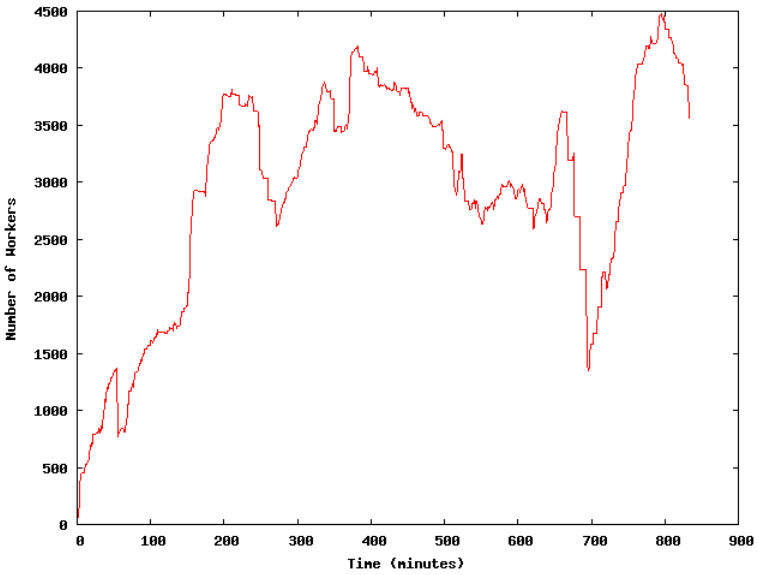
Table 4: Computation Statistics

	$M = 69$	$M = 70$
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}

5. Computational Results

The solution of the integer programs in Table 2 to solve the football pool problem have been ongoing since 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 $|C_6^*| \geq 71$ for the football pool problem, an improvement of 6 over the best published bound by Östergård and Wassermann (2002). In Table 4, we show aggregated computational results for a portion of our computation. Specifically, we show the work required to solve the aggregated y-sequence IPs in Table 2 to establish that $|C_6^*| \geq 70 (M = 69)$ and $|C_6^*| \geq 71$ $(M = 70)$. 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. (2002) required 11 CPU years to solve the nug30 quadratic assignment problem, Mezmaz et al. (2006) used 22 CPU years to solve a flow-shop problem by branch and bound, and Applegate et al. (2006) 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 4.5, we have used simultaneously over 4,500 workers while solving IPs related to establishing $|C_6^*| \geq 72$. Figure 2 shows the number of workers used during a portion of the run in which this maximum was reached.

Figure 2: Workers During Portion of Aggregated Sequence IP Calculation



6. Conclusions

In this work, we have performed extensive high-throughput computations aimed at improving the lower bound on the Football Pool Problem on six matches. To date, we have been able to establish that $|C_6^*| \geq 71$, which is a significant improvement over the previously best published bound of $|C_6^*| \geq 65$. Besides being (to our knowledge) the largest computation of its kind ever undertaken, novel aspects in this work are the combination of isomorphism-free enumeration, aggregation, and integer programming used in establishing the bound. Also novel is the demonstration that properly engineered branch-and-bound algorithms can efficiently scale to over 4500 processors. We note that our work is not a "proof" in the mathematical sense, as the certificate of the proof would involve demonstrating that all computer codes used in our work performed flawlessly. Nevertheless, we view our work as significant evidence of an improvement in the lower bound. Even more important, we hope that this work demonstrates to the Operations Research community the tremendous computing power available on computational grids, especially if the processors are used in a flexible, opportunistic manner. The high-throughput computation continues, and we hope to establish soon that the optimal code size for the football pool problem on 6 matches is $|C_6^*| = 73$.

Acknowledgements

This work has been supported in part by the National Science Foundation, through grant agreements OCI-0330607 and 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. The authors would like to thank Preston Smith of Purdue University for his help in accessing the computing resources there, Dan Bradley of the University of Wisconsin-Madison for help in accessing OSG resources, and the entire Condor team for their tireless efforts to provide a really useful computing infrastructure.

References

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

Applegate, D., R. Bixby, W. Cook, V. Chvátal. 2006. Personal Communication.

Barnhart, C., E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, P. H. Vance. 1998. Branch and price: Column generation for solving huge integer programs. *Operations Research* **46** 316–329.

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

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

Butler, G., W. H. Lam. 1985. A general backtrack algorithm for the isomorphism problem of combinatorial objects. *Journal of Symbolic Computation* **1** 363–381.

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

Danna, E., M. Fenelon, Z. Gu, R. Wunderling. 2007. Generating multiple solutions for mixed integer programming problems. M. Fischetti, D. Williamson, eds., *IPCO 2007: The*

Twelfth Conference on Integer Programming and Combinatorial Optimization. Springer, 280–294.

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

Foster, I., C. Kesselman. 1997. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications* **11** 115–128.

Foster, I., C. Kesselman. 1999. Computational grids. I. Foster, C. Kesselman, eds., *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 15–52. Chapter 2.

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

Glankwamdee, W., J. Linderoth. 2006. MW: A software framework for combinatorial optimization on computational grids. E. Talbi, ed., *Parallel Combinatorial Optimization*. John Wiley & Sons, 239–261.

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

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

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

Holm, S., M. Sørensen. 1993. The optimal graph partitioning problem: Solution method based on reducing symmetric nature and combinatorial cuts. *OR Spectrum* **15** 1–8.

Ivanov, A. V. 1985. Constructive enumeration of incidence systems. *Annals of Discrete Mathematics* **26** 227–246.

Kaibel, V., M. Pfetsch. 2007. Packing and partitioning orbitopes. *Mathematical Programming* To appear.

Kreher, D. L., D. R. Stinson. 1999. *Combinatorial Algorithms, Generation, Enumeration, and Search*. CRC Press.

Lenstra, A. K., H. W. Lenstra, L. Lovász. 1982. Factoring polynomials with rational coefficients. *Mathematische Annalen* **261** 515–534.

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

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

Macambira, E. M., N. Maculan, C. C. de Souza. 2004. Reducing symmetry of the SONET ring assignment problem using hierarchical inequalities. Tech. Rep. ES-636/04, Programa de Engenharia de Sistemas e Computação, Universidade Federal do Rio de Janeiro.

Margot, F., , P. Östergård. 2003. Unpublished results.

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

Margot, F. 2003. Small covering designs by branch-and-cut. *Mathematical Programming* **94** 207–220.

McKay, B. 1996. `autoson`—A distributed batch system for UNIX workstation networks (version 1.3). Technical report, Computer Sciences Department, Australian National University.

McKay, D. 1998. Isomorph-free exhaustive generation. *Journal of Algorithms* **26** 306–324.

Méndez-Díaz, I., P. Zabala. 2006. A branch-and-cut algorithm for graph coloring. *Discrete Applied Mathematics* **154** 826–847.

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

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

Östergård, P., A. Wassermann. 2002. A new lower bound for the football pool problem for six matches. *Journal of Combinatorial Theory, Ser. A* **99** 175–179.

Read, R. C. 1998. Every one a winner or how to avoid isomorphism search when cataloguing combinatorial configurations. *Annals of Discrete Mathematics* **2** 107–120.

Sherali, H. D., J. C. Smith. 2001. Improving zero-one model representations via symmetry considerations. *Management Science* **47** 1396–1407.

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

Xu, Y., T. K. Ralphs, L. Ladányi, M.J. Saltzman. 2005. ALPS: A framework for implementing parallel search algorithms. *Proceedings of the Ninth INFORMS Computing Society Conference*. 319–334.