Mathematical Programming manuscript No.
(will be inserted by the editor)

Kurt Anstreicher · Nathan Brixius · Jean-Pierre Goux* · Jeff Linderoth**

Solving Large Quadratic Assignment Problems on Computational Grids.

the date of receipt and acceptance should be inserted later

Abstract. The quadratic assignment problem (QAP) is among the hardest combinatorial optimization problems. Some instances of size $n = 30$ have remained unsolved for decades. The solution of these problems requires both improvements in mathematical programming algorithms and the utilization of powerful computational platforms. In this article we describe a novel approach to solve QAPs using a state-of-the-art branch-and-bound algorithm running on a federation of geographically distributed resources known as a computational grid. Solution of QAPs of unprecedented complexity, including the nug30, kra30b, and tho30 problems, is reported.

Key words. Quadratic assignment problem – branch and bound – computational grid – metacomputing

1. Introduction

The quadratic assignment problem (QAP) is a standard problem in location theory. The QAP in "Koopmans-Beckmann" form is to

$$\min \quad \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij} b_{\pi(i),\pi(j)} + \sum_{i=1}^{n} c_{i,\pi(i)},$$

where a_{ij} is the flow between facilities i and j, b_{kl} is the distance between locations k and l, c_{ik} is the fixed cost of assigning facility i to location k, and $\pi(i) = k$ if facility i is assigned to location k. The problem can alternatively be represented in matrix form

Kurt Anstreicher: Department of Management Sciences, University of Iowa, Iowa City, IA 52242 `kurt-anstreicher@uiowa.edu`

Nathan Brixius: Department of Computer Science, University of Iowa, Iowa City, IA 52242 `brixius@cs.uiowa.edu`

Jean-Pierre Goux: Department of Electrical and Computer Engineering, Northwestern University, and Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439, `goux@ece.nwu.edu`

Jeff Linderoth: Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439, `linderot@mcs.anl.gov`

Mathematics Subject Classification (1991): XXX

* Research of this author is supported under NSF grant CDA-9726385.

** Research of this author is supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U. S. Department of Energy, under Contract W-31-109-Eng-38 and under NSF grant CDA-9726385.

as

$$\mathrm{QAP}(A,B,C): \quad \min \mathrm{tr}(AXB+C)X^T \\ \text{s.t. } X \in \Pi,$$

where $\mathrm{tr}(\cdot)$ denotes the trace of a matrix, Π is the set of $n \times n$ permutation matrices, and $X_{ik} = 1$ if facility i is assigned to location k. Throughout we assume that A and B are symmetric. Well-known applications for the QAP include the layout of manufacturing facilities and hospitals, and ergonomic design. For recent surveys on the QAP see [6, 43].

The QAP is NP-hard, and in practice has proven to be one of the hardest discrete optimization problems to solve to optimality. Problems of size $n = 20$ are challenging, and problems with $n = 25$ have been until very recently the practical limit for exact solution techniques. A variety of heuristics for QAP have been studied, including tabu search [47], simulated annealing [48], GRASP [44], and ant systems [14]. These methods often produce optimal or near-optimal solutions, but their performance can be quite variable (see for example Tables 2 and 4 in [14]).

Most exact solution methods for the QAP have been of the branch and bound (B&B) type. A polyhedral approach to the QAP has also been considered [29,42], but at this point is not competitive with B&B methods. A crucial factor in the performance of B&B algorithms for the QAP is the choice of lower-bounding method. A variety of lower-bounding techniques are known, including the Gilmore-Lawler Bound (GLB), bounds based on linear programming (LP) and dual-LP relaxations, bounds based on eigenvalues, and semidefinite programming (SDP) bounds. Most successful B&B implementations have used the GLB, which is easy to compute but unfortunately tends to deteriorate in quality as the dimension n increases. The LP [45] and SDP [50] bounds are typically tighter on large problems, but can be prohibitively expensive for use in B&B. For implementation in a B&B algorithm the relationship between the quality of a bound and its computational cost is extremely important, as is the quality of the branching information that bound computations provide.

A new convex quadratic programming bound (QPB) for the QAP was introduced in [1], and incorporated into a complete B&B algorithm in [4] (see also [3]). The QPB is based on the projected eigenvalue bound PB of [22], and uses an SDP characterization of a basic eigenvalue bound from [2]. The QPB typically provides better bound quality than the GLB and PB bounds, but at lower computational cost than LP or SDP bounds. The computational results in [4] indicate that for many larger ($n > 20$) QAPs, the B&B algorithm based on QPB has state-of-the-art performance, being rivaled only by the dual-LP based algorithm of Hahn et al. [24,25].

Because of the extreme difficulty of the QAP, exact solution methods have often been implemented on high-performance computers. In the past ten years a number of landmark computational results for the QAP have been obtained using parallel processing hardware. The most commonly used benchmark QAPs are the "Nugent" problems, introduced in 1968 [41]. The original set consisted of problems of size 5, 6, 7, 8, 12, 15, 20 and 30, but instances of several other sizes (14, 16, 17, 18, 21, 22, 24, and 25) have been constructed over the years by modifying the data from larger problems. The nug20 problem was first solved on a 16-processor MEIKO computing system in 1995

[10], and the larger nug21/22 problems were solved two years later on a NEC Cenju-3 [5], using up to 96 processors. The nug25 problem was solved in 1998 [38] using a collection of hardware that included a 128-processor Paragon XP/S22.

Although traditional parallel processing (super)computers continue to become more and more powerful, these resources have inherent limitations arising from their expense and lack of availability. An alternative platform for massively distributed computation is based on the notion of *grid computing*, also referred to as *metacomputing* [13]. A computational grid consists of a potentially large number of geographically dispersed CPUs linked by a communication medium such as the Internet. The advantage of such a platform compared to a traditional multiprocessor machine is that a large number of CPUs may be assembled very inexpensively. The *disadvantage* is that the availability of individual machines is variable, and communication between processors may be very slow.

In this paper we consider a grid computing implementation of the B&B algorithm for the QAP from [4]. In Section 2 we review the algorithm, and describe extensions to branching rules from [4] that we employ here. In Section 3 we consider the problem of estimating the performance of the algorithm on large problems. This issue is important both for selecting various problem-specific algorithm parameters, and for estimating the computational resorces that will be required to solve a given instance. In Section 4 we give some background on grid computing and describe the computational resources that we employ. We make use of the "Master-Worker" parallel computing paradigm, as implemented in the MW class library [18,20]. MW itself utilizes the Condor system [37] to locate available worker CPUs, and manage the interaction between workers and the master machine. Using MW/Condor we have assembled a computational grid of over 2500 CPUs that may participate in the solution of a single problem. In Section 5 we describe details of the MW implementation of our QAP algorithm, `MWQAP`, that have a significant impact on its distributed-processing performance. In Section 6 we give computational results on a number of previously-unsolved large QAPs from QAPLIB [7], including the nug30 problem. For the solution of nug30 an average of 650 worker machines were utilized over a one-week period, providing the equivalent of almost 7 years of computation on a single HP9000 C3000 workstation. The computations to solve another problem, the tho30 QAP, were the equivalent of over 15 years of computation on a single C3000. To our knowledge these are among the most extensive computations ever performed to solve discrete optimization problems to optimality.

Notation. We use $\mathrm{tr}(\cdot)$ to denote the trace of a matrix. If A and B are matrices then $A \bullet B = \mathrm{tr}(AB^T)$, $A \otimes B$ is the Kronecker product of A and B, and $\mathbf{vec}(A)$ is the vector formed by "stacking" the columns of A in the natural order. We use e to denote a vector with each component equal to one. The cardinality of a set D is denoted $|D|$. For convenience we let the name of an optimization problem, like $\mathrm{QAP}(A, B, C)$, also refer to the optimal value of the problem.

2. The branch-and-bound algorithm

The parallel B&B algorithm developed here is based on the serial algorithm described in [4]. The algorithm of [4] uses a quadratic programming lower bound for the QAP introduced in [1]. The quadratic programming bound for $\mathrm{QAP}(A,B,C)$ is of the form

$$\begin{aligned}\mathrm{QPB}(A,B,C): \qquad & \min \mathbf{vec}(X)^T Q\, \mathbf{vec}(X) + C \bullet X + \gamma \\ & \text{s.t. } Xe = X^T e = e \\ & \quad\; X \geq 0,\end{aligned}$$

where $Q = (B \otimes A) - (I \otimes S) - (T \otimes I)$. Let V be a matrix whose columns are an orthonormal basis for the nullspace of e^T. The matrices S and T, and constant γ, are obtained from the spectral decompositions of $V^T AV$ and $V^T BV$. By construction Q is positive semidefinite on the nullspace of the equality constraints $Xe = X^T e = e$, so computing QPB requires the solution of a convex quadratic programming problem. QPB is closely related to the projected eigenvalue bound PB of [22], and by construction $\mathrm{PB}(A,B,C) \leq \mathrm{QPB}(A,B,C) \leq \mathrm{QAP}(A,B,C)$. See [1] for details.

A computational approach for QPB based on the Frank-Wolfe (FW) algorithm [40] is described in [4]. The FW algorithm is known to have poor asymptotic performance, but in the the context of QPB is attractive because the computation required at each iteration is dominated by a single matrix multiplication and the solution of a dense linear assignment problem. See [4] for details. The FW algorithm can be run for any number of iterations, and produces a lower bound z and dual matrix $U \geq 0$ such that

$$\mathrm{tr}(AXB + C)X^T \geq z + U \bullet X,$$

for any permutation matrix X. The B&B algorithm of [4] makes extensive use of the dual matrix U in the branching step. Note that if v is the objective value of a known discrete solution to QAP (i.e. the incumbent value), then $z + U_{ij} > v$ implies that $X_{ij} = 0$ in any optimal solution of QAP. The branching process in [4] uses "polytomic" branching [39], where child problems are created by either assigning a fixed facility to all available locations (*row branching*), or by assigning all available facilities to a fixed location (*column branching*). In both cases logic based on U can be used to eliminate child nodes when branching. Several different branching rules are considered in [4], two of which (Rules 2 and 4) we employ here. We describe these below as they would be implemented at the root node, using row branching. Any node in the B&B tree corresponds to a set of fixed assignments, resulting in a smaller QAP over the remaining facilities and locations on which the implementation of branching is similar. Let $N = \{1, 2, \ldots, n\}$.

Rule 2: Branch on the row i that produces the smallest number of children. In the event of a tie, choose the row with the largest value of $\sum_{j \in N_i'} U_{ij}$, where $N_i' = \{j \in N \mid z + U_{ij} < v\}$.

Rule 4: Let I denote the set of rows having the NBEST highest values of $\sum_{j \in N} U_{ij}$. For each $i \in I$, and $j \in N$, compute a lower bound z^{ij} by forming the reduced problem $\mathrm{QAP}(A', B', C')$ corresponding to $X_{ij} = 1$, and obtaining $\mathrm{QPB}(A', B', C')$. Let U^{ij} be the dual matrix associated with z^{ij}. Let v^{ij} be the maximal row sum of U^{ij}, and let $w^{ij} = (|N| - 1)z^{ij} + v^{ij}$. Branch on the row $i \in I$ having the highest value of $\sum_{j \in N} w^{ij}$.

Table 1. A depth-based branching strategy

Depth	Rule	NFW1	NFW2	NFW3	NBEST	UPDATE
2	4	150	100	50	30	30
4	4	150	100	25	5	30
6	2	150	100	–	–	30
50	2	100	75	–	–	30

Rule 2 simply minimizes the number of children, and in the case of a tie attempts to maximize the bounds of the child nodes. Rule 4 uses prospective bound computations (the z^{ij}) to obtain more information about child bounds before deciding where to branch. In the general B&B context Rule 4 is an example of a *strong branching* rule, see for example [34]. Because of the use of the dual matrices U^{ij}, Rule 4 can also be viewed as a *look-ahead* procedure that tries to maximize the bounds of child nodes two levels deeper in the tree.

Many QAP problems have distance matrices arising from rectangular grids, resulting in symmetries that can be used to reduce the number of children. For such a problem there is a subset of the locations J_1 such that the children of the root node can be restricted to $X_{ij} = 1, j \in J_1$, regardless of the choice of i. In addition, there may be one or more subsets of locations J_2 so that if the set of fixed locations $\bar{J}$ satisfies $\bar{J} \subset J_2$, then the children can be restricted to be of the form $X_{ij} = 1, j \in J_3$, regardless of i. At a node where symmetry can be exploited we consider only row branching, and replace the index set N with an appropriate $J \subset N$. If symmetry is *not* present we consider column branching as well as row branching. The modifications of Rules 2 and 4 to implement column branching are straightforward.

To completely specify a branching rule several parameters must be set. One example is NBEST, described in Rule 4 above. Other required parameters are:

NFW1: Maximum number of FW iterations used.
NFW2: Maximum number of FW iterations used if node cannot be fathomed.
NFW3: Number of FW iterations for prospective bound computations in Rule 4.
UPDATE: Number of FW iterations between update of matrices S, T.

In bound computations a maximum of NFW1 Frank-Wolfe iterations are used, but a node may fathom using fewer iterations. On the other hand the FW algorithm may determine that it will be *impossible* to fathom a node, in which case the maximum number of iterations is reduced to NFW2. The matrices S and T used to define QPB are also periodically updated in an effort to improve the lower bound z, see [4, Section 3] for details.

In [4] branching rules are combined to form a complete branching strategy based on the *depth* of a node in the tree. An example of such a strategy, using only Rules 2 and 4, is given in Table 1. The "depth" parameter specifies the maximum depth on which a rule is used. Several aspects of the branching strategy in Table 1 are noteworthy. In general the use of a more elaborate branching rule (like Rule 4) is worthwhile to reduce the size of the tree. However, because of the typical growth in nodes such a rule becomes computationally more and more expensive as the depth increases. This cost can be mitigated by decreasing NBEST and NFW3 at intermediate depths. Eventually

Table 2. A branching strategy based on gap and depth

Gap	Depth	Rule	NFW1	NFW2	NFW3	NBEST	UPDATE
.42	3	4	150	150	100	30	30
.32	5	4	150	150	50	30	30
.18	5	4	150	100	25	5	30
.09	7	2	150	100	–	–	30
.04	8	2	100	75	–	–	30
0	50	2	75	50	–	–	30

the strategy switches to the cheaper branching rule (Rule 2). At very deep levels, where a high fraction of nodes will fathom and many children are eliminated, further economy is obtained by reducing NFW1 and NFW2.

In [4] excellent computational results are obtained on problems up to size $n = 24$ using depth-based branching strategies. In attempting to solve larger problems, however, limitations of strategies based entirely on depth became apparent. Because of the growth in nodes with depth it becomes impractical to use Rule 4 in a depth-based strategy beyond level 4 or 5. However, on larger problems few nodes at this depth are fathomed, suggesting that the use of more information than obtained by Rule 2 could be very beneficial. Examining the distribution of bounds on each level it became clear that the nodes on a given level are often quite heterogenous. For example, two nodes on level 5 might both fail to fathom, with gaps (incumbent value minus lower bound) equal to .1 and .7 of the gap at the root, respectively. Intuitively the node with the relative gap of .7 is much "harder" than the one with a relative gap of .1, and this harder problem might deserve the use of a more elaborate branching rule, like Rule 4, in an attempt to increase the bounds of its children as much as possible. This observation led us to devise branching strategies based on both gap and depth.

Note that when a child node corresponding to setting $X_{ij} = 1$ is created, an estimate of the lower bound for this node is available from its parent. This estimate is either of the form $z' = z + U_{ij}$ (Rule 2), or $z' = z^{ij}$ (Rule 4). We define the *relative gap* for a node to be

$$g = \frac{v - z'}{v - z_0},$$

where z' is the bound estimate inherited from the node's parent, v is the incumbent value, and z_0 is the lower bound at the root node. In a strategy based on depth and gap these two values are used together to determine which branching rule and parameters are applied at a given node. A typical example of such a strategy is illustrated in Table 2. Each rule has associated with it a minimum relative gap and a maximum depth; for a given node the rules are scanned from the top down (from the most computationally expensive to the least computationally expensive) until one is found whose gap tolerance is below the node's relative gap z', and whose depth cutoff is greater than or equal to the node's depth. The advantage of such a strategy compared to the simpler depth-based approach is that computational effort can be more precisely directed to nodes that most merit it. The main *disadvantage* is that explicit min-gap parameters must be chosen in addition to the max-depth parameters. Good values of these parameters are problem-

Table 3. Summary output from solution of nug25

	Branching Rule						Total	
Level	4a	4b	4c	2a	2b	2c	Nodes	Time
0	1	0	0	0	0	0	1	9
1	1	0	0	0	0	0	6	115
2	0.798	0.117	0.064	0.011	0	0.011	94	2374
3	0.310	0.236	0.263	0.125	0.039	0.028	1853	25410
4	0	0.137	0.338	0.291	0.150	0.084	33475	76038
5	0	0.016	0.170	0.354	0.270	0.189	409466	206104
6	0	0	0	0.323	0.362	0.315	2696219	74940
7	0	0	0	0.274	0.347	0.380	9118149	164240
8	0	0	0	0	0.551	0.449	14970699	187077
9	0	0	0	0	0	1	16800536	149740
10	0	0	0	0	0	1	14056814	101902
11	0	0	0	0	0	1	7782046	46741
12	0	0	0	0	0	1	3355923	16591
13	0	0	0	0	0	1	1217206	4938
14	0	0	0	0	0	1	389522	1313
15	0	0	0	0	0	1	111958	306
16	0	0	0	0	0	1	28709	63
17	0	0	0	0	0	1	6623	12
18	0	0	0	0	0	1	1497	2
19	0	0	0	0	0	1	345	0
20	0	0	0	0	0	1	85	0
21	0	0	0	0	0	1	10	0
22	0	0	0	0	0	1	2	0
Nodes	6.6E2	1.2E4	8.1E4	3.5E6	1.3E7	5.5E7	7.1E7	
Time	2.1E4	1.1E5	1.7E5	1.3E5	2.1E5	4.2E5		1.1E6

specific, and the performance of the algorithm can be relatively sensitive to the values used (in particular to the parameters for the last use of Rule 4).

In addition to the use of branching strategies based on depth and gap, we made a number of smaller modifications to the B&B code from [4] to improve its efficiency. The most significant of these was modifying the integer LAP solver from [28], used on the FW iterations, to run using floating-point data. The use of a floating-point solver eliminates the need for the scaling/round-down procedure described in [4, Section 2], and also improves the quality of the lower bounds. Although the LAP solver itself runs faster with integer data, we found that the overall performance of the B&B algorithm was improved using the floating-point solver, especially for larger problems.

When a problem is solved using a branching strategy based on gap and depth, the B&B code reports a statistical summary of how the different branching rules were used at different levels of the tree. An example of this output, using the branching strategy in Table 2 applied to the nug25 problem, is shown in Table 3. Each row reports the fraction of nodes on a given level where each branching rule was applied. The total number of nodes, and total CPU seconds, are also reported for each level and each branching rule. Note that although Rule 4 is used on an extremely small fraction of the nodes (about 0.1%), it accounts for approximately 27.4% of the total CPU time. On large problems we typically invest up to 40% of the total CPU time using Rule 4 in an attempt to minimize the size of the tree.

Additional information collected for each level includes the fraction of nodes fathomed, the fraction of possible children eliminated when branching, and the mean and standard deviation of the inherited relative gap. This information is extremely useful in determining how a given branching strategy might be modified to improve overall performance.

3. The branch-and-bound tree estimator

In this section we consider a procedure for estimating the performance of our B&B algorithm when applied to a particular QAP instance. The ability to make such estimates is important for two reasons. First, the branching strategies described in the previous section require the choice of several parameters, such as the gap and depth settings for the different branching rules. Good values of these settings are problem-specific, and to evaluate a choice of parameters we need to approximate the algorithm's performance without actually solving the problem. Second, we are ultimately interested in solving problems that are at the limit of what is computationally possible using our available resources. In such cases it is imperative that before attempting to solve a problem we have a reasonable estimate of the time that the solution process will require.

A well known procedure due to Knuth [31] can be used to estimate the performance of tree search algorithms, and in fact has previously been used to estimate the performance of B&B algorithms for QAP [5,9]. Let α denote a node in a tree T, rooted at α_0. Let $C(\alpha)$ be a "cost" associated with node α, and let $C(T) = \sum_{\alpha \in T} C(\alpha)$ be the total cost for the tree. For example, if $C(\alpha) = 1$ then $C(T)$ is the number of nodes in the tree. Alternatively if $C(\alpha)$ is the time required for a search algorithm to process node α then $C(T)$ is the total time to search the tree. Let $D(\alpha)$ denote the descendants (or children) of α as the tree is traversed from the root α_0. Knuth's estimation procedure is as follows:

procedure `EstimateCost`
$k = 0$, $d_0 = 1$, $C = 0$
For $k = 0, 1, 2, \ldots$
  $C = C + d_k C(\alpha_k)$
  $n_k = |D(\alpha_k)|$
  If $n_k = 0$ **Return** C
  Else choose α_{k+1} at random from $D(\alpha_k)$
    $d_{k+1} = d_k n_k$
Next k

Procedure `EstimateCost` makes a random "dive" in the tree, at each node choosing uniformly from the available children until a terminal node (leaf) is reached. The utility of the procedure stems from the following easily-proved result.

Theorem 1 (Knuth). *Let C be computed by procedure* `EstimateCost` *applied to a tree T. Then $E[C] = C(T)$.*

By Theorem 1 the quantity C is an unbiased estimate of $C(T)$. It follows that if the procedure is applied m times, resulting in estimates $C_1, \ldots, C_m$, then the sample

Table 4. Actual vs. estimated performance on nug20

	Actual		Estimated	
Level	nodes	time	nodes	time
0	1	3.20	1	3.24
1	6	43.18	6	43.22
2	97	219.89	97	217.84
3	1591	601.91	1598	612.46
4	18521	776.27	18763	863.05
5	102674	921.26	106975	944.13
6	222900	1208.09	245746	1459.19
7	221873	795.82	270000	924.53
8	124407	317.92	94878	287.22
9	47930	97.81	0	0.00
10	11721	20.85	0	0.00
11	2509	3.67	0	0.00
12	450	0.65	0	0.00
13	73	0.06	0	0.00
14	5	0.00	0	0.00
15	3	0.00	0	0.00
16	1	0.00	0	0.00
17	1	0.00	0	0.00
Total	754763	5010.58	738066	5354.89

mean $\bar{C} = (1/m)\sum_{i=1}^{m} C_i$ should be close to $C(T)$ if m is sufficiently large. Note that for B&B algorithms $|D(\alpha_k)|$ depends on the value of the upper bound, so procedure `EstimateCost` can most accurately estimate $C(T)$ when an accurate estimate of the optimal value is known. By making appropriate choices for the cost function $C(\cdot)$ we can obtain estimates for virtually all of the summary statistics associated with our B&B tree, described at the end of the previous section.

In Table 4 we illustrate the performance of the estimation procedure on the nug20 problem, using m = 10,000 dives. The table compares estimates for the number of nodes and time required on each level with the values obtained when the problem is actually solved using the B&B code. The accuracy of the estimator through level 8 is quite good, in accordance with the typical behavior described by [31]. Note that although the estimator fails to obtain values for levels $k \geq 9$, this omission is of little practical consequence since the actual node and time figures drop off very rapidly after level 8. The errors in the estimates for total time and nodes are -2.2% and +6.9%, respectively.

In our experience the performance of Knuth's estimator on small and medium-sized QAPs is excellent. In applying the procedure to larger ($n \geq 24$) problems, however, serious difficulties arise. The probability of a random dive reaching deep levels of the tree is very small, but the contribution of these deep levels to the overall nodes, and time, becomes substantial for larger problems. Assuming that $C(\cdot)$ is nonnegative, the sampling distribution of $\bar{C}$ becomes more and more skewed, with a long right tail. This situation was already predicted by Knuth, who states [31, p.129] "There is a clear danger that our estimates will almost always be low, except for rare occasions when they will be much too high."

We observe exactly this behavior when the estimation procedure is applied to larger QAPs. In Figure 1 we illustrate the performace of the estimation procedure (labeled

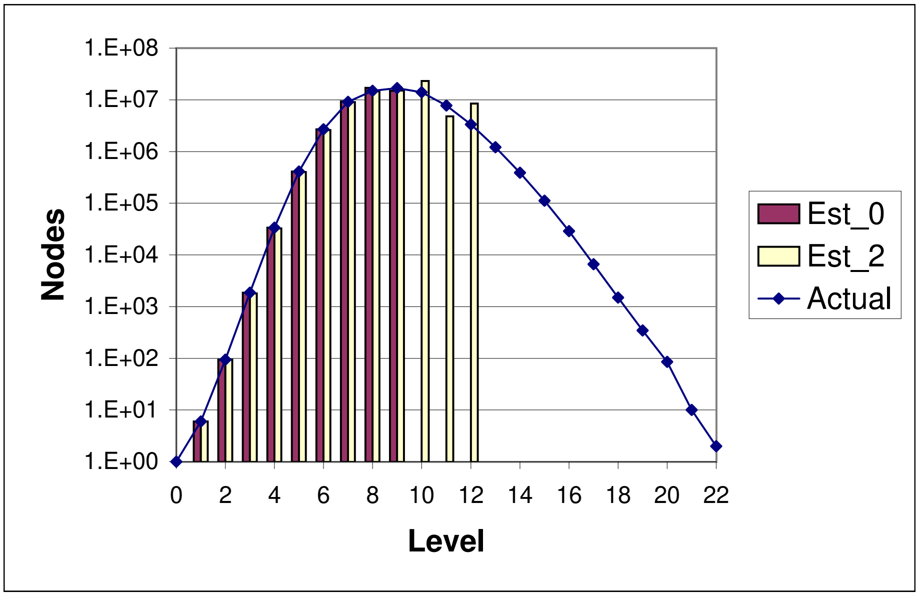

Fig. 1. Performance of estimator on nug25

Est_0) applied to the nug25 problem, using $m = 10{,}000$ dives. Both the estimate and the actual values are obtained using the parameters in Table 2. The estimator obtains no values for nodes at levels $k \geq 9$, but the nodes at these levels are a nontrivial fraction of the total (note the logarithmic scale for the number of nodes). The errors in the estimates for total nodes and time are -37% and -27%, respectively. These errors typically worsen as the dimension n increases, and for $n = 30$ the actual values for total nodes and time may be two or three times the estimated values. With errors of this magnitude the estimator becomes essentially useless in trying to decide on good values for algorithm parameters. It is also worth noting that some previous researchers have used Knuth's procedure to estimate the complexity of solving large QAPs, without being able to evaluate the accuracy of their estimates. As a result we believe that past work (for example [9]) has tended to *underestimate* the difficulty of large problems like nug30.

In addition to noting the potential problems with his estimator, Knuth [31] proposed a possible solution. By using non-uniform probabilities to choose the child at each node during a dive, `EstimateCost` can be induced to follow paths that contribute more to the actual total cost $C(T)$. For a node α_k at level k suppose that $D(\alpha_k) = \{\beta_1, \ldots, \beta_{n_k}\}$. Assign a probability $p(\beta_i)$ to each β_i, and choose α_{k+1} to equal β_i with probability $p(\beta_i)$. Let p_k be the probability corresponding to the child picked. Knuth shows that Theorem 1 remains true if the statement $d_{k+1} = d_k n_k$ in `EstimateCost` is replaced by $d_{k+1} = d_k/p_k$, so the procedure can be applied as before. Note that for uniform sampling $p_k = 1/n_k$, and the procedure reverts to the original version.

The use of non-uniform probabilities in `EstimateCost` is an example of *importance sampling*. In our context "important" paths are deep ones, so we need a method of generating probabilities for children that will bias the search towards deeper paths in

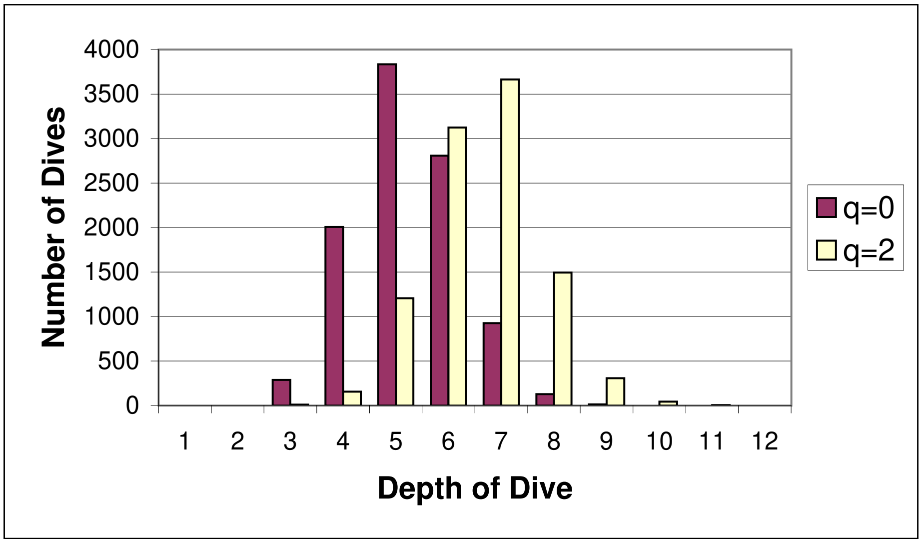

Fig. 2. Dive depths using estimator on nug25

the tree. Intuitively children that have larger gaps will require more branching levels before fathoming occurs, suggesting the use of probabilities that depend on the gaps. Let $g(\alpha)$ denote the inherited relative gap of a node α, as defined in the previous section. If $D(\alpha_k) = \{\beta_1, \ldots, \beta_{n_k}\}$, we define sampling probabilities

$$p(\beta_i) = \frac{g(\beta_i)^q}{\sum_{j=1}^{n_k} g(\beta_j)^q},$$

where $q \geq 0$. If $q = 0$ the sampling is uniform. Setting $q = 1$ makes the probability that a child is chosen proportional to its gap, and increasing q futher gives higher probabilities to children with larger gaps. For QAPs of dimension $24 \leq n \leq 32$ we have obtained good results using q between 1 and 2. As an example, the series labeled "Est_2" in Figure 1 illustrates the performance of the estimator on nug25, using $q = 2$ and $m =$ 10,000 dives. Note that estimates are now obtained through level 12, three levels deeper than when using $q = 0$. Although the errors in these estimates are greater than those at lower levels, the values obtained are of considerable use in estimating the overall performance of the algorithm. The errors in the estimates for total nodes and time using $q = 2$ are +10% and -1%, respectively. In Figure 2 we show the distribution of the dive depths (out of a total of 10,000) for the estimator applied to nug25, using $q = 0$ and $q = 2$. The effect of biased sampling in increasing the depth of the dives is clear. The average depth is increased from 5.25 for $q = 0$ to 6.62 for $q = 2$. For $q = 2$ four dives reached level 11, and one reached level 12.

When m trials of Knuth's estimator are used to estimate a total cost $C(T)$ with a sample mean $\bar{C}$, one can also compute a sample standard deviation, and sample standard error of the mean $S_{\bar{C}}$. However $S_{\bar{C}}$ may be a poor indicator of the accuracy of $\bar{C}$ as an estimate of $C(T)$ for large problems, because of the skew in the sampling distribution of $\bar{C}$. The use of importance sampling reduces the true standard error $\sigma_{\bar{C}}$, but more importantly reduces the skew of the distribution so that typical estimates are closer to the true value $C(T)$.

In addition to the use of importance sampling based on gap estimates we made one other small modification to the estimation procedure. Rather than begin the dives in `EstimateCost` at the root node, we first run the B&B algorithm in "breadth-first" mode to generate all nodes at a prescribed depth NBFS, and then initialize `EstimateCost` by sampling a node at level NBFS. This is useful for two reasons. First, there is no error in the node values through level NBFS (and time values through level NBFS-1), and the variance of estimates for the deeper values is reduced. Second, this strategy avoids wasteful duplication of the computations at low levels in the tree in the course of the dives. The latter point is particularly important because we use more expensive branching rules at the low levels, and the cost of repeating these computations thousands of times can be nontrivial. We typically choose NBFS to be the highest value so that the number of nodes is below the estimation sample size m; for QAPs of size $24 \leq n \leq 32$ and $m = 10{,}000$ this usually results in NBFS = 3.

Using our refined version of Knuth's estimator we were able to obtain estimates for the performance of our B&B code on a number of unsolved QAPs. For example, we determined that approximately 5-10 years of CPU time on a single HP9000 C3000 would be required to solve the nug30 problem. This estimate was superior to the projected time for any previously-known solution method for the QAP, but still indicated that powerful computational resources would be required to solve nug30 in a reasonable time. We decided that an implementation of our algorithm on a computational grid offered the best chance for obtaining these resources.

4. Computational grids

A *computational grid* or *metacomputer* is a collection of loosely-coupled, geographically distributed, heterogenous computing resources. Our focus is on the possibility of using this collection of resources—in particular, the idle time on such collections—as an inexpensive platform that can provide significant computing power over long time periods. A computational grid is similar to a power grid in that the provided resource is ubiquitous and grid users need not know the source of the provided resource. An introduction to computational grids is given by Foster and Kessleman [13].

Although computational grids are potentially very powerful and inexpensive, they have a number of features that may make them difficult to use productively. In particular, computational grids are:

- Dynamically available – Resources appear during the course of the computation;
- Unreliable – Resources disappear without notice;
- Weakly linked – Communication times between any given pair of processors are long and unpredictable;
- Heterogeneous – Resources vary in their operational characteristics (memory, processor speed, and operating system).

In all these respects, metacomputing platforms differ substantially from conventional multiprocessor platforms, such as the IBM-SP or SGI Origin machines, or dedicated clusters of PCs.

4.1. Grid computing toolkits

In order to harness the power of a computational grid, *resource management software* is required. Resource management software detects available processors, determines when processors leave the computation, matches job requests to available processors, and executes jobs on available machines. A number of different grid computing toolkits have been developed that perform these resource management functions [11,21,49]. Our implementation relies on two – Condor [37] and Globus [12].

Most of our efforts rely on the Condor system, which manages distributively-owned collections of workstations known as *Condor pools*. A unique and powerful feature of Condor is that each machine's owner specifies the conditions under which jobs are allowed to run. In particular, the default policy is to stop a Condor job when a workstation's owner begins using the machine. In this way, Condor jobs only use cycles that would have otherwise been wasted. Because of the minimal intrusion of the Condor system, workstation owners are often quite willing to donate their machines, and large Condor pools can be built.

Another powerful feature of Condor is known as *flocking*, whereby Condor pools in different locations are joined together, allowing jobs submitted to one Condor pool to run on resources in a different pool. The flocked Condor pools may be located anywhere on the Internet.

A drawback of the Condor system is that system software must be installed and special Condor programs (daemons) must be run in order for a machine to belong to a Condor pool. Some administrators—in particular administrators of large supercomputing platforms—are often unwilling to run Condor daemons. Resource management for these machines is usually done through a batch scheduling system, where dedicated reservations for processors are made and accounts are charged for their use. To include these resources in our computational grid managed by Condor, we use a bridge between Condor and the Globus software toolkit known as *Condor glide-in*. Using Condor glide-in, Globus authenticates users on the computing resources, makes processor requests to the batch scheduling system, and spawns the proper Condor daemons when the processor request is granted.

By using the tools of flocking and glide-in, we were able to build a federation of over 2500 CPUs distributed around the globe. Table 5 shows the location and number of processors in our computational grid, the architecture and operating system of the processors making up the grid, and the mechanism by which the processors were accessed. The processors are a mix of networks of workstations, dedicated clusters, and traditional supercomputing resources. The 1024 processor SGI/IRIX machine currently ranks as the 51st fastest supercomputer in the world; see `http://www.top500.org/`.

Table 5. A computational grid

# CPUs	Architecture/OS	Location	Access Method
246	Intel/Linux	Wisconsin	Main Condor Pool
146	Intel/Solaris	“ ”	“ ”
133	Sun/Solaris	“ ”	“ ”
414	Intel/Linux	Argonne	Glide-in
96	SGI/Irix	“ ”	“ ”
1024	SGI/Irix	NCSA	“ ”
16	Intel/Linux	“ ”	Flocking
45	SGI/Irix	“ ”	“ ”
190	Intel/Linux	Georgia Tech	“ ”
94	Intel/Solaris	“ ”	“ ”
54	Intel/Linux	Italy (INFN)	“ ”
25	Intel/Linux	New Mexico	“ ”
12	Sun/Solaris	Northwestern	“ ”
5	Intel/Linux	Columbia U.	“ ”
10	Sun/Solaris	“ ”	“ ”
2510			

4.2. The master-worker paradigm

To bring the large federation of resources in a computational grid together to tackle one large computing problem, we need a convenient way in which to break apart the computational task and distribute it among the various processors. The parallelization method that we employ is known as the *master-worker paradigm*. A master machine delegates tasks to worker machines, and the workers report the results of these tasks back to the master. Branch and bound algorithms fit perfectly into the master-worker paradigm. The master keeps track of unexplored nodes in the search tree and distributes them to the workers. The workers search their designated nodes of the tree and report unfathomed nodes back to the master. Many authors have used this centralized control mechanism for parallelizing B&B algorithms [16].

The master-worker paradigm is also perfectly suited to the dynamic and fault tolerant nature of the computational grid. As worker processors become available during the course of the computation they are assigned tasks. If a worker processor fails, the master reassigns its task to another worker. In alternative parallel processing structures where algorithm control information is distributed among the processors, complex mechanisms are needed to recover from the loss of this information when a processor fails [27].

To implement our parallel B&B algorithm for QAP, we use a software framework operationalizing the abstractions of the master-worker paradigm called MW. MW is a set of C++ abstract classes. In order to parallelize an application with MW, the application programmer reimplements three abstract base classes—`MWTask`, `MWDriver`, and `MWWorker`, that define the computing task and the actions that the master and worker processors take on receiving a task or the result of a task. See [18,20] for a more complete description of MW. Several grid-enabled numerical optimization solvers have been built with MW [8,19,35].

MW includes an abstract interface to resource management software (such as Condor) and will automatically (re)assign tasks when processors leave or join the computation. MW also has an interface allowing different communication mechanisms. Currently, communication between master and worker can be done via PVM [15], or using Condor's remote system call functionality [36] to write into a series of shared files. The parallel QAP solver employs the shared file communication mechanism.

Because MW reschedules tasks when the processors running these tasks fail, applications running on top of MW are fault tolerant in the presence of all processor failures except for that of the master processor. In order to make computations fully reliable, MW offers features to periodically checkpoint (or save to disk) the state of the computation on the master process. MW can then resume the computation from the checkpointed state if the master process fails. In a computation involving many days of computation on a large number of machines the checkpointing of the master process is an important reliability feature.

The heterogeneous and dynamic nature of a computational grid makes application performance difficult to assess. Standard performance measures such as wall clock time and cumulative CPU time do not separate application code performance from computing platform performance. By normalizing the CPU time spent on a given task with the performance of the corresponding worker, MW aggregates time statistics that are comparable between runs. The user can register an application-specific benchmark task that is sent to all workers that join the computational pool. For the parallel QAP solver, the benchmark task is a small, specific portion of the branch and bound tree to evaluate. The CPU times from a parallel run can then be normalized to an equivalent time on any reference machine by simply running the benchmark task on that machine. In addition to the normalized CPU time statistic $\mathcal{T}$, MW collects a number of other statistics such as the wall clock time $\mathcal{W}$, the amount of time U_i worker i was available, and the amount of CPU time t_j spent completing task j. At the end of the run, MW reports useful statistics such as the average number of available workers during the course of the run $\mathcal{N}$,

$$\mathcal{N} \equiv \frac{\sum_i U(i)}{\mathcal{W}},$$

and the parallel efficiency of the process η,

$$\eta \equiv \frac{\sum_j t(j)}{\sum_i U(i)}.$$

In developing the parallel QAP solver, these statistics have proven to be quite valuable to assess both the sequential and parallel aspects of the application code performance.

5. The grid-enabled QAP solver

Our goal is to develop a parallel QAP solver that can efficiently harness the available resources of a computational grid. While parallelizing the B&B algorithm of Section 2 is in principle quite simple, we have carefully considered many design choices to best match a parallel B&B algorithm with the charactersitics of our particular computational platform. In this section we explain these design choices and show through a suite of experiments that they lead to an efficient parallel algorithm running on a computational grid.

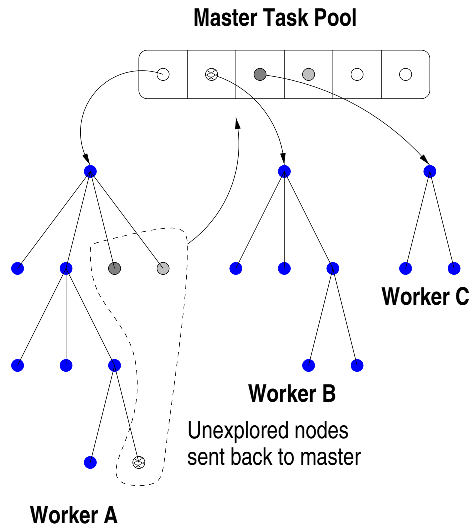

Fig. 3. Parallel depth-first oriented search strategy

5.1. Grid computing considerations

As explained in Section 4.2, the master-worker paradigm is well suited to the dynamic nature of a computational grid. However, given the large number of resources that may be available on a computational grid, we must ensure that the master-worker paradigm scales effectively to this size. Algorithm features must be exploited to avoid overwhelming the master with requests for new tasks, which would reduce the overall efficiency.

To achieve a high parallel efficiency η in a computational grid context, our parallel algorithm should strive to:

(I) keep the arrival rate of worker requests small;
(II) keep the service rate of the master machine large;
(III) avoid large messages;
(IV) achieve load balance.

Items (I) and (II) will minimize master contention, item (III) will reduce the impact of the computational grid's poor and unpredictable communication properties, and item (IV) is a concern of all parallel algorithms.

Employing a *parallel depth-first oriented* search strategy helps us to attain many of these goals. The strategy works as follows. An available worker is given the deepest active node in the master's list. The worker evaluates the subtree rooted at this node in a depth-first fashion. If after $t_{\max}$ seconds the worker has not completely evaluated the subtree, its remaining active nodes are passed back to the master. Figure 3 depicts the parallel depth-first oriented search strategy. In the figure worker A has returned 3 nodes to the master, 2 of which have subsequently been processed by workers B and C.

The arrival rate of worker requests is kept small (Goal I) by having workers compute for $t_{\max}$ seconds before reporting their results. The periodic reporting of unfinished tasks back to the master helps to balance the load among processors (Goal IV) and also reduces the amount of work that is lost if a worker processor fails, which increases the parallel efficiency η. In order to keep the service rate of the master large (Goal II), the list of nodes it manages should be kept small. This is accomplished by two aspects of the search strategy. First, by sending workers the deepest nodes, they are less likely to report back many new tasks. Second, by having workers search in a depth-first fashion, the number of active nodes that the worker reports back to the master is minimized. Performing depth-first search on the workers also helps to keep the size of the messages small (Goal III).

It is important to note that in the general B&B context this depth-first oriented strategy could result in a parallel algorithm exploring many more nodes than its sequential counterpart [33]. In our case, however, this undesirable effect is eliminated by the fact that a very good (in many cases optimal) solution to each QAP instance that we are attempting to solve is known. If such a solution is not known, the algorithm can be adapted in a straightforward manner to quickly find good feasible solutions.

5.2. Tuning the parallel QAP solver

In order to test and improve our grid-enabled parallel QAP solver `MWQAP`, a suite of computational experiments using the nug25 QAP were undertaken. The nug25 instance is itself very challenging, being only a short time ago the limit of what was computationally practical [23]. However, given the efficient algorithm and powerful computational platform at our disposal, we were able to use the nug25 instance to tune `MWQAP` for the solution of even more difficult problems. The computational resources used in our suite of experiments were the machines running the LINUX operating system in Table 5. To reduce the random effects of the changing nature of the computing platform, each experiment consisted of solving nug25 seven times.

With a maximum worker CPU time of $t_{\max} = 100$ seconds, the initial `MWQAP` implementation achieved an average parallel efficiency of $\eta = 41.8\%$ in solving nug25. Since there are no large synchronous portions in the branch and bound algorithm, this efficiency is surprisingly low. Examining the logs from these runs, we discovered that there were a large number of tasks that workers completed very quickly; in other words many j for which $t(j)$ was small (see Section 4.2). The distribution of the tasks times for a typical run of nug25 using this initial implementation is shown as the "Initial" series in Figure 4. The large number of short tasks leads to contention at the master processor. As a result workers sit idle waiting for the master to respond, reducing the overall efficiency. In addition communication time is large relative to the time required to complete many tasks, again reducing parallel efficiency.

To improve the initial parallelization strategy in `MWQAP`, the number of fast worker tasks must be reduced. Fast tasks come from workers returning "easy" nodes of the B&B tree. An effective parallel search strategy should aim to ensure that workers evaluate as many easy nodes of their given subtree as possible. The first modification to `MWQAP` was to re-order the child nodes by difficulty, based on relative gap (see Section 2) before adding them to the worker's queue of unprocessed nodes. By altering the search order to investigate easy nodes first, the average parallel efficiency achieved

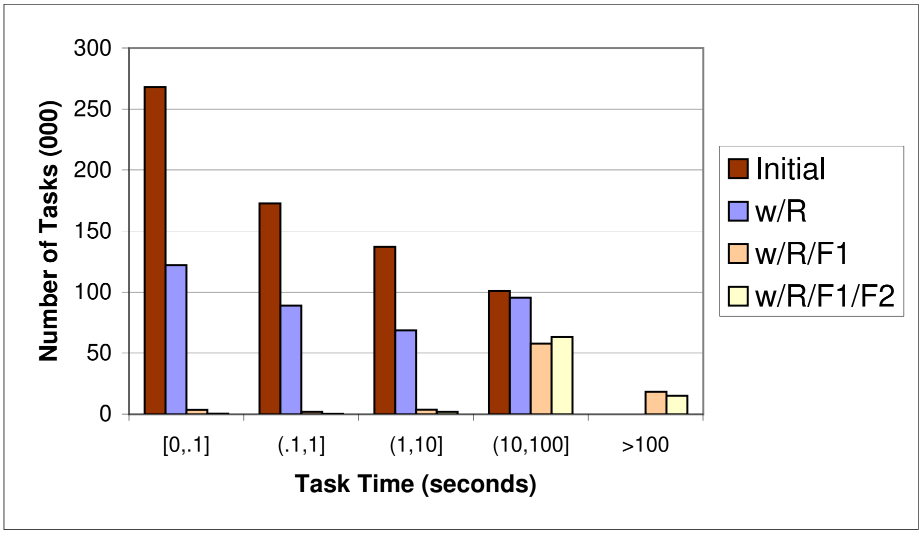

Fig. 4. Distribution of task times for `MWQAP` on nug25

when solving the nug25 instance seven times increased to $\eta = 66.1\%$. The distribution of the task computing times, labeled "w/R" in Figure 4, was also considerably improved.

Not surprisingly, most fast computing tasks come from nodes that are deep in the B&B tree. Figure 5 shows the probability that a worker requires less than one CPU second as a function of the depth of the initial node from the nug25 tree it is given. Note the large increase in this probability as the depth goes from 5 to 6. To reduce the number of fast tasks further, the parallel strategy in `MWQAP` was modified to allow the workers a *finish-up* phase. Workers are allowed an additional $2t_{\max}$ seconds to process all nodes (still via depth-first search) deeper than $d_{\max} = 5$ whose relative gap is also less than $g_{\min} = 0.28$. (Optimal values for the parameters $d_{\max}$ and $g_{\min}$ are obviously problem-specific. However reasonable values can easily be obtained from the output of the estimator described in Section 3.) Using both the node re-ordering and the finish-up strategies, the average efficiency of `MWQAP` on the nug25 instance increased to $\eta = 82.1\%$. Figure 4 shows the resulting distribution of task execution times, labeled "w/R/F1." Note that because $t_{\max} = 100$, task times greater than 100 seconds correspond to the use of the finish-up strategy. Although the finish-up strategy is used on a relatively modest fraction of the tasks, it results in a very large reduction in the number of fast tasks.

In an attempt to reduce the fast tasks even further, the finish-up strategy was refined to allow for an additional period of time $2t_{\max}$, where the parameter $d_{\max}$ was increased to 6, and $g_{\min}$ was relaxed to 1.0. With the additional finish-up period the average parallel efficiency of the parallel QAP solver on nug25 increased to $\eta = 84.8\%$. Figure 4 shows the resulting task time distribution, labeled "w/R/F1/F2." The second finish-up period is used on a very low fraction of tasks ($< 1\%$), but results in almost complete elimination of tasks requiring less than 10 seconds. The simple strategies of intelligently

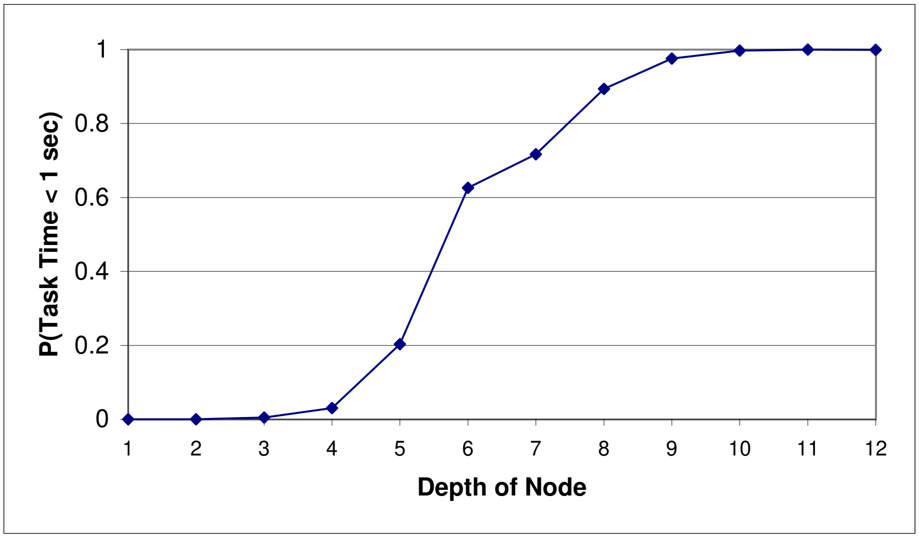

Fig. 5. Likelihood of fast tasks for nug25

ordering the search, and allowing finish-up periods for workers to eliminate fast tasks remaining in their pool of unexplored nodes, virtually elimated contention and more than doubled the parallel efficiency.

The fast-task reduction techniques are designed to eliminate contention at the master, but this is not sufficient to ensure maximum efficiency. The dependency between tasks in the B&B algorithm can ultimately lead to situations where workers are sitting idle waiting for other workers to report back their results. In the case of our QAP solver, this occurs when the master pool is near-empty (having less tasks than participating workers). In order to keep the master task populated with nodes for workers to process, we made two other modifications to the search strategy in `MWQAP`. These are:

1. Reduce the grainsize (by decreasing $t_{\max}$) when there are less than s_1 nodes in the master task pool.
2. Re-order the master task pool to force workers to explore difficult subtrees, rather than deep ones, when there are less than s_2 nodes in the master task pool.

When there are less than $s_1 = 100$ nodes in the master's task list, $t_{\max}$ is reduced to 10 seconds. This increases the utilization of resources at the beginning and the end of the search. When there are less than $s_2 = 3000$ nodes remaining in the master's task list, workers are given subtrees to explore whose root nodes have the highest relative gaps. This ensures that the task list does not become empty until the search is complete. A positive side-effect of this *lazy best-first* search is that the optimal solution is often found earlier in the solution procedure, somewhat reducing the total number of nodes that must be explored. Using the fast task elimination strategies and the worker idle time reduction strategies, the efficency of the parallel QAP solver was increased to $\eta = 85.6\%$.

Table 6. Performance statistics for `MWQAP` on nug25

	Initial	w/R	w/R/F1	w/R/F1/F2	Final
N	74,677,341	71,486,576	72,059,881	71,893,497	71,770,751
	(776,194)	(1,468,152)	(476,303)	(213,131)	(234,248)
$\mathcal{N}$	213	211	172	190	185
	(12)	(4)	(29)	(10)	(17)
$\mathcal{W}$	8675	5561	4948	4356	4757
	(1820)	(309)	(1521)	(312)	(323)
$\mathcal{T}$	205,359	198,717	198,130	195,621	196,523
	(2355)	(3656)	(2795)	(1125)	(1185)
η	41.8%	66.1%	82.1%	84.8%	85.6%
	(6.79%)	(3.68%)	(4.63%)	(4.30%)	(2.11%)

Performance improvements in solving the nug25 instance are summarized in Table 6. For each version of the code, we list the average and standard deviation of each of the following statistics over seven solution runs: number of nodes N, average number of machines $\mathcal{N}$, wall clock time $\mathcal{W}$, normalized CPU time $\mathcal{T}$, and parallel efficiency η. (Note: The normalized time must be multiplied by the time to perform the benchmark task to obtain an equivalent time on a given machine. For the HP9000 C3000 the time to perform the benchmark task is 3.64 seconds, so the equivalent time to solve nug25 on a single C3000 machine is about 7.15E5 seconds, or 8.3 days.) Besides the large improvement in parallel efficiency achieved between the initial and final `MWQAP` implementations, a few other items are of note. First, the variance in nodes N arises entirely from when the optimal solution is obtained. Second, despite the changing nature of the computational platform and efficiency of the underlying parallel implementation, the normalized CPU time statistics exhibit relatively low variance. This variance is particularly low for last two versions. Third, adding the idle time reduction strategies in the final version significantly reduces the variance in the parallel performace statistics.

It is worth noting that for the final version of `MWQAP` the average wall time to solve nug25, using about 200 processors, is approximately 80 minutes. At this point, `MWQAP` exhibits predictably scalable performance and is ready to efficiently provide the CPU time required to tackle unsolved QAP instances.

6. Computational results on large problems

In the previous sections we explained the design of a powerful sequential branch-and-bound code for the QAP, and how to implement it efficiently to harness the power of large computational grids. The resulting code has been used to solve instances of the QAP unsolved for decades, the most famous among them being the nug30 problem.

In 1968, Nugent, Vollman, and Ruml [41] posed a set of quadratic assignment problem instances of sizes 5, 6, 7, 8, 12, 15, 20 and 30. The distance matrices for these problems correspond to Manhattan distances on rectangular grids. Additional instances have been introduced over the years by removing facilities from larger instances, and either removing selected locations or using the distance matrix from an appropriately-sized grid. The Nugent problems are the most-solved set of QAPs, and the solution of

Table 7. Solution statistics for large QAPs

Problem	Nodes	Time (Days)	C3000 Years	Workers Ave.	Workers Max	Pool Factor	Parall. Eff. η
nug27	4.02E8	.69	.18	185	275	96	.91
nug28	2.23E9	3.73	.88	224	528	86	.90
nug30	1.19E10	6.92	6.94	653	1009	366	.92
kra30b	5.14E9	3.79	2.67	462	780	257	.92
kra32	1.67E10	12.26	10.35	576	1079	308	.87
tho30	3.43E10	17.18	17.48	661	1307	371	.89

Table 8. Optimal solutions for large QAPs

Problem	Value	Optimal Solution
nug27	5234	23,18,3,1,27,17,5,12,7,15,4,26,8,19,20,2,24,21,14,10, 9,13,22,25,6,16,11
nug28	5166	18,21,9,1,28,20,11,3,13,12,10,19,14,22,15,2,25,16,4,23, 7,17,24,26,5,27,8,6
nug30	6124	14,5,28,24,1,3,16,15,10,9,21,2,4,29,25,22,13,26,17,30, 6,20,19,8,18,7,27,12,11,23
kra30b	91420	23,26,19,25,20,22,11,8,9,14,27,30,12,6,28,24,21,18,1,7, 10,29,13,5,2,17,3,15,4,16
kra32	88900	31,23,18,21,22,19,10,11,15,9,30,29,14,12,17,26,27,28,1,7, 6,25,5,3,8,24,32,13,2,20,4,16
tho30	149936	8,6,20,17,19,12,29,15,1,2,30,11,13,28,23,27,16,22,10,21, 25,24,26,18,3,14,7,5,9,4

the various instances have marked advances in both processor capability and QAP solution methods. See [23] for an excellent history of these problems. Most results reported for Nugent problems of size $n \leq 24$ have been based on the GLB; see for example [5, 10, 39]. Prior to the work reported here the largest Nugent instance solved to optimality was the nug25 problem. Nug25 was first solved using the dynamic programming lower bounding approach of Marzetta and Brüngger [38], and was subsequently solved more efficiently using the dual-LP approach of Hahn et al. [25].

In Table 7 we report solution statistics for a number of previously-unsolved large QAPs. In Table 8 we give optimal permutations (assignments of facilities to locations) for these same problems. The problems are listed in the order in which they were solved. For all problems considered here the optimal objective value is known to be an even integer, and the initial upper bound (incumbent value) was set equal to BKV+2, where BKV is the best known objective value. With this choice of the initial upper bound the algorithm must recover an optimal solution of the problem, in addition to proving optimality.

The “Time” statistic in Table 7 is the total wall time that the master process was running; time during which the master was shut down is not counted. “C3000 Years” is the total CPU time, normalized to time on a single HP9000 C3000. The pool factor is the equivalent number of such machines that would have been required to complete the job in the given wall time. In each case the B&B algorithm was applied the branching strategy from Table 2, with settings of the gap and depth parameters chosen for the particular problem. Several runs of the estimator described in Section 3 were made for each problem in an attempt to obtain good parameter choices. The exact parameters used for the different problems, and complete solution statistics, are available on request. For all of these problems the estimated solution time was within 15% of the actual time, normalized to the C3000.

The first two problems, nug27 and nug28, were created from nug30 by removing the last 3 (respectively 2) facilities from nug30, and using a distance matrix corresponding to a 3 by 9 (respectively 4 by 7) grid. Since these were new problems we ran several heuristics, including GRASP [44] and simulated annealing to obtain a good feasible solution. For both problems the simulated annealing code of Taillard (available from `http://sunst50.einev.ch/prive/pro/taillard`) produced the best solution, and this was used as the BKV to initialize the B&B procedure. In both cases this BKV was proved optimal.

Following the solution of nug28 the large computational grid described in Table 5 was assembled for the solution of the nug30 problem. The nug30 computation was started on June 8, 2000 at 11:05, using a master machine located in the Condor pool at the University of Wisconsin-Madison. The computation was initialized using the BKV from QAPLIB [7], which was ultimately proved optimal. The computation completed on June 15, at 21:20. In the interim the process was halted five times, twice due to failures of the resource management software and three times to perform maintenance. Following each interuption the process was resumed using the checkpointing feature of MW described in Section 4.2. The progress of the computation was viewed via the Internet as it happened using the iMW-QAP environment described in [17].

In Figure 6 we show the number of worker machines over the course of the nug30 run. As described in Table 7 there were an average of about 650 workers, with a peak of over 1000. The five interuptions in the solution process are clearly visible. Figure 7 shows the evolution of the size of the master queue during the course of the nug30 computation. The effect of the “lazy best-first” strategy described in Section 5.2 is clearly visible; the queue size trends downward until the master has 3000 tasks on hand, at which point the pool is re-sorted to give out more difficult tasks first. This immediately causes the pool to be re-populated with unprocessed nodes, and the cycle repeats. One remarkable statistic for the nug30 run is that on average approximately one million linear assignment problems were solved *per second* during the one-week period.

The problem kra30b arose from a hospital planning application in 1972 [32]. The related kra30a problem was first solved by Hahn et al. [25]. The flow matrices for these problems are identical, and the distance matrices are based on 3-dimensional rectangular grids with unit costs of 50, 50 and 115 for the x, y, and z directions, respectively. The grid for kra30a is 4 by 4 by 2, with 2 points on opposite corners of one level removed, while kra30b uses a 5 by 3 by 2 grid. See [26] for an interesting discussion of these problems. In the solution of kra30b we again used the BKV from QAPLIB, but divided the distance matrix by 5 before solving the problem. The BKV was proved optimal. The fact that kra30b was considerably easier to solve than nug30 (see Table 7) is not surprising given that kra30b has an 8-fold symmetry, compared to the 4-fold symmetry of nug30. (Recall that the B&B algorithm, as described in Section 2, fully exploits any symmetry to reduce the number of children created when branching.) The problem kra32 was formed by using the distance matrix for the complete 4 by 4 by 2 grid (resulting in a 16-fold symmetry), and adding 2 dummy facilities. The problem

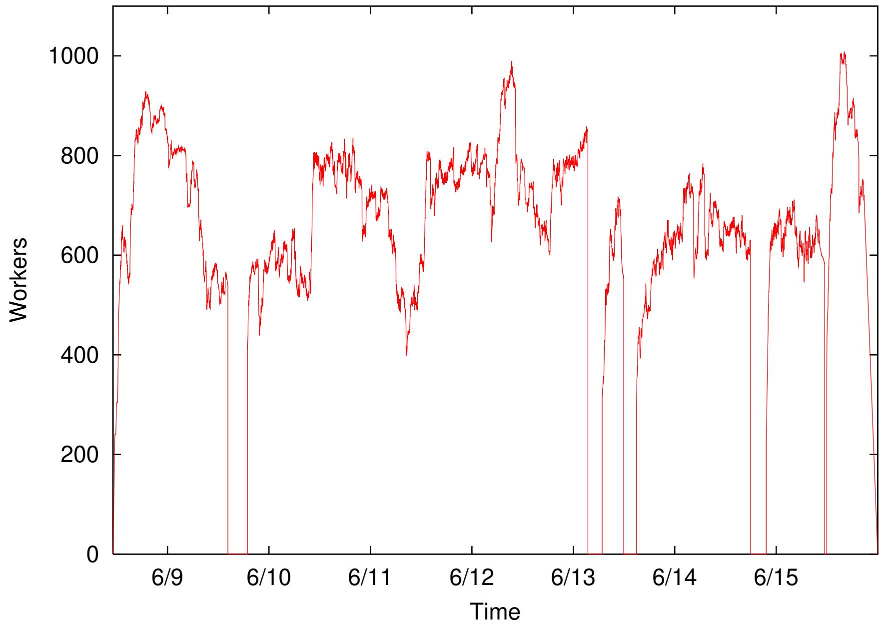

Fig. 6. Number of workers in nug30 computation

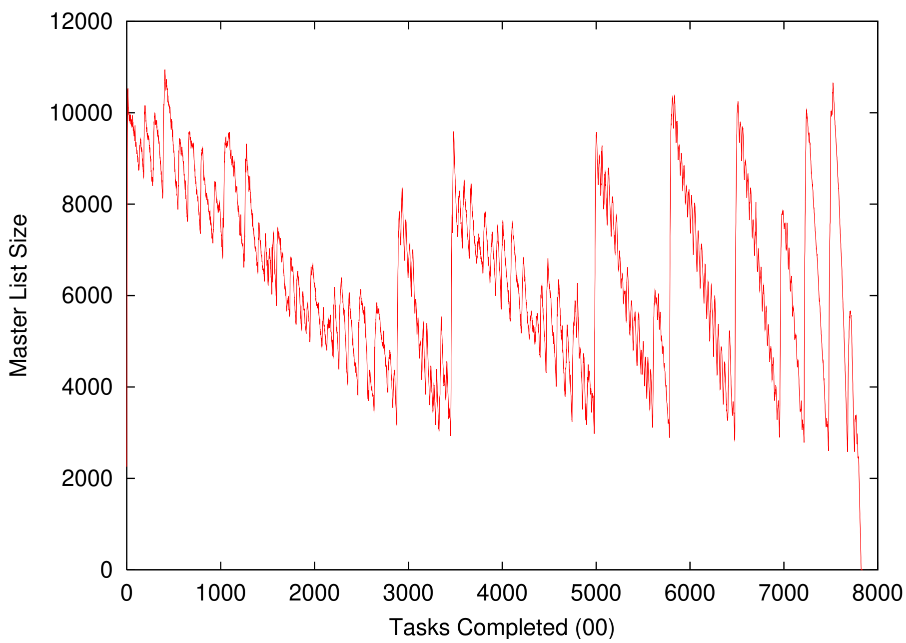

Fig. 7. Nodes in master queue during nug30 computation

was initialized with the BKV from kra30a, with the distance matrix again divided by 5. The BKV was proved optimal, showing for the first time that the choice of grid points removed to form the kra30a problem is in fact optimal.

The tho30 problem, posed in [48], uses a distance matrix arising from a 3 by 10 grid. From Table 7 it is clear that this problem was substantially more difficult to solve than the nug30 problem. It is worth noting that the root gap between the BKV and QPB was approximately 17.1% for tho30, versus 12.4% for nug30 (see [1, Table 1]). For both problems the best known root bound is obtained using the "triangle decomposition bound" of [30]; the corresponding gaps are 9% for tho30 and 5.75% for nug30 [7].

With the solutions of the nug30, kra30b, and tho30 problems, all symmetric QAPs of size $n \leq 30$ in QAPLIB [7], with the exception of the tai25a and tai30a problems, have now been solved. The "taixxa" problems, from [47] have dense, randomly generated flow and distance matricies, and as a result have virtually no structure of any kind. Based on runs of the estimator described in Section 3, the rate with which QPB increases on these problems as branching occurs is not rapid enough for their solution using `MWQAP` to be practical, even with our large computational grid.

Larger unsolved QAPLIB problems include the esc32a/b/c/d/h and ste36a/b/c problems. The esc32x problems are QAP representations of "semi-assignment" problems, and as a result have very sparse flow matrices with large blocks of zeros. (The problems esc32e/f/g, solved in [5], have flow matrices that are almost identically zero.) The best known bounds for these problems have been obtained using polyhedral methods that explicitly use the semi-assignment structure [29]. The ste36a/b/c problems are similar to other grid-based problems such as nug30 and tho30, and arose from a "backboard wiring" application dating back to 1961 [46]. These problems present an outstanding open challenge in computational optimization which further advances could soon bring within reach.

7. Conclusions

Three advances were vital in bringing about the solutions of the large QAP instances described herein: the development of a new lower bounding technique, an intelligently engineered branch and bound algorithm using this lower bounding technique, and an efficient parallel implementation of the branch and bound algorithm on a large computational grid. In our opinion the potential of future algorithmic advances for the QAP and other difficult optimization problems can be realized by utilizing the power that computational grids have to offer. To date this power has been largely untapped. We hope this demonstration of the synergetic effects of combining improvements in both algorithms and computing platforms will inspire other optimization researchers to more fully exploit the power of the computational grid.

Acknowledgements. We are foremost very grateful to Steve Wright of Argonne National Lab and Miron Livny of the University of Wisconsin for their support of this research, under the auspices of the metaNEOS project. This large computing effort required the support of many institutions. In particular, we would like to acknowledge the contributions of the entire Condor and Globus teams. We would also like to acknowledge the National Computational Science Alliance under grant number MCA00N015N for providing resources at the University of Wisconsin, the NCSA SGI/CRAY Origin2000, and the University of New Mexico/Albuquerque

High Performance Computing Center AltaCluster; the IHPCL at Georgia Tech, supported by a grant from Intel; and the Italian Istituto Nazionale di Fisica Nucleare (INFN), Columbia University, and Northwestern University for allowing us access to their Condor pools.

References

1. K.M. Anstreicher and N.W. Brixius. A new bound for the quadratic assignment problem based on convex quadratic programming. Technical report, Dept. of Management Sciences, University of Iowa, 1999.
2. K.M Anstreicher and H. Wolkowicz. On Lagrangian relaxation of quadratic matrix constraints. Research report CORR 98-24, Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Ontario, Canada, 1998.
3. N.W. Brixius. Solving large-scale quadratic assignment problems. Technical report, Department of Computer Science, University of Iowa, 2000. Ph.D. Thesis.
4. N.W. Brixius and K.M. Anstreicher. Solving quadratic assignment problems using convex quadratic programming relaxations. Technical report, Dept. of Management Sciences, University of Iowa, 2000.
5. A. Brüngger, A. Marzetta, J. Clausen, and M. Perregaard. Solving large-scale QAP problems in parallel with the search library ZRAM. *Journal of Parallel and Distributed Computing*, 50:157–169, 1998.
6. R.E. Burkhard, E. Çela, P.M. Pardalos, and L.S. Pitsoulis. The quadratic assignment problem. In D.-Z. Du and P.M Pardalos, editors, *Handbook of Combinatorial Optimization*, volume 3, pages 241–337. Kluwer, 1998.
7. R.E. Burkhard, S.E. Karisch, and F. Rendl. QAPLIB – a quadratic assignment problem library. *Journal of Global Optimization*, 10:391–403, 1997.
8. 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*, 2000. To appear.
9. J. Clausen, S.E. Karisch, M. Perregaard, and F. Rendl. On the applicability of lower bounds for solving rectilinear quadratic assignment problems in parallel. *Computational Optimization and Applications*, 10:127–147, 1998.
10. J. Clausen and M. Perregaard. Solving large quadratic assignment problems in parallel. *Computational Optimization and Applications*, 8:111–127, 1997.
11. G. Fagg, K. Moore, and J. Dongarra. Scalable networked information processing environment (SNIPE). *International Journal on Future Generation Computer Systems*, 15:595–605, 1999.
12. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl. J. Supercomputer Applications*, 11:115–128, 1997. Available as `ftp://ftp.globus.org/pub/globus/papers/globus.ps.gz`.
13. I. Foster and C. Kesselman. Computational grids. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998. Chapter 2.
14. L.M. Gambardella, É.D. Taillard, and M. Dorigo. Ant colonies for the quadratic assignment problem. *Journal of the Operational Research Society*, 50:167–176, 1999.
15. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. The MIT Press, Cambridge, MA, 1994.
16. B. Gendron and T.G. Crainic. Parallel branch and bound algorithms: Survey and synthesis. *Operations Research*, 42:1042–1066, 1994.
17. M. Good and J.-P. Goux. iMW : A web-based problem solving environment for grid computing applications. Technical report, Department of Electrical and Computer Engineering, Northwestern University, 2000.
18. J.-P Goux, S. Kulkarni, J. Linderoth, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing*, pages 43–50, Los Alamitos, CA, 2000. IEEE Computer Society.
19. J.-P. Goux, S. Leyffer, and J. Nocedal. Solving mixed-integer nonlinear programming problems on metacomputing platforms. Technical report, Dept. of Electrical and Computer Engineering, Northwestern University, 1999. Working paper.
20. J.-P. Goux, J.T. Linderoth, and M.E. Yoder. Metacomputing and the master-worker paradigm. Preprint ANL/MCS-P792-0200, Mathematics and Computer Science Division, Argonne National Laboratory, 2000.
21. A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Legion: An operating system for wide-area computing. Available as `http://legion.virginia.edu/papers/CS-99-12.ps.Z`, 1999.
22. S.W. Hadley, F. Rendl, and H. Wolkowicz. A new lower bound via projection for the quadratic assignment problem. *Mathematics of Operations Research*, 17(3):727–739, 1992.

23. P.M. Hahn. Progress in solving the Nugent instances of the quadratic assignment problem. Working Paper, Systems Engineering, University of Pennsylvania, 2000.
24. P.M. Hahn, T. Grant, and N. Hall. A branch-and-bound algorithm for the quadratic assignment problem based on the Hungarian method. *European Journal of Operational Research*, 108:629–640, 1998.
25. P.M. Hahn, W.L. Hightower, T.A. Johnson, M. Guignard-Spielberg, and C. Roucairol. Tree elaboration strategies in branch and bound algorithms for solving the quadratic assignment problem. Technical report, Systems Engineering, University of Pennsylvania, 1999.
26. P.M. Hahn and J. Krarup. A hospital facility layout problem finally solved. Working paper, Systems Engineering, University of Pennsylvania, 2000.
27. A. Iamnitchi and I. Foster. A problem-specific fault-tolerance mechanism for asynchronous, distributed systems. In *Proceedings of the International Conference on Parallel Processing 2000*, August 2000.
28. R. Jonker and A. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, pages 325–340, 1987.
29. V. Kaibel. Polyhedral methods for the QAP. In P.M. Pardalos and L. Pitsoulis, editors, *Nonlinear Assignment Problems*, pages 1–34. Kluwer Academic Publishers, 1999.
30. S.E. Karisch and F. Rendl. Lower bounds for the quadratic assignment problem via triangle decompositions. *Mathematical Programming*, 71:137–152, 1995.
31. D.E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29:121–136, 1975.
32. J. Krarup and P.M. Pruzan. Computer-aided layout design. *Mathematical Programming Study*, 9:75–94, 1978.
33. T.H. Lai and S. Sahni. Anomalies in parallel branch and bound algorithms. In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 183–190, 1983.
34. J.T. Linderoth and M.W.P. Savelsbergh. A computational study of branch and bound search strategies for mixed integer programming. *INFORMS Journal on Computing*, 11:173–187, 1999.
35. J.T. Linderoth and S.J. Wright. Implementing decomposition algorithms for stochastic programming on a computational grid. Technical report, MCS Division, Argonne National Lab, 2000. Working Paper.
36. M. Litzkow. Remote Unix – Turning idle workstations into cycle servers. In *Proceedings of Usenix Summer Conference*, 1987.
37. M. Livny, J. Basney, R. Raman, and T. Tannenbaum. Mechanisms for high throughput computing. *SPEEDUP*, 11, 1997. Available from `http://www.cs.wisc.edu/condor/doc/htc_mech.ps`.
38. A. Marzetta and A. Brüngger. A dynamic-programming bound for the quadratic assignment problem. In *Computing and Combinatorics: 5th Annual International Conference, COCOON'99*, volume 1627 of *Lecture Notes in Computer Science*, pages 339–348. Springer, 1999.
39. T. Mautor and C. Roucairol. A new exact algorithm for the solution of quadratic assignment problems. *Discrete Applied Mathematics*, 55:281–293, 1994.
40. G.G.L Meyer. Accelerated Frank-Wolfe algorithms. *SIAM Journal on Control*, 12:655–663, 1974.
41. C.E. Nugent, T.E. Vollman, and J. Ruml. An experimental comparison of techniques for the assignment of facilities to locations. *Operations Research*, 16:150–173, 1968.
42. M.W. Padberg and M.P. Rijal. *Location, Scheduling, Design and Integer Programming*. Kluwer, 1996.
43. P.M. Pardalos, F. Rendl, and H. Wolkowicz. The quadratic assignment problem: A survey and recent developments. In P.M. Pardalos and H. Wolkowicz, editors, *Quadratic assignment and related problems*, volume 16 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–42. AMS, 1994.
44. M.G.C. Resende, P.M. Pardalos, and Y. Li. Algorithm 754: Fortran subroutines for approximate solution of dense quadratic assignment problems using GRASP. *ACM Transactions on Mathematical Software*, 22:104–118, 1996.
45. M.G.C. Resende, K.G. Ramakrishnan, and Z. Drezner. Computing lower bounds for the quadratic assignment problem with an interior point algorithm for linear programming. *Operations Research*, 43:781–791, 1995.
46. L. Steinberg. The backboard wiring problem: A placement algorithm. *SIAM Review*, 3:37–50, 1961.
47. É.D. Taillard. Robust taboo search for the quadratic assignment problem. *Parallel Computing*, 17:443–455, 1991.
48. U.W. Thonemann and A. Bölte. An improved simulating annealing algorithm for the quadratic assignment problem. Working Paper, School of Business, Dept. of Production and Operations Research, University of Paderborn, Germany, 1994.
49. M. van Steen, P. Homburg, and A. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, 7:70–78, 1999.
50. Q. Zhao, S.E. Karisch, F. Rendl, and H. Wolkowicz. Semidefinite programming relaxations for the quadratic assignment problem. CORR Report 95-27, University of Waterloo, February 1998.