Memory-Aware Parallelized RLT3 for solving Quadratic Assignment Problems

P. Hahn[*] A. Roth[†] M. Saltzman[‡] M. Guignard[§]

hahn@seas.upenn.edu roth.amir@gmail.com mjs@clemson.edu guignard_monique@yahoo.fr

October 9, 2013

Abstract — **We present a coarse-grain (outer-loop) parallel implementation of RLT1/2/3 (Level 1, 2, and 3 Reformulation and Linearization Technique—in that order) bound calculations for the QAP within a branch-and-bound procedure. For a search tree node of size S, each RLT3 and RLT2 bound calculation iteration is parallelized S ways, with each of S processors performing $O(S^5)$ and $O(S^3)$ linear assignment problem solution calculations, respectively. Our implementation is aware of memory usage and availability and uses this information to throttle parallelism as appropriate and to manage resources during the branch-and-bound search. Our new code succeeded in solving for the first time the Tai30b instance of the QAP.**

Keywords — quadratic assignment, integer programming, reformulation linearization

[*]Corresponding Author; Department of Electrical and Systems Engineering, University of Pennsylvania, Philadelphia, Pennsylvania. Mailing Address: 2127 Tryon St., Philadelphia, PA 19146-1228.

[†]Technical Development Department, U.S. Department of Energy, Washington, DC.

[‡]Department of Mathematical Sciences, Clemson University, Clemson, SC 29634

[§]Operations and Information Management, The Wharton School, University of Pennsylvania, Philadelphia, Pennsylvania 19104

1 Introduction

The objective of this paper is to report on computational progress in solving the Quadratic Assignment Problem (QAP), and implications for solving related non-linear mixed 0-1 integer programming problems. The QAP addresses important research issues including the design of computing and communication systems, the planning of manufacturing facilities and the design of efficient signal sets for communication in difficult environments.

The exact solution method discussed in this paper is based on the Reformulation Linearization Technique (RLT) developed by Sherali and Adams [21]. RLT rewrites the assignment restrictions in terms of both binary and continuous variables via two steps:

Step 1. Reformulate the assignment problem through the multiplication of the equality constraints defining the assignment set by product factors of the 0-1 variables.

Step 2. Linearize the objective function and all constraints by substituting a non-negative continuous variable for each distinct nonlinear term.

Step 3. Tighten the relaxation by substituting the original variable for each new variable representing a power of that original variable.

Depending on the product factors used, different formulations emerge. The result is an multi-level hierarchy of mixed 0-1 linear representations of the original problem. Each level of the hierarchy provides a program whose continuous relaxation is at least as tight as the previous level. The highest level gives a convex hull representation. Consequently, each level increase results in a stronger (i.e., tighter) relaxation. We denote the ith level by RLTi. Sherali and Adams, on pp. 57–60 of their 1999 book [20] present both the first and second-level RLT constructs for the QAP as an illustration of the general RLT methodology.

We refer the reader to [2] and [9] to note the first attempts at exploiting the RLT1 structure for solving the QAP. References [1], [11] and [12] develop solution methods based on RLT2 and RLT3.

Balancing problem size with relaxation strength, the most promising method for solving the QAP appears to be RLT3. An important point made in [11] is that, in all but three test instances, the RLT3-based bounding algorithm found lower bounds that were so close to optimum that the best-known solution was confirmed as optimal. Just because a lower bound is tight, however, does not mean that one can count on it to be useful in a branch-and-bound algorithm. The bound has to be calculated quickly. The dual ascent bounds that we developed for RLT3 are calculated successively, so that if a bound from a weaker relaxation is adequate for its node, more expensive calculations with stronger relaxations are avoided. The graph in Figure 1 of [11] shows the fraction of the Nug24 optimum solution value that is reached by the RLT3-based lower bound, as a function of runtime. Excellent lower bounds are reached in only a few minutes.

The QAP is discussed in Section 2. Figure 1 illustrates how the branch-and-bound runtimes resulting from the work described in this paper compare with RLT2 branch-and-bound runtimes achieved in 2006. These improvements are due to the introduction of parallelism, the management of memory and recent innovative improvements in RLT3 bound strength. Optimization problems that can be solved as QAPs are discussed in Section 3. The computational approach we use for memory management and parallelizing our RLT3 branch-and-bound algorithm is described in Section 4. Table 1 details the experimental performance of this new software and compares these to earlier RLT3 results published in [11]. Conclusions are found in Section 5.

We have applied RLT1/2 to difficult quadratic and linear 0-1 problems other than the QAP with promising results. This encourages us to consider extending the parallelization effort we have made with the QAP to other problems, as well.

2 The Quadratic Assignment Problem (QAP)

The QAP was introduced by Koopmans and Beckmann [13], where N facilities must be assigned to N locations. Briefly, the Koopmans-Beckmann QAP may be formulated as follows: Let N denote the number of facilities and locations, and denote by $\boldsymbol{M}$ the set $\{1, 2, \ldots, N\}$ then:

$$\min_{p \in \boldsymbol{P}} \left\{ \sum_{i=1}^{N} \sum_{j=1}^{N} f_{ij} d_{p(i)p(j)} + \sum_{i=1}^{N} b_{ip(i)} \right\},$$

where $\boldsymbol{P}$ is the set of all permutations on $\boldsymbol{M}$ and $p : \boldsymbol{M} \to \boldsymbol{M}$ is a permutation function, $\boldsymbol{F} = [f_{ij}]$ is a flow matrix, $\boldsymbol{D} = [d_{p(i)p(j)}]$ is a distance matrix, $f_{ij}d_{p(i)p(j)}$ is the cost of assigning facility i to location $p(i)$ and facility j to location $p(j)$, $b_{ip(i)}$ is the fixed cost of placing facility i at location $p(i)$ and $\boldsymbol{B} = [b_{ip(i)}]$ is the fixed cost matrix. The problem is $\mathcal{NP}$-hard.

Lawler [15] introduced a four-index cost array $\boldsymbol{C} = \{c_{ikjl}\}$ instead of the three matrices $\boldsymbol{F}$, $\boldsymbol{D}$ and $\boldsymbol{B}$ and obtained the general form of the QAP as

$$\min_{p \in \boldsymbol{P}} \left\{ \sum_{i=1}^{N} \sum_{j=1}^{N} c_{ijp(i)p(j)} \right\}.$$

The relationship with the Koopmans-Beckmann problem is

$$c_{ikjl} = f_{ij}d_{kl},\ (i, j, k, l = 1, 2, \ldots, N, i \neq j \text{ and } k \neq l)$$

and

$$c_{ikik} = f_{ii}d_{kk} + b_{ik}\ (i, k = 1, 2, \ldots, N).$$

The standard mathematical programming formulation of the QAP is

$$\min \left\{ \sum_{i} \sum_{j} \sum_{k} \sum_{n} c_{ijkn}x_{ij}x_{kn}, \boldsymbol{x} \in \boldsymbol{X}, \boldsymbol{x} \text{ binary} \right\},$$

where

$$\boldsymbol{X} = \left\{ \boldsymbol{x} \geq \boldsymbol{0} : \sum_{i=1}^{N} x_{ij} = 1,\ j = 1, 2, \ldots, N;\ \sum_{j=1}^{N} x_{ij} = 1,\ i = 1, 2, \ldots, N \right\}$$

While great progress has been made on generating good solutions to large and difficult instances, this has not been the case for finding optimal solutions. In the mid 1960s, Nugent, Vollmann, and Ruml [18] posed a set of problem instances of sizes 5, 6, 7, 8, 12, 15, 20, and 30, noted for their difficulty. In the 1970s and 80s, one could only expect to exactly solve difficult instances for $N \leq 16$. It was not until the mid-1990s that Clausen and Perregaard [6] were able to solve the Nugent-20 instance. It was not until the year 2000 that optimum solutions were proven for the Nugent-30 and other difficult problems of similar size.

In 2006 the best exact solution solver on the difficult Nugent instances was the RLT1/2 solver [1]. No other exact solver found in the 2006 literature came even close [16]. The runtimes for RLT1/2 are shown in diamonds on the graph in Figure 1. These runtimes, achieved on Clemson University's earlier Palmetto Cluster were adjusted to what the runtimes would be on today's Palmetto Cluster. The newly achieved memory-aware parallel RLT1/2/3 results achieved for the same Nugent instances on today's Palmetto Cluster are shown in triangles. These constitute the main contribution reported in this paper.

Trend lines are drawn for both sets of experiments, in order to extrapolate runtimes for larger problems. For the size 30 Nugent problem, memory-aware RLT1/2/3 is 81.4 times faster than RLT1/2. Because runtime apparently grows exponentially with size, one could conclude that RLT1/2/3 could be of the order of 4,000 times faster for solving a size 40 problem than would RLT1/2, assuming one had the memory to do the calculations. In Section 4 we report our recent experiments that show significant improvement in both runtimes and memory management that makes RLT1/2/3 a viable approach to solving the QAP.

To get an idea of the memory required for RLT formulations of the QAP, the reader is referred to Figure 1 in [1], which plots the total RAM required for solving the RLT2 QAP formulation, as a function of problem size N.

Not all QAPs are Koopmans-Beckmann. One example is in balancing hydraulic and jet turbine blades [14, 17, 19]. This can be formulated as a QAP, where the 0-1 decision variable x_{ij} is 1 if blade i is allocated to position j and x_{ij} is 0 otherwise. The quadratic objective function corresponds to minimizing the distance between the center of gravity of the turbine shaft and its center of rotation. Our exact solution approaches always solve the general QAP form, not the Koopmans-Beckmann form.

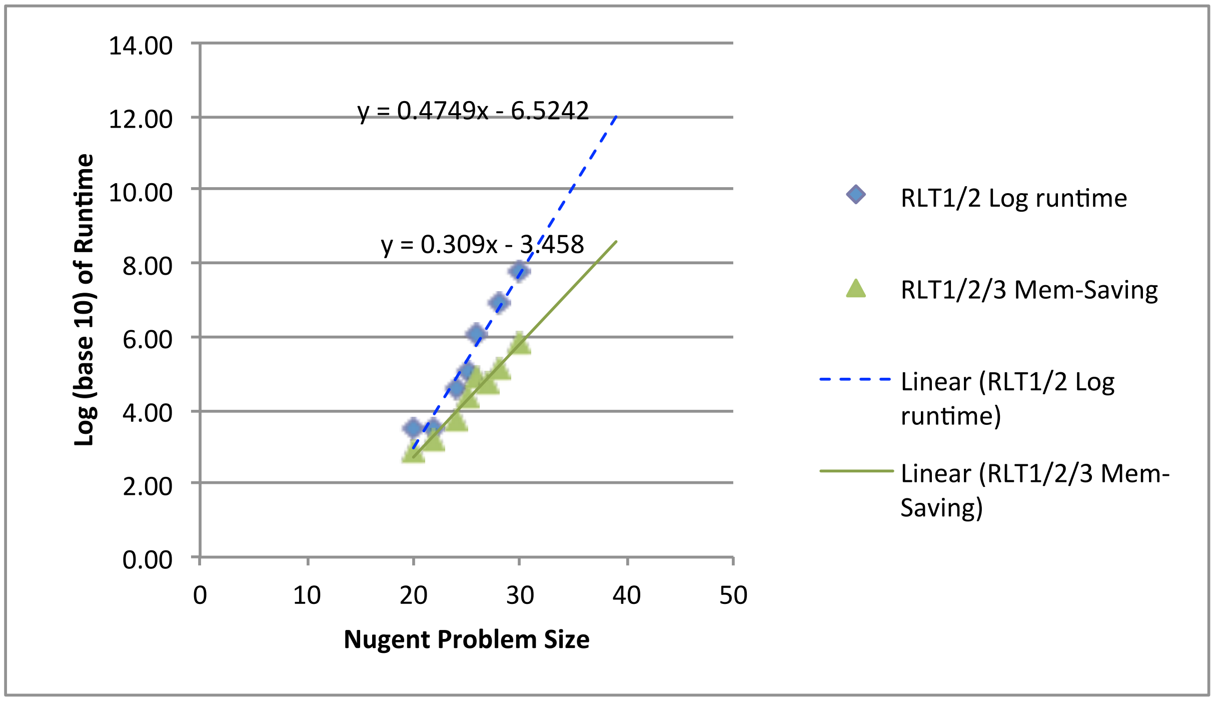

Figure 1: Comparison of Best 2006 QAP Solver and Today's best QAP Solver

3 Other problems that can be solved as a QAP

3.1 *The Bi-Quadratic Assignment Problem (BiQAP)*

The Bi-Quadratic (or Quartic) Assignment Problem (BiQAP) is a generalization of the QAP. It is a nonlinear 0-1 problem where the objective function is a fourth degree multivariable polynomial and the feasible domain is the assignment polytope. The BiQAP has applications in VLSI synthesis, where programmable logic arrays have to be implemented. Due to its difficulty, there appear to be no exact solvers in the existing literature. Only heuristic solution approaches have been found. For details of the VLSI application see Burkard, Çela and Klinz [5], who studied the BiQAP, derived lower bounds and investigated the asymptotic probabilistic behavior of these problems. Burkard and Çela [4] developed metaheuristics for the BiQAP and compared their computational performance.

It turns out that the RLT2 formulation of the QAP is in fact the BiQAP formulation (see [1]).

Thus our RLT2 based exact solution algorithm is perfectly capable of solving the BiQAP exactly. Furthermore, the parallelized code described in this paper is also applicable, because the RLT3 formulation of the QAP can be shown to be the same as the RLT2 formulation of the BiQAP.

3.2 *The Cubic Assignment Problem (CAP)*

Lawler first posed the Cubic Assignment Problem (CAP) in his seminal paper on the QAP [15]. Burkard, DellAmico and Martello describe the CAP in a recent SIAM Monograph [3]. The CAP is also mentioned, but not discussed in detail in the book by Du and Pardalos [7]. The CAP optimizes the problem of placing n entities at exactly n locations, where the cost of placement involves the interaction between triplets of entities, rather than the interaction between pairs of entities as is found in the QAP. Winter and Zimmermann [23] used a cubic assignment problem for optimizing the movement of materials in a storage yard. Burkard et al. [3] point out that the objective function (2.1.1) in [23] contains some typos in the indices, but is actually the objective function of a cubic assignment problem.

As far as we know, there exists no exact solution method for the CAP. But, in [1], we state that the formulation RLT2 of the QAP can be applied to the CAP. Thus, the RLT2 exact solution solver for the QAP is capable of solving the CAP exactly. To the best of our knowledge, no researchers have yet reported computational experience on solving the CAP using the RLT2 form, though the RLT theory establishes the mathematical equivalence between these representations. Not only is our RLT2-QAP solver capable of solving the CAP exactly, but so is our parallelized RLT3-QAP solver.

4 Computational advances

We base our computation of optimal value of QAP on the following RLT3 formulation derived from the standard form, as shown in Section 2 of our previous paper on RLT3 [11].

$$\min \left\{ \sum_i \sum_j c_{ijij} x_{ij} + \sum_i \sum_j \sum_{k \neq i} \sum_{n \neq j} c_{ijkn} y_{ijkn} \right.$$
$$+ \sum_i \sum_j \sum_{k,k \neq i} \sum_{n,n \neq j} \sum_{p \neq k,i} \sum_{q \neq j,n} d_{ijknpq} z_{ijknpq}$$
$$\left. + \sum_i \sum_j \sum_{k \neq i} \sum_{n \neq j} \sum_{p \neq k,i} \sum_{q \neq j,n} \sum_{g \neq i,k,p} \sum_{h,q \neq j,n,q} e_{ijknpqgh} v_{ijknpqgh} \right\}$$

$$\text{s.t. } \boldsymbol{x} \in \boldsymbol{X},\ \boldsymbol{x} \text{ binary},$$
$$\boldsymbol{y} \in \boldsymbol{Y}_{ij},\ \boldsymbol{y} \text{ binary},$$
$$\boldsymbol{z} \in \boldsymbol{Z}_{ijkn},\ \boldsymbol{z} \text{ binary},$$
$$\boldsymbol{v} \in \boldsymbol{V}_{ijknpq},\ \boldsymbol{v} \text{ binary},$$

where

$$\boldsymbol{Y}_{ij} = \left\{ y_{ijkn} \geq 0 : \sum_n y_{ijkn} = x_{ij}, \sum_k y_{ijkn} = x_{ij}, k \neq i, n \neq j \right\},$$
$$\boldsymbol{Z}_{ijkn} = \left\{ z_{ijknpq} \geq 0 : \sum_q z_{ijknpq} = y_{ijkn}, \sum_p z_{ijknpq} = y_{ijkn}, \right.$$
$$\left. \text{alldiff}(p,k,i),\ \text{alldiff}(q,n,j) \right\},$$

and

$$\boldsymbol{V}_{ijknpq} = \left\{ v_{ijknpqgh} \geq 0 : \sum_h v_{ijknpqgh} = z_{ijkn}, \sum_g v_{ijknpqgh} = z_{ijkn}, \right.$$
$$\left. \text{alldiff}(g,p,k,i),\ \text{alldiff}(h,q,n,j) \right\}.$$

Note that the RLT1 and RLT2 formulations of the QAP are subsumed by this formulation. And, we have chosen to not show the lower and upper bounds for the index variables. For more detail about the derivation of this formulation, please see Chapter 6 of [24]. For detail about the branching methods used in our branch and bound algorithm, please see [10].

The drawbacks of RLT3 bound calculations, and any branch-and-bound procedures that use them, are the large memory requirements and long computational times. The memory requirement for the coefficient matrix is $O(S^8)$, $(S(S-1)(S-2)(S-3))^2/24$, to be precise, where S is the problem size at a specific branching node. Note each coefficient corresponds to each of the $v_{ijknpqgh}$ variables but we only need to store one coefficient for 24 complementary variables. Assuming 64-bit integer data, this works out to 144 GBytes for problem size $S = 30$, 1.6 TBytes for $S = 40$, 10.1 TBytes for $S = 50$, and so on. Branch-and-bound search typically requires 3 to 4 times as much memory as the largest single bound calculation meaning that even one TByte of memory can accommodate problems of at most $N = 30$. Compute time is also $O(S^9)$, as each bound calculation must solve $O(S^6)$—i.e., $(S(S-1)(S-2))^2$—linear assignment problems (LAPs) of size $(S-3)$ each of which requires $O(S^3)$ runtime, in the worst case [8].

To accommodate this situation, we developed a thoroughly parallelized RLT3 implementation. The structure of the process at each node in the branch and bound tree is presented in Table 2. For a node of size S (i.e., with S unfixed assignments), each RLT3 and RLT2 bound calculation iteration is parallelized S ways, with each of S processors performing $O(S^5)$ and $O(S^3)$ LAP calculations. Therefore, the memory requirement for each node is$O(S^7)$ and $O(S^5)$ respectively. By default, parallelization is implemented at the outermost level, which corresponds to S possible assignments of the first fixed variable. Because all LAP calculations corresponding to one assignment are independent of all calculations corresponding to a different assignment—i.e., they do not access any of the same cost matrix elements—this parallelization scheme can be implemented with minimal synchronization overhead. "Child" sub-problem creation and solution testing are also parallelized S ways.

To control memory requirements, we store the RLT3 objective function cost matrices on disk and load them into memory on demand. Our branch-and-bound code can operate at near peak efficiency given only enough memory to store the cost matrices of the largest two nodes in memory—sufficient to parallelize both RLT3 bound calculations and sub-problem creation. The storing/loading of cost matrices to/from disk is itself parallelized S ways. To cut memory usage, disk traffic, and runtime even further, we create the RLT2 and RLT3 cost matrices on demand—the RLT2 matrix if RLT1 stalls at a given node, and the RLT3 matrix if both RLT1 and RLT2 stall. We exploit this structure to speculatively branch to a child node if we estimate that we can reach the bound using RLT1 and RLT2 calculations only. If this strategy fails (i.e., the ensuing RLT1/RLT2 bound calculation does not eliminate the child node), our code backtracks to contimue applying RLT3 to the child's parent, this time knowing just how much progress RLT1/RLT2 at the children nodes can make. In other words, knowing how difficult it will be to eliminate children nodes makes it easier for our algorithm to be more patient using RLT3 to eliminate a potential parent.

The result of this heuristic is that we visit a large number of nodes, but incur the memory and runtime costs of RLT3 on a small fraction of them. For instance, on the Nugent 20 problem, we visit 39 nodes, but use RLT3 on only nine.

To illustrate the capabilities of our new code, we tested it on the standard benchmark Nugent problem instances, which are difficult to solve exactly. Table 1 presents nodes visited, memory requirements, and runtimes for an earlier serial FORTRAN implementation and our new code which is written in C and parallelized using the pthreads library. Our new code needs about one-third the memory and outperforms the old code by a factor of just under $N/2$. The factor of 2 slowdown can be attributed to the difference between the FORTRAN and C compilers—the former produces serial code that is roughly twice as fast as the latter. The remaining deficit from N is small—the program is parallelized through and through—and is due to disk operations, primarily those used for checkpointing. Recall from the previous paragraph that the parallelized C-code aggressively searches more nodes than the older serial FORTRAN case, in order to minimize total time spent in RLT3. In the older serial FORTRAN case, however, most of the node bounds were calculated with RLT3. So, for serial FORTRAN, it did not seem important to distinguish between nodes visited and nodes that required an RLT3 calculation.

Our new code succeeded in solving for the first time the Tai30b instance of the QAP [22]. Like the previous experiments, this was done on a single node of the Palmetto Supercomputing Cluster at Clemson University. Specifically, we used 30 2.2 GHz Intel Xeon processors (out of 64 available)

	RLT1/2/3 Serial Fortran			RLT1/2/3 Parallel C			
Nugent instance	Nodes visited	Memory (GB)	Time (sec)	Nodes visited	Nodes RLT3	Memory (GB)	Time (sec)
20	368	20.9	15,571	39	9	18.4	736
22	138	52.6	44,077	52	16	40.7	1,578
24	291	192.2	44,142	102	16	85.0	5,097
25	793	328.7	169,549	267	31	120.3	23,072
26	556	442.6	712,939	574	54	167.9	80,269
27	697	651.7	418,960	359	46	231.5	53,237
28	737	883.6	1,433,178	1,538	115	321.4	130,527
30	N/A	N/A	N/A	3,383	251	722.6	733,812

Table 1: Branch and Bound Exact Algorithm Results

and 753 GBytes (of 2 TBytes) to find an exact solution in 315,584 seconds.

4.1 *Using Finer-grain Parallelism and Pipelined Memory Management to Tackle Larger Problems*

Our current implementation has achieved memory footprint reductions of 3X and reductions in runtime on the order of problem size N. This allows us to obtain exact solutions for problems of size 30 on current "large-memory" shared multiprocessors in the time frame of days to weeks. Tackling larger problems requires a more flexible, finer-grain approach managing parallelism and memory.

Table 2 shows the structure of the RLT3 bound calculation, and the memory requirements and parallelism at each loop nest.

Our code currently parallelizes the j loop. Strictly speaking, in the worst case scenario where we would need to parallelize at the beginning of the tree, parallelization at this level would require $O(S^7)$ memory—enough to contain all S assignments of *one* of the S variables. S^8 memory is preferred because the RLT3 bound calculation at each node typically requires multiple iterations and keeping the entire RLT3 cost matrix in memory saves us from having to read slices of it to and from disk. However, making do with $O(S^7)$ memory is certainly possible. And, it may not even result in perceived slowdown, if memory management and disk operations are overlapped with computations in a pipeline fashion. For instance, using two chunks of $O(S^7)$ memory we

```
distribution
for $i$ = 1 to $S$                                   $S^8$ memory
    for $j$ = 1 to $S$                               $S^7$ memory, $S$ parallelism
        sub-matrix $ij$ distribution
        for $k$ = $i$ to $S$                         $S^6$ memory
            for $l$ = 1 to $S$                       $S^5$ memory, $S$ parallelism
                sub-matrix $ijkl$ distribution
                for $m$ = $k$ to $S$                 $S^4$ memory
                    for $o$ = 1 to $S$               $S^3$ memory, $S$ parallelism
                        sub-matrix $ijklmo$ distribution
                        sub-matrix $ijklmo$ cost subtraction
                sub-matrix $ijkl$ cost subtraction
        sub-matrix $ijkl$ cost subtraction
cost subtraction
```

Table 2: Parallelism and memory usage in RLT3 bounds calculation for subproblem of size S

could perform RLT3 bound calculation on all assignments of variable i in one chunk. In parallel, we could save the contents of the other chunk—containing the RLT3 cost matrix slice for variable $i-1$—to disk and then read into it the contents of the slice corresponding to variable $i+1$. The next iteration, we operate on the second memory chunk we just loaded while storing and reloading the first chunk. If the ratio of computation to memory management is too small, we can use a third $O(S^7)$ memory chunk to parallelize storing of $i-1$ and loading of $i+1$ with one another.

Even larger problems can be tackled by parallelizing the l loop instead of the j loop, requiring two or three chunks of memory sized to hold only $O(S^5)$ elements. With this parallelization strategy, 60-variable problems could be tackled with only 150 MBytes of memory although TBytes of disk would still be required and the calculation would take years.

Our code maximizes parallelism adaptively, given the available memory. If memory can fit only a single RLT3 cost matrix, creation of a child problem of size S is still parallelized to a degree of $S-1$. This is done by changing the parallelism granularity to the second (inner) variable assignment loop, and performing child creation in S steps. After each step, the just created sections of the child cost matrix are stored to disk. Similarly, if available memory cannot fit even a single RLT3 cost matrix, the parallelism granularity of RLT3 bound calculations is changed to $S-1$ implemented at the second (inner) variable assignment loop, and each bound calculation is performed in S steps with the corresponding matrix sections swapped from and to disk before and after each iteration. Essentially, our code contains an application-specific mini-operating system kernel that explicitly manages its usage of memory, disk, and processors. It accepts memory size and number of processors as parameters and dynamically "squeezes" each stage into the available resources. Incidentally, our code is capable of exploiting parallelism of $2S$, $3S$, etc., up to S^2. However, whereas shared-memory systems with $2n$ and $3n$ processors and S^8 memory exist for problem sizes of interest (e.g., $S = 30$), systems with S^2 processors do not. S^2 processors are available on graphics processing units (GPUs), but these have significantly less memory.

5 Summary and future efforts

In this paper we describe a coarse-grain (i.e., outer-loop) parallel implementation of RLT3 (Level 3 Reformulation and Linearization Technique) bounds calculations for the QAP embedded within a branch-and-bound procedure. Our implementation is aware of memory usage and availability and uses this information to throttle parallelism as appropriate and to manage resources during branch-and-bound traversal (see Section 4).

We plan to parallelize other bound calculation algorithms and to develop strategies to manage parallelism at multiple levels dynamically. This last capability is important not only to be able to tackle very large problems, but also to exploit emerging Graphics Processing Units (GPUs), which are quickly growing in popularity as scientific computing platforms due to their extremely high FLOPS/Watt and FLOPS/dollar. Aligning state-of-the-art solvers with this emerging platform offers tremendous potential in terms of scaling up parallelism. To experiment with this new substrate, we applied for and received two 448-core Tesla C2075 boards via an NVIDIA Professor Partnership Grant.

We hope that this paper will encourage others to pursue RLT for solving other 0-1 problems that are resistant to fast exact solution. Representations for all such problems will have exploitable block-diagonal structures that lend themselves to parallelization, though the individual blocks will generally not be assignment sets.

Acknowledgments

First and foremost, thanks go to William Hightower, retired Professor of Computer Science and Mathematics, without whose insights the QAP-RLT2 and QAP-RLT3 algorithms would never have been conceived.

The computational advances reported herein were made possible through the advanced computing resources provided by Clemson University's Computing and Information Technology Center and International Center for Automotive Research Center for Computational Mobility Systems and the computing facilities made available by the Operations and Information Management Department of the University of Pennsylvania's Wharton School. The Palmetto Cluster resources used in this research were acquired under NSF award 1026385.

The authors are particularly thankful for the competent assistance of Clemson's Cyber Infrastructure Technology Integration group and the unflagging technical assistance provided by Fabiano Prestes, Sr. IT Support Specialist at the Wharton School of the University of Pennsylvania.

Some of the experimental work reported herein was done on the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575. Also, we thank the NVIDIA Corporation for providing two 448-core Tesla C2075 boards to implement our algorithms via their Professor Partnership program.

For his excellent work editing this paper, we give thanks to Heng Zhang, graduate student in the Electrical and Systems Engineering Department of the University of Pennsylvania.

Last but not least, we thank David O'Neal Senior Scientific Specialist at the Pittsburgh Supercomputing Center (PSC) for his expert advice on parallelization of our RLT1 and RLT2 codes and the strategies he gave us for getting the most out of PSC's powerful Blacklight shared memory computing system.

References

[1] W. P. Adams, M. Guignard, P. M. Hahn, and W. L. Hightower. A level-2 reformulation-linearization technique bound for the quadratic assignment problem. *European Journal of Operational Research*, 180:983–996, 2007.

[2] W. P. Adams and T. A. Johnson. Improved linear programming-based lower bounds for the quadratic assignment problem. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 16:43–75, 1994.

[3] R. Burkard, M. Dell Amico, and S. Martello. *Assignment Problems*. SIAM Monograph, 2009. Hardcover ISBN: 978-0-898716-63-4.

[4] R. E. Burkard and E. Çela. Heuristics for biquadratic assignment problems and their computational comparison. *European Journal of Operational Research*, 83:283–300, 1995.

[5] R. E. Burkard, E. Çela, and B. Klinz. On the biquadratic assignment problem. In P. M. Pardalos and H. Wolkowicz, editors, *Quadratic Assignment and Related Problems*, volume 16 of *DIMACS Series*, pages 117–146. American Mathematical Society, Prvidence RI, 1994.

[6] J. Clausen and M. Perregaard. Solving large quadratic assignment problems in parallel. *Computational Optimization and Applications*, 8:111–128, 1997.

[7] D. Z. Du and P. M. Pardalos. *Handbook of Combinatorial Optimization*. Kluwer Academic Publishers, Boston MA, 1998.

[8] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. J. ACM. 10:248-264, 1972.

[9] P. M. Hahn and T. L. Grant. Lower bounds for the quadratic assignment problem based upon a dual formulation. *Operations Research*, 46(6):912–922, 1998.

[10] 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. *Yugoslav Journal of Operations Research*, 11(1):41–60, 2001.

[11] P. M. Hahn, Y.-R. Zhu, M. Guignard, W. L. Hightower, and M. Saltzman. A level-3 reformulation-linearization technique bound for the quadratic assignment problem. *INFORMS Journal on Computing*, 24:202–209, 2012. published online before print April 7, 2011, doi:10.1287/ijoc.1110.0450.

[12] P. M. Hahn, Y.-R. Zhu, M. Guignard-Spielberg, and J. M. Smith. Exact solution of emerging quadratic assignment problems. *International Transactions on Operations Research*, 17(5):525–552, 2010. invited survey paper.

[13] T. C. Koopmans and M. J. Beckmann. Assignment problems and the location of economic activities. *Econometrica*, 25:53–76, 1957.

[14] G. Laporte and H. Mercure. Balancing hydraulic turbine runners: A quadratic assignment problem. *European Journal of Operational Research*, 35:378–381, 1988.

[15] E. L. Lawler. The quadratic assignment problem. *Management Science*, 9:586–599, 1963.

[16] E. M. Loiola, N. M. M. de Abreu, P. O. Boaventura-Netto, P. M. Hahn, and T. Querido. A survey for the quadratic assignment problem. *European Journal of Operational Research*, 176:657–690, 2006. invited review.

[17] A. Mason and Rönnqvist M. Solution methods for the balancing of jet turbines. *Computers and Operations Research*, 24(2):153–167, 1997.

[18] 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.

[19] L. S. Pitsoulis, P. M. Pardalos, and D. W. Hearn. Approximate solutions to the turbine balancing problem. *European Journal of Operational Research*, 130:147–155, 2001.

[20] H. D. Sherali and W. P. Adams. *A Reformulation-Linearization Technique for Solving Discrete and Continuous Nonconvex Problems*. Kluwer Academic Publishers, Norwell MA, 1999.

[21] H. D. Sherali and W. P. Adams. A hierarchy of relaxations between the continuous and convex hull representations for zero-one programming problems. *SIAM Journal on Discrete Mathematics*, 3:411–430, 2990.

[22] E. D. Taillard. Robust tabu search for the quadratic assignment problem. *Parallel Computing*, 17:443–455, 1991.

[23] Th. Winter and U. Zimmermann. Dispatch of trams in storage yards. *Annals of Operations Research*, 96:287–315, 2000.

[24] Y.-R. Zhu. *Recent advances and challenges in the quadratic assignment and related problems*. PhD thesis, University of Pennsylvania, August 2007. ESE Department.