Volume 195, Issue 3, 16 June 2009
ISSN 0377-2217
ELSEVIER
EUROPEAN JOURNAL OF OPERATIONAL RESEARCH
Available online at
ScienceDirect
www.sciencedirect.com
Feature Clusters:
Formal Methodologies and Tools for Decision Support Systems
Guest Editor: Pascale Zaraté
Metaheuristics: Transportation and Logistics
Guest Editors: Patrick Siarry and Éric Taillard
Knowledge-based Multi-criteria Decision Support
Guest Editor: Constantin Zopounidis
LAST ISSUE OF THIS VOLUME
www.elsevier.com/locate/ejor

Available online at www.sciencedirect.com
ScienceDirect
European Journal of Operational Research 195 (2009) 810–826
EUROPEAN JOURNAL OF OPERATIONAL RESEARCH
www.elsevier.com/locate/ejor

A cooperative parallel tabu search algorithm for the quadratic assignment problem

Tabitha James [a,*], Cesar Rego [b], Fred Glover [c]

[a] *Department of Business Information Technology, Pamplin College of Business, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061, USA*
[b] *School of Business Administration, University of Mississippi, University, MS 38677, USA*
[c] *University of Colorado, Boulder, CO 80309-0419, USA*

Received 2 February 2007; accepted 16 June 2007
Available online 22 November 2007

Abstract

In this study, we introduce a cooperative parallel tabu search algorithm (CPTS) for the quadratic assignment problem (QAP). The QAP is an NP-hard combinatorial optimization problem that is widely acknowledged to be computationally demanding. These characteristics make the QAP an ideal candidate for parallel solution techniques. CPTS is a cooperative parallel algorithm in which the processors exchange information throughout the run of the algorithm as opposed to independent concurrent search strategies that aggregate data only at the end of execution. CPTS accomplishes this cooperation by maintaining a global reference set which uses the information exchange to promote both intensification and strategic diversification in a parallel environment. This study demonstrates the benefits that may be obtained from parallel computing in terms of solution quality, computational time and algorithmic flexibility. A set of 41 test problems obtained from QAPLIB were used to analyze the quality of the CPTS algorithm. Additionally, we report results for 60 difficult new test instances. The CPTS algorithm is shown to provide good solution quality for all problems in acceptable computational times. Out of the 41 test instances obtained from QAPLIB, CPTS is shown to meet or exceed the average solution quality of many of the best sequential and parallel approaches from the literature on all but six problems, whereas no other leading method exhibits a performance that is superior to this.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Tabu search; Combinatorial optimization; Parallel computing; Quadratic assignment problem

1. Introduction

The quadratic assignment problem (QAP) is a well-known NP-hard combinatorial optimization problem. Its most common application is in facility location. In this context, the problem consists of assigning n facilities (or warehouses) to n locations (or sites) with the objective of minimizing the transportation costs associated with the flow of materials between facilities and the distances between locations. The QAP can be formulated as a permutation problem as follows:

$$\min_{\pi\in\prod} \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{\pi(i)\pi(j)},$$

where π is an assignment vector of size n, f is a matrix of flows of items to be transported between facilities and d is a matrix containing the distances or costs of transporting a single item between any two locations. The objective is to find an assignment vector which minimizes the total transportation costs given by the sum of the product of the flow and distance between all pairs in $\prod$.

Although facility location has been the most popular applied area of the QAP, the problem finds applications in numerous settings that include scheduling, statistical data analysis, archeology, chemistry, information retrieval, parallel and distributed computing, facility layout and

* Corresponding author. Tel.: +1 540 231 3163.
E-mail addresses: tajames@vt.edu (T. James), crego@bus.olemiss.edu (C. Rego), fred.glover@colorado.edu (F. Glover).

0377-2217/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.ejor.2007.06.061

transportation. The problem of assigning gates to incoming and departing flights at an airport is a transportation application that may be formulated as a QAP (Haghani and Chen, 1998). Allocation is a popular logistic application of the QAP. Ciriani et al. (2004) explore assigning rooms to persons with undesirable neighborhood constraints formulated as a QAP. A generalization of the QAP is also used for a service allocation problem (Cordeau et al., 2005) with the purpose of minimizing the container rehandling operations at a shipyard. Parallel computing and networking provide other location analysis problems that can be formulated as QAPs (Gutjahr et al., 1997; Siu and Chang, 2002). For a comprehensive survey of these and other applications of the QAP and its special cases, we refer to Cela (1998) and Loiola et al. (2007). It is also possible to formulate several other well-known combinatorial optimization problems as QAPs, including the traveling salesman problem, the bin-packing problem, the maximum clique problem, the linear ordering problem, and the graph-partitioning problem, each embracing a variety of other applications in the fields of transportation, manufacturing, logistics, economics, engineering, science, and sociology.

The solution complexity of the QAP is widely accepted. No known polynomial time algorithm exists to solve QAP instances with more than a relatively small number of inputs. The exact solution of even small problems ($20 < n < 30$) is considered computationally non-trivial. To obtain optimal solutions for modest size QAP instances ($30 < n < 40$) a massively parallel computing environment is required. One of the most successful exact approaches for the QAP is due to Anstreicher et al. (2002). Their parallel branch and bound approach for the QAP is implemented on a large grid and can obtain optimal solutions for problems of size 30. Due to the acknowledged difficulty of solution, many sequential and parallel metaheuristic approaches have been applied to the QAP. The current study proposes a cooperative parallel tabu search for the QAP that exploits the parallel computing platform chosen to provide good quality solutions for 41 QAPLIB test instances in acceptable computational times that are attractive compared to competitive methods. The remainder of this paper is organized as follows. A review of sequential and parallel metaheuristic algorithms for the QAP is provided in Sections 2 and 3, respectively. Section 4 presents the CPTS algorithm. Section 5 discusses design considerations of the CPTS algorithm inherent to the parallel platform used for the implementation. A computational analysis is presented in Section 6 followed by the conclusions in Section 7.

2. Sequential metaheuristics for the QAP

Sequential metaheuristic algorithms for the QAP include a variety of approaches. Tabu search (TS) approaches include the robust tabu search (RTS) algorithm due to Taillard (1991). This algorithm capitalizes on simple TS principles to quickly find high quality solutions to the QAP. The RTS algorithm has been further popularized in the literature as an improvement method for many of the most successful metaheuristic approaches for the QAP. A multi-start tabu search (JRG-DivTS) that demonstrates improved solution quality compared to RTS, is introduced by James et al. (2006). JRG-DivTS is a basic tabu search which is restarted from a diversified copy of the best solution found when the search stagnates. When JRG-DivTS is restarted, the tabu restrictions are released and the tabu tenure parameters are modified. Another TS approach for the QAP is due to Misevicius (2005). Misevicius combines a simplified version of RTS with rudimentary diversification operators that periodically perturb the solution provided by the TS procedure (M-ETS-1, M-ETS-2, and M-ETS-3). This approach is demonstrated to work well on the symmetric and asymmetric Taillard QAP instances.

Many of the sequential metaheuristic approaches for the QAP that demonstrate the ability to obtain high quality results are genetic algorithm (GA) variants which incorporate a tabu search (TS) method to improve the solutions provided by the GA operators. Misevicius presents two algorithms that combine a simplified version of the RTS algorithm with rudimentary diversification operators which are then embedded within a GA framework. The first GA variant (M-GA/TS) applies a "ruin-and-recreate" procedure. M-GA/TS uses a modified version of RTS to "recreate" a solution that has been randomly perturbed by the "ruin" procedure or the crossover operator (Misevicius, 2003). The second approach (M-GA/TS-I) uses a less random "ruin" procedure, called a shift operator, to perturb the solutions provided by the GA once the algorithm starts to converge (Misevicius, 2004).

Drezner (2003, 2005) explores several TS variations within a hybrid GA that are shown to perform well on the Skorin-Kapov QAP instances. The first, D-GA/SD, uses a special crossover called a "merging process" coupled with a greedy local search procedure that executes swap moves until a local optimum is found. The D-GA/S-TS algorithm is the same as D-GA/SD except that it extends the descent local search method with a simple TS. The third algorithm, D-GA/C-TS, further enhances the improvement method with a concentric tabu search operator that examines moves based upon their distance from a "center" solution. D-GA/IC-TS, the last GA hybrid, extends D-GA/C-TS to permit more moves.

Other types of sequential metaheuristics applied to the QAP include: ant colony optimization (Stutzle and Dorigo, 1999; Gambardella et al., 1997), path-relinking (James et al., 2005), and GRASP (Li et al., 1994). A GRASP algorithm enhanced by the TS path-relinking strategy is given by Oliveira et al. (2004). Tseng and Liang (2005) introduce an algorithm that combines ant colony optimization with a genetic algorithm and a simple local search (TL-ACO/GA/LS).

The solution difficulty and computational requirements of the QAP make the application of parallel computing to this problem particularly attractive. Metaheuristic approaches can provide good solutions, but not necessarily optimal solutions, to complex problems in acceptable computational times, less by many orders of magnitude than the times required by exact methods to solve problems of much smaller size. Sequential metaheuristic methods have made tremendous amounts of progress towards this goal for the QAP. However, parallel metaheuristic techniques have been explored to a much lesser extent despite the obvious benefits of improving solution quality in less wall clock time. The following section will provide an overview of a parallel heuristic taxonomy and the parallel metaheuristic approaches for the QAP that correspond to these classifications.

3. Parallel metaheuristics for the QAP

Parallel designs for metaheuristic methods that can be applied to combinatorial optimization problems are attractive as they may provide for both better solution quality and reductions in run time to reach a good solution. The repetitive nature of metaheuristic methods and the complexity of solution, characteristic to many of the problems to which metaheuristics are applied, make the use of parallelism an attractive alternative. As access to parallel hardware becomes more prevalent, the possibilities of exploring the usefulness of such techniques have become greater. Crainic and Toulouse (2003) develop a classification scheme for parallel metaheuristic methods. Their taxonomy provides a valuable means by which to highlight differences between the various parallel metaheuristic approaches for the QAP as well as illustrate the design choices for the development of a parallel metaheuristic.

Crainic and Toulouse group strategies for parallel heuristic methods into three categories based upon the division of work assigned to each processor. In the first strategy, a low-level parallelization strategy deemed Type 1; a computationally expensive part of a heuristic is parallelized with the sole purpose of deriving run time speedups. In this case, a master process is responsible for controlling the assignment of work to each processor and the collection, as well as the aggregation, of the results. Decomposition (or Type 2) strategies consist of the division of the decision variables among processors. By dividing the decision variables among processors, the size of the solution space being explored by each processor is decreased. The resulting partial explorations must be combined to obtain a feasible solution and this is typically performed by a master process. This type of parallelization changes the base execution nature of the original sequential algorithm and therefore may influence the solution quality.

Crainic and Toulouse's last classification is a multi-heuristic (or Type 3) parallel model wherein each processor executes a complete copy of some heuristic on the entire problem. The model comprises two variants of parallel designs: independent search strategies and cooperative multi-thread strategies. These two strategies are differentiated by how the exchange of information between processors is handled. Both strategies include the exchange and aggregation of some information that occurs either at the end (independent search) or during the execution (cooperative multi-thread) of the concurrent searches. Several variations of this strategy are common, including variants where different heuristics may be executed on the same problem on different processors. The searches may also start from the same initial solution or different initial solutions. We should mention that the term thread is used in this classification to generally describe a process executing a portion of the entire algorithm, which technically does not necessarily materialize through a thread of execution in a concurrent or parallel system.

3.1. Low-level parallelism (Type 1) strategies for the QAP

Taillard (1991) proposes a low-level parallelism (Type 1) strategy for the QAP. In this algorithmic design, Taillard examines subdividing the neighborhood exploration between multiple processors. Each processor does not operate on a different solution, but rather each processor operates on a different portion of the same permutation. The exchanges (an exchange referring to the swap of two facilities to the location the other previously occupied) are subdivided between processors. Each processor calculates all the move costs for the exchanges assigned to it. Then every individual processor broadcasts its best move to all other processors and each processor performs the global best move on its copy of the permutation. The algorithm then continues to iterate from the new permutation in the same manner. This parallelization is much different than more recently proposed parallel algorithms for the QAP including the one in this study. However, for the particular parallel architecture utilized and the quality of hardware at the time it was introduced, Taillard's procedure was a highly sophisticated and beneficial approach. While the approach is technically sophisticated, the method results only in a faster RTS algorithm identical to its sequential counterpart. As with all Type 1 strategies, this design will not change the quality of the results, but will only speed up the sequential algorithm execution.

Chakrapani and Skorin-Kapov (1993) also examine a similar algorithm for the QAP. Their implementation is written for an older fine grain SIMD parallel architecture called a Connection Machine. SIMD stands for single instruction – multiple data. A SIMD machine executes the same instruction on different data elements at the same time. The architecture in this case restricts the flexibility in parallelization options. However, this study presents a typical example of a low-level parallelism (Type 1) strategy. In their algorithm, the evaluations of the pairwise exchanges on the permutation are divided among multiple processors. Processors cooperate to compute the costs of all possible pairwise exchanges and the results are aggregated. The best exchange is performed on the permutation and the algorithm continues to iterate in the same manner.

Another algorithm that falls within this category is the parallel path-relinking algorithm given in James et al.

(2005). This algorithm generates multiple solutions by applying path-relinking to solutions maintained in a global reference set, incorporating the customary path-relinking strategy of using one or more solutions to guide the moves performed on another solution (in this case, the permutation being manipulated). The exchanges employed in this previous study move the facilities into the locations they occupy in the guiding solution. To improve the solutions created by the path-relinking procedure, the RTS algorithm is run as an operator on each newly generated solution. In the parallel implementation of this algorithm, the path-relinking operator is used to generate a set of trial solutions sequentially and the RTS algorithm is then run in parallel on multiple trial solutions. This leads to a decrease in run time, since the TS is an expensive operation, but does not change the basic nature of the sequential algorithm or the solution quality.

3.2. Decomposition (Type 2) strategies for the QAP

While we are not aware of any decomposition (Type 2) strategies for the QAP, they have shown promise for the TSP which is a special case of the QAP (Felten et al., 1985; Fiechter, 1994).

3.3. Multi-heuristic (Type 3) strategies for the QAP

The parallel tabu search proposed by Taillard (1991) is an independent search algorithm which falls within the multi-heuristic (Type 3) classification. The algorithm consists of running concurrent, independent tabu searches starting from different initial solutions. No information is exchanged throughout the run of the TS algorithms and the best solution is determined at the end of the execution all of the threads.

Battiti and Tecchiolli (1992) also propose an independent search strategy that runs concurrent executions of a tabu search algorithm. Talbi et al. (1997) propose a parallel multi-heuristic (Type 3) tabu search with each processor running a complete TS algorithm starting from a different initial solution. This algorithm differs from the two previous algorithms by varying the TS parameters on each processor. A central memory (or master process) is used to collect the solutions from the slave processors and keep track of the global best solution found. All three of the algorithms above are independent search strategies and follow the same design.

Pardalos et al. (1995) examine a parallel GRASP implementation for the QAP which follows the same independent search strategy of the above TS variants but running parallel GRASP algorithms concurrently on multiple processors and keeping a global best from all processors.

De Falco et al. (1994) examine a cooperative multi-thread tabu search strategy. In this algorithm each processor examines a set of moves on its starting (or seed) solution. Then a subset of the processors are synched by exchanging their current best solutions found and the best of these solutions is used to update the processors current best. This info r-mation is exchanged between processors throughout the search rather than aggregated only at the end by a master process.

Both an independent search strategy TS and a cooperative multi-thread metaheuristic, combining several techniques, are proposed by Talbi and Bachelet (2006). The independent search strategy is simply concurrent executions of a TS algorithm. In this parallel algorithm the global best solution is obtained from the multiple processors at the end of the run and there is no cooperation between the searches. On the other hand, the cooperative multi-thread (COSEARCH) algorithm combines multiple tabu searches with a GA used for diversification purposes. Intensification is achieved by focusing the search in the vicinity of high quality (or elite) solutions which are slightly perturbed by means of a special *kick* operator before the actual intensification starts. The global memory, accessible by all processors, consists of a set of elite solutions, a set of initial (or reference) solutions and a frequency matrix. Each TS is run on a solution from the reference solutions matrix which is periodically updated by the GA algorithm. The TS on each processor updates both the elite solution matrix and the frequency matrix. The reference solutions are used to upgrade the GA population and the frequency matrix is used in the fitness function to penalize assignments that have high frequencies. The *kick* operator takes a solution from the elite solution matrix and applies a controlled number of random pairwise exchanges to the permutation. These newly created solutions are then inserted into the reference solutions matrix. This algorithm is highly cooperative with information being exchanged between the processors throughout the whole search.

Battiti and Tecchiolli (1992) also propose a parallel genetic algorithm for the QAP which is classified as a multi-heuristic strategy by Crainic and Toulouse. This parallel GA is generally referred to as a course-grained parallel GA in the literature. The course-grained parallel GA functions by subdividing the population of solutions and running each subpopulation on an independent processor. Some defined exchange of information between subpopulations (or processors) occurs throughout the execution of the algorithm.

The above algorithms constitute many of the proposed parallel algorithms for the QAP that fall into each category of Crainic and Toulouse's taxonomy. For a broader review of parallel heuristic methods for all problems, the interested reader may refer to Crainic and Toulouse (2003). The parallel architectures used in the above studies widely vary. These platform differences may influence both parallel algorithm design and performance (both in terms of run time and solution quality). The algorithm proposed in the current study is a cooperative multi-thread (Type 3) tabu search algorithm. We incorporate variations of some of the parallelization techniques outlined above. The following sections will detail the implementation and the design considerations of the proposed algorithm.

4. Cooperative parallel tabu search for the QAP

The cooperative parallel tabu search algorithm (CPTS) developed in the current study consists of the parallel execution of several TS operators on multiple processors. The base TS operator is a modified version of Taillard's robust tabu search (RTS) algorithm, which although relatively simple, has been shown to perform well on the QAP (Taillard, 1991). The RTS procedure is modified by changing the stopping criterion and the tabu tenure parameters for each processor participating in the algorithm. The modification to the stopping criteria and tabu tenure parameters as well as the choice of the diversification operator were made to the RTS as they proved beneficial in a previous study (James et al., 2006). The TS operators share search information by means of a global memory which holds a reference set of solutions and is managed in a way that promotes both strategic diversification and intensification. These mechanisms make CPTS a cooperative multi-thread (Type 3) parallel algorithm according to the classification outlined in the previous section.

The main feature of the cooperative parallel tabu search (CPTS) is to take advantage of parallel computing in the implementation of adaptive memory programming (Glover and Laguna, 1997). The CPTS algorithm makes use of adaptive memory at various levels and stages of the search. Short-term memory is implemented by the tabu restrictions and aspiration criteria considered in each individual tabu search operator and it is local to each execution of the procedures. Although tabu restrictions implicitly implement some level of intensification and diversification and are commonly utilized in TS algorithms for the QAP, higher levels of these strategies are usually necessary to find the best solutions. In CPTS this is achieved by appropriately aggregating information derived from the local short-term memories into a global longer-term memory structure that can be shared by all tabu searches. Intensification and diversification are promoted via the global reference set as follows. Half of the global reference set always contains the global best solution found over all searches. This ensures that at least half of the concurrent tabu searches are operating on a copy of the best solution found at the point the execution of the individual TS was started on a given processor, thus promoting the search in vicinity of high quality solutions. Diversification is obtained by means of an extra strategic diversification operator. Each member of the global reference set has an updated flag that indicates whether or not the item was changed on the last attempt to update that location in the reference set. If the reference set item has been updated, then the next TS operator that accesses the item will use an exact copy of the solution stored in that location. However, the next TS operator will have different tabu search parameters (upper and lower bounds from which to draw the tabu tenure for an element) to induce the search to take a different trajectory than that of the TS operator that found the improved solution. This intensifies the search from that good solution. If the location was not updated by the last TS operator, then the current solution in that reference set location is first passed to the diversification operator before the next TS operator receives it. The pseudocode for the algorithm is shown in Fig. 1.

The algorithm considers as many reference set solutions as the number of the processors available and the parallel execution is conceived by assigning each processor a specified reference set location. Processors and reference set locations are identified by their corresponding number and index location. The algorithm begins by initializing the reference set. This is accomplished by running one independent tabu search on each of the processors being used so that each process updates the solution of the reference set whose index corresponds to its processor number. For the initialization phase, the stopping conditions for the individual TS operators are set to be shorter than in the cooperative parallel search of CPTS. The initialization phase provides the opportunity to seed the reference set with good solutions while maintaining some level of diversity. At initialization, all the update flags are set to be true so that the diversification operator will not be called the first time each solution is used to seed a TS process in the cooperative parallel search loop.

After initialization, each processor starts a TS process using the solution from its associated next index of the reference set and this process is circularized for each subsequent TS each individual processor executes. When a processor's individual TS terminates execution, it updates the reference set item it was working from if the solution the processor found is better than what is currently in that location and sets the update flag. If a processor updates a reference set location, an additional check is run to see if the new solution is a global best. In such a case, the new global best is threaded through the reference set so that half the solutions in the reference set become the new best solution. Any time a processor switches to a new reference set location it first checks the status of the update flag. If the update flag is set to false, the diversification operator is called and a diversified copy of the solution in that location is used to seed the TS. The cooperative parallel search phase is executed by all participating processors for a globally defined number of nMax iterations at which point the CPTS algorithm terminates. The parameter nMax in Fig. 1, which defines the total number of tasks assigned to all processors by the parallel loop, is a user defined value. The final solution of CPTS's run is the best solution contained in the reference set at the termination of the algorithm. The following subsections describe the major components of the CPTS algorithm and provide relevant implementation details.

4.1. The tabu search

The tabu search used in CPTS is a modification of Taillard's robust tabu search procedure (Taillard, 1991). The pseudocode for the procedure is given in Fig. 2. The

```
Input parameters: number of processors (nProcessors)
Global Data:
Distance/Flow Matrices, Reference Set, Tabu Tenure Parameters, Updated Flags, Steps

Parallel Initialization
Private Data: permutation, cost, myPid
Parallel Loop: For each loopIndex =1 to nProcessors
    Obtain its processor identification number: myPid
    Generate a random permutation: myPermutation
    Update the (myPid) reference set location with myPermutation
    Perform TS(myPid, myTabuTenure, myPermutation, myMaxFailures)
    Write the improved myPermutation into the myPid reference set location
    Set flagUpdate to TRUE
End Parallel Loop (Synchronizes all the tabu search processes)

Parallel Search: schedule (dynamic)
Private data: permutation, cost, myPid, myMaxFailures
Parallel Loop: For each loopIndex =1 to nMax
   Lock Critical Section
      myIndex++ (circularize counter if at end)
      Get its permutation from current (myIndex) reference set location: myPermutation
      Get its own value tabu search parameters: myTabuTenure, myMaxFailures
      If processors own reference set location (myIndex=myPid) or flagUpdate=FALSE then
           Perform Diversification (myPermutation, myStepSize)
           Update diversification step size for that reference set location
      End If
   Unlock Critical Section
   Perform TS(myPid, myTabuTenure, myPermutation, myMaxFailures)
   Lock Critical Section
         Set flagUpdate to FALSE
         If myPermutation better than what is in current index (myindex) location
             Update current reference set location with myPermutation
             Set flagUpdate = TRUE
         End If
         Loop: For each loopIndex=1 to nProcessors  (Increment loopIndex by 2)
             If myPermutation better than what is in loopIndex reference set location  then
                  Replace solution in loopIndex with the newly best solution, myPermutation
                  Set flagUpdate = TRUE
             End If
         End Loop
   Unlock Critical Section
End Parallel Loop (Synchronizes algorithm)

Find Overall Best Solution in Reference Set and Print
```

Fig. 1. Cooperative parallel tabu search pseudocode.

```
Procedure TS (pid, tabuTenure, bestPermutation, maxFailures)

Set currentPermutation to bestPermutation
Set numFailures = 0
Loop: While numFailures < maxFailures do
          Evaluate the neighborhood the currentPermutation and select the exchange
                    that is not tabu or is tabu but meets all aspiration criteria
          Update tabu restrictions
          Perform the exchange on currentPermutation
          If strictly improving
                    Set bestPermutation to currentPermutation
                    Set numFailures = 0
          Else
                    numFailures++  (Increment numFailures)
          End If
End Loop

End Procedure
```

Fig. 2. Pseudocode for the tabu search component.

solution encoding used consists of assigning an integer value representing each location from 1 to n. The array indexes represent the facilities. A permutation s can then be illustrated as:

$$s = (2, 4, 10, 7, 5, 3, 1, 6, 9, 8).$$

The classical two-exchange (or swap) neighborhood is employed. A "move" consists of exchanging two elements of the permutation. This neighborhood definition is commonly applied to the QAP.

The tabu search begins by setting the working solution to its initial best solution obtained from the reference set.

Each iteration examines all possible moves in the neighborhood of the working permutation choosing the best non-tabu or aspired exchange. This exchange is performed and the modified permutation becomes the new working solution. The procedure keeps track of the best solution the TS procedure has found which is updated whenever the new working solution is better than the current best. The procedure terminates when its local best solution is not updated for a defined number of iterations.

Short-term memory is maintained by means of tabu restrictions and associated aspiration criteria. When an exchange is made on the working solution, a tabu tenure is drawn from a defined range for each element of the exchange. Specifically, the length each element will remain tabu is drawn randomly from between the upper and lower tabu tenure bounds. In CPTS, these bounds are globally defined and are different for each processor. The bounds for each processor were randomly chosen within the original upper and lower bounds used in RTS. A subsequent move consisting of the exchanged elements is not allowed for the length of the tabu tenure unless the aspiration criteria are met. The aspiration criteria first checks to ascertain if the forbidden exchange results in a solution that is better than the best solution stored for that TS. The exchange may also be considered if the tabu status of at least one of the elements is less than a predefined ceiling. If one of these two conditions holds then a further check is conducted to test if the swap is the first forbidden exchange considered during the current iteration. If it is, the exchange is accepted. Otherwise the forbidden exchange must also be better than any exchange that has been considered during the current iteration of the TS. These aspiration criteria are those used in RTS.

4.2. The diversification operator

In CPTS, the diversification procedure starts from a seed solution obtained from the global reference set and is run only if the update flag for the corresponding seed solution is false. The diversification operator employed was suggested by Glover et al. (1998). The pseudocode for this procedure is given in Fig. 3. The operator uses a step to define a starting position and an increment within the permutation. This step determines which elements are chosen from the original permutation to create the new solution. In CPTS, a step for each reference set item is stored and incremented every time the diversification procedure is called. The step values start at 2 and are incremented by 1 until equal to the size of the problem (n) and then reset to 2 if necessary.

```
Procedure Diversification (permutation, step)

position = 1
Set seedPermutation to permutation
Loop: For start = step to 1 (decrement start by 1)
      Loop: For j = start to $n$ (increment j by step)
            permutation(position) = seedPermutation(j)
            position++  (increment position by 1)
      End Loop
End Loop

End Procedure
```

Fig. 3. Pseudocode for the diversification operator.

To illustrate the procedure, assume the following seed permutation and a step size of 2:

$$s = (2, 4, 10, 7, 5, 3, 1, 6, 9, 8).$$

The procedure sets the *start* variable to 2 and steps through the permutation, resulting in the following partial permutation:

$$d = (4, 7, 3, 6, 8, \_, \_, \_, \_, \_).$$

The start variable is then decremented by 1 and the permutation is stepped through again, obtaining the final diverse permutation:

$$d = (4, 7, 3, 6, 8, 2, 10, 5, 1, 9).$$

The new diversified permutation, d, is then used to seed the TS rather than the original solution in the global reference set.

4.3. Reference set management

The reference set and associated control structures define a global memory used by CPTS to manage and allow the exchange of search information between the processors. In globally accessible memory, four items are kept and the access to these items is managed through use of the parallel programming implementation. The size of the reference set is defined by the number of processors used. Each reference set item contains one permutation, its associated objective function value and all the control structures necessary to manipulate the search drawing on that reference set location. These include an update flag for the permutation, the last diversification step size that was used on the corresponding reference set location, the reference set index that the processor owning the location is currently using, and the processor tabu search parameter settings, namely the tabu tenure ranges and the maximum number of failures.

4.3.1. Reference set management in the initialization

At initialization, each TS algorithm randomly creates one seed permutation, works on its local permutation until it reaches its stopping condition, and updates its own reference set location. The processors local permutation and its maximum failure value are both private. In this phase, no critical section is necessary to enforce mutual exclusion in the access to the reference set locations as the processors will only update the location corresponding to their processor identification number. The stopping condition is met when the TS fails to update its own best solution for a defined number of iterations. Even though this stopping condition ensures that all the initial TS processes will terminate after a defined number of failures, at any point the procedure finds a new best solution, this counter is reset. Consequently, there is no guarantee that all TS processes will run the same length of time. For this reason, the maximum failures value in the initialization is set to be static parameter and shorter than in the cooperative parallel search to minimize idle processors at the first synchronization point. The impact of the synchronization is discussed in more detail in Section 5.3. Upon termination of the initial TS operators, each processor updates the reference set location that corresponds to its processor identification number and denotes that the location has been updated. The initialization phase must finish completely before the cooperative parallel search of CPTS is allowed to proceed. This synchronization point ensures that all locations in the reference set are seeded before the asynchronous parallel execution of the various tabu searches begins, hence preventing any of the processors from attempting to operate on a null solution.

4.3.2. Reference set management in the cooperative parallel search

Once all the processors have completed their initial TS operators and the reference set is fully initialized, the cooperative parallel search of CPTS begins. Each processor shifts one position in the reference set with respect to its current location and retrieves the corresponding reference set solution. The last processor obtains its solution from the first reference set location. A private index is maintained by each processor that allows it to keep track of its current location in the reference set. Each processor runs a TS on its new working solution using its unique tabu parameters. Although in the initialization phase the maximum failure value was a static parameter, in the cooperative parallel search phase it is a dynamic parameter drawn from a range of values. This parameter may therefore be different for each TS operator executed.

The range is set in such a way that the maximum failures parameter drawn will be equal to or greater than what was used for the initialization phase. Varying the maximum failure parameter was done purposefully to aid in the maintenance of the reference set as well as load balancing and task management.

As soon as a TS operator terminates on a processor, it attempts to update the location in the reference set corresponding to its current index and then increments the index value to refer to the next reference set location of the circularization process.

If the processor's local best solution is better than the solution in the reference set at the processor's index location it updates the solution. It also sets the update flag to true. If a processor updates a reference set location then another check is triggered. If the processor's local best solution represents a global improvement, this newly best solution is propagated through half the processors by copying it to every odd index location of the reference set. This ensures that at least half the processors will work off of this best solution on their next iteration. On the other hand, if the new processor best solution is not a local improved solution the update flag is set to false so that the next processor to read that memory location will first diversify the solution before operating on it.

Only one processor at a time may update the reference set. This maintains the data integrity but is not a synchronization point as all processors do not have to reach the same point before continuing. If the reference set is locked the processor would wait for the lock to be released and then perform the update. Since each processor terminates its TS on a variable condition, the TS operators are less likely to finish at the same time. This makes the design of the updating scheme less of a bottleneck.

Once a TS processor is finished on a processor, it will then proceed with its next TS operator based on the conditions described above. This process continues until a defined number of iterations have been performed. When the last TS operator terminates on all processors, the best solution in the reference set is determined as the output of the CPTS algorithm.

5. Platform design considerations

5.1. Parallel architecture and programming implementation

The CPTS algorithm was written to take advantage of the parallel platform on which it was implemented. CPTS was developed and run on an SGI Altix, using an Intel C compiler and the OpenMP (Chandra et al., 2001) parallel programming implementation. The SGI Altix is a CC-NUMA shared-memory machine. This means that the memory is globally addressable and the architecture provides the illusion of equidistant memory. The memory is not technically equidistant but the platform manages access to provide this appearance. The shared-memory architecture allows for memory stores to be globally addressable.

OpenMP uses compiler directives and libraries to parallelize code written in C. OpenMP and the architecture manage the access to and location of the memory requiring little, or in some cases no, involvement from the programmer. The memory is globally accessible by all processors unless specified otherwise. If data dependencies exist, access to memory that is globally accessible must be managed by the programmer, or marked as private, to prevent erroneous reads/writes. The platform utilized in this study can be used to easily parallelize code with moderate programming skill due to the high level of abstraction in OpenMP. However, OpenMP does provide a certain level of control over parallel design considerations if desired.

5.2. Memory management and algorithm design considerations

The architecture and the parallel programming implementation under which CPTS implemented influenced the design considerations of the algorithm. CPTS was developed to take advantage of the platform being utilized and design tradeoffs were carefully considered during the design of the algorithm. The rest of this section will outline several of the most well-known parallel design considerations as well as the design choices made for CPTS and the tradeoffs for those choices.

Memory management is the first consideration. In CPTS, there are three types of memory stores that need to be handled. The first are inputs that are strictly read by the processors and are never updated. Since these data elements never change, they can be handled by the platform. The second type of data is the local variables (or memory locations) that need to be private to each processor. The third type of data is the shared reference set that is used to exchange search information. This data must be globally accessible and will be updated by all processors. These last two data categories must be explicitly handled by the programmer.

5.2.1. First data type

In the current study, the use of a shared-memory platform allows for globally accessible flow and distance matrices. Since these matrices are only read and not written to, uncontrolled global access can be given to the algorithm for these data stores and the platform can manage their access. Each processor has its own tabu tenure parameters. This was accomplished by means of an array referenced by processor identification number. The parameter array is never written to, so it can be globally accessible as well and referenced through use of an index. These data locations correspond to the first type of data.

5.2.2. Second data type

Each TS on each processor needs a memory location to keep a working solution, a current best solution, the index of its current position in the reference set, and its maximum failure value. This data must only be used by the processor that owns it. While this is also true for the tabu tenure parameters in the first data type, the key difference between the two data types is that the second data type elements will be written as well as read. These data elements will change each time a TS finishes and thus are temporary variables. Globally accessible data elements that will be written are dangerous due to possible data dependencies. Hence, these variables need to be local to that processor and correspond to the second type of data. These memory locations need to be restricted to only be accessible to a particular processor. In this manner, it is assured that only the processor that owns these elements will modify them. Technically, this means that each processor creates its own copy of these variables. This is accomplished in OpenMP by making these variables private.

5.2.3. Third data type

The third type of data includes the reference set variables. These variables need to be globally accessible to allow for the exchange of information between processors. However, they are also updated so the access to these memory locations needs to be explicitly controlled by the programmer. This is done using a *critical* directive in OpenMP, which allows only one processor at a time to update or read the shared data. This creates a small delay since the memory location is locked until the processor currently using the shared data completes the corresponding critical section. This is necessary to maintain the integrity of the data store in case more than one processor attempts to read/write the same location at the same time.

5.3. Parallel search

5.3.1. Synchronous parallel search

In the initialization phase, each processor updates one location of the reference set. In this phase each processor runs a tabu search operator concurrently. The parallelization for this phase is accomplished by a "parallel do" directive in the first loop of Fig. 1. A hard synchronization is performed at the end of this phase which means that all processors must finish their tabu search operator before the rest of the algorithm executes. The synchronization is accomplished by performing the initialization phase in a separate parallel do directive which spawns as many tabu search processes as the number of available processors and then waits until the processes terminate before continuing. The synchronization is necessary since the next passes do not reference the processors own location but rather rotate through the reference set within the cooperative parallel search loop. This condition requires that there be a solution in a location before it is used which cannot be guaranteed without enforcing synchronization at the end of the initialization phase. Even though this synchronization may cause some processors to be idle while all TS operators finish execution, this was an algorithm design choice necessitated by the desired functionality of the algorithm. Although some initialization of the reference set must be performed, the preferred quality and diversity of these initial solutions is left to the metaheuristic designer. We chose to start the base parallel algorithm from relatively good seed solutions. To obtain an improvement in overall solution quality of the initial seed solutions, some idleness in the processors was tolerated. The compromise made in the design was to reduce the number of iterations of the initial TS operators thereby reducing the idleness of the processors incurred at this synchronization point. This becomes a more important tradeoff as the size of the QAP increases.

5.3.2. Asynchronous parallel search

The cooperative parallel search is also parallelized using a parallel do construct. The critical sections discussed above do not enforce any type of synchronization in the cooperative parallel search. These directives simply protect the integrity of the reference set. This is important in the cooperative parallel search because the iterations continue regardless of the order the TS operators start or finish. The platform controls the order in which the processors update if there is a tie or a queue. Since the platform controls access through the critical section, the iterations are allowed to proceed in any order. The execution time of the TS operator running on any processor is variable. The previous conditions result in updating occurrences that should be mentioned. First, some reference set locations may be updated (read or written to) multiple times before others. Secondly, the locations may be updated in any order by any processor.

5.4. Process scheduling and load balancing considerations

5.4.1. Explicit load balancing

The choice of not enforcing synchronous access to the reference set was done purposefully. This design choice helps maximize the processors utilization since all processors can be busy except when waiting to update the reference set. Since all processors are not synched at each set of TS operators, some searches may start from a lower quality solution. A TS operator is likely to execute longer if the solution keeps improving. This may cause good solutions to enter the reference set more slowly. The variability of the maximum failure parameter in the cooperative parallel search loop was done to counteract this by causing some TS operators to terminate more quickly than others. The propagation of the better solutions through half of the reference set as previously mentioned was conducted to intensify the search. This intensification also counteracts the longer runs of the more successful TS operators by quickly propagating good solutions through the reference set.

5.4.2. Implicit load balancing

Due to the conditional stopping criterion used in the TS operators, some intervention was required to improve the load balancing on the platform. By default OpenMP uses static scheduling for the processors. This type of scheduling divides the iterations up at the beginning of the cooperative parallel loop. In CPTS an iteration (or chunk) consists of everything within the "Parallel Search" section of Fig. 1. If some processors run longer than others, static scheduling will cause the wall clock time to increase since the algorithm will not terminate until the last TS is run. Dynamic scheduling was applied in the cooperative parallel search so that each processor was assigned smaller chunks of work. The default chunk size of one was used for the dynamic scheduling option in CPTS. The processors were then given more work as they finished the tasks, one task (or loop) at a time. The compiler handles the distribution of work. This incurs some additional management overhead which was negligible compared to the unequal loads if this directive was not used. Refer to Chandra et al. (2001) for more information on the dynamic scheduling option in OpenMP.

These considerations illustrate examples of the tradeoff between load balancing, task management, and synchronization. In an ideal parallelization, all processors should be working at all times to maximize utilization. Requiring all processors to synchronize causes some of the processors to idle for a time, dependent on the variation among the run times of the various tasks. Since the stopping criterion is conditional, the run times may vary resulting in idle processors. This was deemed acceptable to initialize the reference set when the number of iterations per TS was purposefully shortened, but became too costly as the iteration number in both the main loop and for each processor's TS increased. In the latter case, asynchronous updates were allowed to improve the load balancing on the machine. In the cooperative parallel loop it was preferable to have the TSs operate longer on each solution as well as make several passes through the reference set. The decision to vary the maximum failure parameter was made for solution quality reasons. These decisions required explicit control of the task management and caused minimal additional overhead but improved the load balancing and thus the processor utilization and the run time.

5.5. Scalability

CPTS is scalable in two ways if more processors are available. First, the number of iterations of the cooperative parallel search could be increased. This would correspond to operating on each item in the reference set more times. In this scenario, the reference set would stay the same size or increase but not proportionally to the number of processors used. At the same time, the number of tabu searches that operate on each reference set item would increase. This proposed scaling would result in each reference set item being operated on more times. In this manner, the dynamics of the algorithm would change slightly as the quality of the solutions being operated on may not be as good for every TS. However, it would also allow for more tabu searches to be run, as well as more perturbation to occur by the use of different bounds on each processor and the diversification operator, which is likely to improve the solution quality.

Secondly, the maximum failure parameter could be increased. This would cause the individual TS operators to run longer. With more processors, the size of the reference set and the maximum failures could be increased. If the number of iterations through the reference set were reduced in this scenario, the algorithm could be scaled in this manner. The absolute minimum run time of CPTS in this scenario would be equal to the wall clock time cost of one TS operator on each reference set item. This would turn the CPTS algorithm into an independent search strategy. Compromises can be made between the run time of the individual TS operators and the number of iterations in the cooperative parallel search loop. This would allow the algorithm to be scaled and still be cooperative. It may be possible to see improvements in both solution quality and run time with either of these scaling scenarios.

6. Computational results and discussion

The cooperative parallel tabu search algorithm (CPTS) was run on a set of 41 test instances obtained from QAPLIB (Burkard et al., 1997). The tai∗a, tai∗b, and sko∗ test instances were chosen as they are the most popular test sets run by competing algorithms in the literature. The tai∗a and tai∗b test instances are commonly run by about half of the current algorithms in the literature and the sko∗ problems by the other half. In order to be able to compare against the most recent parallel algorithms from the literature, a set of selected test instances from QAPLIB were also run. This expanded our runs to provide results for a broader selection of problems. Additionally, we provide results for 60 tai∗e test instances. These instances are newly introduced to the literature (Drezner et al., 2005) and are demonstrated to be difficult for metaheuristics.

The technical specifications for the computing architecture and programming languages used were provided in Section 5.1 The parameters were chosen such as to obtain wall clock run times similar to the run times of popular sequential algorithms for the QAP. All parameters were fixed for all runs and test problems except the seed used to generate initial feasible solutions. Table 1 shows the parameter settings used in the algorithm.

Table 1
Parameter settings

Parameter	Value[a]
Number of processors	10
Maximum failures[a] (initialization)	$100 * n$
Maximum failures[a] lower bound (cooperative search)	$100 * n$
Maximum failures[a] upper bound (cooperative search)	$200 * n$
Maximum number of iterations (nMax)	$50 * n$
Diversification step size (incremented by 1 at each use)	2 (initial value)

[a] Where n = number of facilities/locations and maximum failures = numFailures in Fig. 2.

6.1. Computational analysis of the CPTS algorithm

Table 2 summarizes the computational results obtained by the CPTS algorithm on the 41 problems tested from QAPLIB. Ten (1.3 GHz) Intel Itanium processors (1.3 GHz) were used. Computational resource restrictions limited the processor use to 10. All results are averages over 10 runs of the algorithm on each test instance. The table presents for each instance the best known solution (BKS) obtained from QAPLIB, average percent deviation from the BKS (APD), the number of times the best known solution was found by CPTS out of the 10 runs (#BKS), the average wall clock time for the algorithm (Wall Clock

Table 2
Computational results and times for CPTS

Problem	BKS	APD	#BKS	Wall clock time	CPU time	Problem	BKS	APD	#BKS	Wall clock time	CPU time
Skorin-Kapov instances						*Taillard symmetric instances*					
sko42	15812	**0.000**	**10**	5.3	52.2	tai20a	703482	**0.000**	**10**	0.1	1.5
sko49	23386	**0.000**	**10**	11.4	110.9	tai25a	1167256	**0.000**	**10**	0.3	5.2
sko56	34458	**0.000**	**10**	21.0	209.9	tai30a	1818146	**0.000**	**10**	1.6	10.6
sko64	48498	**0.000**	**10**	42.9	429.6	tai35a	2422002	**0.000**	**10**	2.3	20.2
sko72	66256	**0.000**	**10**	69.6	695.0	tai40a	3139370	0.148	1	3.5	37.7
sko81	90998	**0.000**	**10**	121.4	1212.5	tai50a	4938796	0.440	0	10.3	104.6
sko90	115534	**0.000**	**10**	193.7	1935.3	tai60a	7205962	0.476	0	26.4	265.1
sko100a	152002	**0.000**	**10**	304.8	3042.4	tai80a	13515450	0.570	0	94.8	947.9
sko100b	153890	**0.000**	**10**	309.6	3091.4	tai100a	21059006	0.558	0	261.2	2608.7
sko100c	147862	**0.000**	8	316.1	3154.4						
sko100d	149576	**0.000**	**10**	309.8	3091.9						
sko100e	149150	**0.000**	**10**	309.1	3085.6						
sko100f	149036	0.003	4	310.3	3096.4						
Average		0.000	9.4	178.8	1785.2	Average		0.244	4.6	44.5	444.6
Selected instances						*Taillard asymmetric instances*					
els19	17212548	**0.000**	**10**	0.1	1.4	tai20b	122455319	**0.000**	**10**	0.1	1.7
bur26d	3821225	**0.000**	**10**	0.4	6.3	tai25b	344355646	**0.000**	**10**	0.4	6.3
nug30	6124	**0.000**	**10**	1.7	11.0	tai30b	637117113	**0.000**	**10**	1.2	13.8
ste36c	8239110	**0.000**	**10**	2.5	28.9	tai35b	283315445	**0.000**	**10**	2.4	25.9
lipa50a	62093	**0.000**	**10**	11.2	112.6	tai40b	637250948	**0.000**	**10**	4.5	48.3
tai64c	1855928	**0.000**	**10**	20.6	206.5	tai50b	458821517	**0.000**	**10**	13.8	138.1
wil100	273038	**0.000**	**10**	316.6	3161.3	tai60b	608215054	**0.000**	**10**	30.4	305.0
tho150	8133398	0.013	0	1991.7	19885.4	tai80b	818415043	**0.000**	9	110.9	1106.3
tai256c	44759294	0.136	0	7377.8	73298.8	tai100b	1185996137	0.001	8	241.0	2403.8
						tai150b	498896643	0.076	0	7377.8	73298.8
Average		0.023	7.0	1127.2	11217.5	Average		0.000	9.7	45.0	449.9

Time), and the actual computational time (CPU Time) used, both in minutes (mm.ss).

All results and computational times for the sequential algorithms used for comparison were obtained from the literature, except for JRG-DivTS which is our own. It should be noted that the hardware on the current architecture is at least marginally better than all of the sequential architectures obtained from the literature used for comparison (comparing processor to processor only). Also, CPTS is a parallel algorithm and used 10 processors, so the total computational time is greater than a sequential algorithm even if the run times are not. Fig. 4 depicts the difference between the wall clock time of CPTS for the test set versus the actual total computational time expended using a logarithmic scale for the instances of Table 2. The parameter settings could be changed to reduce the run times for larger problems. Since CPTS is a parallel algorithm, the run times could also be reduced by implementing the scaling techniques described earlier if more processors became available. The actual computational time for the algorithm increases to over 3000 minutes for most problems with 100 facilities/locations while the wall clock time is less than 500 minutes for all problems. The wall clock time for these problems is within a reasonable range, but the actual computational time used would not be feasible for a sequential algorithm.

The results reported in Table 2 show that CPTS provides high quality solutions for most test instances in acceptable wall clock times. For the symmetrical sko∗ test set, CPTS solves 12 out of the 13 problems to 0.000 percent deviation on average from the BKS. For 11 of these instances, the algorithm finds the BKS on all 10 of the runs. For sko100c, the BKS is found 8 out of 10 times but the average percent deviation from the BKS is still 0.000. For the last instance, sko100f, CPTS obtains an average percent deviation of 0.003 and finds the BKS 4 out of 10 times. This is the worst performing instance for this test set, though the deviation is still very small. Also, CPTS obtains exceptional solution quality for the selected test instances, producing solutions that are on average 0.023% from the BKS over all instances in this set. For all but 3 of the 10 selected problems, CPTS finds the BKS for all of the 10 runs. The remaining three test instances were the largest instances run. The percentage deviations for tai150b and tho150 were 0.076 and 0.013, respectively. A deviation of 0.136 was obtained for tai256c, which was the largest test instance. Notably, the wall clock time was less than 320 minutes for all but these largest three instances. These runtimes could be further reduced with the addition of more processors. Scaling the algorithm could result in improvement in solution quality as well as solution time. The wall clock times for the largest 3 instances, tai150b, tho150, and tai256c were 1549, 1992, and 7378 minutes. Though they compare favorably with the performance achieved by other methods, these times may be a somewhat larger than desired and scaling the algorithm may be preferable.

CPTS also performs well on the Taillard test instances (tai∗a and tai∗b). On the symmetric test instances (tai∗a), which are among the most difficult, CPTS finds the BKS on the smaller instances for all runs. On the larger instances (of size 40 or greater), CPTS does not perform as well but the results are still of acceptable quality. Given longer runs CPTS may perform better on these instances. As previously mentioned, the scalability of the algorithm would allow for these improvements with little or no additional increase in actual wall clock time. On the asymmetric test set (tai∗b), CPTS performs well obtaining 0.000 percent deviations from the BKS for all but one problem. For tai80b instance,

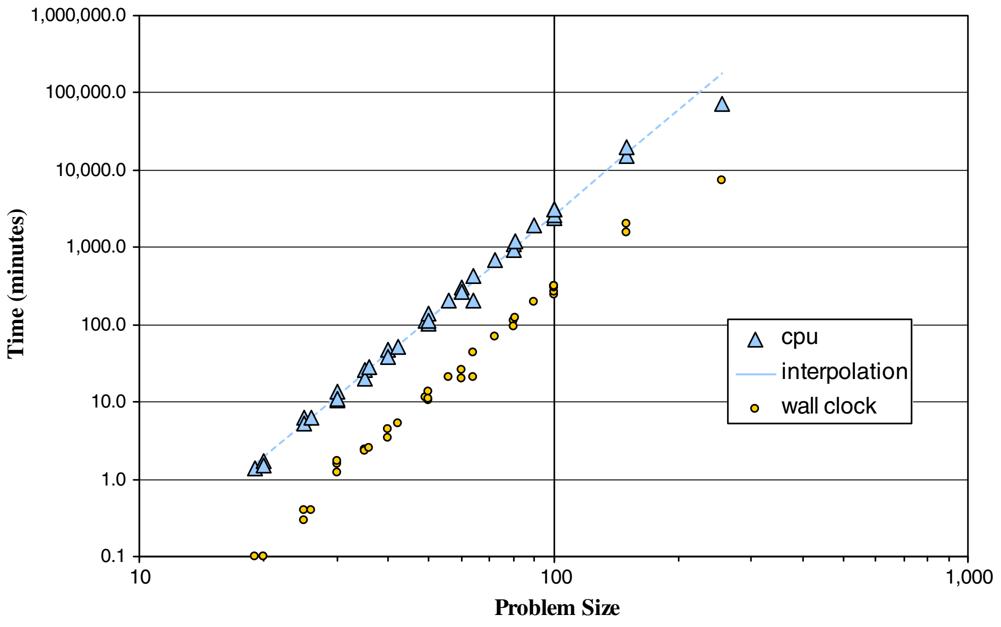

Fig. 4. Wall clock time versus CPU time.

the algorithm finds the BKS on 9 out of 10 runs with a deviation of 0.000%. The BKS was obtained 8 out of 10 times for the tai100b instance with an average deviation of 0.001%.

Table 3 reports results for the 60 tai∗e problems and provides some of the first heuristic results for these problems to the literature. Results were obtained for these problems using Intel Itanium processors (1.6 GHz). The BKS values for this test set were obtained from Taillard (2007). For the tai∗e problems of size 27, CPTS finds the best known solution 100% of the time in about 4 seconds. For the size 45 problems, the average percent deviations appear to be more sporadic. However, the average number of times the BKS is found is 9.15. This shows that CPTS finds the best known solution almost all of the time. In the cases where the BKS is not found, the local optima found is somewhat larger than the BKS resulting in relatively larger APDs. The algorithm produces relatively large APDs on the problems of size 75, indicating that indeed these problems are very difficult for metaheuristic methods as CPTS found higher quality solutions for problems of size 100 or larger in other test sets.

6.2. Comparative analysis with alternative sequential and parallel algorithms

Additional analysis was carried out in order to provide comparisons of CPTS to some of the best performing sequential and parallel algorithms from the literature. The following algorithms are considered:

– A Multi-Start Tabu Search Algorithm – JRG-DivTS (James et al., 2006).
– An Ant Colony Optimization/Genetic Algorithm/Local Search Hybrid – TL-ACO/GA/LS (Tseng and Liang, 2005).
– A Genetic Algorithm Hybrid with a Strict Descent Operator – D-GA/SD (Drezner, 2003).
– A Genetic Algorithm Hybrid with a Simple Tabu Search Operator – D-GA/S-TS (Drezner, 2003).

Table 3
Computational results for the Taillard 27, 45, and 75 and instances

Problem	BKS	APD	#BKS	Wall clock time	CPU time	Problem	BKS	APD	#BKS	Wall clock time	CPU time
Taillard 27e instances						*Taillard 45e instances*					
tai27e01	2558	0.000	10	0.4	6.2	tai45e01	6412	0.000	10	4.3	43.7
tai27e02	2850	0.000	10	0.4	6.2	tai45e02	5734	0.000	10	5.5	51.0
tai27e03	3258	0.000	10	0.4	6.1	tai45e03	7438	0.000	10	5.0	49.2
tai27e04	2822	0.000	10	0.4	6.3	tai45e04	6698	0.021	9	5.2	51.0
tai27e05	3074	0.000	10	0.4	6.3	tai45e05	7274	0.168	7	5.2	51.2
tai27e06	2814	0.000	10	0.4	6.1	tai45e06	6612	0.000	10	5.3	50.8
tai27e07	3428	0.000	10	0.4	6.4	tai45e07	7526	0.000	10	4.6	45.3
tai27e08	2430	0.000	10	0.4	6.3	tai45e08	6554	0.000	10	4.6	48.4
tai27e09	2902	0.000	10	0.4	6.7	tai45e09	6648	0.361	7	4.5	46.7
tai27e10	2994	0.000	10	0.4	6.2	tai45e10	8286	0.019	7	4.5	48.1
tai27e11	2906	0.000	10	0.4	6.2	tai45e11	6510	0.000	10	4.7	48.7
tai27e12	3070	0.000	10	0.4	5.5	tai45e12	7510	0.053	8	4.5	47.0
tai27e13	2966	0.000	10	0.4	6.6	tai45e13	6120	0.000	10	5.3	51.2
tai27e14	3568	0.000	10	0.4	6.3	tai45e14	6854	0.000	10	5.2	52.4
tai27e15	2628	0.000	10	0.4	6.3	tai45e15	7394	0.141	8	4.5	47.0
tai27e16	3124	0.000	10	0.4	6.5	tai45e16	6520	0.000	10	5.2	49.8
tai27e17	3840	0.000	10	0.4	6.4	tai45e17	8806	0.055	8	4.5	47.5
tai27e18	2758	0.000	10	0.4	6.2	tai45e18	6906	0.000	10	5.0	49.0
tai27e19	2514	0.000	10	0.4	6.4	tai45e19	7170	0.061	9	4.7	48.5
tai27e20	2638	0.000	10	0.4	6.3	tai45e20	6510	0.000	10	4.6	48.3
Average		0.000	10	0.4	6.3	Average		0.044	9.15	4.8	48.7
Taillard 75e instances						*Taillard 75e instances (cont)*					
tai75e01	14488	5.915	0	47.7	475.2	tai75e01	15250	3.721	0	44.2	442.5
tai75e01	14444	8.751	0	47.3	472.4	tai75e01	12760	8.196	0	47.1	469.9
tai75e01	14154	2.681	0	48.2	482.0	tai75e01	13024	5.123	0	45.3	450.0
tai75e01	13694	3.705	0	46.4	464.7	tai75e01	12604	3.659	0	44.8	448.1
tai75e01	12884	2.412	1	46.4	464.1	tai75e01	14294	2.976	0	46.3	461.7
tai75e01	12554	4.005	0	48.5	485.2	tai75e01	14204	3.389	0	46.4	463.1
tai75e01	13782	8.015	0	43.2	431.7	tai75e01	13210	3.653	0	48.6	483.5
tai75e01	13948	8.523	0	42.7	425.7	tai75e01	13500	7.176	0	47.3	472.5
tai75e01	12650	4.798	0	48.7	485.9	tai75e01	12060	2.861	1	46.1	458.2
tai75e01	14192	5.740	0	46.4	463.1	tai75e01	15260	2.743	0	48.3	479.0
Average						Average		4.902	0.1	46.5	463.9

– A Genetic Algorithm Hybrid with Concentric Tabu Search Operator – D-GA/C-TS (Drezner, 2003).
– A Genetic Algorithm Hybrid with an Improved Concentric Tabu Search Operator – D-GA/IC-TS (Drezner, 2005).
– Three Tabu Search Variants (M-ETS-1, M-ETS-2, and M-ETS-3) (Misevicius, 2005).
– Two Genetic Algorithm Hybrids with a Tabu Search – M-GA/TS and M-GA/TS-I (Misevicius, 2003, 2004).
– An Independent Parallel Tabu Search – TB-MTS (Talbi and Bachelet, 2006).
– A Cooperative Parallel Tabu Search Hybrid with a Genetic Algorithm – TB-COSEARCH (Talbi and Bachelet, 2006).

Tables 4–6 summarize the computational results for the various algorithms on the three sets of instances used in Table 2.

The results in Table 4 show that CPTS ties or outperforms all other algorithms on all of the asymmetric test instances (tai∗b) with the exception of tai100b. For this instance the M-GA/TS-I algorithm of Misevicius (2004) slightly edges out CPTS by 0.001%. On the symmetric (tai∗a) instances, CPTS finds a 0.000 percent deviation for the first 4 instances of this set. Of the competing methods, only JRG-DivTS and M-GA/TS-I achieve this result. For the next 5 symmetric instances (considered large) the method that obtains the best solution is different in each case, each method finding only one of these solutions; hence none of the algorithms strictly dominates all the others. Only one of Misevicius's TS variants (M-ETS-2) provided a smaller percentage deviation for tai40a than CPTS. All three of these TS variants and one of the GA hybrids provide better solution quality for tai50a. M-GA/TS finds good solutions for the two largest instances (tai80a and tai100a), but CPTS strictly outperforms this algorithm on all other symmetric instances. M-GA/TS-I performs quite well overall obtaining 0.000 percent deviation for the smaller 4 instances and obtaining the best solution over all algorithms for tai100a. M-GA/TS-I does outperform CPTS on 3 out of the 5 large problems and ties for the 4 smaller instances. Similarly, all TS variants (M-ETS-1, M-ETS2, M-ETS-3) find better solutions than CPTS for these two largest instances, but these algorithms fail to find the BKS for several of the smaller instances. Although JRG-DivTS is very effective for the smaller instances it is slightly outperformed on the larger ones. TL-ACO/GA/LS does not find a BKS on any problem of the symmetric test set and the average solution quality of this algorithm is not on a par with any of the alternative algorithms.

Table 5 presents comparisons of CPTS to several sequential heuristic methods from the literature that have provided some of the best results for the Skorin-Kapov test set. As seen, CPTS ties or outperforms all algorithms on every test instance of the sko∗ test set.

Only JRG-DivTS and TL-ACO/GA/LS run both the Taillard problems and the Skorin-Kapov test set. The Skorin-Kapov problems are run exclusively by the GAs due to Drezner and the Taillard problems exclusively by the GAs

Table 4
CPTS comparisons on the symmetric and asymmetric Taillard instances

Problem	BKS	CPTS		JRG-DivTS		M-ETS-1	M-ETS-2	M-ETS-3		TL-ACO/GA/LS		M-GA/TS	M-GA/TS-I	
		APD	Time	APD	Time	APD	APD	APD	Time	APD	Time	APD	APD	Time
Taillard symmetric instances														
tai20a	703482	**0.000**	0.1	**0.000**	0.2	**0.000**	**0.000**	**0.000**	0.0			0.061	**0.000**	0.0
tai25a	1167256	**0.000**	0.3	**0.000**	0.6	0.037	**0.000**	0.015	0.1			0.088	**0.000**	0.1
tai30a	1818146	**0.000**	1.6	**0.000**	1.3	0.003	0.041	**0.000**	0.2	0.341	1.4	0.019	**0.000**	0.3
tai35a	2422002	**0.000**	2.3	**0.000**	4.4	**0.000**	**0.000**	**0.000**	3.7	0.487	3.5	0.126	**0.000**	0.6
tai40a	3139370	0.148	3.5	0.222	5.2	0.167	0.130	0.173	28.3	0.593	13.1	0.338	0.209	1.4
tai50a	4938796	0.440	10.3	0.725	10.2	0.322	0.354	0.388	116.7	0.901	29.7	0.567	0.424	5.0
tai60a	7205962	0.476	26.4	0.718	25.7	0.570	0.603	0.677	116.7	1.068	58.5	0.590	0.547	12.0
tai80a	13515450	0.570	94.8	0.753	52.7	0.321	0.390	0.405	200.0	1.178	152.2	0.271	0.320	53.3
tai100a	21059006	0.558	261.2	0.825	142.1	0.367	0.371	0.441	666.7	1.115	335.6	0.296	0.259	200.0
Average		0.244	44.5	0.360	26.9	0.199	0.210	0.233	125.8	0.812	84.9	0.262	0.195	30.3
Taillard asymmetric instances														
tai20b	122455319	**0.000**	0.1	**0.000**	0.2	**0.000**	**0.000**	**0.000**	0.0			**0.000**	**0.000**	0.0
tai25b	344355646	**0.000**	0.4	**0.000**	0.5	**0.000**	**0.000**	**0.000**	0.0			0.007	**0.000**	0.0
tai30b	637117113	**0.000**	1.2	**0.000**	1.3	**0.000**	**0.000**	**0.000**	0.1	**0.000**	0.3	**0.000**	**0.000**	0.0
tai35b	283315445	**0.000**	2.4	**0.000**	2.4	**0.000**	0.019	**0.000**	0.2	**0.000**	0.3	0.059	**0.000**	0.1
tai40b	637250948	**0.000**	4.5	**0.000**	3.2	**0.000**	**0.000**	**0.000**	0.2	**0.000**	0.6	**0.000**	**0.000**	0.1
tai50b	458821517	**0.000**	13.8	**0.000**	8.8	**0.000**	0.003	**0.000**	4.5	**0.000**	2.9	0.002	**0.000**	0.3
tai60b	608215054	**0.000**	30.4	**0.000**	17.1	**0.000**	0.001	0.003	22.5	**0.000**	2.8	**0.000**	**0.000**	0.7
tai80b	818415043	**0.000**	110.9	0.006	58.2	0.008	0.036	0.016	116.7	**0.000**	60.3	0.003	**0.000**	2.5
tai100b	1185996137	0.001	241.0	0.056	118.9	0.072	0.123	0.034	333.3	0.010	698.9	0.014	**0.000**	7.3
Average		**0.000**	45.0	0.007	23.4	0.009	0.020	0.006	53.1	0.001	109.4	0.009	**0.000**	1.2
Overall		0.122	44.7	0.184	25.2	0.104	0.115	0.120	89.4	0.407	97.1	0.136	0.098	15.8

Table 5
Comparative results on Skorin-Kapov instances

Problem	BKS	CPTS		JRG-DivTS		TL-ACO/GA/LS		D-GA/SD		D-GA/S-TS		D-GA/C-TS		D-GA/IC-TS	
		APD	Time	APD	Time	APD	Time	APD	Time	APD	Time	APD	Time	APD	Time
Skorin-Kapov instances															
sko42	15812	**0.000**	5.3	**0.000**	4.0	**0.000**	0.7	0.014	0.16	0.001	0.3	**0.000**	1.2		
sko49	23386	**0.000**	11.4	0.008	9.6	0.056	7.6	0.107	0.28	0.062	0.5	0.009	2.1		
sko56	34458	**0.000**	21.0	0.002	13.2	0.012	9.1	0.054	0.42	0.007	0.7	0.001	3.2	**0.000**	13.6
sko64	48498	**0.000**	42.9	**0.000**	22.0	0.004	17.4	0.051	0.73	0.019	1.2	**0.000**	5.9	**0.000**	26.2
sko72	66256	**0.000**	69.6	0.006	38.0	0.018	70.8	0.112	0.93	0.056	1.5	0.014	8.4	**0.000**	38.3
sko81	90998	**0.000**	121.4	0.016	56.4	0.025	112.3	0.087	1.44	0.058	2.2	0.014	13.3	0.003	63.1
sko90	115534	**0.000**	193.7	0.026	89.6	0.042	92.1	0.139	2.31	0.073	3.5	0.011	22.4	0.001	102.3
sko100a	152002	**0.000**	304.8	0.027	129.2	0.021	171.0	0.114	3.42	0.070	5.1	0.018	33.6	0.002	177.6
sko100b	153890	**0.000**	309.6	0.008	106.6	0.012	192.4	0.096	3.47	0.042	5.1	0.011	34.1	**0.000**	170.2
sko100c	147862	**0.000**	316.1	0.006	126.7	0.005	220.6	0.075	3.22	0.045	4.7	0.003	33.8	0.001	158.4
sko100d	149576	**0.000**	309.8	0.027	123.5	0.029	209.2	0.137	3.45	0.084	5.2	0.049	33.9	**0.000**	164.2
sko100e	149150	**0.000**	309.1	0.009	108.8	0.002	208.1	0.071	3.31	0.028	4.7	0.002	30.7	**0.000**	169.6
sko100f	149036	0.003	310.3	0.023	110.3	0.034	210.9	0.148	3.55	0.110	5.3	0.032	35.7	0.003	174.6
Average		**0.000**	178.8	0.012	72.1	0.020	117.1	0.093	2.1	0.050	3.1	0.013	19.9	0.001	114.4

Table 6
Comparative results to parallel heuristics on sample QAP instances

Problem	BKS	CPTS		TB-MTS	TB-COSEARCH
		APD	Time	APD	APD
Selected test instances					
els19	17212548	**0.000**	0.1	**0.000**	**0.000**
tai25a	1167256	**0.000**	0.3	0.736	0.736
bur26d	3821225	**0.000**	0.4	**0.000**	**0.000**
nug30	6124	**0.000**	1.7	**0.000**	**0.000**
tai35b	283315445	**0.000**	2.4	**0.000**	**0.000**
ste36c	8239110	**0.000**	2.5	**0.000**	**0.000**
lipa50a	62093	**0.000**	11.2	**0.000**	**0.000**
sko64	48498	**0.000**	42.9	0.004	0.003
tai64c	1855928	**0.000**	20.6	**0.000**	**0.000**
tai100a	21059006	0.558	261.2	0.783	0.513
tai100b	1185996137	0.001	241.0	0.397	0.135
sko100a	152002	**0.000**	304.8	0.073	0.054
wil100	273038	**0.000**	316.6	0.035	0.009
tai150b	498896643	0.076	1549.4	1.128	0.439
tho150	8133398	0.013	1991.7	0.012	0.065
tai256c	44759294	0.136	7377.8	0.271	0.170
Average		0.049	757.8	0.215	0.133

of Misevicius. Since approximately half of the test problems are run by one set of algorithms and the other half by a different set, a globally dominant algorithm cannot be determined.

Since CPTS is a parallel algorithm and all other algorithms in Tables 4 and 5 are sequential, runtime comparisons convey no useful information other than to demonstrate that the wall clock times for CPTS are within reasonable ranges.

Table 6 presents results for our cooperative parallel tabu search algorithm (CPTS), an independent parallel tabu search algorithm (TB-MTS), and a cooperative tabu search hybrid with a genetic algorithm (TB-COSEARCH). To the best of our knowledge, MTS and COSEARCH are the only two parallel algorithms whose computational results can be used for comparison with those of CPTS. The other parallel algorithms from the literature either were not tested for the same test problems or their outcomes were not presented in a format that could be utilized. Many of the earlier studies adopted a theoretical approach and presented results on timing and solution quality for only a single run, the one where the algorithm gave its best performance, or else presented no results at all.

TB-MTS and TB-COSEARCH are both due to Talbi and Bachelet (2006) and are the most recent parallel algorithms for the QAP from the literature. As can be seen in Table 6, our CPTS method outperforms MTS on all test instances. CPTS similarly outperforms COSEARCH on all instances but one (tai100a). Time comparisons were not possible because the authors of these algorithms did not present times in their tables. In addition, the parallel architecture used in their study differs substantially from the one of the current study, making time comparisons of questionable value. The difficulty of making such comparisons is further magnified by the differences in platforms and number of processors available (where 150 processors were incorporated into the Talbi and Bachelet study while system constraints restricted us to only 10 processors in ours).

7. Conclusions

This study introduced a cooperative parallel tabu search algorithm (CPTS) for the quadratic assignment problem (QAP) that provides high quality results for a large set of benchmark instances from QAPLIB. We also provided some of the first heuristic results for a new, difficult set of 60 test instances.

CPTS performed well when compared to the independent search strategy (and the cooperative search strategy) of Talbi and Bachelet (2006). Solution quality results indicate an advantage of cooperative strategies over independent search strategies. CPTS outperformed JRG-DivTS, our multi-start TS run on the same platform, which further illustrates the possible benefit of parallel strategies. Although CPTS provides results that are competitive with the best sequential algorithms in the literature, only JRG-DivTS was run on all the same test instances and was uniformly dominated by our method. The best performing sequential algorithms were run on either the tai∗a/b test instances or the sko∗ instances only, not both, which make the identification of a globally dominant algorithm impossible. The comparisons provide an indication of the strength of our algorithm in relation to other well performing methods. While a couple of the hybrid GAs are strong contenders on one test set or the other, we note that algorithms can behave quite differently on different test sets. The best performing hybrid GA on the Taillard instances was not run on the Skorin-Kapov instances and vice versa. This complicates overall comparisons and the designation of an overall best algorithm.

By embedding simple tabu search strategies in a parallel design the algorithm implements a cooperative search based on adaptive memory. The algorithm is tailored to take advantage of the platform on which it is run. By leveraging the characteristics of the parallel programming implementation and architecture, CPTS demonstrates the benefits that can be obtained by parallelizing metaheuristic methods for difficult combinatorial optimization problems. Determining the impact of the platform and number of processors used on the performance of the algorithm provides an appealing area for future research.

We anticipate that our outcomes can be enhanced in settings where additional processors are available. Only 10 processors were employed in the CPTS runs of our study, which may have limited the quality of the solutions found within the time limits imposed. Future research investigating the scaling options of the algorithm and the impact of the number of processors utilized may provide interesting insights if more processors become available. Likewise we anticipate that our outcomes can be improved by incorporating more sophisticated adaptive memory components. Varying the neighborhood used and the diversification operator employed are anticipated to produce interesting outcomes. Future research will be devoted to investigating such components within environments provided by differing types of parallel platforms.

References

Anstreicher, K.M., Brixius, N.W., Goux, J.-P., Linderoth, J., 2002. Solving large quadratic assignment problems on computational grids. Mathematical Programming 91 (3), 563–588.

Battiti, R., Tecchiolli, G., 1992. Parallel based search for combinatorial optimization: genetic algorithms and TABU. Microprocessors and Microsystems 16 (7), 351–367.

Burkard, R.E., Karisch, S.E., Rendl, F., 1997. QAPLIB – A quadratic assignment problem library. Journal of Global Optimization 10 (4), 391–403.

Cela, E., 1998. The Quadratic Assignment Problem: Theory and Algorithms. Kluwer Academic Publishers.

Chakrapani, J., Skorin-Kapov, J., 1993. Massively parallel tabu search for the quadratic assignment problem. Annals of Operations Research 41, 327–341.

Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R., 2001. Parallel Programming in OpenMP. Morgan Kaufman.

Ciriani, V., Pisanti, N., Bernasconi, A., 2004. Room allocation: A polynomial subcase of the quadratic assignment problem. Discrete Applied Mathematics 144, 263–269.

Cordeau, J-F., Gaudioso, M., Laporte, G., Moccia, L., 2005. The service allocation problem at the Gioia Tauro Maritime Terminal. European Journal of Operational Research, Forthcoming.

Crainic, T.G., Toulouse, M., 2003. Parallel strategies for metaheuristics. In: Glover, F., Kochenberger, G. (Eds.), Handbook of Metaheuristics. Kluwer Academic Publishers, pp. 475–513.

De Falco, I., Del Balio, R., Tarantino, E., Vaccaro, R., 1994. Improving search by incorporating evolution principles in parallel tabu search. In: Proceedings International Conference on Machine Learning, pp. 823–828.

Drezner, Z., 2003. A new genetic algorithm for the quadratic assignment problem. INFORMS Journal on Computing 15 (3), 320–330.

Drezner, Z., 2005. The extended concentric tabu for the quadratic assignment problem. European Journal of Operational Research 160, 416–422.

Drezner, Z., Hahn, P., Taillard, E., 2005. Recent advances for the quadratic assignment problem with special emphasis on instances that are difficult for meta-heuristic methods. Annals of Operations Research 139 (1), 65–94.

Felten, E., Karlin, S., Otto, S.W., 1985. The traveling salesman problem on a hypercube, MIMD computer. In: Proceedings 1985 of the International Conference on Parallel Processing, pp. 6–10.

Fiechter, C.N., 1994. A parallel tabu search algorithm for large traveling salesman problems. Discrete Applied Mathematics 51 (3), 243–267.

Gambardella, L., Taillard, E., Dorigo, M., 1997. Ant Colonies for the QAP, Tech. Report IDSIA-4-97, IDSIA, Lugano, Switzerland.

Glover, F., 1998. A template for scatter search and path relinking. In: Hao, J.-K. et al. (Eds.), Artificial Evolution, . In: LNCS 1363. Springer-Verlag, pp. 13–54.

Glover, F., Laguna, M., 1997. Tabu Search. Kluwer Academic Publishers.

Gutjahr, W.J., Hitz, M., Mueck, M.A., 1997. Task assignment in Caley interconnection topologies. Parallel Computing 23, 1429–1460.

Haghani, A., Chen, M., 1998. Optimizing gate assignments at airport terminals. Transporation Research A 32 (6), 437–454.

James, T., Rego, C., Glover, F., 2005. Sequential and parallel path-relinking algorithms for the quadratic assignment problem. IEEE Intelligent Systems 20 (4), 58–65.

James, T., Rego, C., Glover, F., 2006. Multi-start tabu search and diversification strategies for the quadratic assignment problem, Working Paper, Virginia Tech. tajames@vt.edu.

Li, Y., Pardalos, P.M., Resende, M.G.C., 1994. A greedy randomized adaptive search procedure for the quadratic assignment problem. In: Pardalos, P.M., Wolkowicz, H. (Eds.), . In: Quadratic assignment and related problems, vol. 16. DIMACS Series on Discrete Mathematics and Theoretical Computer Science, pp. 237–261.

Loiola, E., Maia de Abreu, N., Boaventura-Netto, P., Hahn, P., Querido, T., 2007. A survey for the quadratic assignment problem. European Journal of Operational Research 2 (16), 657–690.

Misevicius, A., 2003. Genetic algorithm hybridized with ruin and recreate procedure: Application to the quadratic assignment problem. Knowledge-Based Systems 16, 261–268.

Misevicius, A., 2004. An improved hybrid genetic algorithm: new results for the quadratic assignment problem. Knowledge-Based Systems 17, 65–73.

Misevicius, A., 2005. A tabu search algorithm for the quadratic assignment problem. Computational Optimization and Applications 30, 95–111.

Oliveira, C.A., Pardalos, P.M., Resende, M.G.C., 2004. GRASP with path-relinking for the quadratic assignment problem, Efficient and Experimental Algorithms. In: Ribeiro, C.C., Martins, S.L. (Eds.), . In: Lecture Notes in Computer Science, vol. 3059. Springer-Verlag, pp. 356–368.

Pardalos, P.M., Pitsoulis, L.S., Resende, M.G.C., 1995. A parallel GRASP implementation for the quadratic assignment problem. In: Ferreira, A., Rolim, J. (Eds.), Parallel Algorithms for Irregular Problems. Kluwer Academic Publishers, pp. 111–130.

Siu, F., Chang, R., 2002. Effectiveness of optimal node assignment in wavelength division multiplexing networks with fixed regular virtual topologies. Computer Networks 38, 61–74.

Stutzle, T., Dorigo, M., 1999. ACO algorithms for the quadratic assignment problem. In: Corne, D., Dorigo, M., Glover, F. (Eds.), New Ideas for Optimization. McGraw-Hill, pp. 33–50.

Taillard, E., 1991. Robust taboo search for the quadratic assignment problem. Parallel Computing 17, 443–455.

Taillard, E., 2007. Summary of Best Solution Values Known, Retrieved May, 2007. http://ina2.eivd.ch/Collaborateurs/etd/problemes.dir/qap.dir/summary_bvk.txt.

Talbi, E-G., Bachelet, V., 2006. COSEARCH: A parallel cooperative metaheuristic. Journal of Mathematical Modeling and Algorithms 5, 5–22.

Talbi, E-G., Hafidi, Z., Geib, J-M., 1997. Parallel adaptive tabu search for large optimization problems. In: MIC'97-2nd Metaheuristics International Conference, Sophia Antipolis, France.

Tseng, L., Liang, S., 2005. A hybrid metaheuristic for the quadratic assignment problem. Computational Optimization and Applications 34, 85–113.