Parallelization of the Quadratic Assignment Problem on the Cell

Jose Antonio Pascual[1] Jose Antonio Lozano[2] and Jose Miguel Alonso[3]

Abstract— **The mapping problem involves the assignation of a set of tasks of a parallel application onto a set of computational nodes. This problem can be formulated as an instance of the Quadratic Assignment Problem (QAP), being this one of the most difficult combinatorial optimization problems. The solution for the QAP is a permutation that minimizes an objective function. This paper proposes two algorithms to generate sequences of permutations with distance one between them, and to map each permutation of size n with a number in the range {1,$n!$}. This is very important because, once the objective function for a given permutation has been computed, the computation for the next permutation is very simple. Once we have a fast solution to the QAP, we can use it to solve non-trivial cases of mapping problems, in this case checking the complete permutation space.**

Keywords— **Mapping Problem, Quadratic Assignment Problem, Permutations Set Parallelization.**

I. Introduction

A Permutation of order n is an arrangement of n symbols. Given a set $X=\{1,2, \ldots, n\}$, the set of all permutations over X is denoted by π_n, representing the whole set of bijections of the X set onto itself. The cardinality of π_n is $n!$.

There are many problems, most of them related to heuristic search and combinatorial optimization, in which the search of the solution is performed over a set of permutations. Generally, these problems belong to the NP-complete complexity class, because of the cardinality of the solution space. The TSP (Travelling Salesman Problem) [1] and the QAP (Quadratic Assignment Problem) [2] belong to this class.

NP-complete problems include those for which there is not known algorithm capable of solving them in polynomial time, and even small size instances require long computation times. Due to this complexity, the exhaustive search of the optimum solution becomes impracticable when the size of the problem grows. For example, the exact resolution of QAP for problem sizes greater than 30 require a huge amount of compute time [2]. For this reason, several heuristic methods that find good solutions (but cannot guarantee optimal solutions) have been developed.

The above mentioned heuristic methods reduce the time required to find solutions for these problems, but they are still time-consuming for large-size problem instances. Execution speed can be reduced using parallel computing, and any techniques have been developed to tackle this. In our case, given the structure of the problem at hand, the most efficient parallelization approach involves the partition of the search space across the different computational elements. This is because of the independency between permutations: each permutation corresponds to a possible problem solution, that can be computed without knowledge of the solutions for the remaining permutations.

A more in-depth analysis of the problem reveals a trivial acceleration strategy for this problem. The generation of permutations at distance one will allow the reutilization of previously done calculations, reducing the number of operations to perform for the next permutation. This paper focus on the creation of consecutive (distance one) permutations [3], and well as on a method to map each size-n permutation with a natural number in the range $\{1,n!\}$. This second issue is important in order to be able to distribute de whole the permutations set along a collection of processors. The process of mapping a permutation to a natural number is known as ranking [4] [5], and the inverse of this operation (that is, computing the permutation that corresponds to a given natural number in the range $\{1,n!\}$) is known as unranking [4] [5]. The ranking function is bijective, so for every permutation belonging to π_n exists exactly one i in the range $\{1,n!\}$ such that $rank(\pi_i) = i$. The existence of an unranking function is guaranteed, because a ranking is a bijection and the unranking must be its inverse function.

The context in which these techniques will be used is in finding solutions for the mapping problem [6] stated as an instance of the QAP. This problem studies the mapping of a set of tasks belonging to a parallel application onto an available (pre-assigned) set of nodes. This problem has been studied by many researchers [2] [7] [8] [9], and many techniques for solving it have been proposed. Some of the approaches will be discussed later. Our main interest is on providing mechanism to accelerate them, using parallel processors. We provide an implementation of a parallel program that finds the optimal solution of the QAP by means of an exhaustive sweep of the solution space. This program has been implemented for the Cell Broadband Engine [10].

The rest of the paper is organized as follows. In Section II the motivation to develop this parallelization technique is explained. Section IV states the Mapping problem as a QAP instance, and discusses some previous work on how to solve it, making in Sec-

[1]Department of Computer Architecture and Technology, The University of the Basque Country, e-mail: `joseantonio.pascual@ehu.es`.
[2]Department of Computer Sciences an Artificial Intelligence, The University of the Basque Country, e-mail: `ja.lozano@ehu.es`.
[3]Department of Computer Architecture and Technology, The University of the Basque Country, e-mail: `j.miguel@ehu.es`.

tion III a brief review of the work done in the permutations area. Section V presents the algorithms that have been designed and implemented, making in Section VI an analysis of the obtained results with some classical QAP problems. Section VII closes the paper with some conclusions and future lines of research.

II. Motivation

Most current processors are composed a collection of cores that can work together towards the resolution of problems. To that extent, some sort of internal interconnection network is required. For example, in the Cell a ring is used, while Intel's Larrabee incorporates a mesh [11]. The efficient exploitation of these multi-core architectures depends on many factors, such as the design of the application (as a collection of collaborative tasks) and the communication patterns between tasks, which can put a considerable stress on the interconnection network. A good (optimal) allocation of tasks to cores is required to reduce this stress. This is the reason why the mapping problem is of interest in the design of multi- and many-core computing systems.

To define it formally, the mapping problem involves the search for the best assignment of a group of tasks belonging to a parallel application onto a given set of computational cores. This assignment must minimize some function. Typically, this function is the mean distance of the messages (packets, or any class of communication unit) interchanged by the cores. That is, given a set of tasks $T = \{t_1, ..., t_n\}$ and a set of cores $C = \{c_1, ..., c_n\}$, the objective is to find a mapping function π that assigns a task t to a core c trying to minimize some objective function.

$$\begin{aligned} \pi : T &\longrightarrow C \\ t_j &\longrightarrow \pi(t_j) \end{aligned}$$

The result of the application of this function to a set of tasks will be a mapping vector that associates each task of the set T to one core of the set C.

As we anticipated, the mapping problem can be stated as an instance of the Quadratic Assignment Problem (QAP), a standard problem in location theory that consists in finding the optimal assignment of a certain number of facilities to a certain number of locations with the minimum cost. For each pair of locations a distance between them is given, and for each pair of facilities a weight is provided, which represents the flow between them. The problem involves the search of a mapping vector that assign each facility to each location minimizing the sum of the distances multiplied by the corresponding weights. The mapping problem is clearly a QAP instance where the locations are the cores, the facilities are the application tasks and the flow between them is the amount of information that tasks interchange.

The QAP is one of the most challenging NP-Complete combinatorial optimization problem [6]. Due to its complexity, there is not an algorithm capable of solving it in polynomial time. In addition, in [6], the authors show that the problem belongs to the hardest core of this complexity class. Due to this, extensive research has been done in methods to solve it either in an optimal or a sub-optimal way.

In either case, the parallelization of these methods is the natural way to reduce the execution times. The resolution of a size n QAP requires the evaluation of only a subset of the permutations space, if heuristic methods are being used, or the exhaustive evaluation of the complete set if using exact. Due to the problem structure, the simplest way to parallelize the problem is to distribute the set of permutations across the different compute nodes of a parallel computer, and perform independently the calculations of the objective function in each of these nodes. With this kind of parallelization strategy each node will look for its best, local solution. At the end of the program, the best global minimum will be selected.

III. Permutations and Ranking Algorithms

Certain classes of algorithms require working with non-overlapping subsets of a permutation space. To help with this subsetting it is necessary to create ranking/unranking functions that, given a permutation, maps it onto a natural number between $\{1, n!\}$ (and unranks a given number into the corresponding permutation). The lexicographical order is a natural order structure of the Cartesian product of two ordered sets being the most used to order permutations. A ranking function is called lexicographic if it maps a permutation and its lexicographically next permutation into consecutive integers.

A distance between permutations can be defined as follows: given two permutations π_i and π_j, the distance between them is the lower bound of pairwise exchanges needed when transforming one to the other [3].

Table I shows the first five permutations of four numbers (0 to 4) in lexicographic order. Note how this order does not guarantee that two consecutive permutations are at distance one.

In this paper will use this topological characteristic of the permutation space for speeding up calculations when this space is traversed looking for the optimization of some objective function.

TABLE I
A sequence of permutations generated in lexicographical order. Two consecutive permutations are not always at distance one.

Rank	Permutation	d
0	0 1 2 3	-
1	0 1 3 2	1
2	0 2 1 3	2
3	0 2 3 1	1
4	0 3 1 2	2
5	0 3 2 1	1

In Table II a different permutation sequence has been generated, but instead of using the lexicographic order, care has been taken to maintain distances one between two consecutive ones. This is the ordered permutation space that is used in this paper in order to facilitate the parallelization of algorithms.

TABLE II
A SEQUENCE OF PERMUTATIONS GENERATED, MAINTAINING DISTANCE ONE BETWEEN EVERY TWO CONSECUTIVE ELEMENTS.

Rank	Permutation	d
0	0 1 2 3	-
1	1 0 2 3	1
2	2 0 1 3	1
3	0 2 1 3	1
4	1 2 0 3	1
5	2 1 0 3	1

Extensive research has been done in the creation of ranking and unranking functions for permutations [4] [5]. To our knowledge, none of the previous works deal with ranking functions that take into consideration the guarantees of distance one between consecutive permutations.

IV. The Mapping Problem Formulated as a QAP instance

The formal definition of the problem is as follows. Given F and L, two equal size sets representing facilities and locations, $w : F \times F \longrightarrow \Re$ the weight function and $d : L \times L \longrightarrow \Re$ the distance function, find the bijection $\pi : F \longrightarrow$L such that:

$$min \sum_{i,j \in F} w(i,j) \cdot d(\pi(i), \pi(j)) \ . \qquad (1)$$

or, in its matrix form:

$$min \sum_{i,j \in F} W_{i,j} \cdot D_{\pi(i),\pi(j)} \ . \qquad (2)$$

It is clear that the mapping problem has the same structure of the QAP problem. The relationship between each other is as follows:

1. Locations – Network nodes
2. Facilities – Parallel application tasks
3. Weight matrix – Amount of communication between each pair of nodes
4. Distance matrix – Distance between the nodes in the network (depends on the network topology)

The formal definition of the mapping problem formulated as a QAP instance is as follows. Given T and C, two equal-size sets representing the parallel application tasks and the cores in the processor, $w : T \times T \longrightarrow \Re$ the weight function representing the amount of information interchanged between each pair of nodes and $d : C \times C \longrightarrow \Re$ the distance function representing the distance between each pair of nodes in a concrete network topology, find the bijection $\pi : T \longrightarrow$C such that:

$$min \sum_{i,j \in T} W_{i,j} \cdot D_{\pi(i),\pi(j)} \ . \qquad (3)$$

Extensive research has been conducted to efficiently solve QAP problem. Most works were focused on the search of sub-optimal solutions using heuristic methods, due to the complexity of providing exact solutions. The most used techniques to tackle the problem are:

1. **Exact algorithms:** Exact algorithms try to find the optimum solution for a given problem. The two main approaches developed include dynamic programming algorithms [12] and branch and bound techniques [7]. These algorithms uses lower bounds to reduce the number of nodes to visit. The formal definition is as follows. Given a subset S of a partially ordered set P, the lower bound is defined as an element of P which is lesser or equal than every element of S. The lower bounds are key for algorithms in combinatorial optimization [13]. Generally, problems of size greater than 15 are hard to solve, and only certain instances of the problem, with certain structural characteristics, have been solved for sizes up to 31 [2].
2. **Sub-optimal algorithms:** Due to the inherent complexity of the problem, several sub-optimal algorithms have been developed. These include improvement methods such as local search and tabu search [14], simulation approaches like simulated annealing [15] and genetic algorithms [8]. One of the most used heuristic procedures is the GRASP (Greedy Randomized Adaptive Search Procedures). This is an iterative randomized technique [9]. The construction of the solution is composed by two steps. In the first, an initial solution is constructed via an adaptive greedy randomized function; afterwards, a local search is carried out in order to improve the solution. This two-step process is iterated, keeping the best solution of the whole process as the final result.

All of these techniques evaluate the whole set of permutations or a subset of them. Those that the search for sub-optimal solution values dramatically reduce the computation time to find a solution, compared to the time required to search for the optimal solution. However, when the problem size increases, even sub-optimal methods require unaffordable computation times. Mechanisms to accelerate program execution, which include parallelization, are required to deal with problems of realistic sizes.

V. Design and Implementation of Algorithms to Solve the QAP

In this section we explain in detail the design and implementation of a parallel algorithm to solve the QAP problem using an exhaustive search of the permutation space. The parallel approach is very simple: just distribute evenly the permutation space along the processing elements. Knowing the number of tasks, each one can easily compute the number of iterations it has to perform in order to sweep its permutation sub-space. This will be done after obtaining its initial permutation using the *get_perm* function, as well as a control vector using the *get_control* function. After discussing these functions, we will explain how they will be combined to solve the QAP.

A. Algorithms Dealing with Permutations

The following three algorithms are designed to generate the permutation sequences (with distance one between each consecutive pair) and to unrank a permutation given an index.

A.1 Permutations Generator Algorithm

The *get_perm* algorithm sweeps the permutation space. The input is composed by two vectors containing the current permutation and the current control vector. In each iteration, the next permutation at distance one is created. Note that when *iter_num* is equal to the size of the permutations set, and *perm* is the permutation with index 0, the algorithm explores the whole set of permutations. This is way this function is used in the sequential version of our program to solve the QAP. The *iter_num* parameter controls the number of iterations, and is used in the parallel version to control the number of permutations that each thread has to process.

```
gen_perm(iter_num,i_c,perm,control)
   cont=0;
   while(cont < iter_num){
      control[i_c]--;
      j=(i_c%2==0)?0:control[i_c];
      tmp = a[j];
      a[j] = a[i_c];
      a[i_c] = tmp;
      i_c = 1;
   }
   while (!control[i_c]){
      control[i_c] = i_c;
      i_c ++;
   }
   cont ++;
}
end gen_perm
```

A.2 Algorithm to Obtain the Control Structure

In order to generate consecutive, distance one permutations, an auxiliary structure called control vector is used. It is required to control which permutations have been already visited. This algorithm constructs the control vector that correspond to the permutation with index k, which is provided as parameter.

```
get_control(n,k,control)
begin
   num_perm = factorial(n);
   aux = num_perm-k;
   ind = n-1;
   for(i=n-1;i>=0;i--){
      f = factorial(ind);
      coc = aux / f;
      aux = aux % f;
      control[i] = coc;
      ind --;
   }
   control[n] = n;
end get_control
```

A.3 Algorithm to Obtain the Permutation Vector

This algorithm computes the permutation that corresponds to a given control vector. It takes as input the size of the permutation and a control vector, generating as result the permutation.

```
get_perm(n, control, perm)
begin
   for(i=0;i<n;i++){
      perm[i] = i+1;
   }
   for(i=n-1;i>=0;i--){
      aux = control[i];
      while(aux<i){
         aux2 = perm[i];
         for(j=i;j>=1;j--){
            perm[j] = perm[j-1];
         }
         perm[0] = aux2;
         aux ++;
      }
   }
   i = 1;
   while(!control[i]){
      control[i] = i;
      i ++;
   }
   return(i);
end get_perm
```

B. Exhaustive Search Algorithm for the QAP

In order to test the algorithms described above, three implementations of a QAP solver have been developed. The first version implements the simplest way to solve the QAP problem, making all the required calculations to compute the value of the cost function to minimize for each permutation. A second sequential version includes important improvements to accelerate the process, taking advantage of the property of consecutively generated permutations being at distance one. The final, parallel version is a modification of the previous one, using the unrank function to implement the partition of the permutation space between the threads. The reader will note that the number of operations that the algorithm must compute is proportional to a constant k that only depends on the size of the elements.

B.1 Sequential Version without Improvements

```
procedure QAPseq(n)
begin
   foreach permutation
      s = calc_cost(perm);
      s_min = min(s,s_min);
   end foreach
end QAPseq
```

The function *calc_cost(perm)* computes the Equation 3 given a size n permutation. This algorithm runs over the whole permutations set performing a total of $n! \cdot n^2$ operations.

B.2 Sequential Version with Improvements

```
procedure QAPseq_dist(n)
begin
   foreach gen_permutation
      s = calc_cost(perm[i],perm[j]);
      s_min = min(s,s_min);
   end foreach
end QAPseq_dist
```

In this case, the function *gen_perm* generates permutations with distance one between them, allowing to the function *calc_cost(perm[i],perm[j])* calculate only these two elements cost reducing the amount of calculations needed. This function calculates on each iteration the value of the permutation with and without make the change, comparing this new value of the objective function with the previously calculated value. If this new value is lower, the new permutation is stored as the current optimal solution. This algorithm runs over the whole permutations space performing a total of $n! \cdot k$ operations where k is constant.

B.3 Parallel Version

```
procedure QAPsec(n)
begin
   size = factorial(n);
   size_proc = size / num_procs;
   foreach procs
      QAPpar(n,size_proc);
   end foreach
   wait();
   s_min_total = min(s_min);
end QAPsec

procedure QAPpar(n,size_proc)
begin
   perm = unrank(proc_id*size_proc)
   foreach perm from perm
      s = calc_cost(perm[i],perm[j]);
      s_min = min(s,s_min);
   end foreach
end QAPpar
```

This algorithm is a modified version of the QAPseq_dist allowing its execution in a parallel machine. The sequential code calculates the number of permutations that a thread must do and executes the function QAPpar onto each thread. This function calculates the first permutation, unranking the number passed to it and traverses the permutations space until *size_proc* calculations of the function emphcalc_cost are done.

These algorithms runs over the whole permutations space. Each thread performs a total of $n! \cdot k/nt$ operations, where nt is the number of threads and k is a constant.

C. Implementation on the Cell Broadband Engine

The platform selected to run the parallel program was a PlayStation 3, which incorporates a Cell processor. The tests have been carried out as follows. First, the non-optimized sequential version was run on a SPE (Synergistic Processing Element). Results were compared with those obtained with the optimized (sequential) version. Finally, we ran tests with the parallel version using from one to the six SPEs available in the Cell of a PS3, running a single, separate thread on each SPE.

VI. Results and Analysis of Experiments

In this section we carry out a collection of experiments to thoroughly evaluate the performance of our programs when solving different instances of the QAP. The sequential versions were created in order to assess the benefits of the acceleration of the incremental computation of the cost function. Additionally, the are the yardstick against to compare the performance of the parallel version. The problems to solve have been extracted from the QAPLIB [16], the de-facto standard for benchmarking QAP solutions. We have selected problems with three different sizes to carry out the experiments.

Table III shows execution times of the two sequential programs (un-optimized and optimized) when solving QAP problems of size 12, 14 and 15. The reader can observe the notable reduction of time of the optimized version, which is achieved because the cost function for a given permutation is computed easily, in constant time, from the cost already computed for the previous, at distance one.

TABLE III
Execution times (minutes) for the two sequential programs.

Problem	Size	Time	Time Opt
chr12a	12	13	2.5
had12	12	13	2.5
nug12	12	13	2.5
had14	14	54000	455
nug14	14	54000	455
chr15a	15	810154	6825
chr15b	15	810154	6825

The parallel version further accelerates execution times. Figures 1 and 2 shows the execution times, for different numbers of SPEs (threads), for two QAP problems of size 12 and 14 respectively. We can observe how the algorithm scales almost linearly with the number of SPEs. This is because the total independency between the tasks assigned to each SPE, that do not require interchanges of information. Only at the end the threads communicate their partial solutions, that are merged to select the final one.

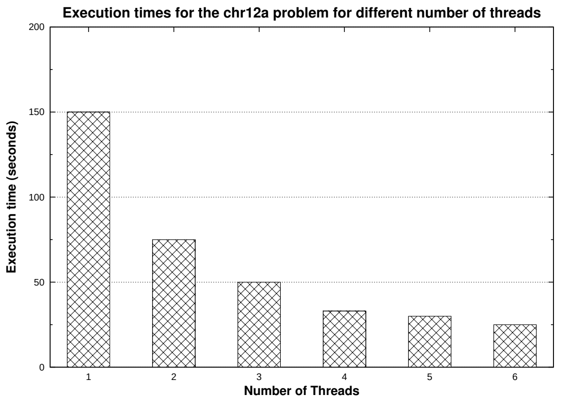

Fig. 1. Execution times for the parallel version applied problem chr12a, for 1-6 threads.

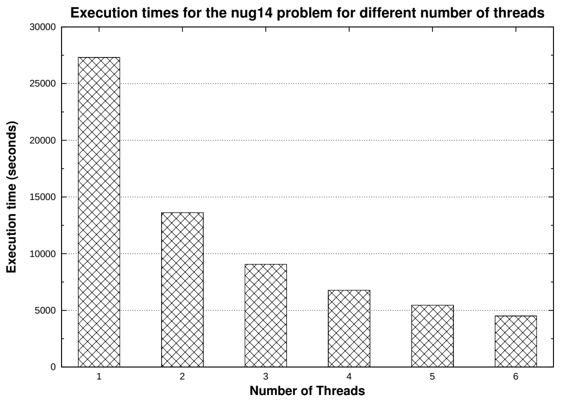

Fig. 2. Execution times for the parallel version applied problem nug14, for 1-6 threads.

VII. Conclusions and Future Work

In the computer architecture area, the mapping problem (assignment of parallel tasks to computational units taking into account the characteristics of the interconnection fabric) has received important attention, particularly in the field of on-chip multiprocessors, homogeneous as well as heterogeneous. We have shown too how this is a particular instance of the most general Quadratic Assignment Problem. A good mapping is especially relevant on embedded systems, where a single (parallel) application will be the only one running on a chip, but can also be applicable to general-purpose, multi-application environments. In the first case, system designer can use complex (slow) algorithms to search for very good (optimal) solutions; for the latter, sub-optimal solutions are acceptable, because they have been obtained rapidly.

Throughout this paper we have show how to accelerate the solution of the QAP problem (and, therefore, of the mapping problems). Two orthogonal acceleration techniques have been identified and implemented: acceleration of the evaluation of the cost function (taking advantage of the generation of sequences of permutations at distance one), and parallelization. These techniques have been tested with problems taken from the QAPLIB, and both of them have proven its usefulness, alone and in combination. Programs have been tested on a Cell processors, but the underlying acceleration techniques are valid for any computing platform.

The main limitation of our experiments is that acceleration techniques have been used only within programs that carry out an exhaustive sweep of permutation spaces. Even with accelerator, this approach is useless for problem sizes larger than 16. As a future line of work, we plan to implement them within heuristic algorithms that are not exhaustive, but provide (supposedly) good solutions in much shorter times. As the problem still consists on checking parts of a permutation spaces, the acceleration techniques should be applicable too.

The main contributions of this work have to be found in the mechanism designed to generate sequences of permutations with distance one between consecutive elements, and the associated ranking/unranking functions. This has been the key point in providing excellent acceleration levels for both the sequential and the parallel QAP-solving algorithms. Additionally, this allows for a simple partition of the permutation space, which is the key for implementing simple but efficient (linear speedup) parallel versions of the algorithms.

We plan to use these algorithms in research work related to dynamic assignment of parallel tasks to compute nodes in supercomputing environments. The dynamic nature of this assignment requires fast generation of good mappings. Acceleration techniques can help us to obtain acceptably good solutions within a restricted time budget.

References

[1] E.L. Lawler, J.K. Lenstra, Karl Rinnooy, and D.B. Shmoys, "The traveling salesman problem: A guided tour of combinatorial optimization," 1985.

[2] Panos M. Pardalos, Franz Rendl, and Henry Wolkowicz, "The quadratic assignment problem: A survey and recent developments," in *In Proceedings of the DIMACS Workshop on Quadratic Assignment Problems, volume 16 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. 1994, pp. 1–42, American Mathematical Society.

[3] Chong Zhang, Zhangang Lin, Qi Zhang, and Zuoquan Lin, "Variable neighborhood search with permutation distance for qap," in *Knowledge-Based Intelligent Information and Engineering Systems*, London, UK, 2005, pp. 81 – 88, Springer-Verlag/Heidelberg.

[4] Wendy Myrvold and Frank Ruskey, "Ranking and unranking permutations in linear time," *Inf. Process. Lett.*, vol. 79, no. 6, pp. 281–284, 2001.

[5] Mares Martin and Milan Straka, "Linear-time ranking of permutations," in *Algorithms - ESA 2007*, London, UK, 2007, pp. 187–193, Springer-Verlag/Heidelberg.

[6] S. H. Bokhari, "On the mapping problem," *IEEE Trans. Comput.*, vol. 30, no. 3, pp. 207–214, 1981.

[7] K. G. Ramakrishnan, M. G. C. Resende, and P.M. Pardalos, "A branch and bound algorithm for the quadratic assignment problem using a lower bound based on linear programming," in *In C. Floudas and P.M. Pardalos, editors, State of the Art in Global Optimization: Computational Methods and Applications*. 1995, pp. 57–73, Kluwer Academic Publishers.

[8] Charles Fleurent, Jacques, and Jacques A. Ferland, "Genetic hybrids for the quadratic assignment problem," in *DIMACS Series in Mathematics and Theoretical Computer Science*. 1993, pp. 173–187, American Mathematical Society.

[9] Carlos A. S. Oliveira, Panos M. Pardalos, and Mauricio G.C. Resende, "Grasp with path-relinking for the

qap," in *5th Metaheuristics International Conference MIC03*, 2003.

[10] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM J. Res. Dev.*, vol. 49, no. 4/5, pp. 589–604, 2005.

[11] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–15, 2008.

[12] Ambros Marzetta and Adrian Brungger, "A dynamic-programming bound for the quadratic assignment problem," in *Computing and Combinatorics*, London, UK, 1999, pp. 339 – 348, Springer-Verlag/Heidelberg.

[13] Peter Hahn and Thomas Grant, "Lower bounds for the quadratic assignment problem based upon a dual formulation," .

[14] Alfonsas Misevicius, "A tabu search algorithm for the quadratic assignment problem," *Comput. Optim. Appl.*, vol. 30, no. 1, pp. 95–111, 2005.

[15] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.

[16] Rainer E. Burkard, Stefan E. Karisch, and Franz Rendl, "Qaplib – a quadratic assignment problemlibrary," *J. of Global Optimization*, vol. 10, no. 4, pp. 391–403, 1997.