A *SIMD Tabu Search* Implementation for Solving the Quadratic Assignment Problem with GPU Acceleration

Clara Novoa
Ingram School of Engineering
Texas State University
601 University Drive
San Marcos, TX 78666-4684
cn17@txstate.edu

Apan Qasem
Computer Science Dept.
Texas State University
601 University Drive
San Marcos, TX 78666-4684
apan@txstate.edu

Abhilash Chaparala
Computer Science Dept.
Texas State University
601 University Drive
San Marcos, TX 78666-4684
chaparala.abhilash@gmail.com

ABSTRACT

In the Quadratic Assignment Problem (QAP), n units (usually departments , machines, or electronic components) must be assigned to n locations given the distance between the locations and the flow between the units. The goal is to find the assignment that minimizes the sum of the products of distance traveled and flow between units. The QAP is a combinatorial problem difficult to solve to optimality even for problems where n is relatively small (e.g., $n = 30$). In this paper, we develop a parallel *tabu search* algorithm to solve the QAP and leverage the compute capabilities of current GPUs. The algorithm is implemented on the Stampede cluster hosted by the Texas Advanced Computing Center (TACC) at the University of Texas at Austin. We enhance our implementation by exploiting dynamic parallelism made available in the Nvidia Kepler high performance computing architecture. On a series of experiments on the well-known QAPLIB data sets, our algorithm produces solutions that are as good as the best known ones posted in QAPLIB. The worst case percentage of accuracy we obtained was 0.83%. Given the applicability of QAP, the algorithm we propose has very good potential to accelerate scholarly research in Engineering, particularly in the fields of Operations Research and design of electronic devices. To the best of our knowledge, this work is the first to successfully parallelize the *tabu search* metaheuristic to solve the QAP with the *recency-based* feature, implemented serially in [11]. Our work is also the first to exploit GPU dynamic parallelism in a *tabu search* metaheuristic to solve the QAP.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; I.2.8 [**Artificial Intelligence**]: Search—*Heuristic Methods*

General Terms

Algorithms, Performance

XSEDE '15, July 26 - 30, 2015, St. Louis, MO, USA

DOI: http://dx.doi.org/10.1145/2792745.2792758

Keywords

Tabu search, Quadratic Assignment Problem, Parallel computing, GPU computing, Dynamic parallelism

1. INTRODUCTION

The Quadratic Assignment Problem (QAP) is an NP-hard combinatorial optimization problem [24, 31, 39]. The problem is to assign n units to n locations to minimize the total cost measured as the sum of the products of flows between units and distances between locations [35]. Flow and distance matrices are known. QAP has many applications [6, 13] and the most common one is the design of facility layouts in manufacturing systems. In such application, the QAP finds an optimal allocation of n facilities (departments, machines, or workstations) to sites to minimize the total layout costs [11]. Other applications of QAP include the placement of modules on board positions so that the total wire length to connect them is minimized (i.e. computer backboard wiring [33]), campus planning [15], hospital layout [16], ergonomic design [1] (i.e., keyboard and control panel design [30]), scheduling problems [17], placement of electronic components [26], and memory layout optimization in signal processors [38]. The problem of assigning docks in a cross-docking facility is modeled also as a special case of the QAP [12]. In this problem, the objective is to assign the incoming trailers to the inbound door positions and the outgoing trailers to the outbound door positions to minimize material handling cost. This cost is incurred by the fork lifts or similar material handling systems when they move the product between the inbound and the outbound positions.

The QAP has attracted many researchers not only because of its practical and theoretical importance but also due to its complexity [4]. In general, instances with $n > 30$ are difficult to solve by exact methods in reasonable time [1, 24, 6]. This fact has motivated the use of the problem as a benchmark to test solution methodologies. Metaheuristic approaches have been applied most often than heuristics [2]. QAPLIB serves as the most prominent source for problem instances and known solutions [8]. An increased interest in parallel computing to solve the QAP was evidenced in the 1990's [24]. This trend began with the development of CPU parallel implementations such as the ones in [9], [19] and [34]. In the last few years, there has been a shift to finding solutions using GPUs [5, 14, 25, 37, 39].

GPU's are powerful accelerators, require less energy than other computing devices, and are widely available and rela-

tively cheap. These GPU characteristics and the fact that GPU computing has been identified as a very promising direction in the field of Operations Research, motivated the authors of this paper to solve the QAP using the GPU. We chose to parallelize the *tabu search* metaheuristic because, it has been reported as the most efficient approximate method for solving the QAP[32, 11, 35, 34]. Even if several variants of *tabu search* can be implemented, the *tabu search* method have provided equal or better solutions when compared to other approximate methods.

In this paper, we present a single instruction multiple data *tabu search* algorithm for the QAP implemented on the GPU. The computational framework used is the TACC Stampede cluster at the University of Texas at Austin. TACC is one of the XSEDE partner institutions. In the computational study in this paper, we investigate the performance and accuracy of the proposed *tabu search* GPU implementation. The performance is assessed by comparing our implementation to a parallel *2-opt* algorithm recently implemented by the authors of this paper [10] and to the *tabu search* GPU implementation described in [39]. The accuracy is studied comparing the best results from our *tabu search* implementation vs. the best known solutions posted in QAPLIB.

The paper is organized as follows. Section 2 presents two formulations for the quadratic assignment problem and a brief description of a *tabu search* meta-heuristic for solving the QAP in a serial environment. Section 3 provides an overview of GPU computing. Section 4 provides a description of the *tabu search* implementation in GPU. Section 5 contains a literature review focused on work solving QAP using GPU. Section 6 presents the computational experiments and the numerical results. Section 7 summarizes the conclusions and discusses future research.

2. QAP FORMULATIONS

2.1 Koopmans-Beckman QAP Formulation

Koopmans and Beckmann provide the following formulation [22]: let F and D be two given $n \times n$ matrices that represent flows between units and distances between locations such that $F = [f_{kl}]$ and $D = [d_{ij}]$. Consider the set of positive integers $1, 2, ..., n$ and let Π_n be the set of all permutations of $1, 2, ..., n$. The QAP can be defined as finding a permutation $\pi^* \in \Pi_n$ such that the sum of the products below is minimized.

$$z_\pi = \sum_{i=1}^{n} \sum_{j=1}^{n} f_{\pi_i \pi_j} \cdot d_{ij} \tag{1}$$

2.2 Quadratic 0-1 Formulation

Koopmans and Beckmann also stated a quadratic 0-1 integer programming formulation [22]. In this formulation, $X = [x_{mp}]$ represents an $n \times n$ matrix of decision variables. The variable x_{ki} takes the value of 1 if unit k is assigned to location i and 0 otherwise. The solution finds the values of all variables x_{ki} and x_{lj} that minimize the sum of the products of flows and distances (Eq. 2) and satisfy the constraints expressed in Eqs. 3-5. Constraints in (3) assign each unit k to a single location i. Constraints in (4) assign only one unit to each location. Constraints in (5) force the decision variables to take binary values

$$min \quad z = \sum_{k=1}^{n} \sum_{l=1}^{n} \sum_{i=1}^{n} \sum_{j=1}^{n} f_{kl} d_{ij} x_{ki} x_{lj} \tag{2}$$

s.t

$$\sum_{i=1}^{n} x_{ki} = 1, \qquad k = 1, 2, ..., n \tag{3}$$

$$\sum_{k=1}^{n} x_{ki} = 1, \qquad i = 1, 2, ..., n \tag{4}$$

$$x_{ki} \in \{0, 1\} \qquad k = 1, 2, ..., n \quad i = 1, 2, ..., n \tag{5}$$

The information encoded in a permutation π in Π_n has a one-to-one correspondence to the information stored in the $n \times n$ matrix $X = [x_{ki}]$. If x_{ki} equals to 1 then the i-th element in π is k. Since the formulations in 2.1 and 2.2 are equivalent, we call Eq. (2) (or alternatively Eq. (1)) the objective function. Its value (or cost) permits to assess and rank different problem solutions. In practice, the total material handling cost computed as in equations (1) and (2) is one of the most common ways to measure the efficiency of a facility layout. Authors in [36], mention that over $300 billion will be spent annually in the US alone on facilities that will require planning or replanning. By identifying fast implementations that produce cost effective solutions this expense can be reduced.

2.3 Serial Tabu Search for QAP

Following is a description of a basic serial *tabu search* (TS) method to solve the QAP. It resembles the description for the robust TS method in [34]. The reader that is not familiar with the TS method may consult [18]. TS is a metaheuristic approach for solving combinatorial optimization problems. Applications of TS include scheduling, transportation, network design, layout and circuit design problems, telecommunications, probabilistic logic and expert systems, neural network pattern recognition [11].

A basic TS method to solve the QAP starts with a randomly generated permutation of the integers $1, 2, ..., n$. This permutation will be called the *initial solution*. It is also set as the *current solution*. The *initial solution* and its cost are stored in *best solution so far* array and *best cost so far* variable. An iteration counter is set to one. *At any iteration of the TS method, a local exchange procedure is applied to the *current solution* to generate a list of candidate solutions (i.e. neighborhood solutions) that represent moves from the *current solution*. The objective function value or cost is computed for all the candidate solutions. If the candidate solutions list is large, the user might restrict the cost computation to a subset of solutions. The TS method chooses the candidate solution with the best cost. If the solution selected is forbidden because it is in the *tabu list* and does not satisfy the aspiration criteria, the TS method drops this solution from consideration and proceeds to select the next best cost solution. The meaning of a *tabu list* and an *aspiration criteria* are explained more in detail in the next paragraph. The final selected solution becomes the *current solution*; it may be a non-improving move with respect to the previous *current solution*. If the cost of the *current solution* is less than the cost of the *best solution so far*, the *best solution so far* and the *best cost so far* are updated to match the *current solution* and its associated cost. The it-

eration counter is increased by one. From the new *current solution*, TS finds a new list of neighborhood solutions. This means, the steps described in this paragraph starting from the sentence marked with an * are repeated.

The iterative procedure described in the previous paragraph goes until some predetermined stopping criteria are reached. Common stopping criteria are: (1) the number of iterations performed equals the maximum number of predetermined iterations or (2) the number of iterations since the last update of the *best cost so far* and *best solution so far* is larger than a specified threshold. The *tabu list* stores solutions that the TS method does not want to select in the next few iterations. The objective of the *tabu list* is to avoid a cycling behavior. For instance, if the search is in a solution that corresponds to a local minimum, the best move in the next iteration could be a deteriorating one. If the local minimum solution is not stored in the *tabu list*, in a new iteration the algorithm will return to this previous solution and then cycling around the local optimum will occur. Since the *tabu list* may forbid critical promising moves, TS enables the user to apply a feature known as aspiration criteria. The aspiration criteria are ways to override the tabu status of a solution. A commonly used aspiration criterion is to allow to select a tabu move if it leads to a solution whose cost is better than the cost of the *best solution so far*.

In this paragraph, we provide details about a simple and convenient local exchange procedure that can be applied to the *current solution* to generate the list of candidate solutions at any TS iteration. From the introduction we recall that one of the most common applications of QAP is the design of layouts in manufacturing systems. Without loss of generality we will call the permutation π a *current solution* layout. The permutation element stored at position i will represent the unit or department k assigned to location i. A simple move from a *current solution* layout to another one results from the interchange of the units or departments k and l stored in two arbitrarily selected permutation positions i and j. This move is simple and convenient since it doesn't change the location of any other unit or department in the permutation and permits a fast evaluation of the cost of the move. The systematic way to obtain the entire six pair-wise interchange of departments for a current solution or permutation of size four is illustrated in Fig. 1. Over each arc in Fig. 1 is the new solution or permutation resulting from the pair-wise exchange or move. In general, for a problem of size n there are $n*(n-1)/2$ pair-wise exchanges.

Burkard and Rendl [7] provide a formula to compute the change (i.e. *delta*) in the objective function value after a pair-wise exchange (i.e. a swap). The advantage of this formula is that it can be evaluated in $O(n)$ operations for all the $O(n^2)$ pair-wise exchanges. In contrast, the computation of cost using Eq. (1) requires $O(n^2)$ operations. Following is the formula in [7] for the case in which both flows and distances are asymmetric.

$$\begin{aligned}\Delta_{ij} &= (d_{ji}-d_{ij})(f_{\pi_i\pi_j}-f_{\pi_j\pi_i}) \\ &+ \sum_{k\in n\setminus\{i,j\}}((d_{jk}-d_{ik})(f_{\pi_i\pi_k}-f_{\pi_j\pi_k}) \\ &+ (d_{kj}-d_{ki})(f_{\pi_k\pi_i}-f_{\pi_k\pi_j})) \qquad (6)\end{aligned}$$

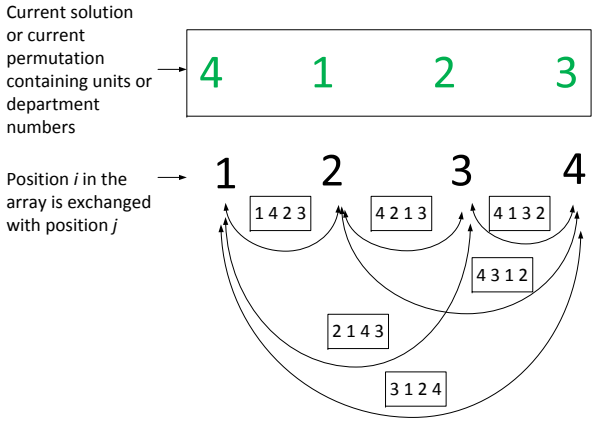

Figure 1: Two-way exchange procedure to generate moves and candidate solutions in the TS method

3. GPU COMPUTING

Microprocessors based on a single central processing unit (CPU) drove rapid performance increase and cost reductions in computer applications. This drive slowed due to the energy-consumption and the heat-dissipation that limited the clock frequency and the level of productive activities performed in each clock period within a single CPU [21]. Semiconductor industry then settled on two trajectories for designing microprocessors, multi-core and many-core.

In the past, GPUs were special-purpose hardwired application accelerators, suitable only for conventional graphics applications. Modern GPUs are fully programmable, autonomous parallel floating point processors which may simultaneously execute the same program instruction on different data. Nvidia, the leading manufacturer of GPUs, released CUDA, a parallel computing platform and programming model that provides a C programming language interface to program the GPU hardware. CUDA enables dramatic increases in computing performance by harnessing the power of the GPUs.

One appealing characteristic of the GPU is that it efficiently launches many threads and executes them in parallel to enable computational throughput across large amounts of data. Each thread runs the same program named a kernel. Threads are grouped into thread blocks and all threads in a thread block may cooperate to solve a sub problem. A block has a dimensionality of one, two or three. A grid is a set of blocks which are completely independent. A grid has dimensionality of one or two. A warp is a group of threads within a block that are launched together and execute together. Warp size is typically 32 threads on current generations of GPUs. Shared memory can be accessed by all threads within a block but not across blocks. Luong et al. describe several factors that affect the performance of GPU-based QAP applications [25]. These include efficient distribution of data processing between CPU and GPU, the level of required communication and synchronization among threads, the optimization of data transfer between the different parts of the memory hierarchy, and the capacity constraints of these memories.

One attractive feature of the Nvidia K20 GPU card is that

it supports a technology called dynamic parallelism [29] that permits the GPU to operate in a more autonomous way. The user may launch a grid of parent threads that run the same program or kernel and these threads may launch a new grid of child threads without returning to the host. The grid dimensions and thread block sizes are set at the time of the call. This effectively eliminates superfluous back and forth communication between the GPU and CPU through nested kernel computations. The invocation of the child threads is properly nested and implicitly synchronized, meaning that the parent threads are not complete until all child threads created have completed.

In his technical report [20] presents a figure similar to the one in Figure 2. It illustrates the GPU operation without dynamic parallelism (left side) and with dynamic parallelism (right side). As the figure shows dynamic parallelism permits the development of algorithms that can do dynamic run time decisions. Besides, dynamic parallelism frees the CPU for more time to do other tasks. It also favors more CPU power conservation. The Nvidia K20 is equipped with the Grid Management Unit (GMU) that manages the dynamic execution by generating, suspending and resuming kernels, as well as tracking dependencies from multiple sources. A layer of system software running on the GPU interacts with the GMU, enabling the CUDA Runtime application-programming interface (API) to be used from within a kernel program.

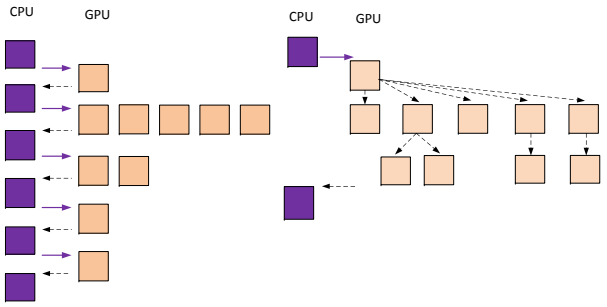

Figure 2: Flow between the host CPU and the GPU without and with dynamic parallelism

4. GPU IMPLEMENTATION OF TABU SEARCH

This section describes the *Tabu Search* (TS) algorithm parallelized in this work. The parallelized TS includes the *recency-based* memory feature proposed in [11] and the dynamic *tabu list* size strategy cited in [11], [34], and [35]. Both the *recency-based* memory and the dynamic *tabu list* size feature are explained below. Authors in [11] claimed that these TS features plus intensification strategies and a long term memory structure to further implement diversification strategies lead the TS algorithm to converge to very good solutions at a reasonable speed regardless of the chosen initial solution.

The *recency-based* feature in [11] keeps track of the tabu status of a move. The feature is implemented by creating a two-dimensional square array named *Tabuarr* with all its cells containing zeros as initial values. For $i < j$, (i.e. upper triangle of the two-dimensional array *Tabuarr*), the *i-th row* and *j-th column* identifies the layout move that results if department stored in the permutation π at location i is interchanged with the department or unit stored at location j. Every time units or departments in positions i and j of a current solution layout are exchanged, the cell $Tabuarr(i, j)$ (for i<j) stores an integer value equal to $current\_iter + t$ where *current_iter* is the current iteration number and t is the number of iterations in which the move will be kept tabu. Therefore, t corresponds to the *tabu list* size. The value for t may be fixed or randomly generated to implement a dynamic *tabu list* size. In this way, if $taburarr[i][j] \leq current\_iter$, the move defined as exchanging the units or departments i and j is not tabu, otherwise the move is in the *tabu list*.

As suggested in [18], the cells in $Tabuarr[i][j]$ (for i>j) (i.e. lower triangle of the two-dimensional array *Tabuarr*) may store the number of times units or departments i and j have been exchanged. Thus, if at iteration one units or departments in positions 2 and 3 are interchanged, the cell $Tabuarr[3][2]$ becomes 1, and if at iteration six units or departments in positions 2 and 3 are interchanged again, the cell $Tabuarr[3][2]$ becomes 2. This frequency of use information is a long term memory structure helpful to diversify the search. However, in the parallelized TS algorithm implemented in this work we diversify the search only through the implementation of a dynamic *tabu list* size. We don't utilize the information stored in the lower triangle of *Tabuarr*.

In his seminal paper, Taillard [34] mentions that the choice of the size of the *tabu list* is critical to diversify the search. Cycling may occur if the *tabu list* size is too small. On the contrary, if the size of the list is too large, promising moves may be forbidden deviating the exploration to solutions of lower quality and increasing the number of iterations to find a good or optimal solution. To overcome this problem, we opted to implement a variable *tabu list* size as proposed in [34]. Since the minimum and maximum list size will be problem dependent we experimented with the recommendations in [34] and[11]. We finally set the list size between $0.1n$ and $0.33n$, where n is the problem size (i.e. number of units or departments and also the number of locations). At every iteration, when a move is set as tabu, a random number in the interval [0.1n,0.33n] is generated to determine t, the number of iterations for which the move will be tabu.

In this paragraph, we summarize the parallel TS algorithm we implemented in the GPU. It is depicted also in the flowchart in Fig 3. Using specific seed values, a set of N initial random permutations or layout solutions of size n is generated in the CPU and stored in a matrix of size $N \times n$. Each permutation is assigned to a GPU thread. At each thread an iteration counter variable named *current_iter* is set to 0 and a two-dimensional array *Tabuarr* of size *nxn* is created. The cost of the initial random permutation is computed using Eq. (1) described in Section 2.1. The initial permutation is copied into an array named *best-solution-so-far* and the associated permutation cost is copied to a variable named *best-cost-so-far*. The permutation or solution is also stored in the *current solution* array and the *current_iter* counter is increased by one. Next, each thread should perform pairwise interchanges to the current solution to generate a set of S neighborhood solutions with $S = n \times (n-1)/2$. Instead

of asking the threads to generate all these neighbor solutions, we take advantage of CUDA dynamic parallelism. A number of $2S/n$ (i.e. $n-1$) child threads (CT) are created to ask them to efficiently generate subsets of the neighbor solutions and compute their costs. The subsets are of size $n/2$ and the costs are computed using Eq. (6) from Section 2.3. Since the flow and distance matrices are needed to compute the costs of every solution in the neighborhood set, these matrices are maintained in global memory to be accessible by all threads. Using the cost information returned by the child threads, each parent thread identifies the (i,j) best cost pair-wise interchange (with $i < j$) and proceeds to determine if such a move is not tabu. If this is the case, the cell *Tabuarr[i][j]* ($i < j$) is updated to *current_iter*$+t$ to set the movement as tabu and the cell *Tabuarr[j][i]* is updated to increase the frequency of the selected movement by one. If the movement is tabu, its associated cost is compared to *best-cost-so-far* to determine if using this aspiration criteria the tabu status can be overridden. If the tabu status can be overridden, the permutation associated with the selected move becomes the *current solution*; otherwise the TS algorithm identifies the next best pair-wise interchange. The step of identifying a pair-wise exchange that can be selected is done for as many times as necessary. If there are no more pair-wise exchanges to select, the TS procedure should stop prematurely. However, with an appropriate choice of the *tabu list* size it can be avoided.

At every iteration it is possible that *best-cost-so-far* and *best-solution-so-far* need to be updated. This is done by comparing the cost of the selected permutation or *current solution* to the the value stored in *best-cost-so-far*. If the cost of a *current solution* is lower than *best-cost-so-far* then the *current solution* (i.e current permutation) is stored in *best-solution-so-far* and its cost is stored in *best-cost-so-far*. Now the algorithm proceeds to validate the stopping criterion. It compares the value of *current_iter* to the value of *Max_iter*, a variable that stores the predetermined maximum number of iterations to perform the TS algorithm in each thread. If the stopping criterion is not reached, the algorithm proceeds to start a new TS iteration. The total number of iterations the TS algorithm is repeated is set as a function of the problem size n. After the total number of iterations is reached, each thread returns its *best-solution-so-far* and *best-cost-so-far* values. The process of comparing the results returned by the threads and finding the single permutation with the minimum cost is done on the CPU. Once the solution and its cost is identified and output to a file the TS algorithm terminates.

5. PREVIOUS WORK ON SOLVING QAP USING TABU SEARCH AND GPU

Zhu *et al.* proposed a single-instruction multiple data (SIMD) *tabu search* (TS) for the QAP using the GPU [39] on a personal computer. The parallelization consisted of running 6144 simultaneous independent tabu searches (6144 threads, 32 blocks, 192 threads per block) on 128 processors. Texture memory (a fast read-only memory) was used to store the distance and flow matrices. To assure each thread searches a different but promising area the authors implemented diversification and intensification operations every m iterations. The authors demonstrated the implemented algorithm was effective. They used instances of different sizes ($30 < n < 90$) from QAPLIB and the worst performance gap is 0.85% (by comparing the solution they reported and the latest best known solution for the instance named *tai80a*). The authors also stated that the cache size (8k) of the texture memory affected the experimental performance. Since their TS implementation only had short-term memory, as a future research they proposed to develop long-term memory.

Czapinski proposed an effective parallel multistart TS (PMTS) for the QAP on the CUDA platform [14]. The technique consisted of diversifying an initial solution, running multiple tabu searches on each diversified solution, and re-starting the search with the best solutions after a certain number of iterations. The set of tabu searches to run in different threads results from systematic swaps of an initial solution. It permits the author to conclude that each thread can save just two rows or two columns of the flow and distance matrix for symmetric matrices. It avoids keeping the whole matrices in shared memory. In the non-symmetric case, two rows and two columns of the matrices are needed. To get a full-benefit of coalescing transposed copies of the matrices are stored. The proposed search also benefits from communication between parallel *tabu search* instances which is achieved by passing the best obtained solutions to the CPU, examining them and choosing new configurations in the CPU, and re-starting the parallel TS in the GPU. From initial experiments the author agreed with [39] that 192 threads per block was the best choice. Instances of size 50-70 ran faster in GPU when compared to a six-core MPI implementation.

Other work related to solving a 3D extension of QAP with *tabu search* using GPU is the one in [25]. At the best of our knowledge, our work is the first one on successfully parallelizing the TS metaheuristic with the *recency-based* feature implemented serially in [11]. Our work is also the first one on exploiting GPU dynamic parallelism in a TS implementation to solve the QAP.

6. EXPERIMENTAL RESULTS

6.1 GPU Platforms and Benchmark Data Sets

The computational experiments were executed on the Stampede cluster on the TACC system. Stampede is a 10 PFLOPS Dell Linux Cluster based on 6,400+ Dell Zeus PowerEdge server nodes, each outfitted with 2 Intel Xeon 8-Core 64-bit E5 processors (2.7 GHz) and an Intel Xeon Phi Co-processor (1.1.GHz). Each node runs Centos 6.3 (2.6 32x86_64 Linux kernel). The nodes are managed with batch services through SLURM 2.4. Stampede has 128 compute nodes outfitted with a single Nvidia K20 GPU on each node with 5GB of on-board GDDR5 memory. Each K20 GPU has 2496 CUDA cores distributed over 13 streaming multiprocessors (SM's). Each SM can hold a maximum of 2048 thread contexts, which amounts to 26624 (13*2048) threads that can simultaneously be active on the GPU. The clock speed for each core is 0.706 GHz, L1 cache size is 64 KB/SM and L2 cache size is 768 KB (shared).

The CUDA code was compiled with `nvcc` using CUDA version 5.5. The `sbatch` script was used to submit jobs to the cluster and to specify the node configuration. For our experiments, we ran four jobs simultaneously by assigning each job to a different Stampede node. This significantly expedited our evaluation process.

The instances used to test our TS algorithm come from

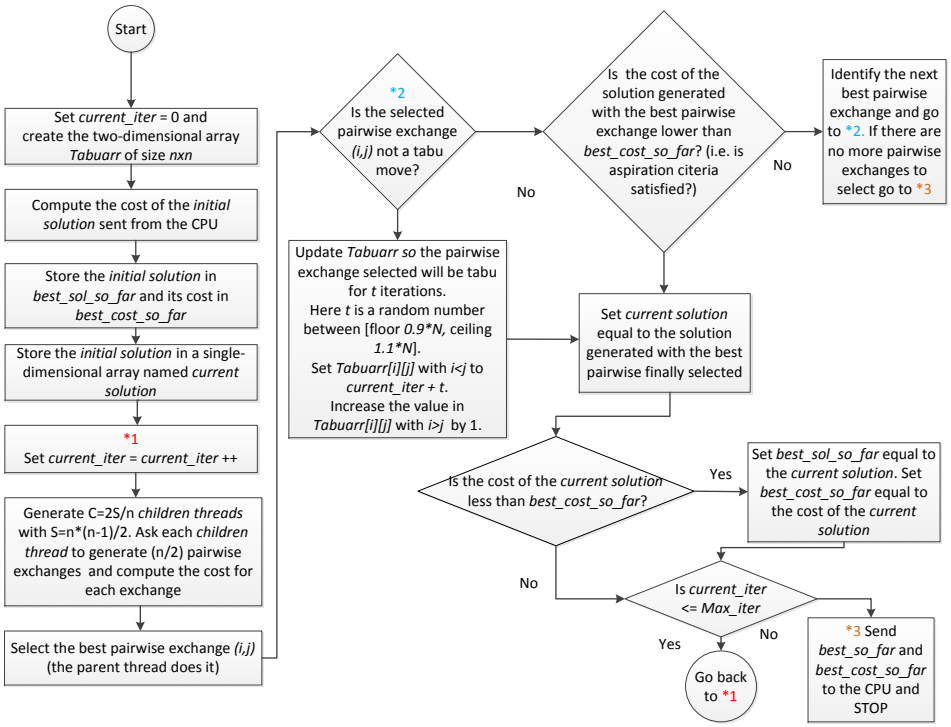

Figure 3: Parallel TS algorithm executed at each GPU thread

QAPLIB, a library of published test problems for the QAP described in [8]. We selected 2 instances from [23], 6 from [34] and 6 from [35]. The *Lipa* instances come from problem generators described in [23]. These generators provide asymmetric instances (i.e. non-symmetric flow and/or distance matrices) with known optimal solutions. The instances named *Taixxa* have random flows and distances generated from uniform distributions [34]. The problems labeled as *Taixxb* are introduced in [35] and they are asymmetric and randomly but not uniformly generated. We ran the TS algorithm in each problem instance eight times and computed the average values as well as the standard deviation, minimum, maximum, and coefficient of variation (standard deviation/average). All coefficients of variation are low.

6.2 Performance and Accuracy

Table 1 compares the accuracy and performance of our GPU accelerated tabu search algorithm to the following algorithms:

1. *2-opt (Chaparala et al.):* a GPU implementation of the *2-opt* algorithm to solve approximately the QAP we developed one year ago and described in [10]
2. *Tabu (Zhu et al.):* a GPU implementation of Tabu search to solve the QAP performed by Zhu *et al.* [39].

The accuracy gap, measured as the percent difference between the best known cost and the cost discovered by each algorithm, is reported for the 14 different instances or data sets selected from QAPLIB. The best known cost for a particular instance is the one reported at QAPLIB (`http://anjos.mgi.polymtl.ca/qaplib/inst.html`). The work in [8] also reports solutions and costs for the data sets studied. However, the QAPLIB website updates the best known solutions every time a new one is discovered. The instances starting with an * have known optimal solutions. For the other instances, the best known solution and cost comes from the Robust Tabu Search in [34], the Reactive Tabu Search in [3] or the Iterative Tabu Search in [27, 28].

The performance in terms of total execution time of the GPU kernel is also reported for each instance and algorithm. Times for *2-opt (Chaparala et al.)* are reported in seconds while times for the TS implementations are reported in minutes. For all implementations, the number of initial random solutions generated for each instance was $N = 6144$. Based on the results from experiments we performed in [10], the total number of threads, blocks and threads per block were set to 6144, 24, and 256, respectively for *2-opt(Chaparala et al.)* and our TS method. The total number of threads, blocks and threads per block used by Zhu *et al.* was 6144, 32 and 196, respectively.

We observe that *Tabu (Zhu et al.)* and our TS implementation have the same accuracy for 8 of the 15 instances studied. Our TS implementation has better accuracy in 5 instances and *Tabu (Zhu et al.)* has the best accuracy in 1 instance. The maximum percentage difference between our solution and the best known is 0.83% for the instance named *tai80a*. Table 1 also shows that the execution times for our TS algorithm are comparable to the ones in *Tabu (Zhu et al.)*, but not consistently smaller. The running times for our algorithm ranged from 1.79 to 362.89 minutes. The running

Table 1: Computational results for problems from the QAPLIB

	2-Opt (Chaparala et al.)		*Tabu Search (Zhu et al.)*			*Tabu Search (Novoa et al.)*		
Problem	GAP	Seconds	GAP	Seconds	Minutes	GAP	Seconds	Minutes
tai30a	1.10%	3.84	0.00%	18.60	0.31	0.00%	107.62	1.79
*tai30b	0.00%	3.78	0.00%	192.00	3.20	0.00%	107.56	1.79
tai40a	1.55%	11.83	0.07%	442.20	7.37	0.07%	906.75	15.11
tai40b	0.02%	11.68	0.00%	508.20	8.47	0.00%	906.60	15.11
tai50a	1.78%	29.40	0.58%	1210.80	20.18	0.39%	971.48	16.19
tai50b	0.15%	29.17	0.05%	574.20	9.57	0.05%	971.25	16.19
tai60a	2.50%	62.15	0.45%	1144.80	19.08	0.64%	3897.38	64.96
tai60b	0.23%	61.19	0.12%	2091.00	34.85	0.06%	3678.45	61.31
tai80a	2.48%	202.11	0.85%	11230.20	187.17	0.83%	10607.20	176.79
tai80b	0.52%	199.20	0.25%	10976.40	182.94	0.09%	14714.80	245.25
tai100a	2.35%	501.65	0.72%	23215.80	386.93	0.72%	21773.40	362.89
tai100b	0.89%	493.62	0.53%	33167.40	552.79	0.39%	16234.90	270.58
*lipa70a	0.77%	117.08	0.00%	1172.40	19.54	0.00%	7214.00	120.23
*lipa90a	0.64%	327.19	0.00%	7585.20	126.42	0.00%	20594.60	343.24

time for *Tabu (Zhu et al.)* ranged from 0.31 to 552.79 minutes. The running times for these TS methods are small if compared to the ones of exact methods for solving the QAP such as branch and bound. . We are also aware that a straight forward comparison between the running times for our TS algorithm and *Tabu (Zhu et al.)* is not entirely meaningful since the algorithms were implemented in different computer architectures with different GPU card.

7. CONCLUSIONS AND FUTURE WORK

This paper presented a very accurate GPU implementation of *tabu search* to solve the QAP. It exploits the CUDA dynamic parallelism available in the Nvidia K20 GPU card, by asking multiple child threads to generate the pairwise exchanges or moves from a *current solution*. The child threads also evaluate the costs of these new generated solutions. Experimental results show that the TS algorithm with the dynamic parallelism was successfully implemented since it provides excellent accuracy on the instances studied. We solved the accuracy issue we had with the *Taixxa* instances when developing our accelerated 2-Opt algorithm described in [10]. We will do further research on ways to improve the TS computational times. We will look to speed-up the procedure that sorts the solutions generated by the child threads with respect to cost at each iteration. We will do also more experimentation to find the optimum number of child threads to launch.

Practical OR problems that may be impacted by the development of this parallel algorithm are in the contexts of facility layout, cross-docking, warehousing, ergonomics, health care optimization and scheduling. The OR community researching on exact solutions to the QAP may benefit also from the TS accelerated algorithm. A solution from a fast heuristic or meta-heuristic permits to establish bounds at the initial stages of exact solution approaches such as branch-and-cut and branch-and-bound methods. On the other hand, the accelerated TS algorithm can be hybridized with other heuristics to coin new approximate solution methods. The GPU accelerated TS implementation may be used by electronic industries working on the layout of electronic devices in computer backboards or the location of memories in signal processors.

We plan also on incorporating a long term *frequency based* memory feature by using the information currently stored in the lower diagonal of the two-dimensional array *Tabuarr*. This feature can be used to diversify the search even more. It could permit to find even more accurate solutions or solutions that beat the best known ones for some instances. The accessibility to the Stampede cluster reduced significantly the time to complete the experimentation phase. The online documentation from TACC and the suggestions from its staff members were very helpful. These facts should motivate more OR practitioners to use a computational cyberinfrastructure similar to the Stampede cluster.

8. ACKNOWLEDGMENTS

The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing high performance computing resources that have contributed to the research results reported within this paper. URL: http://www.tacc.utexas.edu. The second author acknowledges support from the National Science Foundation through awards CNS-1253292 and CNS-1305302.

9. REFERENCES

[1] K. Anstreicher, N. Brixius, J. P. Goux, and L. Linderoth. Solving large quadratic assignment problems on computational grids. *Mathematical Programming Series B*, 91:563–588, 2002.

[2] M. Bashiri and H. Karimi. Effective heuristics and meta-heuristics for the quadratic assignment problem with tuned parameters and analytical comparisons. *Journal of Industrial Engineering International*, 8(6):1–9, 2012.

[3] R. Batitti and G. Tecchiolli. The reactive tabu search. *ORSA Journal on computing*, 6(2):126–140, 1994.

[4] M. Bazaraa and H. Sherali. Benders partitioning scheme applied to a new formulation of the quadratic assigment problem. *Naval Research Logistics Quarterly*, 27:29–41, 1980.

[5] V. Boyer and D. El Baz. Recent advances on GPU computing in operations research. In *2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum (IPDPSW)*, pages 1778–1787. IEEE, May 2013.

[6] R. Burkard. Quadratic assignment problems. *European Journal of Operational Research*, 15(3):283–289, 1984.

[7] R. Burkard and F. Rendl. A thermodynamically motivated simulation procedure for combinatorial optimization problems. *European Journal of Operational Research*, 17(2):169–174, 1984.

[8] R. E. Burkard, S. E. Karisch, and F. Rendl. QAPLIB-A quadratic assignment problem library. *European Journal of Operational Research*, 55(1):115–119, 1991.

[9] J. Chakrapani and J. Skorin-Kapov. Massively parallel tabu search for the quadratic assignment problem. *Annals of Operations Research*, 41(4):327–341, 1993.

[10] A. Chaparala, C. Novoa, and A. Qasem. A SIMD solution for the quadratic assignment problem with gpu acceleration. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, XSEDE '14, pages 1:1–1:8, New York, NY, USA, 2014. ACM.

[11] W. C. Chiang and P. Kouvelis. An improved tabu search heuristic for solving facility layout design problems. *International Journal of Production Research*, 34(9):2565–2585, 1996.

[12] Y. Cohen and B. Keren. Trailer to door assignment in a synchronous cross-dock operation. *International Journal of Logistics Systems and Management*, 5(5):574–590, 2009.

[13] C. Commander. A survey of the quadratic assignment problem, with applications. *Morehead Electronic Journal of Applicable Mathematics*, 4:1–15, 2005.

[14] M. Czapinski. An effective parallel multistart tabu search for quadratic assignment problem on CUDA platform. *Journal of Parallel and Distributed Computing*, 73:1461–1468, 2013.

[15] J. Dickey and J. Hopkins. Campus building arrangement using topaz. *Transportation Research*, 6:59–68, 1972.

[16] A. Elshafei. Hospital layout as a quadratic assignment problem. *Operations Research Quarterly*, 28:167–179, 1977.

[17] A. Geoffrion and G. Graves. Scheduling parallel production lines with changeover costs: Practical applications of a quadratic assignment/lp approach. *Operations Research*, 24:595–610, 1957.

[18] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Boston, MA, 1997.

[19] T. James, C. Rego, and F. Glover. A cooperative parallel tabu search algorithm for the quadratic assignment problem. *European Journal of Operational Research*, 195:810–826, 2007.

[20] S. Jones. Gpu technology conference - introduction to dynamic parallelism. Technical Report SO338-GTC2012, NVIDIA Corporation, USA, 2012.

[21] D. B. Kirk and W. W. Hwu. *Programming massively parallel processors*. Morgan Kaufmann, Burlington, Massachussetts, 2010.

[22] T. Koopmans and M. Beckmann. Assignment problems and the location of economic activities. *Econometrica*, 15:53–76, 1957.

[23] Y. Li and P. Pardalos. Generating quadratic assignment test problems with known optimal permutations. *Computational Optimization and Applications*, 1:163–184, 1992.

[24] E. M. Loiola, N. de Abreu, P. Boaventura Netto, P. Hahn, and T. Querido. A survey for the quadratic assignment problem. *European Journal of Operational Research*, 176(2):657–690, 2007.

[25] T. Luong, L. Loukil, N. Melab, and E. Talbi. A GPU-based iterated tabu search for solving the quadratic 3-dimensional assignment problem. In *2010 IEE/ACS international Conference on Computer Systems and Applications (AICCSSA)*, pages 1–8. AICCSA, May 2010.

[26] G. Miranda, H. Luna, G. Mateus, and R. Ferreira. A performance guarantee heuristic for electronic components placement problems including thermal effects. *Computers and Operations Research*, 32:2937–2957, 2005.

[27] A. Misevicious. A tabu search algorithm for the quadratic assignment problem. *Computational Optimization and Applications*, 30:95–11, 2005.

[28] A. Misevicius. An implementation of the iterated tabu search algorithm for the quadratic assignment problem. *OR Spectrum*, 34(3):665–690, 2008.

[29] NVIDIA. Dynamic parallelism in cuda. Technical report, NVIDIA Corporation, USA, 2012.

[30] M. Pollatschek, N. Greshoni, and Y. Raddday. Optimization of the typewriter keyboard by simulation. *Angewandte Informatik*, 17:438–439, 1976.

[31] S. Sahni and T. Gonzalez. P-complete approximation problems. *Journal of Association for Computing Machinery*, 23(3):555–565, 1976.

[32] J. Skorin-Kapov. Extensions of a tabu search adaptation to the quadratic assignment problem. *Computers and Operations Research*, 21(8):855–865, 1994.

[33] L. Steinberg. The blackboard wiring problem: A placement algorithm. *SIAM Review*, 3:37–50, 1961.

[34] E. Taillard. Robust taboo search for the quadratic assignment problem. *Parallel Computing*, 17(3-4):443–455, 1991.

[35] E. D. Taillard. Comparison of iterative searches for the quadratic assignment problem. *Location Science*, 3(2):87–105, 1995.

[36] J. Tompkins, J. A. White, Y. A. Bozer, and J. M. Tanchoco. *Facilities Planning 4th Edition*. Wiley, Hoboken, NJ, 2010.

[37] S. Tsutsui and N. Fujimoto. Solving quadratic assignment problems by genetic algorithms with GPU computation: A case study. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, pages 2523–2530. ACM, July 2009.

[38] B. Wess and T. Zeitlhofer. On the phase coupling problem between data memory layout generation and address pointer assignment. *Lecture Notes in Computer Science*, 3199:152–166, 2004.

[39] W. Zhu, J. Curry, and A. Marquez. SIMD tabu search for the quadratic assignment problem with graphics hardware acceleration. *International Journal of Production Research*, 48(4):1035–1047, 2010.