Third International Workshop on Advanced Computational Intelligence
August 25-27, 2010 - Suzhou, Jiangsu, China

A Parallel Ant Colony Optimization Algorithm with GPU-Acceleration Based on All-In-Roulette Selection

Jie Fu, Lin Lei, Guohua Zhou

***Abstract*—Ant Colony Optimization is computationally expensive when it comes to complex problems. The Jacket toolbox allows implementation of MATLAB programs in Graphics Processing Unit (GPU). This paper presents and implements a parallel MAX-MIN Ant System (MMAS) based on a GPU+CPU hardware platform under the MATLAB environment with Jacket toolbox to solve Traveling Salesman Problem (TSP). The key idea is to let all ants share only one pseudorandom number matrix, one pheromone matrix, one taboo matrix, and one probability matrix. We also use a new selection approach based on those matrices, named AIR (All-In-Roulette). The main contribution of this paper is the description of how to design parallel MMAS based on those ideas and the comparison to the relevant sequential version. The computational results show that our parallel algorithm is much more efficient than the sequential version.**

I. Introduction

ANT Colony Optimization (ACO) is a metaheuristic inspired by the behavior of real ant colonies in which a colony of artificial ants cooperate in finding good solutions to difficult discrete optimization problems like the traveling salesman problem (TSP) [1]. ACO is naturally amenable to parallelism [2]. So, the algorithm will gain a significant improvement in performance by paralleling.

A key limitation of the MATLAB environment is that the computational speed in most cases is much slower than other programming tools, such as C or Fortran. However, MATLAB provides a script, called MEX, to compile a MEX file to a shared object that can be loaded and executed inside a MATLAB session. Nowadays, Graphics Processing Unit (GPU) has emerged as a promising parallel computing solution for large scale scientific computation within a dominant computing system [3]. The MATLAB Jacket toolbox use MEX files to invoke code on the GPU and to handle the data transfer between the CPU and GPU [4, 5].

The objective of this research is to find how and how much the ACO algorithm can be improved on a GPU+CPU platform with MATLAB and Jacket toolbox. We use the TSP, an NP-hard problem, as a case study and use MAX-MIN Ant System (MMAS) [1]—currently one of the best-performing ACO algorithms—as a basis for our parallel adaption.

The remainder of this paper is organized as follows. Section 2 gives background information about TSP, ACO, MMAS, GPU, and Jacket MATLAB toolbox. Section 3 presents the implementation of our parallel ACO algorithm. Section 4 provides the experimental results and analysis. Conclusion of our project and future research tasks are given in Section 5.

Manuscript received April 9, 2010.

Jie Fu is with Wuhan Digital Engineering Institute, Wuhan, China (jeffrey.full@hotmail.com).

Lin Lei was with Huazhong University of Science and Technology, Wuhan, China. She is now with the School of Computer Engineering, Nanyang Technology University, Singapore (leilin@pmail.ntu.edu.sg).

Guohua Zhou is with Wuhan Digital Engineering Institute, Wuhan, China (godzoo@21cn.com).

II. Basic Ant Colony Optimization Algorithm and GPU Computing

A. Traveling Salesman Problem

Traveling Salesman Problem (TSP) can be represented by a complete graph G=(V, E, d) with V being the set of nodes, also called cities, E being the set of arcs fully connecting the nodes, and d being the weight value of arcs (i, j)∈E, usually d_{ij} or d_{ji} is the distance between cities i and j, with (i, j)∈V. The TSP then is the problem of finding a shortest closed tour visiting each of the n=|V| nodes of V exactly once, in which the length is given by the sum of the lengths of all the arcs which have been visited.

B. Ant System

The ACO algorithm simulates the behavior of ants in depositing pheromone while building solutions or after they have established paths from the colony to food. It allows m artificial ants concurrently build a tour of the TSP. Initially, ants are located on randomly chosen cities. At each step, ant k applies a probabilistic action choice rule to decide which city to visit next. In particular, the probability with which ant k, currently at city i, chooses to go to city j is

$$P_{ij}^{k}(t)=\begin{cases}\dfrac{[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{l\in\mathcal{N}_i^k}[\tau_{il}]^{\alpha}[\eta_{il}]^{\beta}}, & j\in\mathcal{N}_i^k;\\ 0, & j\notin\mathcal{N}_i^k;\end{cases} \tag{1}$$

where $\eta_{ij}=1/d_{ij}$ is a heuristic value of moving from city i to city j, τ_{ij} represents the amount of pheromone trail on path(i,j) at time t, α and β are two parameters which regulate the relative influence of the pheromone trail and the heuristic information, and $\mathcal{N}_i^k$ is the feasible neighborhood of ant k when being at city i, that is, the set of cities that ant k has not visited yet.

After all the ants have built their paths, the pheromone trails are updated. This can be done by first lowering the pheromone value on all arcs by a constant factor, and then each ant deposits a quantity of pheromone on each path it has used. The pheromone updating is implemented by

$$\tau_{ij}(t+1)=(1-\rho)\tau_{ij}(t)+\sum_{k=1}^{m}\Delta\tau_{ij}^{k}(t), \tag{2}$$

where $0 < \rho \le 1$ is the pheromone evaporation rate, $(1-\rho)$ is the residual coefficient of pheromone, and $\Delta\tau_{ij}^{k}$ is the deposits of pheromone that the k_{th} ant left after one loop. It is defined here:

$$\Delta\tau_{ij}^{k}(t) = \begin{cases} 1/L^{k}, & \text{if } k_{th} \text{ ant passes through arc(i,j);} \\ 0, & \text{otherwise;} \end{cases} \quad (3)$$

where L^{k}, the length of the tour established by the k_{th} ant, is calculated as the sum of the lengths of the arcs belonging to the tour.

C. MAX-MIN Ant System

MAX-MIN Ant System (MMAS) [2] introduces four major improved features with respect to ant system: (a) either the iteration-best ant or the best-so-far ant is allowed to deposit pheromone(in this paper, we only use the iteration-best); (b) pheromone trail values are confined to the interval $[\tau_{\min}, \tau_{\max}]$; (c) trails are initialized to the upper pheromone trail limit; (d) pheromone trails are reinitialized each time the system reached stagnation.

Following rules are applied to calculate the τ_{min} and τ_{max}:

$$\tau_{\max} = 1/\rho L^{ib}, \quad (4)$$

$$\tau_{\min} = \tau_{\max}(1-\sqrt[n]{0.05})/((avg-1)\times\sqrt[n]{0.05}), \quad (5)$$

where L^{ib} is the iteration-best length, and avg is the average number of different choices available on an ant at each step while constructing a solution[1].

D. GPU Computing with MATLAB and Jacket Toolbox

GPU computing follows a parallel pattern called Single Instruction Multiple Thread (SIMT) [6]. With SIMT, a GPU executes the same instruction set on different data elements at the same time. A GPU can process thousands of light-weight threads simultaneously to obtain high computational throughput across a large quantity of data. The Compute Unified Device Architecture (CUDA) developed by NVIDIA is a parallel computing architecture for the GPU. The Jacket MATLAB Toolbox developed by AccelerEyes is a toolbox designed to write and run MATLAB code on the GPU in the native M-Language used in MATLAB [5]. Jacket accomplished this by automatically wrapping the M-Language into a GPU compatible form, which allows high level language as MATLAB to utilize GPU computational platform and therefore speeds up different algorithms.

III. Parallel MMAS Algorithm

A. Principles of Algorithm

CPU, in substance, is a model for computing scalars, lacking computing units compared with GPU. CPU mainly aims at optimizing complicated controlling system with low latency instead of high throughput, while in the opposite, GPU has high-bandwidth data access and executes the same set of instructions on large scale data concurrently [7]. Often, we exploit a space-time tradeoff in GPU. Nowadays, GPU is still deficient on branch and iterative feedback processing [8]. So it is wise to avoid bringing branch divergence and iterative operations into programming to the greatest extent. Because all the operations in MATLAB are based on matrix, the main idea of the algorithm is to pack data into large scale matrices, which can be processed aggressively to access the most benefits of GPU. Meanwhile, some pieces of small scale data will still be processed in CPU.

B. Implement MMAS on GPU+CPU

Our parallel implementation of MMAS is based on the public ACOTSP software package which is developed in C (http://www.aco-metaheuristic.org/aco-code/public-software.html).

1) Initialize a square matrix of pheromone trails and a square matrix of distances among cities (in GPU):

Generate an n×n matrix TAU, n being the number of cities, and set all elements to some arbitrarily high value while leaving the diagonal elements to 0. Generate an n×n matrix DIST, which stores all the calculated distances among cities, while leaving the diagonal elements to infinity.

2) Initialize a matrix of the tours of all ants (in CPU):

Generate an m×(n+1) matrix TRAVELED, where m is the number of ants and n is the number of cities. Set the first column of this matrix to a vector of uniformly distributed pseudorandom integers between the interval of [1,n] using Mersenne Twister, which stands for the starting point of each ant.

3) Initialize a matrix of TABOO (in CPU):

This TABOO matrix (m×n matrix of ones) indicates which cities have been visited by those ants. Set those elements according to the matrix TRAVELED. For instance, if TRAVELED(4,1) is 5, which means at the 1_{st} step, ant 4 has moved to city 5, therefore TABOO(4,5) becomes 0.

4) Initialize a square matrix of probability of moving from one city to another (in GPU):

This n×n matrix, called PROB, is stored with values calculated with (1).

For (iteration j=1, 2, ..., j_{max})

For (step k=1, 2, ..., n)

5) Generate pseudorandom numbers (in GPU):

The m×n matrix of uniformly distributed pseudorandom numbers, called $DICE^{k}$, is confined to the interval (0,1), and calculated using Mersenne Twister concurrently in GPU.

6) Generate the next batch of cities for all ants (in GPU):

Recombine PROB according to values of TRAVELED's k_{th} column, to generate a new matrix $PROB^{k}$. For example, if the 1_{st} column of TRAVELED is [2,1], which means at the 1_{st} step, ant 1 is in city 2 and ant 2 is in city 1, then swap the 1_{st} row in PROB with the 2_{nd} row in that. Copy the TABOO matrix from CPU into GPU to form a new matrix $TABOO^{k}$. After this, multiply $TABOO^{k}$, $DICE^{k}$, and $PROB^{k}$ in an element-by-element manner. Then, collect the max elements' column indices of every row of the resulting matrix, and form a new vector VISIT. The VISIT indicates the next collection of cities which the ants will visit.

In [6] and [7], the authors use the traditional roulette wheel selection, which inevitably involves conditional statements. Although they apply some methods (such as parallel prefix sum, parallel reduction, etc.) to improve the performance, those methods are too complex and not essentially suitable for GPU. Our algorithm can fully utilize GPU to speed these

computations up, since they are the computing bottlenecks of MMAS, and both MATLAB and Jacket perform best on vectorized code. We call this variation of selection approach AIR (All-In-Roulette).

7) Update TRAVELED and TABOO (in CPU):

Since these updating operations are done one-by-one and the GPU-CPU communication overhead is relatively small [9], we transform the VISIT into CPU and use it to update TRAVELED. After this, update TABOO according to TRAVELED. For instance, if the TRAVELED(3,2)=4, which means ant 3 at the 2_{nd} step has visited city 4, then we will set TABOO(3,4) to 0.

End For (each step)

8) Evaluate each ant tour's length (in GPU)

9) Refresh TAU with (2, 3, 4, 5) (in GPU):

The addition and comparison operations based on matrix in GPU are done in a parallel pattern.

10) Conditional re-initialization (in GPU)

End For (each iteration)

11) Collect final results (in CPU)

IV. Experimental Results and Analysis

The architecture used for the experiments is a workstation with an NVIDIA Tesla C1060 GPU(240 cores), an Intel i7 3.3 GHz, and 12GB memory. The algorithms are run under Windows 7 64-bit, MATLAB 2009B, Jacket 1.2.2, and CUDA 2.3. All the TSP instances used in this paper are taken from the TSPLIB benchmark library (available from http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/).

For many nature inspired problem solving techniques in Artificial Intelligence, parameter setting is a key issue in the performance of such a system [10]. The following parameter settings are used [1]:

α=1, β=2, ρ=0.02, m=n, where m is the number of ants.

For benchmarking, the algorithm using classic sequential MMAS procedures [2] (such as roulette wheel selection), with the same parameter settings was also implemented only on CPU in order to compare with the GPU+CPU version. Currently, Jacket toolbox lacks some key functions which are needed to employ the local search procedures efficiently, thus both of the algorithms are run without them.

A. Comparison of Solution Quality

Table 1 shows the time cost, the solution value, and the first reach iteration (indicates when the best solution value comes out at the first time) of MMAS being run on both the GPU+CPU and CPU platform to solve 3 TSP instances. All the tests are limited to 1000 iterations. All the results are based on 10 independent runs.

It can be seen that the solution results obtained by GPU+CPU are quite close to those obtained by CPU. But the computation times required by GPU+CPU are much less than those required by CPU version.

B. Computational Performance Analysis

The performance of element-by-element multiplication between the three matrices ($TABOO^k$, $DICE^k$, and $PROB^k$) is crucial. The analysis of it is done as follows: first, multiply three random square matrices of different sizes and measure the execution time in GPU without for statements; second, multiply three random square matrices of different sizes and measure the execution time in CPU with for statements which simulates the traditional roulette wheel selection procedures; finally, calculate the speedup of GPU-to-CPU. Fig.1 shows the experimental results when the matrix size is from 3×3 to 2500×2500 (relevant benchmark code available from [9]).

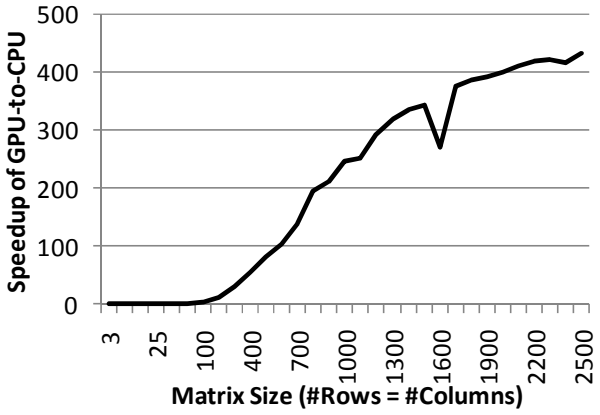

Fig.1. Speedup of GPU-to-CPU in multiplication between three matrices

C. Comparison of Computation Time

Fig.2 gives the comparison of overall computation time between the GPU+CPU and CPU implementation of the MMAS algorithm for 9 TSP instances. Due to the tiny percentage of initialization, all the tests are limited to 50 iterations.

The GPU can only work when the data is moved from CPU to it; the movement from GPU to CPU is also necessary. The absolute transfer time is linearly dependent on matrix size, and is with a small initialization penalty (less than a hundred microseconds).

Fig.3 shows that within our parallel MMAS algorithm, the percentage of time-consuming in data transfer increases steadily with the growth of the number of TSP nodes. For matrix size in the range 0.25-1.25MB, the transfer rate reaches the peak, which is above 3.5GB/s [9].

The results indicate that, when the TSP nodes are approximately less than 450, the GPU-to-CPU speedup increases steadily with the increase of number of TSP nodes. After the number of nodes reaches 450, the growth of speedup slows down significantly.

Although the speed-up of GPU-to-CPU in the generation of pseudorandom number matrix and the element-by-element multiplication increases with matrix size, some limitations (such as the absence of certain functions) of the current Jacket toolbox force us to frequently do data-transfer operations between GPU and CPU. Thus the bottleneck of our parallel algorithm should be the data transfer between GPU and CPU.

V. Conclusion and Future Work

In this paper, we presented a parallel ACO algorithm using MATLAB and Jacket toolbox with GPU to solve TSP problems. Compared with [6] and [7], our method is more suitable for GPU computing. And, with the help of Jacket

toolbox, it is oriented toward problem solving rather than toward GPU hardware details too much, thus simplifying the design of parallel ACO algorithm. Through the results and analysis, the classical ACO algorithm can be well adapted for fully matrix-based data-parallel computing in GPU within MATLAB. The computation time is greatly reduced while the solution quality nearly remains unchanged. Our method can also be used to solve other similar computationally intensive problems using MATLAB and Jacket toolbox.

Since we mainly aimed at the parallel accelerating property of the algorithm fitting GPU, refining the optimization results should be our next project. What's more, the next generation GPU applies a new parallel pattern named Multiple Instruction Multiple Thread (MIMT), and the upcoming release of new Jacket toolbox includes some features which we need urgently. So, the future work will focus on testing possible ways of using them to obtain better optimization results based on few shared matrices and to reduce the data transfer operations between GPU and CPU.

ACKNOWLEDGMENT

We wish to thank Dr. Torben Larson, the professor of Aalborg University, for his Jacket Performance Index and for the time he spent with us discussing subjects related to Jacket toolbox. We are grateful to Yun Liu, the professor of Wuhan Digital Engineering Institute, for valuable comments on an earlier version of this paper.

REFERENCES

[1] S. Thomas and H.H. Hoos, "MAX-MIN Ant System," *Future Generation Computer Systems*, 16(8): pp. 889-914, 2000.

[2] D. Marco and S. Thomas, *Ant Colony Optimization*. Cambridge: The MIT Press, 2004, ch. 4.

[3] D. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. San Fransisco: Morgan Kaufmann, 2010, ch. 1.

[4] NVIDIA. *Accelerating MATLAB with CUDA™ Using MEX Files*. 2007. Available: http://developer.nvidia.com/object/matlab_cuda.html

[5] AccelerEyes. *Jacket User Guide*. 2010. Available: http://www.accelereyes.com/content/doc/JacketUserGuide.pdf.

[6] W. Zhu and J. Curry, "Parallel ant colony for nonlinear function optimization with graphics hardware acceleration," *Proceedings of the 2009 IEEE international conference on Systems*, pp. 1803-1808, Oct. 2009.

[7] J. Li, X. Hu, Z. Pang, and K. Qian, "A Parallel Ant Colony Optimization Algorithm Based on Fine-Grained Model with GPU-Acceleration," *International Journal of Innovative Computing Information and Control*, vol. 5, no. 11, pp. 3707-3716, Nov. 2009.

[8] J. Wang, C. Zhang, and J. Dong, "Implementation of ant colony algorithm based on GPU," *Sixth International Conference on Computer Graphics, Imaging and Visualization*, pp. 50-53, Aug. 2009.

[9] T. Larsen, *Jacket Performance Index*. 2010. Available: http://wiki.accelereyes.com/wiki/index.php?title=Torben's_Corner

[10] L. Shi, J. Hao, J. Zhou, and G. Xu, "Ant colony optimization algorithm with random perturbation behavior to the problem of optimal unit commitment with probabilistic spinning reserve determination," *Electric Power Systems Research*, vol. 69, no. 2-3, pp. 295-303, May 2004.

TABLE 1. COMPARISON OF SOLUTION QUALITY BETWEEN GPU+CPU AND CPU

TSP	Version	Solution Value				First Reach Iteration				Time (sec)
		Best	Avg	Worst	Standard Deviation	Best	Avg	Worst	Standard Deviation	
eil51	GPU+CPU	426	428.6	431	1.5055	546	741	977	117	62
	CPU	427	428.9	430	0.8756	749	842	954	66	349
berlin52	GPU+CPU	7542	7544.0	7547	2.1082	708	870	992	89	65
	CPU	7542	7543.5	7547	1.9003	531	713	997	189	361
rat99	GPU+CPU	1211	1218.9	1230	5.9339	652	816	957	95	127
	CPU	1211	1215.0	1220	3.5590	755	878	996	78	1409

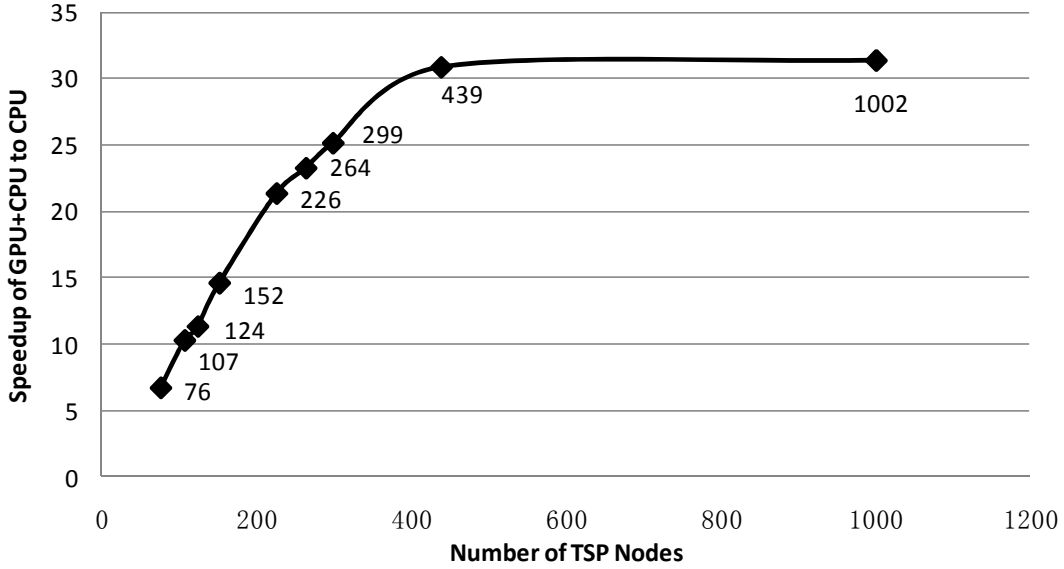

Fig.2. Overall Speedup of GPU+CPU to CPU. Running from pr76 to pr1002.

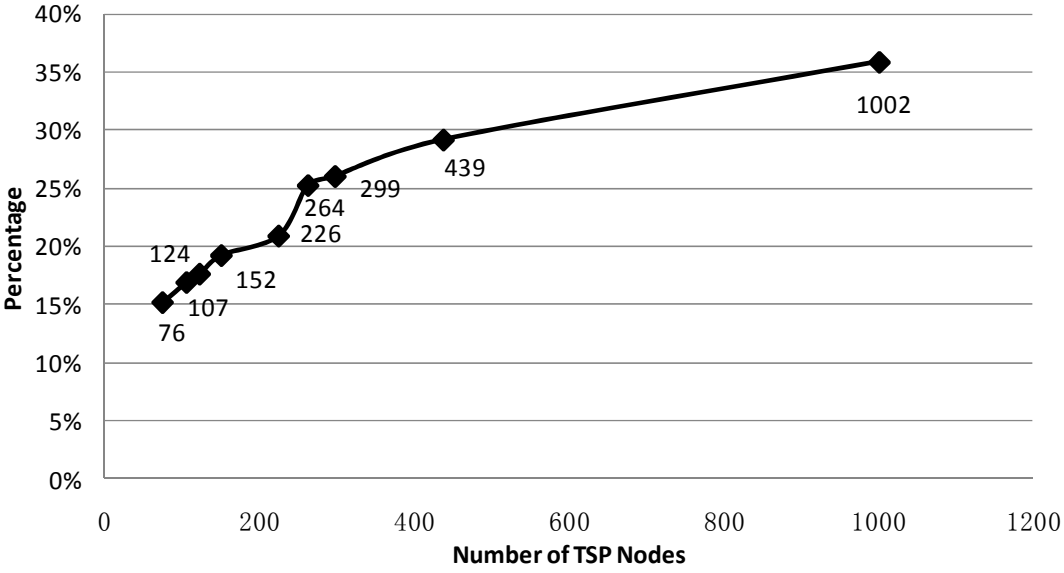

Fig.3. Percentage of time-consuming in data transfer within the parallel MMAS algorithm. Running from pr76 to pr1002.