An Implementation of Coincidence Algorithm on Multi-core Processors

W. Srimook and P. Chongstitvatana

Abstract— This paper presents an implementation of Coincidence Algorithm on multi-core processors. The algorithm is suitable to solve combinatorial problems. The implementation uses Intel Threading Building Blocks library for parallel computation. The performance improvement is measured using several Traveling Salesman Problems. The result shows that a speedup of a dual-core processor over a single-core processor is 76% and 230% for a quad-core processor.

Keywords— coincidence algorithm, evolutionary computation, multi-core system, parallel processing.

I. Introduction

FINDING an optimal solution of complex problems usually is a difficult task and mostly not very efficient on today computers. One popular approach for solving these problems is Evolutionary Algorithm (EA). Traveling Salesman Problem (TSP) represents a typical optimization problems of this class. Coincidence Algorithm (COIN) is an evolutionary algorithm specialized in combinatorial problems. TSP belongs to this class of problems. A typical EA consumes long running time, therefore to improve the execution time a parallel computation is introduced. Presently, the technology has changed very fast and the multi-core processors now already a common place in the market. This paper presents to an implementation of COIN algorithm with parallel programming on multi-core processors.

The structure of this paper is organized as follows. Section II presents the related work. Section III gives an overview the coincidence algorithm. Section IV presents a parallel implementation on multi-core processors. The experimental evaluation is shown in Section V and the conclusion is presented in Section VI.

II. Related Work

Parallel processing is well known today. Many work show how to improve the computing performance. For example, Hongzhong Shan proposes a hybrid programming for multicore processors [1]. His research used MPI and OpenMP programming model. He reported the result on performance and memory usage. The research investigates three problems: Lower-Upper Symmetric Gauss-Siedal (LU), Scalar Penta-diagonal (SP) and Block Tri-diagonal (BT). The memory usage depends on number of MPI process. The performance, when using MPI and OpenMP is better than pure MPI model. The limitation is when there is more MPI process, the load imbalance will occur. Parallel K-Nearest Neighbor Algorithm on CUDA-enabled GPU (CUNN) proposed by Shenshen Liang et al. [2]. The research uses two CUDA kernels, distance calculation and selection. The experimental evaluation shows that the processing time is reduced by 46.71 percent for the synthetic datasets and 42.49 percent for the physical simulation dataset. The I/O cost is included.

However to implement a parallel program for GPU, the developer has to use thread and manage a low level details such as block selection, thread control and data transfer between main memory and GPU memory. Salman Yussof et al. demonstrates a parallel genetic algorithm for shortest path routing problem [3]. Coarse-grained GA has been chosen and is implemented on MPI cluster. The computation time is faster than GA but the accuracy of an algorithm will decrease when using more computing nodes.

III. Coincidence Algorithm (COIN)

COIN was proposed in 2009 [4]. It is an algorithm in the class of Estimation Distribution Algorithms (aka a modern GA) which employs a model. The model in COIN is a Markov Chain Matrix. This matrix is used to generate the next generation population. COIN generates the population by probability of each coincidence. The representation of solution for TSP is a tour designated by a permutation of label of each city. A coincidence is a pair of adjacent labels. For example, we have path of ten cities TSP problem and represented by A, B, C, D, E, F, G, H, I and J.

```
ABCDEFGHIJ
```

From this tour, we have ten coincidences namely A-B, B-C, C-D, D-E, E-F, F-G, G-H, H-I, I-J and J-A back to the start point. Using these pair the probability in the matrix can be updated. The steps of COIN can be summarized as follows:

Step 1 Initialize the matrix
Step 2 Sampling the population
Step 3 Evaluate the population
Step 4 Select the candidates
Step 5 Update the matrix
Step 6 repeat Step 1-5 until converge

The most important step is the update step.

W. Srimook and P. Chongstitvatana are with the Computer Engineering Department, Engineering Faculty of Chulalongkorn University, Bangkok, Thailand (email: rathebest@gmail.com and prabhas.c@chula.ac.th)

A. The Generator

COIN algorithm used the generator for sampling the population and represented by matrix, H, of size $n \times n$ containing probability of each pair of member. In the proposed algorithm each H_{xy}, where x represented as a row and y represented as a column of the matrix, has a value between 0.0 to 1.0 (except the diagonal are always zero) and the sum over each row equals 1.0.

B. Initial the Generator

The generator, for the beginning, all H_{xy} except H_{xx} are updated with a value $\frac{1}{(n-1)}$ where n is a problem size.

C. Sampling the Population

The starting point is sampled, says it is x and the next sample point is y. This denotes the edge xy of the tour. The probability of sample depends on H_{xy}. The next edge is again generated from H starting from y. This process is repeat until the length of problem size (n) is reached. All population are generated in the same way.

D. Evaluate the Population

Every candidate is evaluated for it fitness by the function according to the objective.

E. Selection the Candidates

All of population is ranked by the fitness value. Then select c percent of top as a better-group and c percent of bottom as a worse-group.

F. Update the Generator

The better-group is use for reward by increase of H_{xy}. Every pair of xy founded in this group is use for update. The reward equation is:

$$H_{xy}(t+1) = H_{xy}(t) + \frac{k}{(n-1)}\left(r_{xy}(t+1)\right) - \frac{k}{(n-1)^2}\left(\sum_{j=1}^{n} r_{xj}(t+1)\right) \quad (1)$$

The punishment is to decrease of H_{xy} of any pair of xy founded in the worse-group. The punishment equation is:

$$H_{xy}(t+1) = H_{xy}(t) - \frac{k}{(n-1)}\left(p_{xy}(t+1)\right) + \frac{k}{(n-1)^2}\left(\sum_{j=1}^{n} p_{xj}(t+1)\right) \quad (2)$$

The new value of H_{xy} when founded in both group, the equation with combining from both is:

$$H_{xy}(t+1) = H_{xy}(t) + \frac{k}{(n-1)}\left(r_{xy}(t+1) - p_{xy}(t+1)\right) + \frac{k}{(n-1)^2}\left(\sum_{j=1}^{n} p_{xj}(t+1) - \sum_{j=1}^{n} r_{xj}(t+1)\right) \quad (3)$$

Where k denotes the learning step, n is the length of problem size, r_{xy} the number of xy founded in better-group, p_{xy} the number of xy founded in worse-group. The incremental and decrease step is $\frac{k}{(n-1)}$ and term $\frac{k}{(n-1)^2}\left(\sum_{j=1}^{n} p_{xj}(t+1) - \sum_{j=1}^{n} r_{xj}(t+1)\right)$ represents the step to adjust all other H_{xj} where $j \neq x$ and $j \neq y$ to keep the sum of each row remain to one.

IV. Implementation

Most programs in the past decades are written for serial computers running on a single computer with a single-core. In the present, the technology has been changed, the processor was developed with increasing number of cores and in the next 5-10 years the processor will comes out with tens or even a hundred of cores [5]. For use full account of processor performance the parallel programming is important.

A. Intel Threading Building Blocks (TBB)

This techniques for parallelism have been developed in many languages for ease of programming. The Intel Threading Building Blocks (TBB) library [6], [7] is one of parallelism library extension use in high level language (C++) for multithreads applications. Intel TBB is a task-base parallel developed by Intel. It operates by defining tasks that can be executed concurrently. Intel TBB performing parallelism by using the work-stealing task schedules to distribute tasks and balance a load between the available threads [8].

1) The Operation: Intel TBB for generic parallel algorithm using the *parallel_for*, *parallel_reduce*, *parallel_scan*, *parallel_do*, *pipeline*, *parallel_pipeline*, *parallel_sort* and *parallel_invoke*. The simplest of parallelism is a loop of iterations, in TBB *parallel_for* function template is used and also it provides *parallel_reduce* for calculation of a reduction and *parallel_scan* for calculation of a parallel prefix. TBB applies to a range of elements concurrently [8]. An advance algorithm for streams uses *parallel_do*, *parallel_sort* and *pipeline*. The *parallel_for* template works with a fixed number of independence loop iterations. It breaks iteration space into a chunk which run on separate threads. The first parameters of the template is a *blocked_range* object that describes the *lower bound* and the *upper bound* of the iteration space. The second parameter is the function object which calls the *body* object. The overload the *operator* method is applied to each subrange.

B. COIN Parallel on Multi-core Processors

The parallel computation for COIN algorithm takes place in three modules, 1) generate the population, 2) evaluate the population and 3) update the generator. This section shows the differences between the Sequential and the Parallel algorithm implemented by TBB.

1) Generate the Population: Each population is independent from the others. The goal of this module is to use full account the concurrency of population generating processed in the different tasks. Generation of the population can be fully parallelized. This algorithm generates, for each generation, the population by breaking the amount of population into subgroups for different tasks (thread). The process is illustrated in Figure 1. The implementation of the algorithm using TBB code shown in Figure 3. For parallelization of the algorithm, *parallel_for* of TBB library is used. In this code at line 19 creates the task scheduler object for mapping tasks to physical threads. The *parallel_for* function in line 22, the *block_range*, the first argument, defined whole range of the population. The grain size, the *iRange*, is defined by dividing with 10 in this case. The *Generate* class, the second argument, to generate the population is defined in lines 1 thru 8. The *parallel_for* invokes the *Generate* class on each subranges. At line 4, the call *range.begin()* method returns the start of subrange and *range.end()* methods returns the end of subrange. The sequential code of the *Generate population* is shown in Figure 2. The function uses a normal *for-loop*, in line 7, to generate the population.

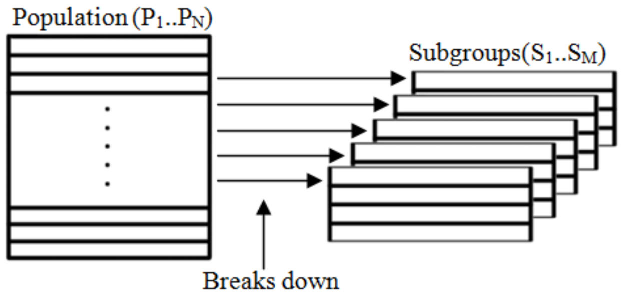
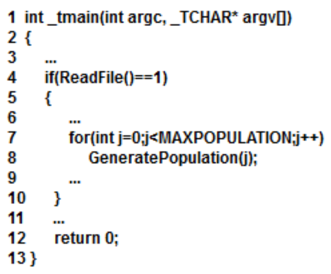
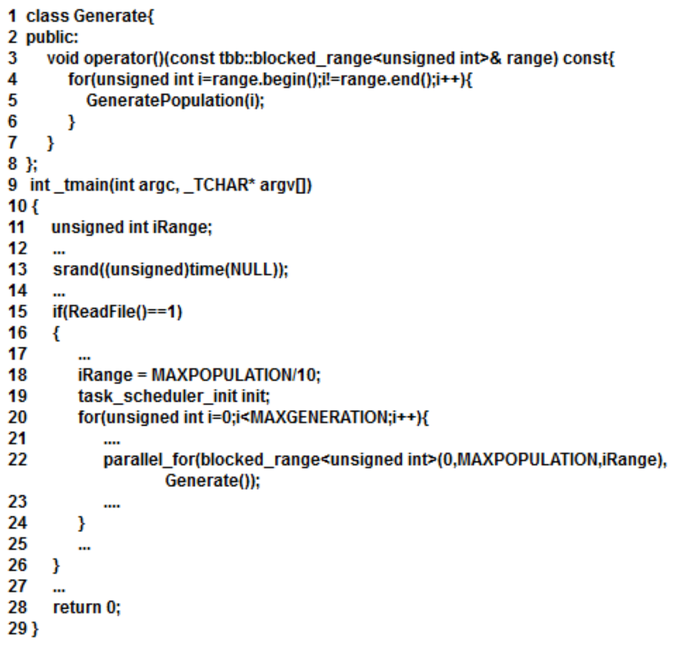


Fig. 1 Parallelization of *Generate the population* algorithm



```
1  int _tmain(int argc, _TCHAR* argv[])
2  {
3     ...
4     if(ReadFile()==1)
5     {
6        ...
7        for(int j=0;j<MAXPOPULATION;j++)
8           GeneratePopulation(j);
9        ...
10    }
11    ...
12    return 0;
13 }
```

Fig. 2 The sequential algorithm of *Generate the population*



```
1  class Generate{
2  public:
3     void operator()(const tbb::blocked_range<unsigned int>& range) const{
4        for(unsigned int i=range.begin();i!=range.end();i++){
5           GeneratePopulation(i);
6        }
7     }
8  };
9  int _tmain(int argc, _TCHAR* argv[])
10 {
11    unsigned int iRange;
12    ...
13    srand((unsigned)time(NULL));
14    ...
15    if(ReadFile()==1)
16    {
17       ...
18       iRange = MAXPOPULATION/10;
19       task_scheduler_init init;
20       for(unsigned int i=0;i<MAXGENERATION;i++){
21          ....
22          parallel_for(blocked_range<unsigned int>(0,MAXPOPULATION,iRange),
                     Generate());
23          ....
24       }
25       ...
26    }
27    ...
28    return 0;
29 }
```

Fig. 3 The *Generate population* algorithm implemented by TBB

2) Evaluate the Population: Once the population has been generated, evaluation of the population is performed to calculate the fitness value and to rank all candidates. The fitness value calculation is computed as the tour length of every candidates in the generation. With data-parallelism, we can concurrently calculate the fitness value of the population. First, the population is divided into subgroups map to TBB task (see Figure 4 (a)). Second, each subgroup contained multiple candidates and will be processed in a sequential method. The length of the tour is the sum of distance between each pair of the tour. Since pairs of the tour are independent, the distance calculation can be performed in a parallel (see Figure 4 (b)).

3) Update the Generator: Before generating the next population, the generator have to be updated. The algorithm to update the generator has two major steps. First, increase the probability of each pair found in the good group. Second, decrease the probability of each pair found in the worse group. The parallel algorithm to update the generator is described in two parts. Part one, traversing the matrix. We separate the matrix by row into subgroups for concurrent update. Part two, counting the number of occurrence of pairs from the selected candidates (good and worse groups). Counting method concurrently looks at the selected candidates then returns the final answer. The algorithm is illustrated in Figure 5.

TABLE I
PERFORMANCE OF COMPUTATION TIME COMPARE WITH SINGLE CORE CPU

COIN on CPU core(s)	TSP Problems							
	Eil 51		Eil 76		Eil 101		Ch 130	
	Time (seconds)	S.D.	Time (seconds)	S.D.	Time (seconds)	S.D.	Time (seconds)	S.D.
Single-core	38.84	0.30	115.65	2.38	252.42	17.53	531.23	25.97
Dual-core	21.80	0.50	66.83	0.36	152.38	1.38	309.87	3.87
Quad-core	12.08	0.47	35.06	1.32	77.72	1.40	160.32	2.84
Speed up of 2 cores	78.17%		73.05%		65.65%		71.44%	
Speed up of 4 cores	221.52%		229.86%		224.78%		231.36%	

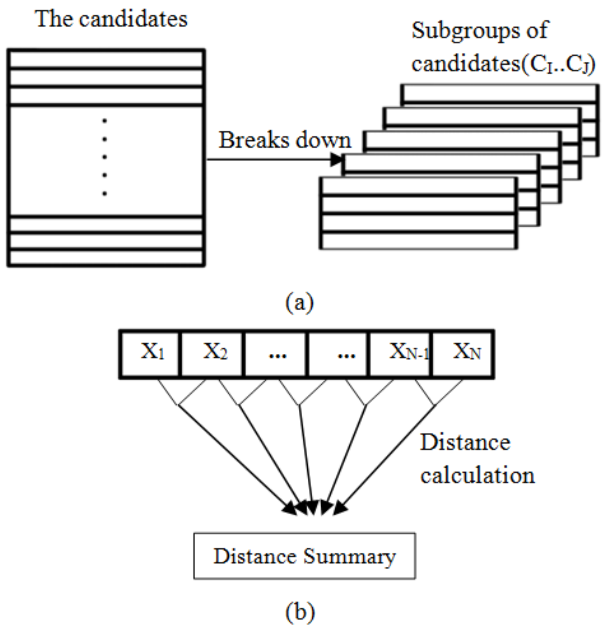


Fig. 4 Parallelization of *Evaluate the population* algorithm

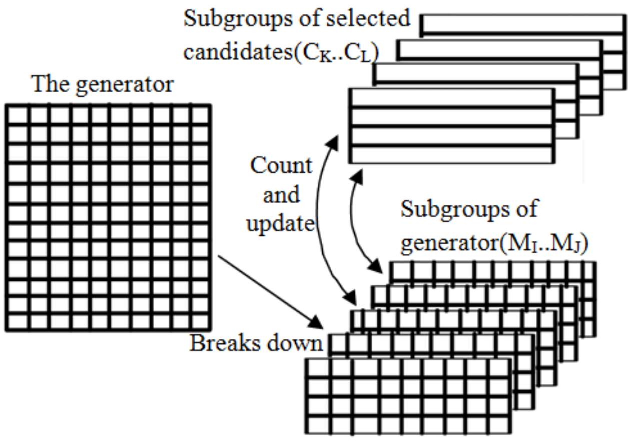

Subgroups of selected candidates(C$_K$..C$_L$)
The generator
Count and update
Subgroups of generator(M$_I$..M$_J$)
Breaks down

Fig. 5 Parallelization of *Update the generator* algorithm

V. EXPERIMENTAL EVALUATION

A. Experiment Environment

The environment used in the experiment is the processor with a quad-core 3.30 GHz Intel Core i5 and 8 MB main memory, running Windows 7 64-bit with service pack 1. The algorithm is implemented in C++ using Microsoft Visual Studio 2010 and Intel Parallel Studio XE 2011 with Intel TBB 3.0. The four TSP problems use symmetric TSPs data sets from TSPLIB [9]. The problems are Eil51, Eil76, Eil101 and Ch130. The main objective is to measure the performance of the computation time of the proposed algorithm. Computation time is obtained from the beginning to the end of the processing. The performance measurements are compared between a sequential version and parallel version of the same algorithm.

B. Result

The algorithm of COIN has a special parameter, the leaning step is set to 0.1. Other parameters use the same setting, the max generation is 500 and the population size is 500.

TABLE I shows the performance of the proposed algorithm for solving the TSP problems. The result shows a comparison of single-core with dual-core and single-core with quad-core. The speedup of the algorithm running on dual-core is 78% over single-core and the speedup of quad-core over single-core is 230%. The computation time reported the average value from 10 runs of the test. The standard deviation (S.D.) explains the spread of data, a low S.D. shows that the data values trend to be close to the mean, whereas high S.D. indicates that the data values are spread out in a large range. The S.D. of all four problems are small. That means the data points are reliable.

VI. CONCLUSION

This paper proposed to improve the performance of COIN algorithm with a parallel computation on multi-core platform. The result of the test by solving 4 TSP problems is faster than the sequential version of the algorithm. Based on the experiment, when solving the larger problem, the algorithm takes longer time but both of parallel implementation are still faster than the sequential implementation.

References

[1] Hongzhong Shan, "Hybrid Programming for Multicore Processors," in *2011 Fourth International Joint Conference on Computational Sciences and Optimization (CSO)*, Yunnan , 2010, pp. 261-262.

[2] Shenshen Liang, Ying Liu, Cheng Wang, and Liheng Jian, "Design and evaluation of a parallel k-nearest neighbor algorithm on CUDA-enabled GPU," in *2010 IEEE 2nd Symposium on Web Society (SWS)*, Beijing, 2010, pp. 53-60.

[3] S. Yussof, R.A. Razali, and Ong Hang See, "A Parallel Genetic Algorithm for Shortest Path Routing Problem," in *2009 International Conference on Future Computer and Communication*, Kuala Lumpur, 2009, pp. 268-273.

[4] W. Wattanapornprom, P. Olanviwitchai, P. Chutima, and P. Chongstitvatana, "Multi-objective Combinatorial Optimisation with Coincidence algorithm," in *2009 IEEE Congress on Evolutionary Computation*, Trondheim, 2009, pp. 1675-1682.

[5] http://www.intel.com/ [Online].

[6] J. Reinders, *Intel Threading Building Blocks, Outfitting C++ for Multi-core Processor Parallelism*.: O'Reilly, 2007.

[7] http://threadingbuildingblocks.org/ [Online].

[8] Wooyoung Kim and M. Voss, "Multicore Desktop Programming with Intel Threading Building Blocks," *IEEE Software*, vol. 28, no. 1, pp. 23-31, Jan.-Feb. 2011.

[9] http://comopt.ifi.uni-heidelberg.de/index.html [Online].