THE 8th INTERNATIONAL SYMPOSIUM ON ADVANCED TOPICS IN ELECTRICAL ENGINEERING
May 23-25, 2013
Bucharest, Romania

Impact of Problem Dimension on the Execution Time of Parallel Particle Swarm Optimization Implementation

O. Tolga Altinoz[1], *Graduate Member IEEE*, A. Egemen Yilmaz[2], *Member IEEE*, Gabriela Ciuprina[3], *Senior Member IEEE*
[1]TED University, Turkey
[2]Ankara University, Turkey
[3]Politehnica University of Bucharest, Romania
tolga.altinoz@tedu.edu.tr, aeyilmaz@eng.ankara.edu.tr, gabriela@lmn.pub.ro

***Abstract*- In this study, parallel particle swarm optimization algorithm has been investigated as regards the impact of the problem properties on the execution time. Two major factors affect the performance of parallel evolutionary algorithms: the population size and the problem dimension. In this study, five well-know benchmark functions have been applied with different dimensions. Then, these functions have been compared as regards the execution time. Finally, uniformly distributed population has been compared with the chaotic distributed population based on the dimension and population size from previous discussion.**

***Keywords*: particle swarm optimization, parallel computing, CUDA**

I. INTRODUCTION

Traditional computation structures, like central processing units (CPU), execute programs in series. They fetch the codes from the memory and execute them; this process is repeated until the program is terminated. Engineers have been working on faster architectures in order to speed up this process. However, instead of faster serial processes, recently parallel structures have been introduced due to the physical limits of semiconductor-based microelectronics manufacture. One of the easily-affordable and parallelizable structures is the graphic processing units (GPU).

Nowadays, GPUs, having large computation capacities and parallel structures, have been attracting many researchers of different disciplines. Instead of CPUs, GPUs have been designed to deal with large amount of computational processes in a short time, at once. Also, GPUs are specialized for computations contrary to the CPUs, which carry out storage and control tasks [1]. Besides, they have been developing more rapidly and frequently compared to CPUs with the aid of game and movie sectors [2].

The important improvement on this field was the introduction of CUDA (Compute Unified Device Architecture) in 2007 [1]. CUDA adds on some basic prefix to their sequential codes to convert them parallel functions [3]. Some parts of the sequential code can be divided into parallel modules, and CUDA supported environment is used to run the parallel code. By this way, the programmers can write computer programs easily with the aid of CUDA.

CUDA has become a common tool for parallel computing for many application areas, such as: dynamics of materials, fluid mechanics, computational electromagnetics [3], antenna design [4], medical imaging [5], nuclear engineering problems [6], biomechanical system identification [7], harmonic minimization of inverters [8], and document classification [9]. In addition to such application areas of CUDA codes, in this study, particle swarm optimization has been converted from the sequential form into a parallel form; this implementation will be called as parallel particle swarm optimization (P-PSO) throughout this paper.

In this paper, the impacts of the problem dimension and population size on the P-PSO have been investigated and discussed as regards to the execution time. Then, a variant of P-PSO is executed. This new version of P-PSO, which will be referred to as chaotic P-PSO (CP-PSO) from now on, is reproduced by changing random initialized of the position vectors with the aid of logistic map. The comparison has been made by considering the solution accuracy and average execution time.

This study has been presented in 4 sections following the introduction. In section 2, general programming features of the CUDA have been given briefly. PSO, P-PSO, and chaotic CP-PSO has been explained in section 3. Implementation results have been discussed in section 4. Comparisons together with relevant plots have been given in this section. The last section is the conclusion of this paper.

II. FUNDAMENTALS OF PARALLEL PROGRAMMING WITH CUDA

The researchers familiar with the C/C++ programming language used to desire a programming language or framework letting them write parallel codes easily. For this purpose in 2007, NVidia [1] introduced a software framework called CUDA. By means of this, a sequential function code can be converted to a parallel *kernel* by using the libraries and some prefix expressions. By this way, the programmers do not need to learn a new programming language. They are able to use their previous know-how related to C/C++, and enhance this knowledge with some basic expressions introduced by CUDA. However, without the knowledge

978-1-4673-5980-1/13/$31.00 ©2013 IEEE

about the CUDA software and the parallel architecture hardware, it is not possible to write efficient codes.

CUDA programming begins with the division of the architectures. It defines the CPU as *host* and the GPU as *device*. The parallel programming actually is the assignment of duties to parallel structure and collection of the results by CPU. In summary, the codes are written for CPU on C/C++ environment, and these codes include some parallel structures. These codes are executed by *host*. *Host* commands *device* for code executed. When the code is executed by the *device*, the *host* waits until the job is finished then a new parallel duty can be assigned, or results from the finished job can be collected by the host. Thus, *device* becomes a parallel computation unit. Hence, parallel computing rely on the data movement between *host* and *device*. Even both *host* and *device* are very fast computation units, the data bus is slower. Therefore, in order to write an efficient program, the programmer must keep his/her code for minimum data transfer between the host and the device.

The GPU has stream multiprocessors (SMs). Each SM has 8 stream processors (SPs), also known as cores, and each core has a number of threads. In a *tesla* architecture there are 240 SPs, and on each SP has 128 threads, which is the kernel execution unit. The bodies of threads are called groups. The groups are performed collaterally with respect to the core size. If the GPU architecture has two cores, then two blocks of threads are executed simultaneously. If it has four cores, then four blocks are executed collaterally.

Host and device communicate via data movement. The host moves data to the memory of the GPU board. This memory is called global memory which access from all threads and the host. The host can also access to constant and texture memories. However, it cannot access the instead of to shared memory, which is a divided structure assigned for every block. The threads within the block can access their own shared memory. The communication of the shared memory is faster than the global memory. Hence, a parallel code must contain data transfers to shared memory more often, instead of global memory.

In this study, random numbers are needed to execute the algorithm. Hence, instead of the *rand()* function of the C/C++ environment, CURAND library of the CUDA pack has been employed. In addition, the CUDA Event is preferred for accurate measurement of the execution time. In the next section, the parallel particle swarm optimization algorithm will be presented.

III. PARALLEL PARTICLE SWARM OPTIMIZATION

Particle swarm optimization (PSO) algorithm was introduced in 1995 by Kennedy and Eberhart [10]. The algorithm consists of four fundamental steps. The first step is the initialization. Before beginning the iterations of the algorithm, the positions and velocities of the population members are assigned randomly. The population spreads randomly on the search space. In this paper, two cases of population position initialization are discussed. In the former case, the population is distributed randomly by using the uniform pseudo-random number generator. In the latter case, the logistic map is preferred for random number generation. A sequence of random numbers must be truly random. Hence, such a sequence can be obtained from physical processes, which have nonlinear behavior, sensitive to initial conditions and system parameters. Therefore, the logistic map is selected as a non-linear equation with chaotic behavior. Equation (1) presents the logistic map function (Fig. 1).

$$x_{i+1} = v x_i \left(1 - x_i\right) \tag{1}$$

where v is set to 4 for ergodicity, which means long-term predictions of x_i are not possible. After the initialization phase, the iteration begins and continues until the maximum number of iteration is reached. The iterations begin with the computation of the relevant benchmark function. Namely, the benchmark function is evaluated at the position of each particle. Hence, throughout this study, benchmark function evaluations are assigned and performed at the kernel.

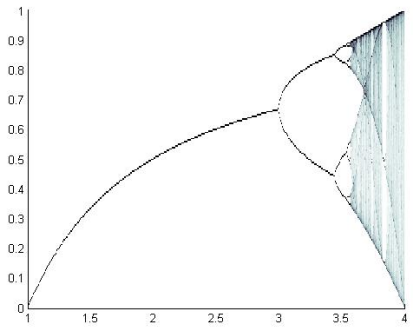

Fig. 1. The bifurcation diagram of logistic map.

The cost value and the position vector of each particle are stored in the memory of GPU board. These vectors are taken to find the personal best and global best values. It is assumed that each particle has a memory. However, each of them can remember only one vector. This vector is the its minimum/maximum cost value until the current iteration. If the new cost value is better than the one at their memory, the new vector is replaced. The global best value is the best position vector of the population at the current iteration. The CUDA functions can be used in this phase to find minimum or maximum values. These values are not stored/remembered. The last part of the algorithm is the position and the velocity update formulations. The same equations are applied to each particle. Hence, this part can be converted into parallel kernel similar to the benchmark function evaluation phase. Fig. 2 gives these steps as a flow diagram.

IV. IMPLEMENTATION RESULTS

The performance of the P-PSO has been investigated based on incremental of population size and dimension with respect to the execution time. After this process, two methods for

position initialization have been compared with reference to the cost value accuracy and execution time. For this purpose, five well-known benchmark functions have been selected where table I [11] presents the mathematical models of these functions.

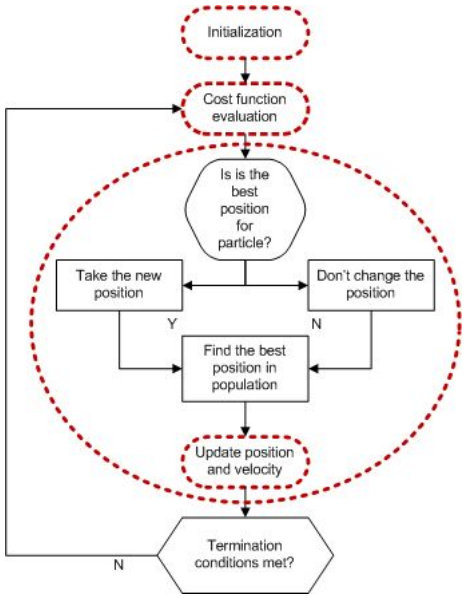

Fig. 2. The flow diagram for the particle swarm optimization, blocks inside the dashed circles are executed in parallel.

TABLE I
BENCHMARK FUNCTIONS

Mathematical Equation	Optimum	Search Space
$f_1 = \sum_{i=1}^{D} x_i^{\,2} - 450$	-450	$[-100,100]^D$
$f_2 = \sum_{i=1}^{D} \left(\sum_{j=1}^{i} x_i \right)^2 - 450$	-450	$[-100,100]^D$
$f_3 = \sum_{i=1}^{D} \left(10^6\right)^{\frac{i-1}{D-1}} x_i^{\,2} - 450$	-450	$[-100,100]^D$
$f_4 = \sum_{i=1}^{D} \left(x_i^{\,2} - 10\cos(2\pi x_i) + 10 \right) - 330$	-330	$[-5,5]^D$
$f_5 = -20\exp\left(0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D} x_i^{\,2}} \right) - \exp\left(\frac{1}{D}\sum_{i=1}^{D} \cos(2\pi x_i) \right) + 20 + e - 140$	-140	$[-32,32]^D$

The implementations were tested on a GTX 550Ti GPU in PC with AMD FX 3150 processor. First, the P-PSO algorithm has been implemented (with 1000 iterations) as explained in the previous section. The impact of the population size and dimension on P-PSO has been investigated. For this aim, the dimension and the population size of all benchmark algorithms have been altered. The algorithm has run 20 times independently, and population size has been changed from 10 to 10000. The results are categorized as maximum, minimum, average, and standard deviation of the execution time. Table II presents these results.

TABLE II
THE IMPACT OF THE POPULATION SIZE ON THE EXECUTION TIME (MS) OF THE P-PSO ALGORITHM FOR 20 INDEPENDENT MONTE CARLO

Pop.Size	Function	Max.	Min.	Average	Std.Dev
10	f_1	253,57	240,34	246,57	4,26
	f_2	254,63	241,83	249,67	3,55
	f_3	285,78	270,96	279,37	4,61
	f_4	297,23	294,79	297,67	1,52
	f_5	332,55	316,94	327,69	4,92
50	f_1	260,39	256,07	260	2,55
	f_2	267,11	254,86	259,23	3,55
	f_3	301,06	290,59	297,37	4,79
	f_4	315,23	308,18	310,76	1,91
	f_5	347,36	339,84	344,78	3,69
100	f_1	269,09	250,17	260,21	7,46
	f_2	268,14	265,47	269,6	2,74
	f_3	306	294,07	304,65	5,26
	f_4	312,74	306,28	310,18	2,34
	f_5	345,57	340,09	344,84	2,64
500	f_1	267,26	261,84	265,92	2,68
	f_2	267,6	262,32	266,9	2,89
	f_3	322,06	296,82	345,66	29,48
	f_4	320,64	313,57	318,78	2,42
	f_5	372,23	350,3	362,17	7,06
1000	f_1	270,99	263,36	268,19	2,13
	f_2	280,55	272,65	278,1	3,48
	f_3	491,57	309,94	387,16	58,97
	f_4	325,34	319,56	326,68	4
	f_5	372,48	353,37	374,87	14,73
2000	f_1	289,04	278,95	289,62	9,95
	f_2	297,21	288,36	298,8	10,58
	f_3	720,51	397,54	498,08	106,41
	f_4	395,71	382,73	395,54	8,19
	f_5	495,6	439,54	493,36	35,85
4000	f_1	343,36	324,63	332,85	4,88
	f_2	350,55	335,88	346,66	5,79
	f_3	680,26	442,22	626,17	164,25
	f_4	514,82	478,25	493,85	11,25
	f_5	617,32	509,69	598,2	73,15
5000	f_1	380,33	369,43	378,77	5,47
	f_2	389,91	376,11	381,36	3,42
	f_3	639,53	515,27	559,66	50,87
	f_4	584,99	545,01	565,12	14,82
	f_5	749,27	647,24	706,84	41,87
6000	f_1	397,05	385,59	393,53	3,32
	f_2	412,77	402,24	407,6	3,17
	f_3	642,38	542,02	585,22	29,43
	f_4	609,93	593,48	615,63	16,45
	f_5	769,84	708,19	814,04	147,88
8000	f_1	520,09	505,77	512,21	3,26
	f_2	533,53	521,78	526,86	3,43
	f_3	758,5	685,05	740,43	42,21
	f_4	780,46	770,32	802,74	26,08
	f_5	1038,9	920,26	978,05	62,14
10000	f_1	643,19	593,15	626,82	26,14
	f_2	675,71	603,46	628,82	23,36
	f_3	960,39	797,9	849,97	47,37
	f_4	952,67	889,71	989,24	117,11
	f_5	1453	1066,7	1279,2	143,65

Table II presents the execution time comparison for different population sizes, and Fig. 3 reveals the graphical

description of average time of the results given in table II. Also, Fig. 4 shows the average time for population sizes between 0 and 2000. As seen in these results, as population size increases, the execution time increases. The percentage increase in the execution time is presented in Table III.

TABLE III
PERCENTAGE INCREASE IN THE EXECUTION TIME WITH RESPECT TO THE CHANGE OF THE POPULATION SIZE

Size	f_1	f_2	f_3	f_4	f_5
10-50	5,44%	3,82%	6,44%	4,39%	5,21%
50-100	0,08%	4,00%	2,44%	-0,18%	0,01%
100-500	2,19%	-1,00%	13,46%	2,77%	5,02%
500-1000	0,85%	4,19%	12,00%	2,47%	3,5%
1000-2000	7,99%	7,44%	28,64%	21,07%	31,60%
2000-4000	14,92%	16,01%	25,71%	24,85%	21,25%
4000-5000	13,79%	10,00%	-10,62%	14,43%	18,16%
5000-6000	3,89%	6,88%	4,56%	8,93%	15,16%
6000-8000	30,15%	29,25%	26,52%	30,39%	20,14%
8000-10000	22,37%	19,35%	14,79%	23,23%	30,79%
10-10000	154,21%	151,86%	204,24%	232,32%	290,36%

Table III shows that, when the mathematical description of the benchmark function becomes complicated or the population size is raised, the execution time increases. Fig. 3 and Fig. 4 reveal that there is a rapid rise at the execution time at population sizes between 2000 and 8000. Therefore, it can be suggested that a good selection of the population size could be 1000. Hence, the population size of 1000 has been chosen as the reference throughout the analyses regarding impact of the problem dimension on execution time.

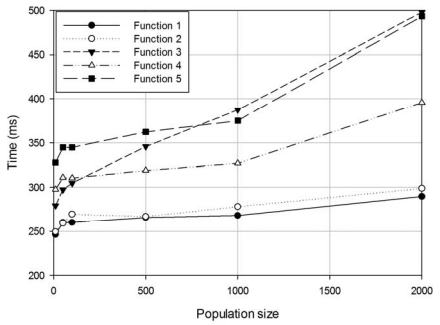

Fig. 3. Graphical description of average time of the results gives in Table II for population sizes between 0 and 2000.

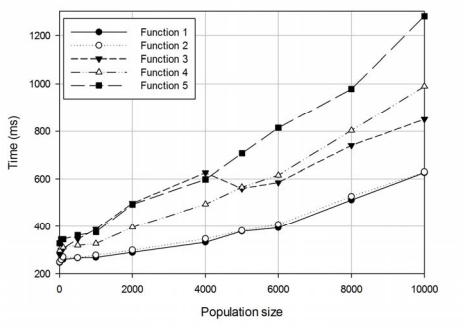

Fig. 4. Graphical description of average time of the results given in Table II.

The next analysis is the comparison of the execution time with respect to the change in the problem dimension. For this analysis, the population size is selected as 1000, and number of independent Monte Carlo runs has been selected as 20. At each independent run, initial position of the Logistic map begins with the random number. Table IV presents the impact of the dimensions between 10 and 80 on the execution time. As expected, as the dimension of the problem is raised, the execution time also increases, as shown in Fig. 5.

TABLE IV
THE IMPACT OF THE DIMENSION BETWEEN 10 AND 80 ON THE EXECUTION TIME (MS).

Pop.Size	Function	Max.	Min.	Average	Std.Dev
10	f_1	273,48	264,59	269,64	2,17
	f_2	273,39	266,74	270,2	2,24
	f_3	520,16	388,51	465,78	46,93
	f_4	483,33	475,04	483,54	4,03
	f_5	383,62	358,24	384,08	12,59
20	f_1	325,37	317,82	324,64	7,72
	f_2	328,28	318,9	323,73	2,43
	f_3	607,82	448,7	513,52	48,73
	f_4	534,53	523,77	529,63	3,26
	f_5	429,65	409,87	430,88	13,66
30	f_1	386,1	371,02	378,33	3,96
	f_2	384,68	372,58	378,38	3,97
	f_3	672,91	486,91	563,8	59,85
	f_4	590,17	575,31	586,72	5,6
	f_5	491,34	461,95	488,26	16,09
40	f_1	440,54	430,57	437,31	4,86
	f_2	443,12	431,71	438,69	3,44
	f_3	617,47	562,68	623,74	43,69
	f_4	650,43	642,48	648,81	3,4
	f_5	535,04	516,39	538,49	17,57
50	f_1	504,95	489,72	494,69	3,24
	f_2	495,3	489,56	493,78	2,57
	f_3	635,34	611,21	678	56,45
	f_4	710,73	690,44	700,18	7,32
	f_5	593,41	570,76	603,98	18,1
60	f_1	560	542,1	546,6	4,26
	f_2	550,8	542,47	547	2,86
	f_3	745,09	672,13	747,27	53,18
	f_4	750,47	738,96	748,83	6,32
	f_5	495,6	439,54	493,36	35,85
70	f_1	676,29	618,39	664,36	11,55
	f_2	679,05	595,54	620,74	33,22
	f_3	832,58	738,64	788,74	42,14
	f_4	806,67	791,13	797,9	5,25
	f_5	720,45	673,97	708,76	17,46
80	f_1	739,75	731,41	735,71	2,82
	f_2	731,09	718,04	723,88	3,91
	f_3	935,5	834,78	921,24	50,08
	f_4	928,45	910,1	922,147	10,67
	f_5	869,42	772,75	820,49	21,21

Table V presents the percentage of the change in the dimension. This table reveals that, the change of average execution time along with the problem dimension variation is almost linear. As seen in table V and table III, the change of population size and dimension have no relations. For functions f_3 - f_5 the impact of the dimension on the execution time is larger than the effect of the population size. Also, for functions f_1 and f_2, the situation is opposite. The main reason is nothing but the complexity of these functions.

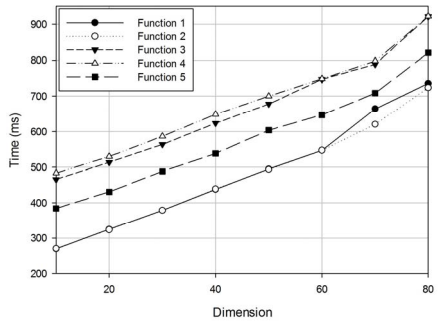

Fig. 5. Graphical description of impact of the dimension change on execution time (ms).

TABLE V
PERCENTAGE INCREASE IN THE EXECUTION TIME WITH RESPECT TO THE CHANGE IN PROBLEM DIMENSION

Size	f_1	f_2	f_3	f_4	f_5
10-20	20,39%	19,81%	10,24%	9,53%	12,18%
20-30	16,53%	16,88%	9,79%	10,77%	13,31%
30-40	15,58%	15,93%	10,63%	10,58%	10,28%
40-50	13,12%	12,55%	8,69%	7,91%	12,16%
50-60	10,49%	10,77%	10,21%	6,94%	7,13%
60-70	21,54%	13,48%	5,54%	6,55%	9,53%
70-80	10,73%	16,61%	16,79%	15,57%	15,76%
10-80	172,84%	167,9%	97,78%	90,71%	113,62%

The mathematical formulations of functions f_3 - f_5 have many linear/nonlinear parts. These parts must be executed in order to complete the equation. However, each thread executes only one function in parallel, and this must be repeated. In conclusion, parallel process must be repeated for different mathematical operators. For example; the function f_5 has two sum operators; however these can be spread into threads, which are executed in parallel. However, the results also need to operate on basic functions, like multiplication. This process also needs to wait until the first parallel thread is finished. After that, the exponential power should be executed. For that reason, for the average execution time, the population size is a more dominant factor compared to the problem dimension. This is due to the fact that for each member of the population, these processes must be executed, even if they are parallel. However, considering the problem dimension, only the number of threads increases; this means that the number of parallel slaves increases.

The last part of the implementation section is the comparison between two different parallel particle swarm optimization methods. In the first method (as implemented previous discussion), initial swarm position vector spread into the search space by using the uniformly distributed random numbers. But, the second method implements the logistic map, which is a chaotic map distributing the population. The comparison is again based on the solution quality and the execution time.

The algorithms have implemented for a population size of 1000, problem dimension of 100, and the number of iterations is 100. Throughout the comparisons, 20 independent Monte Carlo runs are performed. For comparison, the following values are considered: maximum, minimum, average, standard deviation of cost values, and the average execution time. Tables VI and VII give the quality of the solution based on these criteria. The results show that, the chaotic map increases the execution time; however, it improves the quality of the solution. The improvement in the solution quality due to chaotic maps is observed by smaller cost (maximum, minimum and average) values and smaller standard deviations, which points to robustness of the CP-PSO compared to P-PSO

TABLE VI
CHAOTIC PARALLEL PARTICLE SWARM OPTIMIZATION (POPULATION SIZE: 1000, NUMBER OF ITERATION: 100, PROBLEM DIMENSION: 100, NUMBER OF INDEPENDENT MONTE CARLO RUNS: 20)

Func.	Max.	Min.	Average	Std.Dev.	Av.Time (ms)
f_1	-448,9	-449,1	-442,2	2,39	88,2
f_2	-425,7	-438,6	-438,6	5,55	90,6
f_3	-434,4	-442,5	-442,5	3,15	93,7
f_4	-425,3	-428,42	-428,42	4,26	96,06
f_5	-119,9	-138	-138	1,4	105,7

TABLE VII
PARALLEL PARTICLE SWARM OPTIMIZATION (POPULATION SIZE: 1000, NUMBER OF ITERATION: 100, PROBLEM DIMENSION: 100, NUMBER OF INDEPENDENT MONTE CARLO RUNS: 20)

Func.	Max.	Min.	Average	Std.Dev.	Av.Time (ms)
f_1	-442,9	-442,2	-442,2	1,92	85,1
f_2	-438	-449,8	-440,4	8,75	85,4
f_3	-447,9	-446,8	-446,8	3,73	97,54
f_4	-407,3	-429,9	-412,2	14,29	90,26
f_5	-119,5	-120,3	-119,7	0,32	98,9

V. CONCLUSION

The results of this study were presented in two main parts. In the first part, the performance of the parallel particle swarm optimization was discussed. Primarily, the problem dimension and the number of iterations were held as 10 and 1000, respectively. Then, the population size was changed and the execution time of the algorithm was observed. When the population size was raised, the execution time also increased. It was seen from the results that, until the population size became 1000, there was approximately a linear variation. However, for the population sizes larger than 1000, the increase in the execution time becomes quite dramatically. Then, the variation of the execution time along with the problem dimension was investigated. When the dimension was raised, the execution time also increased.

In the second part of the study, the comparison between conventional initialization (i.e. positioning of the particles by means of uniformly distributed random numbers) and chaotic initialization (i.e. chaotic positioning of the particles by means of logistic map generated random numbers) was performed. The results showed that the chaotic P-PSO

outperforms to P-PSO in terms of solution quality, but it requires *5-7%* more execution time.

As a future study, the authors plan to apply P-PSO and CP-PSO methods to the time consuming engineering problems which are electromagnetic design of superconducting magnetic energy storage configuration and antenna design, and propose complexity analysis for PSO, P-PSO and CP-PSO algorithms.

ACKNOWLEDGMENT

This study is made possible by a joint grant from TUBITAK (with Grant Nr. 112E168) and ANCS-UEFISCDI (with Grant Nr. 605/01.01.2013). The authors would like to express their gratitude to these institutions for their support.

REFERENCES

[1] NVIDIA Corporation, "CUDA dynamic parallelism programming," NVIDIA, 2012.

[2] D.D.Donno, A.Esposito, L.Tarricone and L.Catarinucci, "Introduction to GPU computing and CUDA programming: A case study on FDTD," *IEEE Antennas and Propagation Magazine*, Vol. 52, No. 3, pp. 116-122, 2010.

[3] J. Nickolls and W.J.Dally, "The GPU computing era," *IEEE Micro*, Vol. 30, No. 2, pp. 56-69, 2010.

[4] D.Gies and Y.Rahmat Samii, "Reconfigurable array design using parallel particle swarm optimization," *International symposium on Antenna and Propagation Society*, Vol. 30, No. 2, pp. 177-180, 2003.

[5] M.Garland, S.Garland, J.Nickolls, J.Anderson, J,Hardwick, S.~Morton, E.Philips, Y.Zhang and V.Volkov, "Parallel computing experiences with CUDA," *IEEE Micro*, Vol. 23, No. 4, pp. 13-27, 2008.

[6] M.Wainrraub, R.Schirru and C.Pereira, "Multiprocessor modelling of parallel particle swarm optimization applied to nuclear engineering problems," *Nuclear Energy*, Vol. 51, No. 1, pp. 680-688, 2009.

[7] J.F.Schutte, J.A.Reinbolt, B.J.Fregly, R.T.Haftka and A.D.George, "Parallel global optimization with the particle swarm algorithm," *Journal of Numerical Methods in Engineering*, Vol. 61, No. 1, pp. 2296-2315, 2003.

[8] V.Roberge and M.Tarbouchi, "Efficient parallel particle swarm optimizers on GPU for Real-Time Harmonic Minimization in Multilevel Inverters," *IECON International Conference on IEEE Industrial Electronic Society*, pp. 2275-2282, 2012.

[9] J.Platos, V.Snasel, T.Jezowicz, P.Kromer and A.Abraham, "A PSO-Based document classification algorithm accelerated by the CUDA Platform," *IEEE International conference on System, Man, and Cybernetics*, pp. 1936-1941, 2012.

[10] J.Kennedy and A.Eberhart, "Particle swarm optimization," *IEEE International Conference on Neural Networks*, pp. 1942-1948, 1995.

[11] P.N.Suganthan, N.Hansen, J.J.Liang, K.Deb, Y.P.Chen, A.Auger and S.Tiwari, "Problem definitions and evaluation criteria for the CEC 2005 special session on real-parameter optimization," Technical Report Nanyang Technological University, 2005