Accelerating of Program Execution by Discrete Particle Swarm Optimization Method

Sergii Sushko
PhD student
Pukhov Institute for Modeling in Energy Engineering, NASU
Kyiv, Ukraine
sergii.sushko@gmail.com

Alexander Chemeris
Dept. of Modelling and Econometrics
Pukhov Institute for Modeling in Energy Engineering, NASU
Kyiv, Ukraine
a.a.chemeris@gmail.com

Svetlana Reznikova
Dept. of Modelling and Econometrics
Pukhov Institute for Modeling in Energy Engineering, NASU
Kyiv, Ukraine
svetlana.reznikova@gmail.com

***Abstract*— The paper describes actual methods of software optimization. Main attention is dedicated to two optimization methods: tiling and parallelization. The paper is focused on tailoring of tiling method to accelerate speed of software. The authors proposed to use discrete particle swarm optimization method to search better sizes of tiling method. Results of the investigations are shown. As a conclusion the authors distinguish an applicability of discrete particle swarm optimization method in the task of selection optimizations' parameters.**

Keywords— parallelization; software optimization; tiling; discrete particle swarm optimization; loop optimization; energy efficient computing

I. INTRODUCTION

During last years optimization of software was constantly solving task. Optimization of software allows improving one or several optimized parameters without loss of accuracy of computations [1].

In common global optimization task can be divided onto 3 levels of optimization:

1. High level optimization - optimization of algorithm level;

2. Middle level optimization - optimization of commands' sequence;

3. Low level optimization - optimization of binary code and CPU command.

Every level is important in common efficiency but these levels are related to the different areas of developing of software [2]. For example, high level is related to algorithmic developers but low level related more to compiler.

Middle level consists of huge amount of different optimization methods. Some of them are embedded to compilers but some of them can be manually implemented by developers. Most of optimizations methods of this type are applied on computational loops because even small improvement in loop can cause great outcome. Some of such optimizations methods are as follow: fission/distribution, fusion/combining, interchange/permutation, inversion, loop-invariant code motion, parallelization, reversal, scheduling, skewing, software pipelining, splitting/peeling, tiling/blocking, vectorization, unrolling, unswitching. Some of these methods can be used together but some of them are opposite.

II. SOFTWARE OPTIMIZATION PROBLEM

Any software can be considered as the finite quantity of source code parts. Any optimization can be used to all parts or several parts of source code. Optimization itself can be treated as a finite set of optimizations methods with finite set of optimization parameters.

Let consider for every part of source code i that F_i are parameters that should be optimized, M is vector of the optimization methods, P is vector of the parameters of the optimization vectors.

$$Fi = argmin(f(M,P)) \quad (1)$$

where *Fi* is the required parameter of the optimization, *M* is the vector of the optimization methods, *P* is the vector of the parameters of the optimization methods. argmin is defined as:

$$argmin(f(x)) \in \{x | \forall y: f(x) \leq f(y)\}. \quad (2)$$

Optimization problem for entire program with *N* computational blocks can be defined as:

$$F = argmin \sum_{i=1}^{N} f(\vec{M}_i, \vec{P}_i)$$

$$F = argmin\left(\sum_{i=1}^{N} f(\overset{r}{\vec{M}_i}, \overset{r}{\vec{P}_i})\right) \quad (3)$$

It means that every particular part of the source code could have some unique set of the methods and its parameters to achieve the best possible productivity.

There are many methods of the optimizations. In terms of time processing it's obvious that better candidates for optimization are located in the computational loops. It's because body of loop is executed many times. Thus, small improvement of the computational loop can lead to the significant effect.

All operations are executed in the integer basis which defined and limited by loop indices. They define dimension and size of iteration space. Iteration space is a set of all integer vectors $I = (I_1, I_2, \ldots, I_n)$ that satisfy inequalities

$$L_i \leq x_i \leq U_i,, \quad (4)$$

where i=1, 2,..,n. Inequalities (4) define loops' bounds and they restrict the iteration space by the convex polyhedron.

It means that to optimize any part of the source code some particular methods of optimization with some parameters has

to be chosen. Parameters which optimized can be evaluated sometimes after the compilation, for example it can be size of used program memory. Result of optimization in other cases can be performed only after the runtime execution of program. Execution time and energy consumption can be exactly obtained after the program run only.

III. Parallelization and Tiling Methods

At the moment there a lot of optimization methods may be used to optimize the result and effectiveness of parallelization.

One of the most perspective optimization methods is parallelization. This method allows dividing of single threaded code by several independent parts which can process simultaneously by big amount of computational units or more often by CPUs. Recent desktop CPUs usually have 4, 8 or even 16 cores. Server CPUs has tens and hundreds of cores. It means that by using proper programming parallelization paradigm it's possible to dramatically boost a performance comparing with single core.

Another one of efficient but not evident method is a tiling method. Tiling method divides a loop's iteration space into smaller blocks. This approach allows data used in a loop to stay in cache. It enhances cache reuse and reduces cache size requirements. A simple loop

```
for (i=0; i<N; i++)  {  ...  }
```

can be tiled with a tile size B by replacing it:

```
for (j=0; j<N; j+=B)
    for(i=j; i<min(N, j+B); i++)  {  ...  },
```

where min() is a function that returns the minimum of loop parameters.

Different approaches can be used for mathematical description of computing loops. One of the most well-knows and a vivid method of representation of computing loops is a polyhedral model or method of polygons [3]. This model allows representing any computing loop as a certain mathematical abstraction level [4]. Next, such model can be modified in any way that does not change of output results. Also model can then be converted back to source code. This optimization approach is flexible with choice of optimization methods. At the same time, this approach to source code optimization has also advantage that developer can be concentrated on a task of loop transformations only, rather than developing highly complex binary code compiler. Compiler will perform all the additional optimizations at its own level after transformation to the optimized source code.

Another important feature of tiling method is that it can be used together with parallelization method very effectively if no dependencies between tiles [5].

For the verification of tiling method test bench was built. It includes 4 cores desktop Intel Core I5-4670K processor and Pluto software tool [6]. Pluto software tool allows automatically perform tiling method on selected loops. Several types of tiling methods are available which can be used stand alone or with parallelization simultaneously. As source of test algorithms a software Polybench test pack was used [7]. It includes 30 tests from linear algebra, simulation, matrix computing and so on.

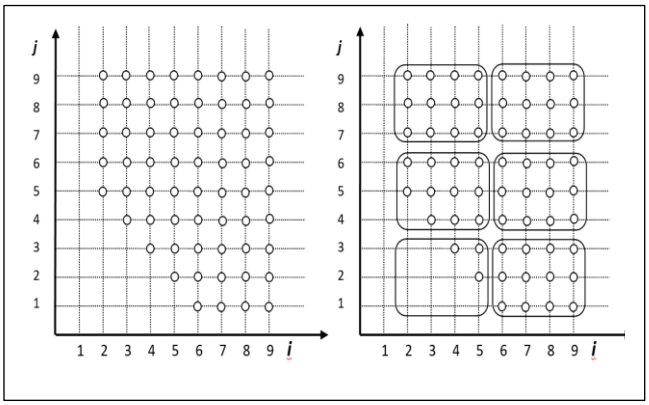

Fig. 1. Example of the implementing tiling method on two-dimensional loop

As it was shown on fig. 1 size of tiles can be chosen in arbitrary way. But how tile of sizes will influence on processing time it's not clear. To verify this fact the test application was ran on different sizes of tiles as it's depicted on fig. 2.

Horizontal axes are different tile sizes which were tested and vertical axe is time of the processing obtained for these sizes. As it is shown on the figure, processing time can vary significantly depending on sizes. It has to be noted that there is no some dependencies between sizes and time. Better set of tiles can be found only with some algorithm which can choose it with several iterations of search.

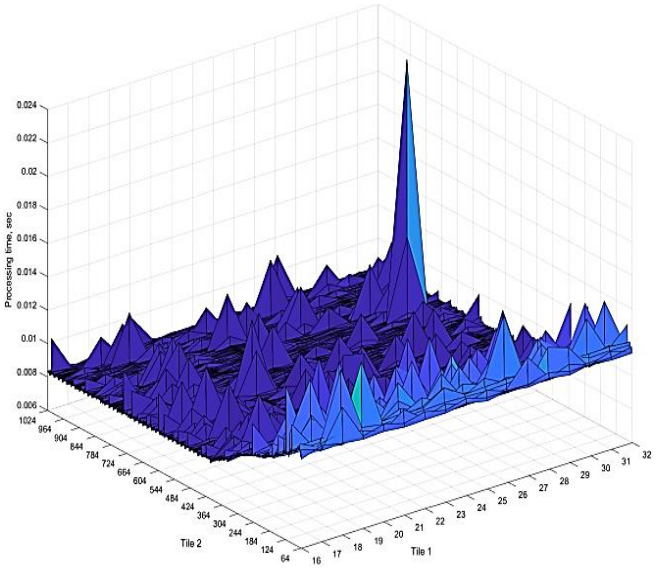

Fig. 2. Example of processing time depending on the different tile sizes

IV. Discrete Particle Swarm Optimization Method

Searching of optimal tile sizes for fastest processing can be treated as task of discrete optimum search. In this case non-smooth relation between parameters is present. For resolving such tasks some complicated methods are applied.

Swarm algorithms is a sort of genetic algorithms which are used widely for solving of traveling salesman problem, assignment tasks, planning etc. [8].

The main ideas of swarm behavior were proposed by Gerardo Beni and Wang Jing in [9]. A swarm is defined as a decentralized system, which consists of a set of simple

elements that interact with each other and with the environment to achieve a predetermined goal in accordance with certain rules. The concept of swarm intelligence is built on an additive, synergistic effect, which is manifested when agents are combined into a system. Elements of the swarm are called particles.

The model describing the decision of particles in a swarm is based on the position of each particle in the swarm and direction vector. The particle decides on movement based on three factors: its current speed, which causes the particle to continue moving and to explore new regions in the search area; knowledge of your own best state and the best state of the entire swarm or the nearest neighborhood of the particle.

.

TABLE I. DISCRETE PARTICLE SWARM OPTIMIZATION METHOD'S RESULTS

Particles count	**Iterations**	**Found minimum**	**Searching time, sec**	**Particles count**	**Iterations**	**Found minimum**	**Searching time, sec**
4	4	0.083637	0.001169	128	4	0.077609	0.015661
	8	0.083637	0.001306		8	0.077609	0.027726
	16	0.079986	0.001847		16	0.077217	0.051691
	32	0.079523	0.002842		32	0.077086	0.098834
	64	0.079430	0.004958		64	0.077086	0.195971
	128	**0.078543**	**0.008814**		128	0.077086	0.391339
	256	**0.078543**	0.016424		256	0.077086	0.773381
	512	**0.078543**	0.031559		512	0.077086	1.551134
	1024	**0.078543**	0.063166		1024	**0.076744**	**3.081073**
8	4	0.083637	0.001301	256	4	0.076955	0.048378
	8	0.083637	0.001817		8	**0.076744**	**0.087142**
	16	0.081941	0.002626		16	**0.076744**	0.162502
	32	0.080030	0.004480		32	**0.076744**	0.316789
	64	0.079785	0.007758		64	**0.076744**	0.626300
	128	**0.078543**	**0.014601**		128	**0.076744**	1.232822
	256	**0.078543**	0.028407		256	**0.076744**	2.462279
	512	**0.078543**	0.058010		512	**0.076744**	4.864909
	1024	**0.078543**	0.112901		1024	**0.076744**	9.697511
16	4	0.078953	0.001891	512	4	0.076955	0.164594
	8	**0.077609**	**0.002662**		8	0.076955	0.294610
	16	**0.077609**	0.004504		16	0.076955	0.557133
	32	**0.077609**	0.007811		32	**0.076744**	**1.084572**
	64	**0.077609**	0.014814		64	**0.076744**	2.130530
	128	**0.077609**	0.027550		128	**0.076744**	4.227274
	256	**0.077609**	0.053936		256	**0.076744**	8.421257
	512	**0.077609**	0.109502		512	**0.076744**	16.808286
	1024	**0.077609**	0.220439		1024	**0.076744**	33.597718
32	4	0.079196	0.003278	1024	4	0.076955	0.628595
	8	0.078119	0.005276		8	0.076955	1.121092
	16	0.078119	0.009532		16	**0.076744**	**2.129166**
	32	0.077910	0.017458		32	**0.076744**	4.148373
	64	0.077609	0.033737		64	**0.076744**	8.152203
	128	0.077609	0.067246		128	**0.076744**	16.216859
	256	**0.077086**	**0.131577**		256	**0.076744**	32.258004
	512	**0.077086**	0.255954		512	**0.076744**	64.408821
	1024	**0.077086**	0.510905		1024	**0.076744**	128.803013
64	4	0.079041	0.006620				
	8	0.079041	0.011354				
	16	0.077609	0.020791				
	32	0.076950	0.039425				
	64	**0.076744**	**0.078210**				
	128	**0.076744**	0.153229				
	256	**0.076744**	0.304029				
	512	**0.076744**	0.600386				
	1024	**0.076744**	1.192888				

Set of particles is denoted by $\boldsymbol{P} = \{P_i, i \in \overline{1..N}\}$, where N is a number of particles in swarm or population size. At t=0,1,2,… coordinates of the particle P_i are determined by vector $X_{i,t} = (x_{i,t,1}, x_{i,t,2}, \dots, x_{i,t,n})$, and its speed is a vector $V_{i,t} = (v_{i,t,1}, v_{i,t,2}, \dots, v_{i,t,n})$. r_1 and r_2 are random values from 0 to 1. Initial coordinates and velocities of the particles P_i are $X_{i,0} = X_i^0$, $V_{i,0} = V_i^0$, respectively.

$$v_{i,t+1} = wv_{i,t} + c_1 r_1[m_{i,t} - x_{i,t}] + c_2 r_2[g_t - x_{i,t}] \qquad (2)$$

Particles change their coordinates by changing their velocities on every iteration. In case of integer coordinates x parameters are integer by default. In addition, velocities v should be rounded to the integer values.

As far as nodes of iteration space are integer and thus size of tiles can be strictly integer Discrete Swarm Optimization Method was used. It means that particles and its speed was rounded to satisfy integer restrictions.

V. Results

Parameters of Discrete Particle Optimization Method consist of number of particles, two coefficients and initial state. During of updating of coordinates of every particle a swarm follows to optimal value. Depending on particle count and number of iteration a different optimal value may be found. To evaluate which number of iteration and count of particle will lead to global minimum several set of tests were performed

Applying Discrete Particle Optimization Method for searching of minimal execution time also is important in terms of required time to process. To find best set of number of iteration and particle count several tests were performed on data depicted on Fig. 3.

Obtained data are shown in Table 1. Best achieved values and fastest searching time for particular particle count are highlighted with bold font.

Obtained data are shown in Table 1. Best achieved values and fastest searching time for particular particle count are highlighted with bold font.

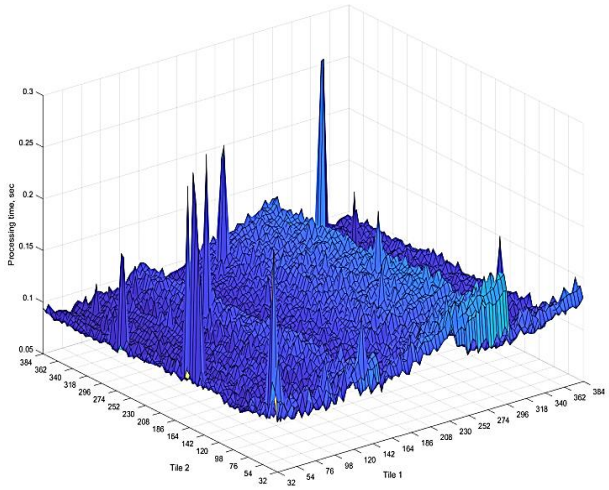

Fig. 3. Test application for Discrete Swarm Optimization Method verification

VI. Conclusion

The paper describes the usage of Discrete Particle Swarm Optimization Method for searching of tile sizes to obtain minimal time of program execution. This method provides a searching of better tile sizes but depending on iteration number and particle count it gives different minimums.

Global minimum was not found which equals to 0.075386 seconds for all sets of particle count and iteration number. In the same time, obtained times are very close to the global minimum. Also it can be noted that for all tests an increasing of iteration number did not give improvement of the result.

Discrete Particle Swarm Optimization Method can be used for searching of optimal tile size but searching of global minimum is not guaranteed.

References

[1] Kennedy, K., Allen, R.: Optimizing Compilers for Modern Architectures—A Dependence Based Approach, Morgan Kaufmann Publishers, San Francisco (2001)

[2] J. McConnell, Analysis of algorithms, 2nd ed. Jones & Bartlett Learning, 2007.

[3] Clauss, P., Loechner, V.: Parametric analysis of polyhedral iteration spaces. J. VLSI Sig. Proc. 19(2), 1–16 (1998)

[4] Darte, A., Vivien, F.: Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. Int. J. Parallel Prog. 25(6), 447–496 (1997)

[5] W. Bielecki and P. Skotnicki, "Insight into tiles generated by means of a correction technique", *The Journal of Supercomputing*, 2018.

[6] Boundhugula, U., Ramanujam, J., Sadayappan, P.: Pluto: a practical and fully automatic polyhedral parallelizer and locality optimizer, Technical Report OSU-CISRC-10/07-TR70, Louisiana State University, Columbus, OH (2007)

[7] Pouchet L.-N.: The polyhedral benchmark suite. [Online]. Available: http://web.cs.ucla.edu/*pouchet/software/polybench/ 22 Sept 2016

[8] A. Engelbrecht, Fundamentals of Computational Swarm Intelligence. Chichester, UK: John Wiley & Sons, 2005.

[9] G. Beni and J. Wang, "Swarm Intelligence in Cellular Robotic Systems", in *NATO Advanced Workshop on Robots and Biological Systems*, Tuscany, Italy, 1989.