IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 9 June 2022, accepted 5 July 2022, date of publication 14 July 2022, date of current version 20 July 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3190483

RESEARCH ARTICLE

A New Combination Method for Improving Parallelism in Two and Three Level Perfect Nested Loops

SHABNAM MAHJOUB[1], MEHDI GOLSORKHTABARAMIRI[1], (Senior Member, IEEE), SEYED SADEGH SALEHI AMIRI[2], MEHDI HOSSEINZADEH[3,4], AND AMIR MOSAVI[5,6,7]

[1]Department of Computer Engineering, Babol Branch, Islamic Azad University, Babol, Iran
[2]Department of Mathematics, Babol Branch, Islamic Azad University, Babol, Iran
[3]Mental Health Research Center, Psychosocial Health Research Institute, Iran University of Medical Sciences, Tehran, Iran
[4]Department of Computer Science, University of Human Development, Sulaymaniyah, Iraq
[5]John von Neumann Faculty of Informatics, Óbuda University, 1034 Budapest, Hungary
[6]Institute of the Information Society, University of Public Services, 1083 Budapest, Hungary
[7]Institute of Information Engineering, Automation and Mathematics, Slovak University of Technology in Bratislava, 811 07 Bratislava, Slovakia

Corresponding authors: Mehdi Golsorkhtabaramiri (golesorkh@ieee.org) and Amir Mosavi (amir.mosavi@kvk.uni-obuda.hu)

ABSTRACT The growth of software techniques for implementing applications must go hand in hand with the growth of computer system hardware in the design of multi-core and multi-processor systems; otherwise, we cannot expect to be able to use maximum hardware capacities. One of the most important and challenging techniques for running applications is to run them in parallel with a focus on loop parallelism to reduce execution time. On the other hand, in recent years, many algorithms have been working on volumetric data, i.e., three-dimensional spaces; therefore, parallelization must be possible for all types of two-dimensional and three-dimensional loops. Uniformization is an important part of loop parallelism, and also the present paper's focus. The proposed algorithm in the present paper performed uniformization with a combination of the frog leaping algorithm and the fuzzy system for two- and three-dimensional loops on a wide range of input dependence vectors and achieved a considerable variety of results in the desired time. The results of this study can be used to facilitate the development of parallel codes.

INDEX TERMS Frog leaping algorithm, artificial intelligence, parallelization, loops, uniformization, parallel compilers.

I. INTRODUCTION

In various fields of study such as engineering, medicine, etc. parallelization of sequential programs is one of the most challenging research areas which helps increase program efficiency through reducing runtime. The purpose of parallelizing programs is to develop a method that runs a wide range of sequential programs in parallel while staying in line with the development of hardware facilities [1]. One view classifies parallelism methods into two categories: automatic parallelism and programmer-assisted parallelism [2]. In automatic parallelization, the compiler parallelizes the sequential program without the intervention of the programmer. But in the programmer-assisted type, the programmer provides tips or instructions to the compiler, based on which the compiler executes the parallelization steps. OpenMP [3] and Cilk [4] are two well-known approaches that use instructions for extracting parallel codes. Another classification, the one in [5], classifies parallelism into three categories. The first is parallel programming which greatly challenges programmer skills. The second category is semi-automatic parallelism, which can be equated with programmer-assisted parallelization in the previous classification. Finally, the third category is to automatically translate sequential programs

The associate editor coordinating the review of this manuscript and approving it for publication was Wai-Keung Fung.

into parallel versions of themselves without any modification, which is equivalent to the first category of the previous classification. However, the fact is that although many years have passed since the invention and use of multi-core systems, programmers still write codes sequentially, meaning that they are actually reluctant to use the first and second methods mentioned above [6]. This is firstly because parallel programming requires a high level of specialized skills, and secondly, there are problems such as deadlocks and so on that have to be dealt with. Perhaps this is why automated application parallelization has been a long-standing and prominent research topic back from 1990 [5], [7]. In recent years, the issue of automatic parallelization of programs with irregular structures has widely been considered [8]–[10]. Most programs that have irregular structures use different levels of data. This feature reduces the parallelization of sequential codes to the same degree that it increases efficiency because reliably determining independent parts for compilers and programmers is not easy. But in practice, most of these approaches lead to semi-automatic parallelization. These techniques can generally be classified into two categories: non-speculative parallelism and speculative parallelism [6]. Non-speculative parallelizing compilers divide the sequential code into independent tasks but there is no guarantee that two tasks are independent at compiling time. The use of Polyhedral compilers [11], [12] is to parallelize loops that have regular accesses into arrays and similar structures but most programs are irregular, and for such programs, compilers have limited visibility when invoking codes, a fact that impedes non-speculative parallelization. On the other hand, speculative parallelism [13]–[15] is a technique that increases memory usage and keeps more processors busy at runtime but suffers from costly speculation and makes parallelism unprofitable in practice for applications with frequent conflicts. For example, [6] proposed a compiler called t4. This compiler parallelizes the whole program and does not only focus on a specific structure of the program; however, it cannot be called a fully-automatic parallel compiler because it suffers from successive failures stemming from order violations, and more precisely, because these failures allow the programmer to annotate code areas to be divided into tasks, which actually makes it semi-automatic parallelization. Since researchers are mainly focused on generating tools that perform parallelization in a fully automated way, they try to perform parallelism by transforming the source-to-source code of the loop. But most of them have one thing in common: they often have to stop parallelization without any explanation to the programmer, like the case of Pluto [11]. For this reason, researchers have subsequently developed a compiler [16] that reduces these compulsions and notifies the programmer when something goes wrong. So, it is still not clear to these compilers how close they are to the goal of auto-parallelizing sequential programs, and they actually have to create a tradeoff between ‘‘fully automatic parallelism’’ and ‘‘loop type’’.

However, in order to achieve fully-automatic parallelism, this paper slightly narrows the scope of research so that its result is closer to reality. Thus, based on the aforementioned issues, although it is not clear how close we are to the goal of fully auto-parallelism of sequential applications, it is clear that we are still at the beginning, and research efforts must continue in this regard considering the great impact of parallelization on the reduction of application runtime. Because of all the limitations mentioned in the above paragraphs, researchers often focus on loop parallelism [17]–[21]. In fact, rather than parallelizing the entire program, which is unable to fully auto-parallelism, the idea is to focus on parallelizing parts of the program that have longer runtime. In applications, most of the runtime is spent on loops [22], [23] and thus the automatic parallelization of loops greatly reduces applications' runtime and increases efficiency. Loop parallelization is performed in four steps [24], namely data dependence analysis, loop tiling, parallel loop generation, and loop scheduling. In the data dependence analysis step, the type of loop dependency is determined; this may be uniform dependent and non-uniform dependent. If the non-uniform structure of the iteration space can be converted to a uniform one using a set of basic dependence vectors, parallelism can be performed with one of the simplest methods available, i.e. the Wavefront method [25]–[28]. Despite the many investigations on loop optimization techniques, Wavefront parallelism is still one of the simplest methods for loop parallelism, and in recent years, it has led to the presentation of code generators that are actually based on uniform dependencies [29]. In this method, none of the iterations of the same Wavefronts are not dependent on each other and depend only on the previous Wavefronts' iterations. In [30], one of the latest research papers on the presentation of parallel compilers is presented. When dependencies are non-uniform, time tiling constraints are also nonlinear and usually increase in size and computational complexity after becoming linear. For this reason, to directly detect dependency constraints, DAPT proposed a way to approximate the original non-uniform dependencies to uniform ones and proved that it is simpler than Pluto. In fact, it has normalized non-uniform dependencies to uniform ones. In addition, Pluto only generates codes that merely provide two-dimensional tiles, but DAPT generates codes that also provide three-dimensional ones. The important point, however, is to use the uniformization method in DAPT, reminding us that uniformization is very important. In addition, in recent years, the lack of research in the field of uniformization has led to the presentation of solutions based on a lack of optimal techniques for uniformization. For example, in [31], which is one of the most recent research studies on the tiling and scheduling of loops, it is clearly stated that the existing methods for parallelization of loops use uniformization, but because of the failure in determining the base dependence vectors, they require a lot of execution time. For this reason, the study proposed an approach that performs tiling directly on non-uniform loops, which, of course, has its limitations, including the fact that it only applies to two-dimensional loops while its complexity is very high for higher-level loops. This paper focuses on the uniformization of two- and

```
For I=1, N Do
        For J=l(I), u(I) Do
S1:         ...
Ss:         A(f1(I, J), f2(I, J))=...
St:                   ...=A(f3(I, J), f4(I, J) )
Sz:         ....
        End For
End For
```

FIGURE 1. Loop model in TLP.

three-dimensional perfect nested loops, which is one of the important steps in parallelizing loops. The following are some important terms used in the paper to better understand the problem and the proposed method:

- **Perfect Nested Loop**: consider a two-dimensional loop in Fig. 1 adapted from [32], where $l(I)$ and $u(I)$ are linear functions of the variable I and N is the upper bound of the outermost loop and constant. in addition, the $S_1, S_2, \ldots, S_z$, are computation statements so that among them, S_s and S_t have accessed the same array A and $f_1(I, J)$, $f_2(I, J)$, f_3 (I,J), and $f_4(I, J)$ are subscript expressions [32].

Loop model in Fig. 1 called a perfect because all the statements S_1 *to* S_z are in the innermost loop. There is a similar definition for three-level loops.

- **Uniform Dependence Loop (UDL)**: A vector $\vec{v} = (i, j)$ is an *iteration vector* if $\Gamma\vec{v}\ \in$, where $\Gamma = \{(i,j)\,|1 \le i \le N, l(i) \le j \le u(i), i,j \in Z\}$ is the iteration space and Z is a set of integers. If two vectors (i_1, j_1) and (i_2, j_2) can be found such that $f_1(i_1, j_1) = f_3(i_2, j_2)$ and $f_2(i_1, j_1) = f_4(i_2, j_2)$, the nested loop has cross-iteration dependences. in fact, if we can find four integers (i_1, j_1, i_2, j_2) that satisfy equations (1) and (2), we can say that the loop has cross dependencies and vice versa [32].

$$\begin{cases} f_1\,(i_1, j_1) = f_3\,(i_2, j_2) \\ f_2\,(i_1, j_1) = f_4\,(i_2, j_2) \end{cases} \tag{1}$$

$$\begin{cases} 1 \le i_1 \le N \\ l\,(i_1) \le j_1 \le u(i_1) \\ 1 \le i_2 \le N \\ l\,(i_2) \le j_2 \le u(i_2) \end{cases} \tag{2}$$

Assuming that x and y are the two free variables, The general solution of (1) as shown in Equation (3).

$$(i_1, j_1, i_2, j_2) = (g_1\,(x, y)\,, g_2\,(x, y)\,, g_3\,(x, y)\,, g_4\,(x, y)) \tag{3}$$

where coefficients of g_i, x and y are integers. The Dependence Vector Function of L is defined as $(h_1\,(x, y)\,, h_2\,(x, y))$, where $h_1\,(x, y) = g_3\,(x, y) - g_1\,(x, y)$ and $h_2\,(x, y) = g_4\,(x, y) - g_2\,(x, y)$ which is called DVF(L). If integer constants are obtained for $h_1\,(x, y)$ and $h_2\,(x, y)$, then the L is Uniform Dependence Loop or UDL. There is a similar definition for three-level loops. To better understand this definition, consider an example of a loop in Fig. 2 which is adapted from [32] and described bellow.

```
For i=1:1000
    For j=1:1000
            A(i+j, 3*i+j+3)=...
            ...=A(i+j+1, i+2*j+4)
    EndFor
EndFor
```

FIGURE 2. An example of a two-level loop.

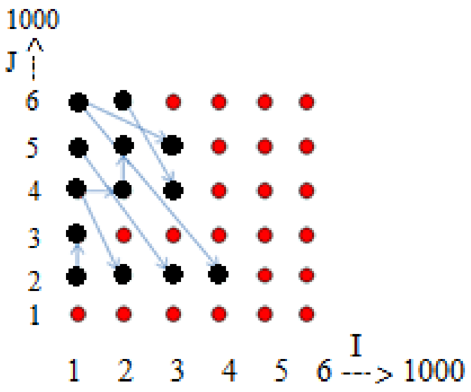

FIGURE 3. The dependence graph of the example in Fig. 2.

The corresponding system of Diophantine Equations is shown in Equation (4).

$$\begin{cases} i_1 + j_1 = i_2 + j_2 + 1 \\ 3 * i_1 + j_1 + 3 = i_2 + 2 * j_2 + 4 \end{cases} \tag{4}$$

The general solution (g_1, g_2, g_3, g_4) is equal to $(x, y, -x + y - 1, 2x)$, and (h_1, h_2) is $(-2x + y - 1, 2x - y)$ where $(x, y) \in Z$. Obviously, this example is not a UDL, as shown in Fig. 3, because some are dependent on others. Some details are not explained [32].

- **Dependence Cone Size:** is abbreviated as (DCS). It is to be considered as the vector of dependency set $D = \{\vec{d}_1, \vec{d}_2, \ldots, \vec{d}_m\}$. Furthermore, the dependence cone C(D) can be explained as follows defined by the Equation (5).

$$C(D) = \{\bar{x} \in R^n : \bar{x} = \lambda_1\vec{d}_1 + \lambda_2\vec{d}_2 + \cdots + \lambda_m\vec{d}_m, \lambda_1, \lambda_2, \ldots, \lambda_m \ge 0\} \tag{5}$$

The DCS is defined as in Equation (6) which is the area of the intersection of C(D) with $x_1^2 + x_2^2 + \cdots + x_n^2 = 1$.

$$\int_{\bar{x}\in C(D), x_1^2+x_2^2+\cdots+x_n^2=1} d\bar{x} \tag{6}$$

In fact, the C(D) as an smaller value for cone maintains all the loop's dependence vectors. The second step in the uniformization process is to create a schedule that can run that nested loop in parallel. The solution space for such an optimal linear schedule is in the form of $\left\{\pi : \pi.\vec{d}_i > 0, \vec{d}_i \in D\right\}$ in which D is the set of dependence vectors which is another

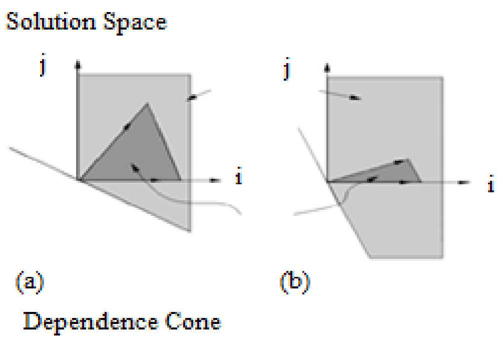

FIGURE 4. **The dependence cone and solution space of the linear schedule vector. (a) Larger dependence cone which is smaller solution space (b) Smaller dependence cone, which is larger solution space.**

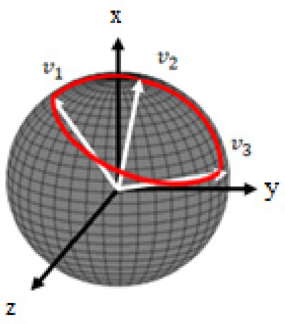

FIGURE 5. **The dependence cone for three vectors corresponding to a three-level loop.**

dependent cone whose size depends on the dependent cone constrained by the $\vec{d}_i$. As shown in Fig. 4 adapted from [33], a smaller DCS leads to a larger solution space. A larger solution space also leads to more choices. Therefore, in this larger solution space, it is more likely to choose the optimal linear schedule than to choose from a smaller solution space [33]. For this reason, in the method proposed in this paper, we are looking for results with the least DCS.

The DCS in a two-dimensional space exactly corresponds to the angle between those vectors. But in a three-level iteration space, it is equal to the volume of the area between the spheres with a unit radius ($x^2 + y^2 + z^2 = 1$) and those vectors. For example, Fig. 5 which was adapted from [24] shows the dependence cone for three vectors corresponding to a three-level loop. Section III describes the details of the DCS calculation for both types of loops.

- **Basic Dependence Vector Sets (BDVS)**: The basis of all uniformization methods is based on vector decomposition so that the set of BDVSs which can represent any vector in the loop iteration space as a nonnegative linear combination of these vectors can be found. To clarify the term, given a two-level nested loop L, a set of vectors $B = \{\vec{b}_1, \vec{b}_2, \ldots, \vec{b}_b\}$ is defined as a BDVS of L, if for any $\vec{d} \in C$, there is $c_1, c_2, \ldots, c_b \in Z^+$, such that $\vec{d} = c_1 * \vec{b}_1 + c_2 * \vec{b}_2 + \cdots + c_b * \vec{b}_b$, where C is the dependence cone of L.

There are many options for the BDVS, but there may be disadvantages to selecting a number of them in the parallelization of programs. For example, vectors may be created with artificial dependence between two points while not at all being related before uniformization. In this paper, the aim is to keep the number of these vectors to a minimum because a smaller number of dependence vectors leads to less constraint in choosing the optimal linear scheduler, as well as in the implementation of hardware for the interconnection of the fundamental dependence vectors. In addition, a smaller number of these vectors leads to a smaller number of artificial dependence vectors after uniformization and reduces uniformization damages.

A. SHUFFLED FROG-LEAPING ALGORITHM (SFLA)

The shuffled frog-leaping algorithm (SFLA) is a population-based meta-heuristic for discrete optimization in which deterministic and random approaches are combined and were first developed by Eusuff *et al.* [34]. In fact, SFLA can take advantage the benefits of two genetic-based memetic algorithm and Particle Swarm Optimization (PSO) at the same time. In the conducted studies [35], SFLA has a higher calculation speed compared to other evolutionary-based algorithms. Depending on the type of application, both two- and three-dimensional loops can be used. Processing of 3D data are very important areas that have been considered in recent years [36]. Working with this type of data requires a lot of memory and high computing power [37]. So far, little work has been done on the uniformization of two- and three-dimensional iteration spaces in a separate manner [24], [38]. But the problem is that for a 2D space, the size of the dependence cone is proportional to the size of the angle between the BDVS, while in a 3D space, it is proportional to the volume enclosed between the BDVS, and the same mechanism cannot therefore be used for uniformization of both types of loops. This paper proposes a single algorithm for the uniformization of both 2- and 3-D loops. It addresses the issue of uniformization for all types of loops since the issue has not been sufficiently addressed. Transforming a non-uniform structure to a uniform one is an NP-Hard problem [24], so evolutionary methods can be used to achieve the optimal result at the desired time. In this paper, among the available evolutionary methods, the Frog Leaping Algorithm (FLA) has been selected. Compared to other similar methods, this algorithm has a higher ability to find the optimal solution in the shortest time due to having a combination of local and global searches. On the other hand, three important factors affect the determination of the fitness function, which are the length of the BDVS, the cone size limited to the BDVS, and the number of vectors that can be decomposed by the BDVS. In the proposed algorithm, instead of considering constant coefficients, these factors are used as inputs of the fuzzy inference system to increase the accuracy and variety of the obtained results. The main contributions of this article are as follows:

- Combination of fuzzy and evolutionary methods for uniformization of loops in the shortest possible time with high result variability, no main vectors, and minimum DCS

- A single algorithm for uniformization of both two- and three-dimensional loops

The rest of this paper is organized as follows: in Section 2, the related studies are described. In Section 3, the proposed algorithm is described in detail, while in Section 4, the results of the implementation of the proposed method on different data sets are expressed and evaluated. Also, in this section, the proposed algorithm is compared with other methods and the reasons for its superiority are explained. Finally, in Section 5, conclusions and future directions will be presented.

II. RELATED WORKS

At all levels of automatic parallelization, including compiler, runtime, and source [39]–[42], parallelism focuses on the entire application, and since it often fails to search for anonymous internal classes, it needs to be combined with other methods to improve in practice and has consequently shifted toward semi-automatic parallelization. For this reason, parallelization of parts of the program that have a longer execution time than others, i.e., loops, is more appropriate in practice. On the other hand, for automatic parallelization of applications, the goal is parallelization by the compiler without programmer intervention. Therefore, parallelism at the compiler level has been more largely considered, the existing methods of which can be classified into two categories [5]: loop parallelization and non-loop parallelization. Among the methods based on non-loop parallelization, in [41], the data is analyzed on reads and writes at the local level, and in [43], the program complies into a speculative C program. The interaction between threads is checked at runtime but does not support I/O instructions. Optimizations that have been made in recent years to automatically modify the parallelization of Java applications [8], [10], [44] either cannot adapt with common multi-core processors, or have poor speed, or need high programming skills. Although several mechanisms have been studied for parallelize to irregular programs [45], [46], no recommendations have been made for their use, as they depend heavily on the details of the program. Among the various parallelization methods at the instruction-level of the loops are the open-loop technique [17], [47], the iterating and checking technique [20], [19], parallel inspection [48], [49], transforming the loop [50]–[52] and so on, most of which create a lot of memory overhead and are not suitable for complex codes. In some methods, parallelization is done on the binary code [53], [54]. But it is very difficult to understand at higher levels. Another method is called Wavefront. In this method, iteratio ns that have the same wavenumber can be executed simultaneously. In [55], a method is presented in which a table is created specifying the number of data iterations in which reading and writing are performed. The results of the experiments obtained in the optimizations performed on the Wavefront method show that the parallelization process is faster and less memory is consumed. This method is the most common, while it is also the simplest for running parallel loops. It highlights the importance of uniformization because the uniform structure of the loops facilitates the detection and execution of parallel codes. The first investigation on uniformization was in [32], after which only few studies were done on it [33], [56], whose main problem was the large DSC and the presence of at least one main vector. The previous study of the authors of the present paper proposed the first approach based on a genetic algorithm to solve this problem [24] on three-level perfect nested loops. However, it still suffered from the problem of high runtime and lack of practicality. In [38], which uses the approach based on FLA, three effective factors in the problem of fixed coefficients were considered, greatly limiting the final results. In addition, experiments and evaluations have been performed on limited data sets that are only parallel to a maximum of two known vectors. In this paper, a single algorithm is presented for uniformization of both two- and three-dimensional loops based on a combination of evolutionary and fuzzy methods to overcome the limitations of previous methods.

III. THE PROPOSED ALGORITHM

This section details the implementation of the FLA and how it is combined with fuzzy with the aim of obtaining optimal BDVS.

A. RESTRICTIONS OF CREATING THE INITIAL POPULATION (FROGS)

Since the algorithm proposed in this paper is used to uniformize both two- and three-dimensional loops, the loop studied in this paper is a perfect nested loop with the structure given in [22], the only difference being that it can be both two- and three-leveled. In the two-level mode, U_1 and U_2 are the upper bounds of the first and second loops, respectively, and for the three-level mode, U_3 is the upper bound of the third nested loop. In the proposed method, each frog represents a set of basic dependence vectors that use two-dimensional arrays for two-level loops and three-dimensional arrays for three-dimensional ones, so that for two dimensions, each member has (x, y), and for three dimensions, each of its members (x, y, z) represents a vector in that space. Suppose (i_1, j_1, k_1) and (i_2, j_2, k_2) are two iterations of a three-level nested loop. The dependence vector obtained from these iterations must be $i_2 > i_1$, or if $i_1 = i_2$, $j_2 > j_1$, or if $i_1 = i_2$ and $j_1 = j_2$, $k_2 > k_1$. For two-level loops, the dependence vector is $i_2 > i_1$, or if $i_1 = i_2$, it is $j_2 > j_1$. If vectors do not have such conditions, they are called invalid vectors. In the proposed method, such vectors will not exist in the population, and during the implementation of the program, if generated, they will be removed and replaced by other vectors that are in the problem space. The number of BDVS also varies between two and three for two-level loops, and between three and five for three-level ones. These numbers are the optimal lengths for the BDVS for two-level and three-level loops. In addition, the vectors of each set are linear and independent. Note that in a two-dimensional space, for example, a maximum of two separate vectors can be defined to be independent, so not all sets with three members can be independent. But here, the independence of vectors means

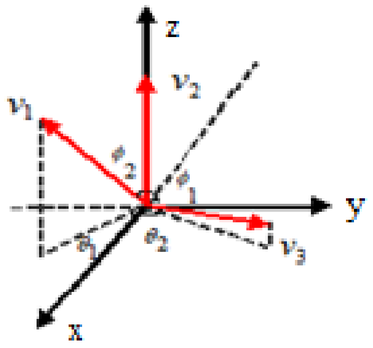

FIGURE 6. Values of vectors in spherical coordinates after rotation.

that not only can vectors not be multiples of each other, but for more than two numbers in a two-dimensional space, no vectors of the other two combinations will be obtained.

B. HOW TO CALCULATE DCS FOR EACH MEMBER (DCS(i))

The size of the cone limited to the BDVS in a two-dimensional space exactly corresponds to the angle between those vectors. Therefore, for two vectors, the angle between them is calculated by internal multiplication. For a larger number, first the size of the angle of each vector with the y-axis, i.e., vector (0, 1) is calculated, and the largest and smallest of them are determined, and finally the difference of these two values determines the DCS. But in a three-level iteration space, it is equal to the volume of the area between the spheres with a unit radius ($x^2 + y^2 + z^2 = 1$) and those vectors. Assuming that BDVS consists of the terms v_1, v_2, and v_3, it is obvious that for such a volume to exist, the three vectors v_1, v_2, and v_3 must be linear and independent. Otherwise, the three vectors would be on the same plane, and DCS = 0. To calculate this volume, the coordinate system can be changed so that one of the vectors matches one of the main axes. In the proposed algorithm, as shown in Fig. 6, the vectors rotate in a three-dimensional space so that one of them coincides with the z-axis.

$$V = \int_{\theta_1}^{\theta_2} \int_{0}^{\varphi_1} \int_{0}^{1} \rho^2 sin\varphi \mathrm{d}\rho \mathrm{d}\varphi \mathrm{d}\theta \tag{7}$$

$$\varphi = f(\theta) = \frac{\Delta\varphi}{\Delta\theta}(\theta - \theta_1) + \varphi_1 \tag{8}$$

$$dcs(i) = \int_{\theta_1}^{\theta_2} \int_{0}^{f(\theta)} \int_{0}^{1} \rho^2 sin\varphi \mathrm{d}\rho \mathrm{d}\varphi \mathrm{d}\theta \tag{9}$$

Assume that α_i is the angle of the vector's image v_i on the plane xy with the positive direction of the x-axis, and β_i is the angle of the vector with the positive direction of the z-axis. Also, assume that the smallest α (α_j) in terms of absolute value and its corresponding β (β_j) belongs to v_i. If R_Z and R_Y are rotation matrices around the z-axis and y-axis with α_j and β_j sizes respectively, these rotation matrices can rotate the vector v_i to match the z-axis. R_Z and R_Y are obtained by Equation (10).

$$R_Z = \begin{bmatrix} cos(\alpha_j) & -sin(\alpha_j) & 0 \\ sin(\alpha_j) & cos(\alpha_j) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathrm{R}_y = \begin{bmatrix} cos(\beta_j) & 0 & sin(\beta_j) \\ 0 & 1 & 0 \\ -sin(\beta_j) & 0 & cos(\beta_j) \end{bmatrix} \tag{10}$$

In fact, the volume of the area enclosed between these three vectors and the spheres with a radius equal to 1 remains constant by changing the coordinate system and the rotation of the vectors so that one of the vectors coincides on the z-axis. After the rotation of the three original vectors, new vectors are obtained which, instead of cartesian coordinates, their spherical coordinates can be used to calculate DCS using Equations (7) to (9) [38]. The vector that matched the z-axis after rotation (v_2) has $\varphi = 0$. Also, because the volume of the area enclosed in the sphere is calculated with a radius equal to one, ρ can be considered equal to one for each of the vectors. Therefore, only calculating φ and θ is sufficient for the other two vectors.

To calculate the DCS for basic vectors with more than three members, the members are divided into sections so that the generalization of the three-vector method can be used for their calculation. To do this, it is enough to consider a vector as the head vector, and since in the absence of a vector corresponding to the z-axis, it is necessary to perform rotation, from the very beginning, this is done for all vectors, and the vector aligned on the z-axis is considered as the head vector. After this step, for BDVSs with a length of four, three vectors, and for BDVSs with a length of five, four vectors remain, the position of which should be specified relative to each other and to the head vector. In so doing, the angle of the image of each vector on the xy-plane with the negative direction of the x-axis is used as the weight (wθ). Since the same value may be obtained for two vectors, this angle is considered to vary from 0 to 360. The vectors are arranged in ascending order based on wθ, and the vector matched on the z-axis has a weight of 1, while the other vectors take on a weight of 2, 3, and so on, respectively. By calculating the size of the dependence cone for two or three times, the total DCS is calculated. Since during the implementation of the algorithm the comparison between frogs is done by calculating the fitness function, and it is not possible to compare the angle values (for two-level loops) with the volume (for three-level loops) at the beginning of the algorithm, depending on the type of loop, one of the two aforementioned methods is used to calculate the DCS.

C. COMBINING FLA AND FUZZY TO DETERMINE THE OPTIMAL BDVSs

As described earlier, the FLA is used to determine the optimal BDVSs due to has more ability in finding the optimal result in the shortest time compared to other similar methods. The number of memeplexes is denoted by *m,* and the number of frogs in each memeplex is denoted by *n,* all received from the input, and we thus have a total population of $F = m \times n$. When generating the initial population of frogs for each member, the fitness function amount is calculated by fuzzy parameters, and the proposed algorithm finds the optimal result in both global and local searches. In the algorithm presented in this paper, the classical fuzzy logic method is used, and it is obvious that having complete basic rules that cover all possible cases is a necessity. In fact, the proposed system uses a complete set of rules for all cases that may occur (this number is limited for the proposed algorithm). Therefore, there is no need to use Fuzzy Rule Interpolation (FRI) methods. Three important factors affect the value of each member i: the length of the dependence vector (L(i)), the size of the cone limited to the dependence vectors (DCS(i)), and the number of vectors that can be decomposed by that member (M(i)). To calculate M(i), one of the Diophantine Equations of Equation (11) or (12) must be computed for each member i for a two-dimensional or three-dimensional loop, respectively. Details for solving such Equations are given in Equation (12). For three-dimensional loops, (X, Y, Z) is one of several input dependence vectors used by members (x_1, y_1, z_1) to (x_n, y_n, z_n) in the ith member which is decomposed through non-negative α. The same parameters are used for the two-dimensional loop in Equation (11).

$$\alpha_1 (x_1, y_1) + \alpha_2 (x_2, y_2) + \cdots + \alpha_n (x_n, y_n) = (X, Y) \quad (11)$$

$$\alpha_1(x_1, y_1, z_1) + \alpha_2(x_2, y_2, z_2) + \alpha_3(x_3, y_3, z_3) + \cdots + \alpha_n(x_n, y_n, z_n) = (X, Y, Z) \quad (12)$$

These factors are considered as inputs to the fuzzy inference system. Given that the algorithm is used to uniformize both two- and three-level loops, for this factor (M(i)), for two-level loops, since the optimal length is between 2 and 3, the three linguistic variables with triangular membership functions of "excellent", "good" and "bad" are used with parameters [0 2 3], [2 3 4] and [4 6 6], while for three-level loops, since the optimal length is between 3 and 5, three linguistic variables with triangular membership functions of "excellent", "good" and "bad" are used with the parameters [3 3 4], [3 4 5] and [4 5 6]. For DCS, three linguistic variables with triangular membership functions "excellent", "good" and "bad" are used depending on the type of the input loop. Depending on whether the loop is a two- or three-level one, the angle between the vectors or the volume enclosed between them is used respectively. For the M, the three linguistic variables "low", "medium" and "high" are considered, depending on how many input vectors they can decompose. Finally, for the fuzzy output based on the inputs, 9 linguistic variables from "very weak" to "very strong" are used. A summary of the if-then rules is given in Table 1. The best case is when L(i) is "excellent", M(i) is "high" and DCS(i) is "excellent", which leads to a "very strong" output, while the worst case is when L(i) is "bad", M(i) is "low" and DCS(i) is "bad," resulting in a "very poor" output. In general, out of the 27 cases available, if M(i) is "low" or "medium", even with desirable DCS(i) and L(i), the output of one of the three modes will still be poor because the goal is to ultimately decompose the input vectors, and this set i is not close to the goal. But in the states where M(i) is "high", depending on the values of DCS(i) and L(i), it can have one of the "medium" to "high" states. These general rules are independent of the number of loop dimensions, that is, these apply to both two- and three-level loops.

TABLE 1. Summary of if-then rules for calculating the fitness function for each frog.

L	DCS	M	Fitness
Bad, good or excellent	bad, good or excellent	Medium or low	Very weak, little weak, weak
Bad or good	bad	high	Less than average
excellent	bad	high	Average
bad	good	high	More than average
bad	excellent	high	Little strong
excellent	good	high	
good	excellent	high	Strong
good	good	high	
excellent	excellent	high	Very strong

Algorithm 1 shows the details of the proposed method, in which in a global search repeated at *global_iteration*, P frogs are randomly selected and sorted in descending order and categorized in m memeplexes. For each memeplex, with a *local_iteration* number, each time q frogs are randomly selected; the best (P_B) and worst (P_W) frogs are selected. The worst frog leaps as much as S to get closer to the best.

To calculate S for three-level loops, first the average angles of the vectors in (P_B) with the positive direction of the z-axis (S_α) and the mean angles of their image on the xy-plane with the positive direction of the x-axis (S_β) are calculated. Then, they are rotated using the matrix R_Z as large as S_β around the z-axis and using the matrix R_Y as large as S_α around the y-axis according to Equation (13).

$$R_Z = \begin{bmatrix} cos(S_\beta) & -sin(S_\beta) & 0 \\ sin(S_\beta) & cos(S_\beta) & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \mathrm{R}_y = \begin{bmatrix} cos(S_\alpha) & 0 & sin(S_\alpha) \\ 0 & 1 & 0 \\ -sin(S_\alpha) & 0 & cos(S_\alpha) \end{bmatrix} \quad (13)$$

But for two-level loops, S is the amount of the frog's leap, which is equal to the value of the angle with respect to the y-axis, and the worst frog must rotate as much as it; it is considered equal to $S = mean(P_B, P_W)$. The allowable amount of rotation for each numerical member is between -180 and 180. For both types of loops, if the new position is based on what was described in section 3.1, and also if the vectors are not invalid and the new member fitness is more than the previous value, the new member replaces the previous one. Otherwise, a new frog will be randomly created and will replace the worst one. Subsequently, the fitness amount of it in the current population is updated. Finally, the member with the highest fitness will be obtained as the optimal result.

Algorithm 1: Pseudo Code of the Proposed Algorith

```
INPUT:
        $L_1, U_1, L_2, U_2, L_3$ and $U_3$: lower and upper bound in three-level loops
        $L_1, U_1, L_2$ and $U_2$: lower and upper bound in two-level loops
        Dependence vectors for loops
        n: number of frogs per memeplex
        m: Number of memeplexes
        local_iter: number of local searches
        global_iter: number of global searches
1. Generate F initial population as F=m*n
2. If loop=2D, calculate FUZZY_Fitness_2D(i) for each frog i in population
          or else,calculate FUZZY_Fitness_3D(i) for each frog i in population
3. Iterate global search to number of global_iter
      3.1 Generate population P randomly
      3.2 If loop=2D, calculate FUZZY_Fitness_2D for P
      or else, calculate FUZZY_Fitness_3D for P
      3.3 Sort P in descending order
      3.4 Partition P into m memeplexes
          3.4.1 Iterate local search to number of
             local_iter
               3.4.1.1 Generate population q
          randomly
               3.4.1.2 Determine the best and worst
          frog ($P_B$ and $P_W$)
               3.4.1.3 Update the worst frog with S
          (Based on formulas for 2D and 3D loops)
OUTPUT:
        Optimal BDVS
```

IV. EVALUATION AND EXPERIMENTAL RESULTS

In this paper, it has been assumed that loop dependencies are predetermined in the form of dependence vectors, and the main focus is on uniformization. The proposed algorithm was tested on a wide range of input datasets with a variety of positive and negative second and third components, both on two- and three-dimensional loops.

The results of the implementation of this algorithm using C-based MATLAB software promise good performance in compilers that are generally implemented in C language. Table 2 shows the details of these tests. In each test, k number of vectors parallel to the vectors written in the input column were given to the algorithm as input dependencies. In this table, *memeplexes* indicates the number of memeplexes, *frogs* the number of frogs in each memeplex, g the number of global and l the number of local searches. In the output column, DCS indicates the size of the dependent cone limited to the optimal BDVS and the fitness value. Each test was performed 10 times with different parameters to finally achieve optimal BDVS. The average execution time of these tests is also given in the time column. The first to seventh tests were performed on the dependence vectors of two-dimensional loops, and the eighth to tenth tests were performed on the dependence vectors of loops that are three-dimensional. To test the correctness of the proposed algorithm, it was first run on vectors that had positive second and third components. In this circumstance, the expectation is that in the worst case, the main vectors, i.e., {(0, 1), (1, 0)} for two-dimensional and {(1, 0, 0), (0, 1, 0), (0, 0, 1)} for three-dimensional loops would be obtained. But the results obtained from the implementation of the proposed algorithm show that a BDVS is obtained whose DCS is much less than the DCS of the main vectors (0.52 for 3D and 90 for 2D). The first, second and seventh experiments confirm this.

In addition, in many results obtained due to the use of the fuzzy system in calculating the amount of fitness (instead of using fixed coefficients), a good variety was obtained. In the first experiment, two sets of BDVS were obtained as the optimal response, from which the set {(1, 1), (3, 2)} had a better result than the latter due to its smaller DCS. Also, in the seventh experiment, three sets of results were obtained as optimal, all three of which could decompose all input vectors well, but set {(1, 2, 1), (1, 2, 3), (1, 0, 0)}, due to its having a smaller DCS, can be a more appropriate choice. Other tests have been performed on dependence vectors parallel to vectors that had a second or third negative component (y or z). For the two-dimensional loop, the third experiment was performed on 400 vectors, among which only one of the second components was negative and parallel to the sets {(1, −2), (2, 1)}, and finally, the same set with DCS = 90, and an average run time of 143.12 seconds for 10 runs. Also, in the fourth, fifth, and sixth tests, all the second components of all vectors were negative. For the fifth test, two sets of vectors were obtained as the optimal BDVS, and since both decomposed all 120 input vectors and had equal DCS, they can be selected as the optimal BDVS. For the sixth experiment, three sets were obtained as results that had equal DCS, and all decomposed all 400 vectors, but since set {(1, −2), (1, 1), (1, 0)} had a main vector, it is recommended to use two other sets to uniformize the loop. Fig. 7 shows the mean fitness of the first to sixth tests corresponding to the implementation of the proposed algorithm on two-dimensional loops. The increasing trend of fitness with increasing iterations shows that the frogs in each iteration are closer to the optimal location than in the previous iteration, which means that at the end of the iterations, we can expect results that are very close to optimal.

Tests eight to ten were performed on dependence vectors parallel to vectors that had at least one second or third negative component. In the eighth experiment, two sets of vectors were obtained as the result. Among these two, the set {(1, 1, 1), (1, −1, 0), (0, 1, −1)} had low DCS, and there was no main vector in it. Therefore, it was selected as the optimal result.

Since most of the previous methods were time-consuming, low-input algorithms were tested to keep runtime low. The tenth experiment was performed to test the speed of the proposed algorithm. This algorithm was executed on 500 parallel vectors with set {(1, 2, 1), (2, −1, 1), (2, 1, −1)} for ten times and three sets of results with an average execution time of 172.35 seconds were obtained. Among the results obtained, the set {(1, 2, 1), (2, −1, 1), (2, 1, −1)} with

TABLE 2. Results from the implementation of the proposed algorithm on two- and three-dimensional loops.

	Input parameters						Output parameters			
	Input dependence vectors parallel to	k	memeplexes	frogs	g	l	DCS	fitness	time	BDVS
Test 1	{(1, 1), (3, 2)}	300	50	50	20	10	11.3099	291.3783	112.36	{(1, 1), (3, 2)}
			30	30	20	10	18.4349	290.001		{(1, 1), (2, 1)}
Test 2	{(1, 2), (1, 1), (3, 1)}	70	50	50	10	10	45	64.4548	136.22	{(1, 2), (1, 1), (3, 1)}
Test 3	{(1, -2), (2, 1)}	400	20	20	20	10	90	383.4868	143.12	{(1, -2), (2, 1)}
Test 4	{(1, -1), (2, -1)}	100	20	30	20	10	18.4345	98.4549	84.42	{(1, -1), (2, -1)}
Test 5	{(3, -1), (2, -2), (1, -2)}	120	100	40	10	10	45	109.4514	245.13	{(2, -2), (1, -2), (3, -1)}
			30	30	20	20	45	109.4514		{(1, -1), (3, -1), (1, -2)}
Test 6	{(3, -1), (1, -2), (1, 1)}	400	40	40	20	20	108.4349	359.79	265.34	{(1, -2), (2, -2), (1, 1)}
			10	10	10	10	108.4349	359.79		{(1, -2), (1, 1), (1, 0)}
			20	20	10	10	108.4349	359.79		{(3, -1), (1, -2), (1, 1)}
Test 7	{(1, 2, 1), (1, 2, 3), (3, 2, 1)}	20	50	50	40	20	0.114	975.0295	267.04	{(1, 2, 1), (1, 2, 3), (1, 0, 0)}
			30	20	30	20	0.523	816.1039		{(0, 0, 1), (0, 1, -1), (1, 0, 1)}
			40	30	20	18	0.523	816.1039		{(1, 0, 0), (0, 1, 0), (0, 0, 1)}
Test 8	{(1, 1, 1), (0, 1, -1), (1, -1, 0)}	50	20	20	20	20	0.698	1035.93	43.36	{(1, 1, 1), (1, -1, 0), (0, 1, -1)}
			50	50	10	12	1.078	840.0786		{(0, 0, 1), (1, -1, 0), (0, 1, -1)}
Test 9	{(1, 2, -2), (1, -2, 2), (0, 1, 1)}	50	20	20	20	20	0.741	1028.80	84.23	{(1, 2, -2), (1, -2, 2), (0, 1, 1)}
Test 10	{(1, 2, 1), (2, -1, 1), (2, 1, -1)}	500	20	20	6	6	1.0196	1200.10	172.35	{(0, 0, 1), (0, 1, 0), (1, -1, -1)}
			20	20	12	12	1.2910	1199.90		{(0, 0, 1), (0, 1, -1), (1, -1, -1)}
			20	20	20	10	0.2670	1479.40		{(1, 2, 1), (2, -1, 1), (2, 1, - 1)}

DCS = 0.267 and a lack of a main vector can be selected as an optimal result. Fig. 8 shows the mean fitness of the seventh to tenth tests corresponding to the implementation of the proposed algorithm on three-dimensional loops. After ensuring the correctness of the proposed algorithm, we also consider performance and correctness on real application loops. For this purpose, the proposed algorithm was run separately on the two-level loop of Fig. 1 (eleventh test) and three-level loop of Fig. 9 adapted from [56]. The proposed algorithm was performed on 13 dependent vectors corresponding to the two-dimensional loop in Fig. 1 for more than ten times. The best response was $\{(0,1), (1, -2)\}$ with DCS = 153.4349 with an average execution time of 47.13 seconds.

The same result was obtained in all ten runs, and therefore the stability of the proposed algorithm for this test was found to be very high. For the three-dimensional loop of Fig. 9, after running ten times on 24 input dependence vectors, the two sets $\{(1, 1, 0), (0, 1, -1), (0, 0, 1)\}$ and $\{(2, 3, -3), (1, 2, 1), (1, 2, 0)\}$ with an average execution time of 142.56 were obtained as the result, between which the second set was a more appropriate result due to the lack of a main vector and DCS = 0.0087.

V. COMPARISON WITH OTHER METHODS

In uniformization methods that are not based on evolutionary algorithms [32], [33], [56], although they do not have

TABLE 3. Comparison of uniformization methods.

Methods	Type_of_loops	Average run time	Attention to three important input parameters	Considering the direction of vectors	Variety in the number of optimal results obtained
Non-evolutionary methods [32], [33], [56]	all	-	-	no	yes
UTLEA [24]	3D	high	yes	yes	no
UTFLA [38]	2D	low	yes	yes	no
The proposed algorithm	all	low	yes	yes	yes

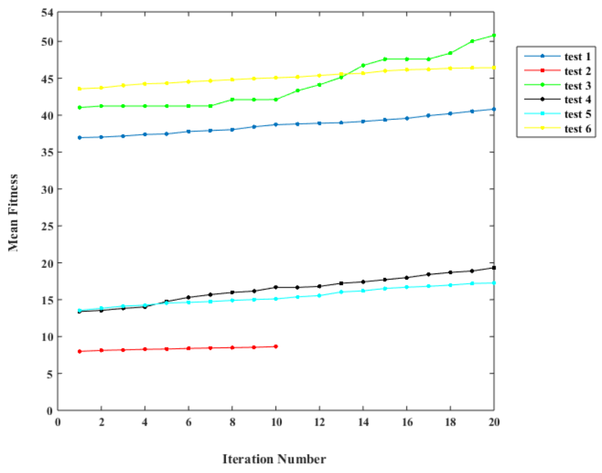

FIGURE 7. Mean fitness of the first to sixth tests on two-dimensional loops.

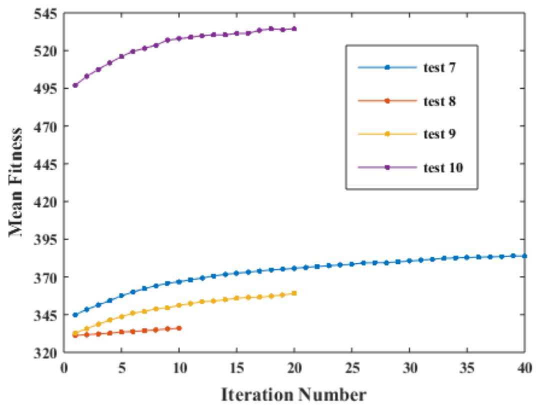

FIGURE 8. The mean fitness of the seventh to tenth experiments on three-dimensional loops.

an average execution time, in practice, their computational complexity is so high that strong mathematical knowledge is required to achieve the result. On the other hand, the existence or non-existence of main vectors is not considered at all.

```
For i=1:15
      For j=1:15
            For k=1:15
            A(3i+j-1, 4i+3j-3, k+1)=…
            …=A(i+1, j+1, k)
            EndFor
      EndFor
EndFor
```

FIGURE 9. Three-dimensional loop corresponding to the twelfth tests.

Moreover, in none of these methods is the direction of vectors considered, while in considering the implementation of loops and dependencies, it is quite necessary to pay attention to this parameter. Another important point is not paying attention to the three effective parameters in calculating BDVS, which has made the obtained results practically unusable in real parallel compilers; however, in the method proposed in this paper, all these shortcomings have been eliminated. Nevertheless, these shortcomings had already been eliminated by a number of solutions that need to be compared with the proposed method.

The first algorithm which is based on the evolutionary algorithm and had previously addressed these shortcomings is called UTLEA, which pays attention to both the three effective parameters and the direction of the vectors. However, due to its use of genetic algorithms, it has a very high execution time and cannot be used in practice for a large number of input vectors. For example, in the same hardware conditions, the average execution time for tests 7 to 10 is between twenty minutes and half an hour. UTFLA was subsequently proposed and was the first algorithm to use the FLA for uniformization. Although this algorithm solved many problems of its previous methods, because of its use of fixed parameters, most of the results obtained through it were not accurate enough, and the DCS remained high for many tests. For a more accurate comparison, all tests performed in the previous methods with the same datasets and similar hardware features were re-performed by the proposed algorithm in this paper, the details of which are given in Section 3.4. By comparing the results obtained in this paper with UTFLA, the following conclusions are evident:

1. The variety of results obtained is much greater due to avoiding the use of fixed coefficients to calculate fitness, along with the use of a fuzzy system.
2. In this paper, to check the low run time, most of the tests were performed on a large number of datasets to be closer to real loops, while in previous methods, the number of datasets was often kept low so that the average run time did not increase.
3. This algorithm can be used for all types of two- and three-dimensional loops, while each of the previous methods focuses on only one type of loop (two- or three-dimensional).

Comparing the results obtained in this paper with those of UTFLA, it can be concluded that with the same input, the present study obtained smaller DCS, a wider variety of results, and higher mean run time with regard to the quality of results. Table 3 summarizes the comparison of all the proposed uniformization algorithms so far.

VI. CONCLUSION AND IMPLICATIONS

There is no doubt that application parallelization is a good way to reduce application runtime. But the idealistic view of full-program parallelism by researchers in recent years has led to solutions that result in aborts when used in real-world applications, deviating from the main purpose of parallelism, which is to reduce runtime. For this reason, many researchers have focused on parallelization of loops rather than parallelization of the entire sequential program, as most applications' runtime is spent in loops. On the other hand, among the various steps of parallelization, most of the work is on tiling and scheduling; a smaller amount is on the method of uniformization. If a suitable solution is provided for uniformization, parallelization can be done using one of the most common methods of parallelization, namely the Wavefront method. On the other hand, in recent years, 3D data-processing has received much attention. These algorithms work with three-dimensional data spaces that require a lot of memory and high computing power due to their handling of three-dimensional loops, and only little work has been done on the uniformization of these loops. In this paper, a single algorithm was proposed for uniformization of both two- and three-dimensional loops, which, with a combination of evolutionary algorithms and a fuzzy system, achieves a considerable variety of results with very low DCS and average runtime. The results of this paper can be very attractive for HPC purposes because they can facilitate the development of parallel codes. Improving the uniformization of the parallelization process can have a significant impact on improving the performance of other steps. In fact, with proper uniformization, there will be no need to use algorithms with high computational complexity in later steps such as tiling, and only using one of the simplest methods available, namely Wavefront, parallelism can be performed. Suppose we could estimate how much the next steps would improve as the performance of the data dependence analysis step improved. In that case, we could use Amdahl's law to estimate how much the system as a whole had improved. Therefore, as a future implication, this algorithm can be used to propose and implement a parallel compiler.

REFERENCES

[1] E. Z. Zefreh, S. Lotfi, L. M. Khanli, and J. Karimpour, "Tiling and scheduling of three-level perfectly nested loops with dependencies on heterogeneous systems," *Scalable Comput., Pract. Exp.*, vol. 17, no. 4, pp. 331–349, Oct. 2016, doi: 10.12694/scpe.v17i4.1205.

[2] S. Kalyur and G. S. Nagaraja, "ParaCite: Auto-parallelization of a sequential program using the program dependence graph," in *Proc. Int. Conf. Comput. Syst. Inf. Technol. Sustain. Solutions (CSITSS)*, 2016, pp. 7–12. Accessed: Jan. 4, 2022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7779431/

[3] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of OpenMP tasks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 3, pp. 404–418, Mar. 2009. Accessed: Jan. 4, 2022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4553700

[4] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement (PLDI)*, 1998, pp. 212–223, doi: 10.1145/277650.277725.

[5] A. Fonseca, B. Cabral, J. Rafael, and I. Correia, "Automatic parallelization: Executing sequential programs on a task-based parallel runtime," *Int. J. Parallel Program.*, vol. 44, no. 6, pp. 1337–1358, Dec. 2016, doi: 10.1007/S10766-016-0426-5.

[6] V. A. Ying, M. C. Jeffrey, and D. Sanchez, "T4: Compiling sequential code for effective speculative parallelization in hardware," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, May 2020, pp. 159–172. Accessed: Jan. 4, 2022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9138940/

[7] P. Feautrier, "Automatic parallelization in the polytope model," in *The Data Parallel Programming Model* (Lecture Notes in Computer Science: Including Subsertes Lecture Notes in Artificial Intelligenceans Lecture Notes in Bioinformatics), vol. 1132. The Netherland: Springer, 1996, pp. 79–103, doi: 10.1007/3-540-61736-1_44.

[8] A. Fonseca and B. Cabral, "Controlling the granularity of automatic parallel programs," *J. Comput. Sci.*, vol. 17, pp. 620–629, Nov. 2016, doi: 10.1016/j.jocs.2016.06.005.

[9] G. Mustafa, W. Mahmood, and M. U. Ghani, "Just-in-time compilation-inspired methodology for parallelization of compute intensive Java code," *Mehran Univ. Res. J. Eng. Technol.*, vol. 36, no. 1, pp. 67–86, Jan. 2017, doi: 10.22581/muet1982.1701.08.

[10] R. Khatchadourian, Y. Tang, and M. Bagherzadeh, "Safe automated refactoring for intelligent parallelization of Java 8 streams," *Sci. Comput. Program.*, vol. 195, Sep. 2020, Art. no. 102476, doi: 10.1016/j.scico.2020.102476.

[11] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "PLuTo: A practical and fully automatic polyhedral program optimization system," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2008, pp. 1–15, Accessed: Jan. 4, 2022. [Online]. Available: https://www.researchgate.net/profile/J-Ramanujam/publication/228732823_PLuTo_A_practical_and_fully_automatic_polyhedral_program_optimization_system/links/0912f50b7b3c59b6c0000000/PLuTo-A-practical-and-fully-automatic-polyhedral-program-optimization-system.pdf

[12] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly—Performing polyhedral optimizations on a low-level intermediate representation," *Parallel Process. Lett.*, vol. 22, no. 4, 2012, Art. no. 1250010, doi: 10.1142/S0129626412500107.

[13] S. Apostolakis, Z. Xu, G. Chan, S. Campanoni, and D. I. August, "Perspective: A sensible approach to speculative automatic parallelization," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2020, pp. 351–367, doi: 10.1145/3373376.3378458.

[14] A. Jimborean, P. Clauss, J.-F. Dollinger, V. Loechner, and J. M. M. Caamaño, "Dynamic and speculative polyhedral parallelization using compiler-generated skeletons," *Int. J. Parallel Program.*, vol. 42, no. 4, pp. 529–545, Aug. 2013, doi: 10.1007/S10766-013-0259-4.

[15] J. M. M. Caamaño, A. Sukumaran-Rajam, A. Baloian, M. Selva, and P. Clauss, "APOLLO: Automatic speculative polyhedral loop optimizer," in *Proc. IMPACT 7th Int. Workshop Polyhedral Compilation Techn.*, Jan. 2017, p. 8. Accessed: Jan. 4, 2022. [Online]. Available: https://hal.inria.fr/hal-01533692

[16] D. Göhringer and J. Tepelmann, "An interactive tool based on polly for detection and parallelization of loops," in *Proc. Workshop Parallel Program. Run-Time Manage., Techn. Many-Core Archit. Design Tools Archit. Multicore Embedded Comput. Platforms*, 2014, pp. 1–6, doi: 10.1145/2556863.2556869.

[17] M. Booshehri, A. Malekpour, and P. Luksch, "An improving method for loop unrolling," Aug. 2013, *arXiv:1308.0698*. Accessed: Aug. 12, 2020.

[18] D. Liu, Y. Wang, Z. Shao, M. Guo, and J. Xue, "Optimally maximizing iteration-level loop parallelism," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 3, pp. 564–572, Mar. 2012, doi: 10.1109/TPDS.2011.171.

[19] C. Q. Zhu and P. C. Yew, "A scheme to enforce data dependence on large multiprocessor systems," *IEEE Trans. Softw. Eng.*, vol. SE-13, no. 6, pp. 726–739, Jun. 1987. Accessed: Aug. 22, 2020. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1702277/, doi: 10.1109/TSE.1987.233477.

[20] S. P. Midkiff and D. A. Padua, "Compiler algorithms for synchronization," *IEEE Trans. Comput.*, vol. C-36, no. 12, pp. 1485–1495, Dec. 1987, doi: 10.1109/TC.1987.5009499.

[21] B. Liu, J. He, Y. Geng, L. Huang, and S. Li, "Toward emotion-aware computing: A loop selection approach based on machine learning for speculative multithreading," *IEEE Access*, vol. 5, pp. 3675–3686, 2017, doi: 10.1109/ACCESS.2017.2684129.

[22] A. T. Chronopoulos, S. Penmatsa, J. Xu, and S. Ali, "Distributed loop-scheduling schemes for heterogeneous computer systems," *Concurrency Comput., Pract. Exp.*, vol. 18, no. 7, pp. 771–785, Jun. 2006, doi: 10.1002/CPE.960.

[23] R. L. Cariño and I. Banicescu, "A load balancing tool for distributed parallel loops," *Cluster Comput.*, vol. 8, no. 4, pp. 313–321, Oct. 2005, doi: 10.1007/S10586-005-4098-Y.

[24] S. Mahjoub and S. Lotfi, "The UTLEA: Uniformization of non-uniform iteration spaces in three-level perfect nested loops using an evolutionary algorithm," in *Proc. Int. Conf. Softw. Eng. Comput. Syst.*, in Communications in Computer and Information Science, vol. 180, 2011, pp. 605–617, doi: 10.1007/978-3-642-22191-0_52.

[25] L. Lamport, "The parallel execution of DO loops," *Commun. ACM*, vol. 17, no. 2, pp. 83–93, Feb. 1974, doi: 10.1145/360827.360844.

[26] S. Parsa and S. Lotfi, "Wave-fronts parallelization and scheduling," in *Proc. Innov. Inf. Technol. (IIT)*, Nov. 2007, pp. 382–386, doi: 10.1109/IIT.2007.4430369.

[27] R. Searles, S. Chandrasekaran, W. Joubert, and O. Hernandez, "Abstractions and directives for adapting wavefront algorithms to future architectures," in *Proc. Platform Adv. Sci. Comput. Conf.*, Jul. 2018, pp. 1–10, doi: 10.1145/3218176.3218228.

[28] Y. Li and L. Schwiebert, "Memory-optimized wavefront parallelism on GPUs," *Int. J. Parallel Program.*, vol. 48, no. 6, pp. 1008–1031, 2020, doi: 10.1007/s10766-020-00658-y.

[29] Y. Zou and S. Rajopadhye, "A code generator for energy-efficient wavefront parallelization of uniform dependence computations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 9, pp. 1923–1936, Sep. 2018. Accessed: Feb. 28, 2022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7935535/

[30] W. Bielecki and M. Poliwoda, "Automatic parallel tiled code generation based on dependence approximation," in *Proc. Int. Conf. Parallel Comput. Technol.*, in Lecture Notes in Computer Science: Including Subsertes Lecture Notes in Artificial Intelligenceans Lecture Notes in Bioinformatics, vol. 12942, Sep. 2021, pp. 260–275, doi: 10.1007/978-3-030-86359-3_20.

[31] A. Abdollahi-Kalkhoran, S. Lotfi, and H. Izadkhah, "TEA-SEA: Tiling and scheduling of non-uniform two-level perfectly nested loops using an evolutionary approach," *Exp. Syst. Appl.*, vol. 191, Apr. 2022, Art. no. 116152. Accessed: Feb. 28, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417421014779

[32] T. H. Tzen and L. M. Ni, "Dependence uniformization: A loop parallelization technique," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 5, pp. 547–558, May 1993, doi: 10.1109/71.224217.

[33] D.-K. Chen and P.-C. Yew, "On effective execution of nonuniform DOACROSS loops," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 5, pp. 463–476, May 1996, doi: 10.1109/71.503771.

[34] M. M. Eusuff, K. E. Lansey, and F. Pasha, "Shuffled frog-leaping algorithm: A memetic meta-heuristic for discrete optimization," *Eng. Optim.*, vol. 38, no. 2, pp. 129–154, Mar. 2006, doi: 10.1080/03052150500384759.

[35] E. Elbeltagi, T. Hegazy, and D. Grierson, "Comparison among five evolutionary-based optimization algorithms," *Adv. Eng. Informat.*, vol. 19, no. 1, pp. 43–53, 2005, doi: 10.1016/j.aei.2005.01.004.

[36] R. Srivastava, S. K. Singh, and K. K. Shukla, *Research Developments in Computer Vision and Image Processing: Methodologies and Applications*. The Netherland: Springer, Sep. 2013, pp. 1–425, doi: 10.4018/978-1-4666-4558-5.

[37] E. Z. Zefreh, S. Lotfi, L. M. Khanli, and J. Karimpour, "3-D data partitioning for 3-level perfectly nested loops on heterogeneous distributed systems," *Concurrency Comput., Pract. Exp.*, vol. 29, no. 5, Mar. 2017, Art. no. e3976, doi: 10.1002/cpe.3976.

[38] S. Mahjoub and H. Vojoudi, "The UTFLA: Uniformization of non-uniform iteration spaces in two-level perfect nested loops using SFLA," *J. Supercomput.*, vol. 72, no. 6, pp. 2221–2234, Jun. 2016, doi: 10.1007/s11227-016-1725-8.

[39] M. Wolfe, "Parallelizing compilers," *ACM Comput. Surv.*, vol. 28, no. 1, pp. 261–262, Mar. 1996, doi: 10.1145/234313.234417.

[40] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua, "Automatic program parallelization," *Proc. IEEE*, vol. 81, no. 2, pp. 211–243, Feb. 1993. Accessed: Aug. 21, 2020. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/214548/

[41] B. Chan and T. S. Abdelrahman, "Run-time support for the automatic parallelization of Java programs," *J. Supercomput.*, vol. 28, no. 1, pp. 91–117, Apr. 2004, doi: 10.1023/B:SUPE.0000014804.20789.21.

[42] D. Dig, J. Marrero, and M. D. Ernst, "Refactoring sequential Java code for concurrency via concurrent libraries," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, May 2009, pp. 397–407, doi: 10.1109/ICSE.2009.5070539.

[43] J. C. Jenista, Y. H. Eom, and B. C. Demsky, "OoOJava: Software out-of-order execution," *ACM SIGPLAN Notices*, vol. 46, no. 8, pp. 57–68, Sep. 2011, doi: 10.1145/2038037.1941563.

[44] R. Khatchadourian, Y. Tang, M. Bagherzadeh, and S. Ahmed, "Safe automated refactoring for intelligent parallelization of Java 8 streams," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 619–630, doi: 10.1109/ICSE.2019.00072.

[45] O. Plata, R. Asenjo, E. Gutiérrez, F. Corbera, A. Navarro, and E. L. Zapata, "On the parallelization of irregular and dynamic programs," *Parallel Comput.*, vol. 31, no. 6, pp. 544–562, Jun. 2005, doi: 10.1016/j.parco.2005.02.012.

[46] S. Campanoni, K. Brownell, S. Kanev, T. M. Jones, G.-Y. Wei, and D. Brooks, "HELIX-RC: An architecture-compiler co-design for automatic parallelization of irregular programs," in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit. (ISCA)*, Jun. 2014, pp. 217–228, doi: 10.1109/ISCA.2014.6853215.

[47] J. C. Huang and T. Leng, "Generalized loop-unrolling: A method for program speedup," in *Proc. IEEE Symp. Appl.-Specific Syst. Softw. Eng. Technol. (ASSET)*, Mar. 1999, pp. 244–248, doi: 10.1109/ASSET.1999.756775.

[48] D.-K. Chen, J. Torrellas, and P.-C. Yew, "An efficient algorithm for the run-time parallelization of DOACROSS loops," in *Proc. ACM/IEEE Conf. Supercomput.*, Nov. 1994, pp. 518–527, doi: 10.1109/SUPERC.1994.344315.

[49] C. Z. Xu and V. Chaudhary, "Time stamp algorithms for runtime parallelization of DOACROSS loops with dynamic dependences," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 5, pp. 433–450, May 2001, doi: 10.1109/71.926166.

[50] L. Rauchwerger and D. A. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 2, pp. 160–180, Feb. 1999, doi: 10.1109/71.752782.

[51] A. W. Lim, S.-W. Liao, and M. S. Lam, "Blocking and array contraction across arbitrarily nested loops using affine partitioning," in *Proc. 8th ACM SIGPLAN Symp. Princ. Practices Parallel Program.*, 2001, pp. 103–112, doi: 10.1145/379539.379586.

[52] K. Iwasawa, "Detecting method of parallelism from nested loops with loop carried data dependences," in *Proc. 5th Int. Multi-Conf. Comput. Global Inf. Technol.*, Sep. 2010, pp. 287–292, doi: 10.1109/ICCGI.2010.11.

[53] K. Jingu, K. Shigenobu, K. Ootsu, T. Ohkawa, and T. Yokota, "Directive-based parallelization of for-loops at LLVM IR level," in *Proc. 20th IEEE/ACIS Int. Conf. Softw. Eng., Artif. Intell., Netw. Parallel/Distrib. Comput. (SNPD)*, Jul. 2019, pp. 421–426, doi: 10.1109/SNPD.2019.8935667.

[54] A. Kotha, K. Anand, M. Smithson, G. Yellareddy, and R. Barua, "Automatic parallelization in a binary rewriter," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2010, pp. 547–557, doi: 10.1109/MICRO.2010.27.
[55] A. Tarhini, "Automatic loop parallelization," Amer. Univ. Beirut, Tech. Rep. 12029, 2013.
[56] W. Shang, E. Hodzic, and Z. Chen, "On uniformization of affine dependence algorithms," *IEEE Trans. Comput.*, vol. 45, no. 7, pp. 827–840, Jul. 1996. Accessed: Aug. 22, 2020. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/508321/, doi: 10.1109/12.508321.

SHABNAM MAHJOUB was born in Lahijan, Iran, in 1986. She received the B.S. degree in software engineering from Islamic Azad University, Lahijan, in 2008, and the M.S. degree in software engineering from Islamic Azad University, Shabestar, Iran, in 2011. She is currently pursuing the Ph.D. degree in software with Islamic Azad University, Babol Branch, Iran. Since 2011, she has been a Faculty Member at Islamic Azad University, Langarud Branch, Iran. Her research interests include the development of parallel processing, and evolutionary-based algorithms and fuzzy systems.

MEHDI GOLSORKHTABARAMIRI (Senior Member, IEEE) received the M.Sc. degree in computer systems architecture engineering, and the Ph.D. degree in computer engineering from the Science and Research Branch, Islamic Azad University, Tehran, Iran. Since 2011, he has been working as an Assistant Professor with the Department of Computer Engineering, Islamic Azad University, Babol Branch, Iran. He has authored and coauthored many high-cited scientific papers published in peer-reviewed conference proceedings and international journals. His current research interests include radio frequency identification (RFID) systems, wireless sensor networks (WSNs), wireless body area networks (WBANs), and computer communication.

SEYED SADEGH SALEHI AMIRI was born in Babol, Iran, in 1980. He received the B.S. degree in pure mathematics from Payam Noor University, Babol, in 2003, the M.S. degree in pure mathematics from the Amirkabir University of Technology, Tehran, Iran, in 2005, the Ph.D. degree in mathematics algebra from Islamic Azad University, Science and Research Branch, Tehran, in 2012, and the Ph.D. degree in pure mathematics analysis trend from the University of Mazandaran, Iran, in 2017. Since 2009, he has been a Faculty Member at Islamic Azad University, Babol Branch, Iran. His research interests include mathematics, algebra, and analysis.

MEHDI HOSSEINZADEH received the B.S. degree in computer hardware engineering from Islamic Azad University, Dezful Branch, Iran, in 2003, and the M.Sc. and Ph.D. degrees in computer system architecture from the Science and Research Branch, Islamic Azad University, Tehran, Iran, in 2005 and 2008, respectively. He has authored/coauthored more than 150 publications in technical journals and conferences. His research interests include SDN, information technology, data mining, big data analytics, e-commerce, e-marketing, and social networks.7.

AMIR MOSAVI received the graduate degree from London Kingston University, U.K., and the Ph.D. degree in applied informatics. He is currently an Associate Professor of computer science interested in big data, the IoT, and machine learning. He is also a Senior Research Fellow at Oxford Brookes University. He is a Data Scientist for climate change, sustainability, and hazard prediction. He was a recipient of the Green-Talent Award, the UNESCO Young Scientist Award, the ERCIM Alain Bensoussan Fellowship Award, the Campus France Fellowship Award, the Campus Hungary Fellowship Award, and the Endeavour-Australia Leadership.