# Multifor for Multicore

Imèn Fassi
Dpt of Computer Science
Faculty of Sciences of Tunis
University El Manar
1060 Tunis, Tunisia
fassi.imen@gmail.com

Philippe Clauss
Team CAMUS, INRIA
University of Strasbourg
boulevard S. Brant
67400 Illkirch, France
philippe.clauss@inria.fr

Matthieu Kuhn
Team ICPS, LSIIT lab.
University of Strasbourg
boulevard S. Brant
67400 Illkirch, France
kuhn@unistra.fr

Yosr Slama
Dpt of Computer Science
Faculty of Sciences of Tunis
University El Manar
1060 Tunis, Tunisia
yosr.slama@gmail.com

## ABSTRACT

We propose a new programming control structure called "multifor", allowing to take advantage of parallelization models that were not naturally attainable with the polytope model before. In a multifor-loop, several loops whose bodies are run simultaneously can be defined. Respective iteration domains are mapped onto each other according to a run frequency – the grain – and a relative position – the offset –. Execution models like dataflow, stencil computations or MapReduce can be represented onto one referential iteration domain, while still exhibiting traditional polyhedral code analysis and transformation opportunities. Moreover, this construct provides ways to naturally exploit hybrid parallelization models, thus significantly improving parallelization opportunities in the multicore era. Traditional polyhedral software tools are used to generate the corresponding code. Additionally, a promising perspective related to non-linear mapping of iteration spaces is also presented, yielding to run a loop nest inside any other one by solving the problem of inverting "ranking Ehrhart polynomials".

## Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors

## General Terms

Performance

## Keywords

programming control structure, parallel programming, polytope model

## 1. INTRODUCTION

We have definitely entered a new era in programming. Parallelism is everywhere, from the many-core processor architectures that are from now on fitting mainstream computers, to the software applications that are now mixing intensive computations, specialized routines, network communication and multi-threading. Many researches focus in proposing new languages supposed to facilitate programming inside this complex environment [8, 10, 11, 13], or in proposing hardware or software support like transactional memory systems [6, 14, 15], supposed to prevent incorrect computations while still providing good performance. However, all these proposals face intractable issues. Most proposed languages imply to change drastically programmers habits and have weak chances to be adopted by the software industry [4]. Moreover, even if they offer interesting constructions to express parallel tasks, they are not solving the fundamental problem of correct and efficient parallelism extraction, which induces dependency analysis, data locality optimization, task grain adjustment, etc. Performance is also strongly dependent of their implementation, i.e. of their compilers or runtime systems. On the other hand, hardware or software support does not either result in hiding parallelization complexity to the user. And overall, these mechanisms are of high complexity by themselves, which mostly often make them unrealistic for a real usage [3].

Nevertheless, parallel programming has already a long history, where gradual extensions have been proposed. Some of them were pretty successful and are still current. For example, directive-based languages, as OpenMP [2], are extensions to mainstream languages. The use of their instructions is not mandatory when inserted in a source code, and they can be discovered and adopted progressively by any developer. They are not breaking the programming habits, while offering efficient parallelization opportunities. Although they are not solving either the fundamental complexity of parallelization, they nicely open the door of high performance computing to anyone.

At the same time, a lot of relevant transformation techniques have been discovered in order to exhibit parallelism or to optimize code, particularly on loops, as software pipelining, loop unrolling, tiling, skewing, etc [1, 16, 12]. These are applied either by experienced programmers, or automatically by compilers or optimization tools. In the parallelism era, compilers and runtime systems have to accelerate their progresses in automatic parallelization, but at the same time, programmers have to be brought to become, at least, who we called "experienced programmers" ten or twenty years ago.

We argue that a good way to achieve such an emancipation to parallel programming is to gradually extend mainstream languages with new programming control structures

that are derived from already existing ones. Our idea is that many well-known optimizing and parallelizing code transformations should now be applied naturally by developers, but only in their minds, while using a control structure translating their enriched algorithmic reasonings. The existence of such control structures will condition them to enlarge their way of reasoning while programming. In the same way that it is currently natural to express the repetition of code blocks by using loops, or to abstract parametrized code by using functions, it should now be natural to bring closer instructions that are using the same operands, or to arrange code snippets in vertical and horizontal directions to express simultaneity, sequencing and overlapping.

Following this idea, we propose a new control structure called *Multifor*, which can be seen as an extension of for-loops to multiple for-loops running at the same time, whose respective instructions are run sequentially inside one loop body as a traditional for-loop, but run in any interleaved order, or in parallel, between the bodies. Additionally to traditional parameters as indices, bounds and steps, we propose to introduce a grain and an offset, allowing to mix loops of different execution frequencies and of different starting positions. Such programming construction translates naturally to code transformations as loop fusion, loop pipelining or loop partitioning. Moreover, it facilitates many code optimizations as data locality improvement or parallelization. It can be seen as an extension of for-loops from "vertical" to "horizontal" programming.

The second motivation of this proposal is related to the parallel programming models that are covered by the polytope model. Traditional application of this model does not allow to naturally express parallel programming models as task parallelism, dataflow or MapReduce. We show that the multifor construct allows to schedule loop nest processes by mapping together their respective iteration domains. A multifor code can be represented geometrically by a particular union of polyhedra, each being previously dilated, compressed or translated, either entirely or partially, according to transformation factors defined by constants or affine functions.

This new programming structure implies interesting implementation challenges for a compiler, from its front-end to its backend. We show that, as it is already the case with for-loops, the polytope model is quite well adapted to analyze, optimize and translate multifor constructs into efficient code.

Finally, as a promising perspective, we also propose a non-linear mapping of the iteration domains guided by the ranks of the iterations. This approach opens the possibility of mapping any iteration domain onto any other domain without being constrained by their shapes. It leads to solve the general problem of executing any loop nest by any other loop nest of the same trip count. Mathematically speaking, the general solution to this problem is based on inverting "ranking Ehrhart polynomials" [5, 9].

The paper is organized as follows. In the next section, syntax and semantics of multifor loops are described, illustrated with a few examples of multifor headers and graphical representations. We also highlight the code parallelization and transformation schemes that are possible with multifor-loops. In Section 3, we discuss the main issues related to the implementation of multifor constructs and their corresponding code generation. Several real and representative code examples are presented in Section 4, highlighting the interesting contributions of this new control structure. The promising perspective of non-linear iteration space mapping is the topic of Section 5, where a solution for inverting ranking Ehrhart polynomials is proposed. Finally, conclusions and further perspectives are given in Section 6.

# 2. SYNTAX AND SEMANTICS

In this paper, we describe the initial syntax and semantics for the multifor construct. However, they can be extended in many ways in the future. We first present the case of one unique multifor construct, as the case of nested multifor-loops present some specificities which are presented afterwards.

## 2.1 Non-nested multifor-loops

The multifor syntax is defined by:

$$
\begin{array}{ll}
\textbf{multifor} \ ( & index_1 = expr, [index_2 = expr, ...] \ ; \\
& index_1 < expr, [index_2 < expr, ...] \ ; \\
& index_1 += cst, [index_2 += cst, ...] \ ; \\
& grain_1, [grain_2, ...] \ ; \\
& offset_1, [offset_2, ...] \ ) \ \ \{ \\
& prefix : \{statements\} \\
\}
\end{array}
$$

where [...] denotes optional arguments, $index_i$ denotes the indices of the loops composing the multifor, $expr$ denotes affine arithmetic expressions on enclosing loop indices, or constants, $cst$ denotes an integer constant, $grain$ and $offset$ are positive integers, $grain \geq 1, offset \geq 0$, and $prefix$ is a positive integer associating each statement to a given for-loop composing the multifor-loop, according to the order in which they are defined (0 for the first loop, 1 for the second loop, etc.). Without loss of generality, we consider in the following that the index steps, $cst$, always equal one, since the general case can be easily deduced.

Each for-loop composing the multifor behaves as a traditional for-loop, but all are mapped on a same global "virtual referential" domain, which can also be seen as a template. The way iterations of the for-loops are mapped is defined by their respective offset and grain. The grain defines the frequency in which the associated loop has to run, relatively to the referential. For instance, if the grain equals 2, then one iteration of the associated loop will run over 2 iterations of the referential. The offset defines the gap between the first iteration of the referential and the first iteration of the associated loop. For instance, if the offset equals 3, then the first iteration of the associated loop will run at the fourth iteration of the referential loop.

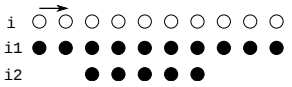
The size and shape of the referential is deduced from the for-loops composing the multifor-loop. Geometrically, it is defined as the disjoint union of the for-loop domains, where each domain has been previously shifted according to its offset and dilated according to its grain. The disjoint union is the union of adjacent convex domains, each being scanned by a referential for-loop. The relative positions of the iterations of the for-loops composing the multifor-loop inside the referential depends of the overlapping of their respective domains. It means that on domains where only one for-loop iterations are run, the grain becomes a compression factor. In general, the greatest common divisor of the grains of all the for-loops overlapping on a same referential domain

is used as the factor for compressing the points of this referential domain, according to the lexicographic order. On domains where several for-loops iterations are run, these are run in interleaved fashion, or simultaneously.

Let us illustrate this definition with a few examples. Consider the following multifor-loop header:

**multifor** $(i_1 = 0, i_2 = 10; i_1 < 10, i_2 < 15; i_1{+}{+}, i_2{+}{+}; 1, 1; 0, 2)$

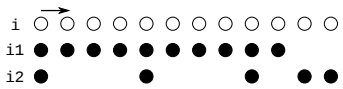
In this example, the offset of index $i_1$ is zero, and the one of index $i_2$ is 2. Thus, the first iteration of the $i_1$-loop will run immediatly, while the first iteration of the $i_2$-loop will run at the $3^{rd}$ iteration of the multifor, but with $i_2 = 0$. This behavior is illustrated by the figure below:




Notice that the index values have no effect on the relative positions of the for-loops bodies, which are uniquely determined by the grain and the offset. Another example is:

**multifor** $(i_1 = 0, i_2 = 10; i_1 < 10, i_2 < 15; i_1{+}{+}, i_2{+}{+}; 1, 4; 0, 0)$

Now, the $i_1$-grain is 1 and the $i_2$-grain is 4. In such a case, for one iteration of the $i_2$-loop, four iterations of the $i_1$-loop will be run on the domain on which they overlap. The second domain is compressed by a factor of 4, since only the $i_2$-loop is run, as it is illustrated below:




## 2.2 Nested multifor-loops

Nested multifor-loops present some particularities and specific semantics has to be described. Without loss of generality, let us consider two nested multifor-loops composed of two for-loop nests:

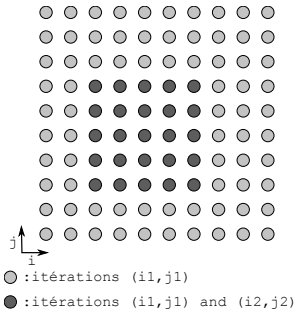
**multifor** ( $index_1 = expr, index_2 = expr;$
$index_1 < expr, index_2 < expr;$
$index_1 += cst, index_2 += cst;$
$grain_1, grain_2;$
$offset_1, offset_2$ ) {
$prefix : \{statements\}$
  **multifor** ( $index_3 = expr, index_4 = expr;$
$index_3 < expr, index_4 < expr;$
$index_3 += cst, index_4 += cst;$
$grain_3, grain_4;$
$offset_3, offset_4$ ) {
$prefix : \{statements\}$
  }
$prefix : \{statements\}$
}

Such a nest behaves as two for-loop nests, $(index_1, index_3)$ and $(index_2, index_4)$ respectively, running simultaneously in the same way as it is for one unique multifor-loop. The grain of the inner multifor-loop introduces a delay for the associated for-loop, since the same reasoning as with the non-nested case is applied at each loop depth. The lower and upper bounds are affine functions of the enclosing loop indices of the same for-loop[1]. Let us consider some examples of nested multifor headers.

[1]Notice that this restriction could be evicted for some amazing extensions.

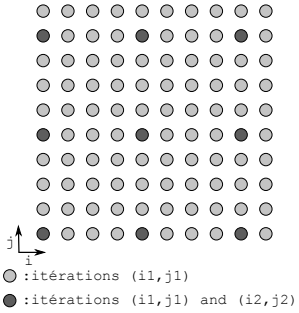
**multifor** $(i_1 = 0, i_2 = 0; i_1 < 10, i_2 < 5; i_1 + +, i_2 + +; 1, 1; 0, 2)$
**multifor** $(j_1 = 0, j_2 = 0; j_1 < 10, j_2 < 5; j_1 + +, j_2 + +; 1, 1; 0, 2)$

The second for-loop nest has a 2-offset at each loop depth. Hence it is delayed in each dimension of the referential domain:



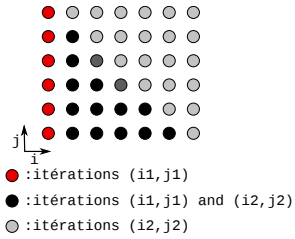

**multifor** $(i_1 = 0, i_2 = 0; i_1 < 10, i_2 < 3; i_1 + +, i_2 + +; 1, 4; 0, 0)$
**multifor** $(j_1 = 0, j_2 = 0; j_1 < 10, j_2 < 3; j_1 + +, j_2 + +; 1, 4; 0, 0)$

The second for-loop nest has a 4-grain at each loop depth. Hence its iterations are spaced by 4 in each dimension of the referential domain:




**multifor** $(i_1 = 0, i_2 = 0; i_1 < 6, i_2 < 6; i_1 + +, i_2 + +; 1, 1; 0, 1)$
**multifor** $(j_1 = 0, j_2 = 0; j_1 < 6 - i_1, j_2 < 6; j_1 + +, j_2 + +; 1, 1; 0, 0)$

In this example, the upper bound of the inner loop of the first loop nest is an affine function.




## 2.3 Multifor-loop parallelization and code transformations

The multifor construct exhibits a straightforward parallelization strategy which is to run, at each iteration, the loop bodies of the defined for-loops in parallel. This model of parallelization provides new opportunities to the polytope model, since it enables the expression of parallel programming models that were unattainable before, as dataflow computing or fixed-depth MapReduce, as it will be shown on code examples in Section 4.

Nevertheless, the multifor construct still allows OpenMP-like loop parallelization for each for-loop of the multifor, thus providing hybrid parallelization strategies.

Moreover, each for-loop, or for-loop nest, of a multifor, can be transformed using any well-known polyhedral transformation. However, in the context of a multifor, these transformations may be guided by the interactions between the for-loops, in order to achieve better performance or better data locality for instance. Another opportunity is the transformation of imperfect loop nests into multifor-loop nests of perfectly nested loops.

# 3. IMPLEMENTATION ISSUES

## 3.1 Reference domain

Consider one multifor-loop level. The referential for-loops cadencing the multifor execution have the constraint of scanning a sufficient number of iterations. Let us denote by $f$ the number of for-loops. By computing the disjoint union of all for-loops iteration domains, we obtain a set of adjacent domains $D_i$ on which some of the $f$ loops overlap. Let us denote by $lb_i, ub_i, grain_i$ and $offset_i, i = 1..f$ the parameters characterizing each for-loop in the multifor header. Let us set $nlb_i = offset_i$ and $nub_i = (ub_i - lb_i + 1) \times grain_i + offset_i$, which define the lower and upper bounds of each loop in the referential domain, since the computation of $nlb_i$ consists in translating the domain and the computation of $nub_i$ in dilating the domain by a factor which equals the grain. The disjoint union of $D_i$'s is computed using these latter bounds. The initial index value of the referential domain is $MIN_{i=1..f}\,(nlb_i)$. Hence, the total number of iterations in the referential domain, before compression, is:

$$MAX_{i=1..f}\,(nub_i) - MIN_{i=1..f}\,(nlb_i) + 1$$

In order to generate the referential for-loops for each domain $D_i$, the last step consists in compressing each $D_i$ by a factor defined by $lcm(grain_j)$, for all loops $j$ overlapping on $D_i$.

More generally for any multifor-loop nest, the computation of the referential domain is performed in three steps. First, each iteration domain associated to one for-loop nest composing a multifor-loop nest is translated from the origin according to its offsets, and dilated according to its grains in every dimension. Notice that values actually taken by the indices of the for-loop nests are not defining their positions in the referential domain. Second, a disjoint union is computed and resulting in a union of adjacent convex domains $D_i$. Third, each $D_i$ may be compressed according to the greatest common divisor of the grains of the associated for-loop nests, and according to the lexicographic order.

## 3.2 Code generation

When considering sequential code, there are two dual ways to generate the code corresponding to a multifor-loop nest. A first way is to generate loop nests scanning the referential domains through a minimal set of convex domains, and to insert guards in their bodies in order to execute the convenient instructions at each iteration. The number of these guards can be optimized by computing their common sub-domains. The second way is to scan each $D_i$ using a dedicated loop nest with a constant loop body, without guards.

Both solutions can be generated automatically using polytope model tools like PolyLib or CLooG, and by inserting phases to compute the translated and dilated domains, or to compress parts of the resulting referential domains.

If for-loops of given depth of a multifor-loop nest have to be run in parallel, each for-loop has to be run in a separated thread and all threads have to be synchronized at the multifor-loop completion. Notice that this could be enriched by providing OpenMP-like options as NOWAIT.

```
for (i = 0; i < K; i++)
 for (j = 0; j < N; j++)
  a[i][j] = ReadImage();
for (i = 1; i < K - 1; i++)
 for (j = 1; j < N - 1; j++){
  Sbl[i][j] = Sobel(a[i - 1][j - 1], a[i][j - 1], a[i + 1][j - 1],
                    a[i - 1][j], a[i][j], a[i + 1][j],
                    a[i - 1][j + 1], a[i][j + 1], a[i + 1][j + 1]);
  WriteImage(Sbl[i][j]);
}
```

**Figure 1: Sobel edge detection code**

```
multifor (i1 = 0, i2 = 1; i1 < K, i2 < K - 1;
          i1++, i2++; 1, 1; 0, 3)
 multifor (j1 = 0, j2 = 1; j1 < N, j2 < N - 1;
           j1++, j2++; 1, 1; 0, 3) {
  0 : a[i1][j1] = ReadImage();
  1 : {Sbl[i2][j2] = Sobel(a[i2 - 1][j2 - 1], a[i2][j2 - 1],
                           a[i2 + 1][j2 - 1], a[i2 - 1][j2],
                           a[i2][j2], a[i2 + 1][j2],
                           a[i2 - 1][j2 + 1], a[i2][j2 + 1],
                           a[i2 + 1][j2 + 1]);
       WriteImage(Sbl[i2][j2]); }
 }
```

**Figure 2: Sobel edge detection multifor code**

Original indices of the multifor ($i_1$, $i_2$, $j_1$, ...) have to be retrieved at the beginning of each loop body, by being computed from the referential loop indices. These computations consists in subtracting offsets, or in adding modulos of the referential loop indices relatively to grains, or in multiplying by grains in case of compressed domains.

# 4. EXAMPLES

**Sobel edge detection:** We first consider the code for performing Sobel edge detection of an image shown in Figure 1. The first loop nest of this program reads the input image, while the second loop nest performs the actual edge detection and writes out the output image.

Note that nine neighboring elements have to be read before its resulting pixel can be computed and written. Hence both loop nests can be naturally overlapped by writing the code using the multifor construct exhibiting a data-flow model of computation, shown in Figure 2. The associated referential domain is shown in Figure 3.

**Red-Black Gauss-Seidel:** The second example is the Red-Black Gauss-Seidel algorithm composed of two phases. The first phase consists in updating the red elements of a grid, which are one point over two in the $i$ and $j$ directions of the grid, starting from the first bottom left corner, using their North-South-East-West (NSEW) neighbors, which are black elements. The second phase consists obviously in updating the black elements from the red ones. For a 2D $N \times N$ problem, the usual code, is of the form shown in Figure 4 (the border elements initialization has been omitted).

On the iteration domain and at each phase, a different

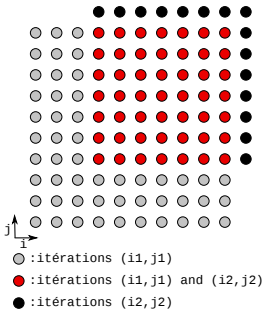



**Figure 3: Sobel edge detection referential domain**

```
// Red phase
for (i = 1; i < N − 1; i++)
 for (j = 1; j < N − 1; j++)
  if ((i + j) % 2 == 0)
    u[i][j] = f(u[i][j + 1], u[i][j − 1], u[i − 1][j], u[i + 1][j]);
// Black phase
for (i = 1; i < N − 1; i++)
 for (j = 1; j < N − 1; j++)
  if ((i + j) % 2 == 1)
    u[i][j] = f(u[i][j + 1], u[i][j − 1], u[i − 1][j], u[i + 1][j]);
```

**Figure 4: Red-Black Gauss-Seidel code**

lattice of iterations is active. Moreover, the NSEW dependencies prevent any linear parallel schedule. This code can be translated into a multifor-loop nest where the red and black phases each yield two for-loop nests with convenient grains and offsets as shown in Figure 5.

The referential initial domain of the multifor code is represented in Figure 6 on the left, also showing the transformed dependency vectors which are now allowing a linear schedule. On the right in Figure 6, the final iteration domains, after compression, are represented.

This example shows that the multifor construct allows to exploit different kind of parallelism – the so-called wavefront parallelism in this case – since the traditional parallelization consists in parallelizing each of the phases, and to execute each phase one after the other. The multifor strategy can be preferable to improve data locality and thus improve the

```
multifor (i0 = 1, i1 = 2, i2 = 1, i3 = 2; i0 < N − 1, i1 < N − 1,
          i2 < N − 1, i3 < N − 1; i0+ = 2, i1+ = 2, i2+ = 2,
          i3+ = 2; 2, 2, 2, 2; 0, 1, 1, 2)
multifor (j0 = 1, j1 = 2, j2 = 2, j3 = 1; j0 < N − 1, j1 < N − 1,
          j2 < N − 1, j3 < N − 1; j0+ = 2, j1+ = 2, j2+ = 2,
          j3+ = 2; 2, 2, 2, 2; 0, 1, 2, 1) {
0 : u[i0][j0] =
    f(u[i0][j0 + 1], u[i0][j0 − 1], u[i0 − 1][j0], u[i0 + 1][j0]);
1 : u[i1][j1] =
    f(u[i1][j1 + 1], u[i1][j1 − 1], u[i1 − 1][j1], u[i1 + 1][j1]);
2 : u[i2][j2] =
    f(u[i2][j2 + 1], u[i2][j2 − 1], u[i2 − 1][j2], u[i2 + 1][j2]);
3 : u[i3][j3] =
    f(u[i3][j3 + 1], u[i3][j3 − 1], u[i3 − 1][j3], u[i3 + 1][j3]);
}
```

**Figure 5: Red-Black Gauss-Seidel multifor code**

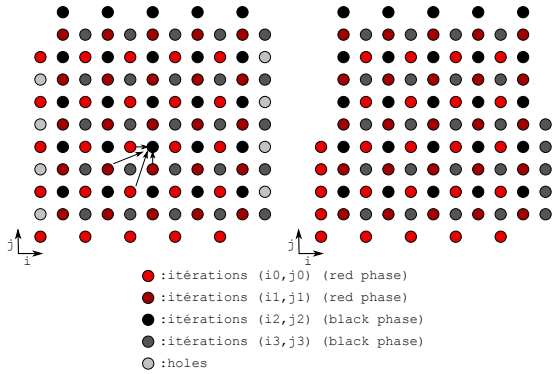



**Figure 6: Red-Black Gauss-Seidel referential domain**

```
for (j = 1; j < N − 1; j+ = 2)
 u[i][j] = f(u[i][j + 1], u[i][j − 1], u[i − 1][j], u[i + 1][j]);
for (i = 2; i < N − 2; i+ = 2) {
 for (j = 2; j < N − 1; j+ = 2) {
  u[i][j] = f(u[i][j + 1], u[i][j − 1], u[i − 1][j], u[i + 1][j]);
  u[i][j + 1] = f(u[i][j + 2], u[i][j],
                  u[i − 1][j + 1], u[i + 1][j + 1]); }
 for (j = 1; j < N − 1; j+ = 2) {
  u[i + 1][j] = f(u[i + 1][j + 1], u[i + 1][j − 1],
                  u[i][j], u[i + 2][j]);
  u[i + 1][j + 1] = f(u[i + 1][j + 2], u[i + 1][j],
                      u[i][j + 1], u[i + 2][j + 1]); } }
for (j = 2; j < N − 1; j+ = 2) {
 u[N − 2][j] = f(u[N − 2][j + 1], u[N − 2][j − 1],
                 u[N − 3][j], u[N − 1][j]); }
```

**Figure 7: Red-Black Gauss-Seidel generated code**

resulting execution time. Here, some parallelism within the red points and within the black points can still be exploited, but also parallelism between red and black points. As an example, we show as a source code the sequential code that could be generated from this multifor-loop nest in Figure 7.

Notice also that more generally, when dependencies allow it, such a decomposition of a loop-nest computation into separated lattices, and expressed as a multifor-loop nest, provides another parallelization strategy that may be often quite interesting due to data locality issues.

**Matrix product by blocks:** The third example is an algorithm to compute the product of two matrices $n \times n$, $(A \times B = C)$, by partitioning the matrices into uniform blocks. The matrix product is then carried out block by block. We split the two matrices $A$ and $B$ as follows:

- matrix $A$ is divided into two matrices $A_1$ and $A_2$ whose dimension is $n/2 \times n$.
- matrix $B$ is divided into two matrices $B_1$ and $B_2$ whose dimension is $n \times n/2$.

The product $A \times B = C$ translates to four products: $A_1 * B_1 = C_1, A_1 * B_2 = C_2, A_2 * B_1 = C_3$ and $A_2 * B_2 = C_4$, i.e.,

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \times \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix}$$

```
multifor (i1 = 0, i2 = 0, i3 = n/2, i4 = n/2;
          i1 < n/2, i2 < n/2, i3 < n, i4 < n;
          i1++, i2++, i3++, i4++;
          1,1,1,1 ; 0,0,0,0 ) {
  multifor (j1 = 0, j2 = n/2, j3 = 0, j4 = n/2;
            j1 < n/2, j2 < n, j3 < n/2, j4 < n;
            j1++, j2++, j3++, j4++;
            1,1,1,1 ; 0,0,0,0 ) {
    0 : c[i1][j1] = 0;
    1 : c[i2][j2] = 0;
    2 : c[i3][j3] = 0;
    3 : c[i4][j4] = 0;
      multifor (k1 = 0, k2 = 0, k3 = 0, k4 = 0;
                k1 < n, k2 < n, k3 < n, k4 < n;
                k1++, k2++, k3++, k4++;
                1,1,1,1 ; 0,0,0,0 ) {
        0 : c[i1][j1] = c[i1][j1] + a[i1][k1] × b[k1][j1];
        1 : c[i2][j2] = c[i2][j2] + a[i2][k2] × b[k2][j2];
        2 : c[i3][j3] = c[i3][j3] + a[i3][k3] × b[k3][j3];
        3 : c[i4][j4] = c[i4][j4] + a[i4][k4] × b[k4][j4];
        }
  }
}
```

**Figure 8: Multifor matrix product code**

The dimension of matrices $C_1$, $C_2$, $C_3$ and $C_4$ is $(n/2 \times n/2)$. These four products might be performed simultaneously and can be naturally expressed using the multifor structure as shown in figure 8.

Geometrically, the multifor iteration domain is a $(n/2 \times n/2 \times n)$ rectangle parallelepiped where each point is associated to four iterations of the four included loop-nests. This execution scheme corresponds to the MapReduce strategy since each for-loop nest computes a $n/2 \times n/2$ sub-block (map step), and the combination of all sub-blocks forms the resulting matrix $C$ (reduce step).

**Steganography:** The fourth example is the decoding phase of a steganography code where an hidden image is extracted from an enclosing one. It is assumed that the upper left pixel of the hidden image is hidden within the upper left pixel of the enclosing image; $HWidth$ and $HHeight$ are the width and the height of the hidden image ; $EWidth$ and $EHeight$ are the width and the height of the enclosing image ; $EImage$ is the image hiding another image ; $HImage$ is the extracted output image that was hidden ; $MImage$ is the output enclosing image hiding no more the image that was hidden in $EImage$. The proposed multifor code version is composed of four simultaneous for-loop nests, the first being dedicated to the extraction of the hidden image, the second to the extraction of the part of the enclosing image which is hiding the hidden image, the third and the fourth being dedicated to copy the pixels directly to the retrieved enclosing image. Since the union of the third and fourth domain is not convex, two loop-nests are necessary to scan it. Notice that we introduce a shortcut in the syntax such that similar loop bodies can be instantiated differently depending on their associated loop-nest. The multifor code is shown in Figure 9 and a view of the enclosing and hidden images, and how the multifor-loop nest scan them is shown in Figure 10. Notice that full parallelism is exhibited with this code.

```
RGBApixel decode_hidden(i, j)
{
 RGBApixel Pixel1 = *EImage(i, j);
 RGBApixel Pixel2;
 Pixel2.Red = Pixel1.Red%2;
 Pixel2.Green = Pixel1.Green%2;
 Pixel2.Blue = Pixel1.Blue%2;
 Pixel2.Alpha = Pixel1.Alpha%2;
 returnPixel2;
}

RGBApixel decode_main(i, j)
{
 RGBApixel Pixel1 = *EImage(i, j);
 RGBApixel Pixel2;
 Pixel2.Red = Pixel1.Red − Pixel1.Red%2;
 Pixel2.Green = Pixel1.Green − Pixel1.Green%2;
 Pixel2.Blue = Pixel1.Blue − Pixel1.Blue%2;
 Pixel2.Alpha = Pixel1.Alpha − Pixel1.Alpha%2;
 returnPixel2;
}

multifor (i1 = 0, i2 = 0; i3 = 0, i4 = HWidth; i1 < HWidth,
          i2 < HWidth, i3 < HWidth, i4 < EWidth;
          i1++, i2++, i3++, i4++; 1,1,1,1; 0,0,0,0)
 multifor (j1 = 0, j2 = 0, j3 = HHeight, j4 = 0; j1 < HHeight,
          j2 < HHeight, j3 < EHeight, j4 < EHeight;
          j1++, j2++, j3++, j4++; 1,1,1,1; 0,0,0,0)
{
 0 : // Retrieve the hidden image
     *HImage(i1, j1) = decode_hidden(i1, j1);
 1 : // Retrieve the enclosing image
     *MImage(i2, j2) = decode_main(i2, j2);
 [2,3] : // Retrieve the enclosing image
     *MImage([i3, i4], [j3, j4]) = *EImage([i3, i4], [j3, j4]);
}
```

**Figure 9: Multifor steganography code for the decoding phase**

**Secret key cryptosystem:** The fifth example is a classic secret key cryptosystem that manipulates binary words. It proceeds by splitting a message $m$ into blocks of constant size. These cryptosystems are characterized by the length of each block, the operating mode and the encryption system of each block. Each cipher mode comprises:

1. Cutting in many blocks $m_1, ..., m_k$ the plain text message $m$;
2. Encrypting the blocks $m_i$ resulting in the encrypted blocks $c_1, ..., c_k$;
3. Concatenating the blocks $c_1, ..., c_k$ to construct the encrypted message $c$.

Each block is encrypted through the product of two cryptosystems $T_1$ and $T_2$. It is classically computed using a loop of the form:

```
for (i = 0; i < k; i++) {
    c[i] = Encrypt(m[i], T1);
    c[i] = Encrypt(c[i], T2); }
```

Suppose the encryption of each block by a given cryptosytem consumes one unit of time. The time required to encrypt the entire message using this loop is $2 \times k$. Let us write this code using a multifor structure:

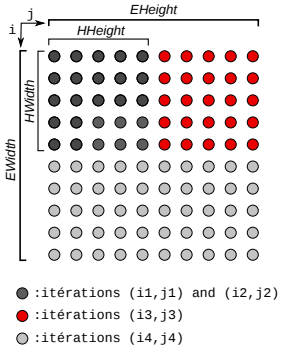



**Figure 10: Enclosing and hidden images scanned by the multifor steganography code**

$$\begin{array}{l}\textbf{multifor } (i_1 = 0, i_2 = 0; i_1 < k, i_2 < k;\\ \qquad i_1++, i_2++; 1, 1; 0, 1) \ \{\\ 0 : c[i_1] = Encrypt(m[i_1], T_1);\\ 1 : c[i_2] = Encrypt(c[i_2], T_2); \}\end{array}$$

This form allows to take advantage of a pipeline scheme where two successive blocks are encrypted in parallel respectively by the cryptosystems $T_1$ and $T_2$. Thus, the time required to encrypt the message is $k + 1$.

# 5. A PROMISING PERSPECTIVE: NON-LINEAR MAPPING

Among the numerous possible extensions, an important one is to map iteration spaces together following a non-linear fashion, such that their shapes has no influence in the mapping. In general, this would allow to execute iterations of any loop nest by any other one of the same trip count, and thus to enlarge significantly the way iterations of different loops can be mapped together. Hence a multifor construct could express quite different computations in a concise way, and augment the number of optimization and parallelization opportunities. Notice that loop nests with different trip counts can also be handled by splitting the largest nest such that one of the resulting nest has the convenient trip count.

Our idea is based on ranking Ehrhart polynomials. It has been shown in previous works dealing with spatial data locality optimization, that it is possible to compute an Ehrhart polynomial associated to a loop nest giving the rank of an iteration [5, 9]. These polynomials have specific properties as being necessarily monotonically increasing according to the lexicographic order of the loop indices, and also defining a bijection between iteration points and the interval of strictly positive integers between one and the total iteration count of the loop nest.

Since ranking Ehrhart polynomials define such bijections, the ability of inverting them would provide a way to retrieve the loop indices corresponding to the rank of an iteration. Hence at any iteration of a loop nest, it would be possible to compute, from the rank, the values of loop indices that would have been reached while running another nest. Thus, any nest could be run by another one, and any iteration space could be mapped onto another one, following the rank of the iterations. Hence this Section deals with the problem of inverting ranking Ehrhart polynomials.

Before proposing a general resolution, we first present a 2-dimensional example.

## 5.1 2-dimensional example

Consider the two loop nests in listings (1) and (2), where $instruction_k(i_1, i_2)$ denotes the instruction block computed at iteration $(i_1, i_2)$. These loops have as respective ranking Ehrhart polynomials:

$$P_1(i, j) = \frac{i(i-1)}{2} + j \text{ and } P_2(i', j') = i'M_2 + j'$$

$$\begin{array}{l}\textbf{for } (i = 1, i < N, i++)\\ \quad \textbf{for } (j = 0, j < i, j++)\\ \qquad instructions_1(i, j);\end{array} \quad (1)$$

$$\begin{array}{l}\textbf{for } (i' = 0, i' < M_1, i'++)\\ \quad \textbf{for } (j' = 0, j' < M_2, j'++)\\ \qquad instructions_2(i', j');\end{array} \quad (2)$$

Taking the assumptions that both nests have the same iteration count and that there is no dependency between $instructions_1$ and $instructions_2$, we could merge the two former loop nests and write (3).

$$\begin{array}{l}\textbf{for } (i' = 0, i' < M_1, i'++)\\ \quad \textbf{for } (j' = 0, j' < M_2, j'++) \ \{\\ \qquad instructions_1(\mathbf{i}, \mathbf{j});\\ \qquad instructions_2(i', j'); \ \}\end{array} \quad (3)$$

However, we need to express indices $(i, j)$ as a function of $(i', j')$ in order to preserve the execution order of the block $instructions_1$. More precisely, for each iteration number $K$ in loop nest (3), we want to execute the $K^{th}$ iteration of loop nest (1). This is why we must invert the ranking Ehrhart polynomial $P_1$, to compute $(i, j) = P_1^{-1}(P_2(i', j'))$.

For any rank $K$, we have to find a couple of indices $(i_0, j_0)$ such that $P_1(i_0, j_0) = K$. The main idea is to cut the 2-dimensional problem into two one-dimensional problems.

Let us define $Q_1(i) = P_1(i, 0) = \frac{i(i-1)}{2}$. As ranking Ehrhart polynomials are (strictly) increasing, the following relation holds:

$$Q_1(i_0) = P_1(i_0, 0) \leq P_1(i_0, j_0) = K \leq P_1(i_0 + 1, 0) = Q_1(i_0 + 1)$$

And, for the same reason, we know that index $i_0$ is unique on $\mathbb{N}_+$. Let us now consider polynomial $Q_1$ as a polynomial over $\mathbb{R}$. By continuity of $Q_1$ over $\mathbb{R}$, there exists $\alpha \in [0, 1[$ such that $Q_1(i_0 + \alpha) = K$. This shows that the equation $Q_1(x) = K$ has at least one real solution. So we have to find $x$ such that:

$$Q_1(x) = K \Leftrightarrow Q_1(x) - K = 0 \Leftrightarrow \frac{x(x-1)}{2} - K = 0$$

Obviously, this last equation has two real roots:

$$x_1 = \frac{1}{2} - \sqrt{\frac{1 + 8K}{4}}, x_2 = \frac{1}{2} + \sqrt{\frac{1 + 8K}{4}}$$

To select the convenient root, we notice that for all $K > 0$, $x_1 \leq 0$. As $i_0 \geq 1$ (according to the loop bounds in (1)), $i_0 + \alpha = x_2$, and thus: $i_0 = \lfloor x_2 \rfloor$. We can now replace $i_0$ by its value in $P_1(i_0, j_0)$:

$$P(i_0, j_0) = \frac{1}{2}\left(\left\lfloor \frac{1}{2} + \sqrt{\frac{1 + 8K}{4}} \right\rfloor\right)\left(\left\lfloor \frac{1}{2} + \sqrt{\frac{1 + 8K}{4}} \right\rfloor - 1\right) + j_0$$

and finally deduce $j_0$:

$$j_0 = K - \frac{1}{2}\left(\left\lfloor \frac{1}{2} + \sqrt{\frac{1 + 8K}{4}} \right\rfloor\right)\left(\left\lfloor \frac{1}{2} + \sqrt{\frac{1 + 8K}{4}} \right\rfloor - 1\right)$$

The resulting code is shown in listing 4.

$$
\begin{array}{l}
K = 0; \\
\textbf{for } (i' = 0, i' < M_1, i'++) \\
\quad \textbf{for } (j' = 0, j' < M_2, j'++) \ \{ \\
\qquad K++; \\
\qquad i = floor(sqrt((1 + 8 * K)/4) + 1/2); \\
\qquad j = K - i * (i-1)/2; \\
\qquad instructions_1(i, j); \\
\qquad instructions_2(i', j'); \ \}
\end{array} \tag{4}
$$

We now present the general case, which can be easily deduced from the 2-dimensional case.

## 5.2 General case

Without any loss of generality, we assume all loop indices lower bounds equal 0. We consider the $N$-dimensional ranking Ehrhart polynomial $P(i, j, k, ...)$, and for each K, we seek the tuple $(i_0, j_0, k_0, ...)$ such that $P(i_0, j_0, k_0, ...) = K$.

Similarly to the previous example, we start with the outermost loop index $i_0$. We define $Q_i(i) = P(i, 0, 0, ..., 0)$ and solve $Q_i(x) - K = 0$. Here is the only issue that differs from the 2D case: as we can't state that $Q_i$ is monotonically increasing on $\mathbb{R}_+$, we have to find a criterion to select the root giving the sought index.

First, we obviously eliminate complex and negative solutions, since $i_0 \in \mathbb{N}_+$, and consider $n \leq N$ positive real roots $\{x_1, .., x_n\}$. As we know that $i_0$ is unique, a way to select the convenient root is to check which $x \in \{x_1, .., x_n\}$ satisfies :

$$Q_i(\lfloor x \rfloor = i_0) \leq Q_i(x) \leq Q_i(\lceil x \rceil = i_0 + 1)$$

However, this strategy is only applicable at runtime, and may add non negligible time overhead. A compile-time solution is to check if $Q_i$ is monotonically increasing on $\mathbb{R}_+$ by examining its derivative. If so, any root in $\{x_1, .., x_n\}$ is suitable.

Once $i_0$ has been found, the process starts again with $Q_j(j) = P(i_0, j, 0, ..., 0)$, and so on until all indices have been computed.

# 6. CONCLUSION

We have proposed a new programming control structure called "multifor" and showed that it allows the polytope model to handle programming models that were not attainable directly before. Important related theoretical studies have still to be conducted, as dependency analysis between the for-loops composing a multifor-loop, or optimizing code transformations that considers interactions between the for-loops.

Many interesting extensions can also be studied as making header parameters, or instructions, dependent of several for-loop indices composing the same multifor-loop level, or defining non-invariant grains and offsets, or introducing conditionals on the effective run of the for-loops, etc. In this paper, we showed that it may be possible to handle non-linear mapping of iteration spaces using inverted ranking Ehrhart polynomials.

The multifor structure can also be used as a representation model for some interacting mechanisms, as concurrent memory accesses, as it is done for sequential codes in [7].

We are planning to implement multifor structures in the Clang/LLVM compiler as an extension to C/C++.

# 7. REFERENCES


[1] U. Banerjee. *Loop Transformations for Restructuring Compilers - The Foundations*. Kluwer Academic Publishers, 1993. ISBN 0-7923-9318-X.

[2] O. A. R. Board. Openmp application program interface, version 3.1, 2011.

[3] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, Sept. 2008.

[4] I. Christadler, G. Erbacci, and A. D. Simpson. Facing the multicore-challenge ii. chapter Performance and productivity of new programming languages, pages 24–35. Springer-Verlag, Berlin, Heidelberg, 2012.

[5] P. Clauss and B. Meister. Automatic memory layout transformations to optimize spatial locality in parameterized loop nests. *SIGARCH Comput. Archit. News*, 28(1):11–19, Mar. 2000.

[6] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proc. of the 22nd annual symp. on Principles of distributed computing*, PODC '03, pages 92–101. ACM, 2003.

[7] A. Ketterlin and P. Clauss. Prediction and trace compression of data access addresses through nested loop recognition. In *6th annual IEEE/ACM int. symp. on Code generation and optimization*, pages 94–103, Boston, United States, Apr. 2008. ACM.

[8] C. E. Leiserson. The cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 522–527, New York, NY, USA, 2009. ACM.

[9] V. Loechner, B. Meister, and P. Clauss. Precise data locality optimization of nested loops. *J. Supercomput.*, 21(1):37–76, Jan. 2002.

[10] S. Marlow, P. Maier, H.-W. Loidl, M. K. Aswad, and P. Trinder. Seq no more: Better strategies for parallel haskell. In *Proceedings of the 3rd ACM SIGPLAN symposium on Haskell*, pages 91–102, Baltimore, MD, United States, Sept. 2010. ACM Press.

[11] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Series. Artima Press, 2011.

[12] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop transformations: convexity, pruning and optimization. In *Proc. of the 38th annual ACM SIGPLAN-SIGACT symp. on Principles of programming languages*, POPL '11, pages 549–562, New York, NY, USA, 2011. ACM.

[13] K. F. Sagonas. Using static analysis to detect type errors and concurrency defects in erlang programs. In *FLOPS*, pages 13–18, 2010.

[14] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proc. of the 11th ACM SIGPLAN symp. on Principles and practice of parallel programming*, PPoPP '06, pages 187–197, New York, NY, USA, 2006. ACM.

[15] N. Shavit and D. Touitou. Software transactional memory. In *Proc. of the 14th annual ACM symp. on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[16] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.