ISSN 2277-3061

Loop Structures Optimization and Reordering for Efficient Parallel Processing

Monica-Iuliana CIACA, Loredana MOCEAN, Alexandru VANCEA, Mihai AVORNICULUI
Babeș-Bolyai University, monica.ciaca@econ.ubbcluj.ro
Babeș-Bolyai University, loredana.mocean@econ.ubbcluj.ro
Babeș-Bolyai University alexandru.vancea@cs.ubbcluj.ro
Babeș-Bolyai University mihai.avorniclului@econ.ubbcluj.ro

ABSTRACT

The problem of choosing an optimal sequence of transformations, leading to the most efficient parallel version of a program remains an open question. Related to this, compilers of the moment only manage to incorporate a set of heuristic decisions. This article treats the transformation of the program, addressing and analyzing the range of transformations of loop structures, that we consider most appropriate today. We tried to exemplify these transformations in case of groups of companies.

Keywords

Group of Companies; Loop Structures; Parallel Computing

Academic Discipline And Sub-Disciplines

Mathematics applied in Computer Science

SUBJECT CLASSIFICATION

JEL Classification: C02, C63

SUBJECT CLASSIFICATION

37N40

TYPE (METHOD/APPROACH)

Quasi-Experimental; Literary

ISSN 2277-3061

INTRODUCTION

With a rather complicated organizational structure, characterized by behavioral flexibility and lack of bureaucracy present in all sectors of industry, commerce and services, groups of companies easily adapts to changing economic and social conditions. In the idea of implementation of new market information and communication technologies, this article proposes a prototype for decomposing existing operations in groups of companies using parallel computing.

During the development of compiler theory, several changes of source code had been proposed to optimize the execution of programs. Most optimizations for sequential cases intend to reduce the number of instructions executed by the program using transformations, based on a quantitative analysis of the values conveyed in the program and data flow analysis. In addition, recent optimization for parallel execution maximizes parallelism and data localization in the memory, using transformations based on the characterization of arrays and data dependency analysis results.

The stages which must be completed by a compiler to perform optimizations are the following:

a) Selection of the part of the source program which shall be optimized and the appropriate processing of a particular purpose;

b) Checking if the semantic transformation ensures consistency;

c) Transformation of the program.

Techniques for data dependency analysis are used for steps a) and b). The selection stage is the most difficult and insufficiently treated topic in the current compiler theory. Due to high costs involved in a full analysis of optimization possibilities, compilers typically restrict their range of action to some transformations considered more efficient by their builders. On the other hand, there may be sequences of transformations that have the opposite effect. For example, an attempt to reduce the number of instructions executed may ultimately reduce performance because of improper use of caching. Architectures become more complex, because of significantly increasing optimization directions and decision making related to the range of transformations is very complicated.

1.Transformations for Execution Optimizing

1.1 Operators Reduction

Reduction of operators aims for replacing a loop expression with an equivalent expression that uses a less expensive operator ([4], [6]). Based on the following loop structure which containing a multiplicity,

```
for  i := 1 to n do
        a[i] := a[i] + c∗i;
end for
```

we can obtain through operator reduction, a transformed version of the loop, in which the multiplication has been replaced by addition.

```
T := c;
for i := 1 to n do
        a[i] := a[i] + T;
        T := T + c;
end for
```

Even if, most of the time, reducing the operators is accompanied by the introduction of an additional variable, time saving is achieved by the loop processing significantly. It is justified to put this transformation in the category of optimization execution. The most common use of operator reduction is the reduction of expressions that contain induction variables ([3],[4],[6]). Table 1 presents various possibilities of reduction of operators. It is assumed that the operation in the first column occurs in a loop with index i from 1 to n at the time of processing and the compiler initializes a temporary variable T in the expression of the second column. Operation inside the loop is replaced by the expression in the third column, and the value of T is updated every iteration with the value in the fourth column. The positive effects are obvious.

Table 1. Operatories reduction (*c* looping invariant; *x* can vary between iterations)

Expression	Initializing	Using	Updating
$c * i$	$T = c$	T	$T = T + c$
c^i	$T = c$	T	$T = T * c$
$(-1)^i$	$T = -1$	T	$T = -T$
x / c	$T = 1/c$	$x * T$	

ISSN 2277-3061

1.2 The Elimination of Induction Variable

A variable whose value is derived from the number of iterations that were executed by a loop is called induction variable. The control variable of a "for" loop is the most common type of induction variable, but there are other variables with this property. The example below illustrates this in case of induction variable j. Induction variable elimination simplifies the analysis of index expressions in the data dependency tests, as it is explained in the example below, where, after removing the variable j, the analysis is based only on the values of the loop variable i and constant n.

```
j := n;
for i := 0 to n do
      a[i] := b[j-1];
      b[j] := c[i];
      j := j - 1;
end for
```

⇒

```
for i := 0 to n do
      a[i] := b[n-i-1];
      b[n-i] := c[i];
end for
```

1.3 Factorization of Loop Invariants

When an operation occurs inside a loop, but its result is not changed between iterations (loop invariant), the compiler can transfer that computation outside the loop ([3]). We give below a code sequence in which a transcendental function of an expensive call is transferred outside the loop.

for i := 1 to n do

a[i] := a[i] + sqrt(x);

end for

The test that appears in the transformed code ensures that if the loop is not executed again then the transfer code is not run either, to prevent triggering an exception.

```
if (n > 0) then C := sqrt(x);
for i := 1 to n do
      a[i] := a[i] + C;
end for
```

1.4 Externalization of Conditional Instructions

This method is applied to loops that contain a conditional instruction with invariant test in the loop. The loop is then replicated at each conditional branch instruction, thus avoiding the disadvantage of conditional branching within the loop, reducing code size representing the loop body and making possible the parallelization of a possible conditional branch instruction as Allen remarks [5].

Conditional instructions that are subject to outsourcing can be detected while analyzing the possibilities of factoring, a process that identifies loop invariants.

In the following example the variable X is loop invariant, allowing the loop to be subjected to the operation of outsourcing and the true branch to be executed in parallel, as shown in the converted code. Notice that, like the factorization of loop invariants, if there's a chance to trigger an exception condition assessment, this should be prevented by a test of the possibility of execution.

```
for i := 2 to n do
  a[i] := a[i] + c;
   if (x < 7) then
    b[i] := a[i] * c[i]
   else
    b[i] := a[i-1] * b[i-1];
   end if
end for
```

(a) initial loop

```
if (n > 1) then
  if (x < 7) then
     for i := 2 to n
       a[i] := a[i] + c;
       b[i] := a[i] * c[i];
     end for
  else
     for i := 2 to n do
       a[i] := a[i] + c;
       b[i] := a[i-1] * b[i-1];
     end for
  end if
end if
```

(b) after externalization

Fig. 1. Externalization of conditional instructions internal to the loop (loop un-switching).

ISSN 2277-3061

In a double nested cycle in which the inner loop has unknown limitations, if the code is generated directly, there will be a test before the inner body of the loop, to determine whether or not it will be executed. The test for the inner loop will be repeated each time the outer loop is executed. When the compiler uses an intermediate representation for the program, then test is explicit and outsourcing can be used to transfer this test outside the outer loop [27].

2. Iteration Reordering Transformations

In this section we describe the transformations that alter the relative order of execution of the iterations of nesting cycles. These changes are mainly used to highlight the opportunities for parallelization and locating data in memory. Some compilers use reordering transformations only for perfectly nested cycles. To increase opportunities for optimization, compilers can sometimes use fission to extract perfectly nested cycles of imperfect nesting. The compiler determines whether a loop can be executed in parallel, examining the associated dependencies induced by loop iterations. If all of the loop dependency distances are 0, this means that there is dependency carried over iterations in the loop.

We give below an example where the loop distance vector is (0,1), this way the outer loop may be parallelized (figure 2).

More generally, the p-th loop of a a nested structure of cycles may be parallelized for any distance vector V = (v1, …, vp, …, vd), vp = 0 ∨ ∃q < p : vq > 0

In the case of (b) the distance vectors are {(1,0), (1, -1)}, so that the inner loop may be parallelized. Both references from the right part of the expression accesses in reading items on line i-1 of the a array, elements updated in the previous iteration of the outer loop. The i-th line items may be calculated and stored in any order.

```
for i := 1 to n do                    for i := 1 to n do
 for j := 2 to n do                    for j := 1 to n do
    a[i, j] := a[i, j-1] + c;             a[i, j] := a[i-1, j] + a[i-1, j+1];
 end for                               end for
end for                               end for
```

(a) outer loop may be parallelized (b) inner loop may be parallelized

Fig. 2. Terms dependency loop parallelization.

2.1 Loops Interchange

This transformation changes the position of the two loops of a PNL (Perfectly Nested Loop), moving usually one of the outer loops innermost position ([7],[34]). Interchange is considered one of the most powerful transformation and can improve performance in many ways. It is used mainly to:

- allow vectorising by exchanging an interior loop that manifests dependencies, with one exterior loop, independently;
- improve vectorization, loop by shifting largest independent position within the loop;
- improve the performance of parallel execution by transferring an outside independent cycle nested loop, thus increase the granularity and reduce the number of iterations required barrier synchronizations;
- to reduce looping step, preferably to 1;
- increase the number of expressions in the loop invariant cycle innermost.

We should consider that these benefits do not exclude each other. For example, a swap which improves the reuse grade of the registers can modify an access pattern with step 1 into an access patern with step n, which may have a much lower overall performance, due to a much larger number of mismatches in the memory cache. In the following example, the inner loop accesses array a with n steps. We use the convention of storing the array elements in columns. With loop exchanging, we convert inner cycle into a cycle where accessing step = 1.

```
for i := 1 to n                       for j := 1 to n
  for j := 1 to n                       for i := 1 to n
   total[i] := total[i] + a[i,j];        total[i] := total[i] + a[i,j];
  end for                               end for
end for                               end for
```

(a) embedding the original (b) PNL after interchange

Fig. 3. Loops interchange.

ISSN 2277-3061

For a large array, for which more than one column fits in the cache memory, the optimization reduces the number of cache misses for a, from n2 to n2 ∗ de / dl, where de is the dimension of the element and dl the dimension of the line.

Anyway, the original loop permits total[i] to be placed in a register, eliminating the load/store operations from inner loop.

This way, the optimized version increases the number of operation load/store for total from 2n to 2n2. If the array a fits in cache, it proves that the original loop is more advantageous. For a vectorial architecture, the transformed loop allows vectorization by eliminating dependency on total [i] in the inner loop.

Interchange of the cycles is legal when changed dependencies are legal and looping limits can be interchanged. If two loops, p and q, from a PNL of d loops, are interchanged, each dependency vector V = (v1 ,…, vp ,…, vq ,…, vd) from the original nested loop becomes V' = (v1 ,…,vq ,…, vp ,…, vd) in the transformed nested loop

If V' is lexicographically positive, then the dependency relationships of the original loop remain good.

A double nested loop interchange may be applied only if it has a dependency vector like (< , >). The figure 4(a) represents the nested loop with dependency (1, -1), which leads to iteration dependencies presented in figure 4(b). The order of iterations performed is indicated by the dotted line. The order of traversal after interchange is shown in figure 4(c): some iterations are executed prior dependent iterations, so interchange is illegal.

```
for i := 2 to n do
      for j := 1 to n-1 do
         a[i,j] := a[i-1,j+1];
      end for
end for
```

(a)

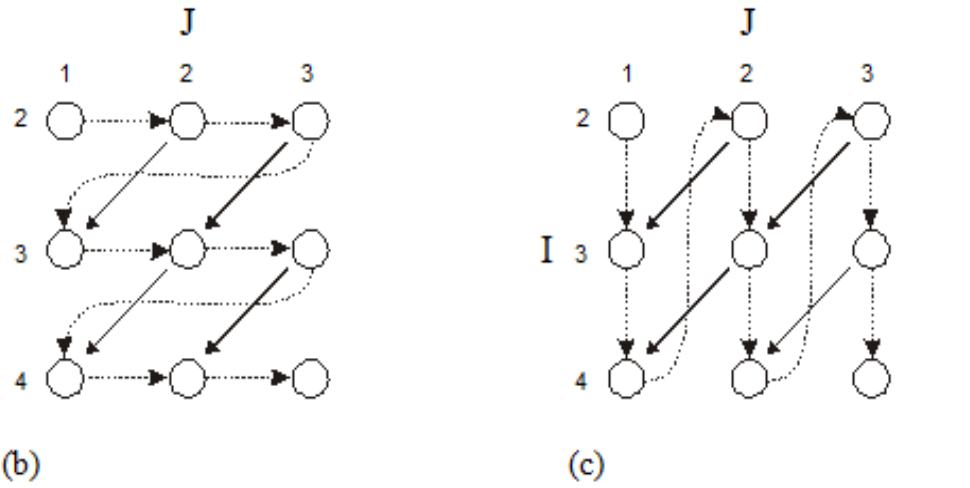


Fig. 4. The original structure(a);Original order of the scroll(b); Order of the scroll after interchange(c).

The looping interchange of the limits is a simple operation when the iteration space is rectangular, as in the previous PNL example.

In this case the limit cycles are independent of indices inside the loop containing it, and the two can simply be interchanged. When the iteration space is not rectangular, calculation of new limit of looping becomes more complex. In programming often triangular spaces and even trapezoidal are used. Cycles often occur imperfectly nested whose management requires more complex techniques. Some of these issues are addressed in detail in ([36],[39]).

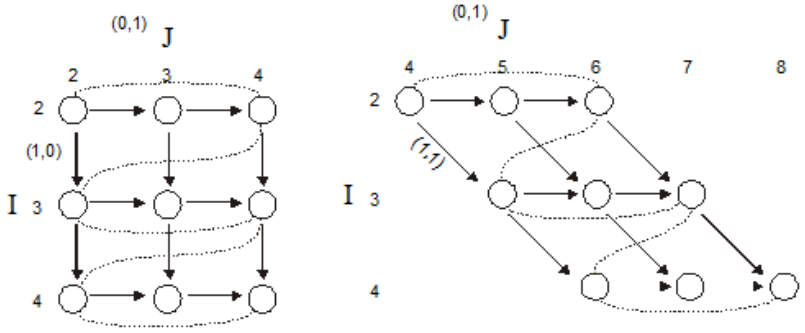
2.2 The Interior Cycle Translation

Translation of interior cycle (loop skewing) is a transformation especially useful in combination with interchanged cycles ([22], [26], [39]). Translation has been introduced to solve the so-called calculation type: "wave crest" (wave front computations). It is called like this because it updates array elements as a wave propagates through space iterations.

ISSN 2277-3061

```
for i := 2 to n-1 do
 for j :=2 to m-1 do
  a[i,j] := (a[i-1,j] + a[i,j-1] + a[i+1,j] + a[i,j+1])/4;
 end for
end for
```

(a) Original source code: the dependencies are {(1,0),(0,1)}



(b) original space (c) translated space

```
for i := 2 to n-1 do
 for j := i+2 to i+m-1 do
  a[i,j-i] := (a[i-1,j-i] + a[i,j-1-i] + a[i+1,j-i] + a[i,j+1-i])/4;
  end for
end for
```

(d) translated code : dependences are {(1,1),(0,1)}

```
for j := 4 to m+n-2 do
 forall i := max(2,j-m+1) to min(n-1,j-2) do
   a[i,j-i] := (a[i-1,j-i] + a[i,j-1-i] + a[i+1,j-i] + a[i,j+1-i])/4;
 end forall
end for
```

(e) translated code and interchanged : dependences are {(1,0)(1,1)}

Fig. 5. The loops translations

Figure 5 (a) shows a typical calculation of wave peak. Each element is calculated by averaging the four nearest neighbors. Although the cycles are not parallelizable in their original form, each diagonal of the array ('wave height') can be calculated in parallel with each other.

Area iterations and dependencies are shown in figure 5 (b), the dotted lines indicate the peaks of the wave.

Translation is performed by adding the outer loop index multiplied by a factor of translation, f, and variable limits inner iteration, followed by reduction using the same values for each variable in the inner iteration cycle. Since looping limits change accordingly and use index variables to compensate, translation does not change the semantics of a transformation program and is always legal.

The cycle from figure 5 (a) may be subject to interchangeability, but the loop can not be parallelized, because of a dependency on both inner loops, (0,1) and in the outer (1.0). This graph is expressed by the existence of edges on the horizontal ((0,1)) and vertical ((1,0)).

The result of the translation by f = 1 is shown in figure 5 (c-d). Transformed code is equivalent to the original, but the effect on space iterations aligning "wave peaks" (diagonals) original nesting cycle (that is diagonal from right to left are vertical lines), so that for a given value of j all i iterations can be executed in parallel (because there is no vertical dependency arcs, iterations for a fixed j don't depend on one another).

To highlight this parallelism, loop structure must also be translated to subject interchangeability. After translation and interchange, the cycle has a nested distance vectors {(1,0), (1,1)}. The first dependency allows the inner loop to be parallelized, because the corresponding dependency distance is 0. The second dependency allows the inner loop to be parallelized because it is a dependency in report to previous iterations of the outer loop.

Translating can highlight parallelism for a nesting of two cycles with the set of distance vectors (Vk) if:

$(\exists Vi = (vi1, vi2) : vi1 = 0 \wedge vi2 > 0) \wedge (\exists Vj = (vj1, vj2) : vj1 > 0 \wedge vj2 \leq 0)$

When we translate with factor f, the originally distance vector (v1, v2) will become (v1, fv1 + v2). For any dependency with $v2 \leq 0$, the scope is to find f so that $fv1 + v2 \geq 1$. Correct translation of factor f is calculated by taking the

maximum of fi = ⌈ (1 – vi2) / vi1⌉ in relation to all the dependencies (Kennedy 1993). The interchanging of translated loop is complicated because their looping limits depend on the loop iteration variables.

For two loops with limits i1 = l1, u1 and i2 = l2, u2 where l2 and u2 are expressions independent by i1, the interior translated loop has limits i2 = fi1 + l2, fi1 + u2.

After interchange, the limits are:

for i2 := fl1 + l2 to fu1 + u2 do

for i1 := max (l1 , ⌈(i2 – u2)/f⌉) to min (u1 , ⌈(i2 – l2)/f⌉) do

…..

An alternative method for treating calculations of wave peak is super-node partitioning (Irigoin 1988).

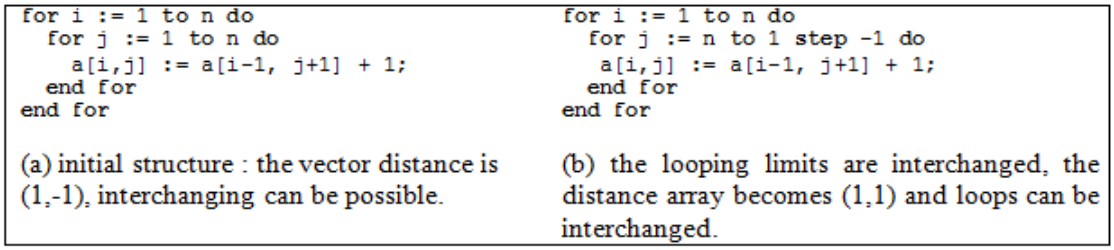
2.3 Reversing the Looping Limits

This transformation changes the direction of the cycle space through its iterations. It is often used in combination with other reordering transformations of space iterations, because it changes depending vectors. As independent optimization, reverse looping can reduce loop overhead by eliminating the need for a comparison instruction on architectures without a comparison-branching instruction such as Alpha [32].

The cycle is reversed so that the iteration variable decreases to zero, allowing the loop to end an instruction of type BNEZ (branch if not equal zero). If loop p from a nesting of d loops is inverted, then for each dependency vector V, the element vp is denied. Reversal is legal if each vector result V' is lexicographically positive: if vp = 0 or ∃ q < p: vq > 0.

For example, the inner loop of a nested steering vector {(<, =), (<,>)} can be reverted, because resulted dependencies are still positive lexicographically.

Figure 6 shows how the reversal can be possible in swap cycles. (a) has the distance vector (1, -1), which prevents the interchange because the vector distance (-1, 1) is not positive lexicographically, which indicates that the swapping would change the order of execution of dependent instructions. Nested reverse cycle (b) may be legally interchanged.



```
for i := 1 to n do
  for j := 1 to n do
    a[i,j] := a[i-1, j+1] + 1;
  end for
end for
```

(a) initial structure : the vector distance is (1,-1), interchanging can be possible.

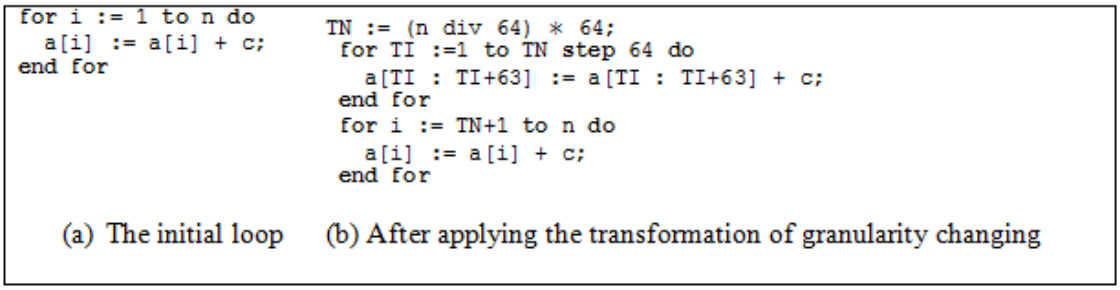
```
for i := 1 to n do
  for j := n to 1 step -1 do
   a[i,j] := a[i-1, j+1] + 1;
  end for
end for
```

(b) the looping limits are interchanged, the distance array becomes (1,1) and loops can be interchanged.

Fig. 6. The interchange of looping limits.

2.4 Changing the Cycle Granularity

Changing the granularity of a cycle (strip mining) is a method for adjusting the granularity of an operation, especially a parallelized operation ([2],[7],[24]). The original definition of this operation involves transforming a one-dimensional cycle to two-dimensional cycles. A dependency on (d) becomes (0, d) and (1, d - s - 1), where S is the step value access (strip size). The transformation is always legal in the sense that it will induce negative dependencies in the transformed loops. But it is justified only if S ≥ d, otherwise it has no positive effect. Changing the granularity is usually performed for the execution on vectorial machines, to make an efficient exploitation of the size of the machine registers. We present an example below.



```
for i := 1 to n do
  a[i] := a[i] + c;
end for
```

(a) The initial loop

```
TN := (n div 64) * 64;
for TI :=1 to TN step 64 do
  a[TI : TI+63] := a[TI : TI+63] + c;
end for
for i := TN+1 to n do
  a[i] := a[i] + c;
end for
```

(b) After applying the transformation of granularity changing

Fig. 7. Granularity changing of a loop structure

ISSN 2277-3061

Calculation with changed granularity is expressed in matrix notation and it is equivalent to a forall loop. If the iteration's length is not divisible by the step size, then additional changes are needed. For this purpose we use a so-called cleanup code [7]- as in the case for the last loop from figure 7 (b).

One of the most common uses of granularity change is choosing the number of independent calculations in the inner cycle of a nested loop structure. For example, in a vectorial machine, the serial cycle can be converted to a series of operations on arrays, each array consisting only of the unit of granularity.

Changing the granularity is also used for some compilations of type SIMD [34] to combine operations in a loop and send on distributed memory multiprocessors [13] and temporarily limit the size of pictures generated by the compiler ([1],[39]).

Changing granularity often requires other changes. Cycle decomposition may reveal simple cycles nested within a cycle that is too complex to undergo to operation of changing granularity. Interchanging of cycles can be used to move a parallelized loop in the inner position or nested cycle, to maximize granularity unit size.

The above examples demonstrate that the granularity changing can create a bigger processing unit, from smaller ones.

Transformation can also be used in the opposite direction, reducing the initial granularity, if execution efficiency is necessary.

2.5 Shrinkage of Loop

The contraction of a loop (cycle shrinking) is a special case of changing granularity. When a cycle displays dependencies which cannot be executed in parallel (i.e. to be converted into a forall statement), the compiler can still detect a certain degree of parallelism possible that the distance dependency is greater than 1.

In this case, the contraction will convert a serial cycle into an external serial cycle and internal parallel cycle [28]. Contraction cycle is especially used to highlight fine granularity parallelism.

For example, in figure 8 (a), a [i + k] is updated in iteration i, and accessed in reading in the iteration i + k, depending on the distance k. As a result, the first k iterations can be executed in parallel only with the condition that none of the following iterations begin the execution until the first k were not finished. The same thing is then carried out with the following k iterations, as shown in figure 8 (b). Space iteration dependencies are shown in figure 8 (c): each group of k iterations is thus dependent only from the previous group.

```
        for i :=1 to n do
S1:         a[i+k] := b[i];
S2:         b[i+k] := a[i] + c[i];
        end for
```

(a) due to the update of a, $S_1 \xrightarrow{(k)} S_2$; due to the update of b, $S_2 \xrightarrow{(k)} S_1$.

```
        for TI :=1 to n  step k do
              forall i := TI, TI + k - 1 do
S1:                 a[i + k] := b[i];
S2:                 b[i + k] := a[i] + c[i];
              end forall
        end for
```

(b) k iterations can be executed in parallel because this is the minimal distance for depedency.

(c) the iteration space for $n = 6$ and $k = 2$.

Fig. 8. Space iteration dependencies

The result is, potentially, an increasing of the speed with k factor, but this k value is usually small (2 or 3). So, this optimization is typically limited, to highlight the parallelism which can be made at the instruction level, for example, by carrying out processing cycles. Note that the value of k must be constant in the cycle, and the compiler must know at least that it is positive.

2.6 Dividing Iteration Space

Dividing (loop tiling) is a multidimensional generalization transformation amending granularity. Dividing is primarily used to improve the reuse of the cache, dividing the iteration space into so-called divisions (tiles) and transforming nested cycle to iterate over them ([2],[12], [21], [39]). Also, the transformation can be used to improve the location of the data to the CPU, registers or memory pages.

```
for i:=1 to n do
  for j:=1 to n do
   a[i,j]:=b[j,i];
  end for
end for
```

(a) original loop;

```
for TI:=1 to n step 64 do
  for TJ:=1 to n step 64 do
    for i:=TI to min(TI + 63,n) do
      for j:=TJ to min(TJ+63,n) do
       a[i,j]:=b[j,i];
      end for
    end for
  end for
end for
```

(b) the loop after the iteration space division;

Fig. 9. The division of iteration space

The need of using this transformation is illustrated in the loop from figure 9 (a) that assigns to a, the transpose of b. The j loop is the most interior, the access to b is made with step 1, while the access to a is with n step.

The interchange is not helpful because it accesses b with n steps. Iterating over divisions (tiles) of space iterations, as it is shown in figure 9 (b), the cycle uses each line of the cache. The two inner cycles of matrix multiplications also have such a structure, the division being necessary to obtain runtime efficiency in dense matrix multiplication.

A pair of adjacent cycles can be divided if it can be legally interchanged. After division, the outer pair of cycles can be shifted to improve data localization at the division level and inner cycles can be interchanged to exploit parallelism and data locality cycle at the registry level.

Dividing can be expressed as an increase of the granularity of a single iteration of the collections of iterations (this collection actually represents division), outer looping having the mission to scroll divisions and the inner are responsible for correctly completing iterations in a division.

2.7 The Looping Decomposition

The decomposition (also called fission cycle - loop distribution, loop fission or splitting) divides a loop structure in several ones. Each new iteration loop has the same space as the original, but contains only a subset of its instructions ([20], [26]).

The decomposition is used to:

• create perfectly nested cycles;

• create sub-cycles with fewer dependencies;

• improve instruction cache allocation due to lower dimensions of cycles;

• reduce memory requirements, iterating over fewer arrays;

• increase the reuse grade of registers.

Figure 10 is an example in which decomposition removes dependencies and allows parts of a cycle to be executed at the same time.

```
for i := 1 to n do
  a[i] := a[i] + c;
  x[i + 1]:=x[i] * 7 + x[i + 1] + a[i];
end for
```

(a) original loop;

```
forall i := 1 to n do
     a[i] := a[i] + c;
end forall
for i := 1 to n do
     x[i + 1]:=x[i] * 7 + x[i + 1] +
a[i];
end for
```

(b) the loop after making of decomposition

Fig.10. Splitting a looping structure.

The decomposition can be applied to any cycle, but all the instructions which belong to a cycle of dependency (called block π , [20]) should be placed in the same loop, and if S1 precedes S2 in the original loop, the loop containing S1 must also precede the one that contains the statement S2. If the loop contains a control flow execution, conversion application can show opportunities of decomposition. An alternative is to use a control flow graph of dependencies [19].

A specialized version of these transformations is the so-called decomposition by name, first called horizontal decomposition partitions by name [2]. Instead of a comprehensive analysis of data dependencies, the loop instructions are partitioned into mutually exclusive sets accessing variables. To those instructions is guaranteed their independence. When

the arrays are large, the decomposition by the name may increase the amount of localization of data in the cache memory. Note that the above loop can't be decomposed using fission by name because the same instruction accesses the array a.

2.8 The Fusion of Loops

Reverse transformation of the decomposition is the fusion can improve performance by:

• reducing delays due to overhead looping (loop overhead);

• increase the level of instruction parallelism;

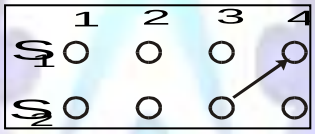
• improving data localization at the level of registries, cache or memory pages [1];

• improving the load balance for parallel execution cycles.

In figure 10, decomposition allows partial parallelization of the cycle. The merging of the two loops improve the location registers and cache, because a[i] does not have to be loaded only once. The fusion also increases the degree of instruction-level parallelism by increasing the ratio of floating-point operations and integer values in the loop structure and reduces the overhead of the second cycle's time. If n is large, the split-cycle to run faster would be a vector machine, while fused cycle should be less in a superscalar machine.

In order to be able to merge two cycles, they must have the same limits. If the limits are not identical, it is sometimes possible to do the same through their adjustment (suggestive technique called a loop peeling) or by introducing conditional expressions in the loop body.

Two loops with the same limits can be merged if there aren't two instructions, S1in the first loop and S2 in the second loop, so that they would have a dependency S2 → S1 with the direction < in merged loop. The reason why this would be incorrect is that before the merge, all instances of S1 are executed before any instance of S2. After the merge, the corresponding instances are executed together. If there is an instance of S1 that has a dependency that must be executed



after an instance of S2, the merger changes the order of execution, as it is shown in figure 11.

Fig. 11. Two loops containing S1 şi S2 cannot merge if S2 → S1 in the looping structure obtained after merge.

3. Case Study

Let us consider the group of companies G which have a mother-firm and n subsidiaries named S1, S2, …, Sn. The reducing of operators is applied in the stage of aggregation of accounts. Mother-firm cumulates all the accounts like in figure 12 (multiplication operation and its transformation in the addition operation).

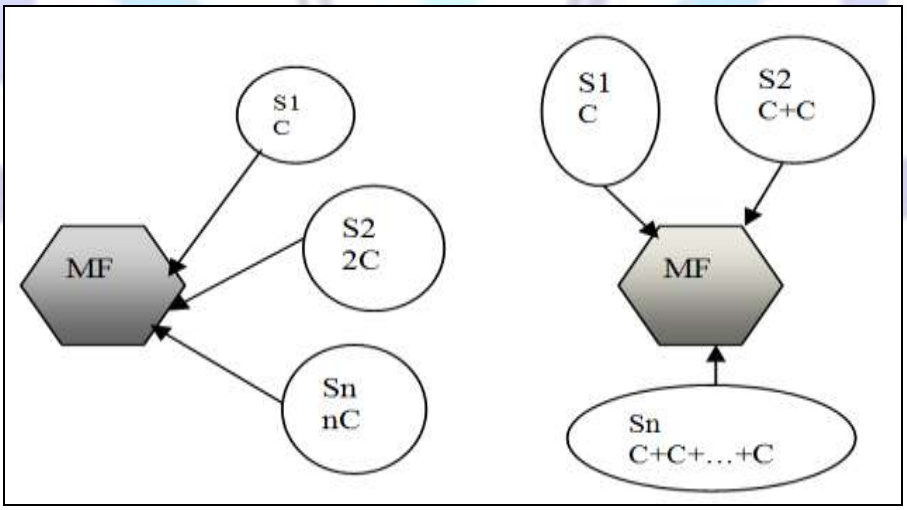


Fig. 12. The reduction of operations in mother-firm

In the same group we can reduce the variable of induction like in the next figure. We start with 2 variables, i and j, we reduce j and we finally have only i.

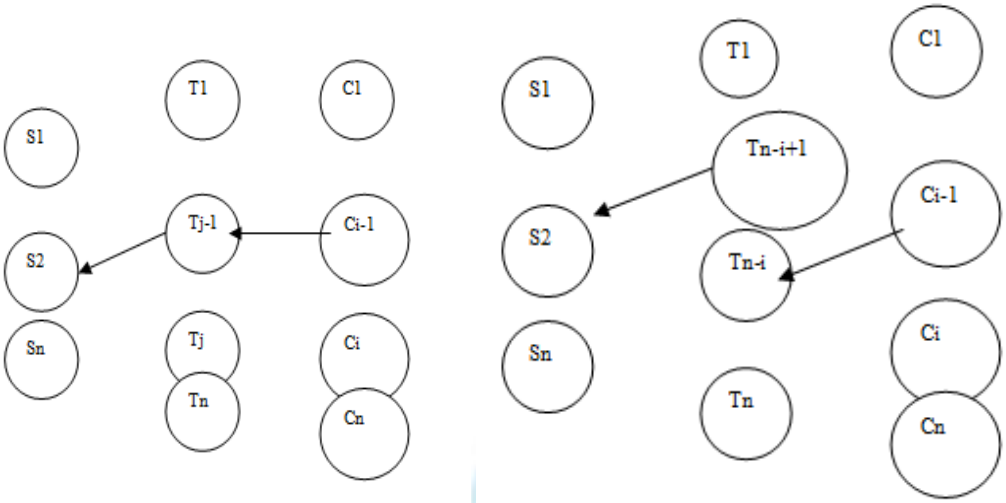


Fig. 13. Elimination of the loop variable j

In the stage of consolidation of accounts of the group of companies, we can reduce the number of variables by eliminating the loop variable j. The operations of the group companies will be reduced to the elimination of mutual accounts, eliminating mutual operations and eliminating reciprocical results using only one variable loop.

In the stage of factorizations of loop invariants, the incomes of subsidiaries which are reflected in mother–firm can be reflected like in Figure 14.

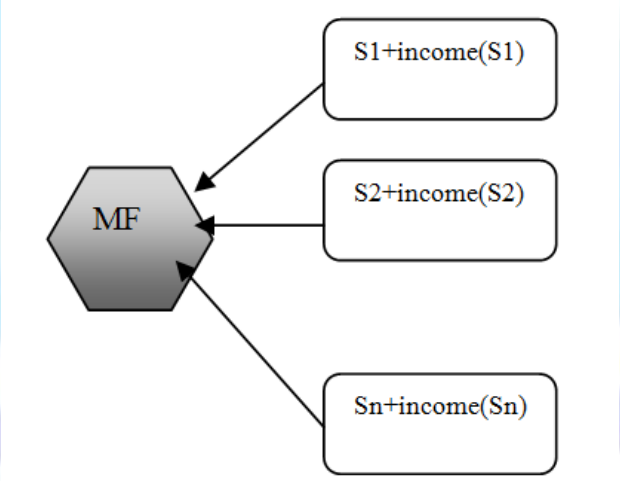


Fig. 14. The incomes of subsidiaries

Income (MF) = $\sum_{i=1}^{n}(Si + Income(Si)$)

Instead of those operations, we can perform a global operation in which we add all the incomes, after we can calculate the income of mother-firm.

Income = Income(S1) + Income(S2) + … + Income(Sn) $= \sum_{i=1}^{n} Income(Si)$

and then, Income (MF):= Income + $\sum_{i=1}^{n} Si$

For determining the consolidation parameter, we use an initial logical schema, see figure 15.

ISSN 2277-3061

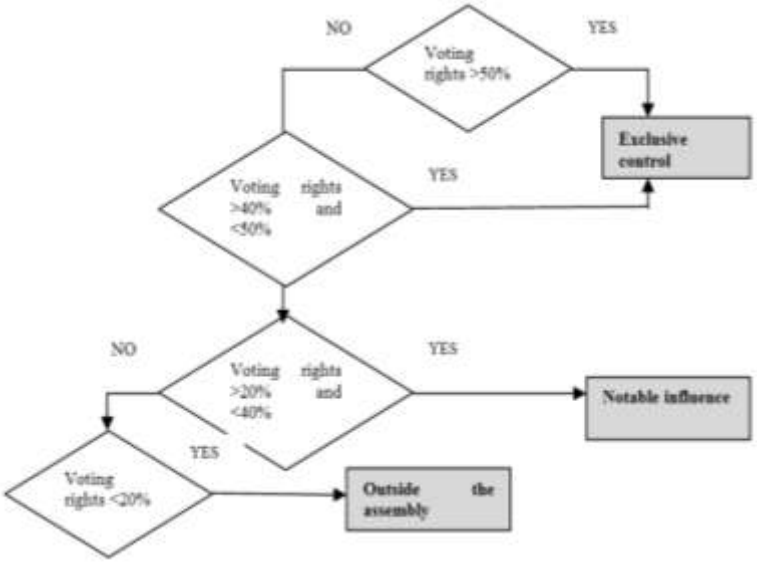


Fig. 15. Determination of the consolidation parameter

The structure Do Case is:

Do Case

Case VotingRights>50% do ExclusiveControl

Case 40%<VotigRights<50% do Exclusive Control

Case 20%<VotigRights<40% do NotableInfluence

Otherwise (VotingRights<20%) Outside the assembly

4. State of Art

Transformation of Looping Structures is a very important procedure used in parallel computing. Traditional approaches to transformation of looping structures are limited in their ability to handle compositions of loop transformations.

In an article written by Chung in 1990 [10], the authors present a formal mathematical framework which unifies the existing loop transformations. This model gave us the idea to apply these transformations in accounting for groups of companies.

The article of Vivek describes a general framework for representing iteration-reordering transformations. These transformations are a special class of program transformations and change the execution order of loop iterations.

Fernandez et al. in 1995 [11], in their article, present a method for code transformation using non unimodular transformations. We described a synergetic model to that presented by Fernandez et al. Jacobson et al. in their article describe current dependency analysis tests that can be used to identify ways for transforming sequential C code into parallel C code. Quing: “To optimize complex loop structures both effectively and inexpensively, we present a level loop transformation, dependency hosting, for optimizing arbitrarily nested loops, and an efficient framework that applies the new techniques to aggressively optimize benchmarks for better localization”.

Jain et al. [17] tell us: Based on important theorems, algorithmic methods are developed for program transformation to improve cache performance. A remarkable article is the one from the paper of Louis et al. [23] is a representative material in which the authors bring together algebraic, algorithmic and performance analysis results to design a tractable optimization algorithm over a highly expressive space.

Our work, aims to address a new concept of integrating groups of companies in parallel computing. This can be done easily by transformation of structures looping: optimization and reordering. In literature this approach has not been found.

5. Conclusions

The selection of transformation of looping structures, such as optimization and reordering, is a complex problem. In this article we have presented and analyzed the most important and commonly used transformations at the level of looping. They are useful in the context of automatic parallelization, although it is interesting to note that some changes were originally introduced as optimization of sequential execution. A future article will contain transformations of reordering iterations; they proved to be really specific purpose to highlight the inherent parallelism in the sequential programs. We tried to apply these transformations in the economics and management groups of firms, their complex activity requiring most often parallelization and business transformation for better organizational management.

REFERENCES

[1] Abu-Sufah, W. 1979. Improving the performance of virtual memory computers. Ph.D. thesis. Technical Report 78-945, University of Illinois at Urbana-Champaign.

[2] Abu-Sufah, W., Kuck, D.J. and Lawrie, D. 1981. On the performance enhancement of paging systems through program analysis and transformations, in IEEE Trans.Comp. C-30, 5, May 1981, 341-356.

[3] Aho, A.H., Sethi, R. and Ullman, J.D. 1986. Compilers: Principles, Techniques and Tools, Addison-Wesley, Reading, Massachussetts.

[4] Allen, F.E. 1969. Program optimization, in Annual Review in Automatic Programming, 5 International Tracts in Computer Science and Technology and their Applications, vol.13, Pergamon Press, Oxford, England, 239-307, 1969.

[5] Allen, F.E. and Cocke, J. 1971. A catalogue of optimizing transfor - mations, in Design and Optimizations of Compilers, R.Rustin ed., Prentice-Hall, Englewood Cliffs, New Jersey,1-30.

[6] Allen, F.E., Cocke, J. and Kennedy, K. 1981. Reduction of operator strength, in Program Flow Analysis: Theory and Applications, S.S.Muchnik and N.D.Jones eds., Prentice-Hall, Englewood Cliffs, New Jersey, 79-101.

[7] Allen, J.R. and Kennedy,K.1981. PFC: A Program to Convert Fortran to Parallel Form, Tech. Rep. MASC TR-82-6, Rice University, Houston.

[8] Barik, R. 2009. Efficient Optimization of Memory Accesses in Parallel Programs, PhD Thesis, RICE UNIVERSITY

[9] Bastoul, C. 2003. Efficient code generation for automatic parallelization and optimization (long version). Technical Report 2003/43, PRiSM, Versailles University, October 2003.

[10] Lee-Chung, L. and Chen, M. A Unified Framework, for Systematic Loop Transformations YALEU/DCS/TR-816, avalaible on-line at www.dtic.mil / dtic/tr/fulltext/u2/a249326.pdf.

[11] Fernandez,A. Labeira, J. 1995. Loop Transformations using Unimodular Matrices. IEEE Transactions and Distributing Systems, 832-840.

[12] Gannon, D., Jalby,W. and Gallivan, K.1988. Strategies for cache and local memory management by global program transformation. In Journal of Parallel and Distributed Computing, 5, 5,October 1988, 587-616.

[13] Hiranandani, S., Kennedy, K. and Tseng, C.W. 1992. Compiling Fortran D for MIMD distributed memory machines. In Communications of the ACM, vol.35, nr.8, August 1992, 66-80.

[14] Irigoin, F. and Triolet, R. 1988. Supernode partitioning. In Conference Record of the 15th ACM Symposium on Principles of Programming Languages, San Diego, California, January 1988, ACM Press, New York, 319-329.

[15] Irigoin, F. and Triolet, R.1988. Dependence approximation and global parallel code generation for nested loops. In Proceedings of the International Workshop on Parallel and Distributed Algorithms, October 1988.

[16] Jacobson,T and G. Stubbendieck. Dependency Analysis of For-Loop Structures for Automatic Parallelization of c Code, avalaible on line at www.cse.unt.edu/~sweany /CSCE5650/HANDOUTS/Jacobson.pdf.

[17] Jain, P., and Devadas, S. A Code Reordering Transformation for Improved Cache Performance, Comp. Struct. Group, Massachussets Inst. of Technology, on-line csg.csail.mit.edu/pubs/memos/Memo-436/memo-436.pdf.

[18] McKinley, K., Carr, S. and Chau-Wen Tseng. Improving Data Locality with Loop Transformations, avalaible on line at citeseerx.ist.psu.edu/…oc/summary?doi=10.1.1.1.9215.

[19] Kennedy, K., McKinley, K. and C.W. Tseng.1993. Analysis and transformation in an interactive parallel programming tool, in Concurrency Practice and Experience, 5, 7 October 1993, 575-602.

[20] Kuck, D. 1977. A survey of parallel machine organization and programming, in ACM Computing Surveys, 9, 1, March 1977, 29-59.

[21] Lam, M.S. 1988. Software pipelining: an effective scheduling technique for VLIW machines. In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, Atlanta, SIGPLAN Notices, 23, 7 , July 1988, pp.318-328.

[22] Lamport, L. The parallel execution of DO loops, in Communications of the ACM, 17(2).

[23] Louis-No¨el Pouchet, Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., Sadayappan, P. and Vasilache,N. 2011. Loop Transformations: Convexity, Pruning and Optimization.

[24] Loveman, D.B. 1977. Program improvement by source-to-source transformation, in Journal of the ACM, 1, 24, January 1977, 121-145.

[25] Marian, S., Corporaal, H., Cotofana, S. and Vassiliadis, S. 2006. Array Based Structure Loop Transformations For Cache Miss Reduction.

[26] Muraoka, Y.1971. “Parallelism exposure and exploitation in programs”, Ph.D. thesis, Tech.Rep. 71-424, University of Illinois at Urbana-Champaign.

[27] O'Brien, K., Hay. B. and J. Minish.1990. "Advanced Compiler Technology for the RISC System/6000 architecture", in IBM RISC System/6000 Technology, Publication SA23-2619, IBM Corporation, Mechanicsburg, Pennsylvania.

[28] Polychronopoulos, C.D. Advanced loop optimizations for parallel computers, in Proceedings of the 1st International Conference on Supercomputing, in Lecture Notes in Computer Science, vol.297, Springer Verlag , Berlin, pp.255-277.

[29] Qing, Y., Kennedy, K. and Adve, V. Transforming Complex Loop Nests For Locality. Avalaible on-line at http://link.springer.com /chapter/10.1007% 2F115 323 78_19.

[30] Bayliss, S. and. Constantinides, G.A. Optimizing SDRAM Bandwidth for Custom FPGA Loop Accelerators. Available on-line at http://www.academia.edu/1459515 optimizing_SDRAM_bandwidth_for_custom_FPGA_loop_accelerators.

[31] Saya, R. A Study of Loop Nest Structures and Locality in Scientific Programs, in dl.acm.org/citation.cfm?id=629363

[32] Sites, R.L. Alpha Architecture Reference Manual. Digital Press, Bedford, Massachussetts.

[33] Sarkar, V. and Thekkath, R. A General Framework for Iteration-Reordering Loop Transformations" (Technical Summary), available on-line at http://link.springer.com/chapter/10.1007%2F3-540-45545-0_18.

[34] Weiss, M. 1991 Strip mining on SIMD architectures, in Proceedings of the ACM International Conference on Supercomputing, Cologne, Germany, June 1991, ACM Press, New York, 234-243.

[35] William, J., Christophe, L., Sid-Ahmed-Ali TOUATI. 2011. An Eficient Memory Operations Optimization Technique for Vector Loops.

[36] Wolf, M.E. and Lam, M.S. 1991. A loop transformation theory and an algorithm to maximize parallelism, in IEEE Transactions on Parallel and Distributed Systems, vol.2, nr.4, October 1991, 452-471.

[37] Wolf, M.E. and Lam, M.S. 1991. A data locality optimizing algorithm, in Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, 30-44.

[38] Wolfe, M.E. 1982. Optimizing Compilers for Supercomputers, Ph.D.thesis, Report 82-1105, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, October 1982.

[39] Wolfe, M.J.1989. Optimizing Supercompilers for Supercomputers, Research Monographs in Parallel and Distributed Computing, MIT Press, Cambridge, Massachussetts.

[40] Agarwal, S. and R. Barik et al. 2008. Static Detection of Place Locality and Elimination of Ryntime Checks, The Sixth ASIAN Symposium on Programming Languages and Systems.

[41] Barik, R. and Sarkar, V. 2009. Interprocedural load elimination for dynamic optimization of parallel programs. In International Conference on Parallel Architectures and Compilation Techniques, (PACT 09), North Carolina, September 2009.

[42] Hoeflinger, J.P. and Supinski, B.R. 2005. The OpenMP memory Model. In First International Workshop on OpenMP (IWOMP 2005), Eugene, OR, June 2005, Springer – Verlag.

Author' biography with Photo

Monica Iuliana CIACA obtained her bachelor's degree at Babes Bolyai University Cluj-Napoca, in the field of Computer Science. After graduation, she has worked as programmer at the Institute for Computation Techniques from Cluj-Napoca. In 1994 she started working at the Babes Bolyai University as teaching assistant, being interested in artificial intelligence, expert systems, business information systems and software engineering. She published various articles, the most important being the one written after her participation in a Tempus Phare project, in Perugia. In 2003 she got her PhD in Mathematics and Computer Science, with a thesis on parallel computing: "Implementation Techniques in Parallel Computing". In the last five years she looked to extend her knowledge in another field: theology. She obtained her Bachelor's Degree and Master's Degree in Biblical Studies and Iconographic exegesis, in 2012, at Babes Bolyai University. Since 2004 she is Associate Professor at Babes Bolyai University, Cluj-Napoca, Faculty of Economics, in the Department of Business Information Systems.

Loredana MOCEAN has graduated Babes-Bolyai University of Cluj-Napoca, the Faculty of Computer Science, she holds a PhD diploma in Economics and she had gone through didactic position of assistant, lecturer and associate professor, since 2000 when she joined the staff of the Babes- Bolyai University of Cluj-Napoca, Faculty of Economics and Business Administration. Also, she graduated Faculty of Economics and Business Administration. She is the author of more than 20 books and over 35 journal articles in the field of Databases, Data mining, Web Ser-vices, Web Ontology, ERP Systems and much more. She is member in more than 20 grants and research projects, national and international.

Alexandru VANCEA has graduated the Computer Science Department of “Babes-Bolyai” University Cluj-Napoca in 1986. Ph.D. in Computer Science in 2000. Research areas and domains of interests: Programming Languages Design and Analysis, Automatic parallelization of programs, Distributed Programming. Teaching: Operating Systems, Computer Architecture, Fundamentals of Programming Languages.

Mihai-Constantin AVORNICULUI has graduated Faculty of Mathematics and Computer Science, Babes-Bolyai University Cluj-Napoca in 2004. He has finished his PhD studies in 2009. He works at the Business Information Systems department of FSEGA, Babes-Bolyai University Cluj-Napoca. His research interests include data mining, information systems, and advantage database systems.