MSC 68W10 DOI: 10.14529/mmp170310

CIRCULAR SHIFT OF LOOP BODY — PROGRAMME TRANSFORMATION, PROMOTING PARALLELISM

O.B. Steinberg,
South Federal University, Rostov-on-Don, Russian Federation, olegsteinb@gmail.com

The article deals with the programme transformation executing the circular shift of loop body statements. It can be used for vectorizing or parallelizing. This becomes possible due to the fact that when the order of loop body statements is changed, some of the bottom-up arcs become top-down arcs. Besides, sometimes loop carried dependence arcs are substituted by loop independent ones. It should be pointed out that in executing the circular shift the number of loop iterations is reduced by one. The transformation can be used both independently and in conjunction with other transformations promoting parallelism. These could be "forward substitution", "scalar expansion", "privatization", "array expansion", etc. The transformation under consideration in this article can be used both in hand parallelization and added to a paralleling (optimizing) compiler. Moreover, the application of the transformation results in the equivalent code only for the loops where loop unrolling is the equivalent transformation. Thus, they can contain nested loops, if statements and other programming language statements.

Keywords: parallel computations; programme transformations; dependence graph; scalar expansion; loop distribution.

Introduction

In parallelizing, attention is primarily focused on loops, they being subdivided into different types according to the extent that they lend themselves to parallel execution. Serial loops, i.e. the ones not suited for parallel execution, are attributed to one type. The loops with all loop iterations suited for parallel computation are attributed to the second type [1, 2]. The loops with only some loop iterations lending themselves to parallel computation belong to the third type [1, 2]. Moreover, some of the serial loops become suitable for parallel execution after they undergo auxiliary transformations [1–3, 5]. The transformations can result both in loops where all the loop iterations admit parallel execution [1, 2], and in loops where only some of the loop iterations are suitable for it [6, 7]. "Statement interchange", "scalar expansion", "forward substitution", "node splitting", "array expansion", "privatization" etc. may be attributed to the first type of parallelizing transformations. The present article introduces a new transformation called "circular shift". It is worth noting that a similar transformation was described in [4, 6]. It is based on the dependent migration but the works describing this transformation consider applying it only to loops free of variables which are independent of the loop count. It is exclusively regarded for maximizing the number of iterations run parallel in case of DOACROSS loops (i.e. the loops where only a part of iterations are suitable for parallel execution) [6].

The transformation dealt with in the article can be used both independently and in conjunction with other parallelizing transformations. It can promote making the loops suitable both for parallel execution and vector execution. This transformation can be both used in hand parallelization and added to a parallelizing (optimizing) compiler [1, 2, 8, 9].

1. Definitions and Concepts of Programme Transformation Theory

Let us define the fundamental concepts used in analyzing dependencies in programmes [1, 2, 10].

Definition 1. *The occurrence of a variable is the name of the variable in conjunction with the place where this variable first occurred. To any occurrence (and for arrays – at a certain value of the index expression) corresponds an access to a memory location. If in case of an access to a memory location the data are read, then such an occurrence is called usage (in), but if they are written, it is called a generator (out).*

Definition 2. *Two occurrences are said to generate information dependence [1–3, 10] if they have access to the same memory location.*

Definition 3. *Depending on what type of occurrences have access to the same memory location, four types of dependencies are distinguished: out-out – an output dependence, out-in – a true (flow) dependence, in-out – an antidependence, in-in – an input dependence.*

Definition 4. *Information dependence is called a loop independent dependence if these occurrences have access to the same memory location in the same loop iteration. Otherwise, the dependence is called a loop carried dependence [3, 10].*

Definition 5. *When making decisions in programme transformation, a dependence graph is crucial [1–3, 11]. The occurrences of variables are the vertices of this graph. The arc is directed from vertex i to vertex j if occurrences, corresponding to these vertices, generate true dependence, output dependence or antidependence. Moreover, first an occurrence corresponding to vertex i has access to the same memory location, then an occurrence corresponding to vertex j does.*

Let us also consider several basic programme transformations you should know to complete understanding of the contents of the following.

Definition 6. *Statement interchange is the name of the programme transformation which changes the order of two neighbouring statements.*

Definition 7. *Full loop unrolling is the name of the programme transformation which substitutes a programme loop:*

```
for(i = 0; i < N; i=i+1)
{
    LoopBody(i);
}
```

by a programme fragment:

```
LoopBody(0);
LoopBody(1);
...
LoopBody(N-1);
```

Remark 1. Note that in this transformation the upper bound (N) should be known as the transformation being executed and it must be greater than 0.

The present paper deals solely with the loops for which loop unrolling is the correct equivalent transformation. In particular, the body of the loop unrolled does not contain "break" and "continue" statements in C++ which are permitted only inside the loop body, and there are no labelled statements, lest several statements with the same label appear in the copies of the loop body. Besides, inside the loop body the upper bound and the loop stride are unchanged and there is no loop count generator.

Definition 8. *The transformation called "loop distribution" is of no small importance in parallelizing programmes. The purpose of the loop distribution is to substitute the loop whose body contains a great deal of statements:*

```
for(i = 0; i < N; i++)
{
    Statement1
    ...
    Statementk
    Statementk+1
    ...
    StatementM
}
```

for the equivalent programme fragment consisting of several loops whose bodies contain fewer statements:

```
for(i = 0; i < N; i++)
{
    Statement1
    ...
    Statementk
}
for(i2 = 0; i2 < N; i2++)
{
    Statementk+1
    ...
    StatementM
}
```

In parallelizing, a big loop often cannot be effectively mapped on the architecture of a parallel computer (for instance, due to the lack of resources). In this case, after the loop distribution all or at least several of the resulting loops can probably be computed in parallel. This transformation was analyzed in [1–3]. The conditions of the loop distribution being executable are similar to the conditions of vectorization or partial vectorization [10] of a one-dimensional loop. From the point of view of the parallel execution, loops can belong to different groups. In this paper the loops where all iterations are suitable for parallel execution will be considered parallel. An example of such a loop can be the one that follows:

```
for(i = 0; i < N; i++)
{
    A[i] = B[i]*C[i];
}
```

Besides, the loops suiting for vectorizing are also distinguished.

Example 1. Consider a loop where no iteration (except the first) can be executed prior to the previous one:

```
for(i = 0; i < N; i++)
{
    A[i+1] = B[i]*C[i];
    D[i] = A[i]*E[i];
}
```

However, all the iterations for the first assignment statement can be executed concurrently first, and then the same can be done for the second one. To designate that vector execution of the loop body statements is possible, we write it in the way similar to Fortran:

```
A[1:N] = B[0:N-1]*C[0:N-1];
D[0:N-1] = A[0:N-1]*E[0:N-1];
```

There are also loops suiting for vector execution using vectors of a certain length. Consider an example of a loop like this.

Example 2. Consider a recurrent loop [7]

```
for(i = 0; i < 4*N; i++)
{
    A[i+4] = A[i]*C[i];
}
```

This loop can be executed using vector registers of length 4.

```
for(i = 0; i < N; i=i+4)
{
    A[i+4:i+7] = A[i:i+3]*C[i:i+3];
}
```

Note that in the initial loop in this example, every four iterations could run in parallel. Thus, it can be both vector executable and run in parallel.

Remark 2. Note that if all the iterations in any loop can run in parallel, it can also be executed using vector architecture. For instance, the research of the simultaneous usage of vectorizing and parallelizing in recurrent loops was examined in [12].

2. The Loop Body Circular Shift

The present work is devoted to the transformation which will be called "circular shift". This transformation consists of a substituting code fragments

```
Statement1(0);
...
Statementk-1(0);
for (i = 0; i < N-1; i++)
```

```
{
    Statementk(i);
    ...
    StatementM(i);
    Statement1(i+1);
    ...
    Statementk-1(i+1);
}
Statementk(N-1);
...
StatementM(N-1);
```

where N $>$ 0, for the loop

```
for (i = 0; i < N; i++)
{
    Statement1(i);
    ...
    StatementM(i);
}
```

In this case a circular shift of length k is said to have been applied.

Theorem 1. *The "circular shift of length k" programme transformation is equivalent.*

Proof. Apply the "loop unrolling" transformation to the cycles of initial and resulting code fragments of the circular shift definition. This will result in two identical fragments. Loop unrolling being an equivalent transformation, initial fragments are equivalent as well. Circular shift is a transformation promoting parallelizing. □

Example 3. Consider loop:

```
for(i = 0; i < N; i++)
{
    A[i] = B[i]+C[i];
    C[i+1] = D[i]*E[i];
}
```

This loop contains a loop carried flow dependence generated by array C occurrences. Its existence prevents different loop iterations from run in parallel. Now apply circular shift to the initial loop.

```
A[0] = B[0]+C[0];
for(i = 0; i < N-1; i++)
{
    C[i+1] = D[i]*E[i];
    A[i+1] = B[i+1]+C[i+1];
}
C[N] = D[N]*E[N];
```

This results in a loop where all the iterations can be computed independently, consequently, it is suitable both for the vector and parallel execution.

3. Using the "Circular Shift" Transformation in Conjunction with Other Programme Transformations

In parallelizing both single transformations and whole sequences can be used. For instance, sometimes a statement interchange promotes distribution [13], and a loop distribution can result in removing the dead code.

3.1. Simultaneous Use of Circular Shift and Forward Substitution

In some cases "forward substitution" programme transformation can be used to eliminate loop carried antidependencies. Forward expression substitution [2, 3] is a transformation substituting the right part of an assignment statement for the succeeding occurrence of the left part of the same statement.

Example 4. Consider the loop which contains a loop carried flow dependence formed by variable C occurrences:

```
for(i = 0; i < N; i++)
{
    A[i] = B[i]+C;
    C = D[i]+E[i];
}
```

Apply the circular shift to it:

```
A[0] = B[0]+C;
for(i = 0; i < N-1; i++)
{
    C = D[i]+E[i];
    A[i+1] = B[i+1]+C;
}
C = D[N-1]+E[N-1];
```

This loop is not yet suited either for the vector or parallel execution. But if forward substitution is applied to this loop, a new loop can be obtained:

```
A[0] = B[0]+C;
for(i = 0; i < N-1; i++)
{
    C = D[i]+E[i];
    A[i+1] = B[i+1]+D[i]+E[i];
}
C = D[N-1]+E[N-1];
```

Now, applying the loop distribution, we obtain as follows:

```
A[0] = B[0]+C;
for(i = 0; i < N-1; i++)
{
    C = D[i]+E[i];
}
```

```
for(i = 0; i < N-1; i++)
{
    A[i+1] = B[i+1]+D[i]+E[i];
}
C = D[N-1]+E[N-1];
```

It can readily be noticed that only the last iteration is left of the first loop, the one which computes the final value of variable C. It is a "dead code" as well, because the value of variable C is not used within the loop, but is calculated anew immediately after it:

```
A[0] = B[0]+C;
C = D[N-2]+E[N-2];
for(i = 0; i < N-1; i++)
{
A[i+1] = B[i+1]+D[i]+E[i];
}
C = D[N-1]+E[N-1];
```

The result is a loop where all the iterations can be computed independently, and, consequently, it is suitable both for the vector and parallel execution. The applicability of this approach to eliminating the loop carried dependence is limited by the applicability conditions of the "forward substitution" transformation.

3.2. Simultaneous Use of Circular Shift and Scalar Expansion

Scalar expansion [1–3, 10] in substituting consists of a certain scalar variable occurrences in the loop body for the occurrences of a temporary array. For instance, it can be used to eliminate the bottom-up arc of the loop carried dependence. Due to this, the scalar expansion is used to make the loop suitable for parallelizing or for its distribution. It should be noted that this approach presupposes extra memory consumption.

Example 5. Let us consider another way of making the loop in example 4 suited for parallel execution. Apply scalar expansion to it:

```
C_temp[0] = C;
for(i = 0; i < N; ++)
{
    A[i] = B[i]+C_temp[i];
    C_temp[i+1] = D[i]+E[i];
}
C = C_temp[N];
```

Now "statement interchange" programme transformation can be applied to assignment statements in the resulting loop, which makes the vector execution of the loop possible:

```
C_temp[0] = C;
C_temp[1:N] = D[0:N-1]+E[0:N-1];
A[0:N-1] = B[0:N-1]+C_temp[0:N-1];
C = C_temp[N];
```

Besides, "loop distribution" transformation can now be applied:

```
C_temp[0] = C;
for(i = 0; i < N; i++)
{
    C_temp[i+1] = D[i]+E[i];
}
for(i = 0; i < N; i++)
{
    A[i] = B[i]+C_temp[i];
}
C = C_temp[N];
```

This results in two loops being obtained, all the iterations of each suitable for independent execution. Thus, a programme fragment suitable for parallel execution is obtained.

Remark 3. Note that vectorizing and parallelizing in this example became possible only due to the feasibility of "statement interchange" transformation. Let us proceed to considering the simultaneous use of the "scalar expansion" and "circular shift" transformations.

Example 6. Apply circular shift to the loop in example 4:

```
A[0] = B[0]+C;
for(i = 0; i < N-1; i++)
{
    C = D[i]+E[i];
    A[i+1] = B[i+1]+C;
}
C = D[N-1]+E[N-1];
```

Now apply scalar expansion to the resulting loop:

```
A[0] = B[0]+C;
C_temp[0] = C;
for(i = 0; i < N-1; i++)
{
    C_temp[i] = D[i]+E[i];
    A[i+1] = B[i+1]+C_temp[i];
}
C = D[N-1] + E[N-1];
```

The result is a loop in which all iterations can be both the vector executed and run in parallel. Besides, if the vector execution is required, the number of the elements in the temporary array created by the scalar expansion can be equal to the vector register length:

```
A[0] = B[0]+C;
C_temp[0] = C;
for(i = 0; i < (N-1)/4; i=i+4)
{
    C_temp[i:i+3] = D[i:i+3]+E[i:i+3];
    A[i+1:i+4] = B[i+1:i+4]+C_temp[i:i+3];
}
```

```
for(i = (N-1) - (N-1)%4; i < (N-1); i++)
{
    C = D[i]+E[i];
    A[i+1] = B[i+1]+C;
}
C = D[N-1] + E[N-1];
```

In the fragment above, loop L1 is suitable for the vector execution on the architecture with vector registers containing 4 elements. Vectors of such length occur, for instance, while using SSE2 and type int arrays.

3.3. Simultaneous Use of Circular Shift and Array Expansion

Let an array occurrence, independent of the given loop count, be present inside the loop. Then within the limits of this loop the given occurrence can be regarded as a scalar variable.

Example 7. Consider loop:

```
for(i = 0; i < N; i++)
{
    A[i] = B[i]+C[j][k];
    C[j][k] = A[i+2]+E[i];
}
```

Having the applied circular shift and the transformation identical to the scalar expansion we obtain the following fragment suitable for the vector execution:

```
A[0] = B[0]+C[j][k];
C_temp[0] = C[j][k];
for(i = 0; i < N-1; i++)
{
    C_temp[i] = A[i+2]+E[i];
    A[i+1] = B[i+1]+C_temp[i];
}
C[j][k] = A[N+1]+E[N-1];
```

Now, having the applied loop distribution, we can obtain a programme fragment containing two loops suitable for parallelizing:

```
A[0] = B[0]+C[j][k];
C_temp[0] = C[j][k];
for(i = 0; i < N-1; i++)
{
    C_temp[i] = A[i+2]+E[i];
}
for(i2 = 0; i2 < N-1; i2++)
```

```
{
    A[i+1] = B[i+1]+C_temp[i];
}
C[j][k] = A[N+1]+E[N-1];
```

Remark 4. It should be noted that when applied to index variables, the transformation identical to the scalar expansion is known as the array expansion [11, 14].

3.4. Simultaneous Use of Circular Shift and Privatization

Sometimes, if a scalar variable is present in a loop body, its separate copy is created for each parallel flow. Such transformation is known as privatization [1, 2].

Example 8. Consider the loop containing a loop generated flow dependence formed by scalar variable C occurrences:

```
for(i = 0; i < N; i++)
{
    for(j = 0; j < M; j++)
    {
        A[j] = B[j]+C;
    }
    C = D[i]*E[i];
}
```

Apply the circular shift to the body of this loop:

```
for(j = 0; j < M; j++)
{
    A[j] = B[j]+C;
}
for(i = 0; i < N-1; i++)
{
    C = D[i]*E[i];
    for(j = 0; j < M; j++)
    {
        A[j] = B[j]+C;
    }
}
C = D[N-1]*E[N-1];
```

This results in loop L2 where if a separate ("private") copy of variable C is created for each iteration, all its iterations can be computed independently. All the iterations of loop L1 can also be computed in parallel. Besides, all the loops of the fragment obtained can be the vector executed using vectors of length 4. To do this, "private" vector CC of length 4 should be created with each coordinate equal to C.

Acknowledgements. *The reported study was funded by RFBR, according to the research project No 16-31-60055 mol_a_dk.*

References

1. Allen R., Kennedy K. *Optimizing Compilers for Modern Architectures.* San Francisco, San Diego, N.Y., Boston, London, Sidney, Tokyo, Morgan Kaufmann Publishers, 2002. 790 p.
2. Wolfe M. *High Performance Compilers for Parallel Computing.* Redwood City, Addison-Wesley Publishing Company, 1996. 570 p.
3. Steinberg B.J. *Matematicheskie metody rasparallelivaniya rekurrentnykh programnykh tsiklov na superkompyutery s parallel'noy pamyatyu* [Parallelizing Recurrent Program Cycles with Irregular Superposition Computation]. Rostov-on-Don, Rostov University Publishing House, 2004. 192 p.
4. Duo Liu, Zili Shao, Meng Wang, Minyi Guo, Jingling Xue. Optimal Loop Parallelization for Maximizing Iteration-Level Parallelism. *Proceedings of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 09).* N.Y., ACM, 2009, pp. 67–76. DOI: 10.1145/1629395.1629407
5. Steinberg O.B. [Parallelizing Recurrent Program Cycles with Irregular Superposition Computation]. *Izvestiya vuzov. Severo-Kavkazskii region. Natural Science*, 2009, no. 2, pp. 18–21. (in Russian)
6. Duo Liu, Yi Wang, Zili Shao, Minyi Guo, Jingling Xue. Optimally Maximizing Iteration-Level Loop Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2012, vol. 23, no. 3, pp. 564–572. DOI: 10.1109/TPDS.2011.171
7. Steinberg O.B., Sukhoverkhov S.E. [Recurrent Program Loops with Stability Check]. *Information Technologies*, 2010, no. 1, pp. 40–45. (in Russian)
8. Muchnick S.S. *Advanced Compiler Design and Implementation.* San Francisco, Morgan Kauffman, 1997. 856 p.
9. Aho A.V., Lam M.S., Sethi R., Ullman J.D. *Compilers: Principles, Techniques, and Tools.* London, Pearson Education, 2007. 1014 p.
10. Evstigneev V.A., Sprogis S.V. [Vectorizing Programmes]. *Vektorizatsiya programm: teoriya, metody, realizatsiya* [Vectorizing Programmes: Theory, Methods, Implementation]. Moscow, Mir, 1991, pp. 246–267. (in Russian)
11. Shulzhenko A.M. Issledovanie informatsionnykh zavisimostey programm dlya analiza rasparallelivayushchikh preobrazovaniy [Researching Information Dependences of Programs for Analyzing Transformations Used for Parallelizing. The Dissertation for Scientific Degree of the Candidate of Technology]. Rostov-on-Don, 2006, 200 p.
12. Steinberg O.B. Rasparallelivanie tsiklov, dopuskayushchikh rekurrentnye zavisimosti [Parallelizing Loops Allowing Recurrent Dependences. The Dissertation for Scientific Degree of the Candidate of Physics and Mathematical Science]. Institute for System Programming of the Russian Academy of Sciences, Moscow, 2014.
13. Steinberg O.B. [Minimizing the Number of Temporary Arrays in Loop Distribution Problem]. *Izvestiya vuzov. Severo-Kavkazskii region. Natural Science*, 2011, no. 5, pp. 31–35. (in Russian)
14. Feautrier P. Array Expansion. *Proceedings of the 2nd International Conference on Supercomputing*, N.Y., ACM, 1988, pp. 429–441. DOI: 10.1145/55364.55406

Received June 16, 2016

УДК 519.685.3 **DOI: 10.14529/mmp170310**

КРУГОВОЙ СДВИГ ТЕЛА ЦИКЛА – ПРЕОБРАЗОВАНИЕ ПРОГРАММ, СПОСОБСТВУЮЩЕЕ РАСПАРАЛЛЕЛИВАНИЮ

О.Б. Штейнберг, Южный федеральный университет, г. Ростов-на-Дону

В статье рассматривается преобразование программ, выполняющее круговой сдвиг операторов тела цикла. Его можно использовать для векторизации или распараллеливания. Это становится возможным благодаря тому, что при изменении порядка следования операторов тела цикла некоторые дуги, идущие снизу вверх, превращаются в дуги, идущие сверху вниз. Также иногда циклически порожденные дуги зависимости заменяются на циклически независимые. Следует отметить, что при выполнении кругового сдвига число итераций цикла уменьшается на единицу. Преобразование может применяться как независимо, так и совместно с другими преобразованиями, способствующими распараллеливанию. Такими преобразованиями могут являться: «подстановка вперед», «растягивание скаляров», «приватизация», «экспансия массивов» и другие. Возможности применения рассматриваемого в статье преобразования распространяются как на ручное распараллеливание, так и на добавление его в распараллеливающий (оптимизирующий) компилятор. При этом ограничение на циклы, применение преобразования к которым будет приводить к эквивалентному коду, сводится к циклам, для которых эквивалентной является раскрутка. Таким образом, они могут содержать вложенные циклы, условные операторы и другие операторы языка программирования.

Ключевые слова: параллельные вычисления; преобразования программ; граф информационных связей; растягивание скаляров; разбиение цикла.

Литература

1. Allen, R. Optimizing Compilers for Modern Architectures / R. Allen, K. Kennedy. – San Francisco; San Diego; N.Y.; Boston; London; Sidney; Tokyo: Morgan Kaufmann Publishers, 2002. – 790 p.
2. Wolfe, M. High Performance Compilers for Parallel Computing / M. Wolfe. – Redwood City: Addison-Wesley Publishing Company, 1996. – 570 p.
3. Штейнберг, Б.Я. Математические методы распараллеливания рекуррентных программных циклов на суперкомпьютеры с параллельной памятью / Б.Я. Штейнберг. – Ростов-на-Дону: Изд-во Ростовского ун-та, 2004. – 192 с.
4. Duo Liu. Optimal Loop Parallelization for Maximizing Iteration-Level Parallelism / Duo Liu, Zili Shao, Meng Wang, Minyi Guo, Jingling Xue // Proceedings of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 09). – N.Y.: ACM, 2009. – P. 67–76.
5. Штейнберг, О.Б. Распараллеливание рекуррентных циклов с нерегулярным вычислением суперпозиций / О.Б. Штейнберг // Известия вузов. Северо-Кавказский регион. Естественные науки. – 2009. – № 2. – С. 18–21.
6. Duo Liu. Optimally Maximizing Iteration-Level Loop Parallelism / Duo Liu, Yi Wang, Zili Shao, Minyi Guo, Jingling Xue // IEEE Transactions on Parallel and Distributed Systems. – 2012. – V. 23, № 3. – P. 564–572.
7. Штейнберг, О.Б. Автоматическое распараллеливание рекуррентных циклов с проверкой устойчивости / О.Б. Штейнберг, С.Е. Суховерхов // Информационные технологии. – 2010. – № 1. – С. 40–45.

8. Muchnick, S.S. Advanced Compiler Design and Implementation / S.S. Muchnick. – San Francisco: Morgan Kauffman, 1997. – 856 p.
9. Aho, A.V. Compilers: Principles, Techniques, and Tools / A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman. – London: Pearson Education, 2007. – 1014 p.
10. Евстигнеев, В.А. Векторизация программ / В.А. Евстигнеев, С.В. Спрогис // Векторизация программ: теория, методы, реализация. – М.: Мир, 1991. – С. 246–267.
11. Шульженко, А.М. Исследование информационных зависимостей программ для анализа распараллеливающих преобразований: дис. ... канд. техн. наук / А.М. Шульженко. – Ростов-на-Дону, 2006.
12. Штейнберг, О.Б. Распараллеливание циклов, допускающих рекуррентные зависимости: дис. ... канд. физ.-мат. наук / О.Б. Штейнберг. – Москва, 2014.
13. Штейнберг, О.Б. Минимизация количества временных массивов в задаче разбиения циклов / О.Б. Штейнберг // Известия высших учебных заведений. Северо-Кавказский регион. Естественные науки. – 2011. – № 5. – С. 31–35.
14. Feautrier, P. Array Expansion / P. Feautrier // Proceedings of the 2nd International Conference on Supercomputing. – N.Y.: ACM, 1988. – P. 429–441.

Олег Борисович Штейнберг, кандидат физико-математических наук, кафедра «Алгебра и дискретная математика», Южный федеральный университет (г. Ростов-на-Дону, Российская Федерация), olegsteinb@gmail.com.

Поступила в редакцию 16 июня 2016 г.