ISSN 1064-2269, Journal of Communications Technology and Electronics, 2014, Vol. 59, No. 6, pp. 639–646. © Pleiades Publishing, Inc., 2014.
Original Russian Text © L.I. Rubanov, 2013, published in Informatsionnye Protsessy, 2013, Vol. 13, No. 4, pp. 295–305.

Parallelization of Nonuniform Loops in Supercomputers with Distributed Memory

L. I. Rubanov

Institute for Information Transmission Problems, Russian Academy of Sciences, Bolshoy Karetnyi per. 19, Moscow, 127994 Russia

e-mail: rubanov@iitp.ru

Received October 17, 2013

Abstract—A template algorithm for parallel execution of independent iterations of the repetitive loop on a multiprocessor computer with distributed memory is constructed. Regardless of the number of processors, the algorithm must provide efficient utilization of computing capacity under essentially different complexities of iterations and/or performance of processors. The interprocessor data communication and control of parallel computations are assumed to be implemented using a standard message-passing interface (MPI), which is widely used in such systems. Existing methods for the loop parallelization are analyzed and the corresponding efficiencies are empirically estimated for various models of iteration nonuniformity.

Keywords: high-performance computing, parallelization, distributed memory, MPI, nonuniform loops

DOI: 10.1134/S1064226914060175

1. INTRODUCTION

In most computer programs, loop operators are quite rare in comparison with arithmetic and logical operators although the computations in the loop are predominantly performed with the aid of the former. If the next iteration in a loop depends on the results of previous iterations, the computations can be accelerated only with the aid of loop optimization, which is beyond the scope of this work. We consider algorithms with independent iterations in a loop. Thus, an arbitrary order of iterations is allowed. Such algorithms are often employed in the simulation of physicochemical and biological processes, processing of large data arrays, numerical integration, etc. For example, each trajectory in the Monte-Carlo simulation of a random process aimed at the estimation of probabilities of final states can be represented as a single iteration of an external loop that is used for the calculations. A relatively large number of the corresponding iterations is needed for reliable and accurate estimations, so that the high-performance computing becomes necessary. A natural method for acceleration of loop calculations involves simultaneous execution of several iterations at different processors (i.e., parallelization of loops (PL)).

The PL possibilities and methods significantly depend on the architecture of the multiprocessor system that is used for computations. In shared-memory supercomputers, in which each processor has an independent access to the shared RAM, the PL is often implemented using an OpenMP system of parallel programming [1], which is user-friendly and efficient. The shared-memory systems are widely distributed (most desktop workstations and PCs are equipped with multicore CPUs) but the total number of processors is rarely greater than 64, which substantially impedes an increase in the efficiency. The rate of computations can be increased by a factor of several hundreds or thousands only with the aid of *distributed-memory* systems (clusters), in which individual memory of a node is unavailable for the remaining processors of the cluster. The contacts of processes on different nodes that can be spatially separated are provided by communication libraries of parallel programming. Below, we consider a standard message-passing interface (MPI), which is widely used in such systems [2]. Note a possibility of PL in a single node using the OpenMP. However, such a complication of the loop structure is unnecessary, since the communications within a node are efficiently performed via shared memory in most MPI systems. Thus, a programmer may develop a single code for parallel execution in a single node and cluster in general.

Another important aspect that must be taken into account in PL involves the comparison of processing times for different iterations (i.e., loop uniformity). The iterations can be synchronous provided that the duration of each iteration is roughly constant. Such an approach is used in systolic architecture. However, the processing times of iterations in the above problems are varied in wide ranges depending on the dimension of data that are processed at the corresponding iteration or a current value of a pseudo-random sequence. A similar scenario corresponds to identical processing times and different performances of the cluster nodes.

In this case, the application of the PL algorithms that involve the processing with the aid of the systolic principle leads to relatively low efficiencies (see below). We primarily consider the parallelization of significantly nonuniform loops in the systems with distributed memory.

2. GENERAL FORMULATION OF THE PROBLEM AND METHODS FOR EFFICIENCY ESTIMATION

The problem under study can be algorithmically represented using the following 1D loop that must be parallelized (without loss of generality, we use the C syntax):

Algorithm 1.

```
for (n = 0; n < N; n++){
  f = Func (param[n]);
  Merge (f);
}
```

Here, Func is a side-effectless function, in which the computational complexity of the loop is concentrated. A scalar function is used in the example but Func may return a pair of values or vector. The same holds true for param, which is a parameter rather than argument that can be absent in a particular case. We assume that the function contains the data that are needed for the calculations (e.g., a static array or the data that are generated using a pseudo-random sequence). Note the possibility of loop processing of data arrays, since loop variable n may serve as the parameter of the function.

Computationally simple function Merge exhibits a side effect that leads to the result of the general computation. For example, such a function can calculate a sum or product of the values of f, search for minimum or maximum value, construct a histogram, etc. In the framework of such a problem, the results of each iteration are not stored (the storage is a problem at relatively large N) and we obtain general statistical data using a sample of calculated values.

The formulation of the problem differs from that in the CENTAUR project [3], in which the side effect is absent in the loop and the main function provides coordinate-wise transformation of the original data vector whose dimension is repeat number N. They implicitly assume that the computational complexity of the transformation is almost independent of n (remains constant). Thus, a search for the most efficient PL is reduced to selection of the appropriate distribution of data elements over the nodes of computational network with hybrid architecture. With allowance for complexity of this problem, we concentrate on the efficient parallelization of significantly nonuniform loops in which repeat number is weakly related to the dimension of input data. Note a possibility of the nested loops, which are useful in combinatorial algorithms (e.g., the processing of all possible pairs of original data can be conveniently implemented as a double nested loop).

The efficiency of the above PL methods is estimated using the following procedure. We employ function Func that provides almost 100% loading of the processor and memory-access channel over a given time interval whose duration serves as a parameter. Immediately prior to and after the end of the loop, the processors are synchronized and a time difference between these time moments, which serves as the computation time of the loop, is measured. The measurement accuracy is preserved under a wide-range variation in the number of processors if an increase in the number of processors is accompanied by a proportional increase in the duration of the working interval of function Func. Thus, the computation time of a loop must be constant in a perfect PL regardless of the number of processors. For a uniform loop with iteration time τ for one processor, the computation time must be $N\tau$ at any number of processors. Tables present the deviations from this value in percents. Naturally, the deviation is positive, since the above level cannot be reached in a real system.

For comparison, we present the results for constant τ (model C). In addition, we consider the following models of the loop nonuniformity:

U, iteration time is a random quantity that is uniformly distributed over interval $(0, 2\tau)$;

P, iteration time is determined by the time interval between sequential events of a Poisson process with intensity τ;

L, the iteration has linear complexity with respect to n, so that the iteration time is $C_1(n + 1)$, where constant $C_1 \approx 2\tau/N$ is chosen in such a way that the mean iteration time is τ;

Q, the iteration has quadratic complexity with respect to n, so that the iteration time is $C_2(n + 1)^2$, where constant $C_2 \approx 3\tau/N^2$ is chosen in such a way that the mean iteration time is τ.

For these models, the expected calculation time is $N\tau$ for any number of processors.

The results are obtained using an MVS-100K supercomputer of the Joint Supercomputer Center, Russian Academy of Sciences [4].

3. DIVISION OF THE DOMAIN OF VARIATION OF LOOP VARIABLE

The iterations of the loop are mutually independent, and PL can be implemented using the execution of a part of iterations on each processor. The domain of variation of integer loop variable $[0, N)$ is divided into m approximately identical parts with respect to the number of processors, so that the kth processor executes iterations with numbers $[ks, (k + 1)s)$, where $s = \lfloor (N + m - 1)/m \rfloor$ is the size of a part and $\lfloor \cdot \rfloor$ denotes integer part. Thus, Algorithm 1 is represented as (evident declarations of variables are omitted)

Algorithm 2.

```
MPI_Comm_size (MPI_COMM_WORLD, &m);
MPI_Comm_rank (MPI_COMM_WORLD, &k);

int s = (N+m-1)/m;
double f0 = -1.0;
double *ff = (double*)malloc(m*sizeof(double));

for (n = k*s; n < (k+1)*s; n++) {
     f = n < N ? Func (param[n]) : f0;
     MPI_Gather (&f, 1, MPI_DOUBLE, ff, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
     if (k==0)
       Merge1(ff);
}
```

Here, root branch with number $k = 0$ collects the values that result from the calculation of f in each branch. For this purpose, we employ function `Merge1` that differs from the original merge function. The difference lies in the fact that the former has vector argument and the dimension of vector is equal to the total number of processors. Note that N can be indivisible by m. Therefore, we introduce specific $f_0 = -1$ that is transferred by inactive branches of parallel program in the last loop pass. An additional task of the `Merge1` function is identification and rejection of such data.

Algorithm 2 corresponds to a relatively complicated scenario in which the merge function acts specifically (e.g., constructs the histrogram of the values of f). For simpler scenarios, the MPI standard provides complete functions of collective data exchange with merging and a possibility of defining new types of functions. For example, the loop of Algorithm 2 can be simplified for the calculation of the mean value of f:

Algorithm 3.

```
double sum = 0;
double r;
for (n = k*s; n < (k+1)*s; n++) {
  f = n < N ? Func(param[n]) : 0.0;
  MPI_Reduce (&f, &r, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
  if (k==0)
    sum += r;
}
```

When the loop is terminated, variable `sum` in the root branch contains the value of f that results from the summation over all iterations of the loop. Then, this variable must be divided by number of iterations N. In Algorithm 3, specific value $f = 0$ does not change the accumulated sum.

Table 1 presents the results of PL using Algorithms 2 and 3 for different models of nonuniformity of the loop iterations. Hereafter, we use $N = 10^6$ and $\tau = 1$ ms, so that the ideal computation time is always 1000 s.

Table 1 shows that such PL can be used for uniform iterations and is hardly applicable for the above models of nonuniform loop. The PL efficiency decreases by a factor of no less than 2 when the number of processors increases. This is an expected result, since the MPI standard [2] allows the library developer to implement collective operations (including MPI_Gather and MPI_Reduce) with the synchronization of all processes. Thus, the collective data exchange is terminated not earlier than function `Func` is calculated in the most difficult of m parallel iterations. The remaining branches wait for this moment (Table 1). If the computational complexity of each iteration can be estimated a priori, the effect can be compensated for using rearrangement of iterations in such order that iterations with almost identical complexities are performed simultaneously. However, the above scheme for the division of the domain of loop variable is inconvenient in this case and an alternative solution is discussed below.

Thus, the main possibility for an increase in the PL efficiency using Algorithms 2 and 3 lies in the elimination of the collective input/output inside the loop and application of the corresponding external operations. The merging of the results of function

Table 1. Deviation (in percents) from an ideal computation time for PL using Algorithms 2 and 3

Number of processors	Model of nonuniformity of loop iterations				
	C	U	P	L	Q
1	0.25	0.18	0.16	0.19	0.17
2	0.25	17.5	33.5	50.1	75.1
4	0.13	26.9	62.1	75.1	131
8	0.34	33.5	89.7	87.6	164
16	0.36	33.5	89.8	93.8	182
32	0.07	33.7	90.0	96.9	191
64	0.06	33.9	91.4	98.4	195
128	0.06	34.6	92.2	99.2	198
256	0.04	35.0	95.4	99.6	199
512	0.06	36.4	98.9	99.8	199
1024	0.06	37.5	101.4	99.9	200

`Func` is locally performed in each parallel branch. When the loop is terminated in all branches, the merging of general data is performed and a collective operation is employed. Thus, we obtain

Algorithm 4.

```
double *ff = (double*)malloc(m*sizeof(double));
double r;
for (n = k; n < N; n+=m) {
  f = Func(param[n]);
  r = Merge(f);
}
MPI_Gather(&r, 1, MPI_DOUBLE, ff, 1, MPI_DOUBLE,0, MPI_COMM_WORLD);
if (k==0)
  Merge1(ff);
```

In this algorithm, we employ an alternative method for the division of the domain of loop variable in which the division into subdomains is performed with interleaving. A specific numerical indicator of going beyond the loop limits (Algorithms 2 and 3) becomes unnecessary and the code is significantly simplified. In addition, such an approach is more convenient in the parallelization of the nested loops. For this purpose, the interleaved scheme is modified using, for example, the mapping of the vector of loop indices on a natural series and assignment of the elements of the *k*th class of residues with respect to the modulus of the number of processors to the *k*th processor:

```
int n = 0;
for (i = 0; i < M; i++) {
  for (j = 0; j < N; j++) {
    if (n++ % m != k)
      continue;
    f = Func(...)
    ...
   }
}
```

Table 2. Deviation (in percents) from an ideal computation time for PL using Algorithms 4 and 5

Number of processors	Model of nonuniformity of loop iterations				
	C	U	P	L	Q
1	0.25	0.18	0.17	0.19	0.17
2	0.10	0.11	0.22	0.11	0.09
4	0.05	0.18	0.24	0.07	0.06
8	0.20	0.22	0.36	0.21	0.17
16	0.29	0.37	0.64	0.14	0.18
32	0.06	0.83	1.17	0.12	0.09
64	0.05	1.14	2.13	0.07	0.06
128	0.06	2.20	2.70	0.04	0.04
256	0.05	2.74	4.04	0.04	0.05
512	0.06	4.29	6.24	0.06	0.08
1024	0.05	6.06	9.28	0.10	0.15

Similarly to Algorithm 2, Algorithm 4 can be substantially simplified if the merging operation of individual iterations is defined in the MPI standard or can be specified by the programmer. Thus, the algorithm for calculation of the mean value is represented as

Algorithm 5.

```
double sum;
double r = 0;
for (n = k; n < N; n+=m)
  r += Func(param[n]);
MPI_Reduce(&r, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Table 2 presents the PL results for the improved algorithm.

It is seen that the elimination of the collective MPI operations inside the loop with nonuniform iterations leads to a significant increase in the PL efficiency. However, in the presence of significant random fluctuations of the complexity of iterations (models *U* and *P*), the PL efficiency substantially decreases with an increase in the number of processors and a further improvement becomes possible.

4. CHANGE OF THE ORDER OF ITERATIONS

The results in Table 2 may seem confusing. The mean iteration time in each case is τ, and the iteration times for models *U*, *L*, and *Q* belong to intervals $(0, 2\tau)$, $(0, 2\tau)$, and $(0, 3\tau)$, respectively. For the *P* model (Poisson flow), the upper boundary cannot be determined in spite of the fact that the probability of relatively large values is extremely low. For definiteness, note that the data for model *P* are obtained using a pseudo-random series with the length $N = 10^6$ that is realized on interval $(0, 13.3\tau)$, which partly accounts for worse results for this model. However, the PL efficiency for model *U* is lower than the efficiencies for model *L* with the same interval of iteration times and model *Q* in which the iteration time is varied in the wider interval.

This effect can be interpreted with allowance for a monotonic increase in the iteration time in models *L* and *Q* with an increase in *n*. Therefore, the total iteration times in parallel branches of Algorithms 4 and 5 are almost identical (at least, commensurable). In model *U*, the iteration times are uniformly distributed with respect to *n*, so that iterations with different durations can be obtained in different branches. Evidently, the total computation time of the loop is determined by the worst result, since the remaining branches wait for the termination of the collective operation that follows the termination of the loop. The effect is less developed at relatively large *N*, since the sum is normalized in accordance with the central limit theorem, but still must be taken into account owing to relatively slow convergence.

A natural solution can be used in scenarios in which an a priori estimation is possible for the complexity of iterations. The iterations can be ordered with respect to complexity in such a way that the iteration times monotonically decrease in each parallel branch of Algorithms 4 and 5. Thus, the working times of the branches become almost equal and, hence, the pauses at the ends of the working sessions decrease.

The advantage of such an approach can be demonstrated using Algorithm 3, which employs a collective operation inside the parallelized loop. We change the scheme of division of the domain of loop variable (see Algorithm 5), since the original scheme does not balance the loading of branches, and obtain

Table 3. Deviation (in percents) from an ideal computation time for PL using Algorithm 6

Number of processors	Model of nonuniformity of loop iterations				
	C	U	P	L	Q
1	0.25	0.21	0.18	0.19	0.19
2	0.25	0.15	0.15	0.18	0.15
4	0.17	0.10	0.09	0.09	0.16
8	0.35	0.31	0.10	0.36	0.31
16	0.36	0.20	0.20	0.24	0.17
32	0.06	0.12	0.15	0.12	0.21
64	0.06	0.08	0.11	0.06	0.06
128	0.07	0.05	0.13	0.04	0.05
256	0.05	0.05	0.22	0.05	0.06
512	0.06	0.06	0.45	0.06	0.09
1024	0.05	0.12	0.93	0.12	0.16

Algorithm 6.

```
MPI_Comm_size(MPI_COMM_WORLD, &m);
MPI_Comm_rank(MPI_COMM_WORLD, &k);
double sum = 0;
double r, f;
int s = ((N+m-1)/m)*m;
for (n = k; n < s; n+=m) {
  f = n < N ? Func(param[n]) : 0.0;
  MPI_Reduce(&f, &r, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
  if (k==0)
    sum += r;
}
```

Table 3 presents the results for Algorithm 6 in the case when the iterations are ordered with respect to a decrease in iteration time with increasing *n*. It is seen that the PL efficiencies are lower than the ideal efficiency by less than 1% in spite of the presence of the collective MPI operation inside the loop.

Finally, note that the a priori estimation of the computation complexity of loop iterations is possible in several cases. In particular, the computation complexity of the processing of large data arrays can be related to the dimension of the processed data element and the corresponding relationship can be available. For example, the dynamic programming is needed for each iteration in the problem of protein clusterization [5] for the pairwise alignment of two amino-acid sequences with lengths *m* and *n* (the computation complexity is $O(mn)$). The pairs of sequences can be ordered with respect to a decrease in quantity *mn* using the above PL procedure. Note also that the a priori estimation of the complexity of iterations makes it possible to construct more complicated (in comparison with simple ordering) dynamic strategies for planning of processing for the given number of processors. However, the problem is beyond the scope of this work.

5. FUNCTIONAL SEPARATION OF BRANCHES

The analysis of parallelization of a nonuniform loop should be supplemented with the analysis of the PL methods based on the functional specificity of processors (i.e., branches of a parallel code). The methods employ the master/slave architecture in which the slave groups execute loop iterations and the master group (normally, a single root process) provides administration of computations (i.e., task distribution for working processors).

The fact that several processors are not involved in the data processing becomes insignificant when the total number of processors increases. Note that the method is convenient for the loop with significantly different complexities of iterations, since new tasks are assigned to idle processors. In this case, two-point data exchange is employed and collective operations are unnecessary. The corresponding algorithms become more complicated but the method remains generally clear.

By way of example, Fig. 1 demonstrates a fragment of the total text of such a parallel code. Control branch (with *rank* = 0) sends parameter for function `Func` or

```
int      size, rank;
int      N = 100000;
int      n, i, free;
double  *param;
double   f, p;
double   Sum = 0.0;
double   term = -1.0;
...
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
if (rank == 0) {     // Master node
   free = size-1;

   // Initial distribution
   for (n = 0; n < N && free > 0; n++) {
      MPI_Send(param + n, 1, MPI_DOUBLE, n + 1, 1, MPI_COMM_WORLD);
      free--;
   }

   // Free extra processors if any
   for (i = n + 1; free > 0; i++) {
      MPI_Send(&term, 1, MPI_DOUBLE, i, 1, MPI_COMM_WORLD);
      free--;
   }

   // Collection/distribution loop
   MPI_Status status;
   while (free < size-1) {
      MPI_Recv(&f, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
               2, MPI_COMM_WORLD, &status);
      i = status.MPI_SOURCE;
      Sum += f;
      if (n < N) {
         MPI_Send(param+n, 1, MPI_DOUBLE, i, 1, MPI_COMM_WORLD);
         n++;
      }
      else {
         MPI_Send(&term, 1, MPI_DOUBLE, i, 1, MPI_COMM_WORLD);
         free++;
      }
   }
}

else {               // Slave node(s)
   while (true) {
      MPI_Recv(&p, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, &status);
      if (p < 0)       // End-of-processing signal
         break;
      f = Func(p);
      MPI_Send(&f, 1, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD);
   }
}
```

Fig. 1. PL using the master/slave scheme.

Table 4. Deviation (in percents) from an ideal computation time for PL using the master/slave scheme

Number of processors	Model of nonuniformity of loop iterations				
	C	U	P	L	Q
2	100	100	100	100	100
4	33.5	33.5	33.4	33.5	33.4
8	14.5	14.6	14.3	14.4	14.4
16	6.86	6.76	6.76	6.84	6.85
32	3.30	3.36	3.30	3.28	3.28
64	1.64	1.66	1.71	1.67	1.70
128	0.85	0.83	0.90	0.86	0.96
256	0.43	0.44	0.55	0.47	0.71
512	0.25	0.29	0.53	0.30	0.47
1024	0.24	0.31	0.84	0.44	0.78

a specific value of −1 as a signal of termination to the idle working branch. These messages are sent with a tag of 1. The working branches wait for parameter (or signal of termination), call the basic function of the loop, and send the value of *f* to the root branch. These messages are sent with a tag of 2. The control branch performs the merging of the results of individual iterations (summation in the given example) up to the end of the loop. At the beginning, the control branch sends tasks to available working branches (and termination signal to the remaining branches if any), waits for the result from the branch, sends a new task, etc. up to the moment when all iterations have been executed.

Table 4 presents the characteristics of the PL algorithm of Fig. 1. It is seen that the elimination of collective operations and more efficient control of working branches lead to the PL efficiency that is lower than the ideal efficiency by less than 1% when the number of processors is no less than 128. However, the method is worse than the above methods when the number of processors is no greater than 64. As was expected, the parallelization quality weakly depends on the type of nonuniformity and is primarily determined by the number of processors. A further increase in the efficiency is possible due to reordering of iterations (see Section 4).

The above PL method must be implemented with allowance for the fact that the efficiency naturally decreases due to overloading of the control branch, which interacts with the working branches, at a relatively large number of processors (several or several tens of thousands) and/or complicated algorithm of the data merging. Note that the code of Fig. 1 can be easily transformed in such a way that two processors with ranks of 0 and 1 control working branches with even and odd numbers, respectively.

6. CONCLUSIONS

The proposed PL methods have been employed at the Laboratory of Mathematical Methods and Models in Bioinformatics, Institute for Information Transmission Problems, Russian Academy of Sciences in several problems [6–9] for which original parallel algorithms have been developed (see Laboratory web site http://lab6.iitp.ru for further details).

ACKNOWLEDGEMENTS

This study was supported by the Ministry of Education and Science of the Russian Federation, grant no. 8481.

REFERENCES

1. *The OpenMP API specification for parallel programming* http://openmp.org/wp/
2. *Message-passing Interface Forum. MPI Documents* http://www.mpi-forum.org/docs/docs.html
3. *CENTAUR Software Tools for Hybrid Supercomputing* http://centaur.botik.ru/home
4. *Joint Supercomputer Center of Russian Academy Science. Computing Systems* http://www.jscc.ru/scomputers.shtml
5. O. A. Zverkov, A. V. Seliverstov, and V. A. Lyubetskii, "Albuminous families typical of plastoms of small taxonomic groups of algas and protozoa," Molekulyar. Biologiya **46**, 799–809 (2012).
6. V. A. Lyubetsky, L. I. Rubanov, and A. V. Seliverstov, "Lack of conservation of bacterial type promoters in plastids of Streptophyta," Biology Direct **5** (34) (2010).
7. V. A. Lyubetsky, O. A. Zverkov, L. I. Rubanov, and A. V. Seliverstov, "Modeling RNA polymerase competition: the effect of σ-subunit knockout and heat shock on gene transcription level," Biology Direct **6** (3) (2011).
8. V. A. Lyubetsky, O. A. Zverkov, S. A. Pirogov, L. I. Rubanov, and A. V. Seliverstov, "Modeling RNA polymerase interaction in mitochondria of chordates," Biology Direct **7** (26) (2012).
9. V. A. Lyubetsky, L. I. Rubanov, L. Yu. Rusin, and K. Yu. Gorbunov, "Cubic time algorithms of amalgamating gene trees and building evolutionary scenarios," Biology Direct **7** (48) (2012).

Translated by A. Chikishev