CEUR-WS.org/Vol-2784/rpaper01.pdf

Parallel I/O and Checkpoints in DVM System

Valery Aleksakhin[1] [0000-0001-8385-8894], Vladimir Bakhtin[1] [0000-0003-0343-3859], Olga Zhukova[1] [0000-0002-1033-6371], Victor Krukov[1] [0000-0001-6630-964X], Olga Savitskaya[1][0000-0002-2174-3212]

[1] Keldysh Institute of Applied Mathematics, Miusskaya sq., 4, 125047, Moscow, Russia
dvm@keldysh.ru

Abstract. DVM-system is designed for the development of parallel programs of scientific and technical calculations in C-DVMH and Fortran-DVMH languages. These languages use a single parallel programming model (DVMH model) and are extensions of the standard C and Fortran languages with parallelism specifications, written in the form of directives to the compiler. The DVMH model makes it possible to create efficient parallel programs for heterogeneous computing clusters. The article presents new features of DVM system for working with checkpoints, which are based on the use of parallel I/O.

Keywords: automation of development of parallel programs, DVM-system, accelerator, GPU, Fortran, C, MPI, OpenMP, OpenACC, DVMH, I/O, checkpoint.

1 Introduction

Many tasks that are being solved today on modern computing systems require a large amount of data to be placed on external memory (disks) and intensive data exchange between disks and RAM. Functionally, these exchanges can be divided into the following groups:

1. Explicit I/O statements required to input initial data and output final results.
2. Checkpoints. For tasks with large execution times, it is necessary to periodically keep the state of the task. If the task is aborted for some reason, it can be continued from the last stored state.
3. Time steps. In some tasks, it is required to save the current state of the arrays at certain time intervals. For example, these data can be used in visualization subsystems.
4. Calculations with arrays on external memory. If some arrays do not fit in RAM (working external arrays), then the execution process can be organized as follows:

– External arrays are placed in files (disks).
– For each external array, a buffer (or several buffers) is allocated in the RAM.

Copyright © 2020 for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

– Use and update of the external array is performed by portions equal to the buffer size. The next chunk is read into the buffer, processed and written to the disk (if necessary).

Different groups of operations require different approaches to their implementation. For example, for checkpoints, first of all, it is necessary to reduce the read/write time of the checkpoint, the number of files, their format fall into the background. When saving data for their later visualization, on the contrary, it is needed to save the data in the required format. For calculations with arrays on external memory, it is possible to implement a mode when the next portion of data from the file will be read while using a previously read chunk of data that will significantly reduce the execution time of the program.

The desire to achieve maximum I/O performance and the lack of a common widely used parallel I/O interface has forced the developers of new programming languages, for example, UPC (Unified Parallel C), Chapel, XcalableMP to develop their own tools for parallel I/O. Such tools were also implemented in the DVM system.

It should be noted that at last time the checkpoints have become increasingly used by application programmers:

– when debugging parallel programs (for example, to detect an error that occurs after several hours of a program execution, a checkpoint may be saved before the error occurrence and then the program can be started repeatedly many times from this point under the control of the debugger);
– when starting a program on a supercomputer with a queue system, if the maximum quant of the time provided by the queue system is less than the
– for the development of reliable, fault-tolerant programs.

The last problem is becoming more and more actual. Modern supercomputers consist of tens of thousands of nodes, and each year this number is increased [1]. At the same time, the probability of failure of separate parts of the system or even the entire system is increased [2]. It requires to look for new approaches to the implementation of checkpoints.

This article presents the capabilities of the DVM system [3], which can be used in the development of fault-tolerant parallel programs with intensive I/O. The article is structured as follows. Chapter 2 discusses the various parallel I/O modes implemented for C-DVMH [4] programs. Chapter 3 presents the directives for parallel I/O in Fortran-DVMH [5]. Chapter 4 describes the new capabilities of the DVM system for working with checkpoints. In Chapter 5 an example of these capabilities usage is shown, and the efficiency of the implemented I/O subsystem is analyzed.

2 Parallel I/O in C-DVMH

The DVM system supports different I/O modes:

1. Serial synchronous I/O.
2. Parallel synchronous I/O:
    a. to a local file
    b. to a parallel file.

3. Asynchronous parallel I/O.

In C-DVMH, the following I/O operations are implemented in full compliance with the C99 standard: remove, rename, tmpfile, tmpnam, fclose, fflush, fopen, freopen, setbuf, setvbuf, fgetc, fgets, fputc, fputs, getc, getchar, gets, putc, putchar, puts, ungetc, fread, fwrite, fgetpos, fseek, fsetpos, ftell, rewind, clearerr, feof, ferror.
Consider some details of the implementation of different I/O modes.

Serial synchronous I/O is performed as follows:

1. The file is opened by only one process from the current multiprocessor system (I/O process).
2. All operations on the file are performed only by the I/O process.
3. When input/output operation of a distributed array is performed, read or write is done by portions of about a 100МБ, then the data are sent to processes or accumulated from processes that are owners of this data.
4. If the operation involves writing user variables or returning a value, the data are sent by the I/O process.
5. If during the execution of the operation an error occurs and the standard prescribes to set errno, then its value is also sent by the I/O process.
6. All operations on a file can be performed only collectively by all processes of the multiprocessor system which created this file.
7. The operations not on the file descriptors (rename, remove, tmpnam) are performed by I/O process of the current multiprocessor system which sends the execution results to all other processes.

To implement **parallel synchronous I/O**, two new modes for fopen and freopen functions were added: local file and parallel file.

The local file is opened independently by each process. All operations on local files are processed by each process independently and do not cause any communications.

When reading (or writing) a distributed array from the **local file**, each process reads (or writes) only the local part of the distributed array. Modern parallel file systems are focused on efficient work with a large number of files. Thus, the work with local files is a fairly convenient means of parallel I/O of distributed data, it is performed efficiently, but requires the coincidence of the data distribution during recording and next reading. The DVMH-program [6], which works with the local files, must run on the same processor grid as the DVMH program that recorded this data.

The main idea of working with **parallel files** is as follows. Let's we have a 4 gigabyte file that stores a two-dimensional distributed array. Let the DVMH-program was launched on 4 processes. In this case, each process is responsible for reading of the ¼ part of the file. Such reading can be performed in parallel. The 1-st process reads the 1-st GB of the file, the 2-nd process reads the 2-nd GB of the file, etc. After reading its own part of the file, the process can send this data to other processes. If the array in the program is distributed by columns (there is a subset of columns of the distributed array on each processor), then the process will leave ¼ part of the read information to itself, and ¾ of the read data will send to other processes ("foreign" columns). But if the array is distributed by rows, then no data passing is required, since each process will read only "own" rows of the array.

Thus, the parallel file is opened by each process of the multiprocessor system. All operations on parallel files are performed in the same way as on ordinary files, with the exception of fread, fwrite, futs operations. Consider the algorithm for implementation of these operations:

1. To synchronize on the file system all operations from the I/O process (since only it performs all other operations). To synchronize the contents of the file in memory with the contents on the disk, the system call _commit on Windows or fdatasync on Linux are used.
2. To pass the position in the file from the I/O process to all other processes of the current multiprocessor system.
3. Each process sets the position in the file to that point from which it will read (or write).
4. Each process performs its part of the work.
   a. If a read operation to replicated variable is requested, then after reading a part, the read chunks are combined (the operation of MPI_Allgather type).
   b. If an operation with distributed array is requested, then each process accumulates the data it needs from all other processes before writing, or after reading its own part of the file sends it to other processes (the operation of MPI_Alltoall type).
5. All processes synchronize their updates to the file system.
6. The I/O process sets the position behind the last read (or write) chunk.

This mode saves the contents of the file as if it were written by a serial program. It also allows to read the files written by a program executed on any number of processes (including the original serial one).

To implement asynchronous I/O, additional I/O threads are created in each of processes. All operations on one file are performed sequentially by one thread. The operations on different files can be performed in parallel by different threads. Asynchronous I/O mode is based on usage of additional memory. During writing a distributed array to a file, a copy of that array is created in the CPU RAM, a task is created to write the array-copy to the file, one of the I/O threads begins to perform this task, and after it the control is returned to the user program. Then, I/O and calculations can be performed simultaneously - the program updates the distributed array, and saved values from the array-copy are written to the file. Such simultaneous execution is possible only for I/O functions that do not return values and do not set errno (a variable that stores the integer code of the last error). If the program uses the result of I/O operation, for example, the number of elements that have been successfully written to the file, this results in waiting for the completion of the operation.

It should be noted that only a small part of standard I/O functions do not return values or do not set errno. These are rewind and clearerr functions. But these values aren’t used in most of user programs. For example, instead of the statement:

n_elem = fwrite (B, sizeof (double), L * L, f)

the statement:

fwrite (B, sizeof (double), L * L, f) is used.

To implement **asynchronous I/O**, additional versions that do not return values have been introduced in the runtime system: fprintf, fscanf, printf, scanf, vfprintf, vfscanf, vprintf, fgets, fputc, futs, gets, putc, putchar, puts, ungetc, fread, fwrite, fseek. They also do not set errno, and it means that the result of their work is only either writing to a file the data passed to the function, or reading this data from the file. An optimization was also introduced in the C-DVMH compiler, which recognizes for this set of functions whether their return value is used and if the value is not used, a call of the function variant not returned value is generated.

To implement asynchronous I/O in the C-DVMH compiler, for the fopen and freopen functions a new mode was added: asynchronous file.

Both usual files (non-local and non-parallel) and local or parallel files (the file cannot be local and parallel at the same time) can be asynchronous.

As already noted, the operations on different files may overlap (were performed by different I/O threads). Any operation that does not allow asynchronous execution results in waiting for all asynchronous operations on the corresponding file before starting the synchronous operation.

Also, some operations (those that require communications) cannot be asynchronous if the MPI implementation does not provide parallel work with it (MPI_THREAD_MULTIPLE mode is required).

To serialize asynchronous operations on each file, the mechanism of dependent tasks and parallel execution of the task graph is used.

Thus, the DVM-system supports several different I/O modes. The I/O mode is set when the file is opened. To do this, the set of file open modes was extended for fopen and freopen functions. If to add the letter "l" or "L" to the file open mode, each processor opens its own local file and all operations are performed by each processor independently from the others. When opening in this mode the '% d' construct can be used in the filename, to specify different names for different processors.

The following statement:

FILE *f = fopen("out_%04d.txt", "wl");

will open N files for write (N is the number of processors in the current multiprocessor system) with the names out_0000.txt, out_0001.txt, out_0002.txt, out_0003.txt...

The all group of the files can be deleted by a special function:

int dvmh_remove_local(const char *filename);

If to add a letter "p" or "P" to the file open mode, all processors will perform parallel input/output to the global file.

Adding a letter "s" or "S" enables asynchronous I/O mode.

Table 1 lists the environment variables that affect I/O performance.

Table 1. Environment variables that affect I/O execution

DVMH_PARALLEL_IO_THRES	positive integer specifying the minimum size of input or output data (in bytes) that the runtime system will attempt to input or output in parallel. The default value is 104857600,

	i.e. 100 MB.
DVMH_IO_BUF_SIZE	positive integer that specifies the maximum size of the I/O buffer when I/O is performed by the I/O processor. The default value is 104857600, i.e. 100 MB.
DVMH_IO_THREAD_COUNT	non-negative integer that specifies a number of I/O threads performing asynchronous I/O operations. The zero value will disable asynchronous I/O. The default value is 5.

3 Parallel I/O in Fortran-DVMH

Prior to the implementation of the parallel I/O subsystem for C-DVMH programs, I/O in Fortran-DVMH programs was performed sequentially and synchronously. A special I/O process performed all file operations, carried out the necessary sending of read data and accumulated the information necessary for recording from all processes. Fortran 95 operators were used for I/O.

There were serious restrictions for I/O of distributed arrays in the Fortran-DVMH language:

- The I/O list had to consist of only one distributed array name and could not contain other I/O objects.
- In format I/O statements only the format specified by '*' was allowed.
- The control information list should not contain ERR, END, or IOSTAT parameters.
- Only replicated variables were allowed in the control information list.
- It was not allowed to use I/O statements for distributed arrays in a parallel loop.

It should be noted that a Fortran-DVMH program executing unformatted I/O of the distributed arrays was not compatible in general with the serial program in Fortran 95. Data that were written by one program couldn't be read by other program, due to a difference in lengths of records. For example, the serial program could write entire array using a single operation, but the parallel program accumulated data in the special I/O buffer that could be flushed to the file several times (as information was accumulated from various processes).

Because of this, the usage of parallel I/O tools implemented for C-DVMH programs in the Fortran-DVMH compiler did not cause any additional problems.

In the DVMH runtime system there were created new functions that are a kind of adapters between Fortran I/O operators and C-DVMH I/O functions. A new version of the Fortran-DVMH compiler has been developed, that uses these functions-adapters.

To set the Fortran-DVMH I/O mode a new directive has been introduced:

!DVM$ IO_MODE ([PARALLEL]

[[,]LOCAL]
[[,]ASYNC])

This directive can be placed before a file open statement and it controls all next I/O operations to this file (unit). If there is no the directive before the file open statement, I/O is performed according to the old scheme (through the I/O processor). The LOCAL, PARALLEL, and ASYNC specifications correspond to the modes that were discussed in the previous chapter.

Figure 1 shows a program fragment that demonstrates the new capabilities of the Fortran-DVMH language: I/O mode setting; operating with distributed array sections, usage of several distributed arrays in a single I/O statement; support for such control parameters as ERR, END, that specify the statement in the program to switch to if an error occurs or the end of the file is reached.

```
        PARAMETER (L=4096)
        FLOAT A(L,L), B(L,L)
!DVM$   DISTRIBUTE ( BLOCK, BLOCK) :: B
!DVM$   ALIGN A(I,J) WITH B(I,J)
        …
!DVM$   IO_MODE (LOCAL,ASYNC)
        OPEN(4, ACCESS='STREAM', FILE='DATA.DAT', ERR=77)
        …
        WRITE(4) A(3:L-2,2:L-1),B
        …
        CLOSE(4)
        …
77:     PRINT *, 'ERROR HAPPENED! PROGRAM TERMINATES'
        STOP
```

Fig. 1. I/O in Fortran-DVMH program.

4 Mechanism for working with checkpoints

The parallel I/O modes described in the previous sections allow to implement effective work with checkpoints in DVMH programs. For example, the use of local files allows each process to quickly write/read its part of a distributed array; the use of asynchronous I/O allows to write a checkpoint simultaneously with execution of program's computational statements.

However, the usage of these I/O operators leads a serious complication of the source program code, adding many new statements when working with checkpoints:

- check existence of data file;
- check the mode in which the file was recorded;
- for a local file, check the number of processors and the processor grid that was used when writing the file;
- check the correctness of the read data. For example, to check same sizes of the array that was saved and the array that was created after the program restart;

– implement a file interleaving mechanism. For example, to write data alternately to file1.dat and file2.dat files so that in case of failure when saving the next checkpoint, at least one of the files has the correct value;
– implement the overlapping of checkpoint recording and a program execution;
– and much more.

A programmer is forced to repeat the described above logic of working with checkpoints in each DVMH program. To simplify the work with checkpoints, new specifications of parallelism have been developed in the DVM system.

Checkpoint definition directive:

```
!DVM$ CP_CREATE cp-name, VARLIST(cp-var-list), FILES(filenames) [, mode]
   cp-name ::= string-expr
   cp-var ::= subarray | variable
   filenames ::= filename-list | filename-array
   filename ::= string-expr
   filename-array ::= string-array
   mode ::= PARALLEL | LOCAL
```

The directive specifies the name of the checkpoint (cp_name), a list of data for saving and reading (VARLIST argument), the list of the files to be used (FILENAMES argument), and the mode of file using (mode). The mode can be local or parallel. The asynchrony of the checkpoint is specified directly in the save statement itself. If mode is not specified, the parallel mode is used by default.

The example of using the CP_CREATE directive:

```
      INTEGER I
      INTEGER, DIMENSION(1:N) ARR
!DVM$ DISTRIBUTE ARR(BLOCK)
!DVM$ CP_CREATE CP1, VARLIST(I, ARR), FILES('file1.dat','file2.dat'), LOCAL
```

In this example, a checkpoint named CP1 is declared. The files, variables, and local open mode are set for it.

To **save the checkpoint**, the CP_SAVE directive is used. It is enough to specify the name of the checkpoint, and optionally specify the asynchrony of this operation in the directive.

The syntax of the directive is as follows:

```
!DVM$ CP_SAVE cp-name [, ASYNC]
```

For example, when executing the directive:

```
!DVM$ CP_SAVE CP1, ASYNC
```

and assuming that the checkpoint CP1 has been defined as above, the scalar variable I and the distributed array ARR will be written to file1.dat or file2.dat file. The file will be selected depending on which of them was last recorded successfully. The file open mode is local, so only the local part of the array ARR will be written to it.

To **load the checkpoint** the CP_LOAD directive is used. Only the checkpoint name is specified in it. The syntax of the directive is as follows:

```
!DVM$ CP_LOAD cp-name
```

For example, when executing the directive:
!DVM$ CP_LOAD CP1

and assuming that the checkpoint CP1 has been defined as above, the scalar variable I and the local part of the array ARR will be loaded from the file1.dat file, or from the file2.dat file. The last correctly recorded file will be selected automatically.

To **wait for the asynchronous checkpoint to be saved** the CP_WAIT directive is used:
!DVM$ CP_WAIT cp-name, STATUS(status-var)
status-var ::= int-variable

For example, when executing the directive:
!DVM$ CP_WAIT CP1, STATUS(st)
and assuming that the checkpoint CP1 has been defined as above, the program waits for the end of the asynchronous write to file1.dat and file2.dat files, if it was, and closes them. One of the following values will be written to the variable st passed as an argument of the STATUS parameter: 0 – asynchronous writing is finished successfully; a non-zero value – if some error occurred.

An experimental version of the DVM system has been developed that supports these specifications for working with checkpoints.

5 Testing the approach

Several test programs have been developed to examine the efficiency of the implemented I/O subsystem, as well as the checkpoints mechanism. One of them, implementing the Jacobi iterative algorithm, is shown in Figure 2. This test simulates the recording of a checkpoint (distributed array B) that is performed every 10 iterations.

```
      PROGRAM JAC2D
      PARAMETER (L=32000, ITMAX=100)
      REAL A(L, L), B(L, L), EPS
      INTEGER ST
!     arrays A and B  with block distribution
!DVM$ DISTRIBUTE(BLOCK, BLOCK) :: A
!DVM$ ALIGN B(I, J) WITH A(I, J)
!DVM$ CP_CREATE CP1,VARLIST(B),FILES('file1.dat','file2.dat'),LOCAL
      MAXEPS = 0.5
      DO IT = 1, ITMAX
         EPS = 0.
         IF (MOD(IT,10) .EQ. 0) THEN
!DVM$        CP_SAVE CP1, ASYNC
             CONTINUE
         ENDIF
!        variable EPS is used for calculation of maximum value
!DVM$    PARALLEL(J, I) ON A(I, J), REDUCTION(MAX(EPS))
         DO J = 2, L - 1
            DO I = 2, L - 1
               EPS = MAX(EPS, ABS(B(I, J) - A(I, J)))
```

```
               A(I, J) = B(I, J)
            ENDDO
         ENDDO
!        Copying shadow elements of array A from
!        neighbouring processors before loop execution
!DVM$     PARALLEL(J, I) ON B(I, J), SHADOW_RENEW(A)
          DO J = 2, L - 1
             DO I = 2, L - 1
               B(I, J) = (A(I,J-1)+A(I-1,J)+A(I+1,J)+A(I,J+1))/4.
             ENDDO
          ENDDO
          PRINT 200, IT, EPS
200       FORMAT (' IT = ', I4, '   EPS = ', E14.7)
        ENDDO
!DVM$   CP_WAIT CP1, STATUS(ST)
        END
```

Fig. 2. Fragment of the iterative Jacobi method in Fortran-DVMH

Figure 3 shows the execution times of the Jacobi program on 1 node of the Lomonosov supercomputer when using different I/O modes. During this experiment, 2 MPI processes by 6 threads were started at the node, PARALLEL, LOCAL and asynchronous/local (ASYNCHRONOUS) modes of the checkpoint saving were tested. For comparison, there is given the time of the array saving, using the old scheme through the I/O processor (OLD).

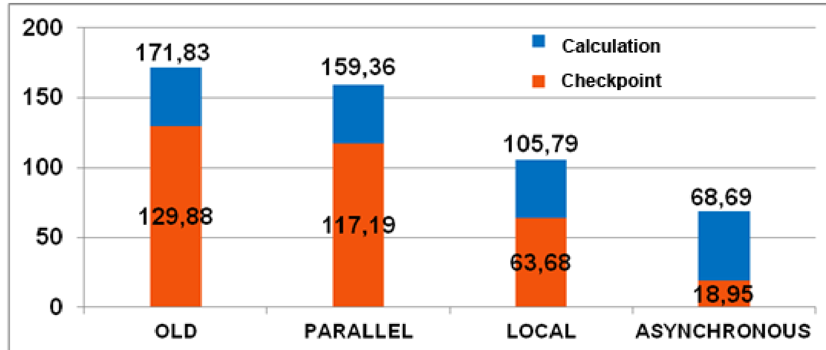

Fig. 3. The execution times of Jacobi's program on the Lomonosov supercomputer

Using the local mode for this experiment made it possible to reduce the time of the checkpoint saving by almost 2 times (in this case, each process writes its own part of the array in the file) in comparison with the serial mode (in this case, single I/O process saves the entire array in the file, plus the time is required to pass part of the array from one process to the I/O process). Using asynchronous mode reduces the I/O execution time by almost 7 times from 128.99 seconds to 18.95 seconds due to overlapping I/O and calculations (during execution of next 10 iterations of the algorithm, the previous checkpoint has time to be written in the file). All asynchronous writes are buffered. For this experiment, the execution time of 10 buffering operations of the array B (memory-to-memory copy) is almost 18 seconds.

Conclusions and Outlook

Inefficient I/O management negates any optimizations of parallel execution for the tasks with intensive I/O. The new features presented in the article allow to increase the efficiency of I/O execution for DVMH programs.

One of the advantages of the developed parallel I/O tools is the ease of use. In his application program a programmer uses the usual I/O operators (C or Fortran) of serial programming language, switching between modes is carried out by changing only one parameter of the fopen and freopen functions (for the C-DVMH language) or using the IO_MODE directive (for the Fortran-DVMH language).

The developed I/O tools are universal - they can be used to save checkpoints, and to save data obtained at the next time step of the algorithm (for example, for visualization), and for calculations with arrays on external memory.

A new version of SAPFOR [7,8] system which automates the development of parallel programs in the DVMH model is currently being developed. In this version automatic checkpoint placement will be implemented. The system will automatically identify the variables that need to be saved/restored in some point and generate the necessary directives.

References

1. TOP500 supercomputers.: https://www.top500.org/, last accessed 2020/11/25.
2. Bondarenko, A.A., Yakobovskii, M.V.: Obespechenie otkazoustojchivosti vysokoproizvoditelnyh vychislenij s pomoshchyu lokalnyh kontrolnyh tochek. Vestn. YUUrGU. Ser. Vych. matem. inform., 3(3), 20–36 (2014).
3. System for automating the development of parallel programs (DVM-system). http://dvm-system.org, last accessed 2020/11/25.
4. C-DVMH language, C-DVMH compiler, compilation, execution and debugging of DVMH programs. http://dvm-system.org/static_data/docs/CDVMH-reference-en.pdf, last accessed 2020/11/25.
5. Fortran DVMH language, Fortran DVMH compiler, compilation, execution and debugging of DVMH programs. URL: http://dvm-system.org/static_data/docs/FDVMH-user-guide-en.pdf, last accessed 2020/11/25.
6. Bakhtin, V.A., Klinov, M.S., Kriukov, V.A., Podderiugina, N.V., Pritula, M.N., Sazanov, Iu.L.: Rasshirenie DVM-modeli parallelnogo programmirovaniia dlia klasterov s geterogennymi uzlami. Vestnik Iuzhno-Uralskogo gosudarstvennogo universiteta, seriia "Matematicheskoe modelirovanie i programmirovanie", 8 (277)(12), 82–92 (2012).
7. Klinov, M.S., Kriukov, V.A.: Avtomaticheskoe rasparallelivanie Fortran-programm. Otobrazhenie na klaster. Vestnik Nizhegorodskogo universiteta im. N.I. Lobachevskogo, (2), 128–134 (2009).
8. Kataev, N.A., Kolganov, A.S., Smirnov, A.S.: Podderzhka interaktivnosti v sisteme SAPFOR. Nauchnyi servis v seti Internet: trudy XIX Vserossiiskoi nauchnoi konferentsii (18–23 sentiabria 2017 g., g. Novorossiisk). M.: IPM im. M.V. Keldysha, 2017. P. 243–249. http://keldysh.ru/abrau/2017/57.pdf doi:10.20948/abrau-2017-57.