Comparative Debugging of Parallel DVMH-programs

Vladimir Bakhtin[1,2][0000-0003-0343-3859], Dmitry Zakharov[1][0000-0002-6319-5090] , Alexander Ermichev[1,2][0000-0002-3678-2580] and Viktor Krukov[1,2][0000-0001-6630-964X]

[1] Keldysh Institute of Applied Mathematics, Miusskaya sq., 4, 125047, Moscow, Russia
[2] Lomonosov Moscow State University, GSP-1, Leninskie Gory, 119991, 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, in the nodes of which accelerators (graphic processors or Intel Xeon Phi coprocessors) can be used as computing devices along with universal multi-core processors. The article describes the method of debugging parallel programs in DVM-system, as well as new features of DVM-debugger.

Keywords: Automation of Development of Parallel Programs, Automation of Debugging of Parallel Programs, DVM-system, Accelerator, GPU, Fortran, C.

1 Introduction

Classic debugging approaches may not always be helpful when it comes to parallel programming. A step-by-step study of the executing process at certain breakpoints or debug prints are hardly applicable to real scientific and engineering parallel software systems designed for continuous execution for hours or even days.

Parallel algorithms are usually much more complicated than sequential solutions of the same problems. Moreover, the parallel code may contain atypical for sequential debugging errors associated with the incorrect use of synchronization primitives and functions that provide parallelism.

The following factors affect the complexity of debugging parallel programs:

- the necessity to track the status of several (or even many) parallel processes/threads;
- the difficulty of reproducing errors caused by non-deterministic execution;
- the debugging tools influence on the execution process (different execution time of the operators, additional internal synchronizations).

To debug parallel programs, automated methods have been developed. Such methods allow to find most errors in the program in automatic mode with minimal involvement of the programmer. One of these methods is the dynamic correctness control. This method is used in many debugging tools for multithreaded programs, such as: Helgrind [1], DRD [2], Intel Parallel Inspector [3]. In the DVM-system, this method is used to debug parallel programs made for heterogeneous clusters.

Another automated debugging method is the comparative debugging of parallel programs. The main principle of this method is to compare the execution process of two programs, by controlling the values of variables at certain controlled points. Comparison can be carried out either between simultaneously running programs, or using trace files, which store all the necessary data about operations and variable values at controlled points. This debugging method was implemented in the Guard [4] and Wizard [5] debuggers. The term "comparative debugging" was introduced in the articles describing these tools. Around the same time, this method was also implemented in the debugger of the DVM-system.

In the Guard and Wizard debuggers, the controlled points at which the variable values are compared are setting by the user. In contrast, in the DVM-debugger, comparison points are setting automatically.

This article briefly describes the approaches for DVMH-programs debugging, pictures the problems that had arose during the use of DVM-system comparative debugging implementation, and suggests new methods to overcome these problems.

2 Approaches for Debugging Parallel DVMH programs

To identify errors that lead to incorrect calculations, the DVM-system [6-8] contains various special tools for automating the debugging process. Such errors can be detected by executing a DVMH-program in the dynamic correctness control mode and/or by starting computations on one or several processors in a comparing mode with the reference results obtained during its sequential execution.

The use of automated debugging methods requires the instrumentation of a parallel program: insertion of special calls to the debugger that allow to control and process execution of a program. There are two approaches for such instrumentation: binary code instrumentation and source code instrumentation. Helgrind, DRD, Intel Parallel Inspector are based on binary instrumentation. But the use of high-level programming models (such as OpenMP [9]), can lead to certain difficulties with that approach.

For example, instrumentation utility must restore the parallelism specifications from binary code of the program in order to be able to generate errors in the context of the source code of that program. To make this possible, the instrumentation utility must know the internal logic of the compiler. If both compiler and debugger are made by the same developer (e.g. Intel), then such recovery can be implemented. But when the compiler is developed by one company (e.g. Microsoft), and the debugger is by another (e.g. Intel), then recovering of parallelism specifications can be difficult. This problem can be avoided by using the source code instrumentation.

Debugging instrumentation of programs in DVM-system is performed by C-DVMH [6] and Fortran-DVMH [7] compilers, which add calls to DVM-debugger functions at the following points of the program:

- beginning of a sequential or parallel cycle;
- completion of a sequential or parallel cycle;
- beginning of a new iteration of a cycle;
- accessing a variable for reading;
- before accessing a variable for writing;
- after accessing a variable for writing.

In addition, the debugger functions are also performed while executing calls to the Lib-DVMH library [8].

Dynamic correctness control of DVMH-directives is based on an analysis of the sequence of calls to Lib-DVMH functions and accesses to variables. Dynamic control is allowing to detect the following types of errors:

- undeclared data dependency in a parallel loop;
- incorrect use of private and reduction variables;
- undeclared access to non-local elements of a distributed array;
- incorrect work with shadow edges of reduction arrays modification of non-local elements of a distributed array in the sequential part of the program;
- going out of bounds in distributed array;
- writing data to the remote access buffer.

It should be mentioned that not all errors can be determined by the dynamic correctness control. For example, external procedures and functions without any source code cannot be analyzed. The comparative debugging method can be used in order to check such programs. This method will be discussed in detail in the next section.

3 Comparative Debugging of DVMH-programs

The intermediate results comparison implemented in DVM-system allows to detect errors in parallel programs that arise due to incorrect DVMH instructions, as well as program errors that do not appear in sequential execution and were not detected by the dynamic control method.

The general scheme of comparative debugging method is as follows:

1. Getting a reference trace (*./dvm trc* command in console interface). Trace contains: readings and modifications of variables, the beginning of each loop iteration, the beginning and end of parallel and sequential loops, the beginning of each parallel task, the beginning and end of the task group. The source for reference trace can be either the sequential execution of the same program or parallel execution (e.g. the trace recorded on the different computing cluster where the error does not occur);
2. Automatic comparison of program execution results with previously accumulated reference trace (*./dvm dif* command). All integer numbers is compared strictly and real numbers are compared with a given absolute

and relative accuracy. Information on the all discrepancies is reported to user.

```
       <trace context info (OS architecture, working directories)>
       MODE = <NONE | MINIMAL | MODIFY | FULL>
       <SL | PL | TR> <structure number> (<parent structure number>)
[<loop rank>] {<file>, <line>} = <NONE | MINIMAL | MODIFY | FULL>,
(<dimension>:<first iteration>, <last iteration>, <loop step>), …
       EL: <structure number>
       …
       <SL | PL | TR> <structure number> (<parent structure number>)
[<loop rank>] {<file>, <line>} = <NONE | MINIMAL | MODIFY | FULL>,
(<dimension>:<first iteration>, <last iteration>, <loop step>), …
       EL: <structure number>
       END_HEADER
```

Fig. 1. Trace header structure

Reference trace file is created for each process and has a text format. This file starts with the special header that contains used trace parameters and hierarchy of loops and task groups (see Fig. 1).

```
       - reading variable:
       RD: [<type of variable>] <name of variable> = <value>; {<file>,
<line>}

       - before modification of variable:
       BW: [<type of variable>] <name of variable>; {<file>, <line>}

       - after modification of variable:
       AW: [<type of variable>] <name of variable> = <value>; {<file>,
<line>}

       - reading reduction variable:
       RV_RD: [<type of variable>] <name of variable> = <value>;
{<file>, <line>}

       - before modification of reduction variable:
       RV_BW: [<type of variable>] <name of variable>; {<file>,
<line>}

       - after modification of reduction variable:
       RV_AW: [<type of variable>] <name of variable> = <value>;
{<file>, <line>}

       - final result of reduction operation:
       RV: [<type of variable>] <value>; {<file>, <line>}
```

Fig. 2. Trace records format (variables)

```
       - beginning of parallel loop:
       PL: <structure number> (<parent structure number>) [<loop
rank>]; {<file>, <line>}

       - beginning of sequential loop:
       SL: <structure number> (<parent structure number>) [<loop rank
(always equals 1)>]; {<file>, <line>}

       - beginning of task group:
TR: <structure number> (<parent structure number>) [<group rank (always
equals 1)>]; {<file>, <line>}

       - next loop iteration of parallel task:
       IT: <absolute index of iteration (calculated from values of all
index variables of the cycle) or task number>, (<value of 1st index
variable>,< value of 2nd index variable>,…).

       - end of loop (parallel or sequential):
       EL: <structure number>; {<file>, <line>}

       - end of the local calculations block in the sequential part of
the program:
       SKP: {<file>, <line>}
```

Fig. 3. Trace records format (loops and parallel tasks)

Global trace details level can be set by modifying the value of the *MODE* parameter in the trace header. Details level can also be specified for each cycle and task group of the DVMH-program by setting the corresponding mode in the line that describes the desired structure in the header.

The main body of the trace contains a sequence of specific records. These records determine the dynamic structure of the program: the sequence of statements during the concrete execution of the program. Fig. 2 and 3 provide a complete list of events that are recorded in the trace.

```
#define L 3
int main(int an, char **as)
{
    #pragma dvm array distribute[block]
    double A[L];
    #pragma dvm parallel([i] on A[i])
    for (int i = 0; i < L; i++)
    {
        if (i == 0 || i == L - 1)
            A[i] = 0;
        else
            A[i] = 2 + i;
    }
    return 0;
}
```

Fig. 4. C-DVMH example program (file “test.c”)

Figure 5 shows the recorded reference trace of the example program (code of which is shown in Fig. 4). This program consists of one parallel loop, which starts on the line 7 of the "test.c" file and initializes the elements of the distributed array A. As a result of this loop, the elements A[0] and A[2] are set to "0", and A[1] becomes "3".

```
# Begin trace header. Don't modify these records
TRACE_TIME = "Mon Apr  15 00:00:22 2019"
ARCHITECTURE = "Machine x86_64"
USER_HOST = "DVM-COREI7@DVM-COREI7"
WORK_DIR = "/home/DVM/dvm_current/dvm_sys/demo"
TASK_NAME = "test"
MODE = FULL
PL: 1() [1] {"test.c", 7} = #, (0:0,2,1)
EL: 1
END_HEADER
# End trace header
PL: 1() [1]; {"test.c", 7}, 1.B
  IT: 0, (0)
  BW: [4] "A[i]"; {"test.c", 10}
  AW: [4] "A[i]" = 0; {"test.c", 10}
  IT: 1, (1)
  BW: [4] "A[i]"; {"test.c", 12}
  AW: [4] "A[i]" = 3; {"test.c", 12}
  IT: 2, (2)
  BW: [4] "A[i]"; {"test.c", 10}
  AW: [4] "A[i]" = 0; {"test.c", 10}
EL: 1; {"test.c", 7}, 1.E
END_TRACE
```

Fig. 5. Reference trace of example program

Comparative debugging proved to be effective on simple model problems, but came across two obstacles: resources and accuracy. Next section describes the essence of these problems.

4 Issues with the Current Comparative Debugger

After instrumentation, each operator of debugged program is surrounded by several calls to debugging library, so it is not surprising that execution time can increase significantly. For example, in experiments with benchmark programs from the NAS NPB package [11], the following results were obtained:

- the slowdown only from debugging instrumentation (i.e. the program was compiled for debugging, but executed without collecting/comparing the trace) was around 50–100 times;
- trace file size (several dozens of bytes for each traced operator) turned out to be completely unacceptable for real programs. The average size of full trace for benchmark program was measured in terabytes, and the approx-

imate time for collecting the trace is ~28000 sec. (with an average runtime of the initial programs of ~5 sec., i.e., a ~4000 times deceleration).

Therefore, full comparative debugging turned out to be applicable only on small "model" data, which are not always available. To overcome this problem, the following control and optimization tools were developed and implemented in the DVM-system debugger [12]:

- selective tracing: only modifications, only distributed arrays, etc. (compilation options *-d1...-d4*);
- local instrumentation, i.e. manual selection of program sections, that will be instrumented for debugging (directives *DEBUG <debug mode> / END DEBUG*);
- preliminary estimation of the trace size by obtaining the trace header, and its manual correction to disable tracing of certain loops or iterations (command *./dvm size*);
- automatic selection of parallel loops iterations for tracing: “borders” and “corners” (compilation options *-dbif1* and *-dbif2*);
- generation of two cycle bodies: one without debugger calls, the other is instrumented; the instrumented loop body is used only on selected iterations, and the original program is executed on the rest;
- recording the checksums of arrays at the end of the loop instead of tracing modifications of their elements in the body of the loop (parameter *TraceOptions.CalcChecksums*).

Described improvements extend the possibility of applying comparative debugging to real applications. For example, when processing large amounts of data through loops, the most probable place for errors occurrence (going out of the array bounds, uninitialized variables) will be boundary iterations. Moreover, the usual structure of computational algorithms is causes that an error occurred at the internal iteration of the loop will propagate to the boundaries. For such tasks, selective tracing and comparing of boundary iterations can be used (*-dbif<level>* options), which allows (see Table 1):

- significantly reduce the size of the trace (100-1000 times);
- significantly reduce the execution time (10-1000 times);
- preserve most coverage of program operators (over 99%).

Table 1. Average trace size and execution time for NAS NPB benchmarks (class A)

	Full trace	**"Corners" width = 1**	**"Corners" width = 2**	**"Borders" width = 1**	**"Borders" width = 2**
Average coverage	100%	99,4%	99,8%	99,8%	99,8%
Average trace size, bytes	6,57E+12 (~6 Tb)	4,6E+07 (~44 Mb)	9,4E+08 (~894 Mb)	1,7E+10 (~16 Gb)	6,0E+10 (~56 Gb)

Average execution time, sec	27915 (7,75 h.)	15	19	105	287

But, despite all optimizations, use of comparative debugging for scientific and technical algorithms, especially on real data (rather than on artificially created small "model" tests) is still very restricted.

Another problem detected during the practical usage of the debugging system was the “allowable” mismatch of the real variables values. Most notable case was with reduction variables. The calculation of the sum of the distributed array elements changes depending on the the parallel program configuration: firstly local sums are calculated, and then they are combined in non-deterministic order. Results of such computations are slightly different (usually in the 1-2 minor digits), but, from the user point of view, resulting values are be equally valid. Reduction operations create four problems for comparative debugging (“false alarms”):

- the intermediate values of the reduction variable do not coincide at all, because only partial sums (maximum values, etc.) are calculated;
- the final value of variable, as mentioned above, may slightly differ (non-determinism in the weak sense);
- the difference in one variable can immediately spread to many others (for example, if the norm of a vector is calculated and then the vector is normalized);
- if the variable, that was influenced by the difference in the reduction results, is used as the condition for ending iterations or choosing a branch of calculations, then comparing programs can diverge at that condition (non-determinism in the strong sense).

These problems were solved by introducing a special reduction processing mode:

- reduction variables are recognized (because they are described in DVMH-directives), and trace records of their reading and modification in the body of the loop are ignored by comparative debugger;
- at the end of the parallel cycle, the final value of the reduction variable (*RV* trace record) is added into the trace for further comparison;
- the values of the reduction variables are compared with some accuracy, which may be less than the accuracy of comparing the values of ordinary variables. This accuracy can be set by the programmer through a special configuration parameter;
- during the comparison the actual calculated value of the reduction variable (after successful comparison with the given accuracy) is replaced by its value from the “reference” trace to eliminate the potentially dangerous discrepancy.

The described approach made it possible to suppress the “false alarms” associated with reduction variables. However, it was further discovered that this problem can appear not only on reduction computations. Executing a program on different machines or using different compilation tools can leads to different results of any arithmetic operation, such as multiplication or division (in 1–2 minor decimal digits of the mantissa) [13].

5 New Approaches to Comparative Debugging

Currently, a new version of the comparative debugging system is in development [14], aimed to overcome described problems. The new debugger will be based not on trace files, but on the exchange of debug information between two simultaneously running instances of the program:

1. program execution will be divided into computational blocks, the order of which is deterministic for any parallel system configuration;
2. during the execution of the next block, each instance of the debugged program will be accumulating needed trace records;
3. upon completion of the block, the accumulated traces will be sent to one process for comparison.

The DVMH parallel model is making it possible to split any program into a deterministic sequence of blocks split by the boundaries of parallel loops.

Proposed extension will support all previously implemented optimizations of the trace size (array checksums and boundary iterations), since all changes made by these optimizations affect the trace locally, within a specific parallel cycle, and therefore, within one specific block of the new trace.

Described approach makes comparative debugging more flexible, allowing the simultaneous launch of the reference and debugged programs on one multiprocessor machine and remote debugging – comparing the execution of the reference program on one machine with an experimental version running on another.

Splitting a parallel DVMH program into deterministic blocks has one more advantage – the values of all variables at the block boundary should be identical regardless of the number of processes and their configuration. Therefore, the block boundaries can be used as controlled points for comparative debugging, instead of tracing each operation of reading and modifying variables. In this case, all calculations within a certain block will be carried out without additional costs for collecting the trace, and values of all variables that was read and/or modified during its execution will be collected and compared at the end of block.

The current implementation of the debugger already includes a special mode based on a similar principle: all calculations intended for execution on graphics accelerator can be also simultaneously performed on the central processor and then compared. In case of discrepancies, the needed information is reported to the user. After that, only results computed on central processor is used for further computations.

Enabling and using this comparative debugging mode does not require the user to make any changes to the program, or even re-compile it. All that is needed is to set the value of the environment variable *DVMH_COMPARE_DEBUG* to 1, or use the *./dvm cmph* command to start the execution of the program.

Also, to address the problem of discrepancy between the results of operations with real numbers, the new version of the debugger includes a mode that extends already described approach of correcting reduction variables to all results of real operations (configuration parameter *TraceOptions.SubstAllResults*). After successfully comparing a real variable with a reference value from trace, this reference will be substituted into the running program and used for further calculations.

This correction approach excludes the possibility of applying the trace size optimizations described above, since both the array checksums and the boundary iterations do not cover a significant part of the modifications of variables inside of parallel cycles. Therefore, in order to ensure the possibility of debugging real scientific and technical algorithms with this mode, it is recommended to use it in conjunction with the comparison of simultaneously running programs.

6 Conclusion

DVM-system was designed to automate the process of developing parallel programs.

Parallel programs with DVMH-directives can be efficiently (and without any additional changes) executed on clusters of various architectures including multi-core universal processors, graphics accelerators and Intel Xeon Phi coprocessors. This is achieved through various optimizations that are performed both statically (during compilation), and dynamically.

An important advantage of the DVM-system is the powerful tools for debugging developed DVMH-programs. These tools include the dynamic correctness control and the comparative debugging. Various instrumentation options were implemented for optimizing the process of debugging.

This article presented the problems that arose during the practical use of comparative debugging implemented in DVM-system, and suggested ways to overcome these problems.

Currently, a new version of the comparative debugging system is being developed, in which both the reference and the debugged program are executed simultaneously and the comparison of the calculation results is carried out directly in the process of execution. The new version of the debugging system will help to solve described problems with accuracy of computations, make the debugging process more flexible and less demanding to the memory of the computing system.

References

1. Helgrind: a thread error detector, http://www.valgrind.org/docs/manual/hg-manual.html, last accessed 2019/11/21.
2. DRD: a thread error detector, http://www.valgrind.org/docs/manual/drd-manual.html, last accessed 2019/11/21.
3. Intel Inspector. Memory and Thread Debugger, https://software.intel.com/en-us/intel-inspector, last accessed 2019/11/21.
4. Guard Parallel Relative Debugger, http://sourceforge.net/projects/guardsoft/, last accessed 2019/11/21.

5. Abramson, D.A., Sosic, R.: Relative Debugging using Multiple Program Versions. In: Intensional Programming I. Sydney: World Scientific (1995).
6. 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 2019/11/21.
7. Fortran DVMH langauge, Fortran DVMH compiler, compilation, execution and debugging of DVMH programs, http://dvm-system.org/static_data/docs/FDVMH-user-guide-en.pdf, last accessed 2019/11/21.
8. Lib-DVM library, http://www.keldysh.ru/dvm/dvmhtm1107/eng/sys/libdvm/rtsDDe0.html
9. OpenMP Application Programming Interface. Version 5.0. November, 2018, https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf, last accessed 2019/11/21.
10. The OpenACC Application Programming Interface. Version 2.6. November, 2017, https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf, last accessed 2019/11/21.
11. NAS Parallel Benchmarks, http://www.nas.nasa.gov/publications/npb.html
12. Krukov, V., Kudryavtsev, M.: Automated debugging of parallel programs. Vychisl. Metody Programm., 7 (4), 102–110 (2006).
13. Monniaux, D.: The pitfalls of verifying floating-point computations. ACM Transactions on Programming Languages and Systems (TOPLAS), ACM, 30 (3), pp. 12 (2008).
14. Ermichev, A., Krjukov, V.: Razvitie metoda sravnitel'noj otladki DVMH-programm. In: Nauchnyj servis v seti Internet: trudy XIX Vserossijskoj nauchnoj konferencii, pp. 150–156, Moscow, IPM im. M.V. Keldysha (2017), https://doi.org/10.20948/abrau-2017-15.