GEPARD – General Parallel Debugger for MVS-1000/M

V.E. Malyshkin and A.A. Romanenko

Novosibirsk State University
Chair of Parallel Computing, Russia
malysh@ssd.sscc.ru, arom@ccfit.nsu.ru

1 Introduction

Installed in Academgorok (Novosibirsk) multicomputer MVS-1000/M [1] is now in intensive use. MVS-1000/M is the multicomputer of cluster architecture. It consists of 11 nodes with coupled Alpha-21264 processors and used mostly for the development and debugging of application programs. Its architecture and software are fully identical to the MVS-1000/M installation in Moscow [2] (more than 350 nodes with coupled Alpha-21264 processors) where real large-scale numerical modeling is accomplished.

One of the problems we have is debugging of application parallel programs. Unfortunately only sequential debuggers and proofing tools are installed on master computer of MVS-1000/M. It's highly important to have specialized parallel MVS-1000/M debugger that should take into account the peculiarities as hardware and software of MVS-1000/M as application area.

2 Parallel Program Debuggers Overview

Unfortunately, considered parallel program debuggers (TotalView [3], RAMPA [4], AIMS [5], Vampir [6], JumpShot [7], Paradyn [8], etc.) have strict functionality, which makes it not suitable for usage on MVS-1000/M. Interactive tools, like TotalView, are not suitable because interactions bring distortion in program behavior negligible at non-parallel program debugging. Some tools, for example Jumpshot, accumulate only information on communication operations, though it is not enough sometimes. The other programs know nothing about MPI (e.g. CXpref) or don't support Alpha architecture, for example Paradyn.

The desirable debugger for MVS-1000/M should meet the following requirements:

- flexibility of the system for debugging interprocesses communications,
- specialization for debugging of numerical models.

The debugger should be designed for usage on multicomputer.

V. Malyshkin (Ed.): PaCT 2003, LNCS 2763, pp. 510–523, 2003.

3 Objectives of GEPARD Development

Basing on above mentioned requirements the project of GEPARD debugger development was initiated that provide:
- minimal influence on the behavior of an application program,
- flexible debugging data gathering and analysis,
- correspondence to the specification of MVS-1000/M.

4 Choice of Debugger Type

A parallel program for multicomputer is represented as a system of sequential asynchronous communicating processes [9,10]. Therefore, the development of a parallel program is carried out in two stages. From the beginning, sequential algorithms development, testing and debugging of implementing their sequential procedures are performed. Then the whole parallel program, assembled out of these procedures and communications, is debugged. In debugging the total correctness of all the interprocesses communications and the whole program should be checked. Not excluded that new errors in separately debugged components (sequential procedures) can be recognized.

Interactive and monitoring debuggers represent two different approaches to the implementation of the debugging tools. Interactive tools allow a user to stop the program, inspect values, and perform step-by-step execution. Monitoring systems are used if there is no possibility to use interactive tools or usage of them takes much time or heavily distorts time diagram of monitored system. Monitoring systems accumulate information to be analyzed on-line or after completion of the program execution. Since the influence of the debugger on real behavior of parallel program should be minimized the monitoring mode was chosen for GEPARD.

Two main ways to implement the collection of the debugging information are known. One of them is to execute a program under external trace tool. It might result that program runtime is substantially increased because program execution context is changed for every executable statement. Another one is to insert instructions into the source code in order to collect only the information required. In this case the debugger less influences on the program behavior, but the program's source code should be changed.

For GEPARD the second strategy was chosen. Gathered data are partially processed (collected, buffered, transferred to the trace file, etc.) by the external process. Data analysis is done after program completion that results in reduction of the debugger's influence on the program runtime.

5 GEPARD

GEPARD consists of the following components:
- debug language,
- data gathering system,
- visualization and analysis system.

An analysis of the program behavior is based on the information gathered at program runtime. The gathered information falls into three groups:

- state of the operations of communication (send/receive/synchronized),
- state of the program (e.g. value of some its variables),
- state of the program executing environment.

5.1 Debug Language

Two levels of information gathering are defined. By default only information on the state of the operations of communications is gathered (MPI function names, source/destination ranks, function call times, procedures run time and position of the called MPI function in source code). In order to get additional information a user should explicitly point out what info is required to be gathered. This is expressed in special debug language.

The debug language consists of instructions that are inserted into debugged program. Programmer performs insertion of the instructions into the source code. Each instruction is a comment of the C language. Its format is:

```
/*GPRD <instruction> */,
```

where debugger instruction defines description of the additional information to be gathered or function of the debugger.

This approach allows us not to rewrite program code. A user should only insert some comments that are processed by debugger's preprocessor. For instance, in order to count the number of loop iterations, user should place the following line as the first statement of a loop's body:

```
/*GPRD count */.
```

Similarly, user can count the number of calls of a certain function.

Expected interprocesses communications (IPC) also can be described in order the debugger could compare at runtime the described/expected and real program behavior. Communications define the relation on the set of all the processes. All the pairs of type (source_process_rank, destination_process_rank) are included into the relation. This relation is described by the instructions of the debug language. Here is a small example that shows the way to point out that the i-th process can sends messages to the next and previous processes in a line system of processes.

```
/*GPRD SCOMMSET $i SEND ($i-1, $i+1) */
```

The IPC system is not static and can be changed at the program runtime.

Debugger preprocessor recognizes these instructions and substitutes for them the proper statements of programming language. MPI function calls are replaced with wrapper-functions, which gather debug information. For example, MPI_Init function call is substituted for the following wrapper-function:

```
/*GPRD mod_MPI_Init(int *, char ***, int) */,
```

where the last parameter defines position of the function call in the source code. Preprocessor provides this information.

For now the only supported language is C with MPI-1.

5.2 Gathering of the Debug Information

Data gathering system consists of monitors (system processes, one for every virtual processor). Before execution of MPI_Init function each process of the parallel program creates monitor (MON) using fork (2) system call. MON is joined to their parent process with pipe. On completion of the monitor creation debugged process calls MPI_Init function. In accordance to the instructions, inserted into the source code, information is gathered and transferred to the MON. So, no activities for storing and processing information are performed by debugged program. It is done by MONs. MONs should put debug information into their internal buffers, gather information referred to program runtime environment, and perform initial data analysis. For example, if a user has described interprocesses communications, monitors compare this description with how the system of communications was really accomplished. The information on recognized mismatch is stored in the buffer too. When the buffer becomes full, debug info is put into the trace file.

Within a wrapper function debug information is sent to the MON twice – before and after replaced function call. It allows the debugger to keep track of information on functions lock. So, the debugger can recognize the situation of making receive call without proper send call.

5.3 Data Analysis and Visualization

In process of trace file creation, or after termination of the debugged program the gathered data can be visualized. The analysis system helps user to find the cause of an error. It is allowed to apply filters, to view different kinds of statistics. Analysis system points out different disparities; for example, mismatching quantities of send and receive calls, unbalanced load of processes, etc.

The visualization part of the analysis system has user-friendly interface in order a user doesn't spend time for study basic operations.

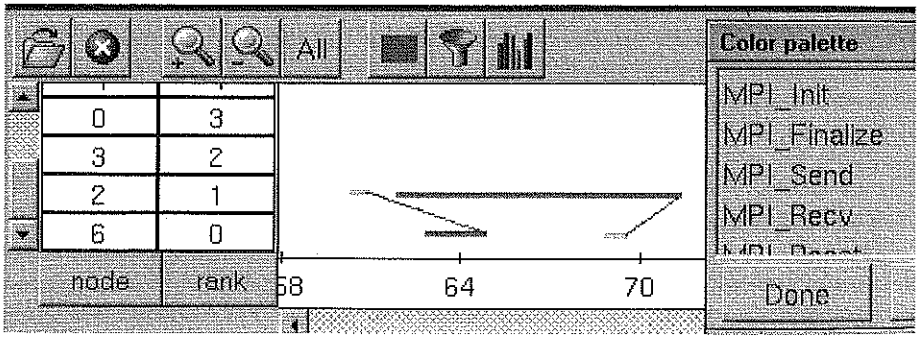

Fig. 1. The main window of visualization program

Event graph is displayed in the main window of visualization program. In the event graph each executing/running process is associated with a line, along which

the time is laid off. Events at process runtime are represented as the segments of the line. The segment length is equal to event duration. The fact of interaction between processes is represented as a line joining two segments. Gaps between segments of a process on the picture should be regarded as program execution.

Analysis of different program behavior with GEPARD enables us to recognize specific peculiarities of MPICH implementation. So, in Fig. 1 one can see that MPI_Send call (light segments) from thread 1 terminates before MPI_Recv function (dark segments) in thread 0 is called. It means, that data to be transferred with MPI_Send call is first stored into an internal buffer.

6 Conclusion

GEPARD demonstrates good ability for program debugging. It was in particular successfully applied for the analysis of the behavior of post-accidents states search parallel program developed for the Russian Energetic Company.

The debugger is now under permanent development. First, the analysis of the gathered information should be improved. Comparison of the different program executions is also planned. It is also planned to extend the debug language by the statements for description of mass operation like "to collect the information on all the function's calls located between statement_1 and statement_2". FORTRAN program debugging also should be supported.

References

1. Official site of the Novosibirsk Supercomputer Software Department. www.ssd2-new.sscc.ru
2. Official site of the Moscow Joint Supercomputer Center. www.jscc.ru
3. University of Karlsruhe: Parallel debugger TotalView. www.uni-karlsruhe.de/~SP
4. Krukov, V.A., Pozdnjakov, L.A., Zadykhailo I.B.: RAMPA – CASE for portable parallel programs development. Proceedings of the Third International Conference on Parallel Computing Technologies, Vol. 3, IPPE, Obninck, Russia (1993)
5. Yan, J.C., Sarukkai, S.R., Mehra, P.: Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit. In: Software Practice & Experience. Vol. 25, No. 4 (April 1995)
6. Pallas GmbH: Vampirtrace User's and Installation-Guide. www.pallas.com (1999)
7. Zaki, O., Lusk, E., Gropp, W., Swider, D.: Toward Scalable Performance Visualization with Jumpshot. www-unix.mcs.anl.gov/perfvis/software/viewers/jumpshot-2/paper.html
8. Miller, B.P., Callaghan, M.D., Cargille, J.M.: The Paradyn Parallel Performance Measurement Tools. In: Special issue on performance evaluation tools for parallel and distributed computer systems. IEEE Computer 28, No. 11 (November 1995)
9. Valkovskii, V.A., Malyshkin, V.E.: Parallel Program and System Synthesis on the basis of Computational models. Nauka, Novosibirsk (1988) [in Russian].
10. Hoare, C.: Communicating Sequential Processes. Prentice-Hall (1985)