Tool Integration Issues for Parallel and Distributed Debugging*

José C. Cunha João Lourenço Vitor Duarte
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Portugal

E-mail: {jcc, jml, vad}@di.fct.unl.pt

Abstract

This paper describes our experience with the design and implementation of a distributed debugger for C/PVM programs within the scope of the SEPP and HPCTI Copernicus projects. These projects aimed at the development of an integrated parallel software engineering environment based on a high-level graphical parallel programming model (GRAPNEL) and a set of associated tools supporting graphical edition, compilation, simulated and real parallel execution, testing, debugging, performance monitoring, mapping, and load balancing. We discuss how the development of the debugging tool was strongly influenced by the requirements posed by other tools in the environment, namely support for high-level graphical debugging of GRAPNEL programs, and support for the integration of static and dynamic analysis tools. We describe the functionalities of the DDBG debugger and its internal architecture, and discuss its integration with two separate tools in the SEPP/HPCTI environment: the GRED graphical editor for GRAPNEL programs, and the STEPS testing tool for C/PVM programs.

1. Introduction

Due to the increased interest in exploiting parallel and distributed applications, the development of adequate software engineering environments became a very important issue in recent years. This goal was the main motivation of the partners involved in the SEPP [1] and HPCTI [2] projects of the Copernicus Programme. Although each partner was responsible for the development of individual tools, a major concern aimed at their coherent integration into the GRADE environment. The whole development cycle is supported, including graphical editing, compilation, simulation and real parallel execution on top of PVM [3]. Associated tools for testing, debugging, performance monitoring, mapping, and load balancing are also supported.

In this paper we discuss the main issues involved in the design and implementation of the DDBG distributed debugger, and its integration into the GRADE environment.

The GRADE environment [4] consists of a set of development tools built around the GRAPNEL model for graphical parallel programming. GRAPNEL [5] is a graph-based visual programming model supporting the structured design of parallel applications. In order to provide an adequate view to the user, all development tools should refer to the abstractions provided by GRAPNEL. For example, as far as debugging is concerned, the inspection and control of the computation state should refer to the GRAPNEL program components and structures, and should be integrated with the graphical user interface supported by GRADE. However, at the same time, debugging at a lower level should also be supported, allowing the user to inspect and control the C/PVM-based components that are part of the GRAPNEL program. This requires that the debugging tool should provide an interface to the GRED graphical editor, while at the same time it should also allow direct access to the C/PVM debugging functionalities.

Another important aspect of a parallel software engineering environment is the possibility of a close integration between static analysis and testing tools, and the dynamic analysis and debugging functionalities. In fact, due to the great complexity of parallel computations, a tool is required to allow the user to generate adequate testing scenarios, depending on the parallel program structure and the dynamically established process interactions. One can obtain further information on program behavior and inspect specific computation paths in greater detail, if a debugging tool can be coupled to a testing tool.

The above interfacing requirements were satisfied by a

*Submited to "*The 3rd SEIHPC Workshop, Madrid, Spain*."

distributed process-level DDBG debugger. The prototype of the DDBG system allows the inspection and control of distributed C/PVM processes. The DDBG architecture can be extended to support further functionalities, such as thread-based models, and can be adapted to other intermediate-level platforms such as MPI[6].

In section 2 of this paper, we describe the DDBG debugger. In section 3 we discuss its integration into the GRADE environment, and with the STEPS testing tool. Finally we conclude by identifying ongoing research directions.

2. The DDBG debugger

2.1 Design Issues

The basic functionalities which are required by a debugging service concern state inspection and control of a computation. This includes abstractions related to individual processes or threads, and coordination-level abstractions such as deterministic re-execution, global distributed breakpoints, and evaluation of global predicates. Such functionalities strongly depend upon each programming and computational model, but it is possible to identify a set of basic debugging mechanisms (e.g. [7]), and use them in order to implement higher level functionalities.

Recently we have been working on the implementation of the DDBG distributed process-level debugger for C/PVM programs [8]. It allows an user or another tool to control and inspect multiple distributed processes. There are the following classes of debugging primitives:

- Control of the debugging session. This includes commands to start or finish a debugging session, to put a process under debugger control, and to remove a process from the debugging environment.
- Control of the process execution. This includes commands that directly control the execution path followed by a process, once it is under debugger control.
- Process state inspection and modification. This includes commands to inspect the state of a process in well-defined points which are reached due to the occurrence of breakpoints or other types of events (process stopped or terminated). The information that can be accessed includes process status, variable and stack frame records. and source code information.

This interface is used to implement the debugging services which are required by the integration with GRED and STEPS tools, as illustrated in section 3.

In order to support easy experimentation with debugging services for distinct computational models, a flexible software architecture is required. This architecture should be able to integrate and manage distinct types of process-level or thread-level debuggers, which depend on each hardware and operating system platform, and on each programming model. In the following, we describe the DDBG architecture, and discuss alternative designs in order to obtain increased flexibility.

2.2 The DDBG Architecture

The DDBG (**D**istributed **D**e**B**u**G**ger) [8] tool provides a set of debugging functionalities for distributed programs written in *C* and using the *PVM* system [3] to support concurrency and interprocess communications. The main features of DDBG are:

- *Simultaneous access from multiple (high-level) client tools*. Multiple tools can (independently) issue debugging commands over the same target application.
- *Dynamic attach and detachment of client tools to the debugging engine*. Client tools can "enter" and "leave" the debugging process dynamically, having their own life cycle independent of the DDBG debugger life cycle.
- *Global view of the system being debugged*. All the client applications share the same information concerning the program state and have the same abilities to issue inspection and control commands.
- *Support for heterogeneity*. Heterogeneity is supported at multiple levels: hardware, operating system, programming language and model, as a process-level debugger is used to access each individual target application process.
- *Easy integration with client tools*. Tool integration features and functionalities have been included in the debugger specification, from the architecture design until the effective implementation.

Three different types of processes can take part in a debugging session with DDBG (as presented in Figure 1):

1. *Client Processes (CP)*. These processes use a *Debugging Library (DL)* that provides access to all DDBG debugging functionalities.
2. *DDBG processes*. The DDBG debugger consists of the following components:
   - *Main Daemon (MD)*. The MD acts as a master or coordinator, and is responsible for receiving the CP requests, convert them into a set of commands and send them to the relevant *Process-level Debuggers (PLDs)* (see below). The MD is also responsible for receiving and processing the PLD

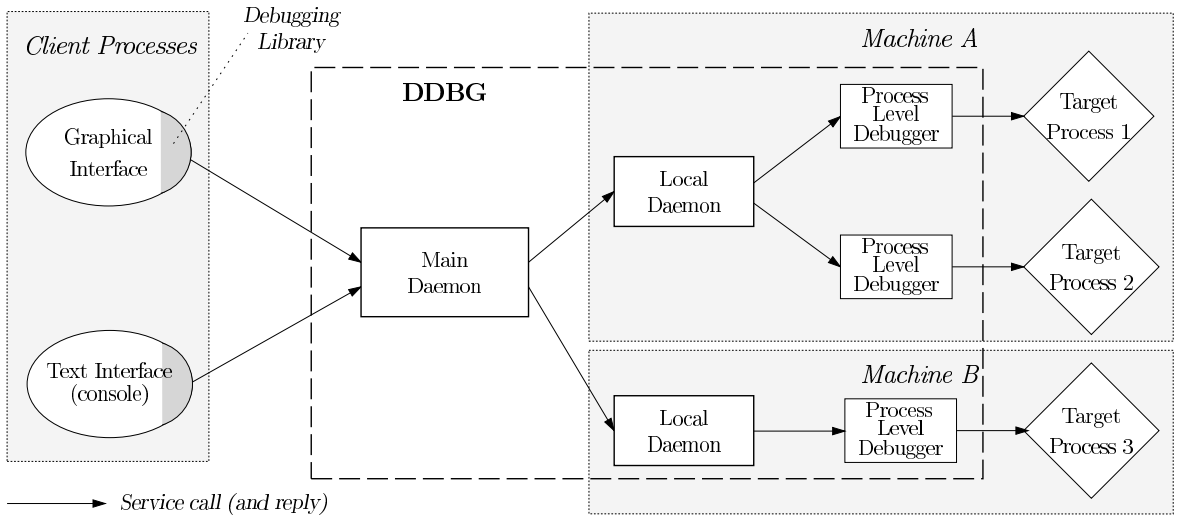

Figure 1. The DDBG distributed debugger

replies, sending them back to the CP through return parameters of the DL.

- *Local Daemons (LD)*. There is a LD in each machine, doing some local interpretation of the debugging commands and working as a multiplexer, forwarding these commands and controlling all the PLDs running on that machine.
- *Process-level Debugger (PLD)*. A system-dependent sequential debugger, that supports the programming language and the underlying hardware. There is a PLD attached to each process of the *Target Application*, issuing the inspection and control commands to that process.
- *Debugging Library (DL)*. This component is not a process by itself, but it's included into any client process in order to provide a set of functions that give access to the DDBG functionalities.
- *Graphical and Text-oriented debugging user interfaces (UI)*. These are two examples of CP (not being part of DDBG by themselves, but are included in the distribution), that were developed and integrated into DDBG, providing two different interfaces (one graphical and the other command-line oriented) to the DDBG debugger.

3. *Target Application Processes*. The application being debugged. This application can have multiple processes spread on multiple machines, with different hardware and operating systems.

There are alternative designs to DDBG, depending on how the responsibilities are distributed among its processes. In a pure hierarchical design the MD is responsible for the interpretation of debugging commands received from the CP, i.e. it performs all the necessary conversions, it forwards the actual PLD-level commands to the corresponding LD and sends the replies back to the CP. In this solution, the LD processes are just gateways with very limited responsibilities: contacting the right PLD and send its answer back to the MD. The MD does all the work. There are several disadvantages in such kind of design:

- *High MD complexity.* The MD process becomes very complex, as it must also support multiple concurrent client connections and so it needs to manage a lot of information concerning pending requests.
- *Hard to support heterogeneity.* It is more difficult to support heterogeneous systems, consisting of distinct types of PLD processes. In heterogeneous distributed computing one can have an application decomposed into multiple parts, each running on distinct sequential or parallel machines, with distinct PLD processes. This design requires the MD process to process all command and data interpretations.
- *Reduced flexibility.* As the MD program becomes very complex, it is more difficult to integrate new services into the architecture, such as performance monitoring and debugging.

A more flexible design would distribute the responsibility for actual command and data interpretation to each LD, and would let the MD do only the interfacing to the client tools. Each LD can then independently perform its tasks, depending on the specific characteristics of each local PD.

This is a better solution to support heterogeneous debugging, as well as to support extended services, because the required modifications are associated with specific LD processes. The functions left to the MD are the interfacing with client tools, the management of multiple connections to the debugging system, and the presentation of global views to the user concerning the global state of the distributed computation. As a result of our past experience in the SEPP and HPCTI projects, a new architecture that reflects the above design options is under development where the MD is a multi-threaded process with associated services.

3. Using DDBG in SEPP/HPCTI

3.1. Experiences With Tool Integration

It is very difficult to provide full integration among a large set of development tools such as the ones found in the SEPP/HPCTI projects. This is due to the need to offer consistent views at several levels: multiple user interfaces, tool behavior, tool interaction, and tool composition. Even in our project, where many of the tools were jointly developed from the beginning, a full integration was a difficult goal to achieve because it required a tight collaborative effort between the involved partners, concerning their design options, and the associated working environments (e.g. with distinct graphical user interfaces, and operating system platforms). However, we have obtained a reasonable degree of integration between several tools, and have opened the way to possible further integrations [2, 1, 9, 10, 11]. Concerning the debugging tool[12], one of the distinctive goals of our approach when designing and implementing the DDBG system was to provide a platform supporting easy experimentation with tool integration as far as debugging is concerned. Two main experiments were performed concerning the interfacing of DDBG with other parallel software development tools which exhibit very distinct functionalities. In the next two sections (3.1.1 and 3.1.2) two successful integrations of DDBG with other tools are presented.

3.1.1. Integrating DDBG into the GRADE Programming Environment

The GRADE (**GRA**pnel **D**evelopment **E**nvironment) is an integrated environment for the development of parallel programs in the GRAPNEL programming language. The GRAPNEL language is a graph-based visual parallel programming language, that supports a structured style for designing parallel applications, and is supported by the GRED graphical editor [5, 4]. in this section we will concentrate in the close integration of DDBG and GRED (**GR**apnel **ED**itor) in order to support debugging of GRAPNEL programs.

In such integrated environment the user involved in the debugging process should work at the same level and with the same abstractions that were used in program development[1] highlighting the entities in the graphical representation and their corresponding lines of source code in the textual program representation.

For such high-level debugging process for GRAPNEL programs, each debugging action at the GRED-level is mapped into a set of debugging actions at DDBG-level. Such commands are then sent and processed by DDBG, which in turn replies with DDBG-level answers that must be converted into the corresponding action in the GRED visual editor. In order to support potential long-execution commands, such as "*proceed until next breakpoint is reached*", an asynchronous (event) notification feature has been integrated into DDBG and used by GRED to detect the completion of such kind of commands.

The integration of DDBG into the GRADE programming environment is detailed in [13].

3.1.2. Integrating DDBG with STEPS

The STEPS testing tool [14, 15], developed by our partners at the Technical University of Gdansk, allows to identify potential critical paths and critical sections in a C/PVM program. The DDBG debugging tool can inspect and control the program behavior, helping in the localization of programs bugs and their causes.

When composing both tools, one must ensure that the program will run and behave as expected, and so the composition of the testing and the debugging tools starts by re-executing the target applications and forcing each process to follow some specific path and until a pre-determined point. It is necessary to ensure that the application will reach the critical points previously identified by the testing tool and will stop in a consistent state (also called a "*Global Breakpoint*"). At this potential critical point, the user can enter an interactive debugging session, using both the graphical and the command-line debugging environments, and issuing typical inspection and control commands directed at any of the target application processes.

The DEIPA (**D**eterministic (re-)**E**xecution and **I**nteractive **P**rogram **A**nalysis) tool was developed to support the integration of the STEPS testing tool and the DDBG debugging tool as presented above. DEIPA acts as an intermediary between those tools, recognizing and processing the output of the STEPS tool—the TeSS file, with a set of *global breakpoints*—and converting it into (a set of) commands for the DDBG tool. To support this functionality, the DDBG capability of having multiple

[1] This is a general concept, as it makes no sense to develop a program using the *C* programming language and then debug this same program at *assembler* level.

simultaneous client tools has been used, by having the DEIPA tool controlling the execution of all the processes of the target application and having the text user-interface (DDBG console) and/or the graphical user-interface to inspect and change each process state.

The DEIPA tool is mainly composed of 3 modules: the *Console*, the *Vid Database Manager*, and the *Replayer*. The architecture of the DEIPA tool and its relations with the STEPS and DDBG tools are presented in Figure 2, and explained below.

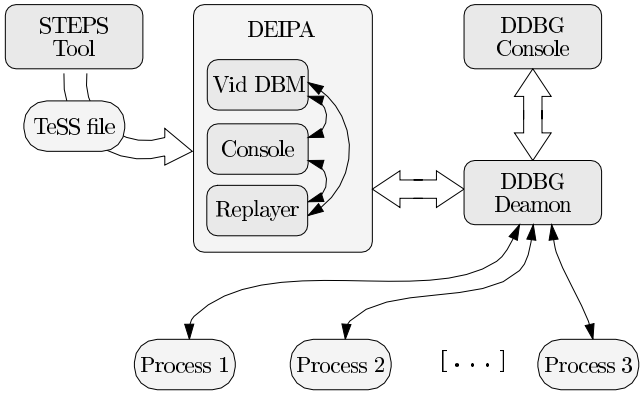

Figure 2. The integration of STEPS and DDBG

- *The Console module*. This module acts as the user interface to the DEIPA tool. Actually, this interface is based on a console (command-line oriented user interface), from which the user can load a TeSS file and control the (re-)execution of the target application, providing some basic commands to: control the DEIPA tool, e.g. load (to load a new TeSS file), or that are converted into (a set of) debugging commands and applied to the target application via the DDBG tool, e.g. step (to proceed into the next *Global Breakpoint*).

- *The Vid Database Management module* During static analysis, STEPS uses a symbolic naming schema (this is mandatory, as process execution isn't real but simulated) and DDBG uses real process identifiers (as the target application is actually running). This module implements the mapping from symbolic to actual process identifiers.

- *The Replayer module*. This is responsible for the mapping of the DEIPA console commands into DDBG commands, e.g. converting a DEIPA step command into a set of DDBG set_breakpoint and continue_execution commands. It's also responsible for the required process control, e.g. setting variables in an **if-then-else** statement, so that a process is forced to follow the specific path as specified in the TeSS file.

In [16] one can find a complete discussion of the DDBG and STEPS integration issues.

4. Conclusions and Future Work

In this paper we have discussed the DDBG debugger, and how it was used to offer debugging functionalities to other tools in a parallel software engineering environment. The DDBG is a distributed process-level debugger, i.e. it allows the control and inspection of distributed processes. Its functionalities were adequate to support the requirements posed by other tools in the GRADE environment.

This experience has allowed us to identify the following main directions to improve current debugging functionalities:

- Concerning computation state inspection and control. This includes the support of process-level and thread-level debugging, as well as the support of coordination-level services, such as distributed global breakpoints, and evaluation of global predicates.

- Concerning tool interaction and integration. This includes more flexible support for interfacing the debugger with distinct concurrent tools and user interfaces.

- Concerning heterogeneity. This includes the support of other parallel and distributed platforms besides PVM, such as MPI and WindowsNT systems.

With new kinds of requirements posed by highly interactive distributed programs, as for component integration in meta-computing problem-solving environments, new abstractions and interactions can be added to the traditional ones, and new tools can be expected to emerge, that share the same observation and control functionalities. A monitoring system for a distributed application can serve several purposes, from simple visualization of the interactions between the distributed components to exhaustive run-time information describing the program states during the execution, for performance evaluation (profiling) or debugging purposes. Typical uses of a monitoring system in parallel and distributed applications concern performance evaluation and visualization/debugging.

There is the need for systems that can work with several distribution support systems and be used by several tools, and can be extended and adapted to new environments and functionalities. Systems like these can be used for testing different support systems, new tools and to integrate them. This guides us to define hierarchically layered architectures that provide a well-defined interface between

levels, including tool interfacing. The ongoing work at the OMIS project[17] is an attempt to define a standard that includes inspection and control functions, and allows a clear separation between user level services and basic monitoring functionalities.

The above aspects are being taking into account within the scope of an ongoing project.

Acknowledgments

This work was partially supported by the EC within COPERNICUS Programme, Research Projects SEPP (Contract CIPA-C193-0251) and HPCTI (Contract CP-93-5383).

References

[1] S.Winter et al. Software Engineering for Parallel Processing, copernicus programme. Final report, University of Westminster, March 1997.

[2] S.Winter et al. High Performance Computing Tools for Industry, copernicus programme. Final report, University of Westminster, September 1996.

[3] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. A User's Guide to PVM Parallel Virtual Machine. Technical Report ORNL/TM-118266, Oak Ridge National Laboratory, USA, 1991.

[4] G. Dózsa, P. Kacsuk, and T. Fadgyas. Development of graphical parallel programs in PVM environments. In *Proceedings of DAPSYS'96*, pages 33–40, 1996.

[5] P. Kacsuk, G. Dózsa, and T. Fadgyas. Designing parallel programs by the graphical language GRAPNEL. *Microprocessing and Microprogramming*, 41:625–643, 1996.

[6] Message Passing Interface Forum. Document for a standard message-passing interface. Technical Report Technical Report No. CS-93-214 (revised), University of Tennessee, April 1994. Available on **netlib**.

[7] J. Brown, J. Francioni, and C. Pancake. White paper on formation of the high performance debugging forum. Available in "http://www.ptools.org/hpdf/meetings/mar97/whitepaper.html", February 1997.

[8] J. C. Cunha, J. Lourenço, and T. Antão. A debugging engine for a parallel and distributed environment. In Hungarian Academy of Sciences-KFKI, editor, *Proceedings of DAPSYS'96, 1st Austrian-Hungarian Workshop on Distributed and Parallel Systems*, Misckolc, Hungary, October 1996.

[9] J. Astaloˇ s L. Hluchý, M. Dobrucký. Hybrid approach to task allocation in distributed systems. In *Lecture Notes in Computer Science 1277*, pages 210–216. Springer, 1997.

[10] T. Delaitre, G. Justo, F. Spies, and S Winter. Simulation modeling of parallel systems. In Hungarian Academy of Sciences-KFKI, editor, *Proceedings of DAPSYS'96, 1st Austrian-Hungarian Workshop on Distributed and Parallel Systems*, Misckolc, Hungary, October 1996.

[11] E. Luque, A. Ripoll, A. Cortés, and T. Margalef. A distributed diffusion method for dynamic load balancing on parallel computers. In IEEE CS Press, editor, *Proceedings of EUROMICRO Workshop on Parallel and Distributed Processing*, San Remo, Italy, January 1995.

[12] J. Cunha and J. Lourenço. An experiment in tool integration: the DDBG parallel and distributed debugger. *EUROMICRO Journal of Systems Architecture, 2^{nd} Special Issue on Tools and Environments for Parallel Processing*, 1997.

[13] P. Kacsuk, J. Cunha, G. Dózsa, J. Lourenço, T. Fadgyas, and T. Antão. A graphical development and debugging environment for parallel programs. *Parallel Computing*, 1997(22):1747–1770, February 1998.

[14] H. Krawczyk and B. Wiszniewski. Interactive Testing Tool for Parallel Programs. In P. Crolll Chapman & Hal: I. Jelly, I. Gorton, editor, *Software Engineer for Parallel and Distributed Systems*, pages 98–109, London, UK, 1996.

[15] H. Krwaczyk and B. Wiszniewski. Structural Testing of Parallel Software in STEPS. In COPERNICUS Programme, editor, *Proceedings of the 1st SEIHPC Workshop*, Braga, Portugal, 1996.

[16] J. C. Cunha, J. Lourenço, H. Krawczyk, P. Kuzora, M. Neyman, and B. Wiszniewski. An integrated testing and debugging environment for parallel and distributed programs. In *Proceedings of the 23^{rd} EUROMICRO 97 Conference*, pages 291–298, Budapest, Hungary, September 1997. IEEE CS.

[17] T. Ludwig, R. Wismuller, V. Sunderam, and A. Bode. OMIS — On-Line Monitoring Interface Specification (Version 2.0). Technical report, LRR-TUM, Munich, Germany, July 1997.