MSDB – A Debugger, Profiler and Visualizer for Modula-2*

Stefan U. Hänßgen*

haensgen@ira.uka.de

Universität Karlsruhe, Department of Computer Science, Germany

Abstract: Problem-oriented explicitly parallel programming languages like Modula-2* move the focus of debugging from machine-level problems to higher levels of abstraction. We present the MSDB semantics of distributed breakpoints and the visualization of distributed data and parallel execution.

1 Introduction

Today, parallel languages, and therefore their debuggers, largely suffer from machine dependence. The user spends too much time debugging machine-level problems, instead of the truly algorithmic ones.

This paper presents concepts for a high-level parallel language debugger for Modula-2* [6]. This language is a parallel extension of Modula-2 featuring an arbitrary number of processes operating in a single address space. Arbitrarily nested synchronous and asynchronous parallelism introduced by the `FORALL` construct.

Due to its high level of abstraction, many of the usual problems of parallel programming are of no concern to the Modula-2*user, e.g. communication operations, virtualization, and communication deadlocks. These are all taken care of by the compiler.

This property shifts the focus of debugging from machine dependent issues to more abstract problems such as visualization of parallel activities, distributed data, dynamic activation trees, and profiling at the language construct level.

2 Concepts

The debugger provides the usual features that are also found in sequential debuggers, such as setting and examining variables, setting breakpoints and stepping through the program. Additionally, MSDB supports different views of distributed data and parallel execution.

In this section, we focus on the concepts that use high-level abstractions in terms of the language and help to understand the parallel execution.

2.1 Activation Trees and Grouping

When many different parallel activities are running in parallel, it is difficult to keep an overview of the program's execution .

The *activation tree* shows, in terms of programming language constructs, where the activities inside the program are located at the moment. It is updated internally each time an activity enters a new statement that affects the control flow, e.g. a loop, an `IF`-statement or a `FORALL`.

The leafs of the tree are the locations of the activities while their predecessors represent their dynamic history. This resembles a stack in a conventional program, only that MSDB shows all constructs, not just function calls, and displays parallel activities in form of a tree.

To make this approach feasible even with a large number of concurrent activities, equivalent leafs are *grouped* together, e.g. all activities which are inside the same round of a loop are shown as one node. A simple example of grouping several nodes in an `IF` and a `WHILE` statement executed in parallel is presented in Fig. 1.

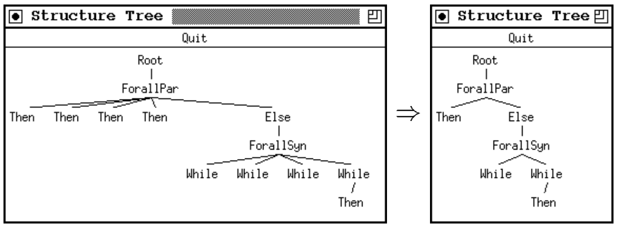

Figure 1: Dynamic activation tree before and after grouping of equivalent nodes

2.2 Multiple program counters

We need to keep track of the program counter positions in the code. Since maintaining one source code window for each virtual PE quickly gets confusing with a large number of PEs, we use multiple program counter arrows in the source code window to represent the *grouped activities.* The size of the arrow indicates the number of processes that are at this position. More detailed information about the original processes can be obtained by clicking on the arrow.

*This work was supported in part by the Graduiertenkolleg "Controllability of Complex Systems" (DFG Vo 287/5-2)

2.3 Visualization of data

In the debugger multidimensional arrays are visualized by 'cutting' through the data and viewing 2-dimensional slices thereof.

Different kinds of visualizers are available: *Comparison* visualizers highlight the array elements which are equal to, greater or less a user defined value. *Range* visualizers represent a generalization of comparison which highlights all values inside a user-defined range. Finally, *Value* visualizers map array elements to a grayscale representation of the data, compensating for statistical peaks.

2.4 Profiling and activities

For performance tuning the user needs to know which parts of the program are executed how often.

MSDB keeps track of the processes entering, repeating and leaving constructs. Thus, it can provide the user with a display of the most-often used *structures*. The number of runs through a statement corresponds roughly to the time spent there.

Abstracting from machine dependent displays of processor activities, such as featured in [4], the debugger has a specialized visualizer for the current *activities* of the virtual PEs.

Although the real PEs are not visible to the Modula-2* programmer, combining this view with the profiling view gives more information about the load distribution: one can easily distinguish one process running a loop 1000 times from 1000 processes running it one time.

2.5 Breakpoints

In MSDB breakpoints are global and asynchronous. A stop command is broadcast whenever one process encounters a breakpoint. The stop is *global* to provide consistent handling among MIMD, SIMD, and SISD machines – on the latter two all stops are necessarily global. In addition, virtualization loops make it difficult to stop just one single process.

Asynchronous breakpoints are consistent with Modula-2*'s semantics: the end of the `FORALL` statement is a synchronization barrier. Thus, the stop command reaches every process eventually. In a synchronous `FORALL`, all processes operate in lockstep, so they are stopped in the same state. The semantics of the asynchronous `FORALL` permit an indeterministic order of execution among the processes, so it is of no concern where a process exactly is stopped.

The semantics of stepping through the program are similar. All processes execute the next statement unless they are at the end of a `FORALL`. There they wait for the remaining processes to synchronize with them.

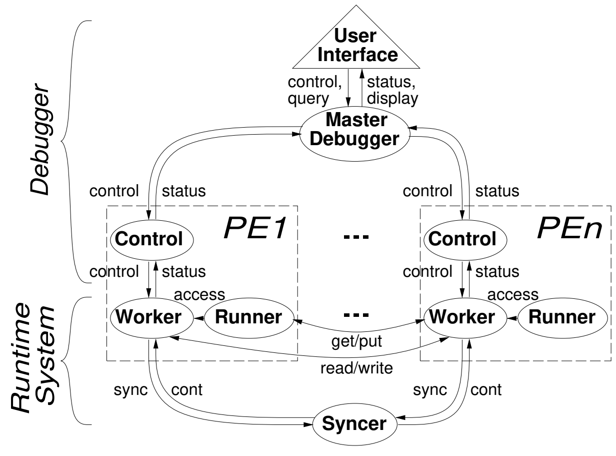

Figure 3: Process interaction in the LAN-based MSDB

3 System Description

Currently, prototype implementations of MSDB exist for a *network of workstations* and a *sequential program* running on a single workstation. The LAN implementation is based on the p4 system [1] for communication and process creation. To inspect and change both data and program state, the UNIX debuggers `dbx` and `gdb` are used. The X Windows based user interface is derived from `xdbx` [2]. Fig. 2 shows a representative screen dump of the whole debugger in action.

Figure 3 illustrates the setup of the processes and their interaction. The runtime system consists of three components per PE which execute all virtual PEs' instructions in virtualization loops. The *Worker* performs the distributed computations. It does synchronized read/write accesses itself, like the control network of a parallel machine. The *Runner* deals with unpredictable get/put memory accesses on remote PEs, like the data network of a parallel machine. It uses the `ptrace` system call for directly accessing data of the Worker. The *Control* process stops the Worker, modifies it, and retrieves information for visualization etc. It also monitors the Worker's state and reports changes.

A central *Syncer* provides global synchronization barriers and I/O. All processes are coordinated in a *Master Debugger* which also handles the user interface and merges information.

To support the debugger, the compiler places additional information in the code. It generates hints for the line and variable mapping as well as debug functions which record the dynamic activation tree and also generate the profiling information.

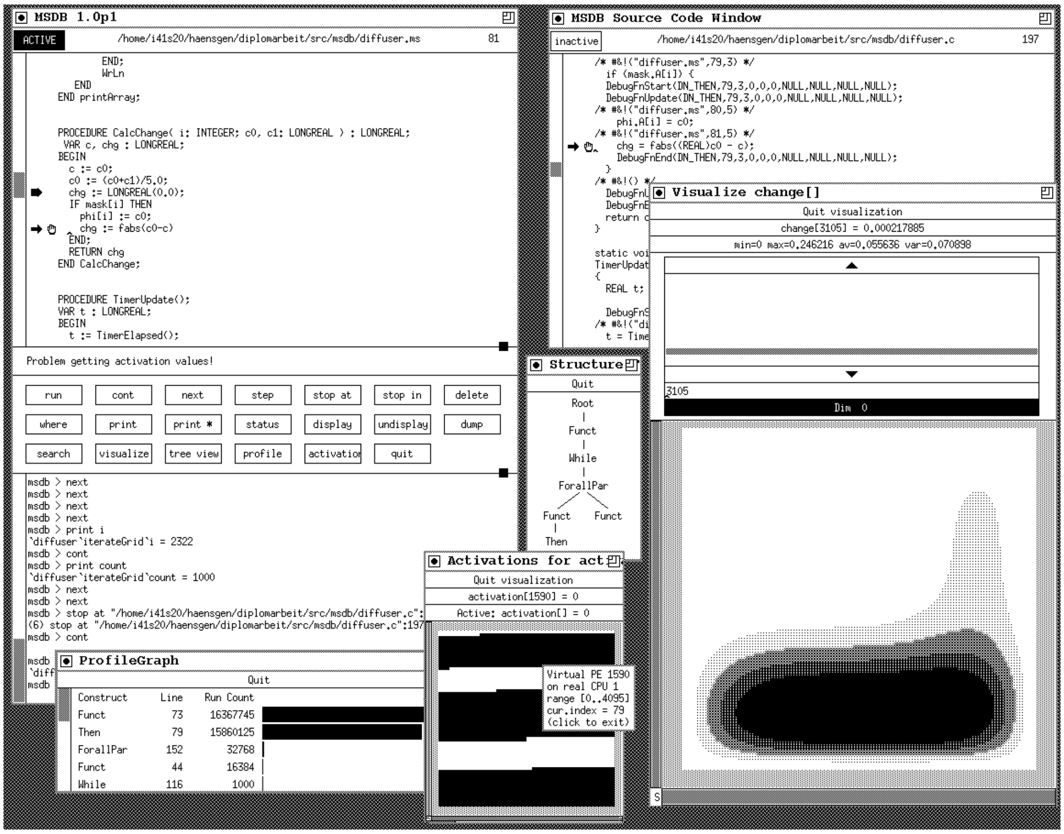

Figure 2: MSDB in action. The program being debugged is a heat diffusion simulation, the results of which are shown in a value visualizer. A visualization of the virtual PEs' activities as well as the activation tree and a profile graph can also be seen. The left window shows the Modula-2* source with two differently sized program counters and a breakpoint, the right window displays the corresponding C source.

4 Status and Future Work

The *distributed runtime system* and the *sequential version* are operational. Work on other platforms such as MasPar and KSR-1 is in progress.

It is planned to design and implement a general debugging kernel to meet the specific needs of Modula-2*. The current solution with separate `dbx` and `gdb` processes is slightly inefficient and requires a special interface to the debugger using pseudo terminals.

5 Conclusion

We presented a debugger for the problem-oriented explicitly parallel language Modula-2*. We described appropriate abstractions and visualizations of activation trees, arrays and activities. The emphasis was put on more user-oriented issues while most machine-level problems could be ignored.

References

[1] R. Butler and E. Lusk: User's Guide to the p4 Programming System. Argonne National Laboratory ANL-92/17 (1992)

[2] P. Cheung: xdbx source code. University of Texas at Austin (1989)

[3] S.U. Hänßgen: Ein symbolischer X Windows Debugger für Modula-2*. Master's thesis, Computer Science Dept., University of Karlsruhe (1992)

[4] MasPar Computer Corporation: System Overview and MPPE Manuals. Sunnyvale, California (1991)

[5] M. Philippsen and W.F. Tichy: Modula-2* and its Compilation. Proceedings of the First International Conference of the Austrian Center for Parallel Computation, Salzburg, Austria (1991) 169–183, Springer Verlag, Lecture Notes in Computer Science 591 (1992)

[6] W.F. Tichy and C.G. Herter: Modula-2* – An Extension of Modula-2 for Highly Parallel, Portable Programs. Internal report 4/90, Computer Science Dept., University of Karlsruhe (1990)

[7] D. Zernik and L. Rudolph: Animating Work and Time for Debugging Parallel Programs, Foundation and Experience. Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices 12/91 (1991), 46–56