Using Control and Data Flow Analysis for Race Evaluation

Dieter Kranzlmüller, Siegfried Grabner, Jens Volkert
GUP Linz, Johannes Kepler University Linz
Altenbergerstraße 69, A-4040 Linz, Austria/Europe
phone: +43 732 2468 / 9499 fax: +43 732 2468 / 9496
[kranzlmueller | grabner]@gup.uni-linz.ac.at

Abstract

The needs for larger problem sizes and for more accurate results force the users in the field of scientific computing towards applying parallel machines. Besides problems with initial program development another hard task arises with parallel program debugging, where severe difficulties appear with nondeterminism and race conditions.

This paper describes the tools ATEMPT and CDFA, two modules of the MAD environment which support the detection of simple errors in the communication structure and race conditions in parallel programs. While ATEMPT generates an event graph and visualizes race condition candidates of an actual execution, CDFA analyzes the source code and produces data structures for investigation of control and data flow graphs. The combination of both tools gives further insight into a program and makes the evaluation of race evaluation more efficient.

Keywords

Debugging, program analysis, race conditions, event manipulation.

1. Introduction

Debugging parallel programs needs more attention than debugging sequential ones. The reasons are errors unknown in the sequential domain and nondeterminism of parallel programs. Although there are promising approaches to detect spurious errors, no tools are available for analysis of program behavior affected by race conditions.

The *M*onitoring *A*nd *D*ebugging environment MAD tries to provide a solution. It is an integrated toolset for debugging of parallel programs based on the message passing paradigm. Several specialized modules for various activities of error detection are provided [Kran 96a].

A main role in the environment plays ATEMPT (*A T*ool for *E*vent *M*ani*P*ula*T*ion). It is used for the visualization of monitored data, which are stored in tracefiles during observed program runs. They are displayed as process time diagrams and can be inspected easily. With the help of ATEMPT the user is able to [GrVo 96]

- inspect communication events,
- automatically detect and visualize errors in the communication structure,
- inspect latencies in communication due to network contention,
- find candidates for race conditions,
- graphically manipulate events that are candidates for race conditions.

The anomalous effects are highlighted with different colors so that the user can easily catch them at a glimpse. Having detected an error at a certain position in the trace the user can inspect the event attributes and the corresponding source code. This activity is assisted by a source code reference between ATEMPT and an integrated filebrowser.

The most important feature that separates ATEMPT from other graphical debugging tools is the event manipulation of race condition candidates. Firstly all *racing receives* [NeDa 94] are evaluated and highlighted together with the messages that race towards these receives. For further analysis the corresponding events of the racing messages can be exchanged graphically with one of the other racing receives. This leads to a different ordering of message arrival in the event graph. (Of course the receive statement itself will not be moved because this would be a change in the program code!)

Afterwards the manipulated part of the communication graph will be stored in a new trace file. With trace driven replay that uses these changes in addition to the original communication graph, the effects of the reordering will be shown. If a different ordering of these events causes different results, a *crucial race* has been detected. Then the code can be changed in order to guarantee correct results.

An improvement to the race detection mechanism comes from CDFA, the *C*ontrol and *D*ata *F*low *A*nalyzer. With static information about the source code and a connection to the dynamic data of ATEMPT, CDFA allows additional investigations of the program flow as well as extended evaluation of race condition candidates.

The intention of this paper is to give an overview of the race condition detection approach. In the next section we will describe types of errors in message passing programs. The debugging approach is summarized in section 3, discussing how ATEMPT detects possible races. Section 4 on CDFA and race manipulation gives an example how CDFA can eliminate races for the replay that are not crucial. This shows some of the tool's capabilities in practice and leads to goals for future improvements.

2. Typical Errors in Message Passing Programs

When searching for errors in parallel programs the following two groups of program faults can be distinguished [Grab 97]:

- *errors of origin* and
- *subsequent errors*.

Errors of origin are those errors, where the location of faulty behavior and original reason is the same. On the contrary the manifestation of *subsequent errors* occurs later in time than the reason for the misbehavior. In this case the control flow has to be followed back in time in order to find the error of origin that was responsible for the erroneous behavior of the subsequent errors.

In sequential programs this backtracking is only carried out on a single existing task. However, in parallel programs the trace of subsequent errors to the original errors can be distributed about more than one process and is therefore more difficult to follow. Some errors in this case are wrong communication structures and wrong data that

are sent to other processes. They can be automatically detected and visualized with ATEMPT.

The worst errors that may occur are known as *race conditions* in literature [HeMc 96]. A race is a typical case of unintended nondeterminism [NeMi 92, ChMi 91]. On shared memory architectures a race occurs when two or more parallel tasks access the same shared variable in an unspecified order and at least one of the accesses is a write access (*data race*) [Helm 91]. With the message passing paradigm on distributed memory machines a race (*message race*) can occur if there exists one or more receive events and the order in which incoming messages are read from the communication buffer is unspecified [Netz 96].

The problem with this kind of bugs is that they appear sporadically and cannot be localized and corrected easily. A technique that is widely known for the investigation of these kind of errors is trace driven simulation [LeMe 87, ChSt 91]. ATEMPT was developed as an extension for such a race investigation mechanism in order to find possible races and to evaluate their influence on a program's results.

3. Overview of the Debugging Approach

3.1 Monitoring of Program Runs

The basis for error recognition and detection of race conditions is a trace of events recorded during an initial program run. In our terminology the most important events occur as result of one of the following programming statements:

- *send*: sending a message from a process to one or more receivers.
- *receive*: receiving a message from a process.
- *test*: probing the arrival of a certain message.

The *E*vent *M*onitoring *U*tility EMU [Kran 96b] records these events in tracefiles during the program execution. The source code is implemented either for the nCUBE 2 multiprocessor with its native communication library or for MPI [MPI 94].

3.2 The Event Graph Display

The tracefiles are the constraint for the generation of the global event graph, which is a partially ordered graph in time that represents the interprocess dependencies between processes. The vertices of the event graph are communication events that are stored in the tracefiles. The arcs are either computation or communication between corresponding send and receive events.

After the connections have been established errors in the communication structure are colored automatically. These bugs are isolated events and corresponding send and receive events using different message lengths as parameters [GrVo 96].

Another useful feature in this tool is the inspection of single events in detail which shows all event attributes. A connection back to the source code allows to easily eliminate located bugs.

3.3 Race Condition Candidates

When debugging a program for race conditions, the following question has to be answered: "*What would have happened if a message had arrived before another one?*" To help the user with this question event manipulation facilities are integrated in ATEMPT.

The starting point is a set of events that are possible candidates for an exchange. These are receive events, where the parameters are wildcards, and therefore allow any message to be received. With such a set of possible races available, the user can perform the following two activities:

a) Correct the program in order to remove the race immediately,
b) Investigate the results of different orderings at race conditions.

Activity a) can be applied, if the resulting behavior is obvious for the user. For example, some algorithms intentionally use nondeterminism, therefore a race condition candidate would show correct behavior. On the other hand, the user might know that nondeterminism is absolutely not intended in a certain part of code and thus, can immediately change the program. This can be done by distinguishing different senders with different message types and/or using the number of the sending process in the receive statement explicitly instead of wildcards.

In case b) the user cannot decide how a race might influence the program's result. Actually, the only thing one knows is the behavior that was observed in an initially monitored program run. To investigate other event orderings than the observed one, the user can graphically exchange the events. Afterwards an execution with the modified event graph can be initiated with PARASIT (*PARA*llel *SI*mulation *T*ool) [Kran 94] in order to test if another program flow with other results might be generated.

3.4 Simulation with PARASIT

With an event graph available, either original from a monitored program run or manipulated with ATEMPT, a technique called trace driven simulation [MiCh 88, ChSt 91] can be used to reproduce a certain execution of a nondeterministic program. This can be done with PARASIT.

The program that has to be simulated will be linked with an extended version of the underlying communication library. The simulation statements perform the normal communication and the simulator's activities. If the program reaches a *receive* statement, the simulator is involved in order to control the communication as defined by the communication graph. It chooses the next message that has to be received. If such a message is not available, the simulator delays the program's execution until the correct message arrives.

Up to the part of manipulation the simulated program run equals the same behavior that has been traced by the monitor. From this point onwards the program flow may deviate from the original trace. Of course, the program cannot be controlled by the simulator any longer. Instead, the monitor observes the execution again. Afterwards the results of this new program run are validated again and, if changes are observed the detected race is crucial and has to be eliminated.

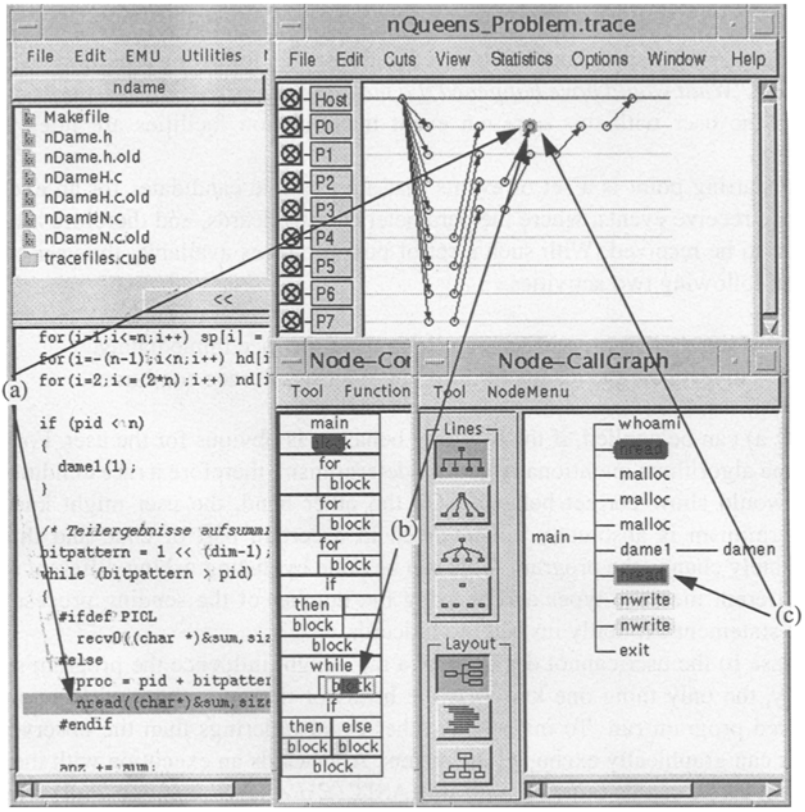

Fig. 1. Connections between (a) event graph in ATEMPT and Filebrowser, and (b) control flow graph and (c) function call graph in CDFA

4. Control and Data Flow Analysis

The solution described so far is a good starting point for error detection and race condition evaluation. However, there are some drawbacks with this approach:

a) Understanding the program from it's abstract representation as an event graph can be difficult.
b) The set of possible races can be rather high [NeDa 94]. One reason is that many intended places of nondeterminism may exist in a program.

4.1 Improved Insight into the Program

Program understanding can be improved, if a connection from the abstract visualization to the original source is provided. The representation of the source can be established as lines of code or on some kind of program flow graphics. The connection to the lines of code has been implemented with a filebrowser, that highlights statements corresponding to events selected in the event graph display of ATEMPT.

Code A on processor P0:

```
...
for (i=0;i<N;i++) {
   receive(data, ANY_PROCESS);
}
```

Code B on processor P0:

```
...
for (i=0;i<N;i++) {
  receive(data, ANY_PROCESS);
  DoSomething(data);
    ...
}
```

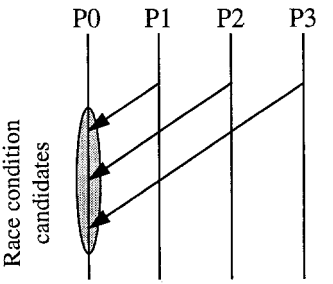

Fig. 2. Code fragements A and B and possible event graph.

A similar connection is provided with the *C*ontrol and *D*ata *F*low *A*nalyzer *CDFA*. Upon start it performs static analysis of the source code and creates data structures for further investigations. With an interface similar to the one of the filebrowser, commands can be send to CDFA, and CDFA returns the results.

In CDFA program understanding is supported by two graphical representations, a function call graph and a program flow graph. The function call graph shows a tree of functions in a program, whereas the control flow graph represents control flow within a specified function. CDFA receives the source code information, displays the function call graph and the corresponding control flow graph and highlights the program blocks containing the selected event. An example for such a connection is visible in figure 1.

A special role is dedicated to the communication functions. As they are most important for parallel debugging, they are special building blocks of the graph. Any block containing one or more communication functions is marked with different colors, indicating different message passing functions.

This functionality is useful mainly for the understanding of the static structure of the program. If the event graph is too abstract and the source code is long and complex, the function call graph together with the program flow graph might help to improve the user's comprehension. With increased understanding of the program, some races can be eliminated before any exchange of events and — even more important — before a tedious simulation of the program. On the other hand race candidates can be sorted out because they represent intended nondeterminism.

4.2 Race Condition Validation with CDFA

The second part of CDFA is more directly connected to the race condition candidates. While ATEMPT only knows about the communication of exactly one program run, CDFA knows about the program flow in general. The solution is based on the combination of the dynamic trace data and the static source code. This is done as follows: ATEMPT generates the list of all possible races and sends them in pairs to CDFA. Then CDFA checks, if one or more of theses races result in a different control flow. The result is returned to ATEMPT, which applies the changes to the set of candidates.

An simplified example for this validation is explained with two code fragments in figure 2. The event graph for both codes is the same. If ATEMPT only uses it's own

race evaluation codes, even the race condition candidates are the same. However, if CDFA is used to check the races, code A does not contain any races at all, while all races are confirmed for code fragment B.

The reasons are simple. In code A the receives get a message, but the data of the message are not processed any further. This can be the case, if A represents a master process that just wants to know if all other processes have reached a certain point (a typical synchronization situation in numerical algorithms). In code B the receive statements also get some data, but the data are used in the following lines of code. Thus, a dependency between message arrival and consecutive processing may exist and can only be checked during replay.

This small example is just used to show the strategy's principle which is more valuable in complex codes. Nevertheless, the methods are similar. Check the code, if the ordering of the data influences the results of the program. If not, remove the candidates from the set. If the set of possible races is decreased, the number of simulation runs and therefore the time for race condition validation is smaller. However, such investigations are only possible if the dynamic representation of a program run — the event graph — is combined with static control and data flow information.

5. Conclusion

The existing tools of the MAD environment, especially ATEMPT, have been very useful in debugging parallel programs. Many errors can be detected and thus corrected. The event manipulation functionality is unique and seems a promising approach to the detection of errors relating to race conditions.

The recent extension of MAD with CDFA tries to further improve the debugging task. Firstly, graphical representations of program flow help to improve program understanding and are therefore useful for any program analysis activity. Secondly, the dedicated race validation functionality helps in reducing the set of race candidates, by checking if a race is actually possible in terms of the program flow.

A new direction which we will follow with our debugging strategy as well as with our environment is the debugging of code written for virtual shared memory machines. Then the problem of races will be much more complicated, since data races and message races might appear concurrently.

Acknowledgement

Many students participated in this project. We would like to thank especially Richard Schall and Andre Christanell.

References

[ChMi 91] J.-D. Choi, S.L. Min, "RACE FRONTIER: Reproducing Data Races in Parallel Program Debugging", Proc. 3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP, Williamsburg, Virginia, pp. 145 - 154 (April 1991).

[ChSt 91] J.-D. Choi, J. Stone, "Balancing Runtime and Replay Costs in a Trace-and-Replay System", Proc. ACM/ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, CA, pp. 26 - 35 (May 1991).

[Grab 97] S. Grabner, *Search strategies for errors in parallel programs*, Ph.D. thesis, Systems Programming Institute, Johannes Kepler University Linz (1997). [in German]

[GrVo 96] S. Grabner, J. Volkert, "Debugging Distributed Memory Programs Using Communication Graph Manipulation", Proc. HPCS 96 Symposium, Montreal, Canada (June 1996).

[Helm 91] D.P. Helmbold, C.E. McDowell, J.-Z. Wang, "Detecting Data Races from Sequential Traces", Proceedings HICSS 24, Hawaii, Vol. 2, pp. 408 - 417 (Jan. 1991).

[HeMc 96] D.P. Helmbold, C.E. McDowell, "Race Detection - Ten Years Later", in: M.L. Simmons, A.H. Hayes, J.S. Brown, D.A. Reed (Eds.), "Debugging and Performance Tuning", IEEE Computer Society, pp. 101-126 (1996)

[Kran 94] D. Kranzlmüller, S. Grabner, J. Volkert, "PARASIT — Parallel Simulation Tool", CEI project PACT Technical Report D7V-1, GUP Linz, University Linz, Austria (Dec. 1994).

[Kran 96a] D. Kranzlmüller, S. Grabner, J. Volkert, "Debugging with the MAD Environment", Proc. of Workshop on Environments and Tools for Parallel Scientific Computing III, Faverges de la Tour, France (Aug. 1996).

[Kran 96b] D. Kranzlmüller, S. Grabner, J. Volkert, "Monitoring Strategies for Hypercube Systems", Proc. of 4th EUROMICRO Workshop on PDP, Braga, Portugal, pp. 486-492 (Jan. 1996).

[LeMe 87] T.J. LeBlanc, J.M. Mellor-Crummey, "Debugging parallel programs with instant replay", IEEE Trans. on Comp., pp. 471 - 482 (April 1987).

[MiCh 88] B.P. Miller, J.-D. Choi, "A Mechanism for Efficient Debugging of Parallel Programs", Proc. SIGPLAN/SIGOPS Workshop on Parallel & Distributed Debugging, Madison, Wisconsin, pp. 141 - 150 (May 1988).

[MPI 94] *MPI: A Message-Passing Interface Standard*, special issue of The Intl. Journal of Supercomputer Applications and High Performance Computing, Vol. 8 (3/4) (Fall/Winter 1994).

[Netz 96] R.H.B. Netzer, T.W. Brennan, S.K. Damodaran-Kamal, "Debugging Race Conditions in Message-Passing Programs", Proc. SPDT'96, SIGMETRICS Symposium on Parallel and Distributed Tools, Philadelphia, PA, pp. 31-40 (May 1996).

[NeMi 92] R.H.B. Netzer, B.P. Miller, "What are Race Conditions? — Some Issues and Formalizations", ACM Letters on Programming Languages and Systems, Vol. 1, No. 1 (March 1992).

[NeDa 94] R.H.B. Netzer, S.K. Damodaran-Kamal, "Accurate Race Condition Detection for Message-Passing Programs", Brown Univ. Dept. of Computer Science Technical Report (April 1994).