PARASIT - Parallel Simulation Tool Technical Report*

Dieter Kranzlmüller, Siegfried Grabner, Jens Volkert
GUP Linz
Johannes Kepler University Linz
Altenbergerstr. 69, A-4040 Linz, AUSTRIA/EUROPE
Phone: +43 (732) 2468 - [9499 | 884], Fax: +43 (732) 2468 - 9496
[kranzlmueller | grabner]@gup.uni-linz.ac.at

*) Deliverable D7V-2 of project CEI-PACT, subproject Work Package 7: "Execution Environment for Parallel Programs", supported by the Austrian Ministry of Science and Research (BMWF), project no. GZ 308.927.

Abstract

The existence of race conditions causes nondeterminism in the program flow of parallel programs. On distributed memory machines such races may occur whenever a process receives messages from other processes in an unspecified order. This leads to the effect that running a program twice with the same input may yield different results. Therefore race conditions can be the most serious faults of parallel programs.

The **M**onitoring **A**nd **D**ebugging environment MAD offers an approach to detect errors in parallel programs for distributed memory computers. The feature of visualizing race condition candidates (these are places where races may occur) and the manipulation of the order of these candidates with ATEMPT (**A** **T**ool for **E**vent **M**ani**P**ula**T**ion) is the basis for the trace driven simulation with the **PARA**llel **SI**mulation Tool PARASIT.

This simulator follows the program flow defined by the source code and the communication graph specified and manipulated with ATEMPT. The detection of race conditions is based on the comparison of the original program behavior and the simulated behavior. Differences in the outputs are symptoms for the existence of race conditions.

Contents

1. Introduction

Although methods of parallel software engineering are steadily improved there are still many possibilities of errors in parallel programs. Due to the complexity of parallel program development and the diversity of feasible bugs debugging tools are necessary to eliminate as many errors as possible. While the debugging of sequential programs is sufficiently supported by tools and methods, the development of parallel programs has proven to be more difficult. In our context parallel programs consist of several instances of sequential code which communicate through a message passing interface. Therefore a debugger for parallel programs has to include features for debugging sequential code as well as functionality for investigating the communication between the parallel tasks.

This technical report describes the PARAllel SImulation Tool PARASIT, which plays a main role in the detection of errors resulting from communication of concurrent processes. The simulator is part of the Monitoring And Debugging environment MAD which supports debugging of programs for distributed memory MIMD computers like the nCUBE2 [GrVo 93]. The front end of the environment is the monitoring utility EMU which produces tracefiles of an actual program run [KrGr 94]. After visualizing the traces and correcting the simple errors, which are automatically detected, ATEMPT - A Tool for Event ManiPulaTion - identifies race condition candidates. These are the places where race conditions may occur.

Race conditions are the more serious errors of parallel programs. Their existence leads to nondeterminism in the program flow and therefore to unpredictable program behavior, which might lead to wrong results. This makes the debugging process extremely difficult. On shared memory architectures a race occurs when two or more parallel tasks access some shared variable in an unspecified order, and at least one of the accesses is a write access [Helm91]. With the message passing paradigm on distributed memory machines a race condition occurs at each two or more receive-events where the order of the incoming messages is free. While one message contains the expected data another message with unexpected data could arrive before and influence the results after the receive events.

Another problem for the debugger is based on the absence of a global clock on most distributed memory machines. Therefore a global state is non-existent and the order relationships among events in parallel programs are only partially defined.

The influence of the debugger on the observed program also establishes the so called “probe effect“ [McHe 89] or “Heisenberg effect” [SnHo 88]. This means that the instrumentation of the program may lead to a different runtime behavior than the program would have without it.

The simulator PARASIT described in this report includes facilities to cope with all the problems mentioned above. It performs trace driven simulation based on the original source code and the generated tracefiles as well as the manipulated ones. One goal of the simulator is the detection of race conditions.

In section 2 the MAD-environment is introduced as the basis for the simulation tool. The debugging approach is described and the reason for using a simulator is presented. Section 3 discusses the ideas for performing trace driven simulation and how this strategy can be used for the detection of race conditions. In section 4 we present the structure of PARASIT, first for the prototype implementation on the monoprocessor and then the multiprocessor simulator, as well as the functionality of it. Finally section 5 describes the specification of race condition candidates and the verification of real race conditions. We conclude with a summary of the current implementation and an outlook on the future work.

2. The MAD Environment

The MAD Environment is a toolset which supports the debugging process on distributed memory machines like the nCUBE 2 [Grab 95b]. The current implementation consists basically of three tools:

- EMU - Event Monitoring Utility
- ATEMPT - A Tool for Event ManiPulaTion
- PARASIT - PARAllel SImulation Tool

For a better understanding the following sections present a short overview of the environment.

2.1 The Event Monitoring Utility EMU

The front part of the MAD Environment is the **E**vent **M**onitoring **U**tility EMU [GrKr 94]. This tool performs the instrumentation of the investigated source code, starts an actual program run and traces all necessary activities of this program run. The result is a collection of tracefiles, one for each parallel process, which consist of information about the observed events. Events are actions which influence the state of the current program run and take place at positions defined by programming statements in the source code. The most interesting events for debugging the communication behavior of message passing programs are the communication events. These events are generated whenever a send or receive statement is called during the program run. When monitoring a program the interesting information about the events is stored in tracefiles each time an event takes place. These are all the data which are necessary for the post-mortem analysis of the program run.

2.2 ATEMPT - A Tool for Event Manipulation

The tracefiles of EMU are read by ATEMPT, **A T**ool for **E**vent **M**ani**P**ula**T**ion, which visualizes them as a global communication graph [Grab 95a]. This eventgraph consists of vertices which define the occurrence of an event and of arcs which define the sequential part between the events or the communication between corresponding send and receive events. As it is not possible to determine the correct occurrence time of an event due to the lack of a global clock, the correct ordering of the events is obtained using either generated logical clocks [Fidg88] or

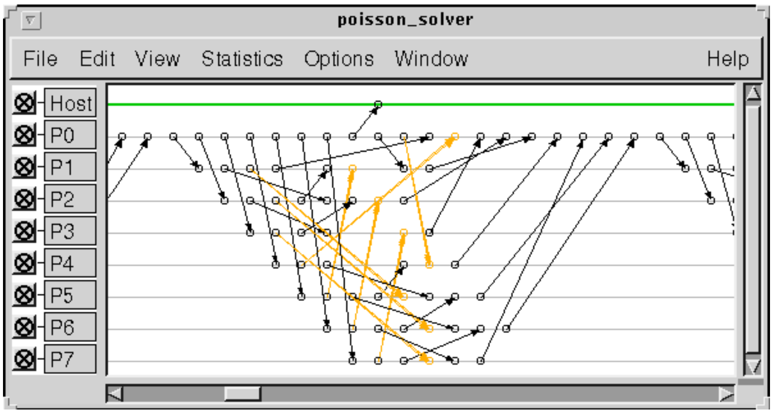

Figure 1. Poissonsolver in ATEMPT

absolute time-stamps generated on each processor. Figure 1 shows an example of an event graph for a parallel poisson solver on the nCUBE 2 distributed memory machine. The horizontal lines show the time axis. Since there is no synchronization between the processors, each time axis is only relevant for one processor. In the example there are 8 processors working on the problem with one horizontal line for each processor and 1 line for the host-program. The host-program is the part which runs on the front end of the nCUBE 2.

The points on the horizontal axes mark the communication events. When using time stamps the distance between two points on one time axis represents the actual time between the two events. The example in figure 1 shows an event graph with logical clocks. Here the event order is visualized in equal intervals, where each interval contains only one event on one processor. This gives a more regular view on the communication graph, but it distorts the actual program run.

The directed edges between two events indicate a communication and the direction of the message flow. The arrow always points to the receiver of the message and therefore to a receive event. The source of the message is the corresponding send event.

Before visualizing the tracefiles ATEMPT performs automatic error analysis in order to detect simple errors within the communication. The simple errors are filtered and colored differently indicating different errors. This offers a way to find bugs like isolated events, where a send event does not have a receive or vice versa, or different lengths of message buffers at corresponding send and receive events (Figure 1, orange arrows).

Another benefit of ATEMPT is the performance analysis facility. This feature includes a statistical overview of the program run (computation time versus communication time) as well as visualization of blocking times at the events where the blocking takes place [Kran 94]. It allows the user to investigate the program for performance bottlenecks as a basis for further tuning and improvements.

The visualization of the communication within ATEMPT is only the basis for further interactive investigations. First of all the user can inspect an event with simply clicking on the appropriate vertex. This opens a window with all the data generated by the monitoring utility, like sender and receiver address, message type, message length and the source code information. This source code connection is combined with a file browser, so that the user can change the code immediately after detecting the origin of an error. Other features for inspecting variables and message queues are also included.

2.3 Race Condition Detection

After the correction of the initial errors further investigations for detecting race conditions can be performed. The interesting question is:

“What would have happened if a message had arrived before another one?“

To help the user with this question, we included the event manipulation features in ATEMPT. The first stage is the visualization of messages, which are possible candidates for the exchange. For that purpose candidates for an exchange are marked with a dedicated color. Afterwards the user can graphically choose the events to exchange, which reorders them and by that changes the communication of the program. The revised event graph is stored in tracefiles again. The original and the revised tracefiles as well as the source code are the input for the PARAllel SImulation Tool PARASIT. This trace driven simulator is the main part of the race condition detection. While following the program flow specified by the source code, the simulator forces the program to follow the user-defined behavior of the revised event graph defined by the tracefiles.

The comparison between the simulated program run and the program run stored in the tracefiles, as well as the comparison of their results, gives the simulator a possible solution to the question specified above. The interesting part for the race condition detection is the part that follows the point where the exchange of events has happened. If this part differs from the corresponding part of the original a possible race has been detected. If no changes are observed no races can occur when an exchange of messages takes place at this point. Therefore PARASIT gives the user the possibility to detect race conditions that may be hidden because of the influence of the instrumentation as well as races that appear only with certain data or even sporadically.

The whole debugging strategy as specified above is supported as one toolset by the MAD-Environment. This environment combines the tools in a way to give the user the opportunity to detect and eliminate the errors of parallel programs easily. The concept is based upon a module strategy, which means that each module can be used by its own as well as in a well defined order. The proposed order is sketched in figure 2. It shows the MAD-Environment in respect to the parallel simulation tool PARASIT and its inputs. The sketched assemblence is only one possibility of how to use the tools. Different philosophies could extract some of the tools and use it for different concepts. We believe that all tools can be extended and adapted to other architectures, even for shared memory machines, although some parts strongly depend on the hypercube topology of the nCUBE 2.

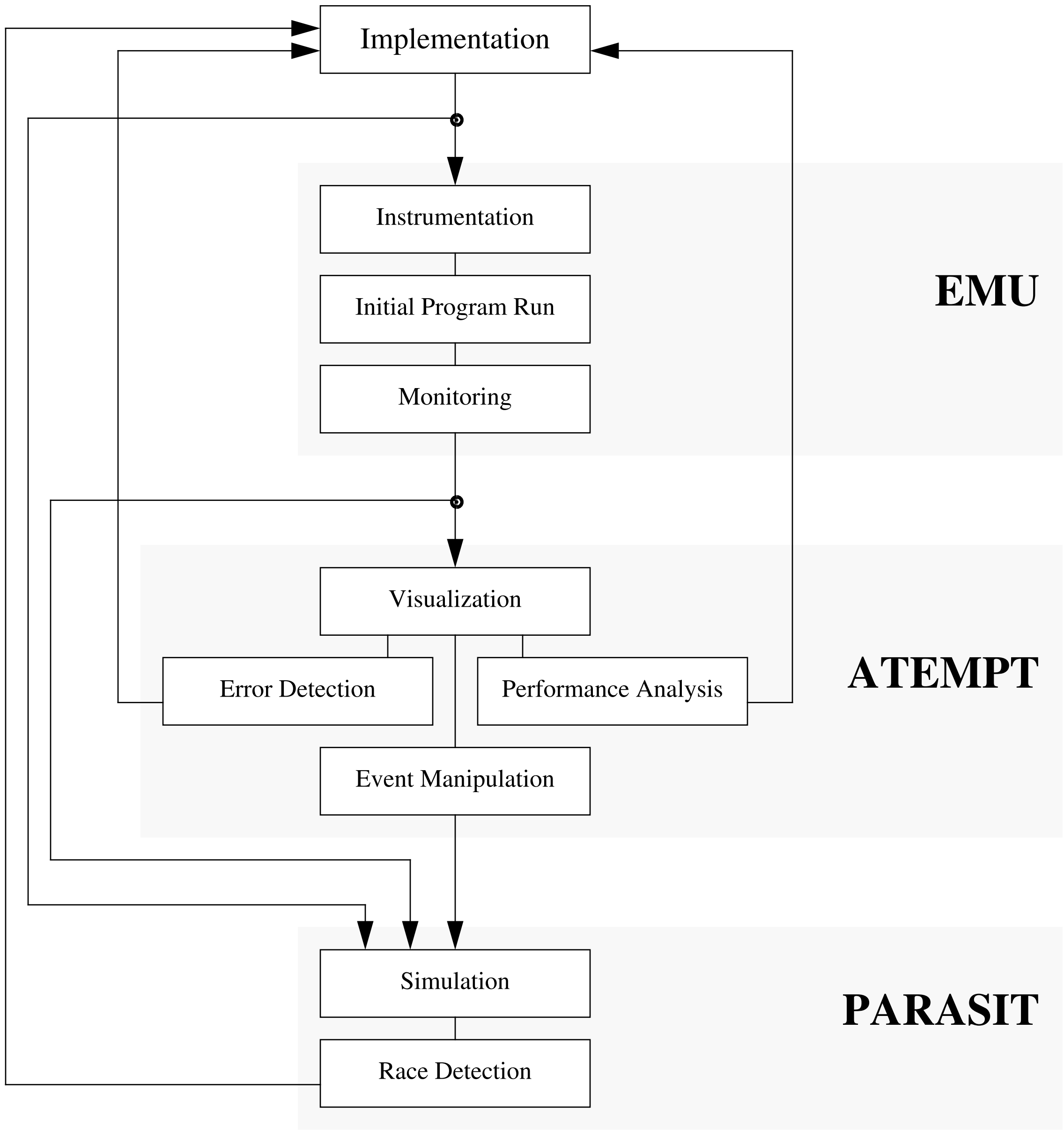

Figure 2. The MAD-Environment

3. Trace Driven Simulation

3.1 Simulation for Detecting Bugs

The process of debugging is usually initiated by a visible failure such as a wrong result or a division by zero. This failure is caused by an error in the program. Through flow-back from the obtained error across preceding errors, debugging has the goal to detect the cause of the erroneous behavior, a so called bug. Therefore the aim of a debugging tool is the guidance in locating bugs of implementations [MiCh 88].

The problem with debugging is the interference a debugger generates on the examined program. The delay caused to perform debugging activities perturbs the parallel program in some way. The most common influence on a program is execution dilation, which means a slowdown in program execution due to the instrumentation. This dilation means usually no problems for sequential programs in a uniprocessor environment. There a slowdown does not affect the computation severely nor does it affect the order of the events because there is only one serial thread of execution. The exception to this rule on uniprocessors are delays through I/O accesses, which can sometimes change a programs behavior.

On multiprocessors the interference is much more sophisticated. Through the multiple threads of control that execute concurrently on several processors and through communication and synchronization between the threads a higher complexity is obtained.

3.2 Equivalent Execution

In parallel programs the dilation can produce a different global order of event execution, which is the sequence of the instructions from all processors in the order of their issue [Kold 91]. For example, if there are two send events on two processors and a third processor is supposed to receive both messages, dilation can result in a revised order of the arrival of the messages. This leads to the effect that successive execution of the same program can lead to different behavior.

One solution for debugging parallel programs is the trace driven replay technique [MiCh 88]. The idea is based on collecting a (representative) trace of a program run during an initial execution. When simulating this trace and following the original program behavior, it is possible to insert breakpoints in the program without disturbing the order of the events and therefore the execution of the program. The basis for the trace driven simulation is an equivalence of the program execution, which can be defined as follows [Leu 90]:

Two executions of a process P are considered to be equivalent if the process P receives the same information from the other processes at the same instants. The instant of an event is defined by the interval in which only this event takes place. This means that two executions of a parallel program will be considered to be equivalent if the execution of each of its processes is equivalent.

The validity of such a trace-driven simulation relies on the assumption that the perturbations to the trace data which is generated by the monitoring process does not affect the simulation results. This includes the fact that the amount of recorded information and therefore the monitor influences the initial execution. The two main goals of any monitoring tool are to cause very little or even no timing overhead and not to change the systems behavior [Mari 90]. For the trace driven simulation the monitor only needs to generate traces for the following primitives:

- Constructs for parallelism.
- I/O accesses.
- Timing instructions.
- State of the random number generator.

The parallelism primitives are all constructs which are needed to transfer messages from one processor to another (communication events like send and receive) as well as constructs for synchronization and global operations. This group is responsible for many errors in parallel programs and the performance of the implementations. The I/O accesses consist of all events which perform input or output in the multiprocessor environment. As this group strongly depends on the operating system, the file system and the interconnection network, its influence on the running program is hard to determine. The timing instructions are important in programs where artificial delays in the program flow are needed. The random number generator influences programs, where the program flow depends on the generated number.

Currently the MAD-Environment only considers the communication events, but future improvements must contain the other groups as well.

3.3 Replay Techniques

When applying trace-driven simulation there are two techniques which can be used. One technique is data driven replay. In this case a trace must contain all data transferred between the processors. The simulation is done by reading the incoming messages for each processor from file and not sending them over the network. The problems with this technique are based on the amount of data and on the fact that the data do not contain the time of arrival of each message. As mentioned above a high amount of data generates a lot of interference during the monitoring process. Due to the absence of arrival times a replay with actual delays is not possible.

The second technique is control driven replay. There the amount of traced data is reduced by the fact that the replay regenerates the data. This means that the simulator actually generates the messages which are transferred over the network and controls the message arrival at their destination. The problem with control driven replay is the ordering of messages, because the regeneration provides no security for an equivalent execution. To solve the ordering problem an event numbering scheme is used. A common idea is the approach with logical clocks which was introduced by [Fidg 88]. There the incoming messages contain the identification of the actual time when they where generated at the source. Then it is possible to exactly determine the corresponding sender and receiver of a message. Messages which are out of order during the replay are stored in a temporary wait queue.

Considering the two replay techniques the advantages and the requirements for equivalent execution are clearly on the control driven replay. Furthermore the trace driven simulation with its control driven replay technique provides more details than analytical models [Kold 91]. Based on an actual program run and on dynamically generated data it provides the most realistic results and overcomes the problem of execution perturbation. Therefore the problem is the generation of the trace data, which is done by EMU [KrGr 94]. As long as the tracing process is kept a minimum the trace-driven simulation produces a valid equivalent execution.

4. Structure of PARASIT

4.1 Events and Event-Attributes

The structure of the trace driven simulator is derived from the ideas described above. First of all the simulator has to follow the information stored in tracefiles. This information consists of events and their attributes, which contain more data about the event. An event is defined as an action without duration that takes place at a specific point in time and changes the state of a process [RiTo 92]. The Event Monitoring Utility EMU [KrGr 92] distinguishes three types of events:

- communication events,
- global events and
- trace events.

The communication events consist of send, receive and test-events. The group of global events are all events concerning all allocated processors like synchronize statements and global arithmetic functions. Trace events are special events which are needed for further analysis. The events for variable inspection and message queue inspection are considered to be trace events. In the context of the trace driven simulation only the group of communication events will be discussed further. All other groups are currently for internal use only.

The attributes of the communication events that are generated by the monitor are needed to perform the analysis. These attributes contain information about

- communication: partner, message type, message length
- timing: logical clocks, timestamps
- source code information.

The data that are transferred between the processes need not to be stored in tracefiles. As mentioned above a simulator that uses the control driven replay technique, reproduces the mes-

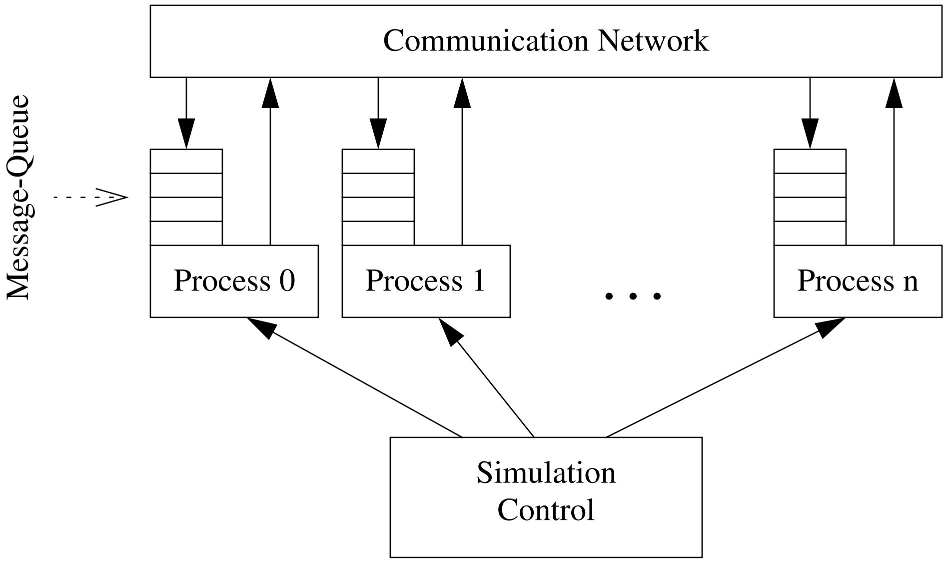

Figure 3. Simulation model

sages during the simulation. To generate the data the simulator must have access to the program parts between each two communication events. This program parts are available through the source code of the program. When considering these tasks and keeping in mind that there exists no communication to other tasks, these sequential blocks are similar to sequential programs. Therefore a traditional sequential debugger can be used to examine the contents and the behavior of these serial tasks. Furthermore this means that a parallel debugging tool must include a standard symbolic debugger to perform error-detection in the sequential tasks between the communication events.

4.2 The Simulation Model

With the events and their attributes the basic functions of the simulator are

- SEND a message
- RECEIVE a message
- TEST for a message
- Execute a TASK.

The communication functions SEND, RECEIVE and TEST must follow the information stored in the tracefiles and combine this information with the data produced by the simulator. The other part of the simulator's job is the execution of TASKs to replay the actual program run and produce the needed message data. Based on this functionality a model of the simulator must consist of the following parts:

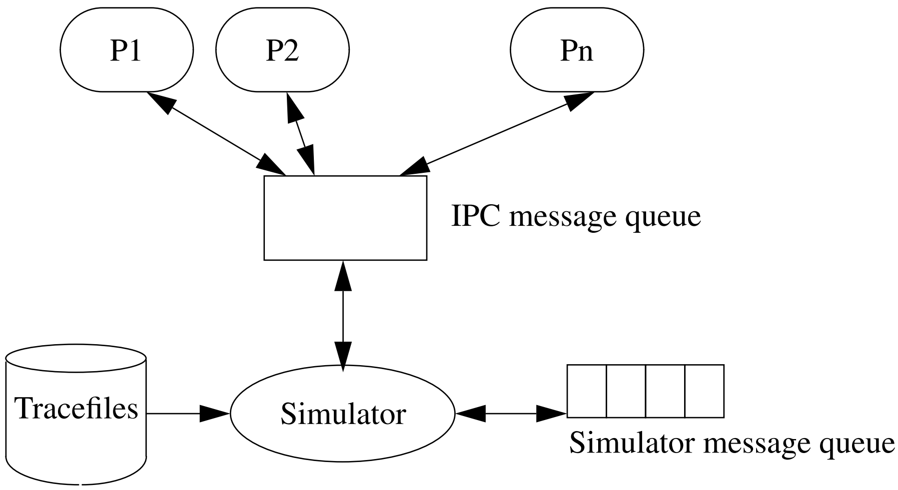

Figure 4. Monoprocessor simulator

- Simulation control
- Communication network
- Simulated processes

The simple simulation model as sketched in figure 3 was used for an implementation on the monoprocessor, a unix workstation, as well as for the multiprocessor simulator PARASIT, which was implemented on the nCUBE 2. The main part of the simulator is the simulation control. Its input is the communication graph that is stored in the tracefiles. The simulators communication behavior must be equal to the communication behavior of the computer system where the initial program run was performed. For the nCUBE 2 this means that each processor can send to each other processor with non-blocking send and blocking receive. The incoming messages are collected in message queues which are part of the communication network.

With the overall communication structure stored in the tracefiles and the possibility to perform message passing on a simulated communication network, the simulation control initiates the execution of the simulated processes in parallel. Each process is running as long as no communication routine is needed. At the point of a send, receive or test the program flow is halted and control is handed over to the simulation control. After the simulated communication event the execution of the task is carried on and the simulation continues.

4.3 Monoprocessor Implementation

There are several ways in how trace driven simulation can be done and how these simulation models can be used in a simulation tool. Our first implementation was produced for a monoprocessor workstation running a UNIX operating system. The advantages and therefore the reasons for this implementations are, that

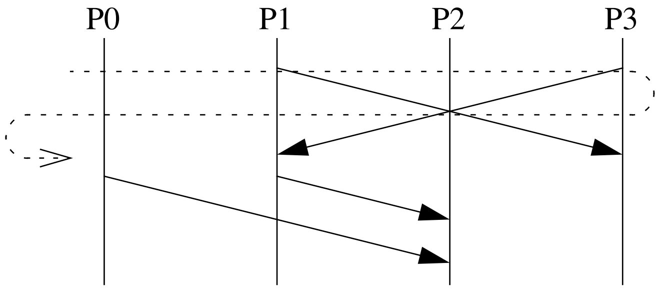

Figure 5. Communication graph for table 1

- workstations are commonly available today,
- CPU costs are not that expensive as on supercomputers,
- the implementation is quite easy.

In figure 4 the structure of the monoprocessor simulator is presented. The main task is the *Simulator*-task. It reads the tracefiles and evaluates the number of processors used in the initial program run. Furthermore it is used as the interface to the user's interactive commands. After the initialization phase the simulator starts as many processes as necessary. Each child process corresponds to one computing process on the original hardware and executes the corresponding node program.

The child processes that are split off the father process are UNIX processes. The communication between these processes is done by an IPC (interprocess communication) mechanism and an IPC message queue. This queue is administrated by the simulator process so that the child processes communicate with each other only via the father process. This guarantees that the user can interfere at any time without losing messages between processes.

The message queue of the simulator itself is used to store the messages sent by the child processes. These messages are sent to the appropriate child at the correct time. Again the order of the messages is generated with logical clocks [Fidg 88].

The main part of the simulator's work is following the communication graph stored in the tracefiles. To perform this task and to simulate the architecture of the nCUBE 2, the simulation control has to fulfill the following requirements:

- The complete communication trace must be processed.
- All processes should run nearly parallel, so that no process is deactivated too long.
- The communication must correspond to that of the real multiprocessor (blocking/ nonblocking communication).
- The setting of breakpoints should be possible.
- The simulator should lead to an equivalent execution of the initial program run.

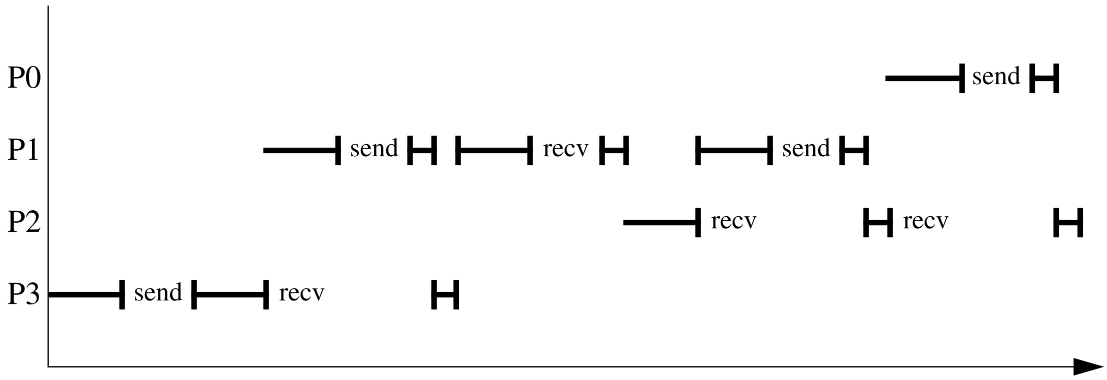

Figure 6. Runtime diagram

These requirements results in a simulation algorithm, which will be described with the following example [Bran 94]. Table 1 specifies the communication trace for 4 processors. The communication graph generated by this trace is visualized in figure 5.

Table 1: Communication trace

Processor 0	Processor 1	Processor 2	Processor 3
Send (P2)	Send (P3)	Recv (P1)	Send (P1)
	Recv (P3)	Recv (P0)	Recv (P1)
	Send (P2)		

After starting and initializing the simulator the algorithm searches in the communication graph for the first occurrence of a receive event, which happens at processor 3 (please note that this is not the order in the trace). To fulfill the requirement that all processors should run nearly parallel, the searchpath is in serpentine from processor 0 across 1, 2 and 3 back to 2, and so on. The searchpath is visualized in the communication graph as a dotted line.

When detecting the first receive the corresponding task is executed until it arrives at this receive statement. During this execution a message is sent to processor 1, which is started as it is the origin of the message to the waiting processor 3. But processor 1 is only executed to the point where the message to processor 3 is sent. Afterwards processor 1 is halted and processor 3 continues because it is the next in the searchpath again. Processor 3 executes its receive and is halted again. The search for the next receive continues as described above. The whole runtme diagram for the example in table 1 is represented in figure 6.

There are several disadvantages of this approach. First of all a normal user may only allocate a certain (small) number of processes in the UNIX operating system. Therefore only the

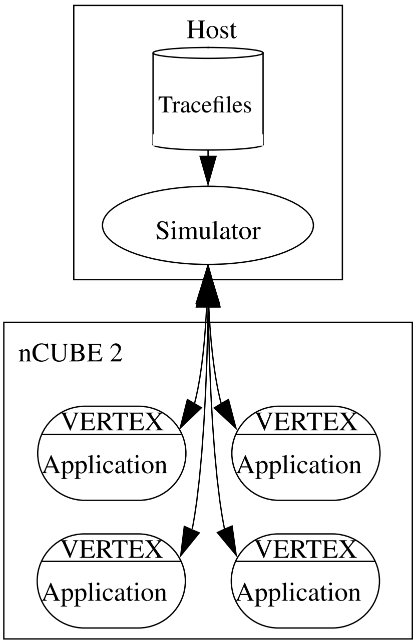

Figure 7. Simple version of multiprocessor simulator

simulation of small hypercubes is possible on the monoprocessor. Another problem is the fact that multiprocessing often needs large data sets. A simulation with many nodes and large amounts of data would easily overload memory and even virtual memory of a workstation.

However the main problem of the monoprocessor implementation is the power of the workstations. With the speed of a workstation the simulation of a problem which is fairly good parallelized for a multiprocessor would need much more time on the workstation. So this approach could never reach a level which would be acceptable for simulating regular problems of parallel processing. This was the reason for the implementation of PARASIT.

4.4 Multiprocessor Implementation

With PARASIT the trace driven simulation is done on a multiprocessor, usually the same as the original program but not necessarily. The functionality is basically the same as the functionality of the monoprocessor simulator. The main advantage is the better performance. Another advantage is that the architecture simulated is easier to rebuild because it is similar to the architecture that the simulator uses. Therefore memory requirements and processor performance would be equal to the initial program run.

On the multiprocessor there is more than one possibility how to implement the simulator. The simplest version is shown in figure 7. There the whole simulator resides on the workstation and communicates with the operating system of the nodes, which is VERTEX, a small kernel version of UNIX on the nCUBE 2.

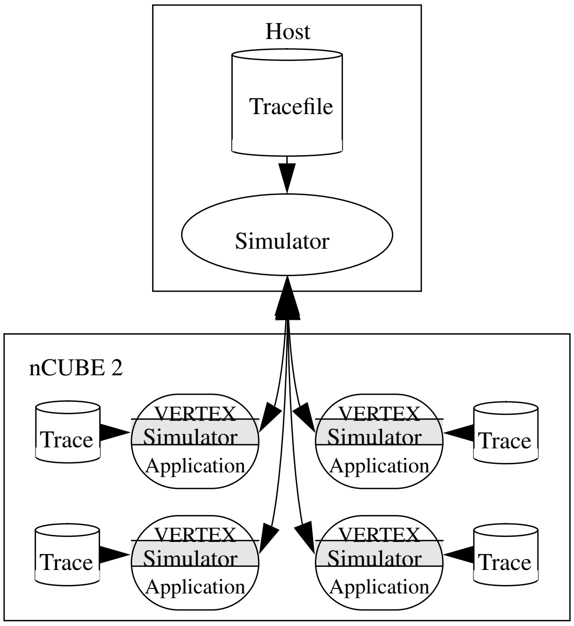

Figure 8. Structure of PARASIT

The simulation can be done step by step as with the monoprocessor simulator. The functionality and the tracefiles are also identical. When investigating this version we encountered that the communication between the host computer and the node processors in the multiprocessor is too slow for a useful simulation. That is the main reason why this approach is not satisfying, too.

A better solution is to include the simulator on the computing nodes, which decreases the host-node communication to a necessary minimum. The only part that still resides on the host-computer is the user-interface. The two functions of the simulator part which runs on the host are

- surveillance of the distributed simulator processes and
- command center for the simulator control.

This means that the host-part concentrates on following the simulation process and visualizing the activities of the simulated processes as well as acting as the interface for interactive commands. The improved structure of PARASIT can be derived from figure 8.

When starting the simulator the process on the host is initialized. The graphical user interface is assembled and the user has the opportunity to select the program and the tracefiles. Then the simulator starts the simulation processes and the application processes, which communicate via IPC message queues.

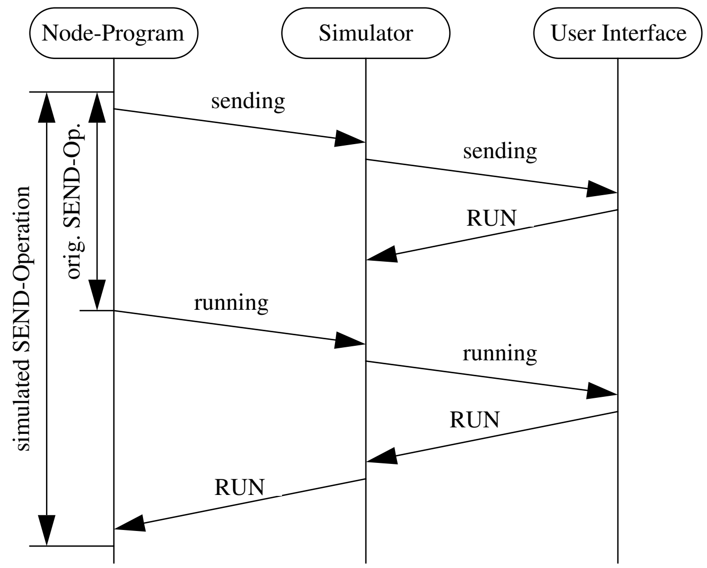

Figure 9. Sequence chart for send-operation

The simulator part on the host visualizes the state of each processor. The state is sent by the simulator process on the nodes to the host and the commands are sent back from the host-simulator to the nodes. Currently the host-simulator defines three modes:

- STEP-mode
- RUN-mode
- KILL-mode.

The selection of one of these modes defines the activity of the simulation processes. The state of the processes can be either *running* or *idle* or the processes are *stopped*. The modes which control the execution are RUN-mode and STEP-mode. In the RUN-mode all the processes are executed from the current position to the end of the program. While executing the processes the simulator visualizes on-the-fly the state of each process using different colors or textual representation.

Another possibility is to use single-stepping of the simulator in the STEP-mode. Such a single-step starts the execution of a sequential task from the current event (send or receive) to the following event. At this point a traditional debugger for the sequential task will be included. Such a debugger can derive all the information from the sequential part as it can be obtained in traditional debugging.

The functionality of the simulator is described with the sequence-chart for a SEND-operation in figure 9. The situation is the following. The node program initializes a SEND-operation. First of all it transmits the status *sending* and some other information (line number, source filename) to the simulator. At this place the verification of race conditions is performed

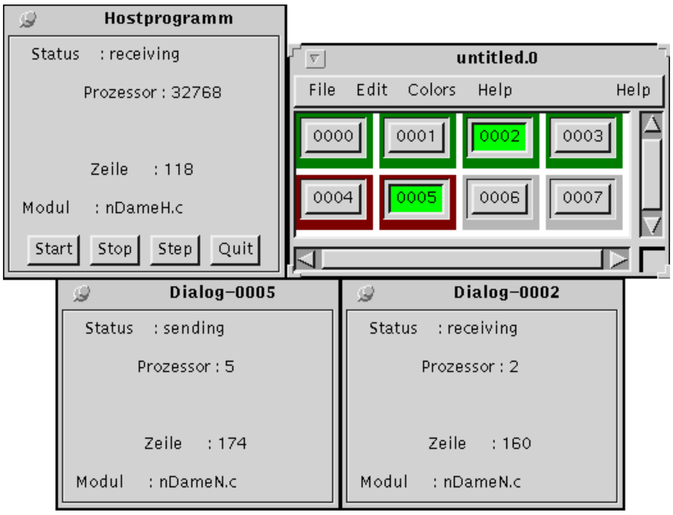

Figure 10. PARASIT - PARAllel SImulation Tool

(see section 6). If everything is correct, the state is set to *running* and the execution proceeds. After the execution of the SEND the node program receives the next command from the simulator, which is either program stop or further execution. This is exactly the place where variable inspection takes place because now it is possible to send this information from the node-program to the simulation-process.

With this version of the simulator a problem that still remains is that the source code of the investigated program has to be recompiled to include the simulator calls. However this is not too problematically because at the stage of the simulator all simple errors have already been corrected.

4.5 Functionality of PARASIT

With the information about the structure of the simulator as stated above the usage of PARASIT and the basic functionality will be described in this section. First of all the simulator is supported by a graphical user interface which has the aim to help the user investigating all points of interest quite easily. The user interface combines all functionality of PARASIT and follows the *look-and-feel* philosophy to allow the usage without a lot of prior knowledge of the underlying implementation.

The host system of the simulator is a UNIX-workstation, which uses X-Windows as window system. We have chosen the ET++ Application Framework [Wein 89] for all the tools, which are integrated in the MAD-Environment. ET++ is based on C++ and includes a OOP-class hierarchy with graphical objects for user interface design. The tasks of the graphical user interface for the simulator are the following:

- presentation of the states of all node processes and of the host program
- interface for user activities
- control of the simulator.

For the visualization of the processor-states the simulator offers two opportunities. First of all there is a global view of the processors and their activities. All allocated processors of the multiprocessor architecture are presented as rectangles. Different colors are used to represent the states. Figure 10 presents a screenshot of PARASIT with 8 nodes of a 3-dimensional hypercube. The upper right window presents the global view. The nodes are surrounded by different colors indicating different activities. In the example of figure 10 the nodes 0, 1, 2 and 3 are currently receiving a message and the nodes 4 and 5 are sending a message, whereas 6 and 7 are running. The display is animated by the simulator process so the colors change as soon as the activities of the simulated processes change. Currently there are the following five colors defined:

- grey: process is running
- green: process is receiving
- red: process is sending
- black: the simulation and the original program run are not equal
- white: an error occurred

Another feature is included in the process visualization itself. The colors mentioned above are only valid for the surroudings of the rectangles. The center of the rectangles present a button for another information window. When clicking on this action-button a window with more details about the selected process is opened. In figure 10 the upper left window gives a textual representation of the running host program. It contains the state of the host, the processor number (which is defined 32768 for the nCUBE host) and the source code connection. This is the corresponding line number (here: 118) and sourcefile name (here: nDameH.c) for the current event.

The two lower windows present the textual information for the selected processors 2 and 5. The information about the state is the same as the color surrounding the action button for processor 2 or 5. This second form of information is only useful for a small group of processors and therefore opened only on user demand. The host program window is always opened because it currently contains the only possibility to communicate with the simulator.

The functionality of the current version of PARASIT is a subset of the intended functionality, but it includes the main features. Additional features will be included to provide further improvements in the debugging process (see section 6).

5. Race Condition Detection

5.1 Event Manipulation

The main reason for implementing a trace driven simulator was the detection of errors like race conditions. These are the most critical errors of parallel programs and they are very hard to detect. A race condition on distributed memory machines may take place whenever the order of two or more incoming message ist not deterministic, and the reults depend on the right ordering of these events. If the order of the incoming events is unpredictable, the program flow is not deterministic. These programs are hard to debug because their behavior is probably not reproducible.

The basis for the race condition detection is the event manipulation feature of ATEMPT. The first step is the evaluation of *race condition candidates*. These are all events where an exchange could take place. At all other events no race conditions can occur. In a first approach the events are only visualized due to their logical clock ordering, so there is no chance of using the actual time stamps of the events. The only criterions for selecting candidates are the parameters of the communication function, which are only the receive statements for incoming messages. A common receive statement can be defined as follows:

receive (buffer, size, source, type)

The first two parameters specify the input buffer and the size of this buffer or the size of the message received, respectively. The *source* parameter defines the origin of the message. The message *type* parameter specifies the contents the purpose of the message, respectively. Usually the *source* and *type* parameters define exactly one processor and a certain message type. It is also possible to specify an arbitrary source processor and/or an arbitrary message type. This allows the program to receive a message from any processor containing any type of message.

Another fact is important for the definition of race condition candidates. The nCUBE 2 communication hardware performs the message passing between the processor based on a special cut-through routing mechanism which is called *wormhole-routing* [NiMc 93]. The path creation follows the *e-cube algorithm* [DaSe 87] which is a right-most selection algorithm. With this algorithms the message passing is deterministic and each two messages from one processor to another must arrive in the same order as they are issued by the sender. This means that overtaking of messages in nCUBE 2 networks is impossible. Therefore this situation needs not to be considered for the detection of race conditions on this particular machine.

5.2 Candidates without Time-Stamps

Without the actual time-stamps the race condition candidates are only determined through the parameters of the common receive statement as specified above. The candidates simply depend on the source and the message-type parameter of any two successive receive-events:

receive1 (... , source1, type1)
receive2 (... , source2, type2)

The detection of race condition candidates is therefore based on the fact that an exchange of events can only occur if the receive parameters allow an unspecified sender and/or an unspecified message type.

Considering the two receives above and declaring the parameter *source1* as any fixed processor-id and the parameter *type1* as any fixed message type, then a race condition candidate can be retrieved from table 2:

Table 2: Race condition candidates

source2	type2	candidate
fixed but not equal source1		No
equal source1	fixed but not equal type1	No
	equal type1	No (because overtaking is not allowed)
unspecified	equal type1	Yes, message type equal
unspecified	unspecified	Yes

Therefore the detection of race condition candidates in a first approach only depends on receive-statements where either the origin is unspecified and the message type is equal or both parameters are unspecified in at least one of the two receives.

5.3 Candidate-Evaluation with Time-Stamps

The extension to the first approach to race condition detection is to include timing measurements. These timing measurements are produced by the Event Monitoring Utility and are only generated on demand. The reason for this is that the timings produce overhead which is not necessary for all debugging activities. The measurements are time-stamps that are generated either before the communication event or before and after the communication event (see [KrGr 94] for further details about the trace-data).

With the time-stamps in the tracefiles and the knowledge about the communication behavior of the target architecture [Kolm 93] it is possible to decrease the number of race condition candidates evaluated with the first approach. The time requirements of a sequential task can be calculated from the start- and finish-times of the events. Each sequential task between two communication events needs the time between the finish-time of the first event until the start-

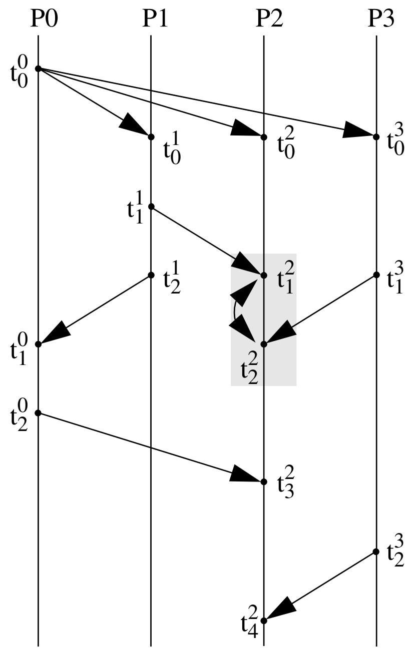

Figure 11. Exchange of race candidates

time of the second event. Therefore the evaluation of these times can be performed during the static analysis of the communication graph. Using this timing information it is now possible to provide the user with more detailed information to guide his decision where to exchange events.

Consider the events traced for processor P2 and visualized as an event graph in figure 11. The superscripted indices show the processor number whereas the subscripted indices indicate the chronological ordering of events on a certain processor. The decision for two exchangeable events is now based on the selection without time-stamps and a specific threshold ε. If there exist two events t_i and t_j on a single processor where $|t_j - t_i| \le \varepsilon$ then these two events are possible candidates for the occurrence of race conditions. Considering the events on processor P2 then $|t_2 - t_1| \le \varepsilon$ holds for a certain ε. Therefore these events will be highlighted as possible candidates. Events t_3 and t_2 or t_4 and t_3, all on processor P2, do not fulfil this requirement.

The main idea behind this concept is that the smaller the amount of time between two events the more likely is the appearance of a message race there. Nevertheless it may not be assumed that events that do not fall into this category cannot be candidates for race conditions.

With the events determined on this basis the event graph manipulation tool ATEMPT gives the user the graphical view on the candidates as sketched in figure 12. The main window shows the event graph for a parallel implementation of a gaussian elimination algorithm. The

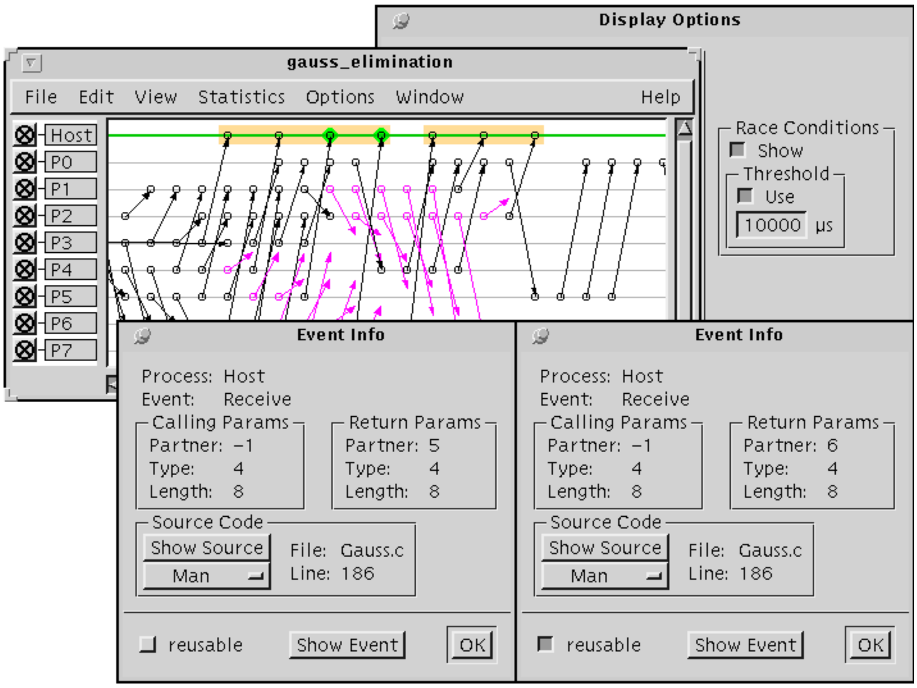

Figure 12. Race Condition Candidates in ATEMPT

events which are candidates for an exchange are all on the host-computer and they are marked with an orange bar. First there is a group of four exchangeable events and then a group of three events. The upper right window shows the display options for the threshold and that the race condition candidate detection is enabled. The lower two windows give more detailed information about the selected events (green dots). Both events are receive events from any (-1) processor by the host and both events contain the same message type (4). Based on table 2 and on the threshold these events have to be marked as race candidates.

After the visualization the user can graphically change the order of the events and store the manipulated communication graph in tracefiles for the trace driven simulation. Then a simulated program run has to be initiated which generates a new communication graph. The interesting part of the event graph is the part that follows the point of the exchange. If the simulated event graph differs from the corresponding part of the old communication graph or the results evaluated after the exchange are different, a possible message race has been detected. If no changes are observed no races can occur when an exchange of messages takes place at this point. Past experience has shown that these assumptions are true for almost all race conditions. At the moment it is not certain that they are valid for all possible race conditions.

6. Conclusion and Future Work

The trace driven simulator PARASIT is a tool which performs control driven replay of parallel programs for distributed memory multiprocessors. To ensure a fast simulation in a powerful environment the whole simulator is implemented on a distributed memory multiprocessor itself. The current implementation allows an equivalent execution of parallel programs. Other features of conventional debuggers are not available at present but will be integrated in the future.

The feature of equivalent execution is necessary to reproduce the actual behavior of a parallel program in terms of process communication and message arrival times. It is the main part of the race condition detection which uses the event manipulation features of ATEMPT - A Tool for Event Manipulation - to specify a different ordering of incoming messages and therefore to specify a different arrival of racing messages. These race conditions are the most critical errors of parallel programs because they may lead to nondeterministic behavior, which may yield different results in successive program runs with the same input data. Such errors are hard to detect and even harder to debug. The tool PARASIT presents an approach to this hard task of parallel program debugging under a look-and-feel user-interface.

The current implementation of PARASIT has proven to be useful but further improvements will be made to introduce different views of the simulated program run and to overcome problems due to runtime- and memory-consumption. A major improvement will be the introduction of CUTS. This technique will permit single-stepping and breakpointing of parallel programs without removing the demand for an equivalent execution. The use of CUTS is also needed for partial replay in time (simulation time) and space (processor-allocation) to reduce the complexity of the debugging process.

7. References

[Bran 94] P. Brandstetter, “Ein Simulator für Parallelrechner mit verteiltem Speicher”, diploma thesis, GUP Linz, University Linz, Austria (Aug. 1994).

[DaSe 87] W.J. Dally, C.L. Seitz, “Deadlock-Free Message Routing in Multiprocessor Interconnection Networks”, IEEE Trans. Computers, C-36 (5) pp. 547-553 (May 1987).

[Fidg 88] C.J. Fidge, “Partial orders for parallel debugging”, Proc. of Workshop on Parallel and Distributed Debugging, ACM, pp. 183-194 (May 1988).

[GrKr 94] S. Grabner, D. Kranzlmüller, “A Comparison of Monitoring Strategies for Distributed Memory Computers”, Proc. of CONPAR 94 - VAPP VI Conference, Linz, Austria, pp. 66-75 (Sept. 1994).

[Grab 95a] S. Grabner, D. Kranzlmüller, J. Volkert, “Debugging parallel programs using ATEMPT”, Proc. of HPCN Europe 95 Conference, Milano, Italy (May 1995).

[Grab 95b] S. Grabner, D. Kranzlmüller, J. Volkert “Debugging of Concurrent Processes”, Proc. of Euromicro PDP 95 Workshop, Sanremo, Italy (Jan. 1995) [to appear].

[GrVo 93] S. Grabner, J. Volkert, “Debugging Parallel Programs using Event Manipulation”, Proc. 1st Intl. Meeting on Vector and Parallel Processing, Porto, Portugal (Sept. 1993).

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

[Kold 91] E.J. Koldinger, S.J. Eggers, H.M. Levy, “On the Validity of Trace-Driven Simulation for Multiprocessors”, Proc. of the 18th Annual Intl. Symp. on Computer Architecture, Toronto, Canada (May 1991).

[Kolm 93] R. Kolmhofer, “Kommunikation in Parallelrechnern mit verteiltem Speicher”, diploma thesis, Institute of Computer Science, University Linz (May 1993).

[Kran 94] D. Kranzlmüller, S. Grabner, J. Volkert, “Combining the Benefits of Error-Debugging and Performance Evaluation”, Proc. of PCAT’94 Conference, Wollongong, Australia, pp. 79-88 (Nov. 1994).

[KrGr 94] D. Kranzlmüller, S. Grabner, “EMU - Event Monitoring Utility”, CEI-PACT Deliverable D7V-1, Linz, Austria (July 1994).

[Leu 90] E. Leu, A. Schiper, A. Zramdini, “Execution Replay on Distributed Memory Architectures”, Proc. of 2nd IEEE Symp. on Parallel & Distr. Processing, Dallas, TX, pp. 106-112 (Dec. 1990).

[Mari 90] D.C. Marinescu, J.E. Lumpp Jr., T.L. Casavant, “Models for Monitoring and Debugging Tools for Parallel and Distributed Software”, Journal of Parallel and Distributed Computing 9 (2), pp. 171-184 (1990).

[McHe 89] Ch.E. McDowell, D.P. Helmbold, “Debugging Concurrent Programs”, ACM Computing Surveys 21 (4), pp. 593-622 (Dec. 1989).

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

[NiMc 93] L.M. Ni, P.K. McKinley, “A Survey of Wormhole Routing Techniques in Direct Networks”, IEEE Computer, Vol. 26, No. 2, pp. 62- 76 (Feb. 1993).

[RiTo 92] M. van Rick, B. Tourancheau, “The Design of a General Parallel Monitoring System”, Programming Environments for Parallel Computing, IFIP, North-Holland, pp. 127-137 (1992).

[SnHo 88] D.F. Snelling, G.-R. Hoffmann, “A comparative study of libraries for parallel processing”, Proc. of the Intl. Conference on Vector and Parallel Processors, in: Computational Science III, Parallel Computing, Vol. 8 (1-3), pp. 255-266 (1988).

[Wein 89] A. Weinand, E. Gamma, R. Marty, “Design and Implementation of ET++, a Seamless Object-Oriented Application Framework”, in Structured Programming, Vol. 10, No. 2, (1989).