Performance Evaluation by Simulation

Helmut Hlavacs[1]

Christoph W. Ueberhuber[2]

[1]Institute for Computer Science and Business Informatics
University of Vienna
hlavacs@ani.univie.ac.at

[2]Institute for Applied and Numerical Mathematics
Technical University of Vienna
christof@aurora.anum.tuwien.ac.at

November 2001

AURORA TR2001-15

The work described in this report was supported by the Special Research Program SFB F011 "AURORA" of the Austrian Science Fund.

Abstract

In this report, the paradigms for simulating the performance of parallel hardware and software are investigated and the principles of discrete event and parallel and distributed simulation are explained in this context.

The results of a comprehensive study of simulation tools for parallel computers are presented.

Contents

Chapter 1

Introduction

In the past, several different methodologies for evaluating the performance of computer systems in general (Jain [33], Menasce, Almeida, Dowdy [45]) and parallel computers in particular (Hu, Gorton [32], Jonkers [35]) have been developed, all suffering from particular drawbacks.

Analytical models include, for instance, queuing networks, Markov chains (Bolch, Greiner, de Meer, Trivedi [10]), and Petri nets (Reisig [55]). Although different tools for automatic evaluation of these models exist, severe problems still remain. Firstly, solutions for these models are very difficult to derive. Secondly, these models may be based on simplifying assumptions which often do not reflect the modelled real systems. Thirdly, analytical models often cannot catch the possibly complicated structure of the simulated environment or the parallel programs.

Comparison by executing benchmark programs or real applications on target platforms and measuring their performance is restricted to available parallel computers only, the program behavior on different platforms, interconnected with different networks, cannot be obtained. Also, the execution of parallel programs might use up large amounts of CPU time, thus consuming computing time possibly needed otherwise. This prohibits large test runs on mission critical production systems.

Simulation tries to bridge the gap between those two approaches. Computer systems and programs may still be represented by mathematical models, yet the simulation environment allows to include features intractable by mathematical analysis only. Also, the program behavior can be observed on platforms behaving differently, and choosing any number of processors, any communication network, and any memory system when evaluating parallel systems. On the other hand, creating accurate simulation models for parallel systems requires much time and effort. Also, care must be taken when implementing the software and interpreting the results.

Evaluating computer systems by using simulation has been treated in the literature extensively (Jain [33], Banks [7]). In this report, an overview over simulation techniques in general and methodologies for using simulation for assessing the performance of both hardware and software of parallel computers is presented.

Simulation is a means for investigating the properties of entities indirectly by investigating a model representing these entities rather than the entities themselves. In order to obtain meaningful estimates for the desired properties, the models used must represent the modelled entities as close as possible. Modern simulation is carried out by using computers. In this case, mathematical models

are defined by a priori parameters describing the model properties, and represented by the changing values of variables stored in the computer memory during the simulation run. Variables representing the state a model are also called *state variables*.

The simulation program carries out calculations on these variables, observing them during the simulation run yields the desired results numerically. There are several types of simulation, depending on whether the notion of time exists in the model or not (Ferscha [23], Jain [33]).

1.1 Monte Carlo Simulation

In this paradigm, the notion of time does not exist. Rather, probabilistic phenomena are modelled that do not change in time. A well known example for Monte Carlo simulation is given by the approximation of certain definite integrals with integrands whose antiderivative cannot be represented by a finite number of terms. In the Monte Carlo approach, the approximation is calculated by creating a large number of random numbers with distribution defined by the integrand. An integral approximation then can be calculated by computing the empirical mean, whereas an estimated for the error can be calculated by computing the variance of these numbers (Krommer, Ueberhuber [36]).

1.2 Continuous Event Simulation

In this paradigm, the model is said to change at every single instant of time. In Jain [33], continuous event simulation is defined by using state variables with continuous values. Usually, models driving continuous simulations are represented by ordinary or partial differential equations.

1.3 Discrete Event Simulation

Following Jain [33], discrete event simulation (DES) is given when using state variables with a discrete set of possible values. Usually, discrete event simulation is equivalently defined to change the model state at discrete time points only (Ferscha [23], Banks [7]), in contrast to a continuous state change. In the remainder of this report, only discrete event simulation will be considered.

Chapter 2

Discrete Event Simulation

Fig. 2.1 shows the evolution of the simulation time, also called *virtual time*, and the model state when using discrete event simulation.

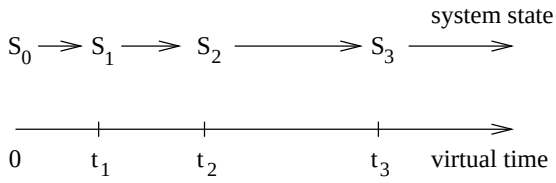

Figure 2.1: Evolution of time and model state in discrete event simulation.

Starting in state S_0, at the virtual time points

$$t_i, \quad i = 1, 2, 3, \ldots$$

the model changes into the states

$$S_i, \quad i = 1, 2, 3, \ldots.$$

It follows that there is no state change in between two successive time points t_i and t_{i+1}. Instead, state changes happen instantaneously.

State changes are carried out by executing *events* scheduled to happen at the respective virtual time points t_i. An event is also said to have a *timestamp* of t_i. Usually, executing an event is done by executing a simulator subroutine associated with this event. This subroutine is often referred to as an *event handler* and a pointer to such a handler is often stored in the data structure representing the event. After having executed an event, discrete event simulation tools usually carry out the following tasks.

1. Choose the next event to be executed. This is the event scheduled to happen at the smallest time point.

2. Move the virtual time to this time point.

3. Execute the event handler.

Usually, events and pointers to their handlers are stored in a so-called *priority queue*, a data structure enabling the following two operations.

Insert (in event *e*, in time *t*). The call `Insert (event e)` inserts the event e into the data structure with respect to the order imposed by time t.

Retrieve (out time *t*). The call `Retrieve()` removes the event with smallest execution timestamp from the data structure and returns it to the caller. Also, the timestamp of e is passed to the output variable t.

Data structures enabling these operations include ordered linear lists, binary heaps, indexed linear lists, calender queues, binomial queues, Fibonacci-Heaps or the Fishbone priority queue (Jain [33]).

Often, direct access to the event queue is forbidden. Instead, a special simulator part called scheduler is responsible for managing events and the event queue(s). Schedulers then may offer the following or a similar interface to event handlers and other parts of the simulation tool:

ScheduleAt (in event *e*, in time *t*). Calling `ScheduleAt(event e, time t)` schedules the event e at time t (or alternatively at time $t_n + t$, if t_n denotes the virtual time when this function is called). e usually denotes a data structure containing a description of the event and a pointer to the respective event handler function.

In addition, this function also makes sure that the execution time of e is equal or larger than the current simulation time t_n.

Run (). This function represents the main simulation loop as described by the following meta-code:

```
t_n = 0;
initialize the simulation run
e_n = Retrieve(t_n);
while(e_n != 0)
    call e_n->handler();
    e_n = Retrieve(t_n);
end while;
```

In such a scheme, events are scheduled either in the initialization of the simulation run, or by event handlers. During each simulation run, the sequence of executed events then may be recorded for later analysis. The overall goal is to derive event sequences showing the same qualitative and quantitative behavior (with respect to the number, times and types of executed events) as the modelled real system. In order to make sure that the simulation model indeed represents the real system, it must be both verified and validated.

Definition 2.0.1 (Model Verification) *A model implementation is* verified, *if it is proven that its implementation is correct. This includes debugging the simulation code and verifying that all files containing information used by the simulation contain the correct data.*

A verified model, however, may still behave differently compared to the modelled real system and thus may yield useless results.

Definition 2.0.2 (Model Validation) *A model is validated by proving that it is an accurate representation of the modelled real system.*

Once simulation output is obtained, special care must be taken in order to correctly interpret these results. Methods for model verification and validation and how to analyze simulation output is described in Jain [33] and Banks [7].

Simulation models generally can be split into *entities* and *resources*. The meaning of resources is to offer services for entities, usually limiting the number or entities that can receive these services concurrently. Entities are created according to a deterministic, randomized or pre-recorded sequence, and start competing for these resources with each other. Often, entities receive service from several resources, the sequence of consumed services being defined by the resource *topology* being part of the simulation scenario.

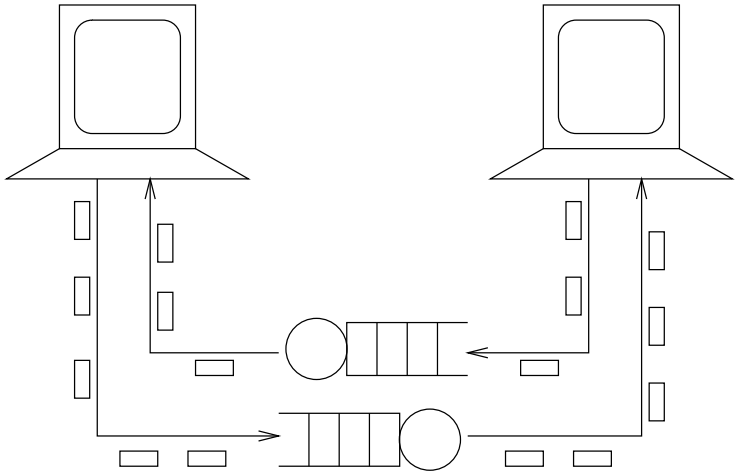

Figure 2.2: Workstations connected by a full-duplex network.

As an example, Fig. 2.2 shows a model for two workstations being interconnected by a full-duplex network. Entities here are given by packets being created inside the two workstation by so-called workload generators (Hlavacs, Kotsis, Steinkellner [29]). The packets are then forwarded to the network model by placing them into a transmit queue. For each direction, there is one shared resource modelling the transmission over the network. Once having been transmitted, the packets travel to the destination workstation where they are destroyed.

Other examples for simulation entities include persons waiting for services for instance in banks, hospitals, or public offices, work items being processed in a production plant or vehicles moving through a network of streets. Possible resources include automatic teller machines in banks, doctors in hospitals, CPUs, hard disks or memory in computers, crossings in streets, or processing stations in factories.

Chapter 3

Parallel and Distributed Simulation

Simulation of complex real systems often implies the creation end execution of millions of events, thus requiring enormous amounts of computational time. Also, due to statistical reasons, for each parameter set, several simulation runs may be necessary in order to compute reliable results.

One possible way for speeding up this process is to utilize several processors instead of just one, resulting in *parallel discrete event simulation* (PDES) or *parallel and distributed simulation* (PADS). The main problem of such an approach is the fact that the execution of events in DES influencing each other must be carried out sequentially. However, there are several levels where a DES system may be parallelized (Vee, Hsu [62]):

Replicated Trials. In this approach, N independent simulation runs are carried out in parallel on P processors. Each run is computed in a separate process, and as intermediate results do not influence each other, no communication between the processes is necessary.

Such an approach can be used, if either runs for different input parameter values are to be computed, or one and the same scenario is simulated several times in order to average results and compute confidence intervals. In case of a large number of simulation runs and similar run-times, this scheme usually results in good speed-ups. If on the other hand the number of different runs is small and run-times vary drastically, only little speed-up can be gained. Also, due to the possibly high memory requirements of each run, this approach may be limited when using shared memory multiprocessors.

Distributed Functions. DES systems depend on various support functions like random number generation, event set processing, I/O, and statistics collection. When independent of each other, such functions may be computed in parallel. However, as the number of independent functions is limited, this scheme may result in little speed-up only.

Distributed Components. This parallelization scheme exploits the fact that simulation models sometimes may be split into several independent components.

Distributed Events. In this scheme, the execution of events retrieved from a global event list is parallelized. The event list may be stored globally in shared memory at one particular processor acting as a central server, or distributed amongst all processors.

Two possible sources for incorrect behavior exist. Firstly, the execution of an event may cancel another event that has already been processed by another

processor. Secondly, an executed event may create a new event that is inserted into the event list prior to already executed events, violating the strict order of execution for events influencing each other. Such violations are called *causality violations*.

Parallelizing Compilers. Optimizing compilers may create assembler code exploiting the low-grain parallelism of modern super-scalar processors, which usually contain several pipelines for integer and floating-point operations. In this scheme, no reprogramming of the DES package is necessary. However, the achieved level of parallelism is usually low.

When parallelizing a single simulation run, the distributed events strategy shows the highest degree of potential parallelism, thus, in the remainder of this report, only this parallelization strategy is considered for PDES.

In PDES, the N processes running in parallel are usually referred to as *logical processes* $LP_0, LP_1, \ldots, LP_{N-1}$. Each logical process consists of the following components (Ferscha [23]).

- A *communication system* (CS) enabling the logical processes to communicate with each other.
- Each logical process LP_i is assigned a region R_i of the simulation model.
- A *simulation engine* (SE) storing a subset of all state variables.
- A *communication interface* handling the communication between the simulation engines.

In such a scheme, events executed by LP_i have either been created by LP_i itself, or have been created by other logical processes LP_j and transmitted over the communication system to LP_i. This interaction between the simulated regions may cause causality violations and special mechanisms must make sure that such violations may either not occur or are properly handled..

A simple version of PDES is given by *synchronous* or *time-stepped* LP simulation. Here, each local clock may only be assigned the discrete values $0, \Delta, 2\Delta, 3\Delta, \ldots, k\Delta$, and events may be scheduled only to these time points. The synchronous PDES then maintains a global clock, and all LPs, setting their local clocks to this global value $k\Delta$, may only execute local events scheduled at $k\Delta$. Causality is inherently guaranteed because events sent to other LPs may not be scheduled in the past, as all LPs have the same local time. Once all such events have been executed, the global clock is advanced to $(k+1)\Delta$ and all LPs may proceed again. LPs having no event scheduled at $k\Delta$, however, remain idle and thus decrease the efficiency or this parallelization scheme.

The main advantage of this approach is its simplicity. The major disadvantages include the high communication overhead necessary for clock synchronization and the inherently limited parallelism due to stalling LPs. Synchronous parallelization is often carried out on SIMD machines.

In *asynchronous* LP simulation, the local clocks may be set to any $t \geq 0$ and LPs are allowed to set their local clocks to different values. There exist two major strategies for guaranteeing correct event execution, called *conservative* and *optimistic*.

3.1 Conservative Simulation

In *conservative* PDES, causality violations are prevented by ensuring that LPs will not receive events from other LPs scheduled to their past. This is achieved by allowing each LP to execute only those events where this constraint can be guaranteed. These events are called *save events*. LPs without save events stall until they receive a wake-up message from another LP. In such a scheme, care must be taken in order to avoid deadlocks.

The first algorithm for deadlock avoidance was introduced by Chandy, Misra and Bryant [14, 15], and was henceforth known as CMB protocol. In this approach, an LP may send *null messages*, containing an empty event and a timestamp, to other LPs, promising that it will not send events scheduled prior to this timestamp. It can be shown that under certain conditions, this protocol prevents deadlocks.

Null messages may be sent after each executed event. A major drawback of this scheme is the possibly large amount of messages sent through the communication system. Advanced strategies thus focus on the reduction of the sent null messages, for example by sending them on a demand-driven basis only.

Another approach for decreasing the number of null messages is to use *lookaheads* (Ferscha [23]). Here, additional information about the internal structure or using pre-sampled random variables can be exploited to further increase the timestamp sent in null messages.

The advantage of conservative strategies is their simplicity in implementation and the smaller amount of memory needed when compared to optimistic approaches (Section 3.2). However, the efficiency of this approach may suffer from stalling LPs waiting for null messages to unlock them.

3.2 Optimistic Simulation

In *optimistic* PDES, each LP may advance its local clock independently of the clocks of all other LPs. Additionally, it periodically saves its state variables to some data structure or log file. Upon the reception of an event from another LP scheduled to be executed in its past (called *straggler*), the LP must carry out a *rollback*, i. e., reload its state variables from the one safe point having the highest timestamp prior to the straggler timestamp, and reset its state and local time to it (see Fig. 3.1). The strategy of resetting the system to a state from the past is also called *time warp* (Jefferson [34]).

Once an LP has been reset to a prior time point, it must make sure that events sent to other LPs in the unrolled time period are annihilated. This is done by

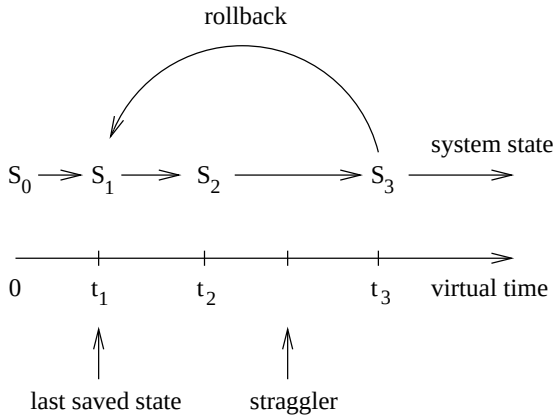

Figure 3.1: Straggler message forcing a rollback to a previous save point.

sending *negative messages* or *anti-messages* for each normally sent event, called *positive message*. Of course, such anti-messages may cause another causality violation at the receiving LP, thus again causing a roll-back to a prior time point. Generally, this scheme may result in a cascade of anti-messages and roll-backs, and it can be shown that the time warp algorithm will eventually remove all causality violations by transferring the simulation to a safe point in the past.

The main advantage of this approach is the possible high speed-up, if stragglers seldom occur. If, however, stragglers occur frequently, much of the simulation may be done in vain and the efficiency of the parallelization may be degraded severely.

Chapter 4

Performance Evaluation of Parallel Computers

When using simulation for assessing the performance of parallel computers, a simulator implementing a model for the parallel computer and the parallel application must be programmed, compiled and linked into an executable. The machine where this simulator is then run on is called the *host machine*. The parallel application that is modelled is called the *target application*, whereas the modelled parallel computer, which may even be unavailable or non-existent, is called the *target machine*.

In the remainder of this report, different approaches for simulating multiprocessors and multicomputers, including PC clusters and networks of workstations are described. Although all approaches use discrete event simulation in one way or another, the used models and workload generators driving the simulation differ significantly.

4.1 Model-Driven Simulation

In *model-driven simulation*, the workload, i. e., the stream of CPU, hard disk, memory, and network requests is generated by a *model* describing the parallel software at an abstract level. In this paradigm, the real software is run only to gather some statistical data describing resource requirements. This data is then analyzed by the simulator user and provided to the program model as deterministic values of random distributions. However, in this paradigm, during the simulation run, no real code of the modelled application is executed. Rather, the application model generates a sequence of events that model the application's behavior.

4.1.1 Generalized Stochastic Petri Nets

A typical example for a model-driven approach is described in Mazzeo et al. [43]. Here, parallel software and hardware is modelled with generalized stochastic Petri nets.

Definition 4.1.1 (Petri Net) *A Petri net is a directed graph containing two types of nodes,* places *(represented by circles) and* transitions *(represented by rectangles). Places can contain* tokens*, the total set of places together with the tokens they contain defines the state the Petri net is currently in. Transitions define the possible state changes, as they are able to create, transport and destroy*

tokens. This is carried out whenever a transition fires. *In this case, tokens in places with an arc leading to the transition are destroyed, whereas tokens are created in places with an input arc from the firing transition. Transitions can only fire, if they are activated, i. e., enough tokens to be destroyed are placed in all input-places, and there is enough space left in the output-places (Reisig [55]).*

Definition 4.1.2 (Generalized Stochastic Petri Net) *A generalized stochastic Petri net (GSPN) is a Petri net containing two types of transitions, called* timed *and* immediate. *A timed transitions has an exponentially distributed firing time, which specifies the time a transition must remain activated before firing. An immediate transition has zero time attached to it and fires as soon as it is activated.*

The set of states of GSPNs can be mapped onto continuous time Markov chains (CTMCs), yielding exact analytical solutions. Due to a possible explosion of GSPN state spaces, however, finding the exact solution may be impossible, and approximative solutions may be computed, for example, by simulating the Petri net (Ferscha [23]).

4.1.2 EPOCA and GREATSPN

For construction of the Petri net, Mazzeo et al. [43] use the Environment for Analysis and Performance Evaluation of Concurrent Applications EPOCA (Donatelli et al. [20]), a workbench containing the Distributed C language (DISC), a version of C appended with CSP-like constructs, and GREATSPN (Chiola et al. [16]), a tool for editing and analyzing GSPNs. Models for the parallel algorithm are first implemented in DISC, which are then converted to an equivalent GSPN representation. Finally, GREATSPN can be used for finding exact solutions or simulate the GSPNs if the state space is too large.

Fig. 4.1 shows a GSPN modelling synchronized communication between two processes. Both input places P_1 and P_3 must contain tokens in order to activate the immediate transition t_1. Upon firing, the input tokens are destroyed and a new token is created in P_5. Whereas t_1 fires immediately after having been activated, T_1 is associated with a random distribution modelling the communication time.

4.1.3 N-MAP

The tool N-MAP [22] follows a similar paradigm, as it allows to specify code fragments or code skeletons defining the *algorithmic idea* only, rather than specifying the whole program. N-MAP is meant to be used early in the program specification process to allow rapid prototyping and to evaluate new parallel algorithms before actually programming the complete program. The evaluation is done by simulating the performance of the provided code skeleton. In order to evaluate an algorithmic idea, the following steps are required.

1. Implement the code skeleton defining the algorithmic idea in C augmented with communication primitives.

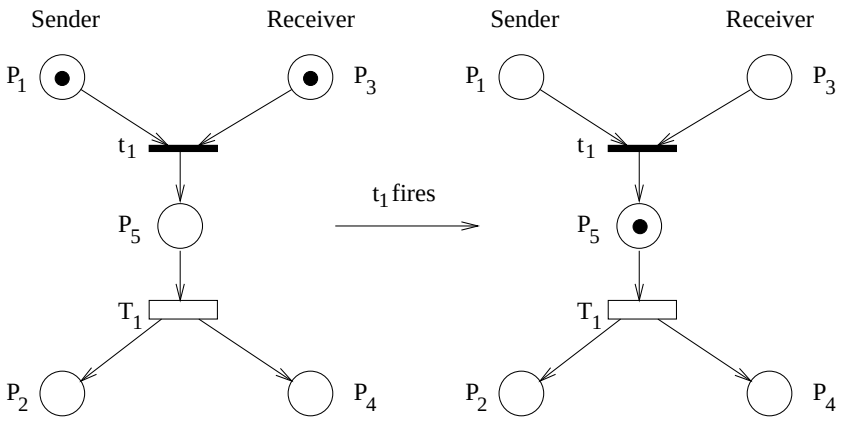

Figure 4.1: GSPN representing synchronized communication.

2. Define the computational requirements of the parallel tasks.
3. Define the communication requirements, including messages sizes.
4. Parse the code skeleton with the source parser YACC. Here, real C code is created.
5. Compile the created C code and create the simulation executable.
6. Run the simulation on either one or several real processors.
7. Collect traces and visualize the results.

Due to the fast evaluation of parallel algorithms, programmers may soon find out whether a parallel algorithm is suitable for a particular parallel machine or not.

4.1.4 OYSTER

Another simulation tool using a programming language for model description is OYSTER (Muller [49]). Models of parallel hardware and software are described by the simulation language PEARL, a special purpose language designed for the simulation of parallel systems only. The syntax of PEARL is C like, but has been augmented with communication and simulation primitives as found in the simulation languages POOL (America, Rutten [2]) and SIMULA (Birtwistle et al. [13]). Like in N-MAP, the program models are first used for creating C source code, which is then compiled and linked to result in the simulation executable which at run-time produces traces for later analysis.

4.1.5 PFPSIM

Whereas the previous models are specified by C like languages, the Parallel Fortran Program Simulator (PFPSIM) by Vassiliou and Taylor [61] simulates the execution time of programs written in High Performance Fortran (HPF). Full programs or code skeletons sketching the algorithmic idea may be used to drive

the simulation tool, which is based on discrete event simulation only, i. e., only the run-time behavior and not the real result is reflected.

4.1.6 PP-MESS-SIM

Yet another input description language has been specified for the point-to-point message simulator PP-MESS-SIM by Rexford et al. [56]. This simulator has been designed to evaluate different network topologies for message passing multicomputers. Thus, it focuses on the network traffic, workload is mainly created by packet generators following given distributions in interarrival time and packet size. The simulation scenario, including the number of nodes, the network type and topology and the workload generators are described by input-scripts following special-purpose grammars, which are later analyzed by the popular lexical analyzer generator FLEX and the parser generator GNU BISON.

4.1.7 DPS

The Distributed Program Simulator (DPS) by Lewis and Bowman [40] requires a so-called *blueprint* of a parallel program for specifying the workload to be simulated. Blueprints consist of a set of *functional units*, the program *shape* being the description of the sequence of calls to functional units, the *texture* being a description of execution details such as CPU utilization or number and sizes of messages, and the *terrain*, a description on how functional units are mapped to the hardware. CPU time required by functional units may be described deterministically or by calling C functions or executing real programs. However, the simulation is still driven by the blueprint defining the program model, thus this simulation tool must be regarded as being model-driven.

4.1.8 PAPS

Like N-MAP, the Parallel Program Performance Toolset (PAPS) by Wabnig and Haring [64] is meant for rapid prototyping of parallel algorithms in early design phases. Models for PAPS are defined by using two different modelling formalisms. The resource layer defining the parallel hardware including the interconnection network is defined by discrete deterministic Petri nets. Each resource type may be modelled by a Petri net basic building block (BBB), being a small template later used for creating one large Petri net defining the topology. Parallel programs, on the other hand, are modelled by so-called acyclic annotated task graphs.

Definition 4.1.3 (Task Graph) *A task graph is a graph describing a set of computational tasks to be computed, together with their interdependencies. A node defines a certain task, an arc between two nodes models either the control flow between successive tasks, if the connected tasks run on the same processor, or data flow, if the connected tasks run on different processors. An annotated task graph additionally contains information about the resource requirements of the various tasks. An example task graph can be seen in Fig. 4.2*

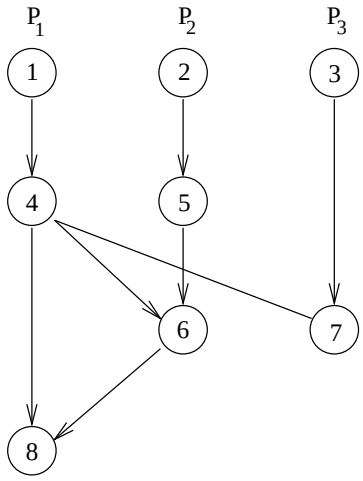

Figure 4.2: Task graph with inter-process communication.

For modelling the complex behavior of computer resources, PAPS exploits the high expressiveness of Petri nets. This way, a compact yet representative library of resource models can be created easily. Task graphs models describing the structure of parallel programs, on the other hand, exploit the fact that such models are easily to be interpreted and changed, yet their expressiveness is sufficient to describe even complex parallel workloads.

4.1.9 EDPEPPS

Yet another approach meant for rapid prototyping is given by the Environment for the Design and Performance Evaluation of Parallel Programs (EDPEPPS) (Delaitre, Justo, Spies, Winter [18]). Here, users may construct application models for PVM programs by using the graphical program representation language PVMGRAPH, resulting in graphs similar to task graphs. The graphs are then transformed into the simulation language SIMPVM, which is used for driving the simulation kernel, itself being based on queuing networks.

4.1.10 PDP Graphs

Finally, Yan, Zhang and Song [66] simulate the performance of networks of workstations (NOWs) using program data-dependence graphs (PDG graphs), which are a generalization of task graphs for modelling parallel programs. The authors also describe a tool covering the whole simulation cycle, from measuring program parameters like CPU demands, to modelling and simulating the program.

4.2 Trace-Driven Simulation

In *trace-driven simulation*, the stream of resource requests driving the simulation is generated by a record containing measured data from real program runs. In order to obtain such a trace, several steps are necessary. At first, the parallel

program (or a representative program skeleton) must be developed. Then, the program must be adapted to generate a trace of its execution. This can be obtained by either inserting explicit tracing statements, or by compiling and linking against special purpose instrumentation libraries which are able to replace key commands like communication calls with their own versions, which during execution in addition to their normal functionality also record relevant computational resource requirements. Fig. 4.3 shows which information is typically stored in traces, together with a sample simulation run driven by this trace. For many trace-driven simulators it is enough to obtain the length of CPU bursts between two adjacent calls to communication routines. When simulating these bursts to occur on the target machine, usually a fixed architecture specific factor is assumed to exist, and the simulator multiplies each recorded CPU burst with this factor. In Fig. 4.3, this factor is assumed to be 0.5, i.e., the target machine is assumed to be twice as fast compared to the machine where the trace was recorded. However, the communication network is assumed to be equally fast. The advantage of this approach is the fact that the events traced and later simulated only encompass CPU bursts and communication occurrences, keeping their number quite low. This usually results in small tracefiles and fewer events to be simulated. Also, this approach has proven to surprisingly accurate. The basic drawbacks include the fact that performance multiplying factors derived for different processors and computers often depend on the respective instruction mix and may vary drastically between different programs, and also between different subroutines inside programs. This may be the case, for example, if parts of a program contain mainly integer operations, while other parts are dominated by floating-point operations.

When sending data to another task, usually the length of the sent message is recorded together with the ID of the receiver. When waiting for a message, it is important to log if the message may be received from any other task, only one particular task, or a particular group of tasks. Upon receiving a message, again the sender ID and the message length must be logged. In general, trace-driven

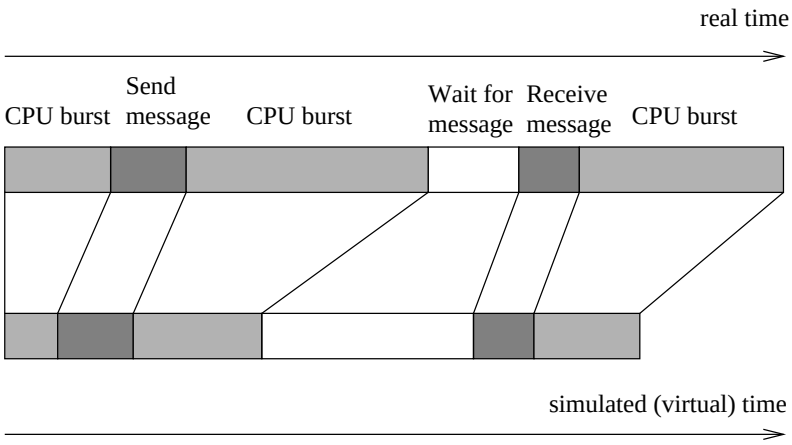

Figure 4.3: A program execution trace.

approaches assume that the communication patterns are deterministic and do

not depend on the run-time situation. This assumption is valid, for example, for routines from the parallel linear algebra package ScaLapack (Blackford [9]). In cases where the communication indeterministically depends on the run-time situation, however, for example when simulating the effect of load balancing mechanisms, in contrast to execution-driven simulation, this approach cannot be used.

4.2.1 HESSE

A typical example for the trace-driven approach is given by the Heterogeneous System Simulation Environment (HeSSE) by Mazzocca, Rak, and Villano [44]. This simulation tool has been developed from an earlier version named Ps (Aversa, Mazzeo, Mazzocca, Villano [3]) and uses execution traces of parallel programs using PVM. Fig. 4.4 shows the steps involved in performance evaluation with the HeSSE simulator. At first, an execution trace is obtained by executing the PVM program or PVM program skeleton on any single-CPU or multi-CPU environment, preferably a workstation. By using a slightly adapted version of the PVM tool Pgpvm (Topol, Sunderam, Alund [59]), a tracefile describing the resource requirements of each single PVM task, including the number and sizes of sent messages, is created. Then, a configuration file must be created, which describes the *components* of the simulated target machine. Also, timing benchmarks or parameter estimations of the simulated parallel computer must be specified in a command file. Eventually, HeSSE may be used to simulate the recorded program run on the target platform. The simulator output may afterwards be used for analysis and report generation.

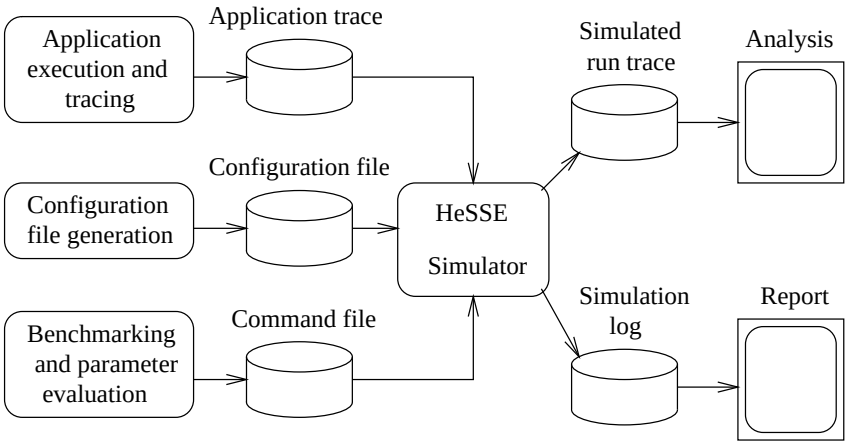

Figure 4.4: Performance evaluation using the HeSSE program simulator.

4.2.2 SPEEDY

Speedy [47] is another performance extrapolation tool that is able to evaluate the performance of parallel programs by trace-driven simulation. Here, programs must be written in the language pC++, a data-parallel extension of C++. Being

part of the Tuning and Analysis Utilities (TAU) (Mohr, Brown, Malony [46]) program and performance analysis environment, SPEEDY works similar to HESSE in the sense that executables may be changed to log away trace files describing the resource requirements of the program. This time, the pC++ compiler itself is able to insert tracing routines into the executable. The obtained trace again is used to drive the simulation engine.

4.2.3 DIP

Another environment for trace-driven simulation of parallel programs is DIP (Labarta et al. [37]). DIP consists of an instrumentalization of several message passing libraries, for example PVM and MPI, the Distributed Memory Machine Simulator DIMEMAS, and the visualization and analysis tool PARAVER. When compiling parallel programs against the chosen message passing library and in addition the DIP instrumentalization, the executable produces a tracefile containing the CPU and communication requirements of all parallel tasks. These traces can then be fed into DIMEMAS to drive the simulation run, producing output to be analyzed by PARAVER.

4.2.4 The Qin-Baer Simulator

A more detailed simulation is carried out by the multiprocessor simulator described by Qin and Baer [53]. In this simulator, accesses to the whole shared memory hierarchy of the cache-only multiprocessor KSR-1 may be simulated. Traces of parallel programs include memory references of all program and data words fetched from the shared memory by the observed program. The simulator then may evaluate different cache sizes and caching strategies. As each memory reference of a simulated program creates an entry in the trace file and a simulation event, the number of events to be simulated is much higher than in the previously described simulators. Also, tracefiles may become very large. On the other hand, the obtained simulation results are usually more accurate than in the previous paradigms.

4.2.5 The Gursoy-Kale Simulator

Yet another approach is described by Gursoy and Kale [27]. Their simulator evaluates the performance of *message-driven* programs written in the language DAGGER for the execution environment CHARM. Like multithreading, in the message-driven paradigm, several processes are assumed to run on each CPU instead of only one. This way, communication can be effectively hidden behind computations, as it is most likely that there is always a runnable process in the processor queue. On the other hand, processes waiting for messages are suspended and woken up as soon as they receive a message. In order to cope with message interdependencies and possible indeterministic message arrivals during simulation runs, traces generated by DAGGER also contain dependence information based

on the program's basic building blocks. This is achieved by static source analysis during compile-time.

4.3 Instruction-Level Simulation

In *instruction-level simulation* or *cycle-by-cycle simulation* each single CPU instruction of a compiled and linked executable is *interpreted* or *emulated* by a special simulator engine. Usually, each instruction of the interpreted application either results in the call of a simulator subroutine or is dynamically replaced by one or several instructions of the simulator executable. Fig. 4.5 shows this situation.

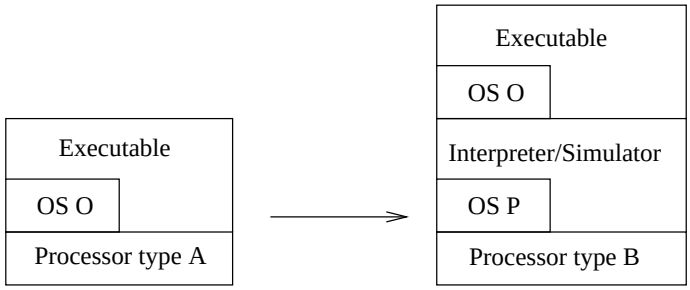

Figure 4.5: Instruction-level simulation replaces every single instruction of the investigated executable (compiled for processor type A and operating system O) with several interpreter/simulator subroutines or instructions (compiled for processor type B and operating system P).

As code of the original application is not necessarily executed directly on the host where the simulation is run on, the simulating host in theory may contain processors and operating systems different to the ones the interpreted application was compiled and linked for. By executing an application on a *virtual* processor and memory system, the resulting simulation timings may be highly accurate, as long as the simulated target computer is modelled exactly. This, however, may not be the case if new processor cores contain different pipelines, or memory systems are adapted by processor vendors.

As each application instruction may result in the execution of several real instructions on the simulating host, simulations of this type usually need vast amounts of run-time for simulating even small applications. The average number of simulator host instructions executed per simulated instructions is called *slowdown*. Depending on the complexity of the simulation engine and the level of abstraction, the slowdown may vary between 10 and 1000, and can in some cases be even as high as $100,000$. Obviously, if slowdowns get too high because of too detailed models, the simulation of complex applications may become impossible. This is even more true if parallel programs consisting of several instruction streams are to be simulated.

Instruction-level simulators show unparalleled accuracy. However, they are difficult to construct (SimICS is said to have required 50 person years) and limited to particular processor instruction sets.

4.3.1 SIMOS

A sophisticated instruction-level simulation tool is given by SimOS (Rosenblum, Bugnion, Devine, Herrod [57]). SimOS simulates the hardware of MIPS based multiprocessors in enough detail to boot, run, and study SGI IRIX workstations, as well as any application running on them. The hardware is modelled in a modular manner, including modules for different versions of MIPS processors, memory management units (MMUs) for virtual to physical address translation and TLB caching, various levels of memory hierarchies, Ethernet, disk I/O including direct memory access (DMA), and consoles. Detailed processor models exist for the MIPS R4000 (called Mipsy) and MIPS R10000 (called Mxs and Flash). Mxs is reported to show a performance of approximately 20,000 simulated instructions per second when executed on a modern workstation, and Mipsy to be faster by one order of magnitude. The Flash model is reported to model the processor at the gate level, thus leading to an very high level of detail, but also to a performance of only a few simulated cycles per second.

4.3.2 EMBRA

Due to the potentially huge simulation times caused by the detailed SimOS models, another processor model called Embra (Witchel, Rosenblum [65]) has been implemented. In this model, instead of calling simulator subroutines, basic blocks of code, i. e., code sequences with no jump or branch instruction, are dynamically translated to equivalent code sequences which change the state of the *simulated* processor, stored in main memory, instead of the real one. To avoid retranslations, Embra stores already translated sequences in a translation cache. In addition, Embra supports full MMU address translation, multiple virtual address spaces, exceptions and interrupts, privileged instructions, I/O communication, DMA, and self-modifying and dynamically generated code. The dynamic translations achieves slowdowns between 2 to 9, thus speeding up the simulation significantly. When running applications on SimOS, is is thus advisable to boot the system with Embra, and use more detailed processor models only for particular code sequences under observation.

4.3.3 MINT

Another instruction-level simulator for MIPS based executables is Mint (Veenstra, Fowler [63]). As input, statically-linked IRIX executables compiled for the MIPS R3000 are required. Instead of keeping the state of the simulated processor in main memory, Mint follows a hybrid approach, where it tries to keep the state of the simulated processor in the real processor as long as possible. This is done by analyzing the object code and synthesizing functions containing straight-line code blocks without interrupts or branches. During simulation, these functions are then called and executed directly. This way, large portions of the code may be interpreted at native speed, thus speeding up the simulation significantly.

4.3.4 TALISMAN

An instruction-level simulator for the multiprocessor architecture Meerkat, based on Motorola MC88110 processors, is given by TALISMAN (Bedichek [8]). Like EMBRA, TALISMAN dynamically translates Meerkat instructions to decoded instructions native to the simulating host. Once translated, instructions are stored in caches to avoid retranslation. For TALISMAN, on the average, a slowdown of 100 is to be observed.

4.3.5 SIMICS

Another instruction-level simulator is SIMICS (Magnusson et al. [42]). This simulator is able to boot and simulate a complete workstation running operating systems and applications for the SPARC V8 processor, including Linux 2.0.30 and Solaris 2.6. The core interpreter of SIMICS is based on the metatool SIMGEN (Larsson, Magnusson, Werner [38]) and implements only 2000 lines of specification. SIMICS does not focus on detailed CPU simulation, but rather on cache, TLB and memory simulation. It also contains models for Ethernet and SCSI.

4.4 Execution-Driven Simulation

In *execution-driven simulation*, the target application is *directly executed* on the host machine to simulate its run-time behavior on the target machine. This is done in such a way that the execution of the target application on the host machine is interleaved with the simulation. Usually, like in reality, P processes are started that consume CPU time and communicate with each other. As an alternative, the processes may be mapped onto P light-weight threads running inside one main simulation process and sharing a global address space.

It must be noted that the actual time to finish the simulation does not affect the simulated run-time. Instead, the simulated run-time is calculated by simulator functions only and is based on the CPU requirements of the executed processes or tasks and the carried out communication and synchronization between them.

There are two event classes for execution-driven simulators (Hu, Gorton [32]). *Local* events, or local computations, have no effect on the behavior of other tasks, and thus need no communication with other instances of the program. *Global* events, also called globally visible events, may change the execution path of the parallel application and thus require synchronization of the simulation. There are usually two types of global events, (i) access to shared memory and (ii) explicit communication by sending messages.

In order to ensure correct timing and that global events are simulated in correct order, the original program is changed in such a way that its tasks continuously call simulator routines in addition to or instead of its normal functions. This process is called *augmentation*. Once called, the simulator functions then drive the simulation.

Augmented code may include functionality for (i) managing virtual time (locally for each task or globally), (ii) synchronization, (iii) communication, or (iv) statistics gathering.

Fig. 4.6 shows the levels at which code augmentation may occur (Veenstra, Fowler [63]).

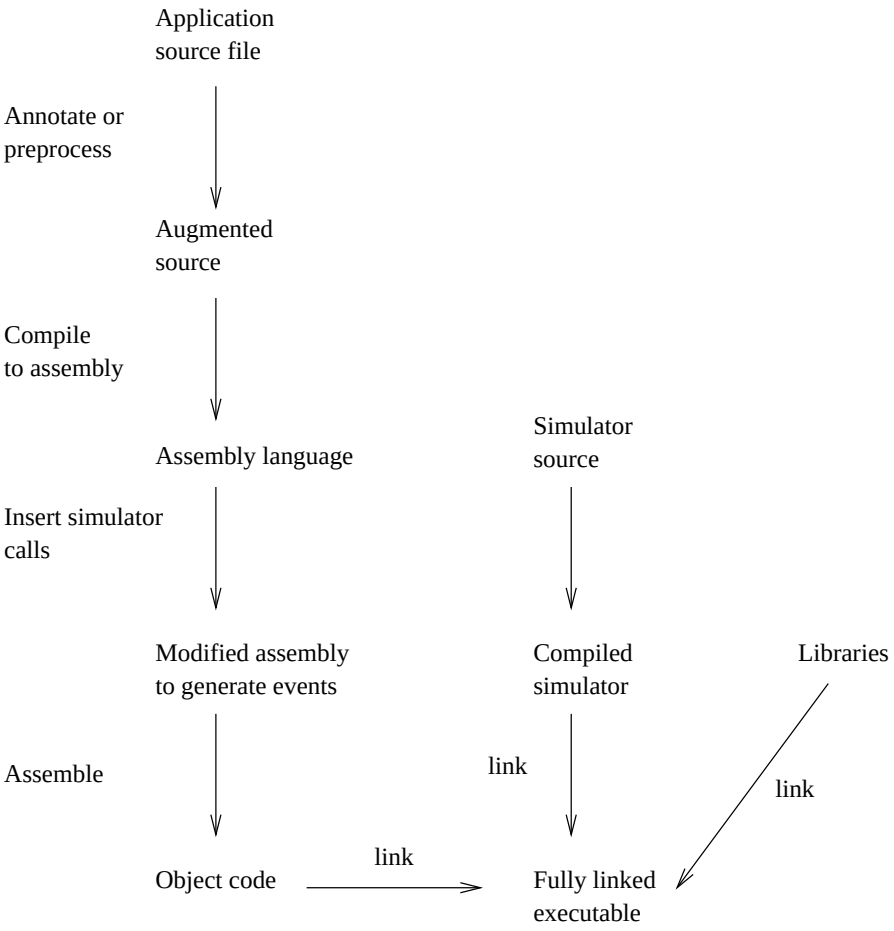

Figure 4.6: Code augmentation for execution-driven simulation.

Code may be augmented at the source level. Popular techniques for automatic source code augmentation include running preprocessors that exchange language constructs or calls to communication libraries with simulator calls, or parsing the source code to find, for instance, accesses to shared memory variables, requiring the addition of explicit simulator calls before and after these accesses. Source code augmentation demands access to the target application source code.

Augmenting code at the assembler level is more complicated and requires access to the assembler code of the target application for the particular processor type and operating system of the host machine. Also, this type of augmentation is not portable between different processor instruction sets, and new augmentation routines must be written for each processor type. As it is generally difficult to read and debug assembler code, augmentations at this level are often restricted to small changes, for instance by including a cycle counter for driving the simulation

time.

Thirdly, the code may be augmented at the object code level. This demands access to a compiled but not yet linked version of the target application.

Finally, the code may be augmented at the executable level. Although it is generally difficult and non-portable to develop code augmentation at this level from scratch, tools exist that enable the construction of executable editors (Larus, Schnarr [39]).

Usually, each process may store its own copy of the virtual time locally. Driving the local virtual time then is usually done at the following points:

1. The consumed CPU time is measured between two successive calls to the simulator. Similar to trace-driven simulation (see Fig. 4.3), this measurement may then be multiplied by a factor depending on the simulating host and the target machine. The result is then added to the local virtual time.

2. When sending messages, the local virtual time is also incremented by a time depending on the length of the message, and the latency and bandwidth of the used network.

3. When receiving messages or synchronizing with other tasks, the waiting time must also be added to the local virtual time.

However, using optimistic approaches (see Section 3.2) is usually difficult, as the complete program state including registers, stack and allocated memory must be saved periodically. This can only be achieved by either changing the operating system accordingly, or by constructing a special purpose compiler for compiling the application (Bagrodia et al. [4]).

Execution-driven simulation has been very popular in the last years, and many execution-driven simulators for assessing the performance of parallel programs have been developed. These include the Rice Parallel Testbed (Dwarkadas, Jump, Sinclair [21]), Lapse (Dickens, Heidelberger, Nicol [19]), Fast (Boothe [11]), Maya (Agrawal, Choy, Leong, Singh [1]), Spasm (Vaidya, Sivasubramaniam, Das [60]), Dert (Frazier, Tamir [25]), Spam (Gefflaut, Joubert [26]), Allegro (Finger, Donnell, Siegelin [24]), Prylli, Tourancheau [52], or Srinivas, Gannon [58]. In the following, some simulators will be described in more detail.

4.4.1 TANGO

One of the first execution-driven multiprocessor simulation systems was Tango (Davis, Goldschmidt, Hennessy [17]). Tango is able to simulate the run-time behavior of parallel programs written in C or Fortran, augmented by the macro package m4 (Lusk et al. [41]) for shared memory communication. After having been recompiled for Tango, it is possible to run all processes on a uniprocessor machine for simulation. Each process maintains its own local time. Local virtual times are advanced by measuring the CPU time consumed by each process. Synchronization is only necessary when accessing shared memory. Tango

implements three different levels of synchronization, which trade accuracy for simulation efficiency.

4.4.2 TANGO LITE

TANGO has been further developed to TANGO LITE (Herrod [28]). Here, each process of the parallel application is represented by a light-weight thread running in one main simulation process. A special scheduler is activated whenever access to a shared memory region is necessary. The scheduler makes sure that the shared memory is accessed in the correct order as would occur during a real execution.

4.4.3 WWT

Another simulator for shared memory multiprocessor applications is the WISCONSIN WIND TUNNEL WWT (Reinhardt et al. [54]). It has been designed to study the performance of shared memory systems based on caches and TLBs. Simulation runs must be executed on the Thinking Machines CM-5, a message passing multicomputer with between 32 and 16,384 SPARC CPUs, itself having no shared memory at all. The WISCONSIN WIND TUNNEL, however, can mimic runs on any target host with shared memory architecture, caches between processors and the shared memory and one process per processor. Access to the shared memory must only be simulated in case of cache misses. This is done by emulating a shared virtual memory. WWT marks blocks of shared memory with error bits, and accesses to these blocks then trap to a software handler satisfying the requests and driving the simulation.

4.4.4 WWT II

The limitations of WISCONSIN WIND TUNNEL to run simulations only on CM-5 multicomputers have been overcome by its second version WISCONSIN WIND TUNNEL II (Mukherjee et al. [48]). Here, the EEL executable editing library (Larus, Schnarr [39]) is used to construct the editor ELSIE. Using ELSIE, timing routines and simulator calls can be put directly into the compiled and linked application executables. Due to the platform independent EEL library, this can be easily ported to all platforms where EEL exists.

4.4.5 AUGMINT

Another simulator for shared memory multiprocessors and the M4 macro package is AUGMINT (Nguyen, Michael, Sharma, Torrellas [50]). Although being based on MINT (see Section 4.3), AUGMINT is execution-driven and represents all parallel processes as tasks inside one main simulation process. Code augmentation is done at several levels. First, the M4 macros are expanded to translate parallel constructs into events and task context switches for AUGMINT. The resulting source code is compiled by the GNU C compiler into assembler code, which is

then automatically augmented by the program AUGMENTER. Here, code for timing and data calculations is inserted.

4.4.6 PROTEUS

An example for a simulator for shared memory and message passing applications is PROTEUS (Brewer, Dellarocas, Colbrook, Weihl [12]). Applications must be written in a superset of C, including extensions for defining shared memory data structures, the PROTEUS library contains calls for message passing, threads management, memory management, and data collection. Applications have been simulated for both the message passing multicomputer nCube and the shared memory multiprocessor Alewife.

4.4.7 TROJAN

The extended version of PROTEUS, called TROJAN (Park, Saavedra [51]), implements different models for virtual memory management. The target application processes are mapped to threads which run inside one simulation process, with the addition of a main simulation thread for scheduling and controlling the correct order of global events. In addition, TROJAN does not require any modifications of the target application source code, but rather inserts its augmentations at the assembler level only.

4.4.8 COMPASS

Another simulator for message passing applications is COMPASS (Bagrodia, Deeljman, Docy, Phan [6]). COMPASS simulates the performance of programs written for the message passing interface MPI and its I/O extension MPI-IO, and consists of various model components. The main simulation component of COMPASS is MPI-SIM (Bagrodia, Prakash [5]). In MPI-SIM, all MPI processes are represented by light-weight threads sharing a common address space. In order to prevent the incorrect sharing of global variables, all MPI programs must be preprocesses at the source level to rename all global variables, making them unique for each thread. This requires the knowledge of the number of threads that will be spawned later. MPI-SIM supports most of the MPI calls, including point-to-point and collective communications. Internally, all collective communication calls are implemented using point-to-point communication, and all point-to-point calls are implemented using a core of non-blocking MPI functions. Sophisticated protocols for parallel and distributed simulation include the Quantum protocol, the null message protocol, the conditional event protocol, and the accelerated null message protocol. Large-scale simulation runs on IBM SP and Origin 2000 systems exhibit slowdowns as low as 2.5.

4.4.9 CLUE

The simulation tool CLUE (Hlavacs, Kvasnicka, Ueberhuber [30]) has been optimized for clusters of PCs or workstations, although it may be applied also to other parallel hardware. It simulates the performance of programs using the message passing library PVM. In order to use CLUE, the parallel software must be recompiled and linked to the runtime library. CLUE supports both real execution and using program models only, and any mixture of them.

Chapter 5

Summary

The simulation of parallel computer performance has been investigated extensively in the past. Several tools following different paradigms have been created. A comprehensive study of the most popular simulation tools has been carried out by the authors and is presented in this report.

There are four major approaches in order to model and simulate parallel programs and computers. In a model-driven approach, the hardware and software is modelled by special data structures only. Pure discrete event simulation is then applied to the model.

In the trace-driven approach, the used parallel communication library is changed in order to log all computation and communication events. This logfile is then used to drive the simulation engine. As a side effect, the software model is implicitly created and simulated.

A completely different approach is used in instruction-level simulation. Here, each assembly instruction of a parallel program is examined and its execution is simulated on a processor model. This way, the simulation results are highly accurate, whereas the simulation times usually explode in an unacceptable manner.

Finally, in the execution-driven paradigm, the simulated software is changed to include calls to the simulation library. The execution of the parallel programs then directly drive the simulation engine.

Bibliography

[1] Agrawal D., Choy M., Leong H. V, Singh A.: MAYA: *A Simulation Platform for Distributed Shared Memories.* In "Proceedings of the 8th Workshop on Parallel and Distributed Simulation", IEEE Computer Society Press, Los Alamitos 1994, pp. 151–155.

[2] America P., Rutten J.: *A Parallel Object-Oriented Language: Design and Semantic Foundations.* Ph.D. thesis, Free University Amsterdam, 1989.

[3] Aversa R., Mazzeo A., Mazzocca N., Villano U.: *Heterogeneous System Performance Prediction and Analysis Using* PS. IEEE Concurrency 6-3 (1998), pp. 20–29.

[4] Bagrodia R. et al.: *PARSEC: A Parallel Simulation Environment for Complex Systems.* IEEE Computer 31-10 (1998), pp. 77–85.

[5] Bagrodia R., Prakash S.: *MPI-SIM: Using Parallel Simulation to Evaluate MPI Programs.* In Proc. of Winter Simulation Conference, 1998.

[6] Bagrodia R., Deeljman E., Docy S., Phan T.: *Performance Prediction of Large Parallel Applications using Parallel Simulations.* In Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 1999), ACM 1999, pp. 151–162.

[7] Banks J.: *Handbook of Simulation.* Wiley, New York 1998.

[8] Bedichek R.: TALISMAN: *Fast and Accurate Multicomputer Simulation.* In "SIGMETRICS '95", ACM, New York 1995., pp. 14–24.

[9] Blackford L. S. et al.: SCALAPACK *Users' Guide.* SIAM Press, Philadelphia 1997.

[10] Bolch G., Greiner S., de Meer H., Trivedi K.S.: *Queuing Networks and Markov Chains.* John Wiley, New York 1998.

[11] Boothe B.: *Fast Accurate Simulation of Large Shared Memory Multiprocessors.* Technical Report CSD 92/682, Computer Science Division (EECS), University of California at Berkeley 1992.

[12] Brewer E. A., Dellarocas C. N., Colbrook A., Weihl W. E.: PROTEUS: *A High-Performance Parallel-Architecture Simulator.* Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, Cambridge 1991.

[13] Birtwistle G.M., Dahl O.-J., Myhrhaug B., Nygaard K.: *SIMULA Begin*, Auerbach, Lund 1973.

[14] Bryant R.E.: *Simulation of Packet Communication Architecture Computer Systems.* Technical Report MIT-LCS-TR-188, Massachusetts Institute of Technology, Cambridge 1977.

[15] Chandy K.M., Misra J.: *Distributed Simulation: A Case Study in Design and Verification of Distributed Programs.* IEEE Trans. on Soft. Engi. SE 5-5 (1979), pp. 440–452.

[16] Chiola G., Franceschinis G., Gaeta R., Ribaudo M.: *GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets.* Performance Evaluation, special issue on Performance Modelling Tools, August 1995.

[17] Davis H., Goldschmidt S., Hennessy J. L.: TANGO*: A Multiprocessor Simulation and Tracing System.* Technical Report CSL-TR-90-439, Stanford University, 1990.

[18] Delaitre T., Justo G. R., Spies F., Winter S.: *Simulation Modelling of Parallel Systems.* In "Proceedings of the 1st Austrian-Hungarian Workshop on Distributed and Parallel Systems." Technical Report KFKI-1996-09/M,N, Hungarian Academy of Sciences, 1996.

[19] Dickens P. M., Heidelberger P., Nicol, D. M.: *Parallelized Network Simulators for Message-Passing Parallel Programs.* In "Proceedings of the Third International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems", IEEE Computer Society Press, Los Alamitos 1995, pp. 18–20.

[20] Donatelli, S. et al.: *Use of GSPNs for Concurrent Software Validation in* EPOCA. Information and Software Technology 36-7 (1994), pp. 443–448.

[21] Dwarkadas S., Jump J. R., Sinclair J. B.: *Execution-Driven Simulation of Multiprocessors: Address and Timing Analysis.* ACM Transactions on Modeling and Computer Simulation 4-4 (1994), pp. 314–338.

[22] Ferscha, A., Johnson, J.: *Performance Prototyping of Parallel Applications in* N-MAP. In "Proceedings of the IEEE Second Int. Conference on Algorithms and Architectures for Parallel Processing", IEEE Computer Society Press, Los Alamitos 1996, pp. 84–91.

[23] Ferscha A.: *Parallel and Distributed Simulation of Discrete Event Systems.* In Parallel and Distributed Computing Handbook (A.Y. Zomoya, ed.), McGraw-Hill, New York 1996, pp. 1003–1041.

[24] Finger U., O'Donnell C., Siegelin C.: ALLEGRO*: an Efficient Execution-Driven Simulator.* Technical Report No. 95 D 012. Computer Science Department, Ecole Nationale Supérieur des Télécommunications, Paris 1995.

[25] Frazier T. M., Tamir Y.: *Execution-Driven Simulation of Error Recovery Techniques for Multicomputers.* In 30th Annual Simulation Symposium, Atlanta 1997, pp. 4–13.

[26] Gefflaut A., Joubert P.: SPAM: *a Multiprocessor Execution-Driven Simulation Kernel.* In International Journal in Computer Simulation 6-1 (1996), pp. 69–88.

[27] Gursoy A., Kale L. V.: *Simulating Message-Driven Programs.* In "Proceedings of Int. Conference on Parallel Processing", vol. III, 1996, pp. 223–230.

[28] Herrod S.: TANGO LITE: *A Multiprocessor Simulation Environment.* Technical report, Computer Systems Laboratory, Stanford University 1993.

[29] Hlavacs H., Kotsis G., Steinkellner C.: *Traffic Source Modeling.* Technical Report No. TR-99101, Institute for Appl. Comp. Science and Inf. Systems, University of Vienna 1999.

[30] Hlavacs H., Kvasnicka D. F., Ueberhuber C. W.: CLUE—*A tool for Cluster Evaluation.* Distributed and Parallel Systems DAPSYS 2000 (P. Kacsuk, G. Kotsis, Eds.), Kluwer Academic Publishers, Boston, 2000, pp. 61–64.

[31] Hlavacs H., Ueberhuber C. W.: *High-Performance Computers—Hardware, Software, and Performance Simulation.* Frontiers of Simulation, SCS Europe, Ghent, *to appear.*

[32] Hu L., Gorton I.: *Performance Evaluation for Parallel Systems: A Survey.* Technical Report UNSW-CSE-TR-9707, University of NSW, Sydney, Australia, October 1997.

[33] Jain R.: *The Art of Computer Systems Performance Evaluation.* Wiley, New York 1991.

[34] Jefferson D.R.: *Virtual Time.* ACM Transactions on Programming Languages and Systems 7-3 (1985), pp. 404–425.

[35] Jonkers H.: *Introduction to Probabilistic Performance Modelling of Parallel Applications.* Technical Report 1-68340-44(1993)04, Delft University of Technology, Netherlands 1993.

[36] Krommer A. R., Ueberhuber C. W.: *Numerical Integration on Advanced Computer Systems.* Springer-Verlag, Berlin Heidelberg New York 1994.

[37] Labarta J. et al.: DIP: *A Parallel Program Development Environment.* In "Proc. Euro-Par '96, vol. II". Springer-Verlag, Berlin Heidelberg New York 1996, pp. 665–674.

[38] Larsson F., Magnusson P. S., Werner B.: SIMGEN: *Development of Efficient Instruction Set Simulators.* Research Report R97:03, Swedish Institute of Computer Science, Kista 1997.

[39] Larus J, Schnarr E.: *EEL: Machine-Independent Executable Editing.* In SIGPLAN Conference on Programming Languages, Design and Implementation, 1995, pp. 291–300.

[40] Lewis M.J., Bowman C.M.: *DPS: A Distributed Program Simulator and Performance Measurement Tool.* M.S. thesis. The Pennsylvania State University (Penn State), State College, PA, 1994.

[41] Lusk E. et al.: *Portable Programs for Parallel Processors.* Holt, Rinehart, and Winston, New York 1987.

[42] Magnusson P. S. et al.: SIMICS/SUN4M: *A Virtual Workstation.* In "Proceedings of the Usenix Annual Technical Conference", New Orleans 1998, pp. 119–130.

[43] Mazzeo A., Mazzocca N., Russo S., Vittorini V.: *A Method for Predictive Performance Evaluation of Distributed Programs.* Simulation: Practice and Theory 5 (1997), pp. 65–82.

[44] Mazzocca N., Rak1 M., Villano U.: *The Transition from a PVM Program Simulator to a Heterogeneous System Simulator: The* HESSE *Project.* In LNCS 1908, Springer-Verlag, Berlin Heidelberg New York 2000, pp. 266–273.

[45] Menasce D. A., Almeida V A. F., Dowdy L. W.: *Capacity Planning and Performance Modeling: From Mainframes to Client-Server Systems.* Prentice-Hall, Englewood Cliffs, New Jersey 1994.

[46] Mohr B., Brown D., Malony A.: *TAU: A Portable Parallel Program Analysis Environment for pC++.* In "Proceedings of CONPAR 94–VAPP VI", University of Linz, Austria 1994.

[47] Mohr, W., Malony, A., Shanmugam, K.: SPEEDY: *An Integrated Performance Extrapolation Tool for* PC++. In "Proc. Joint Conf. Performance Tools 95 and MMB 95". Springer-Verlag, Berlin Heidelberg New York 1995.

[48] Mukherjee S. S. et al.: WISCONSIN WIND TUNNEL II: *A Fast and Portable Parallel Architecture Simulator.* In "Workshop on Performance Analysis and Its Impact on Design" at the IEEE/ACM International Symposium on Computer Architecture, IEEE Computer Society, Los Alamitos 1997.

[49] Muller H. L.: *Simulating Computer Architectures.* PhD thesis, Dept. of Comp. Sys, Univ. of Amsterdam 1993

[50] Nguyen A.-T., Michael M., Sharma A. Torrellas J.: *The* AUGMINT *Multiprocessor Simulation Toolkit for Intel x86 Architectures.* In "Proceedings of the International Conference on Computer Design", IEEE, Los Alamitos 1996, pp. 486–490.

[51] Park D., Saavedra R.: TROJAN: *A High-Performance Simulator for Shared Memory Archtectures.* In "Proceedings of the 29th Annual Simulation Symposium", IEEE Computer Society, Los Alamitos 1996, pp 44–53.

[52] Prylli L., Tourancheau B.: *Distributed Simulation of Parallel Computers.* In "29th Annual Simulation Symposium", IEEE Computer Society, Los Alamitos 1996, pp. 25–34.

[53] Qin X., Baer J.-L.. *A Parallel Trace-driven Simulator: Implementation and Performance.* In "Proceedings of International Conference on Parallel Processing", IEEE, Los Alamitos 1994.

[54] Reinhardt S. K. et al.: *The* Wisconsin Wind Tunnel*: Virtual Prototyping of Parallel Computers.* In "Proceedings of the 1993 SCM SIGMETRICS Conference", Sigmetrics, Santa Clara 1993.

[55] Reisig, W.: *Petri Nets–An Introduction.* EATCS Monographs on Theoretical Computer Science 4. Springer-Verlag, Berlin Heidelberg New York 1985.

[56] Rexford J., Feng W., Dolter J., Shin K.G.: *PP-MESS-SIM: A Flexible and Extensible Simulator for Evaluating Multicomputer Interconnection Networks.* IEEE Transactions on Parallel and Distributed Systems 8-1 (1997), pp. 25–40.

[57] Rosenblum M., Bugnion E., Devine S., Herrod S.: *Using the* SimOS *Machine Simulator to Study Complex Computer Systems.* ACM TOMACS Special Issue on Computer Simulation, 1997, pp. 78–103.

[58] Srinivas S., Gannon D.: *Executing Object-Oriented Parallel Programs on High Performance Simulators.* Technical Report TR423, Department of Computer Science, Indiana University, Bloomington, Indiana 1995.

[59] Topol B., Sunderam V., Alund A.: *PGPVM Performance Visualization Support for PVM.* Technical Report CSTR-940801, Emory University, Atlanta 1994.

[60] Vaidya A. S., Sivasubramaniam A., Das C. R.. *Performance Benefits of Virtual Channels and Adaptive Routing: An Application-Driven Study.* In "Proceedings of ACM International Conference on Supercomputing (ISC'97)", ACM, New York 1997, pp. 140–147.

[61] Vassiliou V., Taylor S.J.E.: *Predicting the Performance of High Performance FORTRAN Programs using Simulation.* Technical Report ISTR-99-08. Department of Information Systems and Computing, Brunel University, Uxbridge 1998.

[62] Vee V.-Y., Hsu W.-J.: *Parallel Discrete Event Simulation: A Survey.* Technical report, Centre for Advanced Information Systems, Nanyang Technological University, Singapore 1999.

[63] Veenstra J., Fowler R.: MINT: *A Front End for Efficient Simulation of Shared-Memory Multiprocessors.* In "Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'94)", Durham 1994, pp. 201–207.

[64] Wabnig H., Haring G.: PAPS—*The Parallel Program Performance Prediction Toolset.* In "Computer Performance Evaluation: Proceedings of the 7th International Conference on Modelling Techniques and Tools." Springer-Verlag Berlin Heidelberg New York 1994, pp. 284–304.

[65] Witchel E., Rosenblum M.: EMBRA: *Fast and Flexible Machine Simulation* In "Proceedings of the 1996 SIGMETRICS Conference on Measurement and Modeling of Computer Systems", Philadelphia 1996.

[66] Yan Y., Zhang X., Song Y.: *An Effective and Practical Performance Prediction Model for Parallel Computing on Non-Dedicated Heterogeneous NOW.* Journal of Parallel and Distributed Computing 38 (1996), pp. 63–80.