Simulating Parallel Program Performance with CLUE

Dieter F. Kvasnicka
Institute for Physical and Theoretical Chemistry,
Vienna University of Technology
dieter.kvasnicka@tuwien.ac.at

Helmut Hlavacs
Institute for Computer Science and Business Informatics,
University of Vienna
hlavacs@ani.univie.ac.at

Christoph W. Ueberhuber
Institute for Applied and Numerical Mathematics,
Vienna University of Technology
christoph@aurora.anum.tuwien.ac.at

Keywords: parallel, message passing, high performance, PVM, PC cluster

ABSTRACT

In this paper the simulation and assessment tool CLUE is described. This tool is able to simulate the performance of parallel programs using the message passing library PVM for communication, run on arbitrary parallel machines, including PC clusters. CLUE redirects calls to PVM to its own functions, providing an additional layer between an application and PVM. The simulation is driven by the application execution itself. The applicability of CLUE is demonstrated in three case studies, where (i) different load predictors are compared, and (ii) the performance of a master-slave parallelization and (iii) subroutines of the widely used ScaLapack linear algebra package are simulated.

INTRODUCTION

The increasing number of available parallel computers or clusters of workstations, interconnected with high speed networks, has created a need for efficient parallel software. Developing such is difficult due to the additional communication overhead often necessary. Factors influencing the efficiency are, for instance, the problem size, the percentage of sequential code, the speed of the communication system, the communication/computation ratio and the number and type of processors used. Parallel programs running efficiently on one parallel computer might be very inefficient on others.

There are several ways of comparing algorithms for parallel computers, all suffering from particular drawbacks. Analytical models are very difficult to create, might be based on simplifying assumptions and often cannot catch the possibly complicated structure of the simulated environment or the parallel programs.

Comparison by executing the programs is restricted to available parallel computers only. Interesting properties like the program behavior on various platforms, interconnected with different networks, cannot be obtained. Furthermore, the execution of parallel programs as part of an extensive study to gain insight into program characteristics might use up large amounts of CPU time, thus consuming computing time possibly needed otherwise.

Simulation tries to bridge the gap between analytical models and extensive tests on existing computer platforms. When simulating the execution of parallel programs, in principle the performance of any program or program model running on arbitrary parallel computers may be analyzed.

In this paper, the simulation and assessment tool CLUE (cluster evaluator) is introduced, which is able to simulate the performance of message passing programs executed on parallel computers. CLUE has been specifically designed to simulate program runs on clusters of workstations or PCs, yet CLUE may also simulate other parallel computers like parallel supercomputers.

One important application of CLUE is the simulation of the performance of parallel programs run on clusters of SMPs *a priori*, i.e., before running them on the real hardware. This way, a potential cluster customer may simulate the performance impact of certain configuration decisions before actually buying the cluster. The idea behind this application is therefore to *adapt the hardware to already existing software*. However, CLUE is not intended to guide hardware development in the narrow sense.

Another application of CLUE is the design and *tuning of parallel programs* run on parallel computers. In this application, the user of CLUE may test the performance of his program on hypothetical platforms, evaluating various strategies of parallelization. In this case, the *software is adapted to hardware*. This adaptation may take place during software development, before actually purchasing the target computing platform.

CLUE has been developed to simulate the execution of real programs on arbitrary parallel computer configurations. As MISS-PVM [Kvasnicka and Ueberhuber 1997], the main part of CLUE, provides a virtual layer between the application program and PVM, the application program must use the message passing library PVM. It is, however, possible to simulate the performance of other message passing libraries like MPI by using the PVM version. The second part of CLUE, the Workstation User Simulator (WUS) has been created to additionally simulate the effect of workstation users starting competing processes, making it also possible to run statistical models instead of real code, thus speeding up simulation runs by several orders of magnitude.

RELATED WORK

In the past, several attempts have been made to simulate the performance of parallel programs. N-map [Ferscha:1996], for example, allows predicting the performance of parallel programs by specifying code fragments only. The PVM Simulator PS [Aversa et al.1998] follows an approach similar to ours, accepting full PVM programs or program prototypes. PS, however, does not use execution driven simulation but produces trace files describing the communication pattern of the application under observation. These trace files then drive the simulation kernel to derive simulation results. Tau [Sheehan et al. 1999] is a performance extrapolation tool that also collects run-time trace information of parallel programs written in pC++, which is later used to drive the trace-driven simulation engine. The same trace-driven approach is used in Dip [Labarta et al. 1996].

In general, trace-driven approaches assume that the communication patterns are fixed and do not depend on the run-time situation. This assumption is valid, for example, for most routines from the well-known linear algebra library SCALAPACK [Blackford et al. 1997]. In cases where the communication depends on the run-time situation, however, for example when simulating the effect of load balancing mechanisms, this approach cannot be used, in contrast to execution driven simulation.

Execution driven simulators include, for example, the simulation platform SimOS [Rosenblum et al. 1997]. The simulation kernels of this platform allow the simulation of various processor models and hardware features with varying degree of complexity. Users of SimOS simply start the binary executable of their code, which then drives the simulator. This approach relies on the availability of a processor model for the processor family the executable was compiled for. Other parallel programming tools like Aims [Yan et al. 1995] observe real program executions and provide sophisticated tools for post-mortem trace file analysis. In this approach, only the performance of parallel programs run on available platforms can be evaluated.

THE CLUSTER EVALUATOR CLUE

The structure of the simulation and assessment tool CLUE can be seen in Figure 1. CLUE is driven by an application program containing the original code or an application model holding only the code skeleton, like calls for message passing or calls to advance the simulation time to simulate the execution of CPU or I/O extensive code fragments. CLUE is thus *execution driven*. The application program or model calls functions provided by either WUS (Workstation User Simulator) and MISS-PVM (Machine Independent Simulation System for PVM 3). If WUS is not used, then the original source code does not have to be changed (with a minor exception), but only has to be recompiled and linked to the MISS-PVM library. If WUS is used, the original source code must be changed to include calls to WUS. Once started, CLUE will be driven by calls of the application program and will advance its virtual time according to the CPU time consumed by the application program and the time used for inter-process communication.

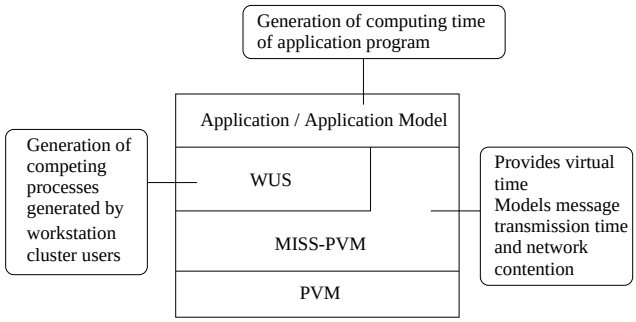

Figure 1. Structure of CLUE.

CLUE has already been applied to different topics:

- Given a fixed program, find the optimum hardware configuration that yields the highest performance for the program.
- Given a fixed program and a certain budget limit, find the optimum hardware that can be afforded.
- Given a set of hardware platforms, evaluate the performance of a certain parallelization strategy.
- Evaluate different dynamic load balancing strategies.

MISS-PVM

MISS-PVM [Kvasnicka and Ueberhuber 1997] is implemented as a layer between the application program and the message passing library Parallel Virtual Machine (PVM). The virtual layer for PVM redirects all calls to PVM to their internal counterparts. Once being called, MISS-PVM measures the CPU time consumed by the application program since its last call to PVM.

This time is then added to the internally maintained *virtual time*. After having performed this task, MISS-PVM will eventually call PVM functions to perform a similar (but not identical) work. For example, sending a message from one process to another will result in several *virtual messages* sent between the instances of the virtual layer.

As a result, the simulator user may observe the simulated virtual time as well as the output trace files, containing information about all sent messages. These trace files must be pre-processed and may be used for post mortem visualization afterwards. This scheme has two major advantages over normal trace file writing: the virtual layer for PVM (i) uses its own *simulated system time* (i.e., the virtual time) and (ii) makes a *virtual machine* available to the user.

Machine parameters are read from a configuration file at program start. The configuration parameters may also be changed dynamically during the program execution. Using MISS-PVM it is possible to compare the performance of programs run on computers with different communication latency and computing speed. WUS enables to additionally create workstation background load of arbitrary complexity. Time-measurements of different load balancing strategies can be made quickly and enable the determination of the optimum strategy for certain architectures.

Using MISS-PVM

PVM is a software system linking a network of heterogeneous computers in such a way that the user may assume the existence of one single parallel computer, the *virtual parallel machine*. It provides message passing and process control routines for tasks running on any of the computers being part of the virtual machine. User processes are connected by TCP to a PVM daemon running on their machine. When sending a message to another process, the sender will pass the message to its PVM daemon, which will transmit the data to the PVM daemon running on the receiver's computer. This daemon will then pass on the data to the receiving process.

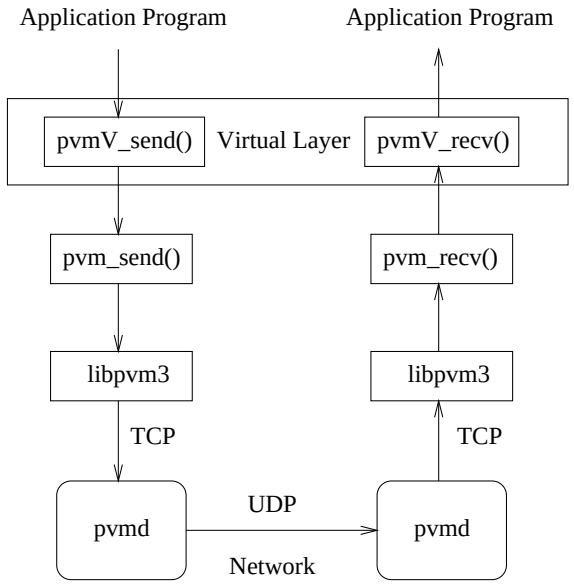

Figure 2. Virtual layer for PVM.

User programs call PVM subroutines in order to send messages or to create and terminate processes on any member of the virtual machine. PVM provides a uniform interface to user programs by hiding different implementations and features of the various flavors of Unix and Windows. In this way a user program may be run on a variety of different computer systems without modification.

In Figure 2 a new level between the user program and PVM is added, the *virtual layer for PVM*. This layer provides the same interface to the user program as PVM does, itself containing no machine-dependent code. Thus, the virtual layer may be used on many different machines.

When using the virtual layer in addition to PVM, the simulator user is provided a *virtual time, virtual machines* with arbitrary characteristics and *output generation* for graphical post mortem visualization. The user programs as well as the PVM level remain unchanged. The only difference is that an include file redirects PVM calls to their virtual equivalents.

As a visualization program, ParaGraph may be used. This graphical tool provides several animated windows, which are, to a great extent, self-explanatory [Tomas and Ueberhuber 1994, Heath 1993].

Virtual Time

The virtual layer for PVM uses an internal time that is based on three components:

- **Computation time** is the CPU time consumed by executing the user programs. This time is measured by calling operating system calls.
- **Communication time** is calculated using the configuration parameters of the virtual machine.
- **Waiting time** is simulated as the time a process waits for the arrival of messages.

These three components are added to result in the *virtual time* of each user process.

Virtual Machines

Virtual machines are defined in a file that is read at the start of the simulation. The first line of this file contains the parameters of the computer used for the master program and as default for all programs started without an explicit machine name or host type given. In the other lines, comments (beginning with the symbol '#'), or additional machine or host type specifications can be put. For each line possible parameters are:

- **Name** of the machine or host type. The machine can either exist in reality or can be a *virtual machine*.
- **Performance factor.** This is a floating-point multiplier *p* for calculating the computation time.
- **Initialization Delay.** This is the time needed for pvm_spawn(), as seen by the spawned program.
- **Spawn Delay.** This is the time spent in pvm_spawn().
- **Send Delay.** This is the time used for sending a message using pvm_send() or pvm_mcast(). This time contains packing the message, resolving the address of the host and starting the transmission (as far as the sending process is involved). The actual send delay is interpolated linearly between given points.

- **Receive Delay.** This is the time used in calling the receive routines pvm_rcv(), pvm_nrecv() and pvm_probe().
- **Transmission Delay.** This is the time used to transfer a message minus the send delay. It is typically a piecewise linear model.
- **Packing Delay.** This is the time used to pack the message into the PVM send buffer.

Send and transmission delay may be specified for any pair of hosts, they may also define sending and transmitting messages from one host to itself, in case multiprocessor machines are to be modeled. Figure 3 shows the assumed model for the send and transmission delay.

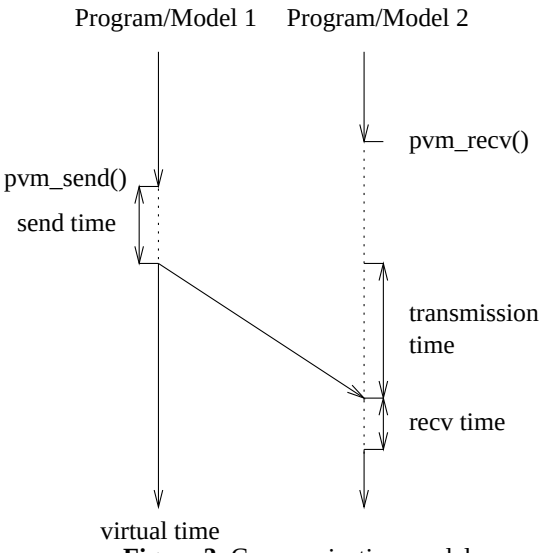

Figure 3. Communication model.

If the actual performance model turns out to be of insufficient accuracy, it can easily be modified in the configuration file. A recompilation of the simulated program is not required.

Development Process

In order to simulate the performance of parallel programs run on a set of hardware platforms, the following steps must be carried out:

1. The parallel program using PVM must be developed.
2. In all source files, the PVM include file must be changed to the MISS-PVM include file (not in Fortran).
3. The Makefile must be changed to link the MISS-PVM library to the executable.
4. For each hardware configuration to simulate, a configuration file has to be created. The parameters for the computational speed and the network properties may be either derived by measuring existing hardware, extrapolating from known parameters, or by using vendor specified information. The parameters used in the case studies described later were derived by taking measurements with standard benchmarking and specially written programs.
5. Then the source files must be recompiled and linked to the MISS-PVM library.
6. The simulation is then executed by starting the PVM program as in a normal program run.

Distributed Simulation Protocol

In order to execute all events according to their virtual time, MISS-PVM uses a conservative protocol for distributed discrete event simulation based on an extra process called *MISSdaemon*. The daemon keeps an internal list of all running PVM processes. Upon receiving messages, the daemon updates its process list by calling the virtual version of pvm_tasks(), which returns a list of all processes with the exception of the MISSdaemon itself. Each entry in this list can have one of the following states:

- **Unknown.** The process is believed to do work.
- **Waiting for line.** The process has called pvm_send().
- **Waiting for non-blocking receive.** The process has called pvm_probe() or pvm_nrecv().
- **Blocked receive.** The process has called the MISS-PVM version of pvm_recv() and is waiting for messages.
- **Deleted.** In this case, the process is removed from the process list and is added to a deletion list.

Once the states of all processes are known, the next event is chosen from the event list and is executed. This may either be a sender waiting for the allowance to proceed, or the delivery of a message to a receiver. In the first case, the sender is simply notified by a virtual message, in the latter case, a virtual message is sent to the receiver, containing information about the message size and the sender PID. Upon reception of this message, the sender of a message may proceed whereas the message receiver unlocks the corresponding data waiting in an internal buffer and pretends to having received the data at the respective virtual time. The protocol needs a total of four virtual layer messages with fixed size and one user data message of arbitrary size. Using the MISSdaemon, the order of messages at the receiver's end is preserved.

THE WORKSTATION USER SIMULATOR

The Workstation User Simulator (WUS) [Hlavacs and Ueberhuber 1998] is the second part of CLUE. WUS simulates the generation of competing processes, running in parallel on interactively used workstation clusters, and taking away CPU cycles there. Processes can be generated by using fixed arrival and departure rates, variable arrival and departure rates provided by trace files [Calzarossa and Serazzi 1985], trace files of real processes [Zhou 1986] and user behavior graphs [Calzarossa and Serazzi 1986].

By constructing stochastic models of real parallel applications or running real applications, different load balancing schemes can be simulated and compared with each other. It is important to note that the competing processes are not started in reality, but are only represented by list entries in the *virtual CPU* (VCPU) queue of WUS. The WUS VCPU is, however, tightly linked to the MISS-PVM virtual time. Whenever a WUS process consumes VCPU time, WUS increases the MISS-PVM virtual time accordingly. On the other hand, if an application consumes real CPU time between two adjacent calls to MISS-PVM, MISS-PVM activates WUS where the consumed CPU time now must compete for the VCPU with other WUS processes. Figure 4 shows such a sequence of calls.

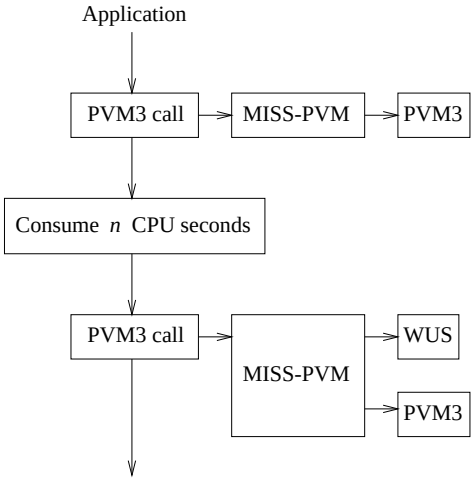

Figure 4. Real application consuming CPU time**.**

First, the real application calls a PVM function, which is replaced by the according MISS-PVM call. MISS-PVM manages the virtual time and communicates with other processes by means of virtual layer messages using PVM. Also, MISS-PVM stores the amount of CPU time this process has consumed so far. The real application consumes *n real* CPU seconds and, in order to do some communication, finally invokes a PVM call, again being replaced by the according MISS-PVM call. The *n* real seconds are modified by MISS-PVM according to the state of WUS and the virtual time is increased.

WUS Scheduling

The application model calls WUS functions to state that it wishes to be granted *n* VCPU seconds. WUS then schedules its VCPU to all running WUS processes by *using priority scheduling* as implemented in the Linux kernel, driven by the standard UNIX nice levels.

Like in the *processor sharing* queuing discipline [Allen 1990], it is assumed that the time-slices scheduled to each process are infinitely small (in contrast, e. g., the duration of each time-slice on an x86-compatible computer running Linux is 10 ms).

Using WUS

WUS mimics a UNIX computer (Figure 5). User models produce workload using trace files of real user sessions, Poisson arrival processes with fixed and variable arrival and departure rates, and user behavior graphs,.

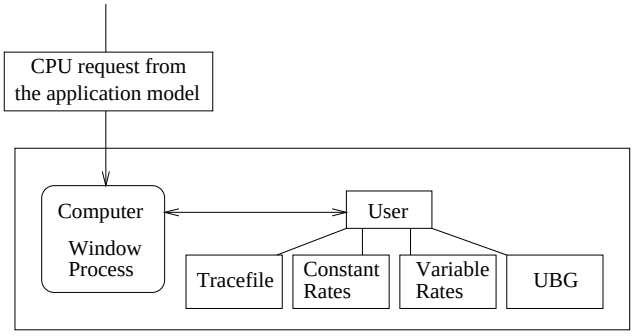

Figure 5. WUS structure.

Designers of parallel programs wishing to use WUS to test load balancing strategies first have to sample statistical data of the CPU requests of their real parallel applications. Using this data, a statistical application model has to be created. An application model program frame looks like the following example.

```
do communication or initialization
comp = new Computer( Workload model );
while ( Loop ) {
        runtime = GetRandomRuntime();
        comp->RunProcess( runtime );
        loadavg = comp->LoadAverage(n);
        do communication or load balancing
}
collect results
```

The Unix Load Average

One important application of CLUE is the development and assessment of dynamic load balancing strategies. Such strategies observe the run-time behavior of parallel programs and identify overloaded and underloaded processors. A processor overload may occur, (i) if one processor has been assigned more work than others (this can be the case if the amount of work necessary to compute the result is not known in advance), or (ii) if workstation users interactively start competing processes on one or more workstations.

The load balancing strategy thus must react to load changes and may decide to transmit work from one processor to another. For measuring case (ii), usually the Unix load average is used, i.e., the exponentially smoothed length of the processor queue, holding all currently running processes. The Unix load average $\hat{X}_t$ is defined to be

$$\hat{X}_{t+1} = \beta \hat{X}_t + (1-\beta) X_t \quad (1)$$

where X_t is the number of running processes at ti me *t*. The load average depends on $\beta \in [0,1]$, the exponential smoothing constant. It defines, how much of the past should be included into the current load estimate. If writing $\beta = N/(N+1)$, then $\hat{X}_{t+1}$ may also be interpreted as an estimate for the arithmetic mean of the last N observations $X_{t-i}, i = 0,1,\mathrm{K}\ ,N-1$ [Schlittgen and Streitberg 1995]. By setting $\beta = \frac{60}{61} \approx 0.984$ and calculating X_t every second, X_t thus may be interpreted as the arithmetic mean number of processes run in the last 60 seconds. Unix traditionally calculates such estimates for the last 60 seconds, the last 5 minutes and the last 15 minutes. WUS allows the comput a-tion of such load averages for any *N*.

CASE STUDY: PARALLEL INTEGRATION

In order to demonstrate the applicability of CLUE and to validate the simulation accuracy, several case studies have been conducted.

In the first case study, a global bag of tasks is defined to contain 10,000 defin ite integrals

$$\mathrm{I}[f,a,b] = \int_a^b f(x)\,dx \quad (2)$$

for given integrands *f*(*x*) and given interval boundaries a and b. This workload is then to be computed in parallel on workstations being interconnected by Fast Ethernet. The integrals are to be calculated using a globally adaptive automatic integration algorithm [Piessens et al. 1983]. This algorithm computes an approximation

$$\mathrm{Q}[f,a,b] \approx \mathrm{I}[f,a,b] \quad (3)$$

and an error estimate

$$\mathrm{E}[f,a,b] \approx \mathrm{e}[f,a,b] = \left|\mathrm{Q}[f,a,b] - \mathrm{I}[f,a,b]\right| \quad (4)$$

such that

$$\mathrm{E}[f,a,b] \le \tau \quad (5)$$

for a given error tolerance τ. The estimates $\mathrm{Q}[f,a,b]$ and $\mathrm{E}[f,a,b]$ are calculated by evaluating *f(x)* at *N* points and applying a so-called *integration formula* [Krommer and Ueberhuber 1994], being a weighted sum of the integrand values. Then, if (5) does not hold, the original interval $[a,b]$ is subdivided into two intervals $[a,(a+b)/2]$ and $[(a+b)/2,b]$, and estimates (3) and (4) are again calc u-lated for both subintervals. If the sum of the two err or estimates still does not fulfill (5), the interval with the largest error estimate is chosen and further subdivided. This pr o-cedure, resulting in a possibly large number of subintervals and therefore integrand evaluations thus depends on the input data *f*, *a* and *b* in an unpredictable way and the needed CPU time for obtaining (5) is not known a priori. This algorithm is difficult to parallelize, as it is intrinsically sequential. Also, if a large number of independent calcul a-tions for (2) are to be perform ed in parallel, a distribution of the tasks *a priori* is difficult, as the CPU requirements of each task is unknown and thus some processors might get overloaded while others might soon be idle because their tasks need only little CPU time.

For the integran d *f*(*x*), three basic integrand classes were chosen:

- Oscillating integrands (7 families).
- Integrands with singularities, peaks or discontinuities (8 families).
- Mixture families (6 families).

Each integrand family depends on a parameter $\alpha \in [0,1]$ defining the severity of the integration problem (2). The higher α is, the more integrand evaluations and thus CPU time is needed to obtain (5). An example for an oscillatory family is given by

$$\int_0^{2\pi} xe^{\alpha x} \sin(600\alpha + x) + 1\,dx\,. \quad (6)$$

Figure 6 shows the number of integrand evaluations for family (6) needed to fulfill the error requirement (5). The results are given for different *N*-point integration formulas.

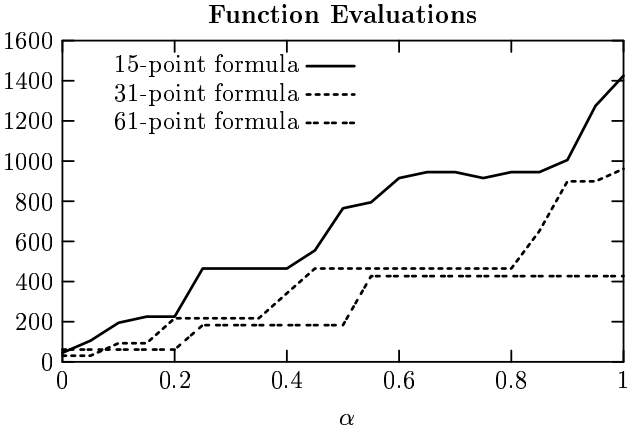

Figure 6. Function evaluations needed for oscillating integrands.

A complete definition of the integrand families as well as a mathematical explanation of the curve shapes can be found in [Hlavacs 2000].

Each of the 10,000 tasks is defined by choosing one in-tegrand family and one particula r $\alpha \in [0,1]$ at random, de-scribing the computation of exactly one definite integral. A central master manages the bag of tasks.

The program has first been executed on a network of workstations (NOW) consisting of five Sun workstations

with Sparc and UltraSparc processors and running the Sun Solaris operating system. These workstations were connected by a switched Fast Ethernet network yielding 100 Mbit/s bandwidth. Additionally, the program has been simulated with CLUE.

Figure 7 shows the measured run-times and the simulation results. It can be seen that in this scenario, the accuracy of CLUE is very high. The experiment also shows that with more than four workstations, no more speed-up is observed.

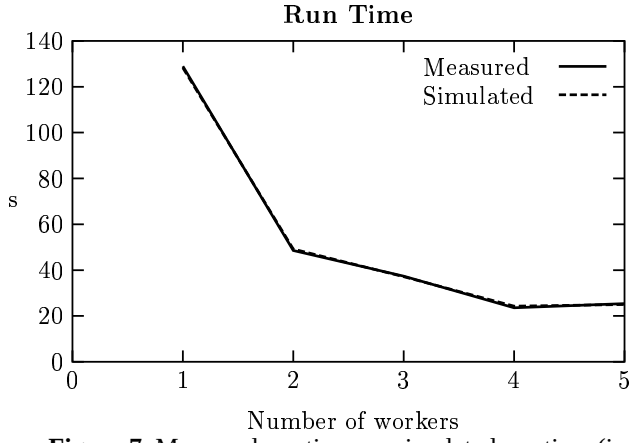

Figure 7. Measured run time vs. simulated run time (in seconds) on the Sun NOW. Task message size is 10 KB.

The same experiment has been repeated on the Beowulf cluster of SMPs maintained by the Institute for Physical and Theoretical Chemistry of the Vienna University of Technology, described in a later case study. This cluster consists of five PCs containing two processors each. Figure 8 shows the simulation results. Again the simulation result yields high accuracy.

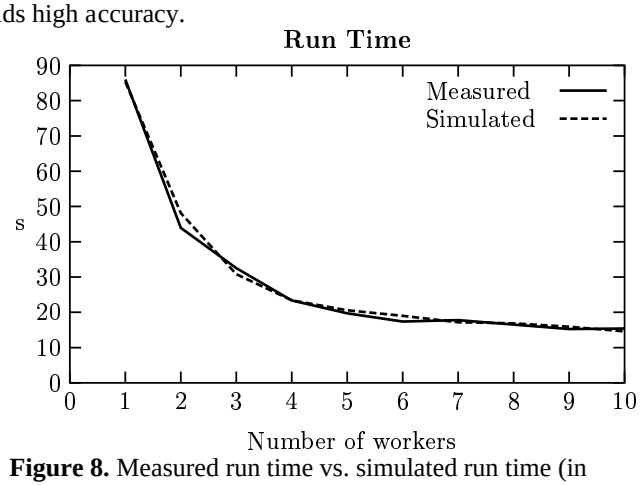

Figure 8. Measured run time vs. simulated run time (in seconds). Task message size is 500 KB.

CASE STUDY: LOAD INDEX EVALUATION

In this case study, the WUS load average simulation is used to find an optimum β for predicting the future workstation workload. One obvious question is, which β is best to predict the future workload of a workstation, and thus the time it takes to compute a task under a given workload.

Sampled Workload

In order to obtain realistic background workload, tracing programs were started on one particular network of workstations being maintained at the Vienna University of Technology. The observed workstations contained DEC Alpha processors under the OSF/1 operating system. The workload was sampled during the period from March 15th, 1998 to May 13th, 1998. For all visible Unix processes, the sampled workload parameters were:

- Time
- Process ID (PID)
- Parent PID
- CPU time consumed so far
- CPU time consumed by all children
- Executable name

Figure 9 shows a typical workstation workload as has been observed on a Monday. It can be seen, how the time of day influences the arrival of processes, thus reflecting the workload that is generated by interactive users. The workload trace files indicate large fluctuations of workload during the day. Especially the faster machines are more likely to get very high workloads.

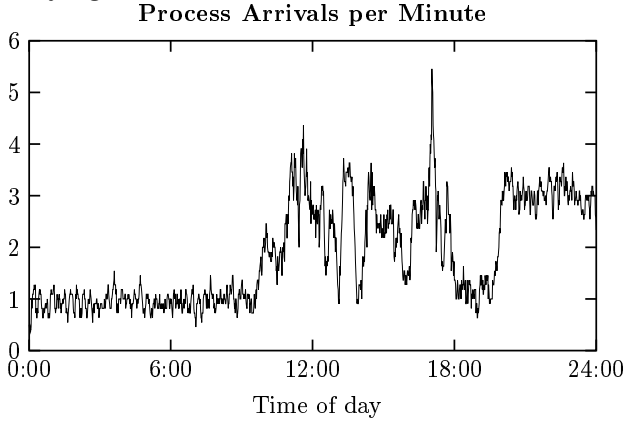

Figure 9. Arrival of processes per minute on one particular workstation day.

Simulation Scenario

For evaluating different load averages, the following simulation scenario was chosen: The workload of one workstation day was provided to WUS, which would use this data to create virtual processes. Beginning at time t, a process consuming $s \in [1, 60]$ CPU seconds was then con-

tinuously created. At start time, the load average (1) was used to predict the actual run-time of this process under the observed load situation. After the process consumed the CPU time, the prediction error, i.e., the difference between the predicted and the actual run-time was computed. Then, another process consuming s CPU seconds was immediately created.

Simulation Results

Figure 10 shows the simulation results. As a measure for the prediction quality, the figures show the mean error of all predictions for one particular tuple (β, s). It can be seen that when averaging over the whole day and for processes consuming only a few CPU seconds, the best prediction is given by $\beta = 0$, which according to (1) denotes the actual number of running processes. As processes consume more CPU seconds, higher values of β produce better results.

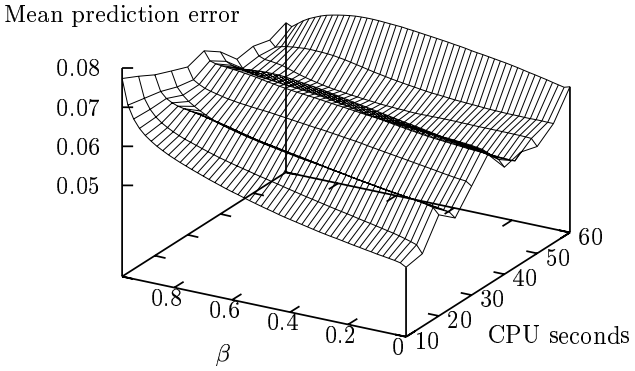

Figure 10. Simulation results using the workload of one particular workstation from 00:00 to 24:00.

CASE STUDY: SCALAPACK ON PC CLUSTERS

In this case study it is demonstrated how to use CLUE for simulating the performance of standard software run on PC clusters. A PC cluster typically consists of N of-the-shelf PCs connected with each other over standard Fast Ethernet or some gigabit class network, each PC containing one, two or four Intel compatible processors working in symmetric shared memory (SMP) mode. PC clusters running the Linux operating system are often called *Beowulf* clusters and have become popular in the last few years due to the fact that they deliver high computing power at a reasonable price. Due to the availability of a large number of different PC hardware components, it is difficult to decide which cluster configuration yields the best performance for a given application. Simulating different cluster configurations before deciding to buy one particular may aid this decision process. In the carried out experiments, two specific PC cluster configurations have been investigated.

The Vienna Cluster.

The first PC cluster of the research project AURORA (http://www.vcpc.univie.ac.at/aurora/) was built for cooperation between the Institutes for Applied and Numerical Mathematics and Physical and Theoretical Chemistry, both part of the Vienna University of Technology. It consists of one master and five dual Pentium II slaves using Fast Ethernet communication. The master is used as file and net server and does all the compilation work.

Figure 11 shows the send and transmission times observed on the Vienna cluster. Both sender and receiver run on the same node, thus the message is not sent over the network. Also, the piecewise linear model used for the simulation is shown as well. These measurements were conducted by running specially written timing software, using both PVM and ordinary UDP packets for time synchronization.

Additionally, for the case of sending messages from one sender to several receivers at the same time, contention has been observed that increases both the send and transmission time.

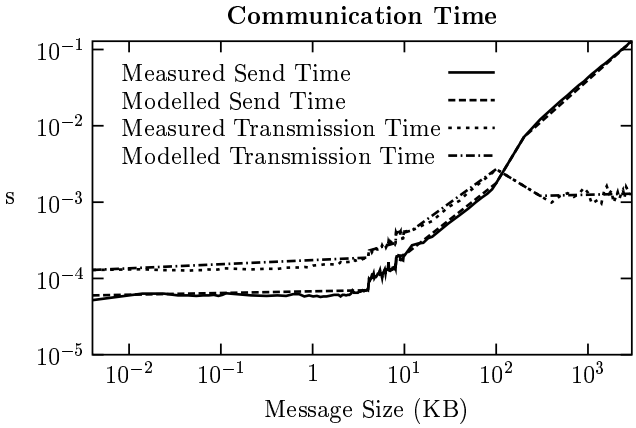

Figure 11. Send and transmission time for the Vienna cluster. Sender and receiver are on the same node.

The Aachen Cluster

The PC cluster Siemens hpcLine consists of 16 dual processor boards using 400 MHz Pentium II. The nodes communicate either via switched Fast Ethernet or SCI (Scalable Coherent Interface). The computational factor of nodes of the Aachen cluster has been measured to be 0.91 relative to the Vienna cluster, where simulation runs have been carried out. Additionally, the SCI network was only available for the MPI version of BLACS. Thus, the communication parameters and the performance of the real runs were collected for the MPI version of BLACS, whereas the simulation runs were still carried out on the Vienna cluster using the PVM version of the BLACS. Communication models for the Aachen cluster SCI network can be found in [Hlavacs 2000].

Simulated Software

The standard parallel software chosen for simulation consists of subroutines of SCALAPACK [Blackford et al. 1997], the parallel version of LAPACK [Anderson et al. 1999], the well-known library for linear algebra. Both LAPACK and SCALAPACK are based on calls to the *basic linear algebra subprograms* (BLAS), their parallel version being called PBLAS. Both PBLAS and SCALAPACK use the *basic linear algebra communication subroutines* (BLACS) for communication, the BLACS itself being based on PVM or MPI.

Three SCALAPACK routines were used to demonstrate the usefulness and reliability of CLUE:

- **Matrix-Matrix Multiplication.** The routine `PBLAS/pdgemm` is used to multiply two matrices.
- **Cholesky Factorization.** The routine `SCALAPACK/pdpotrf` is used to compute the Cholesky factorization of a symmetric, positive definite matrix.
- **LU Factorization.** The routine `SCALAPACK/pdgetrf` is used to compute the LU-factorization of a general matrix.

In this case study matrix sizes have been set to 2000×2000.

Simulation Results

For the real runs, the PVM version of SCALAPACK and PBLAS were used (on the Vienna cluster). Each simulation run was carried out on one workstation only. All executables print out their result in terms of the needed wall clock time.

The simulation runs should answer the following questions:

1. Do the real observations and the simulated runs have the same qualitative properties?
2. Do the real observations and the simulated runs have the same quantitative properties?
3. Can the simulation results be used to evaluate the performance of workstations clusters a priori?

In the following figures, the observed and simulated wall clock times are plotted against the processor grid used. Such a grid or 2-dimensional mesh is always assumed to define the topology of the parallel computer, even if in reality this is a workstation cluster connected over a bus, star or ring topology. Each processor is assigned to a certain place in the virtual mesh topology. Basically, an $N \times M$ grid means that $N \times M$ processors were used for the computation. The relation of N to M defines the communication pattern used, yielding different speed-ups as the below results show. As can be seen, simulation results obtained for a PC cluster with slow communication (using Fast Ethernet) are very accurate. The simulation highly satisfactory has captured both qualitative and quantitative performance behavior of this Beowulf cluster. Inaccuracies only occur for some runs of matrix-matrix multiplication, where the simulation does not reflect contention.

In contrast to simulating the performance of the PVM versions of SCALAPACK and PBLAS by using the same PVM code, simulating their MPI versions by using the PVM versions on a different type of node is far more complicated. Still, the qualitative behavior of the parallel programs is accurately simulated, while the quantitative results are sometimes a little misleading.

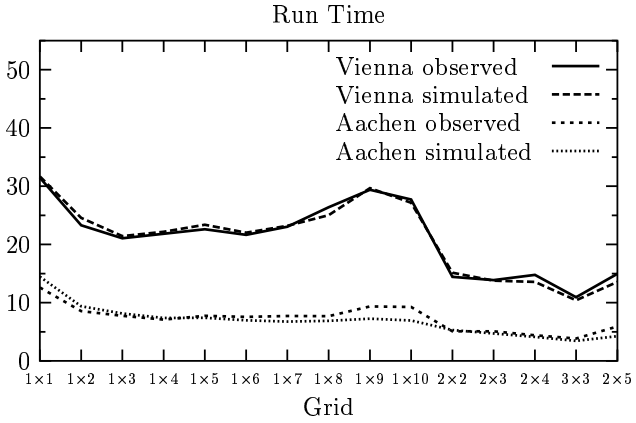

Figure 12. Cholesky factorization run time.

It may thus be concluded, that performance comparisons between different workstation clusters are possible, though experiments must be carefully designed and interpreted. The qualitative behavior of parallel programs running on a workstation cluster though can be simulated accurately, independent of the use of PVM or MPI.

It is thus possible to analyze the behavior of parallel programs and predict their performance, depending on cluster parameters. Simulation results can be used to investigate the influence of different parameters of the simulated workstation or PC cluster, in order to plan new hardware configurations or make an educated choice between several alternatives.

CONCLUSION

In this work, the simulation and assessment tool CLUE has been described. It consists of MISS-PVM, the actual simulation layer, and WUS, the Workstation User Simulator. MISS-PVM allows the simulation of parallel programs using the PVM library for message passing. The simulation may be carried out on one or several computers, whereas the properties of the virtually assumed parallel computer are defined in a configuration file. The conservative distributed discrete event simulation protocol guarantees correct event order.

By linking MISS-PVM to WUS, real applications or statistical models can be used to simulate load balancing on

heterogeneous, interactively used workstation clusters. In order to support this task, WUS allows *priority scheduling* and produces load estimates similar to the standard UNIX load metrics, thus simulating the effect of concurrently running processes.

The applicability and accuracy has been demonstrated in three case studies. Simulation runs show good accuracy when compared to real runs.

REFERENCES

[Allen 1990] Allen A. O.: *Probability, Statistics and Queuing Theory.* Academic Press, Orlando 1990.

[Anderson et al. 1999] Anderson E. et. al: *Lapack Users' Guide,* 3rd ed. SIAM Press, Philadelphia 1999.

[Aversa et al. 1998] Aversa, R., Mazzeo, A., Mazzocca, N., Villano, U.: "Heterogeneous system performance prediction and analysis using PS", IEEE Concurrency 6-3 (1998), pp. 20—29.

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

[Calzarossa and Serazzi 1985] Calzarossa M., Serazzi G.: "ACharacterization of the Variation in Time of Workload Arrival Patterns", IEEE Transactions on Computers C-34-2 (1985), pp. 156—162.

[Calzarossa and Serazzi 1986] Calzarossa M., Serazzi G.: "System Performance with UserBehavior Graphs", Performance Evaluation 11 (1990), pp. 155—164.

[Ferscha 1996] 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 CS Press 1996, pp. 84—91.

[Heath 1993] Heath M. T.: "Recent Developments and Case Studies in Performance Visualization using ParaGraph", In *Performance Measurement and Visualization of Parallel Systems* (G. Haring, G. Kotsis, eds.), Elsevier Science Publishers, Amsterdam 1993, pp. 175—200.

[Hlavacs and Ueberhuber 1998] Hlavacs H., Ueberhuber C. W.: "Simulating Load Balancing on Workstations with Irregularly Fluctuating Capacity", AURORA Technical Report TR1998-11, Technical University of Vienna 1998.

[Hlavacs and Ueberhuber 1999] Hlavacs H, Ueberhuber C. W.: "Simulating Load Balancing on Heterogeneous Workstation Clusters", in *ACPC'99,* Springer Verlag, Berlin 1999.

[Hlavacs 2000] Hlavacs H.: "Cluster Computing - High Performance Solutions of Problems with Unknown Complexity", Ph. D. dissertation, Technical University of Vienna, Austria, November 2000.

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

[Kvasnicka and Ueberhuber 1997] Kvasnicka D. F., Ueberhuber C. W.: "Developing Architecture Adaptive Algorithms Using Simulation with MISS-PVM for Performance Prediction", In *11th ACM International Conference on Supercomputing,* 1997, pp. 333—339.

[Kvasnicka 2000] Kvasnicka D.: "High Performance Computing in Materials Science", Ph. D. dissertation, Technical University of Vienna, Austria, September 2000.

[Labarta et al. 1996] 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.

[Piessens et al. 1983] Piessens A., De Doncker E., Kapenga J., Ueberhuber C.W., Kahaner D.: *Quadpack, A Subroutine Package for Automatic Integration.* Springer-Verlag, Berlin Heidelberg New York 1983.

[Sheehan et al. 1999] Sheehan T., Malony A., Shende S.: "Runtime Monitoring Framework for the TAU Profiling System", In *Proceedings of the Third International Symposium on Computing in Object -Oriented Parallel Environments (ISCOPE'99),* San Francisco, CA, December 1999.

[Rosenblum et al. 1997] 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.

[Schlittgen and Streitberg 1995] Schlittgen R., Streitberg B.: *Zeitreihenanalyse*. R. Oldenbourg Verlag, Muenchen Wien 1995.

[Tomas and Ueberhuber 1994] Tomas G., Ueberhuber C.W.: *Visualization of Scientific Parallel Programs*. Springer-Verlag, Berlin Heidelberg New York 1994.

[Yan et al. 1995] Yan, J., Sarukkai, S., Mehra, P.: "Performance measurement, visualization and modeling of parallel and distributed programs using the Aims toolkit", Software Practice and Experience 25-4 (1995), pp. 429—461.

[Zhou 1986] Zhou S.: "A Trace-Driven Simulation Study of Dynamic Load Balancing", IEEE Transactions on Software Engineering 14-9 (1988), pp. 1327—1341.

BIOGRAPHIES

Dieter F. Kvasnicka received his Masters degree in computer science in 1994 and his Ph. D. in 2000, both from the Vienna University of Technology. Since 1997 he is employed at the Institute of Physical and Theoretical Chemistry of the Vienna University of Technology. He is a member of the Special Research Project AURORA of the Austrian science fund FWF and works on numerical algorithms for high performance computing in materials science, including the development of message passing parallel applications and program libraries. Dieter Kvasnicka is the author of several publications and technical reports about parallel simulation, cluster computing, blocking techniques in symmetric eigenproblems and other numerical software, and high performance linear algebra and FFT algorithms implemented using High Performance Fortran, and he presented his findings on many international conferences and workshops.

Helmut Hlavacs received his Masters degree (Mathematics) in 1993 at the Vienna University of Technology, followed by his Ph.D. in 2000. From 1998 to 2000 he worked as a researcher for the European research project BISANTE at the Institute for Computer Science and Business Informatics at the University of Vienna. Since 2000 he is Assistant Professor at this institute. Furthermore he is member of the Austrian science project AURORA in the area of numerical algorithms for high-performance computing. Helmut Hlavacs is author of several publications, technical reports and project deliverables in the area of numerical mathematics, high-performance computing, workload modeling and network simulation.

Christoph W. Ueberhuber received his Masters degree in 1973 and his Ph.D. in 1976, both in the field of Mathematics. In 1973 he became Assistant Professor for Numerical Mathematics at the Institute for Numerical and Applied Mathematics, Vienna University of Technology, where he advanced to a tenure position in 1980. Since 1998 he is Associate Professor there.

He took part in many completed research projects in the area of numerical analysis, scientific computing, high performance computing, graphical data processing and image processing, technical data processing, and environmental protection. He also participated in two major research projects in the area of parallel computing and numerical algorithms and software for high-performance computers. He is author of 11 books and more than 100 publications in journals, books, and conference proceedings.