Automatic Performance Prediction to Support Cross Development of Parallel Programs *

Matthias Schumann
Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR)
Institut für Informatik / SAB
Technische Universität München (TUM), 80290 München, Germany
email: schumanm@informatik.tu-muenchen.de / matt@wap0207.chem.tu-berlin.de
http://wwwbode.informatik.tu-muenchen.de/~schumanm

Abstract

Cross development techniques are very attractive to be applied in high performance scientific computing because the parallel systems are expensive and should rather be utilized to perform production runs than to debug parallel programs. However, if development and execution platforms differ, techniques are required to efficiently predict the performance that will be actually gained on the target system.

In this paper, we present a performance prediction methodology that is able to efficiently support cross development of deterministic real-life message-passing programs for recent parallel multicomputer systems. The whole prediction process is supported by an environment of automatic tools. We demonstrate the feasibility of our approach by considering four programs from the NAS parallel benchmark suite and a multi-point boundary-value problem solver developed at TUM. The programs are implemented under the NX/NXLib and PVM message-passing environments. Our experimental environment comprises Paragon, iPSC/860, and GC/PowerPlus multicomputer systems, and a small cluster of workstations that serves as development platform.

1 Introduction

Porting and developing programs for multicomputers requires enormous efforts and leads to frequent test runs during the implementation. A computing center, however, is more interested in executing production runs on the costly multicomputers than in spending computing power for debugging purposes. To withdraw this load from the systems, portable parallel programming environments such as NX/NXLib [22] or PVM [9] are applied. Such environments enable the user to cross develop parallel applications on a cluster of workstations (COW). The COW - or even a single workstation - emulates a multicomputer by executing parallel processes in quasi-concurrence, i.e. in multiprocessing mode.

Unfortunately, COWs usually do not provide the degree of parallelism a multicomputer provides. Moreover, the performance characteristics of COWs and multicomputers differ significantly. As a consequence, performance debugging, i.e. identification and elimination of performance bottlenecks, still has to be carried out on the target multicomputers. In order to withdraw a major part of those activities from the multicomputers, methodologies and tools are required that are capable of predicting the desired performance values without actually accessing the target systems. A methodology suitable for this purpose has to meet three key constraints:

1. It has to support a wide range of real-life programs and real-life machines.
2. The prediction time has to be reasonable.
3. The predicted performance values have to be sufficiently accurate.

2 Related Work and Research Objectives

2.1 State of the Art

Traditional performance prediction methodologies require the user to manually model the behavior of programs and target systems at various abstraction levels, and incorporate costly analysis techniques such as stochastic analysis or fine grain simulation. Honoring the second of the three key constraints mentioned above, this is not feasible unless the considered programs and machines are of very limited complexity, leading to a violation of the first constraint.

R. Aversa et al. [1] developed an execution-driven simulator for heterogeneous environments using PVM. While the interconnection network is modeled in great detail, the delay of computational tasks is estimated by means of performance ratios of target platforms and development platforms. They achieve accurate results and reasonable simulation times for small program kernels.

G. Chillariga and B. Ramkumar [6] presented a simulation-based prediction approach focusing on the message-driven programming environment *Charm*. They emphasize relative performance criteria and estimate the delay of computational tasks by linearly scaling the delays measured by executing a reference program version on the target system. Unfortunately, they do not quantify the required simulation times.

A framework based on graph analysis to predict the execution time of parallel SISAL programs within a simulated test-bed to evaluate multiprocessor scheduling algo-

*We thank FORTWIHR, the Bavarian Consortium for High Performance Computing, for funding our work.

SPDT '96, Philadelphia PA,USA
© 1996 ACM 0-89791-846-0/96/05..$3.50

rithms was proposed by V.S. Sarkar [19]. Recently, a graph-based approach to automate the formulation of scalable workload models for message passing programs containing synchronous and pipeline communication phases was introduced by R.J. Block et al. [4].

Performance predictors integrated into parallelizing compilers to automatically guide the selection of data partitioning strategies and program transformations were developed by V. Balasundaram et al. [3] for the ParaScope environment, by T. Fahringer and H.P. Zima [8] for the Vienna FORTRAN Compilation System, and by Y. Seo et al. [20] for the PCASE environment. A performance predictor for the Data-parallel C language was developed by M.J. Clement and M.J. Quinn [7]. A.D. Malony and K. Shanmugam [13] presented a performance extrapolation tool to support cross development of data-parallel C++ programs written in the *p*C++ language. All these approaches are restricted to programs of the highly regular SPMD type as they are generated by parallelizing compilers. As a benefit, most of these approaches incorporate a scalable, mathematical workload model; i.e. they are able to predict the performance of those highly regular programs for varying processor counts and problem sizes by using only a single mathematical workload model.

Yet, existing performance prediction methodologies for parallel programs either require manual modeling efforts or substantial prediction times, or severely restrict the target programs to certain narrow classes. Many of them predict execution time only and provide custom tailored user interfaces. A frequently seen deficiency is that reported results are often obtained by considering case studies of small program kernels rather than real-life parallel programs.

2.2 Performance Requirements and Design Objectives

Our initial claim is that prediction time and generality are more critical than accuracy. Nobody will ever use a prediction methodology if he first has to spend considerable effort to design and validate models of the program and the machine, if the prediction time is by orders of magnitudes slower compared to program execution and measurement, or if predictable programs are restricted to a few classes. Of course, neither will anyone use prediction if it is not reasonably accurate.

As our reference point for the prediction time, we consider a time interval that is most familiar one to users of cross development techniques: a program's compile-and-run time on the development platform.

To be suitable for performance debugging, the predicted performance values have to be sufficiently accurate to properly identify performance bottlenecks of parallel programs and to evaluate the performance gain obtained by different modifications to the program. The predicted performance criteria should cover a wide spectrum of metrics and should be visualized by means of widely used performance analysis tools.

As such, it is the goal of our research to investigate to what extent a compromise between accuracy, prediction time, automation, and generality can be settled in order to efficiently support cross development by performance prediction techniques. In order to evaluate this question, our central objective is the development of a fully tool-supported performance prediction methodology comprising automatic modeling of target machines and programs. The methodology should be so general as to consider widely-used parallel programming environments, recent target systems, and real-life parallel programs.

2.3 Target Environment

The target machines we consider are real-life parallel multicomputers. Concerning the target programs, we focus on real-life message-passing programs from the domain of high performance scientific computing (HPSC). Because most HPSC codes today are still written in FORTRAN, we concentrate on FORTRAN77 programs that make use of widely used message-passing environments. In order to fit those programs onto the development platforms, they have to be scaled down by reducing the process count, the problem size, and/or the iteration count. However, even on the actual target systems, performance debugging is almost ever carried out by using scaled down program versions because the required trace files grow extremely large and the intrusion of the program becomes significant. Fortunately, basic algorithm behavior and most fundamental bottlenecks and inefficiencies in parallel programs are usually already apparent when viewed with scaled down program versions.

We do not consider performance prediction for operating system and I/O activities in this work. Moreover, we focus on a static process model and do not consider multiprocessing on the nodes of the target machines. Many codes applied in the HPSC domain make use of a static process model and reduce their I/O requests to reading the initial data and to writing the results. I.e. operating system activities such as process creation, process scheduling, process deletion, and I/O are only occurring at the start and at the end of a program. Hence, the substantial efforts arising to predict the performance of those activities are not justified by the accuracy that will be gained.

2.4 Our Approach

The kernel of our approach is a general, high level language performance model of parallel multicomputers, denoted as abstract parallel numerical machine (APNM) model.

For every considered target system, the parameters of the machine model are automatically identified by a machine analyzer based on a micro benchmark. For the sake of model generality and prediction time, we ignore modeling network topology and contention by assuming that - concerning state-of-the art networking technology - the corresponding performance impacts are only of second order at moderate node counts as they are utilized in performance debugging.

The workload parameters of the target programs are automatically identified by a static program analyzer/instrumenter tool, and in a subsequent profile run or execution-driven simulation. Integration of workload and machine model and analysis of the resulting system model are carried out either *on-line* during the execution-driven simulation or *off-line* by means of a graph construction and analysis tool. The predicted performance criteria are visualized by a widely used performance analysis tool.

2.5 Organization of the Paper

The remaining sections of the paper are organized as follows. First, we introduce our performance prediction methodology and the supportive environment of automatic tools. Subsequently, we present the underlying machine and workload models and describe the tools used for parameter identification and system model analysis. Finally, we evaluate our

methodology, discuss the results, and present future research directions.

3 Performance Prediction Environment

We incorporate a machine model based on the concept of an abstract parallel machine which is able to directly execute high level language (HLL) instructions. Thus, we are able to exploit target machines offering different compilers, running different operating systems, and providing various message-passing environments. We denote the current implementation of the machine model as the abstract parallel numerical machine (APNM) model, because it emphasizes numerical applications written in the FORTRAN77 language.

The APNM prediction environment offers two prediction methodologies, namely on-line and off-line prediction, as shown in Figure 1. Both methodologies require a parameter analysis of the considered program and of the considered target machine. The *MA* machine parameter identification tool evaluates the execution times of a set of predefined, program independent, high level language (HLL) instructions and communication parameters which are stored in a machine parameter data base. The *A3* workload parameter identification and instrumentation tool identifies basic blocks in the source code, and stores the HLL instructions contained in each basic block in a basic block data base. Furthermore, it inserts basic block counters, simulation time counters, and library hooks into the source code.

To perform an *on-line prediction*, the instrumented program has to be linked with the *simulation library A3simu*. The resulting executable - given the program's basic block data base and the machine parameter data base of the considered target machine as additional input - is executed in an execution-driven simulation on the development platform. After termination of the simulation, the user is provided with the program execution time predicted for the specified target system. Hence, by combing profiling and prediction into a single run, on-line prediction enables the user to obtain a quick execution time prediction. Yet, neither does it provide the user with further performance metrics, nor is it able to visualize the predicted execution behavior. If the user is interested in those features, an off-line prediction has to be performed.

To perform an *off-line prediction*, the instrumented program has to be linked with *tracing library A3trace*. The resulting code is executed in a profile run on the development platform to generate trace files recording the occuring communication events and the basic block executions between them. The trace files, the basic block data base, and the machine parameter data base are fed into the *PE performance predictor tool*. PE assembles a weighted task graph representing the program's execution behavior on the target machine, and performs a critical path analysis upon the resulting graph. Eventually, the estimated performance metrics are provided by means of widely used performance analysis tools. In our approach, we employ *TATOO* [5], a trace analysis tool developed by collaborate researchers at LRR-TUM. Moreover, PE provides a machine independent characterization of the program execution by visualizing the execution count profile of abstract HLL instructions, and a machine dependent characterization by viewing the corresponding execution time distributions.

From a model evaluation point of view, off-line and on-line prediction are completely equivalent. As such, the predicted execution time values are the same, independent of the applied prediction methodology.

Both methodologies provide performance predictions for the problem size, the process count, and the input data specified in the respective profile or simulation run. We do not scale our workload model, because an accurate and automatic scaling (e.g. by statistical regression techniques as proposed by P. Mehra et al. [14]) of models describing irregular programs is not feasible with respect to the prediction time.

4 Modeling Approach

In order to automatically reflect changes in the program code, and to establish a relation between the predicted performance and the program code, performance prediction should be applied at the source code level. Because modeling the compiler and the operating system is not feasible and, moreover, not portable, many existing predictors model real-life machines by deterministic linear HLL execution models, i.e. by assuming that the machines are able to directly execute HLL code.

R.H. Saavedra [18] introduced a sequential abstract FORTRAN machine. He models a program P in terms of a vector E holding the execution counts of primitive FORTRAN instructions occuring during an execution of P. In correspondence, the target machine is described in terms of a vector C of the instruction execution costs arising on the target machine. The predicted execution time $\widetilde{T}$ of P is computed by a linear execution model, i.e. by computing the scalar product of both vectors: $\widetilde{T} = E * C$. A similar model extended by message-passing instructions to handle parallel SPMD programs, was proposed by V. Balasundaram et al. [3].

We extended Saavedra's machine model by an abstract message-passing environment that provides basic communication and synchronization operations. By abstracting and reducing the message-passing functions to a certain core functionality, we achieve a portable approach to performance prediction under different message-passing environments. We put a task graph model on top of the traditional instruction execution count model, in order to be able to handle even irregular parallel programs. Moreover, our APNM model extends Saavedra's model of the abstract FORTRAN machine by considering features of today's RISC microprocessors such as memory hierarchy and parallel execution units.

4.1 Machine Model

4.1.1 Architecture Model

Precisely modeling individual interconnection network topologies and the involved network contention in a detailed simulation is infeasible because the demanded modeling and model analysis efforts contradict the stated prediction time requirements. Currently, predictors that are able to efficiently handle network contention are restricted to the clustered communication occuring in SPMD programs, cf. [3], [8].

These days, however, modern multicomputers are equipped with high bandwidth, worm-hole routing, virtually fully interconnected networks, additional communication processors, active network adaptors, and novel routing strategies, in order to abstract from the underlying topology and to reduce the performance impact of network contention. By ignoring topology and contention in modern

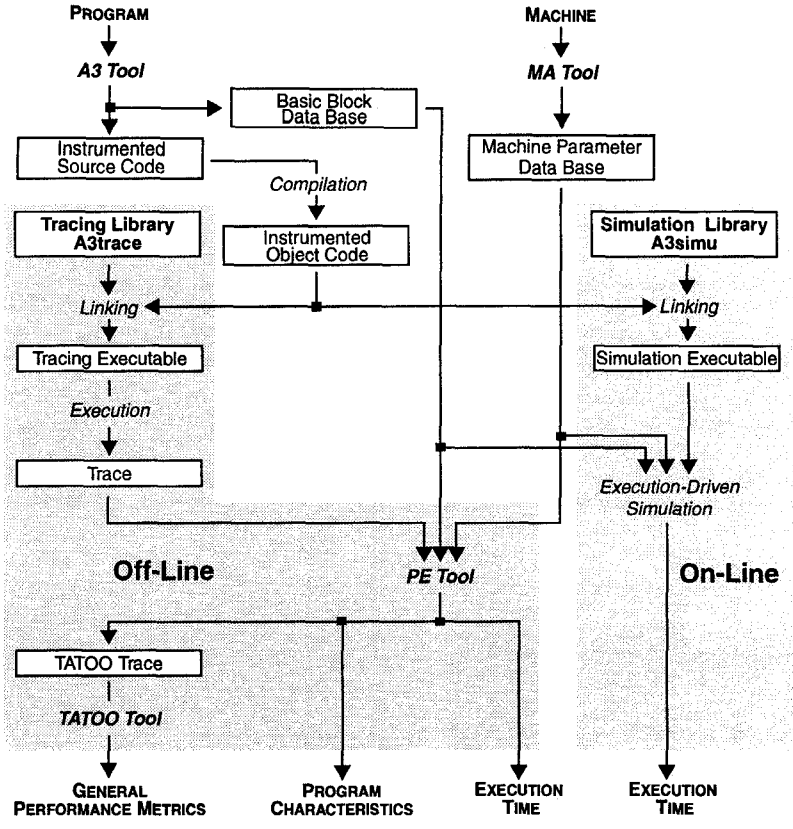

Figure 1: The APNM Performance Prediction Environment

multicomputers - namely an iPSC/860 and an iWARP array - M.J. Clement and M.J. Quinn [7] achieved speedup predictions with an accuracy of 20% in configurations of up to 64 nodes. We believe that - considering moderate process counts as they are usual in performance debugging - this is currently the most feasible approach to predict the performance of programs that involve irregular communication structures. Therefore, we model multicomputers as homogeneous collections of fully connected processing nodes. As a drawback of this approach, our predictions severely overestimate program performance for multicomputers that do not incorporate recent networking technology (see Section 7.4).

Each processing node (PN) is modeled to execute a single process consisting of abstract FORTRAN instructions. A processing node consists of an application processor (AP) that is able to execute abstract instructions, a communication engine (CE) that connects the processing node to the interconnection network, and a main memory (MM). The application processor comprises a computation/control unit (CCU), a load store unit (LSU), a register file (RF), and a first level data cache (DC). The communication engine serves as a network adaptor. By spooling data messages and synchronization messages to and from the interconnection network, it takes care of servicing the communication requests posted by the application processor. The possibility to overlap computation and communication by means of the communication engine reflects the presence of dedicated message co-processors in modern multicomputers. The entire architecture model is summarized in Figure 2.

4.1.2 Computation

The computational performance of a target machine is parameterized by the execution cost vector which holds the execution times for 179 primitive FORTRAN instructions.

Our approach mainly differs from Saavedra's in the way the latency of memory accesses is handled. In their traditional form, linear HLL execution models do not distinguish between latencies caused by accesses to different levels of the memory hierarchy. I.e. they either represent best case models assuming that all the data is in the cache, or worst case models assuming that all the data has to be fetched from main memory. To deal with this problem, R.H. Saavedra makes use of a cache simulator. This is not feasible in our approach because of the substantial time effort required to generate the address trace, to model the memory hierarchy and to carry out the simulation.

In contrast to reality, we model the memory access behavior of a program to be independent of the target machine. By assuming memory bound programs and large data structures as they are typical for the HPSC domain, we heuristically try to determine the target of a memory access from looking at the source code. This is done by means of the A3 tool during the identification of the static workload parameters.

The machine model distinguishes data transfers between the register file and the cache, and data transfers between the register file and the main memory. Accesses to the main memory are further categorized by whether the transfer is carried out in isolation or is part of a stream of transfers to

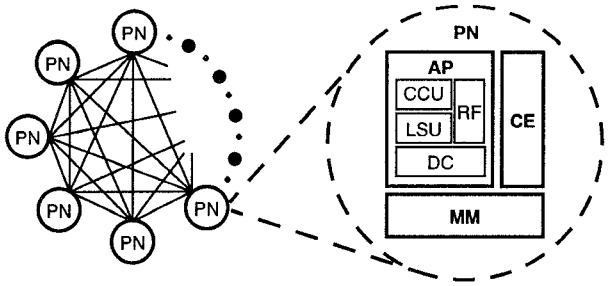

Figure 2: Architecture Model

consecutive memory locations. This distinction reflects the performance differences of isolated and streamed accesses to the main memory caused by the different degrees of spatial locality exploited.

4.1.3 Communication

In order to be able to consider diverse message-passing environments, we define an abstract message-passing environment featuring the core functionality available in almost all message passing environments: request of configuration information, synchronous and asynchronous (following NX semantics) peer-to-peer and broadcast transmission, global synchronization, and typed communication. In the same way, programs from the HPSC domain often use only the core functionality of message-passing environments in order to guarantee high portability between different environments.

Basically, latencies L of communication functions are modeled by dividing the message length S by a bandwidth β, and by adding a start-up time α: $L = \alpha + (S/\beta)$. In order to account for discontinuities caused by packetization effects, individual parameters $\alpha(k)$ and $\beta(k)$ are kept for n intervals $[B_{k-1}, B_k]$, $k \in \{1, \ldots, n\}$, of the message length. The interval bounds B_k are discrete values of the message length. However, the latencies of broadcasts and global synchronizations are further affected by the number of participating processes. Therefore, the communication parameters are additionally parameterized by the process count P, leading to the following piecewise linear model of communication latencies:

$$\begin{array}{l} L(S,P) = \alpha(k,P) + \frac{S}{\beta(k,P)} \\ \forall S \in [B_{k-1}, B_k], \quad k \in \{1, \ldots, n\} \end{array} \tag{1}$$

4.2 Workload Model

Prediction methodologies that abstract computation by a linear HLL execution model require the computational tasks to be modeled by execution count lists of the HLL instructions in the program. Unfortunately, execution count lists lack the possibility to establish communication and precedence relations between the computational tasks. Thus, traditional approaches based on a linear HLL execution model are restricted to certain algorithmic structures, to SPMD programs, or do not consider parallel programs at all. Irregular parallel programs, however, have not been considered yet.

An elegant way to model irregular parallel programs are directed acyclic task graphs. Task graphs decompose a parallel program into its computational tasks, and the precedence and communication relations between them. In order to combine the advantages of the HLL abstraction level and of task graph models, we introduce a hybrid workload modeling approach. The basic idea is to describe the tasks' abstract computational costs by execution count lists of HLL instructions, and to annotate the vertices representing the tasks with the corresponding lists. Furthermore, directed communication edges are annotated with the lengths of the transferred messages. The resulting graph is denoted as annotated directed acyclic task graph (ATG).

There is a very obvious procedure to construct and analyze a program's ATG. It is to explicitly assemble the graph post mortem, i.e from trace data containing the basic block execution counts and the communication events, to subsequently determine the predicted costs of vertices and edges by evaluating the machine model, and to finally analyze the resulting system model by graph theoretic analysis techniques. We denote this form of performance prediction as *off-line prediction*. It is often argued that the size of acyclic task graphs tends to explode if real-life programs are considered. However, as we demonstrate in Section 7, even communication intensive programs can be handled by off-line prediction if they are scaled down as it usually done in order to carry out performance debugging.

However, there is an interesting alternative to explicitly constructing and analyzing a program's ATG. An ATG describes program execution behavior and, vice versa, a program's execution resembles a traversal of its ATG. As such, a program itself can serve as a basis to evaluate its ATG in an execution-driven simulation. In order to derive the HLL instruction execution count lists and the message lengths during program execution, the original program has to be properly instrumented. During program execution, this data can be handed over to a simulation library which evaluates the machine model and keeps track of the simulation time in the system. We denote this form of performance prediction as *on-line prediction*.

5 Parameter Identification

5.1 Machine Parameters

The machine model parameters of a target machine are automatically identified by the MA machine analyzer tool. In fact, the machine analyzer consists of two separate parts. A sequential micro benchmark measures the execution times of the computational HLL instructions, while a separate communication benchmark measures communication performance parameters. Both parts are written in standard FORTRAN77. When migrating to another target machine, only the timer calls have to be adapted. In order to exploit another message-passing environment, the syntax of the message passing calls has to be adapted, too.

In order to measure the execution time of an HLL instruction or of a communication operation, the benchmarks apply a differential measurement technique: a synthetic instruction sequence denoted as *body of test* is executed for a number of iterations $ITER$. From the measured execution time we subtract the execution time of a reference body that differs from the body of test only by the considered operation. The obtained time is denoted as an observation. The value of $ITER$ has to be properly adapted to the clock resolution and the performance of the target system.

In order to obtain a meaningful statistic, a sample of $REPEAT$ observations is taken. A sample's coefficient of

variation indicates the quality of the measurement. For a proper measurement, the coefficient of variation should lie well below 5%. Unfortunately, the measured average values are affected by concurrent operating system activities and by network contention caused by other space sharing users on the system under test. In order to gain load independent and reproducible results, we take the minimum of a sample as our measurement, resulting in a deterministic best case model of the target system.

The runtime of the complete benchmark set on a considered target machine strongly depends on its clock resolution but is usually of the order of hours.

5.2 Workload Parameters

5.2.1 Static Parameters

The static workload model parameters of a target program are automatically identified by the A3 tool which is implemented as a compiler front-end built on top of R.K. Moniot's FTNCHEK tool [15]. The A3 tool parses the program's source code, identifies its basic blocks, stores the contained HLL instructions in the basic block data base, and instruments the program with basic block execution counters for off-line prediction and simulation-time counters for on-line prediction. Furthermore, A3 replaces communication functions with stubs that invoke functions in the trace and simulation libraries and afterwards call the actual communication functions. By a configuration file, A3's instrumentation facilities can easily be adapted to different message-passing environments.

When parsing a basic block to identify the enclosed abstract instructions, A3 heuristically tries to determine whether an operand already resides in the register file or has to be transferred from/to the cache or - in a streamed or isolated access - from/to main memory. Unfortunately, a detailed outline of the heuristic would go beyond the scope of this paper, but the heuristic basically does the following:

- Multiple accesses of the same scalar variable or the same array element in a basic block are reduced to a single, yet undetermined memory access, considering register allocation optimization by the compiler. An access of an array element which is likely to fall in the same cache line as a previously accessed element of the same array is assumed to hit the cache.
- Whether a yet undetermined memory access is assumed to hit the cache or whether the data is assumed to be transfered in a streamed or isolated transfer from main memory, depends on the address reference class. The address reference class indicates whether the addresses referenced in two consecutive executions of the same source code operation are likely to be arbitrarily changing, to be locally changing, or even to be static. In our heuristic, the class *static* corresponds to a cache hit, the class *locally changing* corresponds to a streamed access of main memory, and the class *arbitrarily changing* corresponds to an isolated access of main memory.

  The address reference class of an access to a scalar variable is defined to be *static*. The address reference class of an access to an array element depends on the complexity of the individual index expressions, on whether they contain the loop variables of the immediately surrounding loop, and on their position in the index.

Moreover, by simply counting the number of arithmetic operators and the number of memory accesses in an arithmetic expression, A3 determines whether the execution of a load/store instruction can be overlapped with the execution of arithmetic instructions.

5.2.2 Dynamic Parameters

The dynamic workload parameters, i.e. the computational tasks and their precedence and communication relations, are identified either implicitly during execution-driven simulation or explicitly identified in a profile run. Depending on the library linked to the instrumented code, either on-line prediction is carried out or trace files recording the execution profile of every parallel process are generated.

A trace file contains the encountered communication events in the order of their occurrence and - between those entries - the execution count of every basic block executed in between the respective communications. Here, it should be noted that the trace files are machine independent; i.e. they do not contain time stamps. In order to overlap flushing of the trace buffers and communication, the trace buffers are flushed when a communication event is encountered.

6 System Model Integration and Analysis

6.1 Off-line Prediction

Given the traces generated in a preceding profile run of the considered program, the PE tool is able to assemble the program's annotated task graph by applying an algorithm of complexity $O(|vertices|+|edges|)$. Provided with the target program's basic block data base and with the target machine's parameter data base, the PE tool transforms the annotated task graph into a weighted task graph. The weights of the vertices represent the delays of the corresponding computational tasks. They are computed by applying the linear HLL execution model. The weights of the edges represent communication latencies which are computed by evaluating the piecewise linear communication models.

In a subsequent analysis of the weighted graph, the PE tool computes the tasks' earliest and latest finishing times, and identifies the tasks and communications that belong to the critical path. The graph analysis is of complexity $O(|edges|)$ and is performed by an adapted version of a PERT network analysis algorithm proposed in [21]. Finally, the PE tool provides the user with the predicted execution time for the program, i.e. with the length of the critical path, and with the amount of communication time spent on the critical path. Optionally, the PE tool traverses the weighted graph again and generates a trace file compliant with the TATOO performance analysis tool, or characterizes the program execution in terms of HLL instruction execution frequencies and distributions.

6.2 On-line Prediction

In contrast to the off-line prediction technique described above, the idea behind on-line prediction is to compute and traverse the weighted graph in an execution-driven simulation. To carry out an execution-driven simulation, the program code is instrumented with simulation time counters and library hooks by means of the A3 tool, and the simulation library *A3simu* is linked to the resulting object code.

The simulation library controls the execution of the program. Before starting the program, it reads the program's

basic block data base and the target machine's parameter data base. From these data, the execution times of the individual basic blocks are predicted by applying the linear HLL execution model. During the execution of the program, every process keeps a local simulation time. Every time a basic block in the program is executed, a simulation time counter increments the local simulation time by the basic block's predicted execution time. When a communication event is encountered, the sender computes the estimated transfer time by calling a function in the simulation library, adds it to its local simulation time, and sends the result on top of the actual message to the destination process. By comparing the local simulation time with the transmitted arrival time, the receiving process updates its simulation time and proceeds with its execution. Eventually, after the program's termination, the predicted execution time is determined by finding the process with the largest simulation time.

It has to be noted, that off-line and on-line prediction are completely equivalent techniques. As such, the predicted execution times value are the same, independent of the applied prediction technique.

7 Discussion and Evaluation

7.1 Limitations

As a consequence of the deterministic workload modeling approach based on the annotated task graph, the spectrum of target programs is basically restricted to programs that show deterministic execution behavior. I.e. their execution on the development platform and on the target system must lead to the same graph model. This may especially not be the case if a program contains races, or if a program's behavior depends on the precision of mathematical libraries (e.g. iterative solvers) or on real-time events.

Tools and techniques to detect races in a program's execution are available, cf. [16]. If a program was found to contain races, a tool that enforces deterministic execution as it is proposed in [17] can be applied to generate a set of traces describing multiple characteristic execution behaviors of a single program. By following this approach, the performance for multiple potential execution behaviors can be predicted.

Concerning precision and real-time dependencies, a user has to rely on his own knowledge of whether his program contains such dependencies and how to eliminate them.

7.2 Experimental Environment

As target systems in our experiments, we use two generations of Intel multicomputers, i.e. the i860 based iPSC/860 and Paragon systems, and a PowerPC601 based GC/PowerPlus system [11] manufactured in 1994 by the German vendor Parsytec. We utilize partitions of up to 16 nodes - a size which is typical for performance debugging of parallel programs. A COW consisting of four SPARCstation10 serves as development platform. The parallel programs are developed on this platform by executing their processes in quasi-concurrence.

Our suite of test programs consists of NX and PVM implementations of the NAS benchmarks cg, ep, ft, and lu [2]. The problem sizes of the programs were chosen so that the execution times of the sequential codes are of the order of 30 seconds. While the development code is implemented under PVM, the multicomputer implementations to verify our predictions are running under NX on the Intel systems, and under PVM on the GC/PowerPlus respectively.

7.3 Accuracy

We start our evaluation by determining to what extent the fact of ignoring network topology and contention does affect the accuracy of the predicted results. We characterize accuracy in terms of the relative prediction error $E = (\widetilde{T} - T)/T$ which compares the predicted execution time $\widetilde{T}$ to the actually measured execution time T on a target system. T is obtained by taking the average runtime of a sample of 10 executions. The variability of T is characterized by the coefficient of variation V of the sample.

Table 1 shows accuracies and variabilities, as well as their root-mean-squares RMS and standard deviations σ, for NAS-cg. Containing frequent, irregular, long distance communications, NAS-cg is the most communication intensive of the four test programs.

	iPSC/860			Paragon			GC/Power+		
nodes	T	V	E	T	V	E	T	V	E
1	29	0	-9	23	0	-1	20	0	-16
2	16	0	-13	11	0	-3	13	0	-20
4	9.8	0	-10	6.4	0	4	9.9	0	-25
8	6.9	0	-26	3.8	8	-1	10	0	-40
16	5.8	0	-27	2.9	0	-4	15	0	-46
RMS		0	19		4	3		0	32
σ		0	9		3	3		0	13

Table 1: Relative prediction errors (%) and measurement variability (%) for NAS-cg

For the Paragon system, the predicted execution times always lie within 4% of the measured times and, hence, fall within the variability of the actual measurements. For the other systems, the prediction underestimates sequential performance by 9% and 16%. The deviation steadily increases with the node count and reaches 27% and 46% respectively. This phenomenon clearly is caused by ignoring network contention. It shows that the accuracy is heavily affected by the networking technology of the target systems.

Concerning the Paragon which provides state-of-the art networking technology, the predictions are sufficiently accurate. In the GC/PowerPlus system, however, a communication co-processor is not exclusively dedicated to its processing node; the communication processor also has to take care of routing and forwarding messages that are passing by. As a consequence, the communication bandwidth of a node is significantly reduced under heavy network load. Providing a circuit switched interconnection network, the networking technology of the iPSC/860 exhibits performance in-between those of the two other systems.

Table 2 summarizes the results obtained for the four NAS programs. The relative prediction errors E obtained for an individual test program are summarized in terms of their root-mean-square $\widehat{E}$ and their standard deviation $\sigma(E)$. The variability of the measurements is expressed by the root-mean-square $\widehat{V}$ of the coefficients of variation V. In the AV row, the averages of $\widehat{E}$ and $\widehat{V}$ are listed.

Table 2 confirms the results obtained in Table 1. Being at 7%, the average prediction error for the Paragon system is sufficiently accurate compared to the average measurement variability of 3%. The low standard deviations of the relative prediction errors emphasize the successful prediction. Reaching an average prediction error of 14%, the results ob-

	iPSC/860			Paragon			GC/Power+		
	$\widehat{E}$	$\sigma(E)$	$\widehat{V}$	$\widehat{E}$	$\sigma(E)$	$\widehat{V}$	$\widehat{E}$	$\sigma(E)$	$\widehat{V}$
cg	19	9	0	3	3	4	32	13	0
ep	14	0	0	5	5	1	3	2	1
ft	16	11	1	7	3	3	23	20	0
lu	5	3	0	11	3	2	21	6	0
AV	14		0	7		3	20		0

Table 2: Summary of relative prediction errors (%) and measurement variabilities (%) for the NAS test suite

tained for the iPSC/860 can be considered acceptable. The accuracy obtained for the GC system, however, is definitely insufficient.

The strong relation between the accuracy and the network utilization becomes obvious by looking at the results for NAS-ep. For this program, which has negligible communication, the accuracies obtained for all the three target systems are acceptable.

7.4 Prediction Time

We evaluate the required prediction time by presenting measurements for the moderately communicating NAS-ft program (Table 3) and for the heavily communicating NAS-cg program (Table 4). In both tables, the runtime T of the plain, un-instrumented program on the COW is compared to the prediction times arising for on-line and off-line prediction. On-line prediction is solely characterized by the runtime T_S of the execution driven simulation. Off-line prediction is characterized by the runtime T_T of the trace generating profile run and by the time T_{PE} that is required to analyze the system model by means of the PE tool.

The predictions are carried out during regular office hours when cross development usually takes place. Therefore, the COW and its Ethernet link are not solely dedicated to cross development and considerable variations of the measured execution times are observed. The execution times are listed in terms of the average values of samples of 10 runs and their standard deviations σ. The process count P indicates the number of processes spawned on the COW.

			on-line		off-line		
P	T	$\sigma(T)$	T_S	$\sigma(T_S)$	T_T	$\sigma(T_T)$	T_{PE}
1	96	3	114	3	111	0	1
2	91	9	97	8	102	7	1
4	74	8	107	25	93	15	1
8	89	15	87	8	121	40	3
16	104	4	137	18	182	15	6

Table 3: Prediction times for NAS-ft (seconds)

For NAS-ft, the average simulation time is in the worst case by a factor of 1.44 higher and in the best case by a factor of 0.97 lower than the average execution time of the plain development program.

Compared to the average profile times which exceed the execution time of the plain development program by factors of 1.12 to 1.75, the analysis time T_{PE} is negligible - even at a process count of 16. The size of the traces generated per process reached from 5 KB in the sequential case up to 77 KB for the run with 16 participating processes. This corresponds to trace generation rates per process of 53 bytes/s and 758 bytes/s respectively, expressed in terms of the trace volume generated per second of the runtime of the plain code.

As a result of the low prediction time overheads and of the large runtime variations caused by the ever changing load conditions during office hours, the user is hardly able to recognize the runtime difference between plain development runs and both kinds of prediction runs.

			on-line		off-line		
P	T	$\sigma(T)$	T_S	$\sigma(T_S)$	T_T	$\sigma(T_T)$	T_{PE}
1	45	1	69	1	68	2	1
2	72	0	111	4	140	4	6
4	112	2	164	6	230	41	17
8	72	19	255	20	375	13	34
16	292	24	378	6	856	60	67

Table 4: Prediction times for NAS-cg (seconds)

In the case of NAS-cg, the qualitative behavior of the simulation is similar to that observed for NAS-ft. The runtime averages differ by factors in the 1.30-1.54 range from those of the plain code.

The profile times, however, are stronger increasing with the number of processes, i.e. with the number of communications. While the average profile time exceeds the runtime of the plain code only by a factor of 1.5 in the sequential case, the factor increases up to 2.82 at a process count of 16. This phenomenon is mainly caused by the I/O requests posted to write the trace data. For NAS-cg, the trace size generated per process reaches from 1.7 KB in the sequential case up to 1 MB for the run with 16 participating processes, corresponding to trace generation rates of 39 bytes/s and 3.6 KB/s respectively.

Of the 67 seconds that the PE tool required to analyze the system model at a process count of 16, the CPU was busy 46 seconds to read and parse the trace data, 19 seconds to construct the weighted task graph, and 2 seconds to carry out the critical path analysis. Thus, the main part of the analysis time is spent parsing the trace and allocating small chunks of memory to hold the individual elements of the annotated task graph.

Nevertheless, exceeding the runtime of the plain code by a factor of 3, the time required to predict the performance of a heavily communicating real-life parallel program is well in the order of magnitude of its compile-and-run time on the development platform.

With the current version of the trace library, a trace entry is generated whenever a communication event is encountered. Hence, a more sophisticated trace buffer management should lead to a considerable improvement of the trace generation process. By improving memory allocation and by parallelizing graph construction, it should be possible to improve the graph analysis, too.

7.5 Case Study

In order to successfully conduct performance debugging studies, the programmer has to be able to identify the portions of the programs that require tuning and must not waste time improving code that is not contributing to a bottleneck.

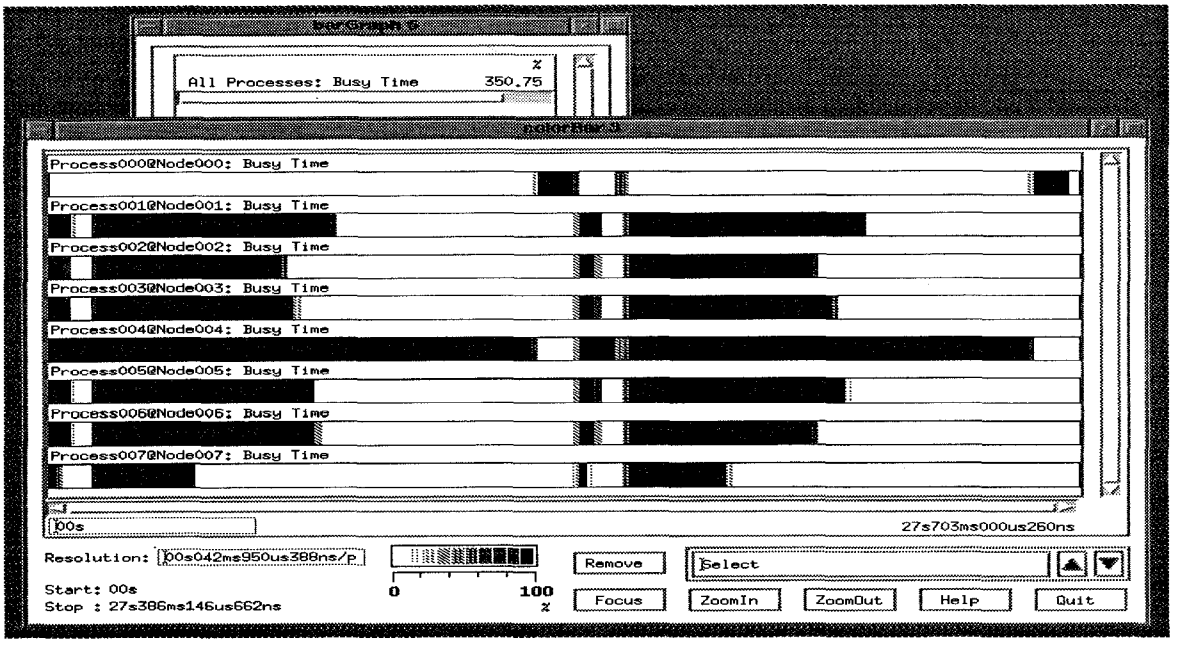

Figure 3: Predicted execution behavior of MUMUS visualized by means of the TATOO tool

We finally present a short case study describing the successful application of our performance prediction environment during the development of a real-life parallel program. The program is MUMUS [10], a multi-point boundary-value problem solver which is applied to solve optimal control problems arising in flight path optimization.

We ran an eight process configuration of the initial code on the development platform, and carried out an off-line performance prediction for the Paragon system. Figure 3 shows the predicted execution behavior visualized by means of the TATOO tool. It shows an extract of TATOO's utilization Gantt chart, and a bar-graph summing up the utilizations of all eight processes in the program to be 350.75%. With respect to the maximum possible utilization of 800%, the prediction of 350% indicates that the eight processes are blocked in communication calls for 56% of their total execution time. The corresponding measurement using the ParAide environment on the Paragon system revealed an actual blocking time of 59%.

The utilization Gantt chart shows a farmer/worker communication pattern whose critical path is dominated by Process004. In order to gain a higher utilization of the processes, the data distribution algorithm was redesigned. Predicting the performance after the integration of the new distribution algorithm lead to a predicted process utilization of 64%, while actual measurements on the Paragon system revealed a process utilization of 65%. Thus, by providing reliable performance estimations, the prediction environment enabled us discover the underlying performance bottleneck.

Here, it should be noted that ratios of predicted performance metrics (the process utilization, for example, is the ratio of the time the process was busy doing computation and the total execution time of the process) become very accurate if the variation of the individual relative prediction errors are of similar size. The reason is, that the errors tend to be eliminated during the computation of the ratio.

8 Conclusions and Future Research Directions

We presented a performance prediction environment comprising two methodologies, namely on-line and off-line prediction, in order to support cross development of deterministic parallel programs. The methodology comprises a set of tools allowing to automatically derive the model parameters of the target programs and of the target machines, and to construct and analyze the system model. The predicted performance is visualized by means of a recent performance analysis tool.

In contrast to other existing work, we consider real-life parallel programs developed for widely used message-passing environments. The achieved prediction time, which includes the procedure of modeling the target programs, is of the order of magnitude of the compile-and-run time on the development platform. The approach is highly portable and, with respect to a certain core functionality, able to predict performance between different message passing environments.

The low prediction time is partly achieved by ignoring performance impacts caused by network contention. For communication intensive programs, hence, prediction errors below 15% can only be gained for multicomputers that are equipped with state-of-the-art networking technology. However, investigations regarding the efficient treatment of network contention are subject of future research.

To date, our environment concentrates on a certain core functionality of message-passing environments. In future implementations, we think of extending the underlying models to cover a great part of the PVM's message-passing functionality functionality, and to integrate the prediction environment into *The Toolset* [12] which is currently developed at LRR-TUM.

Techniques have to be investigated in order to deal with aggressive compiler optimization and microprocessor architectures that provide a high degree of fine granular parallelism. This is generally a weak point of the abstract high

level language machine model. Yet, this should not be a substantial drawback because macroscopic performance bottlenecks should already be visible at low levels of optimization.

Acknowledgment

We thank Stefan Hager, Andreas Lobinger, Thomas Stephan, Norman Thomson, and Michael Uemminghaus for their interest in our work and for supporting tool implementation, test program preparation, and measurements. Furthermore, we thank Thomas Beisel at Rechenzentrum Universität Stuttgart and Thomas Hiller at Technische Universität Hamburg-Harburg for kindly supporting our work on their Paragon and GC/PowerPlus systems respectively.

References

[1] R. Aversa et al. The use of simulation for software development in heterogeneous computing environments. In *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 581–590, Athens, GA, November 1995.

[2] D. Bailey et al. The NAS benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, Moffet Field, CA, March 1994. Available at http://www.nas.nasa.gov/NAS/NPB/.

[3] V. Balasundaram et al. A static performance estimator to guide data partitioning decisions. In *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (POPP)*, Williamsburg, VA, April 1991.

[4] R. Block, S. Sarukkai, and P. Mehra. Automated performance prediction of message-passing programs. In *Proceedings of Supercomputing 1995, San Diego, CA*, December 1995.

[5] R. Borgeest, B. Dimke, and O. Hansen. A trace based performance evaluation tool for parallel real time systems. *Parallel Computing*, 21(4):551–564, April 1995.

[6] G. Chillariga and B. Ramkumar. Performance prediction for portable parallel execution on mimd architectures. In *Proceedings of IPPS'95, Santa Barbara, CA*, pages 630–634. IEEE Computer Society, April 1995.

[7] M.J. Clement and M.J. Quinn. Analytical performance prediction on multicomputers. In *Proceedings of Supercomputing '93, Portland, OR*, pages 886–894. IEEE Computer Society Press, November 1993.

[8] T. Fahringer and H.P. Zima. A static parameter based performance prediction tool for parallel programs. In *International Conference on Supercomputing 1993 (ICS'93)*, pages 177–189, Tokio, 1993.

[9] A. Geist et al. *PVM: parallel virtual machine - a users' guide and tutorial for networked parallel computing.* Scientific and Engineering Computation. The MIT Press, Cambridge, MA, 1994.

[10] M. Kiehl, R. Mehlhorn, and M. Schumann. Parallel multiple shooting for optimal control problems under NX. *Optimization Methods and Software*, 4:259–271, 1995.

[11] F. Langhammer. The PowerStone project. Technical report, Parsytec Computer GmbH, Auf der Hüls 183, 52068 Aachen, Germany, 1993.

[12] T. Ludwig et al. THE TOOL-SET – an integrated tool environment for PVM. In J. Dongarra, M. Gengler, B. Touracheau, and X. Vigouroux, editors, *Proceedings of EuorPVM'95 Short Papers*, Lyon, France, September 1995.

[13] A.D. Malony and K. Shanmugam. Performance extrapolation of parallel programs. In *Proceedings of the 1995 International Conference on Parallel Processing, volume II*, pages 117–120, 1995.

[14] P. Mehra, C.H. Schulbach, and J. C. Yan. A comparison of two model based performance prediction techniques for message passing parallel prograns. In *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems*, pages 181–190, Nashville, Tennessee, May 1994.

[15] R.K. Moniot. FTNCHEK version 2.7. Available from The Netlib Repository, http://www.netlib.org, 1993.

[16] R.H.B. Netzer, T.W. Brennan, and S.K. Damodaran-Kamal. Debugging race conditions in message passing programs. In *Proceedings of the 1st ACM Symposium on Parallel and Distributed Tools, Philadelphia, PA*, May 1996.

[17] M. Oberhuber. Elimination of nondeterminacy for testing and debugging parallel programs. In Mireille Ducassee, editor, *Proceedings of 2nd Int. Workshop on Automated and Algorithmic Debugging*, Available at http://wwwbode.informatik.tu-muenchen.de/~oberhube/PUBS/PS/aadebug95.ps.gz, May 1995.

[18] R.H. Saavedra. *CPU performance evaluation and execution time prediction using narrow spectrum benchmarking.* PhD thesis, University of California at Berkely, 1992.

[19] V. Sarkar. *Partitioning and scheduling parallel programs for multiprocessors.* Research Monographs in Parallel and Distributed Computimg. The MIT Press, Cambridge, Ma., 1989.

[20] Y. Seo et al. Static performance prediction in PCASE: a programming environment for parallel supercomputers. In K.M. Decker and R.M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems,*, pages 287–297. Birkhäuser Verlag, Basel, July 1994.

[21] T.A. Standish. *Data structures, algorithms, and software principles in C.* Addison Wesley Publishing Company, Inc., 1995.

[22] G. Stellner et al. Developing applications for multicomputer systems on workstation clusters. In W. Gentzsch and U. Harms, editors, *Proceedings of HPCN'94*, volume 797 of *Lecture Notes in Computer Science*, pages 286–292. Springer Verlag, Berlin, April 1994.