A Source Code Analyzer for Performance Prediction

MATTHIAS KÜHNEMANN*
Fakultät für Informatik
Technische Universität Chemnitz
kumat@informatik.tu–chemnitz.de

THOMAS RAUBER
Fakultät für Mathematik und Physik
Universität Bayreuth
rauber@uni–bayreuth.de

GUDULA RÜNGER
Fakultät für Informatik
Technische Universität Chemnitz
ruenger@informatik.tu–chemnitz.de

Abstract

Performance prediction is necessary and crucial in order to deal with multi-dimensional performance effects on parallel systems. The increasing use of parallel supercomputers and cluster systems to solve large-scale scientific problems has generated a need for tools that can predict scalability trends of applications written for these machines. In this paper, we describe a compiler tool to automate performance prediction for execution times of parallel programs by runtime formulas in closed form. For an arbitrary parallel MPI source program the tool generates a corresponding runtime function modeling the CPU execution time and the message passing overhead. The environment is proposed to support the development process and the performance engineering activities that accompany the whole software life cycle. The performance prediction tool is shown to be effective in analyzing a representative application for varying problem sizes on several platforms using different numbers of processors.

1 Introduction

One of the major challenges in parallel computing is to support the process of developing parallel software by effective automated performance tools, i.e. performance engineering approaches in certain design phases of parallel software should accompany this process until the completion of the parallel application. Performance engineering activities [15] range from performance prediction in certain development stages, analytical modeling and simulation in design and coding phases to monitoring and measurements in the testing and correction phases. Much work has been done to create simple models that represent important characteristics of parallel programs, such as latency, network contention and communication overhead. But many of this methods still require substantial manual effort to represent entire applications in the format of the model [2, 5, 11]. Generating suitable models for performance estimation after every modification of the source code during the whole development phase is a time-consuming process and many application programmers are reluctant to invest the effort. Furthermore, it is often difficult for a programmer to predict the impact that system parameters will have on parallel code. As a result, software developers do not have the possibility of using an a priori performance prediction of different algorithm implementations in order to get an optimal solution. There are different approaches for performance modeling, e.g. statistical or experimental approaches like simulation, benchmarking or trace-driven experiments. In this paper, we consider an analytical modeling with runtime functions that are structured according to the computation and communication operations of the program. Analytical approaches can be used for data parallel and for mixed task and data parallel programs, also in the case of large and complicated application programs [11, 10]. Static prediction techniques for performance modeling of cluster systems and multicomputers are the only techniques that offer low prediction cost as well as the opportunity for symbolic analysis. Especially in view of the large system dimensions involved, low cost is essential to permit generic prediction technology in automatic compile-time optimization algorithms. The main obstacle to employ runtime functions for performance modeling of arbitrary parallel application programs is the resulting modeling overhead. The application programmer has to derive the runtime function from the source code of the parallel program manually and each modification of the application requires a modification of the runtime functions. This is a time-consuming work, especially for large and complicated applications and the programmer might be discouraged to make a design decision for a parallel application based on analytical performance modeling. In this article, we describe a compiler tool that generates a suitable runtime function for a parallel application automatically and so relieves the programmer from the effort to build a suitable performance model manually. The compiler has been realized with the SUIF system [7]. It can be used for runtime prediction of pure data as well as

*Supported by DFG (Deutsche Forschungsgemeinschaft).

for mixed task and data parallel applications [14]. Specific communication structures like orthogonal processor groups can also be considered [12].
The rest of the paper is structured as follows. Section 2 describes the modeling approach used by the compiler tool to estimate the communication overhead and the computation effort. Section 3 introduces the components and the workflow of the compiler tool. Section 4 describes the user and programmer interface of the compiler tool. Section 5 shows how the execution times of parallel application programs are modeled by the automated performance prediction tool. Section 6 discusses related work and Section 7 concludes the paper.

2 Modeling with runtime functions

For a given application, the source code analyzer yields a performance prediction based on a model for predicted execution time on distributed memory machines. In the following section we describe the modeling approach for execution times on which the compiler is based.

2.1 Modeling of MPI communication phases

An important issue of performance prediction of parallel applications is the modeling of the execution time of communication. Several machine parameters of the parallel architecture like latency, network contention and network bandwidth make it often difficult to handle an automated prediction tool for communication operations. It is possible to model the communication phases exclusively based on the characterizations of the parallel target machine, but such an approach may lead to inaccurate results. To improve the accuracy of the performance prediction with a minimum of programmer assistance we propose the use of predefined runtime functions for different communication operations. Runtime functions have been successfully used to model the execution time of communication operations for various communication libraries [9, 13]. Table 1 shows selected runtime functions for single to single and collective communication operations of the MPI library. The value b denotes the message size in bytes and p is the number of processors participating in the communication operation. For different variants, different coefficient values may have to be used which is identified by an additional parameter V for the variant in the following. The coefficients τ_1 and t_c can be considered as startup time and byte-transfer time. For a given platform, the values of the parameters in the functions are obtained by applying the least squares method to a set of measured execution times. Runtime functions for communication operations from various message passing libraries with appropriate coefficients are available for different clusters and MPPs [11, 5, 13].

runtime function
$t_{single}(b, V) = \tau_1(V) + t_c(V) \cdot b$
$t_{mbroad}(p, b, V) = \tau_1(V) + \tau_2(V) \cdot p + t_c(V) \cdot p \cdot b$
$t_{scatter}(p, b, V) = \tau_1(V) + \tau_2(V) \cdot p + t_c(V) \cdot p \cdot b$
$t_{gather}(p, b, V) = \tau_1(V) + \tau_2(V) \cdot p + t_c(V) \cdot p \cdot b$

Table 1. Runtime functions for selected single to single and MPI communication operations.

Each call of a single-to-single or collective communication operation is detected by the compiler tool and it is replaced by an appropriate call statement of the corresponding runtime function. The number of processors participating in the communication operation is determined from the communicator handle in the parameter list. The variable symbol denoting the number of processors is acquired from the appropriate procedure call statement *MPI_Comm_size()* with the marked communicator handle as argument in the program dependence graph. The specific coefficient values of the runtime functions have to be known for the target platform and are accessed in the data file *MachineProperties*.

2.2 Modeling of computation effort

Another crucial task for an automated performance prediction tool is the modeling of the computation effort of a parallel application program. The execution time of basic arithmetic, relational, logical or binary operations is determined by evaluation of measurements in isolation. The resulting values are stored in the file *MachineProperties*. This has to be done only once for each target machine and is performed automatically using an appropriate test program. Again, the programmer assistance is minimal and the approach has been successfully used to model the computation effort [9, 13].
Since the execution time of different arithmetic operations, like addition and division, can deviate substantially from each other, it can be useful to differentiate the performance modeling of these operations. The compiler tool offers the possibility to classify operations in order to capture different execution times for different arithmetic, relational, logical and binary operations. Another issue to improve the prediction accuracy is the identification of the result type of an expression, which is used to determine the data type obtained after executing an operation of that expression. We especially distinguish different numeric data types like integer and single and double precision floating point that are used as result types for unary or binary expressions in our modeling approach. Furthermore the object names used by the *SUIF IR* (*I*ntermediate *R*epresentation) provides information about the type of the variable access, i.e. whether it

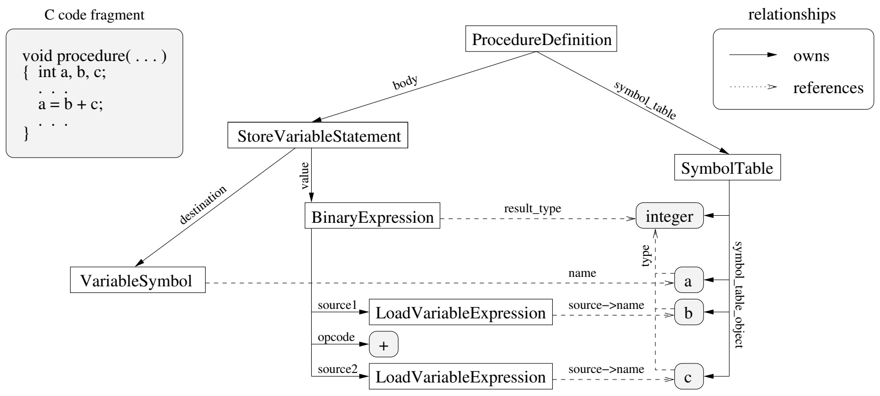

Figure 1. Illustration of *SUIF IR* objects and relationships between the objects to represent a C source code fragment. The definition of a procedure, comprising variable declarations and statements, is represented by an *ProcedureDefinition* object. The source code statement $a = b + c$ is represented by an *StoreVariableStatement* object.

is a memory read access or a memory write access. Thus, cost expressions based on adequate variables and values can be applied to predict the execution time of a specific arithmetic, relational, logical and binary operation. Figure 1 illustrates SUIF IR objects and relationships to represent a procedure definition. The C source code fragment of the procedure definition is shown in the upper left corner of the figure. The procedure contains a statement (*StoreVariableStatement*) to assign to a variable a the sum of the values of variables of b and c. The figure especially illustrates the crucial relationships between the objects, which are used to recognize characteristics of program components used by the SUIF IR. The modeling of memory access times needed to load or store operands for specific operations can be useful to achieve an accurate prediction of the computational effort [8].

3 Internal structure of the analyzer

This section gives an overview of the structure, the components and their functionality within the compiler tool realizing the source code analyzer.

3.1 Realization using the SUIF infrastructure

Our compiler tool for performance predictions has been implemented by using the SUIF system. The SUIF system is a compiler infrastructure based on a specific program representation, also called *SUIF* (**S**tanford **U**niversity **I**ntermediate **F**ormat). The emphasis is to maximize code reuse by providing useful abstractions and frameworks for developing new compiler passes and by providing an environment that allows compiler to inter-operate. The system is based on two fundamental concepts: an extensible program intermediate representation (IR) and a flexible module system. The system provides a set of predefined IR nodes, especially nodes that represent important program structures. In the SUIF infrastructure, various front end passes and back end passes are available for different programming languages, like *FORTRAN*, *C*, *Java* or *C++*, that are used to translate the specific source code into a corresponding SUIF IR. The analyzing process is performed on the SUIF IR independently from the input source code. In this article we especially consider C application programs as input to our compiler tool for performance prediction, but the design works similarly for the programming languages mentioned above.

3.2 Components of the compiler tool

The compiler tool consists of components, which are basically provided in terms of independent compiler passes. The compiler passes are used to gather information and to generate the runtime functions using the SUIF program representation, which has been derived from the source code.

3.2.1 Workflow of the compiler tool

Figure 2 shows the stages of the workflow for the automated performance analyzing and prediction tool. The parallel application program to be analyzed is represented by n source files *source_code_x.c* shown at the top of the figure. At first, each individual file of the application program is translated into an appropriate SUIF-file using the C front end pass of SUIF and then the resulting output files are linked together resulting in a single file. The link process is performed using the standalone compiler pass *link_suif* and the resulting

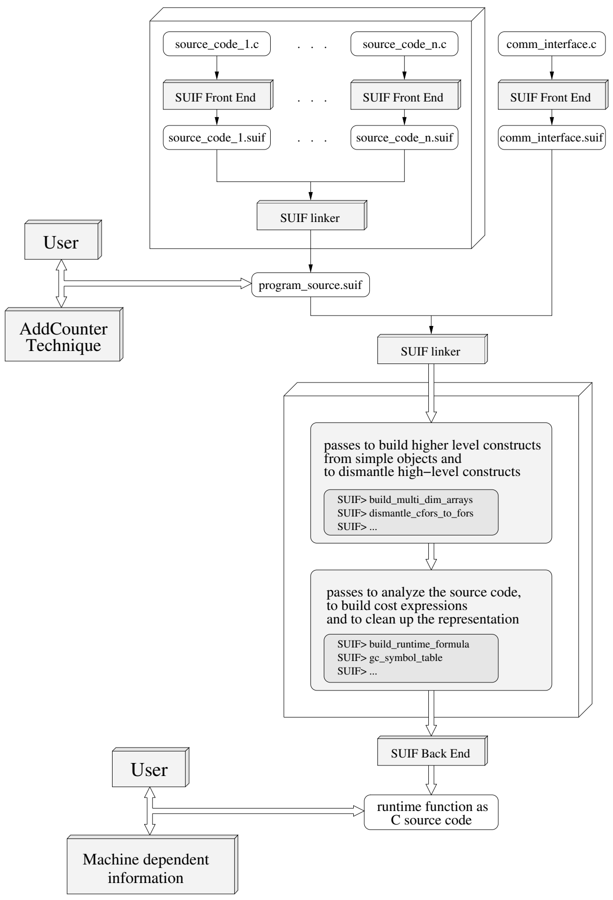
source_code_1.c
. . .
source_code_n.c
comm_interface.c
SUIF Front End
. . .
SUIF Front End
SUIF Front End
source_code_1.suif
. . .
source_code_n.suif
comm_interface.suif
SUIF linker
User
program_source.suif
AddCounter Technique
SUIF linker
passes to build higher level constructs from simple objects and to dismantle high−level constructs
SUIF> build_multi_dim_arrays
SUIF> dismantle_cfors_to_fors
SUIF> ...
passes to analyze the source code, to build cost expressions and to clean up the representation
SUIF> build_runtime_formula
SUIF> gc_symbol_table
SUIF> ...
SUIF Back End
User
runtime function as C source code
Machine dependent information

Figure 2. Illustration of the entire workflow of the source code analyzer. The actual source code analysis of the C-input file and the construction of cost expressions is performed by the compiler pass *build_runtime_formula***.**

file contains a hierarchical SUIF tree, which represents the whole parallel application program.

The file *comm_interface.c* shown in Figure 2 contains interfaces in form of procedure headers for runtime functions of MPI communication operations. The file is translated into the SUIF IR and then linked to the SUIF representation of the parallel application program. The interfaces in *comm_interface.c* are used to represent the procedure call statements of communication operations to obtain performance predictions for the communication overhead. The resulting SUIF IR file represents the whole application program including the interfaces for the runtime functions of various communication operations.

Other passes are used to analyze and annotate the SUIF IR in order to provide additional information. Furthermore, the SUIF environment contains a set of passes that transform the program representation. These are mainly passes to build higher level constructs from simple objects, to dismantle high-level construct, and to clean up the representation. Such transformation passes are performed in the source code analyzer to remove scope-statements, like variable declarations, and to move scope local variables to the procedure scope.

The programmer has the possibility to insert a unique label into the source code of the parallel application program in order to separate the procedure body into two parts, an initialization block and a computational block. This label denotes the end of the initialization block and the begin of the computational block. The advantage of this approach is to transfer the complete initialization block into the runtime function to be generated. By doing this, variable initializations or definitions are preserved for the resulting C program realizing the cost function. In the case that no label has been inserted by the user providing the input C source code to the compiler tool the complete procedure body is considered as computational block.

The SUIF representation obtained from the input C source code provides a procedure definition (*ProcedureDefinition*), that consists of a procedure name (*ProcedureSymbol*), a result type (*ProcedureType*), a list of parameters (*ParameterSymbols*), a local scope of variable declarations (*SymbolTable*) and the body of the procedure (*ExecutionObject*). The parameter list, the local variable declarations, potential definitions and the described initialization block are preserved for the runtime function to be generated. The name of the procedure is extended by a prefix to avoid collisions with the original procedure. Afterwards, several existing compiler passes are performed to avoid variable name collisions and to clean up the representation. The compiler tool recognizes the end of the initialization block and the begin of the computational block by the unique label mentioned above. The computational block is responsible for the main execution time and communication overhead of the procedure to be analyzed.

The computational block is analyzed by the compiler pass and gradually transformed into the runtime function such that each individual program construct is substituted by a corresponding expression representing the costs of this construct in the context of the parallel application program. The runtime function obtained is used in the same manner as the original procedure containing the application source code, but is executed on an arbitrary sequential machine. The value returned by the runtime function is the predicted execution time of the procedure in seconds obtained with the current parameter values. In this way the runtime estimations are evaluated and analyzed by varying the system size and number of participating processors. An advantage is the possibility to use the prediction values as input for further investigations, e.g. to visualize the performance behavior of different variants of the required application on the same

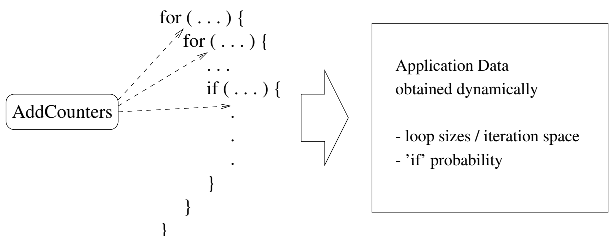

Figure 3. Dynamic instrumentation

target platform.

3.3 Estimation of conditionals and arbitrary loop structures

Many applications exhibit complex, data dependent execution behavior and have time varying resource demands. The key to successfully estimating the computation time is to capture the iteration space of nested loops and to perform an effective branch prediction for conditionals. In this section we propose an approach to estimate the probability of conditions and to determine the iteration space of more general loop structures.

At first, we present our approach to determine the probability of conditions. For the automated prediction tool to be able to work in a dynamic environment we introduce a source code instrumentation and dynamic analysis methodology. A precise method would require a user to input specific application information, which may consist of parameters like iteration spaces of loop structures or conditional statement probabilities to the compiler tool when an analysis is performed. While the user might correctly guess the information in small applications with few loops or conditional statements using small sized data, this task is much harder when large applications with complicated loop structures and input-dependent conditional statements are considered.

A solution to this problem is to use an approach, *AddCounter*, that automatically inserts special sensors into the application. The counters, as shown in Figure 3, reside inside fundamental statements like loops and conditional statements and are triggered when the application is executed for the first time. These counters gather runtime information and data for the user, and annotate them to the appropriate SUIF-objects representing the loop or condition statements in a trial program execution. The information which extends the SUIF IR of the application program as *AnnotableObject*s are used in the following compiler passes to determine complex iteration space of nested loop structures and conditional probabilities. The advantage of this method is that the compiler tool gathers information automatically and dynamically. The *AddCounter* technique introduced can be considered as an extension of the compiler tool to assist the user in performing prediction analysis for a wider range of parallel programs with more complicated or irregular program structures.

4 User and Programmer Interface

The modular structure of the tool allows the user of the compiler tool to adapt the structure of cost expressions and cost formulas which are used to estimate the runtime behavior of specific program constructs to the specific features of a target machine. If such an adaptation is required, it can be performed in a separate modeling step that precedes the actual program analysis. Especially the underlying model structure to predict the communication costs of message passing communication operation can be adapted independently by the user in an estimation phase. For the modeling of message passing communication operations runtime functions are used as described in Section 2.1. Each communication operation, like an *MPI_Bcast()* or *MPI_Allgather()* operation, is modeled by a predefined runtime function with known coefficients to predict the execution time of the operation. To allow a flexible modeling, the user can replace the predefined runtime function by a user-supplied function with the same signature: for a general communication operation, the user-supplied function has to provide the number n of transferred elements, the type of the elements and the number p of processors participating in the communication operation as parameters. For single transfer operations only two parameters are required, since just two processors are involved in the communication operation. The predefined runtime functions can be replaced by the user at any time, since it is independent of the resulting prediction function, which is used to estimate the execution time of the entire application program. The form of cost expressions to estimate the computation effort of program constructs is determined in specific methods of the compiler tool. A modification of the cost format can be obtained by providing a method with the same signature as the original method. The execution times of various computation operations and library calls are represented as specific variables; for instance T_{add_int} denotes the execution time of an *addition* operation with return type *integer*. The values of the variables are provided in a specific data file and can be modified easily at any time. It is even possible that the user provides methods for a sophisticated modeling of memory accesses, for example, a mechanism like the cache miss equations [6]. An adaption of machine-dependent information may be necessary when hardware equipment or software environments are updated for a known target machine.

5 Applications and experimental results

In order to demonstrate the usage of the performance prediction toolkit we consider parallel implementations of an iterative solution method for linear equation systems, the Jacobi iteration.
We consider the applications on three different target machines a Beowulf-Cluster, a dual Xeon cluster and an IBM Regatta p690 cluster. The Beowulf Cluster CLiC ('**C**hemnitzer **Li**nux **C**luster') is build up of 528 Pentium III processors clocked at 800 MHz. The processors are connected by a fast-Ethernet communication network. The Xeon cluster is built up of 16 nodes and each node consists of two Xeon processors clocked at 2 GHz. The nodes are connected by a high performance interconnection network based on Dolphin SCI interface cards. The SCI network is connected as 2-dimensional torus topology and is used by the ScaMPI (SCALI MPI) [1] library. The IBM p690 Regatta cluster is built up of 6 nodes and each node consists of 32 PowerP4+ processors clocked at 1.7 Ghz. We consider the well-known Jacobi iteration for solving a linear system of equations $A \cdot x = b$ with an $n \times n$ matrix A, a vector $b \in \mathbf{R}^n$, and an unknown vector x to be determined. There are different ways to implement the Jacobi iteration in a data parallel way depending on the data distribution of the matrix A. We consider a row-wise distribution and a column-wise distribution. The computational work for computing the new entries of the next iteration $x^{(k)}$ is the same in both cases and is equally distributed over the processors. In each iteration each processor performs $\lceil \frac{n}{p} \rceil \times n$ multiplications and about the same number of additions. But because each processor computes different parts and each processor needs the entire new iteration vector $x^{(k)}$ in the next iteration step, different communication operations are required for the implementations. In the row-wise distribution of matrix A each processor computes $\lceil \frac{n}{p} \rceil$ scalar products yielding $\lceil \frac{n}{p} \rceil$ components of the new iteration vector. To provide the entire vector to each processor for the next step a multi-broadcast operation *(*MPI_Allgather()) is performed.
From the program code, the compiler tool generates the following runtime function to represent the execution time of the row-wise Jacobi iteration with matrix size respective system size n and p participating processors.

$$\begin{aligned} T_{row}(p,n) &= 2 \cdot \frac{n}{p} \cdot t_{op} \cdot (2 \cdot (n-1) \cdot t_{op}) \\ &+ 3 \cdot \frac{n}{p} \cdot t_{op} + T_{mb}(p, \frac{n}{p}). \end{aligned}$$

The function has been simplified so that t_{op} denotes the time for the execution of an arbitrary arithmetic operation and T_{mb} denotes the runtime function of the multi-broadcast operation. The execution time t_{op} generally represents an equation of the form

$$t_{op} = 2 \cdot t_{read} + t_{write} + t(op, dt)$$

where t_{read} and t_{write} denote the time of a memory read access and a memory write access and $t(op, dt)$ denotes the execution time of the arithmetic operation op with result type dt. The body of the innermost loop contains a memory write access and two arithmetic operations to compute the local components of x. To complete the calculation each processor computes $\lceil \frac{n}{p} \rceil$ components of the new iteration vector x. The for-loop used contains 3 arithmetic operations. The multi-broadcast operation performed to provide the entire vector to each processor transfers $\lceil \frac{n}{p} \rceil$ elements of data type *double*.
In the column-wise distribution of matrix A each processor computes a new vector d of size n. Adding up all those vectors gives the new iteration vector x. Since the vectors d are located in different address spaces, collective communication is required to perform the addition. There are two possibilities, a multi-broadcast and a single-broadcast variant.
The multi-broadcast variant uses a MPI_Allreduce and MPI_Allgather operation. The compiler tool generates the following simplified function to represent the execution time

$$\begin{aligned} T_{col\_mb}(p,n) &= 2 \cdot \frac{n}{p} \cdot t_{op} \cdot (2 \cdot (n-1) \cdot t_{op}) \\ &+ 3 \cdot \frac{n}{p} \cdot t_{op} + T_{macc}(p,n) + T_{mb}(p, \frac{n}{p}) \end{aligned}$$

where T_{macc} denotes the time of an MPI_Allreduce operation.
The single-broadcast variant uses a single-accumulation operation and a single-broadcast operation resulting in the following simplified runtime function

$$\begin{aligned} T_{col\_sb}(p,n) &= 2 \cdot \frac{n}{p} \cdot t_{op} \cdot (2 \cdot (n-1) \cdot t_{op}) \\ &+ 2 \cdot n \cdot t_{op} + T_{acc}(p,n) + T_{sb}(p,n). \end{aligned}$$

T_{acc} denotes the runtime function of a single-accumulation operation (*MPI_Reduce*) and T_{sb} denotes the runtime function of a single-broadcast operation (*MPI_Bcast*). Again, the runtime function is obtained automatically. The difference between both variants lies in the computation of an intermediate result for the iteration vector x. The multi-broadcast variant provides the vector to each processor such that only $\frac{n}{p}$ components are computed concurrently $(3 \cdot \frac{n}{p})$, whereas the single-broadcast variant reduces the intermediate result to root processor P_0 such that a single processor computes the entire vector $(2 \cdot n)$. Afterwards, the processor P_0 uses a single-broadcast operation to provide the entire vector x to each processor for the next iteration step.
We consider the quality of the runtime prediction based on the automated performance prediction tool. Figure 4 and Figure 5 show the measured and predicted runtime for adequate values of varying matrix sizes and number of processors on the CLiC, on a dual Xeon cluster and on an IBM p690 cluster. The figures illustrate that the predictions fit the measurements quite accurately for different target machines. The deviations between the predicted and measured runtime lies below 7% for the most cases on the CLiC and

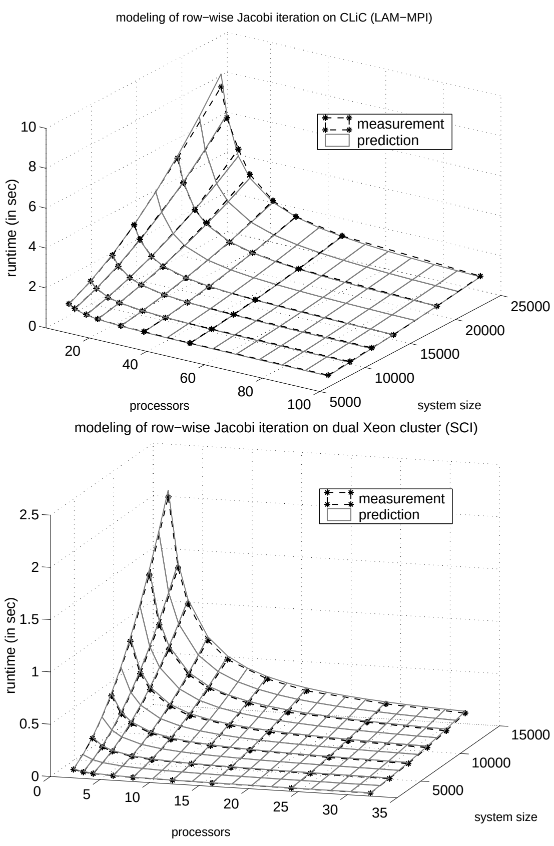

Figure 4. Modeling of row-wise realization of Jacobi iteration on CLiC using LAM-MPI (top) and on dual Xeon cluster using ScaMPI (bottom).

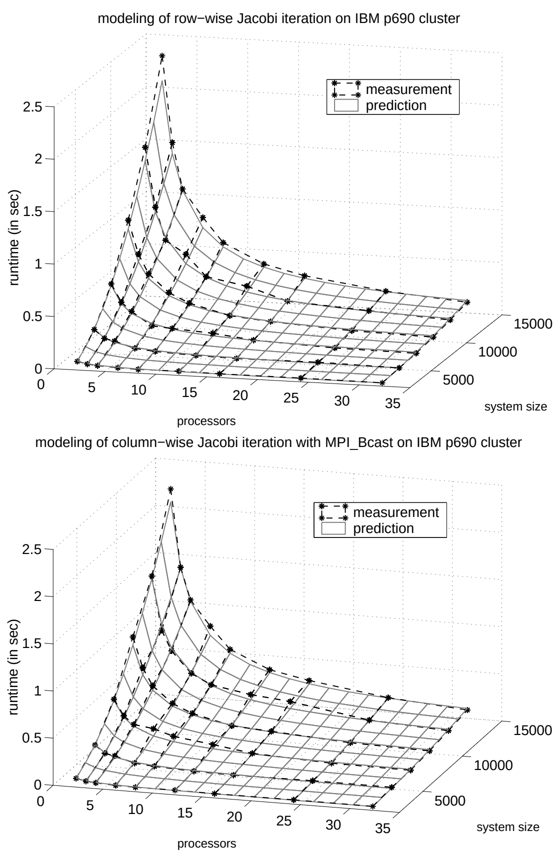

Figure 5. Modeling of row-wise (top) and column-wise (bottom) realization of Jacobi iteration on IBM p690 Regatta cluster.

below 5% on the dual Xeon cluster and the IBM p690 cluster.

6 Related Work

There are several analytical approaches to predict the execution behavior of parallel program performance, e.g. the `Falcon` [4] project of the Georgia Institute of technology, the `Pablo` [3] project of the university of Illinois and the `PAMELA` [16] project.The major conceptual components of `Falcon` are a monitoring specification mechanism, which consists of a low-level sensor specification language and a high level view specification language and mechanisms for on-line information collection, analysis, and for program steering. The primary objective of the tool set is the evaluation of monitoring information to provide a graphical user interface for instrumentation and runtime manipulation. In contrast to the compiler tool introduced in this paper no analytical investigation and evaluation based on the source code of the parallel applications are supported.

The ongoing goal of the `Pablo` project is the development of a portable performance data analysis environment that can be used with a variety of massively parallel systems. The `Pablo` software environment contains both a portable performance data analysis environment and a portable source code instrumentation component that are united via a portable data meta-format called SDDF. This performance file, stored in the `Pablo` Self-Defining Data Format (SDDF), is input to *SvPablo*, which presents the performance data in the context of the original source code. An advantage of the toolkit introduced in this paper lies in the fact that the resulting runtime functions are available in the independent SUIF IR and can be used in a more open and more interdisciplinary manner.
The `PAMELA` project aims at the development of a modeling methodology that yields fast, parameterized performance models of parallel programs running on shared-memory as well as distributed-memory machines. The project uses similar methodologies and approaches to obtain and provide a performance prediction tool as our compiler tool, but ap-

plies a specific performance simulation language to achieve the objectives.

7 Conclusion

In this paper we have illustrated an approach for automated prediction and modeling of parallel application programs. We have presented how the communication and computation times are modeled using runtime functions. The use of static compiler front-end tools and runtime information in the form of counters in the sense of sensors is an useful extension for automated performance modeling, especially for applications containing complex loop structures. The article shows how the compiler technique is used to generate correct and efficient code to model the execution time of application programs. We have considered parallel application programs with multiple communication phases and complex computational structures and the results show that programs with multiple communication pattern can be automatically analyzed and modeled. In particular, our predictions are within approximately 7% of measured execution times for the parallel Jacobi implementation. Furthermore, the models are useful for obtaining scalability metrics on different network environments.
Another advantage of using runtime formulas is that optimization methods can be applied for designing efficient implementations which minimize the communication costs and the load imbalance. The approach is used to obtain an a priori estimation of the prospective gain of a parallel implementation.

References

[1] Scali / ScaMPI commercial MPI on SCI implementation. http://www.scali.com/.

[2] D.E. Culler, R. Karp, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. *Proc. of 4th Symp. on Principles and Practice of Parallel Programming*, 28(4):1–12, 1993.

[3] L. DeRose and D. A. Reed. SvPablo: A Multi-Language Architecture-Independent Performance Analysis System. In *Proc. of the International Conference on Parallel Processing (ICPP)*, September 1999.

[4] G. Eisenhauer, W. Gu, K. Schwan, and N. Mallavarupu. Falcon – Toward Interactive Parallel Programs: The On-line Steering of a Molecular Dynamics Application. In *Proc. of the Third International Symposium on High-Performance Distributed Computing (HPDC-3)*, August 1994.

[5] R. Foschia, T. Rauber, and G. Rünger. Modeling the Communication Behavior of the Intel Paragon. In *Proc. of 5th Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'97)*, IEEE, pages 117–124, 1997.

[6] S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[7] The Stanford SUIF Compiler Group. http://suif.stanford.edu/.

[8] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, 2003.

[9] K. Hwang, Z. Xu, and M. Arakawa. Benchmark Evaluation of the IBM SP2 for Parallel Signal Processing. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):522–536, 1996.

[10] D. Kerbyson, H. Alme, A. Hoisie, F. Petrini, H. Wasserman, and M. Gittings. Predictive Performance and Scalability Modeling of a Large-Scale Application. In *Proc. of IEEE/ACM SC2001*, 2001.

[11] M. Kühnemann, T. Rauber, and G. Rünger. Performance Modelling for Task-Parallel Programs. In *Proc. of Communication Networks and Distributed Systems Modeling and Simulation (CNDS'02)*, pages 148–154, 2002.

[12] T. Rauber, R. Reilein, and G. Rünger. ORT – A Communication Library for Orthogonal Processor Groups. In *Proc. of the Supercomputing 2001 (CD-ROM)*, Denver, USA, 2001. IEEE Press.

[13] T. Rauber and G. Rünger. PVM and MPI Communication Operations on the IBM SP2: Modeling and Comparison. In *Proc. 11th Symp. on High Performance Computing Systems (HPCS'97)*, 1997.

[14] T. Rauber and G. Rünger. Library Support for Hierarchical Multi-Processor Tasks. In *Proc. of the Supercomputing 2002*, Baltimore, USA, 2002.

[15] C. U. Smith. *Performance Engineering of Software Systems*. Addision Wesley, 1989.

[16] A.J.C. van Gemund. Symbolic Performance Modeling of Parallel Systems. In *Proc. of IEEE Transactions on Parallel and Distributed Systems*, volume 14, No. 2, pages 154–165, Feb 2003.