SCALA: A Framework for Performance Evaluation of Scalable Computing

Xian-He Sun[1], Mario Pantano[2], Thomas Fahringer[3], and Zhaohua Zhan[1]

[1] Department of Computer Science, Louisiana State University, LA 70803-4020
{sun,zzhan}@csc.lsu.edu

[2] Department of Computer Science, University of Illinois, Urbana, IL 61801
pantano@cs.uiuc.edu

[3] Institute for Software Technology and Parallel Systems
University of Vienna, Liechtensteinstr. 22 1090, Vienna, Austria
tf@par.univie.ac.at

Abstract. Conventional performance environments are based on profiling and event instrumentation. It becomes problematic as parallel systems scale to hundreds of nodes and beyond. A framework of developing an integrated performance modeling and prediction system, SCALability Analyzer (SCALA), is presented in this study. In contrast to existing performance tools, the program performance model generated by SCALA is based on scalability analysis. SCALA assumes the availability of modern compiler technology, adopts statistical methodologies, and has the support of browser interface. These technologies, together with a new approach of scalability analysis, enable SCALA to provide the user with a higher and more intuitive level of performance analysis. A prototype SCALA system has been implemented. Initial experimental results show that SCALA is unique in its ability of revealing the scaling properties of a computing system.

1 Introduction

Although rapid advances in highly scalable multiprocessing systems are bringing petaflops performance within grasp, software infrastructure for massive parallelism simply has not kept pace. Modern compiler technology has been developed to reduce the burden of parallel programming through automatic program restructuring. Among others, notable examples include the Vienna HPF+ compiler [2] and the Fortran D compilation system [6]. There are many ways to parallelize an application, however, and the relative performance of different parallelizations vary with problem size and system ensemble size. Parallelization of programs is far from being fully automated optimization. Predicting performance variation and connecting dynamic behavior to original source code are two major challenges facing researchers in the field [1]. Emerging architectures such as petaflop computers and next generation networks demand sophisticated performance environments to provide the user with useful information on the scaling behavior and to identify possible improvements.

In this study we introduce the framework and initial implementation of the SCALability Analyzer (SCALA) system, a performance system designed for scalable computers. The goal of SCALA is to provide an integrated, concrete and robust performance system for performance modeling and prediction of real value to the high performance computing community. Distinguished from other existing performance tools, the program performance model generated by SCALA is based on scalability analysis [15, 19]. SCALA is a system which correlates static, dynamic and symbolic information to reveal the scaling properties of parallel programs and machines and for predicting their performance in a scalable computing environment. SCALA serves three purposes: predict performance, support performance debugging and program restructuring, and estimate the influence of hardware variations. Symbolic and scalability analysis are integrated into advanced compiler systems to explore knowledge from the compiler and to guide the program restructuring process for optimal performance. A graphical user interface is developed to visualize the performance variations of different program implementations with different hardware resources. Users can choose the performance metrics and parameters they are interested in.

SCALA supports range comparison [15], a new concept which allows the performance of different code-machine combinations to be compared over a range of system and problem sizes. SCALA is an experimental system which is designed to explore the plausibility and credibility of new techniques, and to use them collectively to bring performance analysis environments to their most advanced level. In this paper, we present the design and initial implementation of SCALA. Section 2 presents the structure and primary components of SCALA. Section 3 and 4 gives a short description of scalability analysis and data analysis respectively. Implementation results are reported in Section 5. Finally, Section 6 gives the summary and conclusion.

2 The Design of SCALA

The design of SCALA is based on the integration of symbolic cost models and dynamic metrics collected at run-time. The resulting code-machine model is then analyzed to predict performance for different software (i.e. program parameters) and architecture characteristics variations. To accomplice this process, SCALA combines performance prediction techniques, data analysis and scalability analysis with modern compiler techniques. An important feature of SCALA is its capability to relate performance information back to the source code in order to guide the user and the compiler in the selection of transformation and optimization strategies. Although SCALA is an integrated system designed to predict the scaling behavior of parallel applications, it supports standard performance information via data analysis and visualization techniques and can be used without compiler support.

The general structure of SCALA comprises several modules which combined together provide a robust environment for advanced performance analysis. SCALA has three input sources of informations (compiler, measurement system, and user) and two outputs (compiler and user). The compiler provides symbolic models of the program and information on the code regions measured. Dynamic performance information collected are provided by the measurement system in terms of tracefiles. The user can also be involved in the process supplying specific information on software parameters and on the characteristics of the input data. The three input sources can be used collectively for the best result or can be used separately based on physical constraints and target applications. On the output side, performance indices can be given directly to the compiler annotating syntax tree and call graph of the application so that the compiler can automatically select the most appropriate transformations to be applied to each code region. Detailed information on the predicted behavioral characteristics of the application will be given to the user by employing visualization techniques. Figure 1 depicts the design structure of SCALA which is composed of the following modules:

- *Data Management:* This module implements data filtering and reduction techniques for extrapolating detailed information on each code region measured. The input of this module is a tracefile in a specific format while its output is given to the analysis modules and to the interface for visualization.
- *Statistical Analysis:* The statistical component of SCALA determines code and/or machine effect, finds the correlation between different program phases, identifies the scaling behavior of "critical-segments", and provides statistical data for the user interface.
- *Symbolic Analysis:* The symbolic module gathers cost models for alternative code variants computed at compile-time. Furthermore, by employing dynamic data, symbolic analysis evaluates the resulting expressions for specific code-machine combinationes.
- *Scalability Analysis:* The scalability analysis module is the most important part of SCALA. The development of this module is based on highly sophisticated analytical results on scalability analysis as described in Section 3. The scalability module implements novel algorithms for predicting performance in terms of execution time and scalability of a code-machine combination.
- *Model Generator and Database:* The performance model generator automatically determines the model of system execution. The database stores previously measured information which will be used to find appropriate values of symbolic constants and statistic data for performance prediction.
- *Interface:* The measured and predicted results can be visualized via an user-friendly graphical interface. The appearance of the interface can be justified by the user and the visualization can be "zoomed" in and out for a specific computing phase and component. The results also can be propagated through the interface to the compiler for automatic optimization purposes.

While each module of SCALA has its specific design goal, they are closely interconnected for a cooperative analysis process that provides the user advanced performance information for a thorough investigation of the application itself as well as of the interaction with the underlying parallel machine. The other important point in this design is the complete integration of SCALA in a restructuring

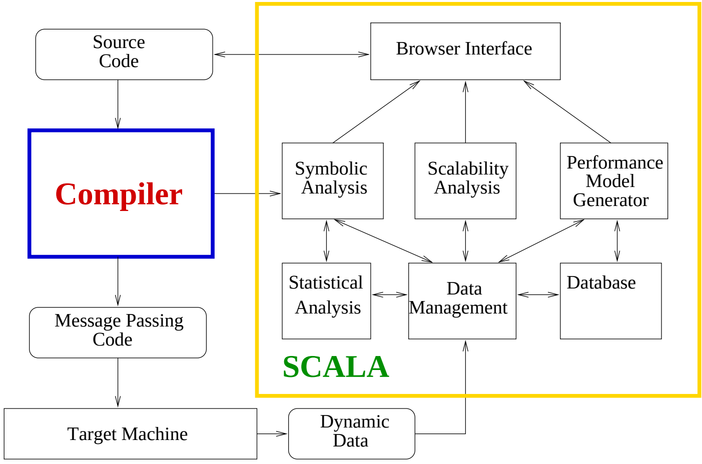

Fig. 1. Structure of SCALA.

compiler. This integration allows the tool to directly gather detailed information on the source code being transformed. As a consequence, performance predictions can be linked back to the user source code providing suggestions for code optimizations. For the sake of brevity, only a brief discussion of the scalability analysis and data analysis modules is given in the following sections. The implementation of these modules, as well as others, can be found in Section 5 where the implementation issues are addressed.

3 Scalability Analysis

Several scalability metrics and prediction methodologies have been proposed [7, 9, 13, 8]. The SCALA's approach is based on *isospeed* scalability [19]: A code-machine combination (CMC) is scalable if the achieved average unit speed can remain constant when the number of processors scales up, provided the problem size can be increased with the system size.

The performance index considered in isospeed scalability, therefore, is *speed* which is defined as work divided by time. *Average unit speed* (or, average speed, in short) is the achieved speed of the given computing system divided by p, the number of processors. The speed represents a quantity that ideally would increase linearly with the system size. Average speed is a measure of efficiency of the the underlying computing system.

For a large class of CMCs, the average speed can be maintained by increasing the problem size. The necessary problem size increase varies with code-machine combinations. This variation provides a quantitative measurement of scalability. Let W be the amount of work of a code when p processors are employed in a machine, and let $W^{'}$ be the amount of work of the code when $p' > p$ processors are employed to maintain the average speed, then the scalability from system size p to system size p' of the code-machine combination is:

$$\psi(p, p') = \frac{p' \cdot W}{p \cdot W'} \tag{1}$$

Where the work W' is determined by the isospeed constraint.

In addition to measuring and computing scalability, the prediction of scalability and the relation between scalability and execution time have been well studied. Theoretical and experimental results show that scalability combined with initial execution time can provide good performance prediction, in terms of execution times.

New concepts of crossing-point analysis and range comparison are introduced. Crossing-point analysis finds fast/slow performance crossing points of parallel programs and machines. In contrast with execution time which is measured for a particular pair of problem and system size, range comparison compares performance over a wide range of ensemble and problem size via scalability and crossing-point analysis. Only two most relevant theoretical results are given here. More results can be found in [14–16].

Result 1: *If a code-machine combination is faster at the initial state and has a better scalability than that of other code-machine combinations, then it will remain superior over the scalable range.*

Range comparison becomes more challenging when the initial faster CMC has a smaller scalability. When the system ensemble size scales up, an originally faster code with smaller scalability can become slower than a code that has a better scalability. Finding the fast/slow crossing point is critical for achieving optimal performance. Definition 1 gives an informal definition of crossing point based on the isospeed scalability. The fomral definition can be found in [15, 16].

Definition 1. *(*scaled crossing point*) Let code-machine combination 1 have execution time t, scalability $\Phi(p, p')$, and scaled problem size W'. Let code-machine combination 2 have execution time T, scalability $\Psi(p, p')$, and scaled problem size W^*. If $t_p(W) = \alpha T_p(W)$ at the initial ensemble size p and problem size W for some $\alpha >$, then p' is a crossing point of code-machine combinations 1 and 2 if and only if*

$$\frac{\Phi(p, p')}{\Psi(p, p')} > \alpha. \tag{2}$$

In fact, as given by [15, 16], when $\Phi(p, p') > \alpha\Psi(p, p')$ we have $t_{p'}(W') < T_{p'}(W^*)$. Notice that since $\alpha > 1$ combination 2 has a smaller execution time at the initial state $t_p(W) > T_p(W)$. This fast/slow changing in execution time gives the meaning of crossing point.

Result 2: *Assume code-machine combination 1 has a larger execution time than code-machine combination 2 at the initial state, then the scaled ensemble size p' is not a scaled crossing point if and only if combination 1 has a larger execution time than that of combination 2 for solving any scaled problem $W^\dagger$*

such that $W^\dagger$ is between W' and W^* at p', where W' and W^* is the scaled problem size of combination 1 and combination 2 respectively.

Result 2 gives the necessary condition for range comparison of scalable computing: p' is not a crossing point of p if and only if the fast/slow relation of the codes does not change for any scaled problem size within the scalable range of the two compared code-machine combinations. Based on this theoretical finding, with the comparison of scalability, we can predict the relative performance of codes over a range of problem sizes and machine sizes. This special property of scalability comparison is practically valuable. Restructuring compilation, or programming optimization in general, in a large sense is to compare different programming options and choses the best option available. Result 2 provides a foundation for the database component of SCALA. Figure 2 gives the range comparison algorithm in terms of finding the smallest scaled crossing point via scalability comparison. While being not listed here, an alternative range comparison algorithm that finds the smallest scaled crossing point via execution time comparison can be found in [15]. In general, there could have more than one scaled crossing point over the consideration range for a given pair of CMCs. These two algorithms can be used iteratively to find successive scaled crossing points.

`Assumption of the Algorithm:` Assume code-machine combinations 1 and 2 have execution time $t(p, W)$ and $T(p, W)$ respectively, and $t(p, W) = \alpha T(p, W)$ at the initial state, where $\alpha > 1$.
`Objective of the Algorithm:` Find the superior range of combination 2 starting at the ensemble size p

Range Comparison
Begin
  $p' = p$;
Repeat
    $p' = p' + 1$;
    Compute the scalability of combination 1 $\Phi(p, p')$;
    Compute the scalability of combination 2 $\Psi(p, p')$;
Until($\Phi(p, p') > \alpha\Psi(p, p')$ **or** p' = the limit of ensemble size)
If $\Phi(p, p') > \alpha\Psi(p, p')$ **then**
    p' is the smallest scaled crossing point;
    Combination 2 is superior at any ensemble size $p^\dagger$, $p \le p^\dagger < p'$;
Else
    Combination 2 is superior at any ensemble size $p^\dagger$, $p \le p^\dagger \le p'$
End{If}
End{Range Comparison }

Fig. 2. Range Comparison Via Crossing Point Analysis

4 Data Management and Statistical Analysis

The dynamic properties of an application can be investigated by measuring the run time of the code and the interactions with the underlying parallel system. Tracing the occurring run time events can lead to large amount of performance information which needs to be filtered and reduced to provide the users with a compact and easy-to-interpret set of indices describing the characteristics of the application. The level of detail of this set depends mainly on the scope of the performance analysis. As an example, if the communication behavior of the application is under investigation, a performance analysis tool should be able to extrapolate from measured data only the most significant information regarding the communication activities such as total time spent transmitting a message, amount of data transmitted, and communication protocols used in each transmission. For modeling purpose, a performance analysis system must analyze the data and combine dynamic indices with symbolic models.

SCALA is an advanced tool that combines both static performance information and measurement/analysis to investigate the scaling behavior of an application varying architecture characteristics and application parameters. The data collected by the measurement system need to be analyzed and reduced so that they can be sufficiently detailed for evaluating the static model generated at compile time. This evaluation is the base for applying scalability analysis techniques and predicting the scaling behavior of the code-machine combination. It is important to note that this prediction need not to be quantitatively exact, only qualitatively with a magnitude of errors as high as 10-20 percent.

The *data management* module of SCALA implements several data-reduction techniques that can be applied in isolation or in combination for data management. Filtering is the simplest form of data reduction to eliminate data that do not meet specified performance analysis criteria retaining only the pertinent characteristics of performance data. Among the range of statistical techniques that can be applied to a data set, mean, standard deviation, percentiles, minimum, and maximum are the most common and provide a preliminary insight on the application performance, while reducing the amount of data. Other statistical methods such as the analysis of distribution and variability provide more detailed information on the dynamic behavior of parallel program. For example, the analysis of distribution gives information on how the communication and computation are distributed across processors and the user or compiler can apply transformations to improve the performance. Moreover, the coefficients of variation of communication time and computation time are good metrics to express the "goodness" of work and communication distributions across processors. In most cases, the data measured need to be scaled in a common interval so that further statistical techniques can be successfully applied. Timing indices also can be scaled in a more significant metric for the analysis such as from seconds to microseconds or nanoseconds.

All these statistics can be applied to the entire data set as well as to a subset of data such as a reduced number of processors or sliding windows of the most recent data. However, to manage large amount of data and large number of performance indices, advanced statistical techniques need to be applied. Clustering is one of the most common techniques used to reduce the amount of performance data [3, 12] and consequently reduce the number of performance indices needed to identify sources of performance losses. Statistical clustering is a multidimensional technique useful to automatically identify equivalence classes. For example, clustering classifies processors such that all processors in one cluster have similar behavior.

Unlike the data management module, the statistical analysis module is designed to handle more sophisticate statistical tasks. For instance, some computing phases are more sensitive than others. The performance of some program segments or run-time parameters may change dramatically at some hardware/software thresholds. In addition to conventional statistical methods such as synthetic perturbation screening and curve fitting [10], new methods are also introduced to SCALA. Noticeably, based on factorial analysis, we have proposed a methodology for examining latency hiding techniques of hierarchical memory systems [17]. This methodology consists of four levels of evaluation. For a set of codes and a set of machines, we first determine the effect of code, machine, and code-machine interaction on performance respectively. If a main effect exists, then, in the second level of evaluation, the code and/or machine is classified based on certain criteria to determine the cause of the effect. The first two levels of evaluation are based on average performances over the ranges of problem size of current interest. The last two levels of evaluation determine the performance variation when problem sizes scale up. Level three evaluation is the scalability evaluation of the underlying memory system for a given code. Level four evaluation conducts detailed examination on component contributions toward the performance variation. The combination of the four levels of evaluation makes the proposed methodology adaptive and more appropriate for advanced memory systems than existing methodologies.

Complicated with architectural techniques and speed and capacity of different storage devices, how to measure and compare the performance variation of modern memory systems remains a research issue. The statistical method is introduced to provide a relative comparison for memory systems. It can be extended to other contexts where the scalability results given in Section 3 are not feasible.

5 Prototype Implementation

The implementation of SCALA involves several steps, which include the development of each component module, integration of these components, testing, modification and enhancement of each component as well as of the integrated system. As a collective effort, we have conducted the implementation of each component concurrently and have successfully tested a prototype of the SCALA system.

5.1 Integrated Range Comparison for Restructuring Compilation

Restructuring a program can be seen as an iterative process in which a parallel program is transformed at each iteration. The performance of the current parallel program is analyzed and predicted at each iteration. Then, based on the performance result, the next restructuring transformation is selected for improving the performance of the current parallel program. Integrating performance analysis with a restructuring system is critical to support automatic performance tuning in the iterative restructuring process. As a collaborative effort between Louisiana State University and University of Vienna, a first prototype of SCALA has been developed and tested under the *VFCS* as an integrated performance tool for revealing scaling behavior of simple and structured codes [18]. In this framework, the static performance data have been provided by P^3T [5], a performance estimator that assists users in performance tuning of programs at compile time. The integration of P^3T into VFCS enables the user to exploit considerable knowledge about the compiler's program structural analysis information and provides performance estimates for code restructuring. The SCALA system has been integrated to predict the scaling behavior of the program by using static information provided by P^3T and the dynamic performance information collected at run-time. In particular, the Data Management Module of SCALA, filters the raw performance data obtained, executing the code on a parallel machine and provides selected metrics which, combined with the static models of P^3T allows an accurate scalability prediction of the algorithm-machine combination. Naturally, the interactions between SCALA and the estimator P^3T is an interactive process where at each iteration the scalability analysis model is refined to converge at the most precise prediction. For this purpose we have designed and implemented within SCALA an iterative algorithm to automatically predict the scalability of code-machine combinations and compare the performance over a range of algorithm implementation (range comparison). As a result, the automatic range comparison is computed within a data-parallel compilation system. Figure 3 shows the predicted crossing point for solving a scientific application with two different data-distributions in VFCS environment on an Intel iPSC/860 machine. We can see that the column distribution becomes superior when the number of processors is greater than or equal to 8. The superiority of 2-D block-block distribution ends when the system size equals 4. The prediction has been confirmed by experimental testing.

In order to verify Result 2 we measured both codes with $n = 33$ and $n = 50$, respectively. In accordance with Result 2 before $p = 8$ there is no performance crossing point, and $p = 8$ may correspond to a crossing point for a given problem size in the scalable range. The results are shown in Figure 3.b. Since the results obtained with this implementation demonstrate the feasibility and high potential of range comparison in performance evaluation, we extended SCALA for its integration in the newly developed Vienna High Performance Fortran Compiler (VFC) [2]. VFC is a source-to-source compilation system from HPF codes to explicit message passing codes. The parallelization strategies of VFC are very gen-

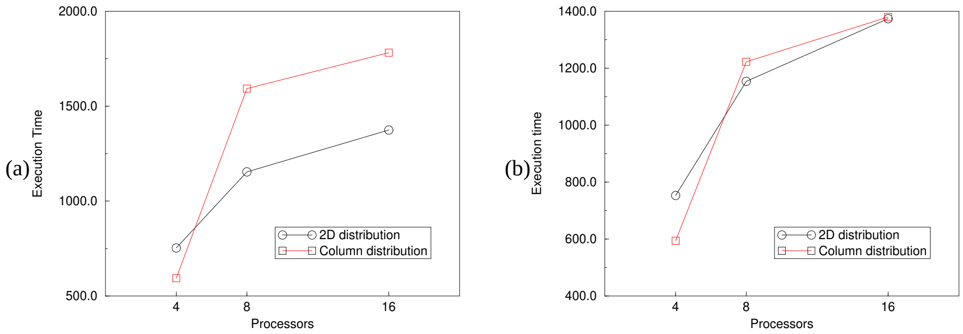

Fig. 3. Scaled crossing point (a) and crossing point (b) for the Jacobi with n=20

eral and applicable to programs with dynamically allocated or distributed arrays. VFC combines compile-time parallelization strategies with run-time analysis.

As a first step for investigating the performance of parallelized codes, we embedded in the VFC the SCALA Instrumentation System (SIS). SIS [4] is a tool that allows the automatic instrumentation, via command line options, of various code regions such as subroutines, loops, independent loops and any arbitrary sequence of executable statements. SIS has been extensively tested in connection with the Medea tool [3] for investigating the performance of applications with irregular data accesses. SCALA is being integrated within new compilation framework. The data management module accepts tracefiles obtained executing the code instrumented with SIS and computes a set of statistical metrics for each code region measured. Here, P^3T is substituted by a new tool that will provide SCALA with symbolic expressions of the code regions. The value of the parameters used in the symbolic expression, however, may vary when system or problem size increase. For this purpose in SCALA we compute and save run-time information in SCALA database (see Figure 1) for specific code-machine combinations. For example, on Cray T3E and QSW (Quadrics Supercomputers World) scalable computing system, the communication cost is given as $T_{comm} = \rho + \beta \cdot D$. D is the length of a single message, ρ, β are values describing the startup time and the time required for the transmission of a single byte. These machine dependent values are either supplied by the manufactures or can be experimentally measured. In order to reproduce the interference between the messages we compute ρ and β for each specific communication pattern used in the application. Table 1 presents the communication values computed for the communication patterns 1-D and 2-D Torus used in the Jacobi implementation. As shown in Table 2, SCALA accurately predicts the scalability of the algorithm and therefore range comparison. More detailed information can be found in [11].

Table 1. Communication Models (in μs)

	Cray T3E				QWS CS-2			
	1-d		2-d		1-d		2-d	
	short	long	short	long	short	long	short	long
ρ	35.6	71.2	40.3	77.5	220.3	297.2	150.4	192.9
β	$2.0 \cdot 10^{-2}$	$12.5 \cdot 10^{-3}$	$2.1 \cdot 10^{-2}$	$14.1 \cdot 10^{-3}$	$5.5 \cdot 10^{-2}$	$3.5 \cdot 10^{-2}$	$6.7 \cdot 10^{-2}$	$4.4 \cdot 10^{-2}$

Table 2. Bidirectional 2-d torus on CRAY T3E: Predicted (P) and measured (M) scalability

	Jacobi 2-d											
$\psi(p,p')$	$p' = 8,\ n = 1519$			$p' = 16,\ N = 2186$			$p' = 32,\ N = 3124$			$p' = 64,\ N = 4372$		
	P	M	%	P	M	%	P	M	%	P	M	%
p=4	0.908	0.899	1.0	0.877	0.871	0.6	0.859	0.856	0.3	0.877	0.871	0.6
p=8	1.000	1.000	-	0.966	0.969	0.3	0.946	0.952	0.6	0.966	0.969	0.3
p=16	-	-	-	1.000	1.000	-	0.979	0.982	0.3	1.000	1.000	-
p=32	-	-	-	-	-	-	1.000	1.000	-	1.021	1.018	0.2

5.2 Browser Interface

A Java 3D visualization environment is developed for the SCALA interface. This visualization tool is designed based on a client-server model. The server side mainly provides data services. At startup, the server accesses certain data files, creates data objects, and waits for the client to call. Currently the server supports two data file formats: Self-Defining Data Format (SDDF) [12] and a simple text format used by SCALA. The driving force behind the client/server approach is to increase accessibility, as most users may not have SDDF installed on their machines and may like to use SCALA over the net. Moreover, current distribution of SDDF only supports a few of computing platforms. Partitioning our tool into separate objects based on their services makes it possible to deploy the data server in a central site and the client objects anywhere on the Internet. The client is implemented in pure Java and the data servant is implemented as a CORBA compliant object so that it can be accessed by clients coded in other programming languages.

For effective visualization, a good environment should allow users to interact with graphical objects and reconfigure the interface. The basic configuration supported by the Java interface includes changing scale and color, and selecting specific performance metrics and views. At the run-time, the user can rotate, translate, and zoom the graphical objects. Automatic rotation similar to the animation is also an enabled feature. The graphical objects are built using Java 2D/3D API, which are parts of the JavaMedia suite of APIs [20].

We have implemented a custom canvas which serves as a drawing area for graphical objects. The paint() method in canvas class is extended to the full capability of drawing a complete 2D performance graph. The classes for 2D performance graphics are generic in the sense that the resulting graph depends only on the performance data. Besides 2D performance graphical objects, Java 3D is used to build three-dimension graphical objects. Java 3D uses the scene-graph based programming model in which individual application graphical elements are constructed as separate objects and connected together into a tree-like structure. It gives us high-level constructs for creating and manipulating 3D geometry and for constructing the structures used in rendering that geometry. As in developing 2D graphical classes, we analyze the 3D performance data and build several generic 3D graphical classes. A terrain grid class based on IndexedQuadArray can be used to describe the performance surface. The rotation, translation, and zooming by mouse are supported.

Figure 4 is the kiviat view which shows the variation of *cpi* (sequential computing capacity), communication latence, and parallel speed when problem size increases. Figure 5 shows the execution time as a function of work and number of processors on a given machine. These views and more are currently supported by the JAVA 3D visualization tool. While this Java visualization environment is developed for SCALA, its components are loosely coupled with other SCALA components and can be easily integrated into other performance tools. It separates data objects from GUI objects and is portable and reconfigurable.

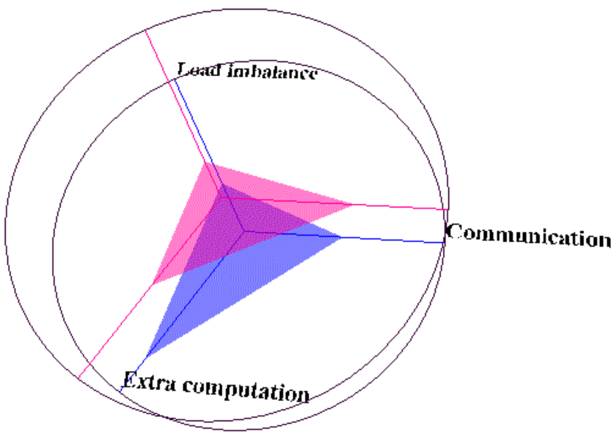

Fig. 4. *cpi*, Communication Latency, and parallel speed as a Function of Problem Size

6 Conclusion

The purpose of the SCALA project is to improve the state of the art in performance analysis of parallel codes by extending the current methodologies and to

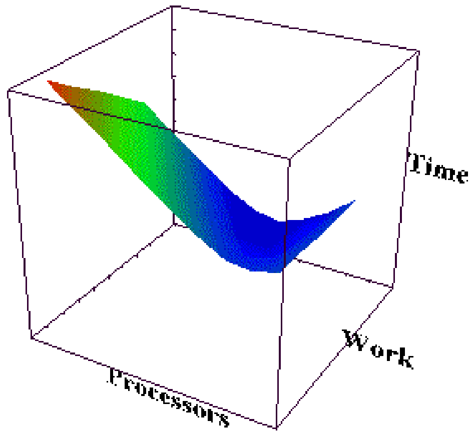

Fig. 5. Execution Time as a Function of Work and Number of Processors

develop a prototype system of real value for the high performance computing community. SCALA combines symbolic and static analysis of the parallelized code with dynamic performance data. The static and symbolic analysis are provided by the restructuring compiler, while dynamic data are collected by executing the parallel code on a small number of processors of the target machine. A performance model is then generated for predicting the scaling behavior of the code with varying input data size and machine characteristics. This approach to performance prediction and analysis provides the user with detailed information regarding the behavior of the parallelized code and the influences of the underlying architecture.

References

1. Adve, V. S., Crummey, J. M., Anderson, M., Kennedy, K., Wang, J.-C., and Reed, D. A. Integrating compilation and performance analysis for data parallel programs. In *Proc. of Workshop on Debugging and Performance Tuning for Parallel Computing Systems* (Jan. 1996).
2. Benkner, S., Sanjari, K., Sipkova, V., and Velkov, B. Parallelizing irregular applications with vienna HPF+ compiler VFC. In *HPCN Europe* (April 1998), Lecture Notes in Computer Science, Springer-Verlag.
3. Calzarossa, M., Massari, L., Merlo, A., Pantano, M., and Tessera, D. Medea: A tool for workload characterization of parallel systems. *IEEE Parallel & Distributed Technology Winter* (1995), 72–80.
4. Calzarossa, M., Massari, L., Merlo, A., Pantano, M., and Tessera, D. Integration of a compilation system and a performance tool: The HPF+ approach. In *LNCS - HPCN98* (Amsterdam, NL, 1998).
5. Fahringer, T. *Automatic Performance Prediction of Parallel Programs* . Kluwer Academic Publishers, Boston, USA, ISBN 0-7923-9708-8, March 1996.

6. Fox, G., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C., and Wu, M. Fortran D language specification. Technical Report,COMP TR90079, Department of Computer Science, Rice University, Mar. 1991.
7. Gustafson, J., Montry, G., and Benner, R. Development of parallel methods for a 1024-processor hypercube. *SIAM J. of Sci. and Stat. Computing 9*, 4 (July 1988), 609–638.
8. Hwang, K., and Xu, Z. *Scalable Parallel Computing*. McGraw-Hill WCB, 1998.
9. Kumar, V., Grama, A., Gupta, A., and Karypis, G. *Introduction to Parallel Computing, Design and Analysis of Algorithms*. The Benjamin/Cummings Publishing Company, Inc., 1994.
10. Lyon, G., Snelick, R., and Kacker, R. Synthetic-perturbation tuning of mimd programs. *Journal of Supercomputing 8*, 1 (1994), 5–8.
11. Noelle, M., Pantano, M., and Sun, X.-H. Communication overhead: Prediction and its influence on scalability. In *Proc. the International Conference on Parallel and Distributed Processing Techniques and Applications* (July 1998).
12. Reed, D., Aydt, R., Madhyastha, T., Noe, R., Shields, K., and Schwartz, B. An overview of the Pablo performance analysis environment. In *Technical Report*. UIUCCS, Nov. 1992.
13. Sahni, S., and Thanvantri, V. Performance metrics: Keeping the focus on runtime. *IEEE Parallel & Distributed Technology* (Spring 1996), 43–56.
14. Sun, X.-H. The relation of scalability and execution time. In *Proc. of the International Parallel Processing Symposium'96* (April 1996).
15. Sun, X.-H. Scalability versus execution time in scalable systems. TR-97-003 (Revised May 1998), Louisiana State Uiversity, Department of Computer Science, 1997.
16. Sun, X.-H. Performance range comparison via crossing point analysis. In *Lecture Notes in Computer Science, No 1388*, J. Rolim, Ed. Springer, March 1998. Parallel and Distributed Processing.
17. Sun, X.-H., He, D., Cameron, K., and Luo, Y. A factorial performance evaluation for hierarchical memory systems. In *Proc. of the IEEE Int'l Parallel Processing Symposium* (Apr. 1999).
18. Sun, X.-H., Pantano, M., and Fahringer, T. Integrated range comparison for data-parallel compilation systems. *IEEE Transactions on Parallel and Distributed Systems* (accepted to appear, 1999).
19. Sun, X.-H., and Rover, D. Scalability of parallel algorithm-machine combinations. *IEEE Transactions on Parallel and Distributed Systems* (June 1994), 599–613.
20. SUN Microsystems Inc. Java 3D API specification. http://java.sun.com/products/java-media/3D, 1998.