SCALA: A PERFORMANCE SYSTEM FOR SCALABLE COMPUTING

Xian-He Sun[1]
Thomas Fahringer[2]
Mario Pantano[2]

Abstract

Lack of effective performance-evaluation environments is a major barrier to the broader use of high performance computing. 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 and symbolic 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 more intuitive level of performance analysis for scalable computing. 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.

Key Words: Software System, Performance Evaluation, Parallel and Distributed Processing, Scalability Analysis, Symbolic Analysis, Statistical Analysis

1 Introduction

Although rapid advances in highly scalable multiprocessing systems are bringing petaflop 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. There are many ways to parallelize an application, but the relative performance of different parallelizations vary with problem size and system ensemble sizerformance systems have been developed for parallel computing (Mohr et al. 1996, Miller et al. 1995, Whaley and Dongarra 1998, Berman et al. 1996, Taylor et al. 2000). These systems have different strengths and focus on different applications. They are an important component of high performance computing. In this study we introduce the framework and initial implementation of the SCALability Analyzer (SCALA) system, a performance system designed for scalable computers. Distinguished from other existing performance tools, the program performance model generated by SCALA is based on scalability analysis (Sun 2002, Sun and Rover 194). 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: to predict performance, to support performance debugging and program restructuring, and to 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 (Sun 2002), 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 that supports performance oriented program development for OpenMP/MPI/HPF and mixed parallel (e.g., OpenMP/MPI) programs. SCALA is designed to explore the plausibility and credibility of new techniques, and to use them collectively to bring performance analy-

[1] DEPARTMENT OF COMPUTER SCIENCE, ILLINOIS INSTITUTE OF TECHNOLOGY, CHICAGO, IL 60616

[2] INSTITUTE FOR SOFTWARE TECHNOLOGY AND PARALLEL SYSTEMS, UNIVERSITY OF VIENNA LIECHTENSTEINSTR. 22 1090 VIENNA, AUSTRIA

The International Journal of High Performance Computing Applications, Volume 16, No. 4, Autumn 2002, pp. 000-000
© 2002 Sage Publications

sis 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 symbolic 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 accomplish 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 sources of information (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 is provided by the measurement system in terms of tracefiles. The user can also be involved in the process of 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 the 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 programmer by using 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 effects, finds the correlation between different program phases, identifies the scaling behavior of "difficult 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, and using dynamic data evaluates the resulting expressions for specific code-machine combinations.
- *Scalability Analysis:* The development of this module is based on the most advanced analytical results on scalability analysis as described in Section 3. The scalability module implements newly developed algorithms for predicting performance in terms of execution time and scalability of a code-machine combination.
- *Model Generator and Database:* The performance model generator determines the model of system execution. The database keeps 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 a 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 passed 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 with advanced performance information for a deep 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 the Vienna Fortran Compiler (VFC) (Benkner 1999, Benkner et al. 1995) which translates Fortran programs (Fortran90, MPI, OpenMP, HPF, and mixed program) into Fortran90/MPI or mixed OpenMP/MPI programs. Moreover, the Scala Instrumentation System (SIS) has been integrated into VFC to alleviate generic instrumentation of source codes for a variety of performance overheads and metrics for arbitrary code regions ranging from the entire program to single statements.

This integration allows the tool to directly gather detailed information on the source code being trans-

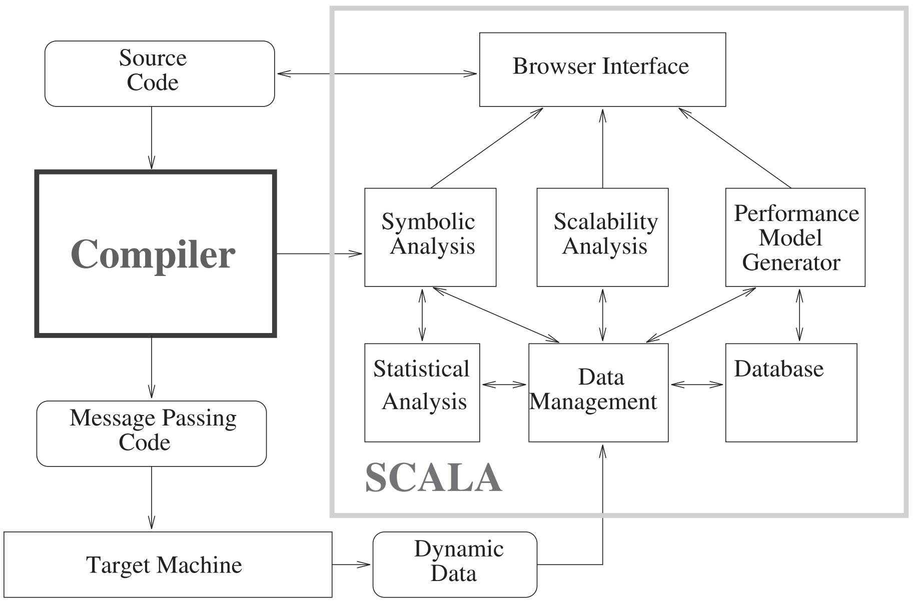

Fig. 1 Structure of SCALA.

formed. As a consequence, performance predictions can be linked back to the user source code providing suggestions for code optimizations. A brief discussion of the scalability analysis, symbolic 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 (Gustafson et al. 1988, Kumar et al. 1994, Sahni and Thanvantri 1996, Hwang and Xu 1998). SCALA's approach is based on *isospeed* scalability (Sun and Rover 1994): 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 underlying computing system. Unlike the widely used, speedup based parallel efficiency, average speed does not need the comparing of sequential single node performance. While the speedup based efficiency is a good measure for parallel performance gain, average speed is a more appropriate metric for the study of scalability.

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 of the code-machine combination is:

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

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

```
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.

In addition to measuring and computing scalability, the prediction of scalability and the relation between scalability and execution time have been well studied. A mechanism is developed to predict the scalability automatically (Sun et al. 1999). 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. If CMC 1 is faster (slower) than CMC 2 at ensemble size p, and $p' > p$ is the first scaled ensemble size such that CMC 1 becomes slower (faster) than CMC 2 at p', then we say p' is a performance crossing-point of CMC 1 and CMC 2, and CMC 1 overperforms CMC 2 over the ensemble range between p and $p'-1$. In contrast with execution time which is measured for a particular pair of problem and system sizes, range comparison compares performance over a wide range of ensemble and problem size via scalability and crossing-point analysis. Only the two most relevant theoretical results are given here. More results can be found in Sun (2002).

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.*

Result 1 shows that a better scalability will maintain a better execution time 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.

CMC 1 and CMC 2 may have different scalability and different scaled problem size W', W^* at the scaled ensemble size $p' > p$, respectively. By equation (1), we can use the scalability of CMC 1 and CMC 2 to find the performance crossing point in terms of the scaled problem size W', W^*, which we call the *scaled crossing point*. As given by Result 2, the scaled crossing point can be used to identify the performance crossing point (with the same problem size) and conduct range comparison.

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 the scalability comparison is practically valuable. Programming optimization in a large sense is to compare different programming options and to choose the best option available. Pre-measured performance can be sorted and used for performance prediction. Since execution time is determined by problem size, using pre-measured execution time to predict performance requires an extremely large database and is impractical. On the other hand, scalability is "dimensionless" and only need to be stored at each ensemble size. Result 2 provides a foundation for the performance prediction component of SCALA. Figure 2 gives the range comparison algorithm in terms of finding the smallest scaled crossing point via scalability comparison. More algorithms can be found in Sun (2002). In general, there could be more than one scaled crossing point over the consideration range for a given pair of CMCs. This algorithm can be used iteratively to find successive scaled crossing points.

4 Symbolic Analysis

The quality of scalability analysis depends significantly on the ability to determine crucial program information such as how much work is contained in a program, how much work has to be processed by a specific processor, communication overhead, etc. This as well as many other program analyses are seriously hampered by program unknowns (problem and machine sizes) and complex expressions that can stem from recurrences, tiling with symbolic block sizes (Benkner et al. 1995), linearized subscripts, and non-linear terms in subscript expressions. Ineffective approaches to gather and propagate sufficient information about variables through the program continue to have a detrimental impact on many compiler analyses and optimizations (Fahringer and Scholz 1997, Fahringer 1998, Blume and Eigenmann 1994, Tu and Padua 1995, Tu 1995). As a consequence worst case assumptions are frequently made or program analysis is done at runtime which increases the execution overhead. Therefore, sophisticated symbolic analysis that can cope with program unknowns is needed to alleviate these compiler deficiencies.

We use symbolic evaluation (Fahringer and Scholz 1997) which combines both data and control flow analysis to determine variable values, assumptions about and constraints between variable values, and conditions under which control flow reaches a program statement. Computations are represented as symbolic expressions defined over the program's problem and machine size. Each program variable is associated with a symbolic expression describing its value at a specific program point. In addition a path condition – represented by a first-order logic formula – describes the condition under which control flow reaches a given program point and the assumptions on variable values made for a given control flow branch.

The goal of our symbolic evaluation with respect to loops is to detect the recurrence variables, determine the recurrence system and finally find closed forms for recurrence variables at the loop exit by solving the recurrence system. We have implemented a recurrence solver written on top of Mathematica. Our symbolic evaluation techniques comprise accurate modeling of assignment and input/output statements, branches, loops, recurrences, arrays and procedures. Efficiency and accuracy are highly improved by aggressive simplification techniques. All of our techniques target both linear as well as non-linear expressions and constraints.

Symbolic evaluation is used as a basis for a variety of symbolic algorithms to compute statement execution counts as well as the work contained in a program and the amount of data transferred as a function of the problem and machine size (Fahringer 1998). The work to be done by an individual processor with respect to a specific program statement S is computed to be the number of times this processor is executing S weighted by the time it takes to execute a single instance of S. Symbolic execution times are functions over machine and problem sizes. The execution time of a single statement is obtained by the measurement system. Based on actual measurements on the target machines of interest the data management component is extracting and summing up all execution times for the operations contained in S. We are currently developing a symbolic execution time estimator for data-parallel Fortran programs which is based on our techniques for computing symbolic statement execution counts combined with execution times of operations (obtained by the data management component).

In the following we describe how to estimate the amount of work to be processed by every processor of a data parallel program. The following code shows a High

Performance Fortran (HPF) (1996) code excerpt with a processor array *PR* of size *p* (machine size) and two problem sizes n_1 and n_2,

```
INTEGER A(n2)
!HPF$ PROCESSORS :: PR(p)
!HPF$ DISTRIBUTE (BLOCK) ONTO PR :: A
DO J1 = 1, n1
 DO J2 = 1, J1* n1
 IF (J2 ≤ n2) THEN
S: A(J2) = ...
 ENDIF
 ENDDO
ENDDO
```

The loop contains a write operation to a one-dimensional array *A* which is block-distributed onto *p* processors. Let *k* ($1 \le k \le p$) denote a specific processor of the processor array. Computations that define the data elements owned by a processor *k* are performed exclusively by *k*. For the sake of simplicity we assume that *p* evenly divides n_2. Therefore, a processor *k* is executing the assignment to *A* based on the underlying block distribution if $n_2 * (k-1)/p + 1 \le J_2 \le n_2 * k/p$. The precise work to be processed by a processor *k* is the number of times *k* is writing *A*, which is defined by *work* (*k*).

The problem of estimating the amount of work to be done by processor *k* can now be formulated as counting the number of integer solutions to $\mathcal{I}$ which is given by:

$$1 \le J_1 \le n_1$$
$$1 \le J_2 \le J_1 * n_1$$
$$J_2 \le n_2$$
$$\frac{n_2 * (k-1)}{p} + 1 \le J_2 \le \frac{n_2 * k}{p}$$

$\mathcal{I}$ contains constraints for every loop bound and condition expression in the above example code. Furthermore, the constraints of the data distribution are included in $\mathcal{I}$. All constraints are defined over loop variables and machine and problem sizes. Note that if we omit the last two constraints of $\mathcal{I}$ then we obtain the overall work contained in a program. In the following we replace $n_2 * (k-1)/p + 1$ by *LB* and $n_2 * k/p$ by *UB*. By using various techniques to simplify systems of constraint (Fahringer 1998) we can determine that $1 \le J_2$ is made redundant by $B \le J_2$ and $J_2 \le n_2$ by $J_2 \le UB$. Therefore, the simplified $\mathcal{I}$ with all redundant inequalities removed is given by

$$1 \le J_1 \le n_1$$
$$LB \le J_2 \le UB$$
$$J_2 \le J_1 * n_1$$

Determining how many times processor *k* is executing statement *S* is identical with the problem to find the number of solutions of $\mathcal{I}$. We estimate *work*(*k*) by *work*(*k*) which is given as follows:

$$\overline{work(k)} = \sum_{1 \le i \le 3} \gamma(C_i) * E_i(k)$$

where

$$C_1 = \{UB \le n_1^2, p \le n_2\}$$

$$E_1(k) = \frac{(n_1 + UB - LB) * (LB - 2 * n_1 + 2 * LB * n_1 + UB)}{2 * n_1^2}$$

$$C_2 = \{\tfrac{UB}{n_1} > n_1, \tfrac{LB}{n_1} \le n_1\}$$

$$E_2(k) = (n_1 - \tfrac{LB}{n_1} + 1) * (\tfrac{n_1^2}{2} - \tfrac{LB}{2} + 1)$$

$$C_3 = \{n_2 \ge p, n_1^2 \ge UB + 1\}$$

$$E_3(k) = \tfrac{n_2}{p} * (n_1 - \tfrac{UB+1}{n_1} + 1)$$

and γ is defined as follows:

$$\gamma(\mathcal{C}) = \begin{cases} 1: & \text{if } \mathcal{C} = \text{TRUE} \\ 0: & \text{otherwise} \end{cases}$$

Notice that the result is given as a function of *k* (processor identification), machine and problem sizes. We have implemented a symbolic sum algorithm (Fahringer 1998) that estimates the number of solutions to a system of linear and non-linear symbolic constraints.

Most conventional performance estimators must repeat the entire performance analysis whenever the problem size or the number of processors used are changing. However, our symbolic performance analysis provides the solution of the above problem as a symbolic expression of the program unknowns (*p*, n_1, n_2, and *k*). For each change in the value of any program unknown we simply re-evaluate the result, instead of repeating the entire performance analysis. Table 1 shows an experiment which compares measured against estimated values for a 4 processor ($p = 4$) version of the example code by varying the values for n_1 and n_2. The measurements have been done by executing the example code on an iPSC/860 hypercube system and enumerating for each processor the number of times it writes array *A* in the loop of the example code. It can be seen that the estimates (*work*(*k*)) are very close to the measurements (*work*(*k*)). In the worst case the estimates are off by 1.19% for a relatively small problem size ($n_1 = 200$, $n_2 = 100$). Moreover, we also observe that for larger problem sizes the estimation accuracy consistently improves.

Table 1
Measured versus estimated values for the amount of work to be done by all processors for *p* = 4

n_1	n_2	$\sum_{1 \le k \le p} work(k)$	$\sum_{1 \le k \le p} \overline{work(k)}$	error in %
100	100	10000	10025	0.25
200	100	20000	20238	1.19
200	200	40000	40450	1.12
400	200	80000	80474	0.59
400	400	160000	160900	0.56
800	400	320000	320950	0.29
800	800	640000	641800	0.28

In order to obtain an execution time estimate for *S* we weight this figure with the pre-measured execution times for each operation contained in *S* as obtained from the data management component.

To compute the amount of data transferred as implied by a parallel program we specify a similar set of constraints that describes the non-local data accesses of a specific processor *k*. The number of solutions to this set of constraints determines the amount of data transferred with respect to *k*. In both cases we use a symbolic sum algorithm to compute the number of solutions to a set of constraints. We also can use this approach to determine whether two processors are exchanging data. For instance, let $\mathcal{I}_1$ and $\mathcal{I}_2$ determine the local data of a processor k_1 and the non-local data needed by a processor k_2, respectively. Then k_1 is sending data to k_2 if and only if there exists at least one solution to $\mathcal{I}_1 \cup \mathcal{I}_2$. We can therefore use our symbolic sum algorithm as a basis to compute the

- overall work contained in a program
- amount of work to be processed by a specific processor
- amount of data transferred, and
- number of transfers.

All of these performance metrics are functions over machine and problem sizes. Communication times can be computed based on the well-known linear communication model $\rho + \beta * D$, where ρ is the message startup overhead, and β is the message transfer overhead for sending a single data element. SIS can be used to determine measurements for ρ and β. Symbolic analysis can then combine these parameter values with symbolic functions for amount of data transferred and number of transfers in order to compute symbolic communication times as functions of machine and problem size.

We have implemented a prototype of our symbolic evaluation framework (Fahringer and Scholz 1997) as well as symbolic sum algorithm (Fahringer 1998) that estimates the number of solutions to a set of symbolic constraint. These techniques are used as part of the VFC – a parallelizing compiler for Fortran programs, PT (Fahringer 1995, 1996a) – a performance estimator, and SCALA in order to effectively parallelize and optimize Fortran programs for distributed and shared memory architectures.

5 Prototype Implementation

The implementation of SCALA involves several steps, which include the development of each component module, integration of these components, and 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.

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 Illinois Institute of Technology and University of Vienna, a first prototype of SCALA was developed and tested under the *VFC* as an integrated performance tool for revealing scaling behavior of simple and structured codes. In this framework, the static performance data have been pro-

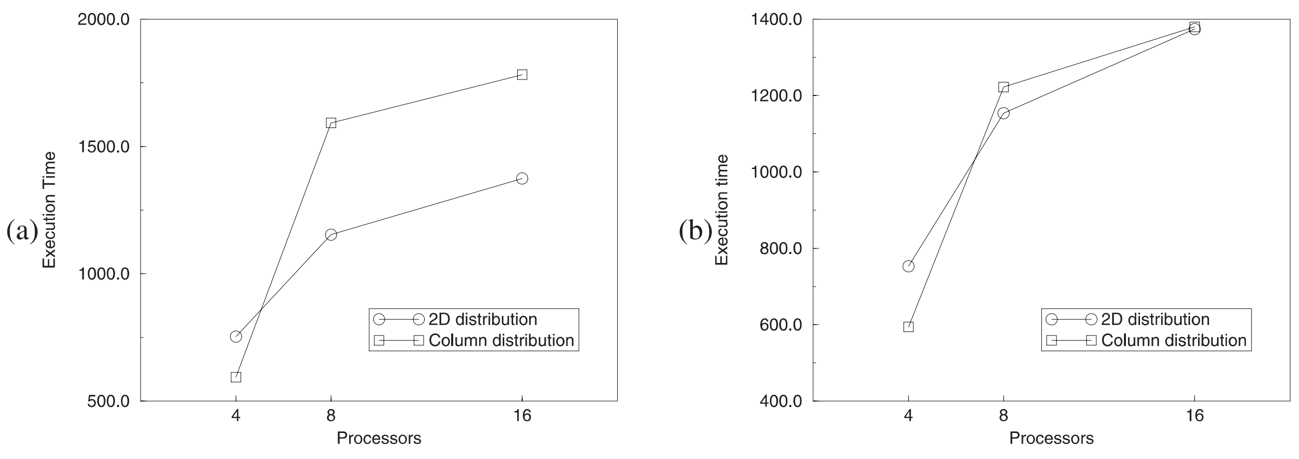

Fig. 2 Scaled crossing point (a) and crossing point (b) for the Jacobi with *n* = 20

vided by P^3T. The integration of P^3T into VFC enables the user to exploit considerable knowledge about the compiler's program structural analysis information and provides performance estimates for code restructuring. Among others P^3T traverses an abstract syntax tree generated by *VFC* in order to locate communication statements inserted by the compiler. For each communication statement, a list of constraints is determined by P^3T that determines which processor is communicating with what other processors. These constraints are based on the owner-computes model which indicates that a processor updates data if this data is owned by the processor. Any remote data that is needed to update local data must be transferred via message passing. In Fahringer (1996b) we have demonstrated how P^3T determines the amount of data transferred and the number of transfers for parallel programs by incorporating compile-time analysis. Such analysis is restricted to so-called regular programs which contain only linear array index and loop bound functions defined over loop variables of enclosing loops. The resulting constraints are incorporated by our symbolic analysis to determine the amount of data transferred and number of transfers as symbolic functions defined over problem and/or machine sizes (see Section 4). 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 VFC 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 $n = 30$ 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).

As a first step for investigating the performance of parallelized codes, we embedded in the VFC the SCALA Instrumentation System (SIS). SIS 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. The data management module accepts trace-files 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

Table 2
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 3
Bidirectional 2D 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

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 a 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 and 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 manufacturer 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 2 presents the communication values computed for the communication patterns 1D and 2D Torus used in the Jacobi implementation. As shown in Table 3, SCALA accurately predicts the scalability of the algorithm and therefore range comparison. More detailed information can be found in Noelle et al. (1998).

In the third experiment we used an OpenMP/MPI Fortran90 version of a code that simulates the ocean in order to explain the westward intensification of wind-driven ocean currents (Stommel 1948). The experiments were conducted on a cluster of SMP nodes, connected by Fast-Ethernet, each node consisting of 4 Intel Pentium III Xeon 700 MHz CPU, with 1MB full-speed L2 cache, 2 GByte ECC RAM, Ethernet and runs Linux 2.2.18-SMP. Jobs are scheduled on dedicated nodes by using PBS (Portable Batch queuing System, Veridian Systems). Our prototype implementation automatically invoked all the corresponding experiments and stored the output results and performance data in a database. We studied the scalability behaviour and the impact of various OpenMP loop distribution strategies on the resulting performance.

The *scalability* of the code was examined by varying: (1) the machine size, which consists of two dimensions: (i) the number of SMP nodes, controlled by directives inserted in the PBS script in order to schedule jobs; (ii) the number of threads per SMP node, controlled by the input parameter to the `omp_set_num_threads` OpenMP library routine; and (2) the problem size, by varying the grid (ocean) size and the number of iterations on the grid. An experiment was conducted to examine the different OpenMP loop scheduling strategies and their performance effects. A key issue for OpenMP programmers is to find an appropriate work distribution for parallel loops that are executed by a set of threads. Various options are provided to change the work distribution of loop iterations, which includes the scheduling strategy (i.e. `STATIC`, `DYNAMIC` and `GUIDED`) and the size of iterations (chunk) distributed onto a set of threads. `STATIC` means that the iterations are assigned to threads statically, before getting executed; `DYNAMIC` means that

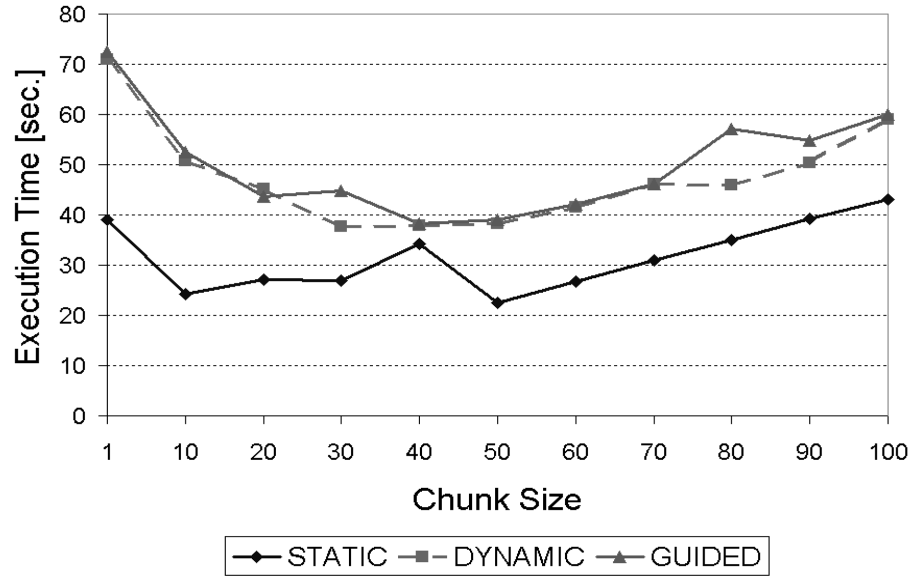

Fig. 4 Performance Study of an Ocean Simulation.

as each thread finishes a set of iteration space, it dynamically gets the next one; `GUIDED` means that the iteration space is divided into pieces such that the size of each successive piece is exponentially decreasing (Dagum and Menon 1998).

Figure 4 shows that for the problem size examined, `STATIC` scheduling performs better than `DYNAMIC` and `GUIDED`. The optimal chunk size is 50. Static scheduling is most likely superior because it implies the least runtime scheduling overhead. SCALA has been used to instrument the OpenMP/MPI Fortran code and measure the various execution times for different chunk sizes.

5.1 DATA MANAGEMENT

Many performance measurement systems exist right now (Miller et al. 1995, Mohr et al. 1996). While these performance systems have made their contribution to the state-of-the-art of performance evaluation, none of them has addressed the data presentation and understanding issue adequately. In SCALA, the (dynamic) data collection module, the data management module, and the database module are integrated to form a Performance Data Representation System (PDRS) (Sun and Wu 2000).

Trace Data Module. This module is in charge of collecting original performance data of parallel programs, and stores them with SDDF (Aydt 1995). The large volume of data involved in parallel computations requires that instrumentation to collect the data selectively and intelligently. One way to collect data of a parallel program is to instrument the program executable so that when the program runs, it generates the desired information. PDRS is designed to use the Scala Instrumentation System (SIS) (Fahringer et al. 2000) to get the SDDF trace data file. PDRS also provides a general interface that can be used under any system, which provides the SDDF trace data interface.

Data Management Module. As discussed in Section 2, this module is in charge of performance data filtering and mapping. Event histories of parallel programs are valuable information sources for performance analysis but the problem is how to extract the useful information from massive amounts of low-level event traces. Our system performs the data filtering as a preparation to store the event history into a relational database. The SDDF is a trace description language that specifies both data record structures and data record instances. The SCALA performance database is based on the SDDF specification. The data management module is designed to be implemented in Oracle DBMS.

Performance Database. We classify the performance data saved in the SDDF tracefiles into five groups: processor information, memory information, program information, communication information and I/O information. Each group is represented as an entity relation in the performance database. An individual event in a relation is treated as a tuple with a given unique identifier. The information retrieval is achieved by the relational database queries. The example below shows how objects can be retrieved using JDBC (Sun Microsystems 1997).

For instance, suppose that we want to get the communication events that occurred in processor 0, the query

```
select sourcePE, destinationPE,
messageLength, event_startTimestamp,
event_endTimestamp from Communication
Information where processor = 0.
```

We may make the following SQL query by JDBC:

```
ResultSet rs = stmt.executeQuery( "select
sourcePE, destinationPE, messageLength,
event_startTimestamp, event_endTimestamp
  from Communication Information where
        processor = 0");
while (rs.next()) {
  Object i1 = rs.getObject("sourcePE");
  Object i2 =
      rs.getObject("destinationPE");
  Object r1 =
      rs.getObject("messageLength");
  Object r2 =
      rs.getObject("event_startTimestamp");
  Object r3 =
      rs.getObject("event_endTimestamp");
}
```

Multiple versions of performance data are handled by specifying a version attribute in each tuple. By specifying a version number in each database query, we can get multiple versions of program performance for comparison. In addition to the default PDRS performance parameters, new performance parameters such as sound files can also be added by users and be supported by the database.

Relational Queries. Relational query is a function built on top of the database module to support the performance analysis modules. It includes four parts: Symbolic Analysis, Statistical Analysis, Scalability Analysis, and Performance Model Generator, corresponding to the analysis modules. This function is implemented in JDBC. Java applications include the PDA, PVA, and GUI module implemented by Java. The JDBC provides a bridge between Java applications and performance database. This function finds the appropriate performance data for the Performance Diagostic Agent (PDA) function and GUI. PDA invokes the performance modules for performance analysis.

Performance Diagnostic Agent (PDA). This function calls for performance analysis. Its function operation algorithm is as follows.

Algorithm (Performance diagnosis):

```
Performance analysis requests;
  switch (analysis type) {
    Statistical:
      Retrieve the performance information
            required;
      Get or compute the predicted
            performance range;
      Compute the real result of requested
            performance parameter;
      Compare the result with the
            performance range;
      If (the result is not in the
            performance range)
          Give an explanation (using graphics
                and sound);
      break;
    Scalability:
      Retrieve the performance information
            required;
      Get or compute the predicted
          scalability results;
      Compute the real scalability results;
      Compare the real result with the
            predicted results;
      Explain the compared results (using
            graphics and sound);
      break;
    Models:
      Retrieve the performance information
            required;
      Get the predicted performance range;
      Compute the real result of requested
            performance parameter;
      Compare the result with the
            performance range;
      If (the result is not in the
              performance range)
      Give an explanation (using graphics
              and sound);
      break;
    Default: printf("No such analysis
      type");
    break;
}
```

The PDA retrieves the performance information from the performance database, and invokes the appropriate analysis module for performance analysis.

5.2 BROWSER INTERFACE

A Java 3D visualization environment is also 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) 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

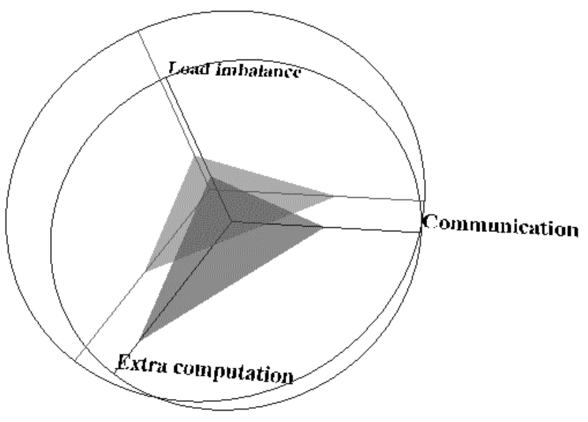

Fig. 5 ***Cpi*, Communication Latency, and parallel speed as a Function of Problem Size.**

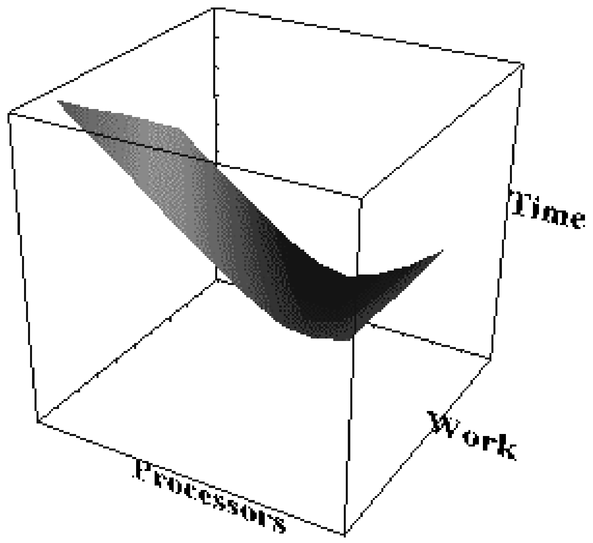

Fig. 6 Execution Time as a Function of Work and Number of Processors.

and may like to use SCALA over the net. Moreover, the current distribution of SDDF supports only a few 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 graphics 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 graphics objects. Automatic rotation similar to the animation is also an enabled feature. The graphics objects are built using Java 2D/3D API, which are parts of the JavaMedia suite of APIs.

We have implemented a custom canvas which serves as a drawing area for graphics 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 graphics objects, Java 3D is used to build three-dimensional graphics objects. Java 3D uses the scene-graph based programming model in which individual application graphics 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 graphics classes, we analyze the 3D performance data and build several generic 3D graphics classes. A terrain grid class based on IndexedQuadArray can be used to describe the performance surface. Rotation, translation, and zooming by mouse are supported.

Figure 5 is the kiviat view which shows the variation of *cpi* (sequential computing capacity), communication latence, and parallel speed when problem size increases. Figure 6 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.

6 Conclusion

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

Current development of SCALA has made its contribution to the state-of-the-art of performance evaluation. However, SCALA is a continuing research. Only a prototype system has been tested for the proof of concept. Many research issues remain open. In the near future we plan to continue this research with three focuses: enhance the symbolic analysis module to support more sophisticate program paradigms, adopt low-level hardware monitor information for advanced performance analysis and characterization, and extend the research to grid computing.

ACKNOWLEDGMENTS

The authors are indebted to several individuals for their help with this research. In particular, Dr. Xingfu Wu participated in the development of the PDRS; Hong Linh Truong and Radu Prodan contributed to the implementation and testing of the scalability analysis module and the SIS system, respectively, and Zhaohua Zhan contributed to the implementation of the visualization environment. The authors are grateful to the referees for their valuable suggestions that helped improve the technical quality and presentation of the paper.

AUTHORS BIOGRAPHIES

Xian-He Sun received his Ph.D. degree in Computer Science from Michigan State University. He was a staff scientist at ICASE, NASA Langley Research Center and was an associate professor in the Computer Science Department at Louisiana State University (LSU). Currently he is a professor and the director of the Scalable Computing Software laboratory in the Computer Science Department at Illinois Institute of Technology (IIT), and a guest faculty member at the Argonne National Laboratory. Dr. Sun's research interests include parallel and distributed processing, software systems, performance evaluation, and scientific computing. He has published extensively in the field and his research has been supported by DoD, DoE, NASA, NSF, and other government agencies. He is a senior member of IEEE, a member of ACM, New York Academy of Science, PHI KAPPA PHI, and has served and is serving as the chairman or is on the program committee for a number of international conferences and workshops. He received the ONR and ASEE Certificate of Recognition award in 1999, and received the Best Paper Award from the International Conference on Parallel Processing (ICPP01) in 2001.

Thomas Fahringer received a Masters degree in 1988 and a Ph.D. in 1993, all in Computer Science from the Technical University of Vienna, Austria. Between 1988 and 1990 he was a visiting scientist at the Engineering Design Research Center at Carnegie Mellon University in Pittsburgh, PA. From 1990–1998 he was Assistant Professor of Computer Science of Computer Science at the Institute for Software Science, University of Vienna. Since 1998 he has been an associate professor of Computer Science at the Institute for Software Science, University of Vienna. His research focuses on software tools and programming environments for distributed and concurrent systems, in particular, programming paradigms and methods, parallelizing compilers, debuggers, symbolic program and performance analysis, and performance-oriented application development for cluster and Grid architectures. Readers may contact Fahringer at the Institute for Software Science, University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria. e-mail: tf@par.univie.ac.at.

Mario Pantano received a Laurea degree in Computer Science from the University of Milan, Italy in 1990 and a Ph.D. in Electronic Engineering and Computer Science from the University of Pavia, Italy in 1994. From 1994 to 1998 he was with the Institute for Software Technology and Parallel Systems, University of Vienna, Austria, first as researcher and then as project manager of the EU Esprit project "HPF+: Optimizing HPF for Advanced Applications". In May 1998 he joined the Department of Computer Science at the University of Illinois at Urbana-Champaign. His main research interests include performance measurement, modeling and analysis of parallel programs, scalability analysis, tools and run-time systems for parallel program instrumentation and measurements, and design of automatic parallelization environments.

REFERENCES

Adve, V.S., Crummey, J.M., Anderson, M., Kennedy, K., Wang, J.-C., and Reed, D.A. 1995. "An integrated compilation performance analysis environment for data parallel programs," in *Proc. of Supercomputing*, (San Diego, CA) December.

Aydt, R. 1995. "The Pablo Self-Defining Data Format." Department of Computer Science, University of Illinois, ftp://bugle.cs.uiuc.edu/pub/Release/Documentation/SDDF.ps, April.

Benkner, S. 1999. "VFC: The Vienna Fortran Compiler," *Scientific Programming* 7(1): 67–81.

Benkner, S., Andel, S., Blasko, R., Brezany, P., Celic, A., Chapman, B., Egg, M., Fahringer, T., Hulman, J., Hou, Y., Kelc, E., Mehofer, E., Moritsch, H., Paul, M., Sanjari, K., Sipkova, V., Velkov, B., Wender, B., and Zima, H. 1995. *Vienna Fortran Compilation System – Version 2.0 – User's Guide*.

Berman, F., Wolski, R., Figueira, S., Schopf, J., and Shao, G. 1996. "Application-level scheduling on distributed heterogeneous networks," in *Proc. of Supercomputing '96*.

Blume, W. and Eigenmann, R. 1994. "An Overview of Symbolic Analysis Techniques Needed for the Effective Parallelization of the Perfect Benchmarks," in *Proceedings of the 1994 International Conference on Parallel Processing*, (St. Charles, IL).

Dagum, L. and Menon, R. 1998. "OpenMP: An industry-standard API for shared-memory programming," *IEEE Computational Science and Engineering* 5(Mar.): 46–55.

Fahringer, T. 1995. "Estimating and optimizing performance for parallel programs," *IEEE Computer* 28(Nov.): 47–56.

Fahringer, T. 1996a. *Automatic Performance Prediction of Parallel Programs*. Kluwer Academic Publishers, Boston.

Fahringer, T. 1996b. "Compile-Time Estimation of Communication Costs for Data Parallel Programs," *Journal of Parallel and Distributed Computing* 39(Nov.): 46–65.

Fahringer, T. 1998. "Efficient Symbolic Analysis for Parallelizing Compilers and Performance Estimators," *Journal of Supercomputing* 12: 227–252.

Fahringer, T., and Scholz, B. 1997. "Symbolic Evaluation for Parallelizing Compilers," in *Proc. of the 11th ACM International Conference on Supercomputing*, (Vienna, Austria), pp. 261–268, ACM Press.

Fahringer, T., Scholz, B., and Sun, X-H. 2000. "Execution-driven performance analysis for distributed and parallel systems," in *Proc. of the Second ACM International Workshop on Software and Performance (WOSP'2000)*, September.

Gustafson, J., Montry, G., and Benner, R. 1988. "Development of parallel methods for a 1024-processor hypercube," *SIAM J. of Sci. and Stat. Computing* 9: 609–638.

"High Performance FORTRAN Language Specification." Technical Report, Version 2.0.δ, Rice University, Houston, TX, October 1996.

Hwang, K. and Xu, Z. 1998. *Scalable Parallel Computing*. McGraw-Hill WCB.

Kumar, V., Grama, A., Gupta, A., and Karypis, G. 1994. *Introduction to Parallel Computing, Design and Analysis of Algorithms*. The Benjamin/Cummings Publishing Company, Inc.

Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., and Newhall, T. 1995. "The paradyn parallel performance measurement tools," *IEEE Computer* 28(11).

Mohr, B., Malony, A., and Cuny, J.E. 1996. "TAU," in *Parallel Programming Using C++* (G. Wilson, ed.), MIT Press.

Noelle, M., Pantano, M., and Sun, X-H. 1998. "Communication overhead: Prediction and its influence on scalability," in *Proc. the International Conference on Parallel and Distributed Processing Techniques and Applications*, July.

Sahni, S., and Thanvantri, V. 1996. "Performance metrics: Keeping the focus on runtime," *IEEE Parallel & Distributed Technology* Spring: 43–56.

Stommel, H.M. 1948. "The western intensification of wind-driven ocean currents," *Transactions American Geophysical Union* 29: 202–206.

SUN Microsystems Inc. 1997. "JDBC: a java SQL API, version 1.20," http://www.javasoft.com/products/jdbc/index.html.

Sun, X-H. 2002. "Scalability versus execution time in scalable systems," *Journal of Parallel and Distributed Computing* 62: 173–192.

Sun, X-H., Pantano, M., and Fahringer, T. 1999. "Integrated range comparison for data-parallel compilation systems," *IEEE Transactions on Parallel and Distributed Systems* 10: 448–458.

Sun, X-H., and Rover, D. 1994. "Scalability of parallel algorithm-machine combinations," *IEEE Transactions on Parallel and Distributed Systems* June: 599–613.

Sun, X-H., and Wu, X. 2000. "PDRS: A performance data representation system," in *Proc. of 5th IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS2000)*, April.

Taylor, V., Wu, X., et al. 2000. "Prophesy: An infrastructure for analyzing and modeling the performance of parallel and distributed applications," in *Proc. of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC's 2000)*, August.

Tu, P. 1995. *Automatic Array Privatization and Demand-Driven Symbolic Analysis*. PhD thesis, University of Illinois at Urbana-Champaign.

Tu, P. and Padua, D. 1995. "Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers," in *9th ACM International Conference on Supercomputing*, (Barcelona, Spain), pp. 414–423, July.

Veridian Systems, "PBS: The Portable Batch System," http://www.openpbs.org.

Whaley, R.C., and Dongarra, J. 1998. "Automatically tuned linear algebra software," in *Proc. of Supercomputing*, November.