Integrated Range Comparison for Data-Parallel Compilation Systems

Xian-He Sun, *Senior Member, IEEE,* Mario Pantano, and Thomas Fahringer

Abstract—A major difficulty in restructuring compilation, and in parallel programming in general, is how to compare parallel performance over a range of system and problem sizes. Execution time varies with system and problem size and an initially fast implementation may become slow when system and problem size scale up. This paper introduces the concept of range comparison. Unlike conventional execution time comparison in which performance is compared for a particular system and problem size, range comparison compares the performance of programs over a range of ensemble and problem sizes via scalability and performance crossing point analysis. A novel algorithm is developed to predict the crossing point automatically. The correctness of the algorithm is proven and a methodology is developed to integrate range comparison into restructuring compilations for data-parallel programming. A preliminary prototype of the methodology is implemented and tested under Vienna Fortran Compilation System. Experimental results demonstrate that range comparison is feasible and effective. It is an important asset for program evaluation, restructuring compilation, and parallel programming.

Index Terms—Performance evaluation, parallel compiler, scalable computing, software systems.

1 Introduction

The most significant question with parallel machines is the same today as it has been for many decades: How can software applications take advantage of hardware parallelism [1]. Traditionally, distributed memory architectures have been programmed using message passing, where the user is responsible for explicitly inserting communication statements into a sequential program. The development of parallel languages, such as Vienna Fortran [2], Fortran D [3], and High Performance Fortran (HPF) [4], improved the situation by providing high-level features for the specification of data distributions. Among others, the Vienna Fortran Compilation System (VFCS) [5] and Fortran D compilation system [3] have been developed to support such languages and to automatically generate a message passing program. However, current technology of code restructuring systems inherently lacks the power to fully exploit the performance offered by distributed memory architectures. The primary motivation of parallel processing is high performance. Effectiveness and efficiency of restructuring compilation are the current barriers for the success of a simple, high-level programming model approach.

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. This iterative process terminates when certain predefined performance criteria are met or as a result of explicit user intervention. Integrating performance analysis with a restructuring system is critical to support automatic performance tuning in the iterative restructuring process. The development of a fully compiler-integrated performance system for scalable parallel machines is especially challenging. In a scalable environment, the performance of a program varies with data distribution, system size (number of processors), and problem size. A superior program implementation is only superior over a range of system and problem sizes. Predicting the performance of parallel programs and integrating performance indices automatically into a restructuring compiler are two major challenges facing researchers in the field [6]. Moreover, current performance analysis and visualization tools are targeted at message-passing programming models where parallelism and interprocessor communication are explicit. They fall short in supporting high-level languages and are not readily integrated into restructuring compilers.

Two major functionalities of data-parallel restructuring compilers are the distribution of data arrays over processors and the choice of appropriate restructuring transformations. A key question in realizing these two functionalities is how to predict the scaled performances of a small number of data distributions and transformations automatically so that appropriate optimization decisions can be made. In order to compare relative performance over a range of problem and system sizes, scalability prediction is proposed as a solution in this study. Scalability is the ability to maintain parallel processing gain when system and problem size increase. It characterizes the scaling property of a code on a given machine. A slow code with good scalability may become

- X.-H. Sun is with the Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803-4020. E-mail: sun@csc.lsu.edu.
- M. Pantano is with the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL.
- T. Fahringer is with the Institute for Software Technology and Parallel Systems, University of Vienna, Liechtensteinsstr. 22 1090, Vienna, Austria. E-mail: tf@par.univie.ac.at.

Manuscript received 24 Apr. 1997; revised 4 May 1998.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 104935.

superior when system and problem sizes scale up. The system sizes for which the performance ranking of different code changes are called crossing points. In this paper, we introduce the concept of range comparison, which is concerned with the determination of crossing points. Based on analytical results given in Section 3.2, automatic crossing point prediction and automatic range comparison are studied in this research. An iterative algorithm is first derived to predict the scalability and crossing point on a given parallel platform. Then, the connection between the iterative algorithm and an existing static performance estimator, P^3T [7], is discussed. A preliminary prototype of automatic range comparison is implemented under the Vienna Fortran Compilation System (VFCS). Finally, two applications are tested with two different data distributions to verify the correctness and feasibility of the range comparison approach. While current experimental results are preliminary, they clearly demonstrate the feasibility and effectiveness of the range comparison approach for program restructuring.

This paper is organized as follows: VFCS and its performance estimation tool are introduced in Section 2. The concept of scalability, performance crossing point and range comparison are presented in Section 3. An iterative algorithm for automatic performance prediction is described in detail. Experimental results are given in Section 4 to illustrate how the newly proposed algorithm can be integrated within VFCS in order to predict the crossing point automatically. Finally, Section 5 concludes with a summary.

2 Vienna Fortran Compilation System

VFCS is a parallelizing compiler for Vienna Fortran and High Performance Fortran. VFCS is integrated with several tools for program analysis and transformation and, among others, provides a parallelization technique which is based upon *domain decomposition* in conjunction with the Single-Program-Multiple-Data (SPMD) programming model. This model implies that each processor is executing the same program based on a different data domain. The *work distribution* of a parallel program is determined—based on the underlying data distribution—according to the *owner-computes rule*, which means that the processor that owns a datum will perform the computations that make an assignment to this datum. Nonlocal data referenced by a processor imply communication which is optimized by several strategies [5], such as extracting single element messages from a loop and combining them into vectors (communication vectorization), removing redundant communication (communication fusion), and aggregating different communication statements (communication aggregation). The analysis described in this paper is targeted toward regular computations, such as stencil computations, and relies heavily on compile-time analysis and optimization as provided by VFCS.

2.1 P^3T: A Performance Estimator

P^3T [7], [8] is an integrated tool of *VFCS* which assists users in performance tuning of regular programs at compile time. P^3T is based on a single profile run to obtain characteristic data for branching probabilities, statement, and loop execution counts. It is well-known [9], [10], [11], [12] that the overhead to access nonlocal data from remote processors on distributed memory architectures is commonly orders of magnitude higher than the cost of accessing local data. Communication overhead is, therefore, one of the most important metrics in choosing an appropriate data distribution. P^3T models communication overhead by two separate performance parameters: number of data transfers and amount of data transferred. For the sake of brevity, only issues of static estimation of communication overhead are discussed in this section. Interested readers may refer to [7], [8], [13] for more information regarding the other performance parameters of P^3T.

Note that, in Section 4, we define communication time that combines the P^3T parameters mentioned above and various machine specific metrics.

2.1.1 Number of Data Transfers

The number of data transfers is a critical parameter which reflects the high message startup costs on most distributed memory architectures. Commonly, the overhead for communication is decreasing if it can be hoisted outside of a nested loop. Moreover, communication inside of a specific loop body in many cases implies that the loop is sequentialized due to synchronization between the processors involved in the communication. P^3T carefully models the loop nesting level at which a communication is placed, array access patterns, data dependences and distribution, control flow, and compiler communication optimizations (e.g., communication vectorization and fusion) in order to determine the number of data transfers with high accuracy.

For communication that can be hoisted outside a loop nest, we assume the loosely synchronous communication model [14], which implies that all involved processors communicate simultaneously. For such a communication statement, the number of data transfers is determined by the maximum number of data transfers across all involved processors. For communication that cannot be hoisted outside a loop nest due to a data dependence, we assume that it sequentializes the loop at which the communication is placed, as well as all data transfers implied by the communication. The number of data transfers for such a communication is given by the sum of data transfers across all processors involved in the communication.

2.1.2 Amount of Data Transferred

The current generation of distributed memory architectures reduces the impact of the message length on the communication overhead. For applications that transmit small data volumes, the startup cost is the predominate communication cost factor. However, for increasing data volumes transmitted, the message transfer time per byte and, in turn, the amount of data transferred become the first order performance effect. In order to provide a highly accurate estimate for the amount of data transferred (given in bytes) as induced by a parallel program, P^3T estimates the number of nonlocal data elements accessed and incorporates machine specific data type sizes. For this purpose, P^3T examines the loop nesting level at which a communication is placed, array access patterns, data dependences and distributions, control flow, and compiler communication optimizations.

As the compiler specifies the communication pattern at the source code level, the target architecture can be for the most part—except for data type sizes—ignored. Consequently, this parameter ports easily to a large class of distributed memory architectures.

3 Performance Range Comparison

While execution time is an important performance metric for optimizing parallel programs, its comparison bonds to a specific pair of system and problem size. Execution time alone is not sufficient for performance comparison over a range of system and problem sizes. Scalability has been recognized as an important property of parallel algorithms and machines in recent years [15]. Several scalability metrics have been proposed [16], [17], [18]. However, scalability has been traditionally studied separately as an independent property. Only very recently has the relation of scalability and execution time been studied and the concept of range comparison been introduced [19], [20]. Unlike conventional execution time comparison, in which performance is compared at a particular system and problem size, range comparison compares the performance of programs over a range of system and problem size via scalability and performance crossing point analysis. To fully understand the concept of range comparison, some background of scalability and crossing point analysis needs to be introduced.

3.1 Isospeed Scalability

A major driving force behind parallel computing is to solve large problems fast. Traditionally, execution time is the measure of choice for fixed-size problems. Execution time by itself, however, is not adequate for scalable computing, where problem size scales up with system size. Speed, defined as work divided by time, has been proposed as an alternative primary metric for scalable computing. Average speed is the achieved speed divided by the number of processors used.[1] Average speed is a quantity that ideally would be unchanged with scaled system size. The following definition was first given in [16].

Definition 1 (Isospeed Scalability of Algorithm-Machine Combination). *An Algorithm-Machine Combination is scalable if the achieved average speed of the algorithm on the given machine can remain constant with the increasing number of processors, provided the problem size can be increased with the system size.*

For a large class of Algorithm-Machine Combinations (AMCs), the average speed can be maintained by increasing the problem size. The necessary problem size increase varies with algorithm-machine combinations. This variation provides a quantitative measurement for scalability. Let W be the amount of work of an algorithm when p processors are employed in a machine, and let W' be the amount of work of the algorithm when $p' > p$ processors are employed to maintain the average speed, then the scalability from system size p to system size p' of the algorithm-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. Finally, let $T_p(W)$ be the time for computing W work on a p processors system; (2) shows how scaled execution time can be computed from scalability

$$T_{p'}(W') = \psi^{-1}(p,p') \cdot T_p(W). \tag{2}$$

Three approaches have been proposed to determine scalabilities [16]. They are: *computing* the relation between problem size and speed, directly *measuring* the scalability, and *predicting* scalability with certain predetermined parameters. While all of the three approaches are practically important, scalability prediction seems to be less expensive and benefits most from compiler support.

The parallel execution time $T_p(W)$ can be divided into two parts: the ideal parallel processing time and parallel processing overhead, $T_o(W)$.

$$T_p(W) = \frac{T_s(W)}{p} + T_o(W) = \frac{W \cdot \Delta}{p} + T_o(W), \tag{3}$$

where T_s is the sequential execution time, Δ is the computing capacity, defined as time per unit of work, of a single processor. The parallel processing overhead T_o contains the load imbalance overhead, communication overhead, and other possible parallelism degradations. By the definition of scalability (see (1)), scalability can be predicted if and only if the scaled work size, W', can be predicted. A prediction formula has been given in [21] to compute W':

$$W' = \frac{a \cdot p' \cdot T'_o(W')}{1 - a\Delta}, \tag{4}$$

where a is the achieved average speed and $T'_o(W')$ is the parallel processing overhead on p' processors. When parallel degradation does exist (i.e., $T'_o(W') > 0$), $a \cdot \Delta < 1$ and, therefore, (4) is traceable. $T'_o > 0$ is a necessary and sufficient condition of (4). When $T'_o = 0$, ideal scalability is achieved with $\psi(p,p') = 1$. Parallel processing overhead $T'_o(W')$ in general is a function of problem size. With unknowns on both sides of the equation, using (4) for scalability prediction is not a straightforward task.

3.2 Performance Crossing Point and Range Comparison

Theorem 1 gives a relation between scalability and execution time of two different algorithm-machine combinations. It has been analytically proven and experimentally confirmed in [19].

Theorem 1. *If algorithm-machine combinations 1 and 2 have execution time $\alpha \cdot T$ and T, respectively, at the same initial state (the same initial system and problem size), then combination 1 has a higher scalability than combination 2 at a scaled system size if and only if the execution time of*

1. How to determine work, in general, is debatable. For scientific applications, it is commonly agreed that the floating point (flop) operation count is a good estimate of work.

`Assumption of the Algorithm:` Assume algorithm-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. 1. Range comparision via crossing point analysis.

combination 1 is smaller than the α multiple of the execution time of combination 2 for solving W' at the scaled system size, where W' is the scaled problem size of combination 1.

Theorem 1 shows that if an AMC is faster at the initial state and has a better scalability than that of others, then it will remain faster over the scalable range. Range comparison becomes more difficult when the initially faster AMC has a smaller scalability. When the system size scales up, an originally faster code with lower scalability can become slower than another code with a better scalability. Finding the fast/slow crossing point is critical for optimizing performance and choosing efficient data distributions and program transformations in a data-parallel environment. Finding the superiority/inferiority crossing point, however, is very difficult. The definition of crossing point is problem size and system size dependent. Definition 2 gives a formal definition of crossing point based on the isospeed scalability [20].

Definition 2 (Scaled Crossing Point). *For any $\alpha > 1$, if algorithm-machine combinations 1 and 2 have execution time αT and T, respectively, at the same initial state, then we say a scaled system size p' is a crossing point of combinations 1 and 2 if the ratio of the isospeed scalability of combination 1 and combination 2 is greater than α at p'.*

Let AMC 1 have execution time t, scalability $\Phi(p, p')$, and scaled problem size W'. Let AMC 2 have execution time T, scalability $\Psi(p, p')$, and scaled problem size W^*. By Definition 2, p' is the crossing point of AMC 1 and 2 if and only if

$$\frac{\Phi(p, p')}{\Psi(p, p')} > \alpha. \tag{5}$$

In fact, by (2), when $\Phi(p, p') \ge \alpha\Psi(p, p')$, we have $t_{p'}(W') \le T_{p'}(W^*)$. Notice that, since $\alpha > 1$, combination 2 has a smaller execution time at the initial state, $t_p(W) > T_p(W)$. This superiority/inferiority changing in execution time gives the meaning of performance crossing point. The correctness of Theorems 2 and 3 is proven in [20], [22].

Theorem 2. *If algorithm-machine combination 1 has a larger execution time than algorithm-machine combination 2 at the same initial state, then, for any scaled system size p', p' is a scaled crossing point if and only if combination 1 has a smaller scaled execution time than that of combination 2.*

Theorem 3. *Assume algorithm-machine combination 1 has a larger execution time than algorithm-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.*

Theorem 3 gives the necessary condition for range comparison of scalable systems: p' is not a crossing point of p if and only if the fast/slow relation of the systems does not change for any scaled problem size within the scalable range of the two compared algorithm-machine combinations. Based on this fundamental finding, with the comparison of scalability, we can predict the relative performance

`Assumption of the Algorithm`: Assume work W and overhead T_o are increasing functions of the scaling parameter n, $W = f(n)$ and $T_o = g(n)$, or $T_o = g(n)$ is a constant, and assume the parallel code under study has been executed on the target machine with W work and p processors.

`Objective of the Algorithm`: Compute the scalability from system size p to p', where $p' > p$, with an error of $\epsilon > 0$.

```
Iterative Method
Begin
     Initial Value: $W_0 = \frac{p' \cdot W}{p}$;
     Compute $\phi(W_0)$;
     If $\phi(W_0) := W_0$ do
        $W' = W_0$;
     Elseif $\phi(W_0) > W_0$ do
        Begin Iteration (k=0; k++)
              $W_{k+1} = \phi(W_k)$;
        until $\|W_{k+1} - W_k\| < \epsilon$;
        $W' = W_{k+1}$;
     Else do
        Begin Iteration (k=0; k++)
              $W_{k+1} = \phi^{-1}(W_k)$
        until $\|W_{k+1} - W_k\| < \epsilon$
        $W' = W_{k+1}$;
     End{If}
End{Iterative Method}

Subroutine $\phi(W)$
Begin
     Solve $f(n) = W$ for $n$;
     Compute $T_o = g(n)$;
     Compute $\phi(W) = \frac{a \cdot p \cdot T_o}{1 - a \cdot \triangle}$;
End{Subroutine $\phi(W)$}

Subroutine $\phi^{-1}(W)$
Begin
     Compute $T_o = \frac{1 - a \cdot \triangle}{a \cdot p} W$;
     Solve $g(n) = T_o$ for $n$;
     Compute $\phi^{-1}(W) = f(n)$;
End{Subroutine $\phi^{-1}(W)$}
```

Fig. 2. An iterative method for predicting scalability.

of computing systems over a range of problem sizes and machine sizes. This unique property of scalability comparison is practically valuable. It provides a more reasonable comparison of computing systems and guides for optimizing computing for a given range of applications. Fig. 1 gives the range comparison algorithm in terms of finding the smallest crossing point via scalability comparison. While not listed here, an alternative range comparison algorithm that finds the smallest crossing point via execution time comparison can be found in [22]. In general, there could be more than one crossing point over the consideration range

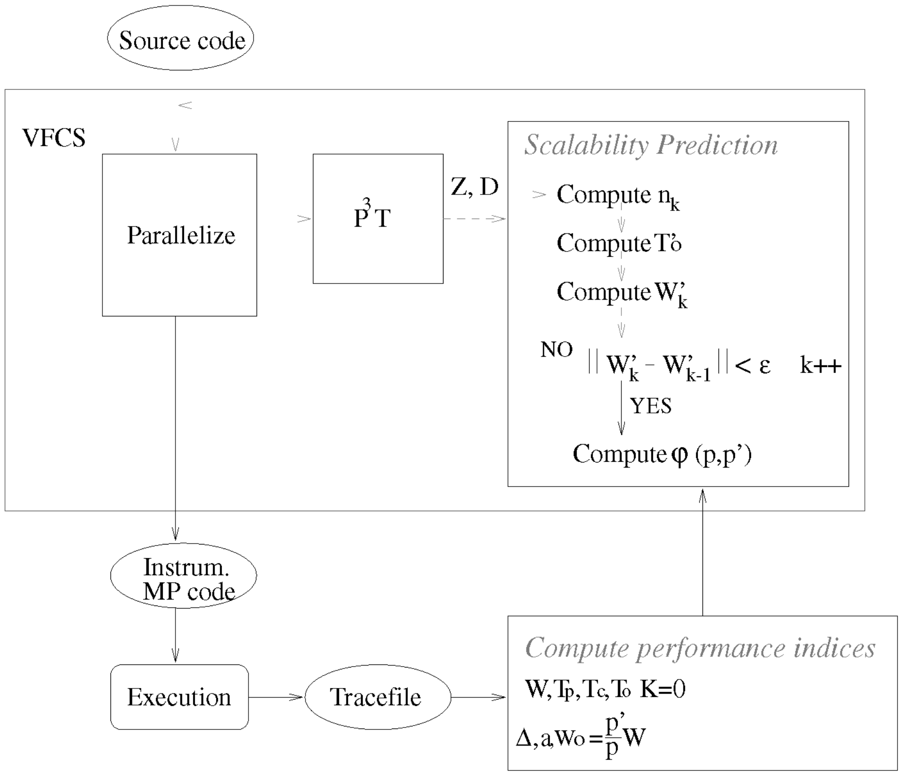

Fig. 3. Scalability prediction within VFCS.

for a given pair of CMCs. These two algorithms can be used iteratively to find successive crossing points.

3.3 Automatic Crossing-Point Prediction

The procedure of range comparison listed in Fig. 1 is in terms of scalability. Scalabilities of different code implementations, or different algorithm-machine combinations in general, still need to be determined for range comparison. Scalabilities of different algorithmic implementations can be prestored for performance comparison. In many situations, however, premeasured results of scaled systems are not available and predictions are necessary. We propose an iterative method, listed in Fig. 2, to compute W' and to predict the scalability automatically. We assume that the underlying application is scalable and its work W is a monotonically increasing function of a scaling parameter n (input data size). We also assume that parallel overhead T_o is either independent of parameter n (ideally scalable) or is monotonically increasing with n (parallel degradation exists). The iterative algorithm consists of three parts: the main program and two subroutines for computing the function of $\phi(W)$ and the inverse of $\phi(W)$. Function $\phi(W)$ is implied by (4). Mathematically, the iterative algorithm is to find a fixed point of $\phi(W)$ such that $W = \phi(W)$. A proof of correctness of the algorithm is provided in [23].

Our correctness proof does not give the convergence rate of the iteration algorithm. Like most iterative methods, the convergence rate of the algorithm is application dependent. It depends on the properties of function $f(n)$. For most scientific computations, $f(n)$ is a low degree polynomial function and the algorithm converges very fast. Our experimental results show that the algorithm only requires three to five iterations to converge to a solution with an error bound of $\epsilon = 10^{-2}$.

TABLE 1
Jacobi: Two-Dimensional Distribution, Predicted, and Measured Scalability

$\psi(p,p')$	$p'=4,\ n=64$		$p'=8,\ n=105$			$p'=16,\ n=161$		
	Pred.	Meas.	Pred.	Meas.	diff.	Pred.	Meas.	diff
p=4	1.000	1.000	0.718	0.738	2.7%	0.605	0.617	1.9 %
p=8			1.000	1.000	0%	0.842	0.819	2.7%
p=16						1.000	1.000	0%

TABLE 2
Jacobi: Column Distribution, Predicted, and Measured Scalability

$\psi(p,p')$	$p'=4$, $n=64$		$p'=8$, $n=105$			$p'=16$, $n=165$		
	Pred.	Meas.	Pred.	Meas.	diff.	Pred.	Meas.	diff
p=4	1.000	1.000	0.721	0.739	2.4%	0.576	0.581	0.8 %
p=8			1.000	1.000	0%	0.796	0.808	1.5%
p=16						1.000	1.000	0%

4 Automatic Performance Comparison under VFCS

We have implemented a prototype version of the iterative algorithm within VFCS for predicting the scalability and execution time of a parallelized code. The functionalities of P^3T and VFCS have been fully implemented as described in Section 2. Fig. 3 shows the structure of the scalability prediction within VFCS. The input program is parallelized, instrumented by VFCS, and a message passing code is generated. This code is then compiled and executed on the target parallel machine. A performance analysis tool analyzes the tracefile obtained and computes (initial) performance indices which are then used by scalability prediction. Finally, scalability prediction implements the iterative algorithm as described in Section 3. At each iteration of the algorithm, the problem size is specified, the source code is automatically parallelized, performance indices (number of transfers Z and the amount of data transferred D) are estimated by P^3T, and scalability prediction is performed. This process iterates until the algorithm converges.

Experimental results show that our approach provides an effective solution for capturing the scaling properties of a parallel code and supports optimizing data-parallel programs. Two cases are presented in detail in this section to illustrate how the iterative algorithm is used within the VFCS environment and how the prediction is carried out automatically.

The experiments have been carried out on an iPSC/860 hypercube with 16 processors. The parallel processing overhead T_o used in the scalability iteration algorithm, as described in Section 3, contains communication overhead and load imbalance. We choose two codes, Jacobi and Redblack, both of which contain several two-dimensional arrays and imply good load balance. T_o, therefore, contains only the communication time that can be obtained by the formula

$$T_o = Z(\rho + (\beta \cdot D) + \gamma \cdot h), \tag{6}$$

where Z and D are predicted at compile time for any problem size W using P^3T. The machine specific parameters, ρ and β, are the startup time and the transfer time per message byte, respectively. γ represents the additional overhead for each network hop and h is the number of hops.

Jacobi and Redblack have been parallelized by VFCS and their performance measured on four processors of an iPSC/860 hypercube. The performance indices obtained and needed for computing the initial state of the scalability prediction are given by the work W, the total execution time on p processors T_p, the computation time T_c, and the communication overhead T_o. The execution models of Jacobi and Redblack, based on (3), are as follows:

$$T_p = T_c + T_o = \frac{W}{p}\Delta + T_o = 11\frac{(n-2)^2}{p}\Delta + T_o$$

and

$$T_p = T_c + T_o = \frac{W}{p}\Delta + T_o = 6\frac{(n-1)^2}{p}\Delta + T_o.$$

We assume that the computations of Jacobi and Redblack are uniformly distributed across all processors.

The computing rate $\Delta = \frac{T_c \cdot p}{W}$ and the average speed $a = \frac{W}{p \cdot T_p(W)}$ can be determined by the measured computation time and total execution time. The initial value of the prediction algorithm, $W_0 = \frac{p' \cdot W}{p}$, is computed based on the

TABLE 3
Redblack: Two-Dimensional Distribution, Predicted, and Measured Scalability

$\psi(p,p')$	$p'=4$, $n=64$		$p'=8$, $n=124$			$p'=16$, $n=189$		
	Pred.	Meas.	Pred.	Meas.	diff.	Pred.	Meas.	diff
p=4	1.000	1.000	0.524	0.565	7.8 %	0.445	0.477	7.1 %
p=8			1.000	1.000	0%	0.851	0.846	0.5%
p=16						1.000	1.000	0%

TABLE 4
Jacobi 2D: Predicted and Measured Execution Times (in μs)

	Jacobi 2D		
	Pred.	Meas.	diff.
$p' = 4,\ n = 64$	-	1869	-
$p' = 8,\ n = 105$	2603	2532	2.7%
$p' = 16,\ n = 161$	3089	3066	0.7%

work W and $p = 4$. Starting with iteration $k = 0$, a new input data size $n_k = f^{-1}(W_k)$ is obtained for $k \geq 0$. The communication overhead T_o' and the scaled work W_k' are predicted using (6) and (4), respectively. Scalability from processors p to processors p' is determined when the terminating condition $||W_k - W_{k-1}|| < \epsilon$ is satisfied for a fixed $\epsilon > 0$ ($\epsilon = 10^{-2}$ is used in our experiments). Otherwise the method iterates with the new parameter n_{k+1}.

Tables 1 and 2 show the measured and predicted scalability of Jacobi algorithm with two different data distribution strategies: two-dimensional block distribution and column-wise distribution of all program arrays to two-dimensional and one-dimensional processors array, respectively. The difference in percentage between the predicted and measured values is given in the third column of the tables.

The experimental results confirm that our predicted scalabilities are very accurate and the variations of scaled performance for various data distributions are also captured.

Table 3 shows the predicted and measured scalability values of the Redblack algorithm with two-dimensional distribution. Tables 4, 5, and 6 present the predicted execution times versus the measured ones for Jacobi with two-dimensional block distribution, one-dimensional distribution, and Redblack with two-dimensional block distribution, respectively.

The initial problem size used in Tables 1, 2, 3, 4, 5, 6 is determined by the asymptotic speed [24] for best performance, where $n = 64$ is chosen. We have measured the average execution time required for a single iteration (covering parallelization, P^3T, and scalability prediction) of Fig. 3. For Redblack, the parallelization time accounts for 0.7 secs, P^3T for 0.3 secs, and scalability prediction for 0.1 secs. Overall, every iteration took approximately 1.1 secs,

TABLE 5
Jacobi C: Predicted and Measured Execution Times (in μs)

	Jacobi C		
	Pred.	Meas.	diff.
$p' = 4,\ n = 64$	-	1711	-
$p' = 8,\ n = 105$	2373	2313	2.5%
$p' = 16,\ n = 164$	2971	2944	0.9%

TABLE 6
Redblack 2D: Predicted and Measured Execution Times (in μs)

	Redblack 2D		
	Pred.	Meas.	diff.
$p' = 4,\ n = 64$	-	5560	-
$p' = 8,\ n = 124$	10611	9840	7.2%
$p' = 16,\ n = 189$	11324	11641	2.7%

which remains constant for changing problem size. The execution time of Redblack can be written as $\alpha T_4(64) = \alpha 1869 = 5560\ \mu s$ and for Jacobi $T_4(64) = 1869\ \mu s$ where α is 2.975. According to Tables 1, 2, and 3, the scalability of Jacobi is higher than that of Redblack. Therefore, by Theorem 1, the smaller initial execution time and larger scalability show that Jacobi scales better than Redblack, which is confirmed by measured results as given in Tables 4, 5, and 6.

A more interesting result is given by the two different Jacobi versions. From Tables 1 and 2, we can see that the 2D distribution implementation has a larger initial execution time and a better scalability, on $p = 16$, than that of column-wise distribution. According to Theorem 2, there will be a crossing point at some scaled system size p'. However, in this case, the crossing point is greater than 16 and cannot be confirmed by our prototype implementation. Fig. 4 shows that there is no crossing point for range of 4 to 16 processors.

As pointed out in [21], scaled performance is more sensitive for small applications, where increasing system size will cause more noticeable change of communication/computation ratio. For Jacobi, the communication/computation ratio increases with the decrease of problem size. At the initial state, where $p = 4$ and $n = 20$, the execution time for Jacobi with column-wise distribution strategy is given by $T_4(20) = 594\ \mu s$ and for Jacobi with 2D distribution it is $\alpha T_4(20) = 753\ \mu s$, where $\alpha = 1.267$. Considering the scalability results of Tables 7 and 8, we see that, for $p' = 8$, the

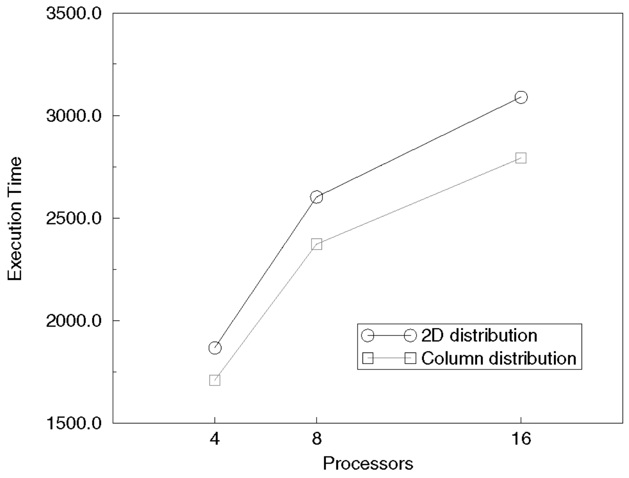

Fig. 4. No scaled crossing point for Jacobi with starting point $p = 4$, $n = 64$.

TABLE 7
Predicted Scalability for Jacobi with 2D Distribution

$\psi(p,p')$	$p' = 4$, $n = 20$	$p' = 8$, $n = 33$	$p' = 16$, $n = 50$
p=4	1.000	0.652	0.548
p=8		1.000	0.840
p=16			1.000

TABLE 8
Predicted Scalability for Jacobi Column-Wise Distribution

$\psi(p,p')$	$p' = 4$, $n = 20$	$p' = 8$, $n = 43$	$p' = 16$, $n = 64$
p=4	1.000	0.373	0.333
p=8		1.000	0.893
p=16			1.000

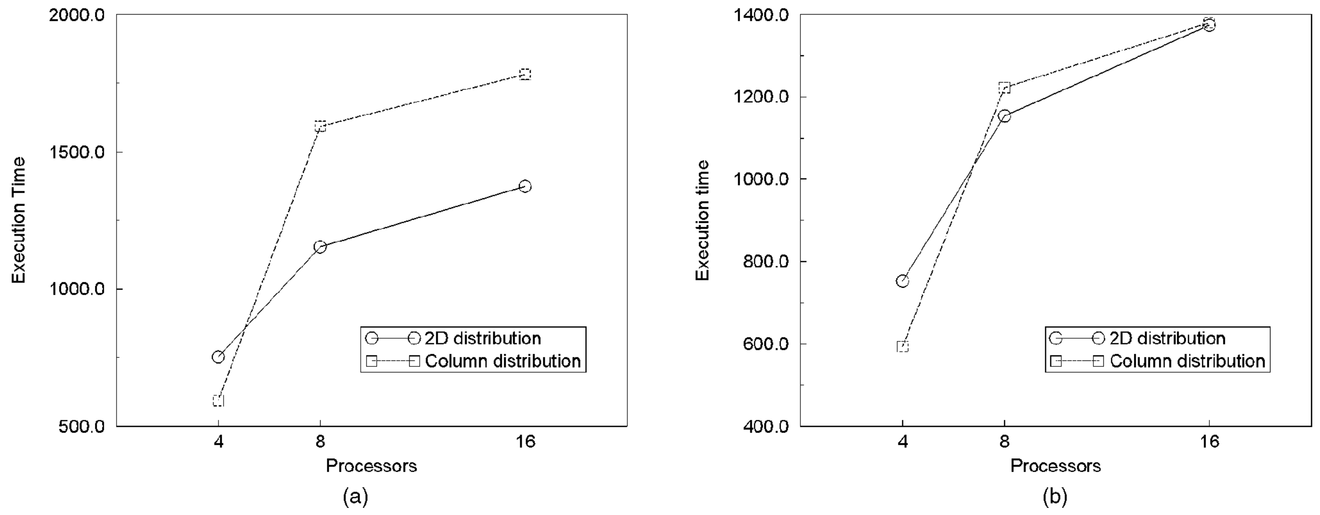

Fig. 5. Scaled crossing point (a) and crossing point (b) for the Jacobi with $n = 20$.

2D distribution scales better than that of column-wise distribution. The ratio between the two predicted scalabilities, $\frac{0.652}{0.373} = 1.747$, is greater than α. Therefore, by Definition 2, $p' = 8$ is a crossing point where the execution time of 2D distribution becomes less than that of column-wise distribution. This crossing point is due to the communication behavior involved on iPSC/860 for $p' = 8$ and is confirmed by the measured execution times as shown in Fig. 5a.

In order to verify Theorem 3, we measured both codes with $n = 33$ and $n = 50$, respectively. In accordance with Theorem 3, 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 Fig. 5b.

In this study, our focus is on range comparison under data-parallel compilation systems. More experimental results of range comparison can be found in [25] for MPI applications.

5 Conclusion

There are many ways to parallelize a program, and the relative performance gain of different parallelizations strategies varies with problem size and system size. Comparing the performance of different implementations of an algorithm over a range of system and problem sizes is crucial in developing effective parallelizing compilers and ultimately in reducing the burden of parallel programming. In this study, a practical methodology is developed for automatic range comparison and it is tested in a data-parallel compilation system. The proposed methodology is built on rigorous analytical models which are both correct and efficient. Experimental results confirm its effectiveness as part of a parallelizing compiler.

This paper offers several contributions. First, we identify the importance and feasibility of range comparison in data-parallel compilation systems; next, an iterative algorithm is developed to experimentally predict the scalability of algorithm-machine combinations and to enable automatic range comparison. P^3T, an existing static estimator, is modified to integrate automatic range comparison into a data-parallel compilation systems. Finally, the range comparison approach is tested as part of Vienna Fortran Compilation System. Our experimental results demonstrate the feasibility and high potential of range comparison in a parallelizing compiler.

The concept and analytical results given in Sections 3.1 and 3.2 are very general. They are applicable to any algorithm-machine combinations. The scalability prediction algorithm given in Section 3.3 assumes that the workload is a deterministic function of a scaling factor n. While this assumption is quite reasonable, the algorithm requires an estimation of the parallel processing overhead. The algorithm has been tested with P^3T static performance estimator under Vienna Fortran Compilation System. Due to the availability of VFCS and P^3T, the experimental results presented in this paper are limited on the 16-node iPSC/860 available at University of Vienna. The integrated range comparison methodology introduced in this research, however, is general. It can be adopted for large parallel systems, as well as for other advance compilation systems [25].

ACKNOWLEDGMENTS

Xian-He Sun's research was supported in part by the U.S. National Science Foundation under Grant ASC-9720215 and by the Louisiana Eduction Quality Support Fund.

REFERENCES

[1] T. Lewis, "The Next $10,000_2$ Years," *Computer*, pp. 64-70, May 1996.

[2] B. Chapman, P. Mehrotra, and H. Zima, "Programming in Vienna Fortran," *Scientific Programming*, vol. 1, pp. 31-50, 1992.

[3] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, "Fortran D Language Specification," Technical Report COMP TR90079, Dept. of Computer Science, Rice Univ., Mar. 1991.

[4] H.P. Fortran Forum, "High Performance Fortran Language Specification Version 1. 0," Technical Report, Dept. of Computer Science, Rice Univ., May 1993.

[5] S. Benkner, S. Andel, R. Blasko, P. Brezany, A. Celic, B. Chapman, M. Egg, T. Fahringer, J. Hulman, Y. Hou, E. Kelc, E. Mehofer, H. Moritsch, M. Paul, K. Sanjari, V. Sipkova, B. Velkov, B. Wender, and H. Zima, *Vienna Fortran Compilation System—Version 2. 0—User's Guide*, Oct. 1995.

[6] V.S. Adve, J.M. Crummey, M. Anderson, K. Kennedy, J.-C. Wang, and D.A. Reed, "An Integrated Compilation Performance Analysis Environment for Data Parallel Programs," *Proc. Supercomputing*, San Diego, Calif., Dec. 1995.

[7] T. Fahringer, *Automatic Performance Prediction of Parallel Programs.* Boston: Kluwer Academic, Mar. 1996.

[8] T. Fahringer, "Estimating and Optimizing Performance for Parallel Programs," *Computer*, vol. 28, pp. 47-56, Nov. 1995.

[9] T. Fahringer and E. Mehofer, "Buffer-Safe Communication Optimization Based on Data Flow Analysis and Performance Prediction," *Proc. 1997 Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '97)*, pp. 189-200, San Francisco, Nov. 1997.

[10] M. Gupta, E. Schonberg, and H. Srinivasan, "A Unified Framework for Optimizing Communication in Data-Parallel Programs," *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 7, pp. 689-704, July 1996.

[11] K. Kennedy and A. Sethi, "A Communication Placement Framework with Unified Dependence and Data-Flow Analysis," *Proc. Third Int'l Conf. High Performance Computing*, Trivandrum, India, Dec. 1996.

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

[13] T. Fahringer and H. Zima, "A Static Parameter Based Performance Prediction Tool for Parallel Programs," *Proc. ACM Int'l Conf. Supercomputing*, pp. 207-219, Tokyo, 1993.

[14] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors*, vols. 1 and 2. Englewood Cliffs, N.J.: Prentice Hall, 1988.

[15] J. Gustafson, G. Montry, and R. Benner, "Development of Parallel Methods for a 1024-Processor Hypercube," *SIAM J. Scientific and Statistical Computing*, vol. 9, pp. 609-638, July 1988.

[16] X.-H. Sun and D. Rover, "Scalability of Parallel Algorithm-Machine Combinations," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, no. 6, pp. 599-613, June 1994.

[17] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing, Design and Analysis of Algorithms.* Benjamin/Cummings, 1994.

[18] S. Sahni and V. Thanvantri, "Performance Metrics: Keeping the Focus on Runtime," *IEEE Parallel and Distributed Technology*, pp. 43-56, Spring 1996.

[19] X.-H. Sun, "The Relation of Scalability and Execution Time," *Proc. Int'l Parallel Processing Symp.*, Apr. 1996.

[20] X.-H. Sun, "Performance Range Comparison Via Crossing Point Analysis," *Lecture Notes in Computer Science*, J. Rolim, ed., vol. 1388. Springer Mar. 1998.

[21] X.-H. Sun and J. Zhu, "Performance Prediction: A Case Study Using a Scalable Shared-virtual-memory Machine," *IEEE Parallel and Distributed Technology*, pp. 36-49, Winter 1996.

[22] X.-H. Sun, "Scalability versus Execution Time in Scalable Systems," Technical Report TR-97-003, Louisiana State Univ., Dept. of Computer Science, 1997 (revised May 1998).

[23] X.-H. Sun, M. Pantano, and T. Fahringer, "Integrated Range Comparison for Data-parallel Compilation Systems," Technical Report #97-004, Dept. of Computer Science, Apr. 1997.

[24] X.-H. Sun and J. Zhu, "Performance Considerations of Shared Virtual Memory Machines," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, no. 11, pp. 1,185-1,194, Nov. 1995.

[25] M. Noelle, M. Pantano, and X.-H. Sun, "Communication Overhead: Prediction and Its Influence on Scalability," *Proc. Int'l Conf. Parallel and Distributed Processing Techniques and Applications*, July 1998.

Xian-He Sun (S'88-M'90-SM'95) received the BS degree in mathematics from Beijing Normal University, Beijing, China, and the MS degree in mathematics and MS and PhD degrees in computer science from Michigan State University. After graduating from Michigan State, he joined the Ames Laboratory, operated for the Department of Energy by Iowa State University. He was a staff scientist at ICASE, NASA Langley Research Center from 1992 to 1993. Since January 1994, he has been with the Department of Computer Science, Louisiana State University. His research interests include parallel and distributed processing, performance evaluation, high performance computing, and software systems. Dr. Sun was a guest editor for the *Journal of Parallel and Distributed Computing* and the *Journal of Supercomputing*. He is on the editorial board of the *Journal of Performance Evaluation and Modeling for Computer Systems* and has served as chairman and program committee member for a number of international conferences and workshops. He is a member of the IEEE, ACM, New York Academy of Sciences, and Phi Kappa Phi.

Mario Pantano received the Laurea degree in computer science from the University of Milan, Italy, in 1990 and the PhD 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 System, 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.

Thomas Fahringer received a Masters degree in 1988 and a PhD in 1993, both in computer science, from the Technical University of Vienna, Austria. He was a visiting scientist at the Engineering Design Research Center at Carnegie Mellon University from 1988-1990. Since 1990, he has been an assistant professor of computer science and, since 1998, associate professor, at the Institute for Software Technology and Parallel Systems, University of Vienna. His research focuses on software tools for parallel programs and multiprocessor architectures, in particular, parallelizing compilers, symbolic program, and performance analysis.