Software Support for Multiprocessor Latency Measurement and Evaluation

Yong Yan, *Member*, *IEEE*, Xiaodong Zhang, *Senior Member*, *IEEE*, and Qian Ma

Abstract—Parallel computing scalability evaluates the extent to which parallel programs and architectures can effectively utilize increasing numbers of processors. In this paper, we compare a group of existing scalability metrics and evaluation models with an experimental metric which uses network latency to measure and evaluate the scalability of parallel programs and architectures. To provide insight into dynamic system performance, we have developed an integrated software environment prototype for measuring and evaluating multiprocessor scalability performance, called Scale-Graph. Scale-Graph uses a graphical instrumentation monitor to collect, measure and analyze latency-related data, and to display scalability performance based on various program execution patterns. The graphical software tool is X-window based and currently implemented on standard workstations to analyze performance data of the KSR-1, a hierarchical ring-based shared-memory architecture.

Index Terms—Latency analysis, parallel computing scalability, performance graphical presentation, software tools, KSR multiprocessors.

1 Introduction

When we evaluate the performance of a parallel program on a parallel machine, we are often interested in knowing its *speedup*, which is defined as the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with multiple processors. Speedup quantitatively measures execution performance gain in terms of execution time reduction achieved by parallelizing the program over a sequential implementation. Another basic performance concept is *efficiency*, which is defined as the ratio of speedup to the number of processors. Efficiency measures the percentage of execution time for which each processor is effectively used. Both speedup and efficiency give direct and simple performance measures, but provide little insight into overhead patterns of programs and systems. In addition, many real-world application programs are scalable with respect to their data sizes and other input parameters. Meanwhile, computer architects are designing and building scalable parallel systems with many processors to solve large scalable application problems. Speedup and efficiency are certainly not satisfactory metrics to study scaling performance of programs and architectures. Parallel computing *scalability* measures the ability of a parallel architecture to increase the number of processors for solving an application problem of an increasing size. The performance gain from scaling both the size of the problem and of the architecture is primarily affected by overhead patterns inherent in the parallel algorithm and in the effects of the architectural interconnection network. Therefore parallel computing scalability also studies architecture scalability, which is related to the bottlenecks inherent in an architecture design, and it studies algorithm scalability which is related to the parallelism inherent in an algorithm design.

The scalability analysis of parallel computing is important and useful for different purposes. For an application programmer or a scientific library designer, scalability analysis can help him/her to identify algorithm and implementation bottlenecks, and to select an optimal algorithm and architecture combination for best utilizing an increment of the number of processors. For an architecture designer, a scalability study investigates overhead sources inherent in interconnection networks. It provides a better understanding of the nature of scientific computations so as to facilitate the building of scalable parallel architectures. The scalability study also plays an important role in parallel computing performance evaluation and prediction, because it can predict the performance of large problems on large systems based on the performance of small problems on small systems if we understand well the scaling behavior of programs [12].

A rigorous scalability definition and metric provides an important guideline for precisely understanding the nature of scalability, and for effectively measuring scalability in practice. Isoefficiency [6] and Isospeed [11] are two useful scalability metrics. The former evaluates the performance of an algorithm-machine combination through modeling an isoefficiency function. The latter evaluates the performance of an algorithm-machine combination through measuring the workload increment with a change of the machine size under the condition of the isospeed. Isoefficiency is considered an analytical method for algorithm scalability evaluation. Although the isospeed metric is an experimental metric, it may be unable to measure certain real machine factors in practice. In this paper, we give an overview of an ex-

- *Y. Yan and X. Zhang are with the High Performance Computing and Software Laboratory, Division of Computer Science, University of Texas at San Antonio, San Antonio, TX 78249. E-mail: zhang@ringer.cs.utsa.edu.*
- *Q. Ma is with The Computer Systems Advisors Group, Beijing Automated Systems Co. Ltd., 100027, Beijing, Peoples Republic of China.*

Manuscript received Mar. 24, 1995; revised July 11, 1996.
Recommended for acceptance by K. Marzullo.
For information on obtaining reprints of this article, please send e-mail to: transse@computer.org, and reference IEEECS Log Number S95716.

perimental metric using network latency for measuring and evaluating parallel program and architecture scalability [16]. We also compare and evaluate merits and limits of existing scalability metrics and models. Computer graphical presentation software has become a common and standard system environment in workstations and in many parallel computers, allowing us to track multiple complex visual patterns and to easily spot anomalies in these patterns. We present a graphical presentation tool to aid latency analysis for parallel computing scalability evaluation.

In the process of evaluating scalability using a metric, different models may be applied in the parallel program execution measurement. The major models include the *problem size bound model*, the *memory-bound model*, the *time-bound model*, and the *efficiency-conserved model*. Here we briefly introduce these measurement models.

Problem Size Bound Scaling Measurement. This basic measurement method quantitatively observes the execution performance changes of a fixed size problem as the number of processors increases. Amdahl's Law roughly models the performance of this measurement,

$$t(n) = t_s + \frac{t_p}{n}, \tag{1}$$

where the execution time $T(n)$ is divided into two parts: t_s is the amount of time spent on serial parts of a program, t_p is the amount of time spent on parts of the program that can be executed in parallel, and n is the number of processors used for the execution. This model indicates that as the number of processors is increased, the parallel execution time of an application is bounded by t_s. In practice, the execution time eventually hits a minimum, after which adding processors can only cause the program to take a longer time to complete, because the overhead portion of the program, t_s, tends to increase with the number of processors. Three serious limitations of this approach are: 1) the complex overhead pattern inherent in the program and the architecture may not be explicitly and precisely evaluated by the term t_s in (1), 2) the model assumes the computation load is balanced, and 3) the model cannot be used to evaluate the performance when the problem is scaled. However, this simple measurement may be used to determine the optimal number of processors to achieve the maximum possible speedup.

Memory-Bound Scaling Model. Scaling the size of a problem is an important function in large-scale scientific computations. The memory-bound scaling measurement scales the problem so that the computation load and memory allocation in each processor is large enough to achieve good parallel performance. The limitation of this approach is that the amount of execution time and memory allocation may be unacceptably large in practice. An example of this scaling method is found in [7]. We will study other limitations of this model later in the paper.

Time-Bound Scaling Model. This method scales the problem so that the execution time is kept constant as the number of processors increases [7]. This approach constrains the execution time but may not apply to all parallel program cases in practice. In later sections, we will also show another limitation of this model in measuring scalability in practice.

Efficiency Conserved Model. This method scales the problem so that the parallel computing efficiency is kept constant as the number of processors is increased. The model is the foundation of the isoefficiency function [6]. We will show that this model is fair and reasonable in measuring scalability.

So far, most scalability studies have been done by considering the input data size of a problem as the only problem size parameter. In practice, the growth and scaling of an application problem is more complicated. In many large scientific simulations of physical phenomena, more than one parameter is used to change the size of the problem. The execution behavior may be significantly different between a program with a single input parameter and a program with multiple input parameters, due to different program execution structures. In addition, the ways of changing multiple input parameters can significantly affect scaling of execution characteristics. In this paper we present an application program with multiple input parameters for our latency analysis and scalability study.

2 Scalability Metrics from Efficiency and Speed

The definition of scalability comes from Amdahl's law, which is tied to efficiency and speedup. Here, we first introduce two scalability metrics: the *isoefficiency* function based on parallel computing efficiency [6], and the *isospeed* metric based on parallel computing speed [11]. The isoefficiency function of a parallel system is determined by abstracting the size of a computing problem as a function of the number of processors, subject to maintaining a desired parallel efficiency (between 0 and 1).

The isoefficiency function proposed by Grama, Gupta, and Kumar [6] first captures, in a single expression, the effects of characteristics of the parallel algorithm as well as the parallel architecture on which it is implemented. In addition, the isoefficiency function shows that it is necessary to vary the size of a problem on a parallel architecture as the machine size scales so that the processing efficiency of each processor can remain constant. However, there are two limits by using this metric to evaluate scalability. First, analytical forms of the program and architecture overhead patterns in a shared-memory architecture may not be as easy to model as in a distributed memory architecture. This is mainly because the computing processes involved in a shared-memory system consist of process scheduling, cache coherence, and other low level program and architecture dependent operations which are more complicated than message-passing on a distributed memory system. It would be difficult to use the isoefficiency metric to precisely evaluate scalability for a program running on a shared-memory system in practice. Second, the metric may not be used to measure scalability of the algorithm-architecture combination through machine measurements. We believe this metric is more appropriate to evaluate the scalability of parallel algorithms.

Sun and Rover [11] take another approach to algorithm-machine combinations. Their metric starts by defining an average unit speed. Scalability is defined as an average increase of the amount of work on each processor needed to

keep its speed constant when the size of the parallel architecture increases from N processors to N' processors. This metric provides more information about architectures and programs. However, we believe there are two limits in the isospeed metric for precisely measuring and evaluating the scalabilities of the application program and the architecture. First, some nonfloating point operations can cause major performance changes. For example, a single assignment to a shared variable in a cache coherent shared-memory system may generate a sequence of remote memory/cache accesses and data invalidations. But this type of operation is excluded in the measurement of scalability. Second, the latency is included in the total execution time in the metric, but it is not defined in the amount of work, W, in the scalability metric. In practice, the execution overhead caused by the interconnection network and the program structure is a function of the problem size. Since many multiprocessor systems have provided hardware and/or software monitors for users to precisely trace program executions, it is possible to evaluate scalability by using lower level architecture effects and program overhead patterns.

3 Overview of the Latency Metric

We define the latency metric through a series of formal definitions and theorems. For detailed analyses of the metric, the interested readers may refer to [16].

Definition 1. *The parallel computing time of an algorithm implementation, denoted by T_{para}, is the elapsed time between starting the program and ending the program on a parallel architecture. The parallel execution time on the ith processor, denoted as T_i, for $i = 1, \ldots N$, is the effective execution time in the processor, where N is the total number of processors used in the computation. (See Fig. 1). The effective execution time in each processor is the time spent to do calculations and to handle network communications, which includes the latency time during execution (see Definition 2 for latency) but does not include the idle time of waiting for the start of execution and the idle time of waiting for the program to end.*

Definition 2. *The overhead latency in the ith processor, denoted by L_i for $i = 1, \ldots, N$, is the sum of the total number of idle time units during the execution in the processor and the time units spent on the work which is not needed in a sequential computer, such as synchronization time, communication time, and thread creation time, etc.*

Definition 3. *The size of a problem, denoted by W, is a measure of the number of basic operations needed by the fastest known sequential algorithm to solve the problem on a sequential computer. In general, the average time of a basic machine operation can be considered as a constant, denoted by t_c. For example, we can use the cycle time of the CPU to be the basic operation time t_c. Therefore, the total sequential computation time of a problem with size of W is Wt_c.*

Definition 4. *The average latency, denoted by L(W, N), is a function of the problem size W and the number of processors used N, and is defined as the average amount of overhead time needed for each processor to complete the assigned work:*

$$L(W, N) = \frac{\sum_{i=1}^{N} (T_{para} - T_i + L_i)}{N}, \tag{2}$$

Fig. 1 provides an example of latency and execution time distributions given in Definitions 1, 2, 3, and 4. The overhead in each processor consists of idle times and network latency which cover the total overhead. The accumulation of the total overhead in each processor divided by the number of processors (N) is defined as the average overhead in (2). Using the average latency should be fair in comparison with using the minimum or the maximum latency.

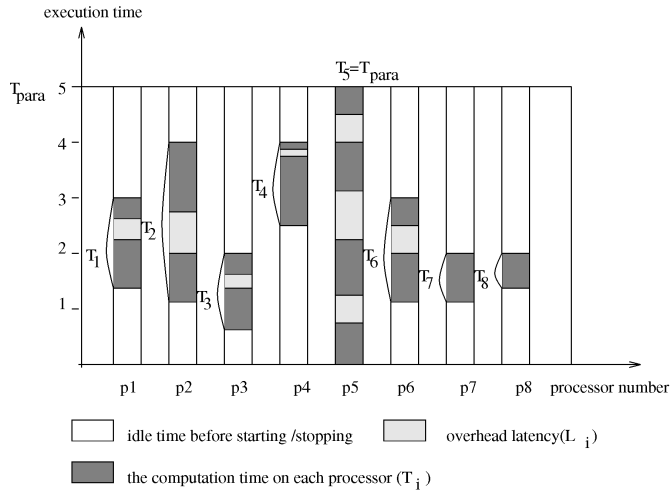


Fig. 1. An example of latency and execution time distributions given in Definitions 1, 2, 3, and 4.

Definition 5. *For a given efficiency, $E \in [0, 1]$ of running a program on N processors, the implementation-machine combination is called scalable if and only if the efficiency of an implementation of an algorithm on a given machine can become equal to or greater than the given E by increasing the size of the problem.*

Definition 5 indicates that an algorithm-machine combination is scalable if and only if we can find a scalable implementation of the algorithm on the machine.

Before formally defining the scalability latency metric, we describe the the parallel computing time, T_{para} based on Amdahl's Law:

$$T_{para} = \frac{Wt_c}{N} + L(W, N), \tag{3}$$

where W is the size of a problem, N is the number of the processors, t_c is the average computing time per operation in the system, and $L(W, N)$ is the average latency time.

Definition 6. *For a given algorithm implementation on a given machine, let $L_E(W, N)$ be the average latency when the algorithm for solving a problem of size W on N processors, and let $L_E(W', N')$ be the average latency when the algorithm for solving the problem of size of W' on $N' > N$ processors. If the system size changes from N to N', and the efficiency is kept to a constant $E \in [0, 1]$, the scalability latency metric* is *defined as*

$$scale(E, (N, N')) = \frac{L_E(W, N)}{L_E(W', N')}. \tag{4}$$

We also call the metric in (4) an *E*-conserved scalability because the efficiency is kept constant. From the efficiency definition, (4) satisfies the following E-conserved condition:

$$\frac{Wt_c}{N(\frac{Wt_c}{N} + L_E(W, N))} = \frac{W't_c}{N'(\frac{W't_c}{N} + L_E(W', N'))}.$$

In practice, the value of (4) is less than or equal to 1. A large scalability value of (4) means small increments in latencies in the program execution, and hence the computation on the parallel system is considered highly scalable. On the other hand, a small scalability value means large increments in latency and therefore the computation on the system is poorly scalable.

We have shown that the latency scalability metric is analytically equivalent to both the isospeed function and the isoefficiency metric, and covers their scalability measures [16]. However, in practice, the latency measurement $L(W, N)$ provides more precise overhead effects of the architecture and the program structure.

4 Scale-Graph Software Structure

4.1 Overview

The Scale-Graph (Scalability-Graph) environment was constructed based on our experience and on existing graphical presentation tools implemented on the BBN Butterfly systems [14], and on the CM-5 [5]. There are two related graphical presentation tools for parallel and distributed computing performance evaluation. PICL, a Portable Instrumented Communication Library and ParaGraph, a graphical presentation environment, both developed at the Oak Ridge National Laboratory [8] are software monitor and graphical presentation tools mainly monitoring and presenting message-passing events. Since the PICL was originally built for the hypercube architecture, which solely supports a message passing paradigm, the events of interest focus on communications among processor nodes. Two other graphical presentation tools, Xab (X-window Analysis and deBugging), and XPVM, [1], [2], [3] have been developed for run-time monitoring of PVM programs on networks of workstations. In such systems, PVM calls are instrumented so that they cause debugging information to be generated. This information may be saved to a file or displayed in an X-window. Another tool for massively parallel systems is the Pablo performance analysis environment [10], which contains both a portable performance data analysis environment and portable source code instrumentation and data capture. This tool provides the basis for the performance analysis tools on the Intel Paragon XP/S, a large scale distributed memory architecture. In contrast, Scale-Graph was developed to particularly collect and graphically present latency information for scalability study on both shared-memory and distributed memory systems. Similarly, Scale-Graph is built in a X-window supported workstation system.

The hardware facilities supporting Scale-Graph are divided into two parts: a graphics supported workstation is used for data analysis and graphical presentation, and a target parallel architecture for program execution. The Scale-Graph software system is composed of seven parts:

1) a visual interface for accepting user requests and performance presentation,
2) a user request analyzer,
3) a workstation / parallel machine communication interface,
4) an instrumentation preprocessor for inserting measurement code in source programs,
5) a trace routine library for providing required trace functions and procedures,
6) an execution I/O controller for parallel job submissions and performance data collections, and
7) a data analyzer for event analysis.

Fig. 2 describes the software structure of Scale-Graph. A user request packet including an application program, a scaling method, and a target parallel machine is submitted to the Scale-Graph system through the *visual interface*. After the request is processed by the *user request analyzer*, the program is submitted to the target parallel machine through the *workstation/parallel machine communication interface*. The program becomes an instrumented program after going through the *instrumentation preprocessor*. The instrumented program executed by the target parallel machine outputs performance data collected by the *execution I/O controller*. Event measurement is actually done when a program runs with inserted tracing code and is linked with the *trace routine library*. The performance data are sent back to the workstation and are studied by the *data analyzer*. Finally the analyzed data are presented by the *visual interface* on the workstation.

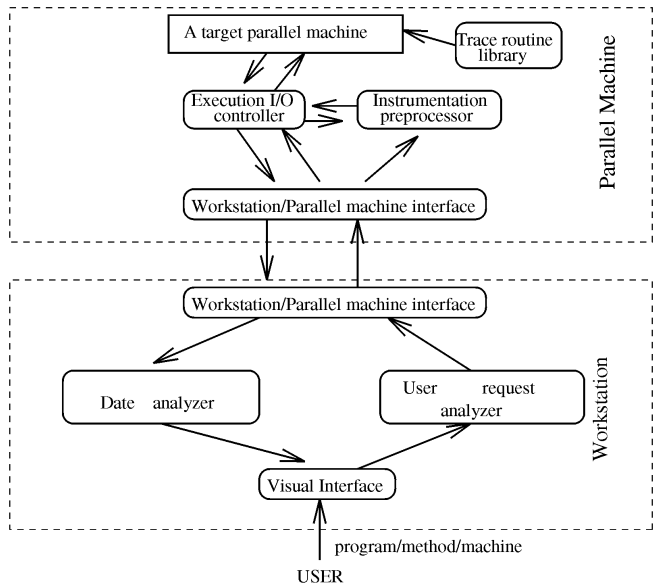


Fig. 2. The software structure of Scale-Graph.

The instrumentation preprocessor is based on software monitoring or hardware monitoring for various latency related measurements. In software monitoring, the instrumentation instructions are inserted in the program at compile time for collecting performance data. The software preprocessor is a syntax analyzer of a high level language. Its input is a parallel program in a high level language, such

as C. Its output is an instrumented program in which trace codes have been implanted. Compiler generator tools Lex and Yacc have been used to build the preprocessor. For detailed implementation of the software instrumentation and overhead reduction, the interested reader may refer to [14], [5].

We have also used a hardware monitor, called *Pmon*, built into the KSR-1 system as a preprocessor for collecting performance data. Each KSR-1 processor contains an event monitor unit (EMU) designed to log various types of local cache events and intervals. The job of the EMU is to count events and elapsed time related to cache system activities. The hardware monitored events provide a set of relatively precise and important data to be used for evaluating the execution performance on the KSR-1. Since the hardware monitor is a major part for data collection, the overhead of the latency measurement and analysis tool is considered relatively low.

4.2 Scale-Graph Software Functions

Fig. 3 gives the function structure of Scale-Graph. There are three major functions supported in this software tool: 1) execution pattern analysis of the sequential program, 2) latency measurement for scalability performance evaluation using the program size bound model, and 3) latency measurement for scalability evaluation using scaling methods that increase both the size of the program and of the system.

4.2.1 Execution Pattern Analysis of Sequential Programs

To understand structures and execution patterns of a program in its sequential form is an important step to study parallel scalability. The sequential program is the original version of the program which runs on a single processor. Scale-Graph provides the following three functions for such a performance study:

- Execution profile of the sequential program, to understand the execution time distributions among the program routines.
- *CPU utilizations* (the ratio between the CPU busy time and the total execution time) of the sequential program using different scaling methods.
- *Memory access rates* (the ratio between the memory access time and the total execution time) of the sequential program using different scaling methods.
- *Memory access miss rate* (the ratio between the number of memory misses and the total number of memory accesses) of the sequential program using different scaling methods.

4.2.2 Latency Evaluation on a Program of Fixed Size

Using this basic measurement, we can observe the changes of speedups and latencies of a fixed size program running on a system with an increasing number of processors. In addition, we use the latency performance data to predict parallel program scalability. This Scale-Graph function provides the following three types of performance data:

- Program execution time measurements on different numbers of processors.
- Measurements and presentations of various types of latency.
- Upper bound latency identifications for predicting program scalability.

4.2.3 Latency Evaluation Using Other Scaling Methods

This function is a major interest of the Scale-Graph tool, which studies and evaluates the computing scalabilities as both the size of the program and of the parallel system are increased. We focus on the following measurements and performance issues:

- Scalability evaluation using memory bound scaling methods.
- Scalability evaluation using time bound scaling methods.
- Scalability evaluation using the latency metric.
- Measurement and evaluation of latency sources inherent in the program and the architecture.
- Program execution patterns.
- Program memory access patterns.

5 An Application Program and Its Scaling Methods

5.1 The Simulation

The application we used for testing the scalability metric and its software environment is a discrete event and time simulation program of a collection of billiard balls constrained to move in one dimension [4]. The billiard balls are used to simulate the hard-sphere liquid. This is a common and accurate model for many types of liquids, particularly liquid metals, where atom motion is simulated.

Initially, the balls are placed in predetermined positions with predetermined velocities. The activity of the system is a sequence of events, each event being a collision between two of the balls. The time duration of a collision is so small that any collision involves exactly two balls. If two pairs happen to collide at exactly the same time, the events can be executed one after another, as if one collision took place an instant after the other. Another dimension of the physical system is the

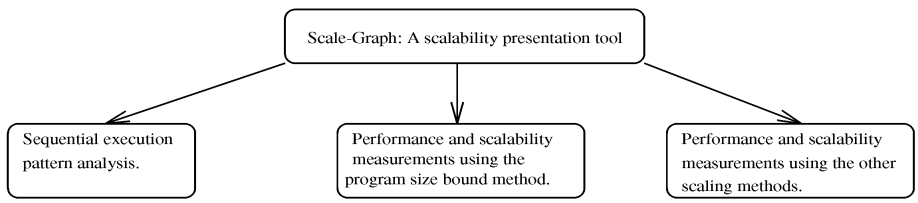


Fig. 3. The software functions of Scale-Graph.

time sequence. Each neighboring pair of balls can be characterized by a time-to-collision, which is the time at which the next collision between those balls occurs.

Two major input parameters to adjust the size of the problem include the size of the event sequence (the number of balls) and the size of the time sequence (the number of steps). Besides not scaling (fixed problem size), and scaling in a single dimension (either the number of balls or the number of the steps), the problem size can also be linearly scaled based on a ratio of the number of balls and the number of steps; and can be nonlinearly scaled based on a function of the two parameters. In addition, there are many other input parameters of scaling the problem in the simulation, such as the length of the moving space, the diameters of the balls, initial positions of the balls, and the initial speed of the balls. Fig. 4 presents the multidimension scaling feature of the program.

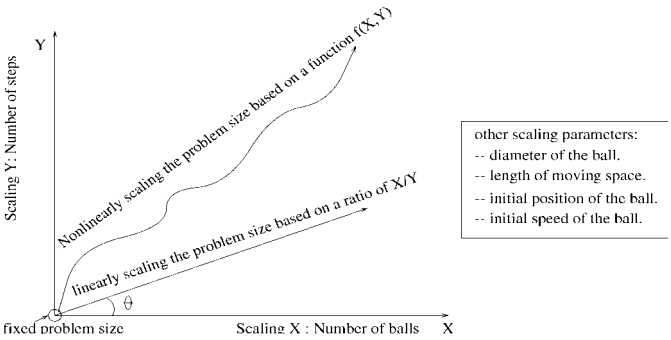


Fig. 4. Multidimensional scaling of the simulation program.

5.2 Program Structure and Its Parallel Implementation

In the implementation, the length of the moving space may be fixed, and the initial distribution of positions and speeds of the balls may be set in a uniform pattern. Because the relative positions of balls in the simulation do not change, each ball is identified by an ordered integer number, starting 0 from the left to the right in a vector. The sequential simulation algorithm is outlined in the following procedure.

Do (*n* steps)

1) Find the pair of balls (*i*, *i* + 1) which have the smallest time-to-collision in the event queue. Let *dt* be the time interval at the end of which the pair collides.

   $$dt = tcol - tnow,$$

   where *tcol* is the time-to-collision of the pair and *tnow* is the current time.
2) Moving colliding balls *i* and *i* + 1 in a straight line. Let the time at which the last collision of ball *i* occurred be *last_update_time*(*i*). Then, if the velocity of the *i*th ball is *v*(*i*), they move the *i*th ball a distance:

   $$v(i) * (tnow - last\_update\_time(i) + dt),$$

   and move ball *i* + 1 a distance:

   $$v(i + 1) * (tnow - last\_update\_time(i + 1) + dt).$$

   After this, ball *i* and ball *i* + 1 will be just touching.
3) Update *last_update_time*, i.e.,

   $$last\_update\_time(i) = last\_update\_time(i + 1) = tnow + dt.$$
4) Interchange the velocities of the two colliding balls (or, if the ball is bouncing off the wall, reverse its velocity).
5) Remove all collisions involving balls *i* and *i* + 1 from the event queue. (Because balls *i* and *i* + 1 collide, the time-to-collision of ball *i* with ball *i* – 1, and of ball *i* + 1 with ball *i* + 2, will be different than the ones in the event queue. Note that balls *i* and *i* + 1 can no longer collide with each other in the next event).
6) Compute the new time-to-collision items of *i* and *i* + 1 with their neighbors, and put the new time-to-collision items in the event queue.

End_Do

The major work involved in parallelizing the simulation is to determine independent events (the collisions) on a multiprocessor. The parallel simulation is defined by the following factors:

- *The input parameters*. The number of balls to be simulated (*num_balls* ≥ 1), and the steps of the computation (*num_of_steps* ≥ 1) which is used to control the precision of the simulation.
- *Memory requirement*. $M = C \times num\_balls$, where *C* is a constant dependent on implementation data structures.
- *Execution time*. Proportionally increases as the number of balls (only for the initialization) and the number of computation steps are increased.

To parallelize the sequential simulation algorithm, an efficient design of the event queue is performance critical. Our event queue is more complex than a priority queue because collision will change queued events. Here, time-division based bins are used. Each bin holds times-to-collision items that occur within a narrow range of times. All the events in a bin are linked together in the increasing order of time-to-collision. All the bins are also linked together in the increasing order of time-to-collision.

There are only two options to parallelize the simulation: parallelizing the simulation steps or parallelizing collision events. In order to determine how to parallelize the simulation, we first need to identify the data dependencies of the program. There are two types of data dependencies: *overlap* and *recollisions*. In the case of overlap, the same ball will be involved in two collisions that are being executed in parallel. In the case of early recollisions, a collision creates a new collision which occurs at an earlier time than some of the collisions being executed in parallel. Because each iteration of the simulation depends on the result of previous iteration, it is hard to parallelize the number of simulation steps. The simulation complexity is mainly determined by the number of events or collisions happening during a given simulation steps, which is proportional to the number of balls. Allocating independent events to be processed in parallel is an effective way to speedup the time-consuming simulation. So, we parallelized the events in the simulation. For given *N* processors, we selected *N* collisions with the smallest times-to-collision in the event queue to execute in parallel. When a processor works on an event, data-

dependencies need to be identified. In this implementation, the number of data-dependent parallel processes in each iteration dynamically changes. However, increasing the number of balls would reduce data-dependency in each iteration because a wider spread of events in the one-dimension space generates less overlap events and less early recollisions in an iteration. The parallel implementation is outlined as follows (*nstep* is the required number of simulation steps, N is the number of processors):

Initialization; $i = 0$;
Do while ($i \le nstep$)

1) Processor $j(1 \le j \le N)$ independently fetches the balls involved in the $(j + 1)$th collision from the event queue, and determines if the selected collision is data-dependent with the collisions on other processors.
2) If data-dependency is found between processors, and the largest processor id, is k, then there is no data dependency among the first $k - 1$ processors.
3) Processor $j(1 \le j \le k - 1)$ executes the allocated collisions by adjusting the positions and velocities of corresponding balls and putting the new events into the appropriate positions to make the event queue sorted.
4) All processors are synchronized by a barrier primitive.
5) $i = i + k - 1$;

End_Do

In each iteration, only the first $k - 1$ processors effectively process a collision between two balls, and the last $N - k - 1$ processors keep idle due to data dependency. The processor with smaller processor id usually has higher utilization than the processor with larger processor id. The load distribution among processors are dynamically changed with data-dependency.

The problem size of the parallel simulation is mainly determined by the number of steps and the number of balls. However, the degree of parallelism mainly depends on the number of balls, less depends on the number of steps. So, when the problem size is scaled only by increasing the number of steps, the efficiency of the parallel simulation is not expected to be improved. This application program feature contradicts a common assertion that the efficiency of a parallel program tends to be improved by increasing the problem size. This also shows the importance to analyze scaling methods. The parallelism degree of the simulation tends to be increased when the problem size is scaled by increasing the number of balls. Thus, the efficiency improves. This parallel simulation is considered interesting to our scalability study due to its unique scaling features.

5.3 Scaling Rules

In general, a complex application program is designed for more than one goal using more than one scaling method to adjust the size of the program. In our application, each simulation goal determines a different scaling method of the program.

For a given application program with n parameters, (we assume parameters are binary values), there are 2^n combinations of the parameters. Each of them can be considered as a scaling method of the application. However, some of them may have no practical meaning. Only those scaling methods matching the application goals are considered as useful scaling methods. Our simulation application has the following four useful scaling methods for four application goals as the number of processors increases.

Scaling Method I. *Fix the number of balls and the number of computation steps.* The memory allocation space / processor decreases as the number of processors increases. The execution time will eventually reach the minimum, and then go up.

Scaling Method II. *Increase the number of balls for a fixed number of computation steps.* The memory allocation space increases linearly with the increase of the number of balls. However, while the increase in the number of balls increases the execution time of the initialization part, the parallel execution time in the main loop of the simulation decreases.

Scaling Method III. *Increase the number of computation steps for a fixed number of balls.* Using this scaling method, the memory allocation / processor is the same as the one in scaling method I. The execution time of the simulation is increased with the increase of the number of computation steps.

Scaling Method IV. *Increase both the number of balls and the number of computation steps.* Since both space and computation are scaled, both the memory allocation and the execution time of the program are increased by this scaling method.

Since the simulation program can be scaled at least by four different ways, it is a good candidate to use for studying scalability performance and for testing the Scale-Graph tool.

6 Latency Measurement and Evaluation Using Scale-Graph on the KSR-1

In this section, we use the previous physics simulation as an example to show the three major functions of the Scale-Graph tool: sequential program analysis, latency measurements using a fixed size program, and latency measurements using other scaling methods. The program and its scaling variations were intensively run on the KSR-1, a hierarchical ring network-based shared-memory system [9].

6.1 Identifications and Measurements of the Latency Sources

Parallel computing scalability is evaluated by the measured latency on a network system, such as KSR-1, Intel Paragon, IBM SP2, and a network of workstations. Here we address the issue of latency sources and measurements in a general way which covers both shared-memory and distributed memory systems, although the experiments we did were on the KSR-1, a shared-memory system.

6.1.1 The Average Latency

The latency definition in (2) conceptually introduces overhead distributions during an execution. In practice, there are three major latency sources to be measured, namely, the memory reference latency, denoted by ML, the processor idle time, denoted by IT, and the parallel primitive execution overhead time, denoted by PT. The average latency to be measured in the latency metric is then defined as

$$L(W, N) = \frac{ML + IT + PT}{N}, \tag{5}$$

where *ML*, *IT*, and *PT* are the sums of memory reference latency, processor idle time and the parallel primitive execution time in each processor, respectively, *W* is the problem size, and *N* is the number of processors used for solving the problem.

- *Memory reference latency* measures the delays caused by communication between processors and memory modules over the network. In a shared-memory system, this mainly comes from remote read and write accesses and corresponding cache coherence operations, while in a distributed memory system, this comes from message passing for remote read and write operations.
- *Processor idle time* is caused mainly by computing load structures of programs. In a shared-memory system, it comes from process scheduling and memory access contention. In a distributed memory system, it comes from message waiting and processor waiting for task scheduling.
- *Parallel primitive execution time* covers the software overhead and related network bandwidth and processor waiting cycles. These execution cycles are used to support unique instructions providing necessary services for parallel programming and computing, such as synchronization locks and barriers, thread scheduling primitives in a shared-memory system, and send/receive and task loading primitives in a distributed memory system.

6.1.2 Measurements of the Latency

Measurement of parallel primitive execution time (*PT*) is relatively straightforward. This may be done by inserting system timers before and after the primitive calls. Many vendors also provide the number of cycles that each primitive uses for the operation, which can be used as software overhead references. These primitive operations are only used in parallel programs. However, the other types of latency sources are related to the same operations in sequential programs.

The processor idle time in a parallel program is measured as follows:

$$IT = NT_{para} - \sum_{i=1}^{N} T_i, \tag{6}$$

where T_{para} is the measured parallel execution time running on N processors, and T_i is the measured execution time of the ith processor.

Memory reference latency caused by read/write operations in a parallel program is also related to the corresponding operations in its sequential program, and is expressed as:

$$ML = ML(N) - ML(1), \tag{7}$$

where $ML(N)$ is the measured memory reference latency of a parallel program on N processors, and $ML(1)$ is the measured latency by the same operations when the program is running on a single processor.

6.2 Execution Pattern Analysis of the Sequential Simulation Program

The multiple windows plotted from Scale-Graph in Figs. 5, 6, 7, and 8 present comprehensive analyses of execution patterns of the sequential program. The window in Fig. 5 presents an execution profile of the sequential program with input data of 5,000 balls and 100 computation steps, and the program subroutine structure. (The program structure of the simulation is described in Section 5.2.) The profile window gives the execution distribution structure of the program where subroutine ADD_TIME_TO_QUEUE spends most of the computing time (95.2%), and subroutine REMOVE_FROM_QUEUE spends the least computing time (0.4%).

We further ran a group of experiments on a single processor of the KSR-1. Although these experiments were very time-consuming, they provided us with important data for choosing parameters for later parallel programs. The three windows in Figs. 6, 7, and 8 present the performance statistics of the program using scaling Methods II, III, and IV, respectively. Using the three figures, we can compare the effects of each scaling method on CPU utilization (the ratio between the CPU busy time and the total execution time), on memory access rate (the ratio between the memory access time and the total execution time), and on memory access miss rate (the ratio between the number of memory misses and the total number of memory accesses), separately. The statistical data indicate that when both the number of balls and the number of steps were scaled, the CPU utilization started at low levels for small sizes of the problem. The utilization reached a stable level after the problem was scaled to a sufficient size. In addition, the performance results indicate that the CPU utilization was almost independent of the number of computation steps or the number of balls if we fix one parameter and scale the other one.

The statistical data also indicate that when both the number of balls and the number of steps were scaled, the memory access rate started at a high level for small sizes of the problem. The memory access rate reached a stable level after the problem was scaled to a sufficient size. This fact also applied to the case when the number of computation steps was fixed and the number of balls was scaled. In addition, the performance results show that the memory access rate was independent of the number of steps when the number of balls was fixed.

Finally, the statistical data indicate that when both the number of balls and the number of steps were scaled, the memory access miss rate was increased moderately. This fact also applied to the case when the number of computation steps was fixed and the number of balls was scaled. In addition, the performance results show that the memory access miss rate was slightly decreased when the number of steps was increased for a given number of balls.

In summary, the number of balls has a more direct effect than the number of computing steps on the CPU utilization and on the memory access and memory access miss rates.

6.3 Latency Measurements of the Program of Fixed Size

We used Pmon, a hardware monitor on the KSR-1, to trace the total number of access-miss operations and to calculate the average latency for running a program on the architecture with different numbers of processors.

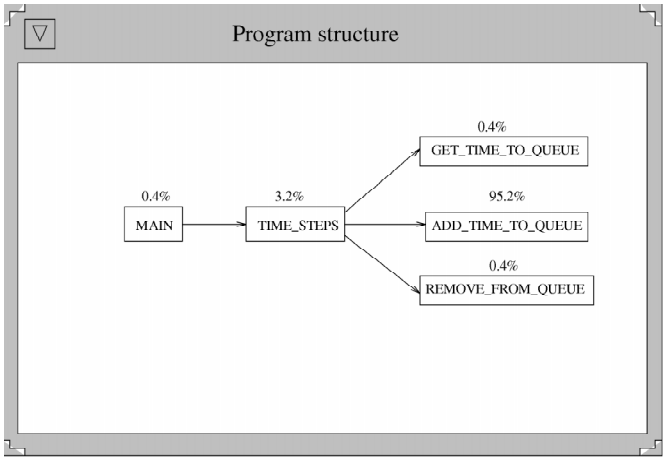


Fig. 5. Execution profile of the sequentila program with input data of 5,000 balls and 100 computation steps.

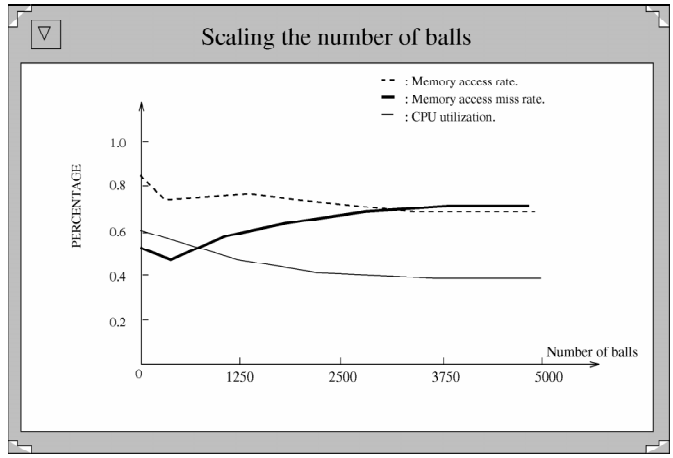


Fig. 6. Performance data using scaling method II: Fix the number of steps and scale the number of balls.

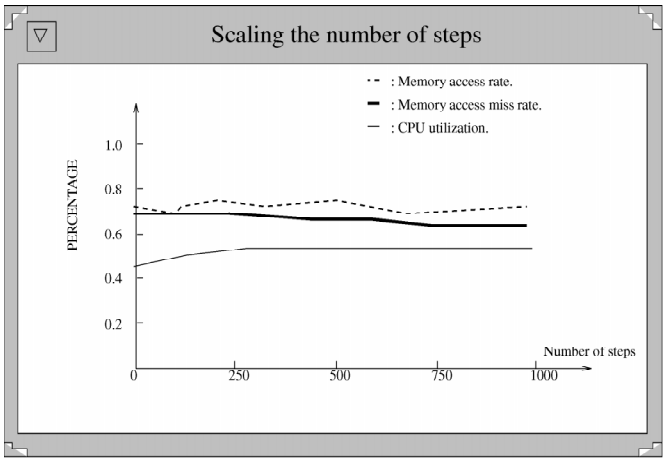


Fig. 7. Performance data using scaling method II: Fix the number of balls and scale the number of steps.

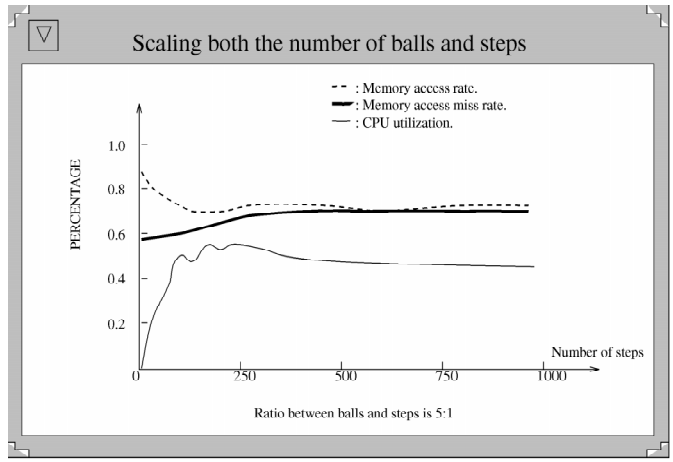


Fig. 8. Performance data using scaling method IV: Scale both the number of balls and the number of steps.

The three windows plotted from Scale-Graph in Figs. 9, 10, and 11 partially present execution patterns and scalability performance of the parallel simulation program for a given problem size on the KSR-1 system with an increase in the number of processors. The window in Fig. 9 presents execution time measurements of the fixed size simulation problem. The execution time of the program was measured by the KSR-1 monitor pmon which is the total time used by the the system for the program. Since the system is dedicated, and the time spent by system kernel is relatively small, this time is equivalent to the "wall clock time."

The performance indicates that the program reached the minimum execution time when 53 processors were used, and then execution time went up. This particular turning point of the number of processors used is denoted as *px*.

The window in Fig. 10 presents latency patterns of the program through the measurements of average latency in a time unit. The performance indicates the latency sharply increased and then reached a level with a stable increase. Based on the execution time curve, the latency at $px = 53$ represents the maximum value for this program to achieve the best possible performance.

The window in Fig. 11 presents the overhead patterns, where barrier overhead time, lock overhead time, CPU waiting time for barriers, CPU stall time caused by access misses, and average of overhead time during execution are plotted for each run on different numbers of processors. As we expected, all of these overheads increase as the number of processors increases.

6.4 Scalability Measurements of the Program Using Other Scaling Methods

The window in Fig. 12 presents the parallelism patterns of the program by fixing the number of steps and scaling the number of balls, on 24, 32, and 64 processors of the KSR-1, respectively. The parallelism defines the percentage of parallel operations in the total number of operations. This window indicates that the parallelism of the program is quite sensitive to the number of processors used.

The latency sources have been collected by Scale-Graph for three scaling methods by keeping a constant efficiency *E*, (under an E-conserved condition). The execution time of the simulation mainly depends on the scaling of the number of steps, where representative latency changing patterns

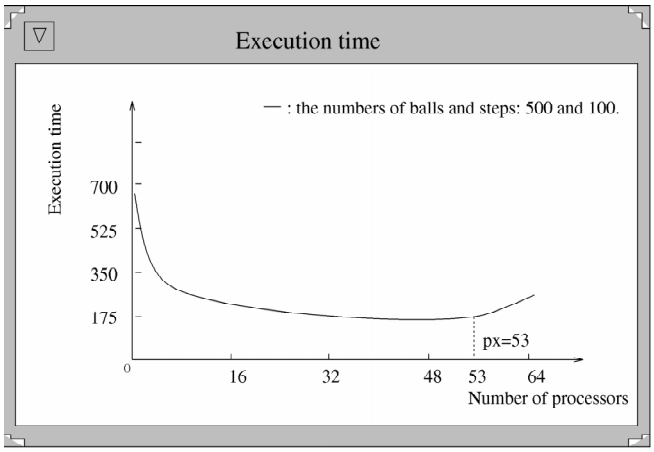


Fig. 9. Execution time measurement of a fixed size simulation on the KSR-1.

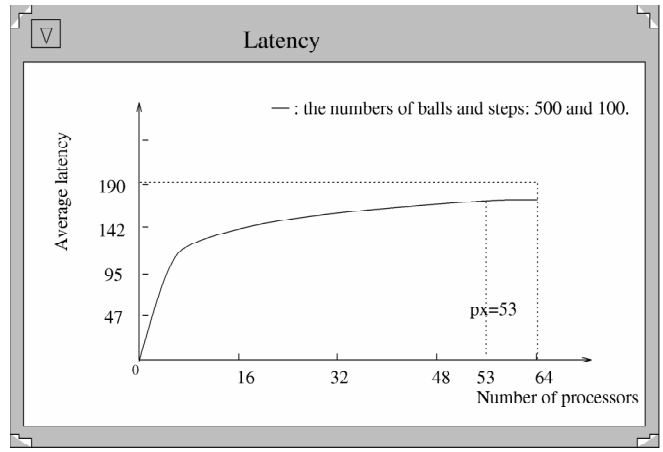


Fig. 10. Latency measurements of a fixed size simulation on the KSR-1.

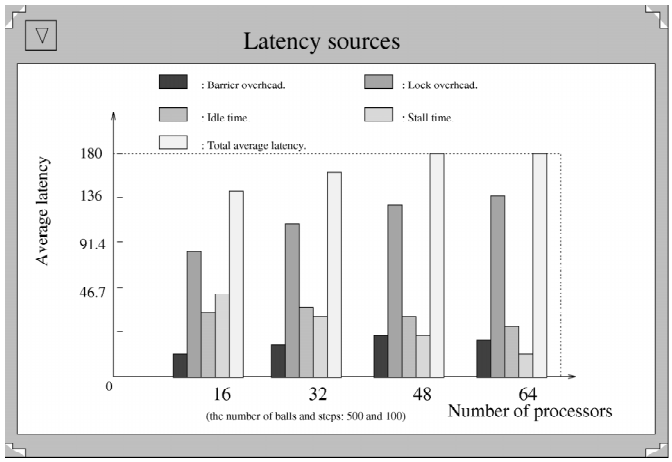


Fig. 11. Latency source patterns of a fixed size simulation on the KSR-1.

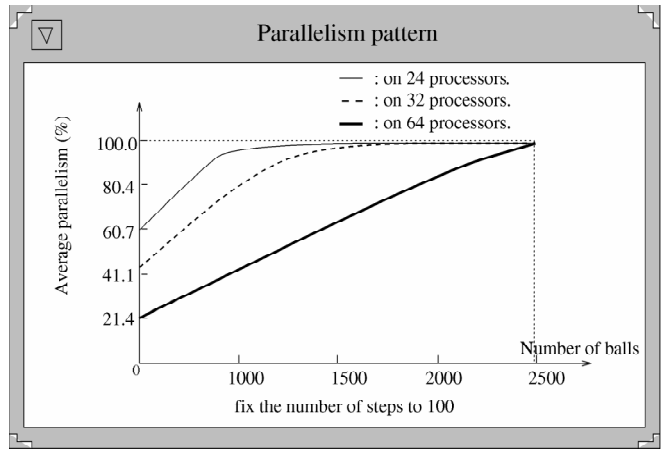


Fig. 12. Parallelism patterns of the program by fixing the number of steps and scaling the number of balls on the KSR-1.

are exhibited. The window in Fig. 13 shows the latency sources of this scaling method. Compared with the fixed size simulation in Fig. 11, the scaling latency patterns are quite identical, but the magnitudes are significantly different. For example, execution time of the scaling simulation is about 8.4 times higher than the one of the fixed size simulation. In addition, while the idle time in the scaling simulation slightly increases with the number of processors, it slightly decreases in the fixed size simulation.

A major function of Scale-Graph is to measure and evaluate scalabilities of a program and an architecture as both the sizes of the program and the architecture are scaled. Using the latency metric as the base, Scale-Graph provides functions to measure the scalabilities based on the time-bound, memory-bound and *E*-conserved methods. The measured scalability results (three numbers using the three methods are grouped together separately by /'s) are filled in a upper triangular matrix, where the processor numbers listed in the first column in the matrix are the N, and the processor numbers listed in the first row in the matrix are the N'. (See the latency metric definition in Section 3.) Fig. 14 presents the measured scalability results for the same simulation program (scaling both the numbers of balls and steps at the ratio of 5:1) by using the time-bound method, the *E*-conserved, and the memory-bound methods, respectively.

6.4.1 A Limit of Time-Bound Measurements

Since the time-bound is set, the execution time is limited to a certain degree. The measured latency using the time-bound method may have smaller increases than it would for a given high latency program. Therefore, the latency scalability measured in this way would be overestimated. To see this quantitatively, we reconstruct the latency metric defined in Section 3. Let W be the problem size, N be the number of the processors and T_{para} be the parallel execution time. Recall that the sequential execution time (T_{seq}) for solving the problem of size W is:

$$T_{seq} = WT_c, \tag{8}$$

where t_c is the average time of a basic machine operation. From the definition of efficiency, we have

$$E = \frac{Wt_c}{NT_{para}} = \frac{Wt_c}{N(\frac{Wt_c}{N} + L(W, N))}. \tag{9}$$

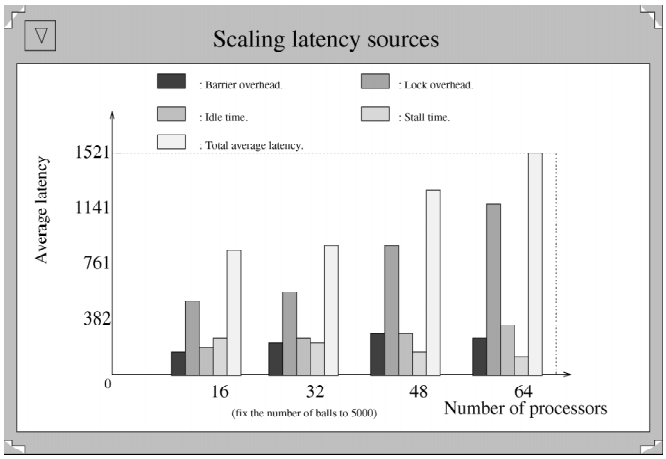


Fig. 13. Latency sources of the program by fixing the number of balls and scaling the number of steps on the KSR-1.

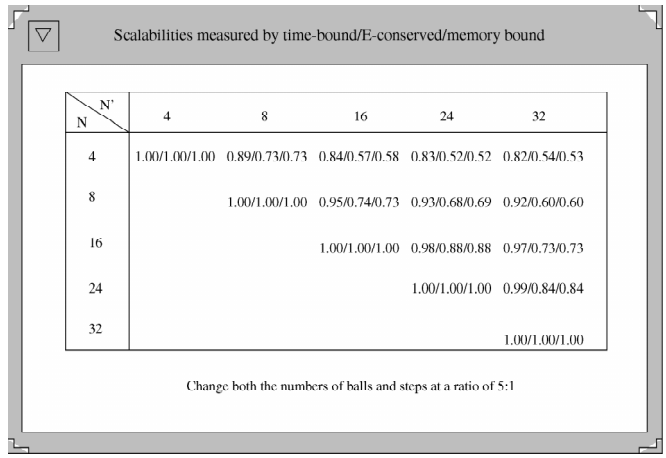


Fig. 14. Scalability measurements using time-bound/ E-conserved/ memory bound methods on the KSR-1.

From (9) and using the definition of T_{seq} in (8), the latency becomes

$$L(W, N) = T_{para}(1 - E). \tag{10}$$

Similarly, from (8), we have

$$W = \frac{T_{para} NE}{t_c} \tag{11}$$

The basic idea of using the time-bound model is to measure the capacity at which a larger application program can be more effectively run on a larger set of processors in a given time-bound. The measurement of this capacity is generally conducted by measuring the increase of the average latency or the increase of the problem size between the two runs.

For a given time-bound T and an application program, let W, L, and E be the average problem size of the program, the average latency, and the efficiency when the program runs on N processors in time-bound T, respectively. Let W' ($W' > W$), L', and E' be the average problem size of the program, the average latency, and the efficiency when the program runs on N' ($N' > N$) processors in the same time-bound T. We can measure the scalability using the time-bound model in the following two cases:

Case 1. If we use the increase of the average latency as the measurement of the scalability, we have the scalability metric from (10):

$$scale(T, (N, N')) = \frac{L(W, N)}{L(W', N')} = \frac{1 - E}{1 - E'}. \tag{12}$$

In practice, for a given poorly scalable program, we can find a time-bound for the computation such that the values of E and E' are very low and very close. In this case, the scalability measured by (12) comes from very low increase of the latency (the value is close to 1). Then the scalability is considered high. This is not correct, because the real scalabilities of the parallel program are low. Therefore, the measured scalability may be overestimated and meaningless.

Case 2. If we use the increase of the average problem size as the measurement of the scalability which is defined in the isospeed metric, we have the scalability metric from (11):

$$scale(T, (N, N')) = \frac{W / N}{W' / N'} = \frac{E}{E'}. \tag{13}$$

For the same reason as *Case 1*, the measured scalability by (13) may be overestimated and meaningless when E and E' are low and close.

In contrast, detailed analysis in Section 2 indicates that the E-conserved method keeps the efficiency as a constant, and has no direct effect on program execution. Therefore, E-conserved methods are a reasonable base and fair for scalability measurements.

In our experiments, the time-bound was set to be 100 sec for measuring the latency scalability in the time-bound method, while the constant efficiency was set to be 50% in the E-conserved method. Comparing the scalability results using the time-bound method and the ones using the E-conserved method, we can see significant differences to confirm the overestimated scalability results in Fig. 14. For example, the scalability measurements from four processors to 32 processors report 0.82 for the time-bound and 0.54 for the E-conserved (and 0.53 for the memory-bound).

6.4.2 A Limit of Memory-Bound Measurements

The scaling method for measuring latency scalability for the memory-bound case is more complex. Assume that the memory-bound is set to be B (bytes). The memory requirement, denoted as M (bytes) is the physical allocation of the program. In this simulation program, $M = C \times num\_balls$, where C is 960 for a data structure of 30 floating representations on the KSR-1. If N processors are used, we have

$$N \times B = C \times num\_balls.$$

The number of balls can then be calculated by,

$$num\_balls = \frac{N \times B}{C}, \tag{14}$$

which is the base formula for the memory-bound scaling method. By changing the memory-bound B, we can perform the scaling method of increasing the number of balls

for a given number of computation steps; we can also perform the scaling method of increasing both the numbers of balls and the computation steps. However, the memory-bound prevents us from using the scaling method of increasing the number of computation steps for a given number of balls, because formula (14) can only be used to adjust the number of processors. This shows a limitation of the memory-bound method in practice.

7 Summary and Current Work

We have measured, evaluated, and visualized parallel computing scalabilities and related execution performance based on the latency metric using different bound methods. The application program we used has multiple input parameters, which can be scaled in four different ways. We show that both time-bound and memory-bound models have their limitations for precisely and completely evaluating the scalability performance in theory and practice. We also show that the *E*-conserved method used in the latency metric can achieve reasonably precise measurements. Since the network latency is a major obstacle to the improvement of parallel computing performance and scalability on all network-based parallel machines, the framework of ScaleGraph based on the network latency can be easily implemented for performance evaluation for other architectures.

Currently, we are applying the same concept of ScaleGraph to provide software and graphical presentation support for parallel computing on heterogeneous networks of workstations (NOW) [13], [15]. The scaling of a NOW system is different from a homogeneous multiprocessor system. It can be scaled in three directions: By increasing the number of workstations (physical scaling), by upgrading workstation powers (power scaling), and by combining both physical scaling and power scaling. In physical scaling, a parallel job tends to increase average overhead latency due to more workstations involved in communication and synchronization. In power scaling, communication complexity remains constant, but the computation and communication ratio increases, which would reduce the computation efficiency. How to balance the two scaling methods to achieve optimal performance for a parallel job is a complex issue we are currently investigating.

Acknowledgments

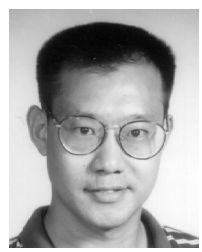
We wish to thank X.-H. Sun for his constructive comments when we first developed the latency scalability metric. We appreciate N. Wagner for carefully reading the paper and making helpful suggestions. Finally, we are grateful to the three anonymous referees for their helpful and detailed comments and suggestions.

This work was supported in part by the National Science Foundation under grants CCR-9102854 and CCR-9400719, by the U.S. Air Force under agreement FD-204092-64157, by the Air Force Office of Scientific Research under grant AFOSR-95-1-0215, and by a visiting scholar fellowship from the United Nations Children's Fund. Part of the experiments were conducted on the KSR-1 machines at Cornell University and at the University of Washington.

References

[1] A. Beguelin, "Xab: A Tool for Monitoring PVM Programs," *Proc. Workshop Heterogeneous Processing*, pp. 92-97. IEEE CS Press, Apr. 1993.
[2] A. Beguelin et al., "Visualizing and Debugging in a Heterogeneous Environment," *Computer*, pp. 88-95, June 1993.
[3] A. Beguelin et al., "Recent Enhancements to PVM," *Int'l J. Supercomputer Applications and High Performance Computing*, vol. 9, no. 2, pp. 108-127, 1995.
[4] S. Brawer, *Introduction to Parallel Programming*. Academic Press, 1989.
[5] S.G. Dykes, X. Zhang, Y. Shen, C.L. Jeffrey, and D.W. Dean, "*Graph: A Tool for Visualizing Communication and Optimizing Memory Layout in Data-Parallel Programs," *Proc. Int'l Conf. Parallel Processing*, vol. II, pp. 121-129. CRC Press, Aug. 1995.
[6] A. Grama, A. Gupta, and V. Kumar, "Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures," *IEEE Parallel and Distributed Technology*, vol. 1, no. 3, pp. 12-21, 1993.
[7] J.L. Gustafson, G.R. Montry, and R.E. Brenner, "Development of Parallel Methods for a 1,024-Processor Hypercube," *SIAM J. Scientific and Statistical Computing*, vol. 9, no. 4, pp. 522-533, 1988.
[8] M.T. Heath and J.A. Etheridge, "Visualizing the Performance of Parallel Programs," *IEEE Software*, vol. 8, no. 5, pp. 29-39, 1991.
[9] Kendall Square Research, *KSR-1 Technology Background*.
[10] D.A. Reed et al., "Scalable Performance Analysis: The Pablo Performance Analysis Environment," *Proc. Scalable Parallel Libraries Conf.*, pp. 104-113. IEEE CS Press, 1993.
[11] X.-H. Sun and D.T. Rover, "Scalability of Parallel Algorithm-Machine Combinations," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, no. 6, pp. 599-613, 1994.
[12] Z. Xu, X. Zhang, and L. Sun, "Semi-Empirical Multiprocessor Performance Predictions," *J. Parallel and Distributed Computing*, vol. 39, no. 1, pp. 14-28, 1996.
[13] Y. Yan, X. Zhang, and Y. Song, "An Effective and Practical Performance Prediction Model for Parallel Computing on Nondedicated Heterogeneous NOW," *J. Parallel and Distributed Computing*, vol. 38, no. 1, pp. 63-80, 1996.
[14] X. Zhang, N. Nalluri, and X. Qin, "MIN-Graph: A Tool for Monitoring and Visualizing MIN-Based Multiprocessor Performance," *J. Parallel and Distributed Computing*, vol. 18, no. 2, pp. 231-241, 1993.
[15] X. Zhang and Y. Yan, "Modeling and Characterizing Parallel Computing Performance on Heterogeneous NOW," *Proc. Seventh IEEE Symp. Parallel and Distributed Processing*, pp. 25-34. IEEE CS Press, Oct. 1995.
[16] X. Zhang, Y. Yan, and K. He, "Latency Metric: An Experimental Method for Measuring and Evaluating Program and Architecture Scalability," *J. Parallel and Distributed Computing*, vol. 22, no. 3, pp. 392-410, 1994.

Yong Yan received the BS and MS degrees in computer science from Huazhong University of Science and Technology, Wuhan, Peoples Republic of China, in 1984 and 1987, respectively, where he has been a member of the faculty since 1987. He is presently a PhD candidate in computer science at the University of Texas at San Antonio. Yan was a visiting scholar in the High Performance Computing and Software Laboratory at UTSA from 1993 to 1995. Since 1987, he has been published extensively in the areas of parallel and distributed computing, performance evaluation, operating systems, and algorithm analysis. He is a member of the IEEE and the ACM.



Xiaodong Zhang (SM'94) received the BS degree in electrical engineering from Beijing Polytechnic University, Peoples Republic of China, in 1982, and the MS and PhD degrees in computer science from the University of Colorado at Boulder in 1985 and 1989, respectively. Dr. Zhang is an associate professor of computer science at the University of Texas at San Antonio, where he is directing the High Performance Computing and Software Laboratory. His research interests

are parallel and distributed computation, computer system performance evaluation, and scientific computing. Dr. Zhang is current chair of the Technical Committee on Supercomputing Applications of the IEEE Computer Society.

Qian Ma received his BS degree in computer science from Beijing Computer Institute, Peoples Republic of China, in 1985. He is a senior systems analyst at the Beijing Automated Systems Company. He was a visiting scholar in the High Performance Computing and Software Laboratory, University of Texas at San Antonio from 1992 to 1993, sponsored by the United Nations Foundation and the National Science Foundation. His research interests are system software development and performance evaluation.