JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 22, 392–410 (1994)

Latency Metric: An Experimental Method for Measuring and Evaluating Parallel Program and Architecture Scalability[1]

XIAODONG ZHANG, YONG YAN*, AND KEQIANG HE†

*High Performance Computing and Software Laboratory, The University of Texas at San Antonio, San Antonio, Texas 78249; *Department of Computer Science, Huazhong University of Science and Technology, Wuhan, People's Republic of China; and †Cartotech Inc., San Antonio, Texas 78233*

Latency measures the delay caused by communication between processors and memory modules over the network in a parallel system. Using intensive measurements and simulation, we show that network latency forms a major obstacle to improving parallel computing performance and scalability. We present an experimental metric, using network latency to measure and evaluate the scalability of parallel programs and architectures. This latency metric is an extension to the isoefficiency function [Grama *et al.*, *IEEE Parallel Distrib. Technology* 1, 3 (1993), 12–21] and isospeed metric [Sun and Rover, *IEEE Trans. Parallel Distrib. Systems* 5, 6 (1994), 599–613]. We give a measurement method for using this latency metric, and report the experimental results of evaluating the scalabilities of several scientific computing algorithms on the KSR-1 shared-memory architecture. Our analysis and experiments show that the latency metric is a practical method to effectively predict and evaluate scalability based on measured latencies inherent in the program and the architecture. © 1994 Academic Press, Inc.

1. INTRODUCTION

Parallel computing scalability does not have a commonly accepted definition yet. However, in scientific computations, people are more interested in knowing if there is a corresponding increase in performance of a computation as the size of a parallel machine is increased. The increase of computing performance is primarily affected by overhead patterns inherent to an application program and the effects of the architecture's interconnection network. Program overhead patterns refer to the synchronization and communication structures of the program, while the effects of the architecture's interconnection network refer to the delays inherent in communication hardwares. Therefore, parallel computing scalability consists of architecture scalability which is related to bottlenecks inherent in an architecture design, and algorithm scalability which is related to the parallelism inherent in an algorithm design.

Scalability is aimed at measuring the ability of a parallel architecture where the parallelism of a given algorithm has already been effectively exploited. Evaluation of the scalability can also be used to predicate the performance of large problems on large systems based on the performance of small problems on small systems. A rigorous scalability definition and metric provides an important guideline for precisely understanding the nature of the scalability, and for effectively measuring the scalability in practice. Isoefficiency [3] and isospeed [9] 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 to be an analytical method for algorithm scalability evaluation. Although isospeed metric is an experimental metric, it may face the difficulty of measuring some real machine factors in practice.

In this paper, we present an experimental metric using network latency for measuring and evaluating parallel program and architecture scalability. We first give the definitions of latency and scalability. Furthermore we show the analytical relationships among the latency metric, the isoefficiency function, and the isospeed metric. Finally we give a measurement method for using the latency metric. We include experimental measurements on the KSR-1 to show the effectiveness of the latency metric in predicting and evaluating the scalability.

The organization of this paper is as follows. Section 2 overviews the two current available metrics. We emphasize the evaluation of the merits and limits of the isospeed metric. We present our latency metric in Section 3. We overview the architectures and the application programs used for the experiments in Section 4. In Section 5, we present traced and monitored program execution results, and address the importance of using the latency to abstract various effects from network architectures and program structures. Both measurement methods and experimental results of several numerical programs on

[1] This work is supported in part by the National Science Foundation under research Grants CCR-9102854 and CCR-9400719, and under instrumentation Grant DUE-9250265, by the U.S. Air Force under research agreement FD-204092-64157, by a grant of Cray Research, and by a Fellowship from the Southwestern Bell Foundation. Part of the experiments were conducted on the BBN TC2000 at Lawrence Livermore National Laboratory, and on the KSR-1 machines at Cornell University, and at the University of Washington.

0743-7315/94 $6.00
Copyright © 1994 by Academic Press, Inc.
All rights of reproduction in any form reserved.

KSR-1 are presented in Section 6. Finally we give summaries and conclusions in Section 7.

2. SCALABILITY METRICS FROM EFFICIENCY AND SPEED

2.1. Overview and Background

The definition of scalability comes from Amdahl's law which is tied to efficiency and speedup. There are two important scalability metrics: the *isoefficiency* function based on parallel computing efficiency [3], and the *isospeed* metric based on parallel computing speed [9]. 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). Specifically, the efficiency is defined as

$$E = \frac{1}{1 + T_0(n, W)/t_c W}, \tag{2.1}$$

where T_0 is the total overhead caused by all processors to do the computation in parallel, t_c is the average executing time per operation in the architecture, W is the problem size, and n is the number of processors. Here $t_c W$ is the sequential runtime of an algorithm.

If the efficiency needs to be maintained at a certain value $E(0 < E < 1)$, then from (2.1)

$$W = \frac{1}{t_c}\left(\frac{E}{1 - E}\right) T_0(n, W),$$

or

$$W = K T_0(n, W), \tag{2.2}$$

where $K = (1/t_c)(E/1 - E)$ is a constant. The scalability is determined by the overhead function $T_0(n, W)$. The larger the $T_0(n, W)$, the lower the scalability of the algorithm on the architecture will be. If an analytical form T_0 of a given algorithm on a given architecture is described as a function of n and W, the isoefficiency ($E(n)$) curve using up to n processors can be generated for evaluating the scalability.

The isoefficiency function (2.2) 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 size changeable parallel architecture so that the processing efficiency of each processor can remain constant. However, there are two limits in this metric to the experimental evaluation of the 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. Because the computing processes involved in a shared-memory system include 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 the scalability for a program running on a shared-memory system in practice. Second, the metric may not be used to measure the scalability of the algorithm–architecture combination through machine measurements. Of course, experiments can be used to verify the analytical isoefficiency for the algorithm on a specific architecture. We believe this metric is more appropriate to evaluate the scalability of parallel algorithms.

Sun and Rover [9] take an approach to algorithm–machine combinations. Their metric starts from defining the average unit speed

$$speed = \frac{W}{Nt}, \tag{2.3}$$

where W is the amount of work quantitatively given by the number of floating point operations in the program, N is the number of processors, and t is the execution time.

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. It can be further expressed as

$$Scalability(N, N') = \frac{W/N}{W'/N'}, \tag{2.4}$$

where W and W' are the amounts of work (or the problem sizes) for the architecture of size N and for the architecture of size N', respectively. This metric provides more information about architectures and programs because the scalability of (2.4) can be well determined through machine measurements.

2.2. Merits and Limits of the Isospeed Metric

Our experimental metric is along the same line of the isospeed metric. Here we first study the merits and limits of this metric. Since the computing speed is used to define the scalability function, the isospeed metric has the following merits in theory and practice. First, the speed comes from two major performance factors: the problem size and the execution time. While the problem size describes a property of the application program, the execution time reflects the effects of the architecture and the efficiency of the program. Second, the speed is a fair quantity for comparisons among various architectures. The execution time includes the pure computing part and the latency part which is the major performance factor. Finally, the isospeed is easy to measure because the problem size is determined by the number of floating

point operations performed in the computation. However, in terms of algorithm–machine combination, the isospeed metric may not explicitly and completely measure the architecture effects and the program overhead patterns, because of the usage of the floating point operations to measure the work (problem size). 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 the scalability. Second, the latency is included in the total execution time in the metric, but is not defined in the amount of work, W, in the scalability metric (2.4). In practice, the execution overhead caused by the interconnection network and the program structure is a function of the problem size. Since the execution patterns can be precisely monitored by hardware and/or software in more and more modern parallel architectures, scalability can be further evaluated at lower application and system levels to capture more precise performance factors of architecture effects and program overhead patterns. In the next section we propose a new metric called the *latency metric*, to enhance the ability of the isospeed metric. Instead of the speed, we use the latency, the average computing delay, as the major factor in the metric. We show in Section 5 that latency includes more precise information of the architecture's interconnection network effects and the overhead patterns inherent in application programs.

3. THE LATENCY METRIC

3.1. Definitions and Assumptions of the Metric

We define the latency metric through a series of formal definitions and theorems.

DEFINITION 1. The parallel computing time of an algorithm implementation, denoted as T_{para}, is the elapse 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 computing. The effective execution time in each processor includes the latency time during the execution (see Definition 2 for latency) but does not include the idle time of waiting for starting the execution and the idle time of waiting for ending the program.

DEFINITION 2. Overhead latency in the ith processor, denoted as L_i for $i = 1, \ldots, N$, is the sum of the total 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.

DEFINITION 3. The size of a problem, denoted as 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 as 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. Average latency, denoted as $L(W, N)$, is a function of the problem size W and the number of processors used N, and is defined as an average amount of overhead time needed for each processor to complete the assigned work:

$$L(W, N) = \frac{\sum_{i=1}^{N}(T_{para} - T_i + L_i)}{N}. \tag{3.5}$$

Figure 1 provides an example of latency and execution time distributions given in Definitions 1–4.

In the isoefficiency function and the isospeed metric, the scalability is defined as an algorithm–machine combination, and may not be used to measure different parallel implementations of a given algorithm. However, in parallel programming, different implementations of an algorithm may have very different computing performance. In contrast, our latency metric defines the scalability of a parallel algorithm implementation–machine combination, simplified as an implementation–machine combination. The next definition combines this concept in the scalability.

DEFINITION 5. For a given efficiency, $E \in [0, 1]$, of running a program on N processors, 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, the implementation–machine combination is called scalable.

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

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

$$T_{para} = \frac{Wt_c}{N} + L(W, N), \tag{3.6}$$

where W is the size of a problem, N is the number of the processors, t_c is the average computing time per opera-

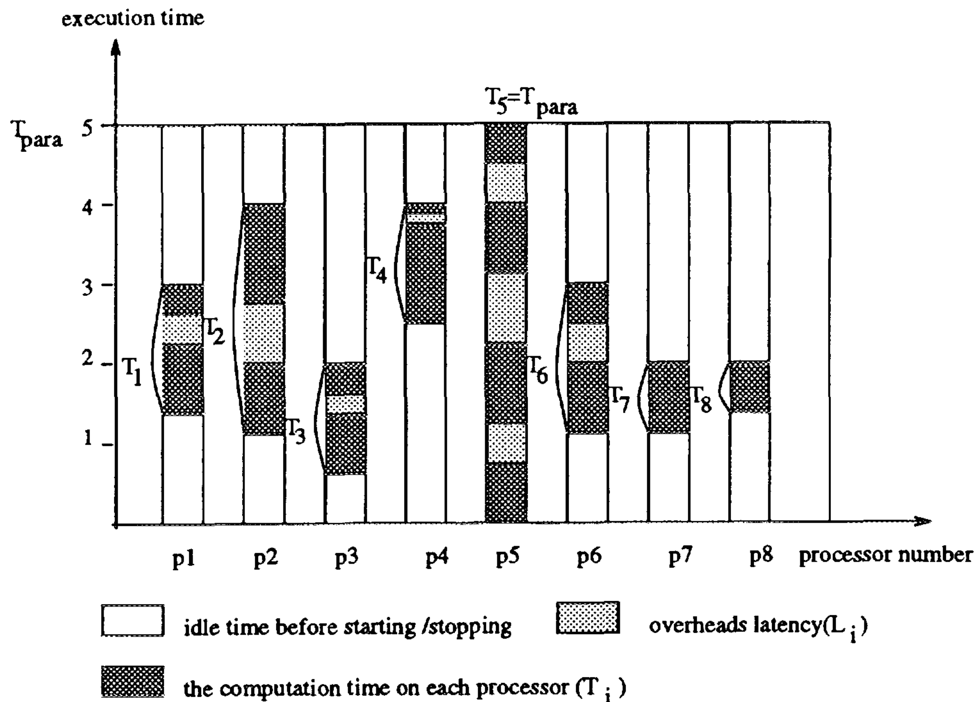

FIG. 1. An example of latency and execution time distributions given in Definitions 1-4.

tion 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 is for solving a problem of size W on N processors, and $L_e(W', N')$ be the average latency when the algorithm is 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

$$scale(E, (N, N')) = \frac{L_e(W, N)}{L_e(W', N')}. \tag{3.7}$$

We also call the metric in (3.7) an E-conserved scalability because the efficiency is kept constant. From the definition of the efficiency in (2.1), (3.7) satisfies the following E-conserved condition:

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

In practice, the value of (3.7) is less than or equal to 1. A large scalability value of (3.7) means small increments in latencies inherent in the program and the architecture for efficient utilization of an increasing number of processors, and hence the parallel system is considered highly scalable. On the other hand, a small scalability value means large increments in latency and therefore a poorly scalable system.

Furthermore, when a set of $scale(E, (N, N'))$ values is measured on a system for different given efficiency Es, an average scalability may be described by an integration of the latency metric from N to N':

$$Scale(N, N') = \int_0^1 scale(E, (N, N'))\, dE. \tag{3.8}$$

3.2. Analytical Relationships among the Three Metrics

Based on the latency metric and the related definitions in the previous section, we are also able to present analytical relationships among the isoefficiency function, the isospeed metric, and the latency metric through the following two theorems.

THEOREM 1. *If we assume the average execution time per operation for solving a problem in an architecture, t_c, is a constant, then the efficiency E and the speed S have the following relationship:*

$$E = St_c. \tag{3.9}$$

This theorem indicates that the isoefficiency is equivalent to the isospeed.

Proof. Let W be the problem size, N be the number of the processors and T_{para} be the parallel execution time. The sequential execution time (T_{seq}) for solving the problem of size W is:

$$T_{seq} = Wt_c. \tag{3.10}$$

From the definitions of the speed and the efficiency, we have

$$S = \frac{W/T_{\text{para}}}{N}, \qquad (3.11)$$

and

$$E = \frac{T_{\text{seq}}/T_{\text{para}}}{N}. \qquad (3.12)$$

Substituting (3.10) into (3.12), and combining the result with the (3.11), we obtain (3.9).

THEOREM 2. *Let $L_e(W, N)$ be the average latency when an algorithm is for solving a problem of size W on N processors, and $L_e(W', N')$ be the average latency when the algorithm is for solving the problem is of size W' on $N' > N$ processors. If the state of a program execution changes from (W, N) to (W', N'), and the* **speed** *or the* **efficiency** *of the system remains unchanged, then the E-conserved latency scalability metric is*

$$scale(E, (N, N')) = \frac{L_e(W, N)}{L_e(W', N')} = \frac{W/N}{W'/N'}. \qquad (3.13)$$

This theorem indicates that the latency metric is identical to the isospeed metric (2.4) in theory. However, in practice we measure the latency $L(W, N)$, which provides more precise overhead effects of the architecture and the program structure.

Proof.

Part 1. Assume that the speed, S, of the system remains unchanged. It can be expressed from the unit speed definition (2.4) and the parallel computing time definition in (3.6) as

$$S = \frac{W/N}{L(W, N) + Wt_c/N}. \qquad (3.14)$$

From (3.14), the latency becomes

$$L(W, N) = \frac{W}{N}\left(\frac{1}{S} - t_c\right). \qquad (3.15)$$

From (3.15), we get (3.13). This means the latency metric can be derived from the isospeed metric.

Part 2. Assume that the efficiency E of the system remains unchanged. It can be expressed from its definition and (3.6) as

$$E = \frac{Wt_c}{NT_{\text{para}}} = \frac{Wt_c}{N(Wt_c/N + L(W, N))}. \qquad (3.16)$$

From (3.16), the latency becomes

$$L(W, N) = \frac{W}{N}\left(\frac{1 - E}{E}\right)t_c. \qquad (3.17)$$

From (3.17), we obtain (3.13). This means the latency metric can also be derived from the isoefficiency function (3.17).

Theorem 2 indicates that the latency scalability metric covers the scalability measures included in **both** the isospeed function and the isoefficiency metric. Again, another important reason for using the latency is related to the real measurements in computer systems. This metric more directly and precisely catches the architecture interconnections network effects and the overhead patterns inherent in the algorithm in the program execution. In general, the average latency defined by Definition 3 is an increasing function of the machine size and the problem size. So the scalability $scale(N, N')$ of an implementation–machine combination is less than 1. When $scale(N, N') = 1$, the average latency from the implementation–machine combination is a constant, and is independent of the problem size and the machine size. In this case, by the definition of the efficiency, we have

$$E = 1/\left(1 + \frac{L(W, N)}{W/N}t_c\right).$$

Here the efficiency E and the latency $L(W, N)$ are constants, so the problem size can be expressed as

$$W = kN,$$

where $k = (L(W, N)E/(1 - E))\, t_c$. This indicates that the problem size W increases linearly with the system size N. This case is the ideal algorithm implementation–machine combination, and gives an important quantitative reference for designers of parallel machines, parallel algorithms, and parallel algorithm implementations.

3.3. Using the Latency Metric for Scalability Prediction

Although the scalability defined in Definition 6 is motivated by architecture and system measurements, we can also predicate it through a so-called E-conserved latency function, denoted as $f(N)$, which is an analytical function of the machine size only. From the efficiency definition, we have

$$E = 1/\left(1 + \frac{L(W, N)}{(W/N)t_c}\right). \qquad (3.18)$$

Then the problem size W can be derived from the efficiency definition:

$$W = \frac{ENL(W, N)}{(1 - E)t_c}. \qquad (3.19)$$

Substituting this W function to the latency metric (3.7), we can always get the E-conserved latency function in the following form:

$$L = f(N). \qquad (3.20)$$

Using the E-conserved latency function, the scalabilities can be predicted by any given machine size.

4. PARALLEL ARCHITECTURES AND APPLICATION PROGRAMS FOR THE SCALABILITY EXPERIMENTS

The architectures we used for scalability testbeds are network-based shared-memory systems which use a logical shared address space, where physical memory is distributed. This type of system intends to combine the scalability of network-based architectures with the convenience of shared-memory programming. The shared-memory systems we used were the KSR-1 and the Cerberus shared-memory simulator. We tested three application programs from the Stanford SPLASH set for latency measurement and scalability evaluation on both architectures. We also implemented three other standard numerical algorithms on the KSR-1 for scalability measurement using the latency metric. In this section, we briefly overview the parallel architectures and the application programs for the scalability experiments.

4.1. The KSR-1 System

The KSR-1 [5], introduced by Kendall Square Research, is a hierarchical ring-based shared-memory multiprocessor system with up to 1,088 64-bit custom superscalar RISC processors (20 MHz). A basic ring unit in the KSR-1 has 32 processors. The system uses a two-level hierarchy to interconnect 34 rings (1088 processors). Each processor has a 32-MB cache.

The basic structure of the KSR-1 is the slotted ring, where the ring bandwidth is divided into a number of slots circulating continuously through the ring. The number of slots in the ring is equal to the number of processors plus the number of directory/routers connecting to the upper ring. A standard KSR-1 ring has 34 message slots, where 32 are constructed for the 32 processors and the remaining 2 slots are used for the directory/router cells connecting to level 1 ring. Each slot can be loaded with a packet, made up of a 16-byte header and 128 bytes of data which is the basic data unit in the KSR-1, called a subpage. A processor in the ring ready to transmit a message waits until an empty slot is available, which rotates through a ring interface of the processor.

4.2. The Cerberus Simulator

In order to evaluate cache and cache coherence effects, we also traced one of the application programs on a cache coherent multistage interconnection network multiprocessor simulated by the Cerberus [1] on the TC2000. The simulator constructs a multistage interconnection network architecture similar to the TC2000. Each processor has a 64-K cache where cache line size is 128 bytes. All processors are connected to a large shared memory. An interleaved shared-memory scheme is supported in the memory system. The cache coherence protocol used is a standard full-map directory cache coherence protocol [2]. The simulator collects detailed statistics on execution behaviors including memory access patterns, cache invalidation patterns, and network traffic.

4.3. Three Application Programs from the SPLASH

Three application programs from the SPLASH parallel benchmark set of Stanford [8] have been ported to the KSR-1 and the Cerberus simulator for the latency pattern and scalability evaluation. These parallel programs are a molecular dynamics simulation (Water), a rarefied fluid flow simulation (MP3D), and a Cholesky factorization of a sparse matrix (Cholesky). The program and synchronization structure overview of the three application programs are listed in Table 1.

4.4. Quick Sort (QS)

The basic data structure of the parallel quick sort uses a task queue to allocate tasks onto each thread dynamically. Since the task queue is globally shared and allocated in a critical section, its size and structure are quite sensitive to the computing performance. In our implementation, we used a single queue structure. Initially, each thread gets a task which is a segment of the data to be sorted from the task queue. It then divides it into two smaller segments. The thread puts one of them back into the queue, and sorts the other segment repeatly until the resulted data segment becomes a set of individual elements. Each thread will repeat the same process until the task queue is empty. For detailed information of the algorithm, the interested reader may refer to [6].

The latency in the parallel quick sort algorithm mainly comes from the waiting time for entering critical section of the task queue, and the time for producing the tasks in the critical section.

4.5. Gauss Elimination (GE)

In this parallel Gauss elimination algorithm, multiple threads are mainly used to do column element elimina-

TABLE I
The Program and the Synchronization Structures of the Three Applications

Programs	Type	Size	# itns	cs-len	# lock-acc/itn	# barr.
Water	Iterative	343 mol	10	Short	76,762	8
MP3D	Iterative	50,000 mol	40	Short	101,900	6
Cholesky	Direct	$n = 3,938$	1	Medium	79,941	4

tion simultaneously. Whenever a column element is eliminated, all of the threads must wait at a synchronization point for the main thread to reorder the elements of the next row. The parallel algorithm performs this operation from the first row to the last row in the matrix. For detailed information of the algorithm, the interested reader may refer to [7].

The latency in the Gauss Elimination program mainly comes from the thread synchronization and the data movement among memory modules.

4.6. *All Pairs Shortest Path (APSP)*

The parallel implementation of All Pairs Shortest Path is based on the Dijkstra's sequential algorithm. It uses a path matrix to store the connection relations among nodes. For a given number of threads, the partition of the shortest path searching work among threads can be done statically. Each thread is responsible for finding its shortest path. For detailed information of the algorithm, the interested reader may refer to [6].

The latency of this implementation only comes from the initialization of the program—the creation and join of the threads.

5. MISS LATENCY AND SCALABILITY PREDICTION

In a large shared-memory multiprocessor, memory modules are distributed and an interconnection network is used to connect them together to construct a shared-memory computing environment. Latency defines a delay in time by any type of nonlocal cache/memory access through the interconnection network. All the nonlocal accesses are caused by local access misses. Thus we call the latency *miss latency*. The miss latency in large-scale shared-memory computing is a major performance bottleneck for parallel processing. The major miss latency sources are nonlocal cache/memory searching/access, synchronization locks and barriers, cache coherence overheads, hot spots, and management of cache/memory localities. These are important program events with heavy network activities. We quantitatively measure, evaluate, and analyze the miss latency through an execution pattern study. The program miss latency is a function of the number of processors used for computing, which quantitatively describes the changes of network delay for an application program as the number of processors is increased. By using the execution measurement results, the program miss latency function can also provide its upper bound for the program to achieve maximum speedup with the maximum number of processors. We also use the miss latency as an important factor to evaluate and predict scalability of application programs on network-based shared-memory architectures.

5.1. *Latency and Locality Analysis on the KSR-1*

We ran the three SPLASH programs on the KSR-1 for latency and locality pattern analysis in order to obtain further understanding of the communication patterns inherent in the programs and the architecture. All the performance data are collected by the hardware monitor which is built into the KSR-1. 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 precise and important data to be used for evaluating the execution performance on the KSR-1.

Figures 2, 3, and 4 show execution time distributions for the programs of MP3D, Water, and Cholesky, respectively, on the KSR-1 with different numbers of processors. There are two time columns for each execution in these figures, where the first column represents the average execution time of each processor, and the second column represents the average idle time of each proces-

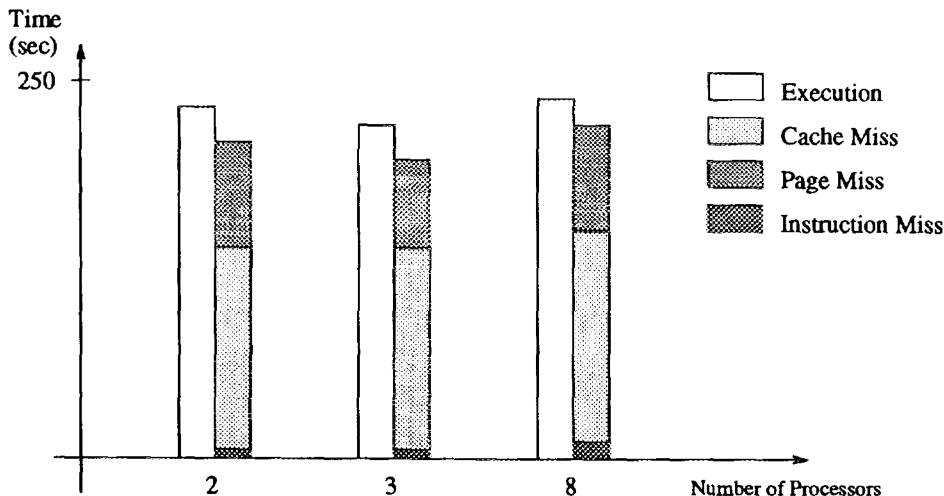

FIG. 2. Execution time distributions for the MP3D program.

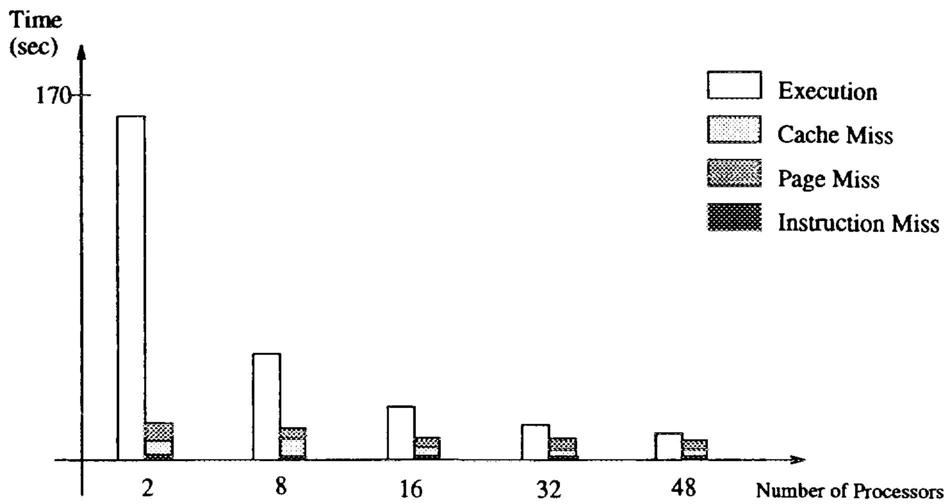

FIG. 3. Execution time distributions for the Water program.

sor. The difference between the first column and the second column is the average effective computing time of each processor. The major problem of the MP3D program running on the KSR-1 comes from low processor locality in execution. The performance of the program has been significantly improved on the KSR-1 by modifying the program data structures for increasing the locality [10]. We chose the MP3D program with poor performance in order to compare to the other two program with higher processor localities.

The idle time is called *CEU_stalls_time* in the KSR-1 system, where CEU is the cell execution unit. During the idle time period, the CEU is stalled; therefore the floating processing unit (FPU) and integer processing unit (IPU) are also stalled. The CEU stalls because the following scenarios occur:

- *data_subcache_miss*. The CEU requests data from its subcache, but data are not there. Therefore *data_subcache_miss* occurs. It takes 23 cycles for subpage to be transferred from the local cache to the subcache via the cache control unit (CCU). Additional stall cycles can occur if write-back occurs, since there is a need to create block descriptors before the data are transferred. They can also occur if some other processor is requesting data from this subcache.
- *cache_subpage_miss*. Data are not in the subcache or the local cache; therefore the cell interconnect unit (CIU) sends a message to the directories at this ring level or upper ring level requesting the subpage. This operation takes about 140–150 cycles on the local ring, plus 23 cycles for an original subcache miss. It takes about 600 cycles for the request from a ring of the upper level.
- *page_miss*. Data are not in the subcache, the subpage is not in the local cache, nor in the directories on the local level and upper level. The operating system needs to create a new page descriptor. This operation takes 163 cycles.

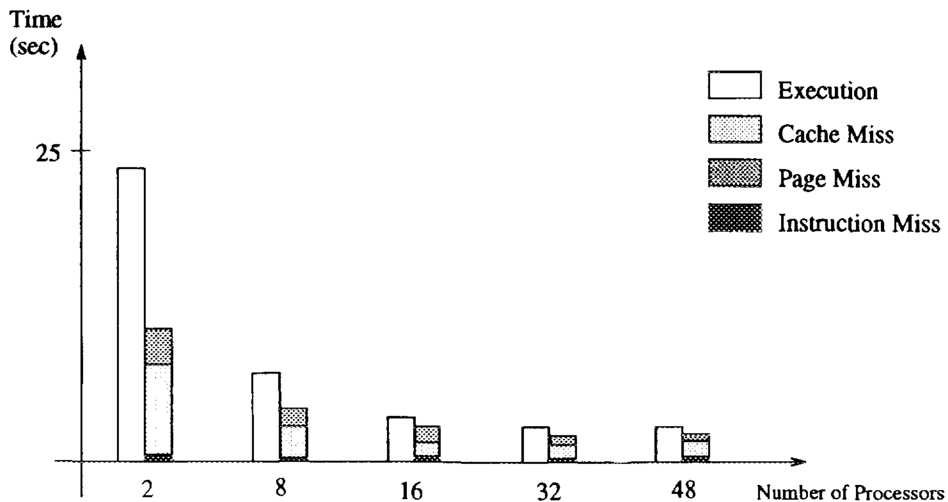

FIG. 4. Execution time distributions for the Cholesky program.

• *cache_ins_instruction_time*. The operating system inserts instructions during program execution for cache instruction misses.

• *io_ins_instruction_time*. The I/O processor inserts instructions during a program execution for I/O instruction misses.

We divide the idle time into three parts: cache miss time including the *data_subcache_miss* and the *cache_subpage_miss*, page miss time, and system instruction miss time including the *cache_ins_instruction_time* and the *io_ins_instruction_time*. Since system instruction misses occasionally happen, it only has a very small percentage in the idle time. Cache miss time is often a large part of the idle time. Page miss time also makes a considerable contribution to the idle time. Figure 2 shows that over 80% total computing time of an MP3D execution is idle when processors are waiting for replacements of cache data, page data, and system instruction misses. In contrast, Figs. 3 and 4 present higher locality patterns in the executions of the Water and Cholesky programs, respectively. The processor idle times are significantly lower because the numbers of various misses are low. Therefore the effective computing times are much higher than that of the MP3D program.

Another important factor which creates program localities is the frequency of nonlocal data movement through the rings. This frequency is high when processor locality of a program is low. Figure 5 presents the three frequency curves of nonlocal data movement in number of packets per μs for the three application programs on different numbers of processors. A communication packet in the KSR-1 is the size of a subpage (128 bytes). The data movement frequency of the MP3D program is significantly higher than that of the Water and Cholesky programs. The high frequency of the MP3D program comes from the result of low processor locality and a large number of cache and page misses in the executions. Tracings of both execution time distributions and frequencies of nonlocal data movement indicate that the Water and Cholesky programs have high hierarchical localities, which can be well exploited by the KSR-1 architecture.

Figure 6 presents average program miss latencies for the three programs, and confirms that the Water and Cholesky programs have high hierarchical localities. The miss latencies of the two programs do not necessarily grow with the number of processors used. Thus, the average distance between an arbitrarily chosen pair of processors does not necessarily grow with the number of processors. This is because communications in executions of these two programs are often conducted on "nearby processors" instead of arbitrary processors, such as two processors on different rings. In contrast, the miss latency of the MP3D program is not just significantly higher, but also grows monotonically with the number of processors.

5.2. *Execution-Driven Simulations and Tracing on the Cerberus*

Performance measurements are limited in their ability to provide insight into dynamic execution patterns of application programs because it may be impossible to capture execution activities at lower system and architecture levels. In addition, measurements may only be used for performance evaluation of an existing system. In order to provide detailed execution patterns of application programs, and to study the effects of some important system modifications, we have conducted execution-driven simulations and tracings using the application programs on large-scale shared-memory multiprocessors.

The architecture we used is a simulated cache coherence multistage interconnection network-based architecture provided by the Cerberus simulator. We use the MP3D as the target parallel program. The size of the problem input was reduced from 50,000 to 3,000 mole-

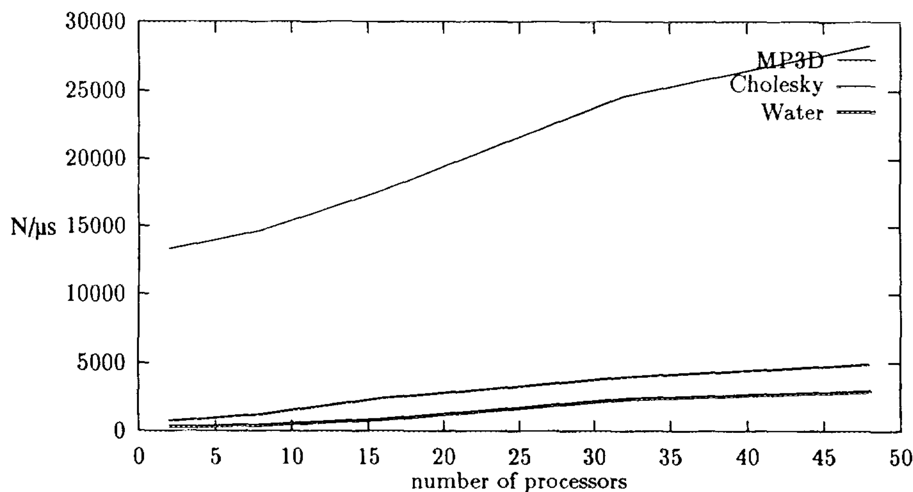

FIG. 5. Frequencies of nonlocal data movement of the three application programs.

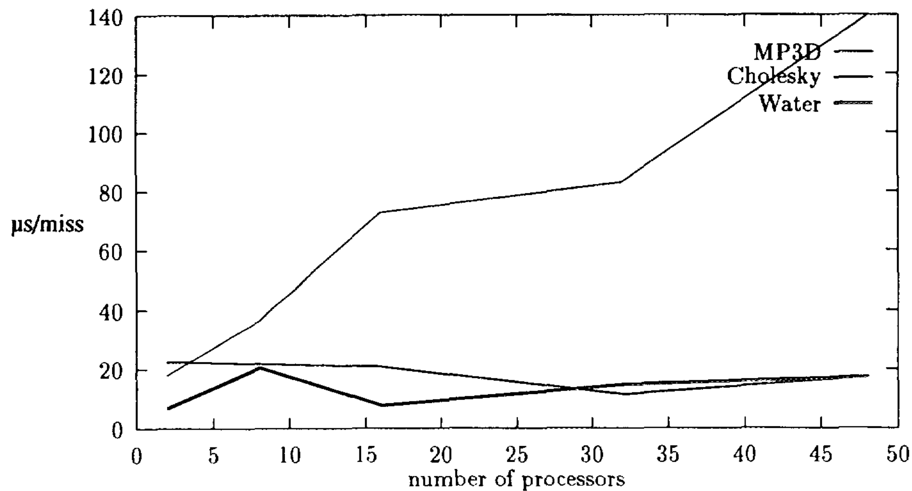

FIG. 6. Average program miss latencies of the three application programs.

cules in order to obtain an affordable simulation time. This reduction did not change the program structure because the structure is independent of the program input size. Before presenting the detailed execution-driven simulation results, we present the execution time measurements of the MP3D program run on the simulated architecture with different numbers of processors. Table II lists the execution times in CPU cycles, utilizing a simple lock and simple barrier schemes. The minimum execution time for the program is printed in boldface. As the number of processors increased to 16, the execution time hit its minimum, after which more processors caused the program to take a longer time to complete. In the following sections, we present our execution-driven simulation results to provide insight into the performance, by investigating the primary factors affecting the parallel scalability of this application program.

5.2.1. Memory Access Patterns and Cache Coherence Effects. Memory access characteristics of the MP3D program running on the simulated architecture are traced. These include shared and private data access, frequency of access, data movement, data locality effects of cache coherence, and other related effects. Table III shows the general memory access information for the MP3D program running on the architecture with a different number of processors. It lists the total number of remote cache/memory accesses through the network, the total number of access-hit rate (read/write), the barrier access-hit rate and the lock access-hit rate. The simulation results in Table III show that the general access-hit rates and the rates of access-hits caused by barriers and locks are slightly changed when the number of processors is increased. However, the number of remote cache/memory accesses is significantly increased as the number of processors is increased. This is because the increase of the number of processors for computing the MP3D program generates more tasks, thus the number of remote cache/memory accesses for task scheduling, data access, and data invalidation is increased accordingly. In addition, the increase of the number of processors in the computation generates more processes to be synchronized at points of barriers and locks; thus the number of remote cache/memory accesses for the same purpose is also increased accordingly. Figure 7 confirms that the increase of the number of remote accesses in the MP3D program almost exclusively comes from the synchronization barriers and locks as the number of processors is increased, while the number of remote accesses for exclusive computing remains at almost a constant level.

TABLE II
Execution Time of the MP3D Program in CPU Cycles on the Cerberus Simulator

# p	8	16	32	64
sim-lock/sim-barr.	1,007,628	**653,326**	904,748	2,417,743

Figure 8 gives the distribution of average access-hit rates among the four groups of multiprocessor sets for executing the MP3D program. The traced data show that the average cache access-hit rate of the system is independent of the number of processors used. This performance result also indicates that the program structure of the MP3D for data accesses on this multistage interconnection network based architecture is reasonably regular.

Table IV gives general cache invalidation information. The average invalidation width defines the average number of cache copies to be invalidated in each invalidation. The invalidation width is only slightly changed as the number of processors is changed. However, the number of invalidations is significantly increased as the number of processors is increased. Table IV also lists the ratio between the number of invalidations caused by barriers (barrier invalidations) and the total number of invalidations in the computation (program invalidations), and the

TABLE III
The General Cache/Memory Access Patterns for the MP3D Program

# proc.	# remote accesses	access-hit rate (%)	barrier hit rate(%)	lock hit rate (%)
8	94,409	84.1	97.1	81.7
16	107,584	83.6	93.0	80.0
32	147,021	85.2	90.9	78.2
64	286,602	92.1	94.2	79.9

ratio between the number of invalidations caused by locks (lock invalidations) and the program invalidations. The large increase in the number of program invalidations mainly comes from the large increase in the number of barrier invalidations as the number of processor is increased. The number of lock invalidations is increased moderately. The percentage of invalidations caused by synchronization barriers/locks is increased from 67 to 87% as the number of processors is increased from 8 to 64.

Figure 9 shows the invalidation distributions for the MP3D program. The distributions are dominated by single invalidations. As the number of processors is increased, the invalidation distribution remains essentially the same. This result is very similar to that of the cache invalidation distribution pattern reported by Gupta and Weber [4], where the MP3D program is executed on another simulated shared-memory architecture. This multiprocessor assumes shared memory partitioned among the processing nodes, infinite caches, and a directory-based cache coherence protocol.

5.2.2. Miss Latencies in Executions. The simulator also conducts two exclusive miss latency measurements. The synchronization miss latency gives the average remote access delay exclusively caused by the synchronization barriers/locks in the program. The computing miss latency provides the average remote access delay exclusively caused by computation without any synchronization process involved. Figure 10 plots the 3 miss latency curves of the MP3D program running up to 64 processors. The major source of network delay comes from the synchronization barriers/locks in the program.

The average program miss latency curve indicates that the upper bound latency for the architecture to achieve maximum speedup of the program is 42 cycles using 16 processors, because 16 is the largest number of processors used to achieve the maximum speedup. The computing miss latency curve can be used as an optimistic bound for the computation, which assumes that there are no synchronization processes in the computation. In this optimistic case, the execution time will hit the minimum when the 30 processors are used. This is because the miss latency of 16 processors (about 42 cycles) is equal to the critical latency of the program (42 cycles). The synchronization miss latency curve can also be used as a pessimistic bound for the computation, which assumes that the programs are dominated by synchronization barriers/locks. If this is the case, the execution time will soon drop to the minimum even when fewer than 8 processors are used. This is because the miss latency of 8 processors (56 cycles) is already larger than the critical latency of the program (42 cycles).

The synchronization effects can be reduced by increasing the input size of the MP3D program. This is because the number of barriers and locks of the program is independent of the size of the problem, and the size of the local computation in each processor is also increased. As

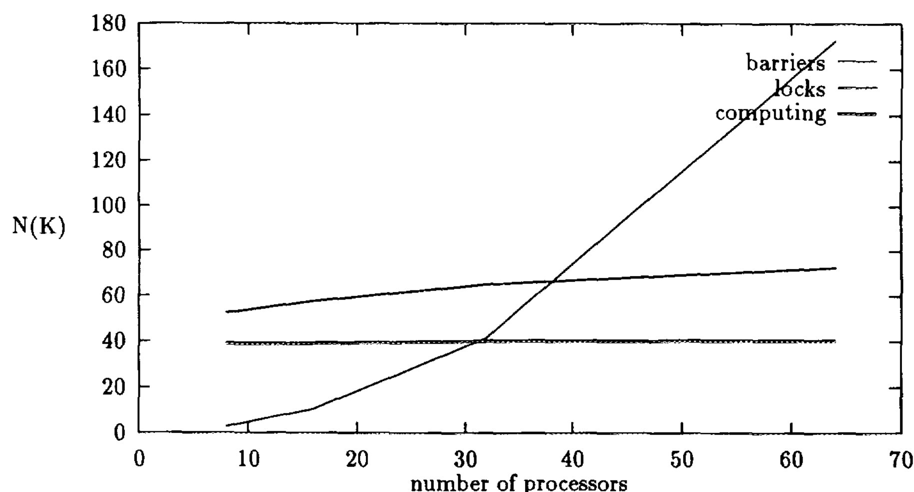

FIG. 7. The numbers of remote accesses caused by barriers, locks, and the normal computing in the MP3D program.

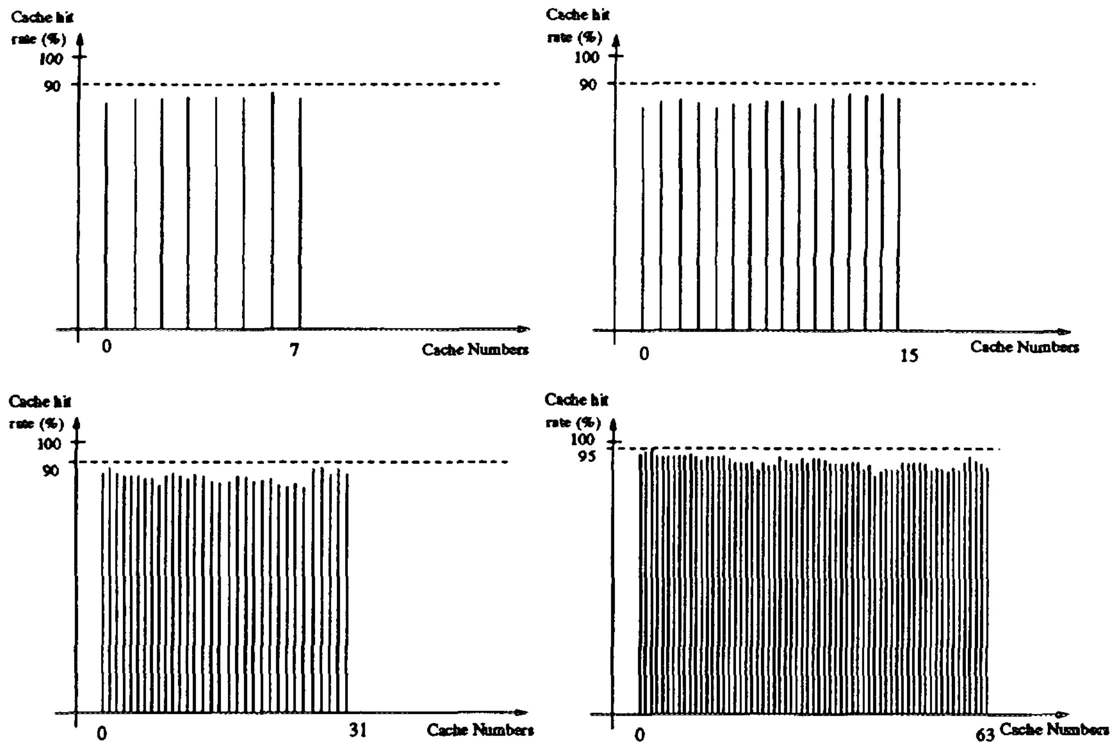

FIG. 8. Cache access-hit distribution patterns for the MP3D program.

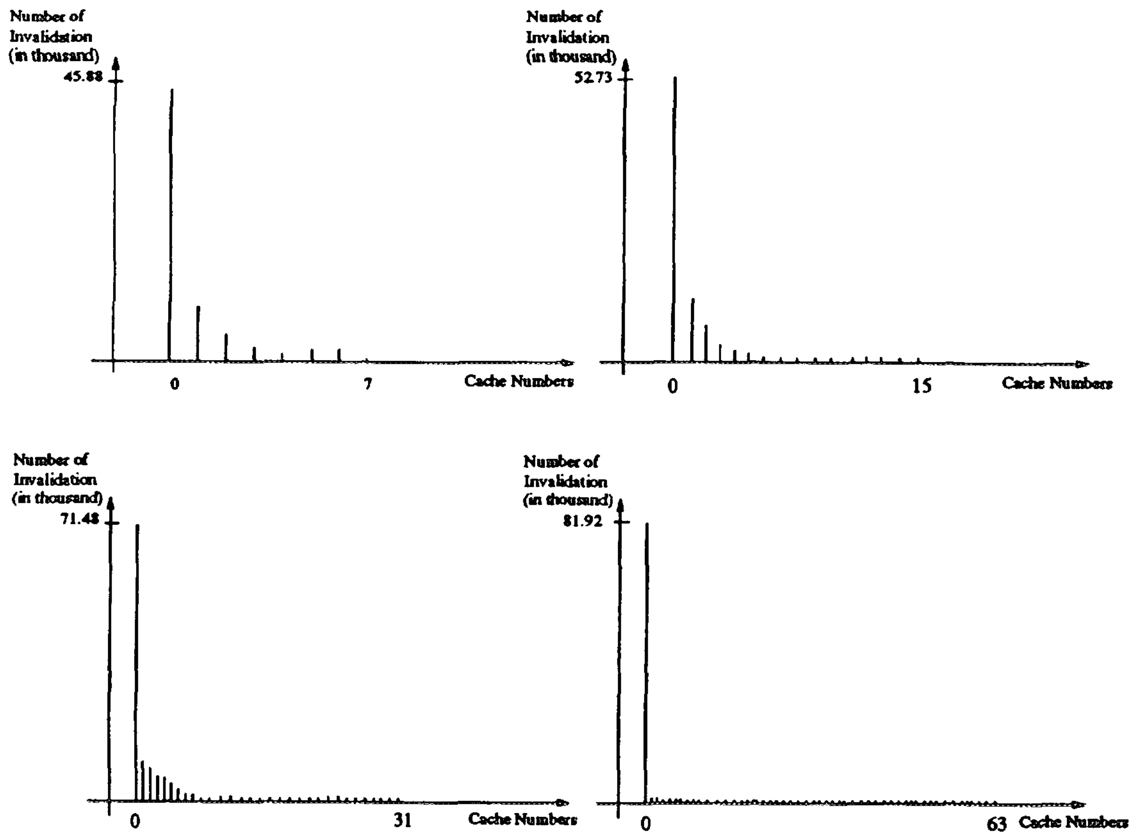

FIG. 9. Cache invalidation distribution patterns for the MP3D program.

TABLE IV
The General Cache Invalidation Patterns for the MP3D Program

# proc.	# prog. invalid.	# of barrier invalid.	# of lock invalid.	avg. invalid. width
8	48,858	1,363 (2.8%)	31,132 (62.7%)	1.12
16	57,526	4,961 (8.6%)	35,603 (61.9%)	2.05
32	76,824	20,287 (26.4%)	38,794 (50.5%)	1.47
64	147,134	84,750 (57.6%)	43,241 (29.4%)	1.72

Fig. 7 shows, the number of remote accesses for exclusive computing is independent of the number of processors used. However, the miss latency of exclusive computing is significantly increased as the number of processors is increased. For example, Fig. 10 shows that the miss latency of exclusive computing for the MP3D program is increased 5 times (from 25 cycles to 124 cycles) as he number of processors is increased from 8 to 64 processors. In other words, no matter how efficient the parallel program is, the built-in latency in the architecture, which is a growing function of the number of processors, will only let the program scale to a certain point, for example, up to 30 processors for this MP3D program in the Cerberus simulator.

5.3. An Experimental Metric of Scalability Prediction

Our execution pattern case studies indicate that the major source for various overheads on a network-based shared-memory multiprocessor architecture, such as synchronization locks/barriers, hot spots, cache invalidations, and remote search/access, can be quantitatively identified by the miss latency. Therefore, the miss latency is a primary factor to measure scalability of a parallel program on a network-based shared-memory architecture. The program miss latency is closely related to the program structure and the architecture used. Since a synchronization-free program, in general, has low network activities and minimum effects from the architecture, the computing miss latency (the latency exclusively caused by the computing of the program) can be used for comparing scalabilities among different architectures. For example, the computing miss latencies of the same program running on different architectures may be used as a factor to distinguish scalabilities of the architectures—the higher the miss latency, the lower scalability the architecture has. The factor of program miss latency may also be used to determine and predict program scalability. By measuring the program execution time running on an architecture with different numbers of processors, and the program miss latencies, the upper bound miss latency for gaining minimum execution time can be determined. This upper bound may be used to determine a justification of the program structure, such as to increase the input size of the program, for the purpose of scaling the program to run on a larger number of processors.

We propose an experimental metric called *miss latency metric* for measuring and predicting a program's capability to effectively utilize an increasing number of processors. The proposed miss latency metric consists of the following steps for studying scalability of a program on a shared-memory multiprocessor architecture:

1. The program with a given small size is run on the multiprocessor system. (The size of the program should be reasonably small enough to reach its minimum execution time when a smaller number of available processors or less are used.) The program miss latency for the maxi-

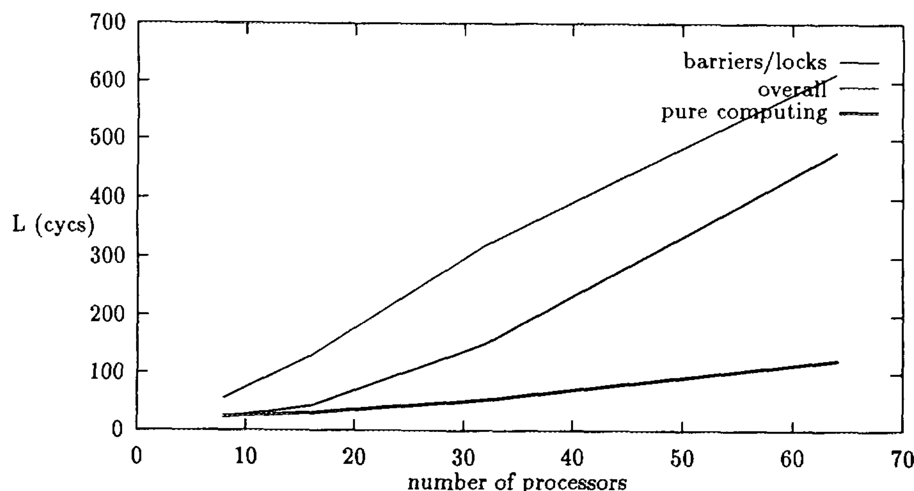

FIG. 10. Miss latencies at different levels of the MP3D program.

TABLE V
The Measured and Predicted Scalability Factors for the MP3D Program on the Cerberus

	L_{max}	$\hat{p}$ (proc.)	$\check{p}$ (proc.)	$\bar{p}$ (proc.)
Software barrier	42	30	6	16
Hardware barrier	42	62	11	32

mum speedup of the execution is measured. This latency, denoted as L_{max}, is the upper bound for this program to reach its minimum execution time for any given problem size on this architecture. We have verified the existence of L_{max} through intensive experiments on the Cerberus simulator and the KSR-1.

2. The pure computing miss latency is measured as a function of the number of processors p, denoted as $L_{comp}(p)$. This function can be used for predicting the optimistic scalability of the program, and it indicates that program execution time would reach its minimum when $\hat{p}$ processors are used for $L_{comp}(\hat{p}) = L_{max}$.

3. The synchronization miss latency is measured as a function of the number of processors p, denoted as $L_{sync}(p)$. This function can be used for predicting the pessimistic scalability of the program, and it indicates that program execution time would soon reach its minimum when $\check{p}$ processors are used for $L_{sync}(\check{p}) = L_{max}$.

4. An average miss latency function can be estimated based on an average of $L_{comp}(p)$ and $L_{sync}(p)$, denoted as $L_{avg}(p)$, which is a predicted miss latency function for the program. Our experiment experiences indicate that the simple average of $(L_{comp}(p) + L_{sync}(p))/2$ presents a close approximation of the program miss latency measured in executions. The slope of $L_{avg}(p)$ gets smaller by increasing the size of the problem. The smaller the $L_{avg}(p)$, the more scalable the parallel system is considered. The predicted scalability of the program can be determined by comparing the $L_{avg}(p)$ function and L_{max}. This would show that program execution time reaches its minimum when $\bar{p}$ processors are used for $L_{avg}(\bar{p}) = L_{max}$.

The measured and predicted scalability factors of the MP3D program on the Cerberus are listed in Table V.

Using the same experimental metric, we can predict the scalability for each of the three programs. The MP3D program would scale to, at most, eight processors with very limited execution time reduction. We expect that both the Water and Cholesky programs scale to a large number of processors, provided the sizes of the problems are also increased on the KSR-1. Table VI lists the measured and predicted scalability factors of the three programs on the KSR-1.

6. MEASURING AND EVALUATING THE SCALABILITIES BY USING THE LATENCY METRIC

In the previous section, we showed by experiments that various latencies are major factors affecting program performance and scalabilities. In this section, we measure and evaluate the scalabilities by using the latency metric proposed in Section 3.

The latency metric is mainly concerned with the average latency increment when both the sizes of the problem and the machine are adjusted to keep the efficiency as a constant. In this section, we present measurement methodology of the latency metric, and report the scalability measurements and evaluation on the KSR-1.

6.1. *Measuring Methodology of the Latency Metric*

Before measuring and evaluating the latency, we need to experimentally determine both the sizes of the problem (W and W'), and the system (N and N') for a given efficiency constant E. Then, the E-conserved latencies $L(W, N)$ and $L(W', N')$ can be either calculated or measured for determining the scalability. Figure 11 gives the basic testing process to determine the problem size for a given efficiency running on a given number of processors.

To effectively and precisely determine the average latency $L(W, N)$ is the key to the measurement and evaluation of the scalability. The following three methods can be used:

Method 1. The average latency can be determined by (3.6) in Section 3.

$$L(W, N) = T_{para}(N, W) - \frac{T_{seq}(W)}{N},$$

TABLE VI
The Measured and Predicted Scalability Factors for the Three Programs on the KSR-1

	L_{max} (μs)	$\hat{p}$ (proc.)	$\check{p}$ (proc.)	$\bar{p}$ (proc.)
Water	40	very large	moderate	very large
MP3D	40	14 proc.	2 proc.	8 proc.
Cholesky	40	very large	moderate	very large

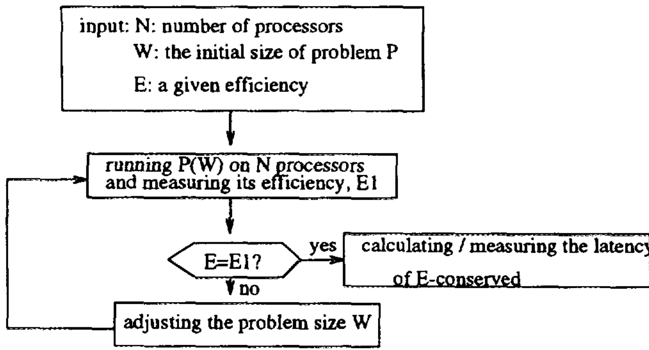

FIG. 11. The testing process to determine the problem size for a given efficiency constant running a given number of processors.

where $T_{para}(N, W)$ is the measured parallel computing time for solving the problem of size W on N processors, and $T_{seq}(W)$ is the measured sequential computing time for solving the same problem on a single processor. The relationship between the efficiency and the measured computing times is

$$E = \frac{T_{seq}(W)}{NT_{para}(W, N)},$$

which may be used as a reference to adjust the problem and system sizes under an E-conserved condition. This method is simple, and does not require any special performance monitor tools to trace program and architecture latency behaviors. There are two disadvantages to this method. First, the sequential time $T_{seq}(W)$ may not be obtained for large application programs due to limited memory space and limited computing power on a single processor in a parallel system. Second, the average latency determined by the above formula is an approximate value, and may not be precise enough.

Method 2. The average latency data can be measured and collected by using software instrumentation which inserts the trace codes at certain important points, such as the synchronization, remote accesses, and cache coherence operations in the program. The advantage is that the latency data may be more precisely and flexibly obtained. But if the overhead introduced by the instrumentation is not reasonably small, the program execution speed may slow down significantly.

Method 3. Hardware monitor can precisely get the latency data with low overhead costs. We used Pmon, a hardware monitor on the KSR-1, to collect the latency data.

The relationship between the efficiency and the parallel computing time can be determined by

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

which may be used as a reference to adjust the problem and system sizes under an E-conserved condition in Methods 2 and 3.

6.2. *Measurement of the Scalabilities Using the Latency Metric*

We measured and evaluated the scalabilities of the three application programs using the latency metric on the KSR-1. The three programs are Quick Sort, Gauss Elimination, and All Pairs Shortest Path. We show that the degree of the program scalability is dependent on program latency which is affected by program structures, program locality, and program task granularity.

A cache/memory miss latency, simplified as miss latency, defines an average time between when a remote cache/memory access (read/write) is requested and when the desired access operation is done. As we described in the previous section, the miss latency covers the overheads of synchronization, cache coherence, remote data accesses, and other events with heavy network activities. We used Pmon, a hardware monitor on the KSR-1 to trace the total number of access-miss operations and calculate the average miss latency for running a program on the architecture with different numbers of processors. We call this measured latency the program miss latency. The program latency measurements for the three application programs are plotted in Figs. 12–14, which show that the program miss latency is a function of the number of processors used for computing. It also quantitatively describes the changes of network delay for an application program as the number of processors is increased.

The scalability results are listed in Tables VII–XII, where the processor numbers listed in the first column in each table are the N, and the processor numbers listed in the first row in each table are the N'. (See the latency metric definition in Section 3).

6.3. *Scalability and Program Structures*

We implemented two versions of Dijkstra's APSP algorithms. The first, denoted as APSP1, includes parallel initialization part in the code, and the second, denoted as APSP2, does not. In order to keep the efficiency E as a constant of 0.25, the problem size of the APSP1 program was adjusted from $W = 12$ on 2 processors up to $W = 229$ on 60 processors. For the same efficiency constant, the problem size of the APSP2 program was adjusted from $W = 10$ on 2 processors to $W = 259$ on 60 processors.

The measured program latency curves plotted in Fig. 12 indicate that the program with parallel initialization becomes effective and has lower average delays when more than 48 processors are used.

The latency scalability results by measurements on the KSR-1 in Tables VII and VIII show that the APSP with parallel initialization is more scalable than that without parallel initialization in most cases. The measurements

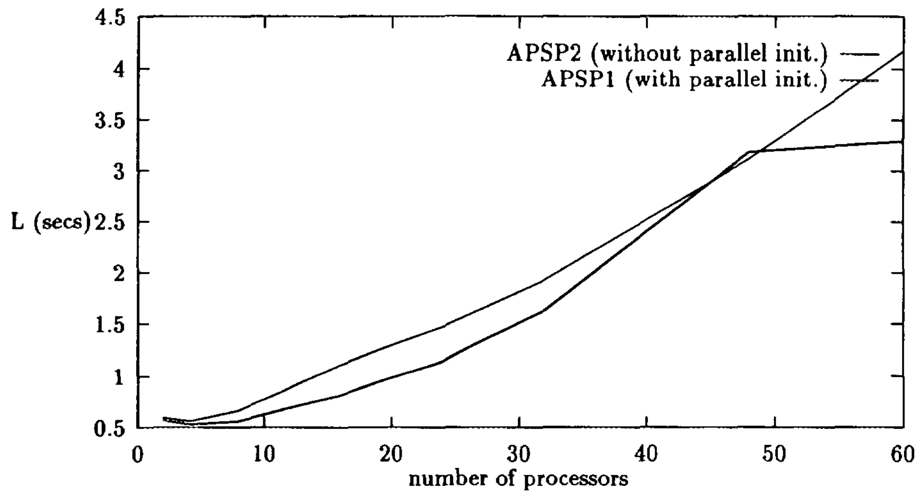

FIG. 12. The latency comparison between the two implementations of APSP by keeping the efficiency as a constant of 0.25.

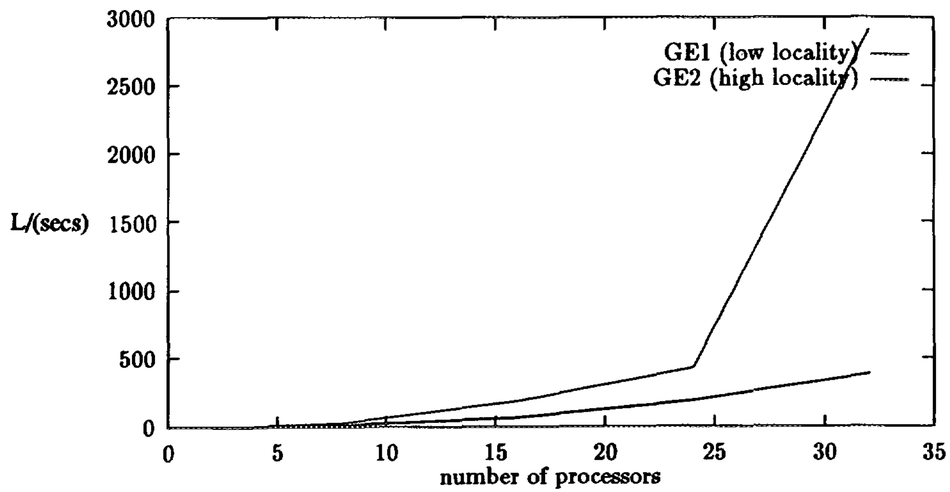

FIG. 13. The latency comparison between the two implementations of GE by keeping the efficiency as a constant of 0.25.

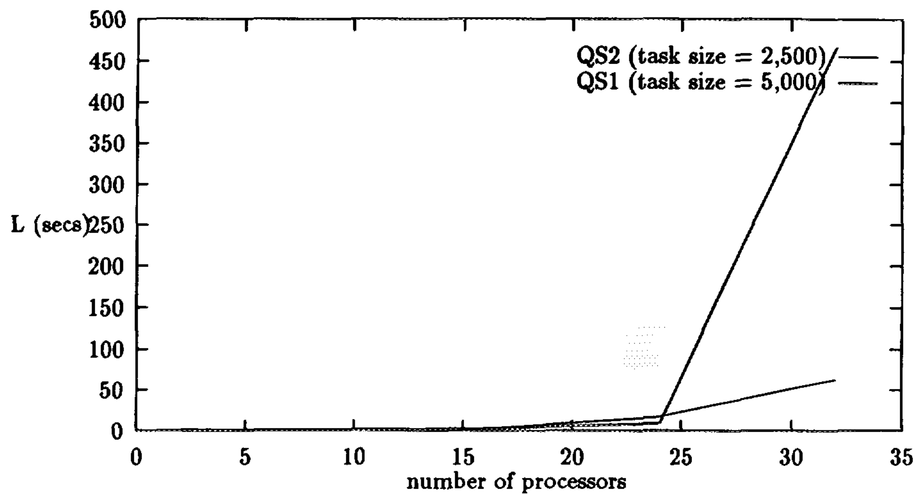

FIG. 14. The latency comparison of the two implementations of QS by keeping the efficiency as a constant of 0.25.

TABLE VII
The Scalability of the Initialization Parallelized APSP Program by Keeping the Efficiency as a Constant of 0.25

Processors	2	4	8	16	24	32	48	60
2	1.0000	0.9769	0.9103	0.6967	0.4958	0.3493	0.1795	0.2042
4		1.0000	0.9381	0.6469	0.4603	0.3243	0.1667	0.1896
8			1.0000	0.6896	0.4907	0.3457	0.1777	0.2021
16				1.0000	0.7116	0.5014	0.2576	0.2930
24					1.0000	0.7045	0.3620	0.4118
32						1.0000	0.5139	0.5845
48							1.0000	0.9375
60								1.0000

TABLE VIII
The Scalability of the Initialization Nonparallelized APSP Program by Keeping the Efficiency as a Constant of 0.25

Processors	2	4	8	16	24	32	48	60
2	1.0000	0.9950	0.8902	0.5323	0.4018	0.3069	0.1904	0.1421
4		1.0000	0.8402	0.5024	0.3792	0.2897	0.1797	0.1342
8			1.0000	0.5980	0.4513	0.3448	0.2139	0.1597
16				1.0000	0.7548	0.5766	0.3577	0.2670
32					1.0000	0.7639	0.4739	0.3538
24						1.0000	0.6204	0.4631
48							1.0000	0.7465
60								1.0000

TABLE IX
The Scalability of GE1 by Keeping the Efficiency as a Constant of 0.25

Processors	2	4	8	16	24	32
2	1.0000	0.163387	0.033667	0.005043	0.002237	0.000326
4		1.0000	0.206053	0.03087	0.013693	0.001997
8			1.0000	0.149814	0.066453	0.009692
16				1.0000	0.443575	0.064695
24					1.0000	0.145850
32						1.0000

TABLE X
The Scalability of GE2 by Keeping the Efficiency as a Constant of 0.25

Processors	2	4	8	16	24	32	48	60
2	1.0000	0.2923	0.0551	0.0114	0.0043	0.0021	0.0014	0.0008
4		1.0000	0.1885	0.0389	0.0145	0.0073	0.0049	0.0028
8			1.0000	0.2065	0.0771	0.0390	0.0258	0.0148
16				1.0000	0.3737	0.1887	0.1252	0.0718
24					1.0000	0.5051	0.3350	0.1921
32						1.0000	0.6631	0.3803
48							1.0000	0.5735
60								1.0000

TABLE XI
The Scalability of QS with Task Size of 5000 Elements by Keeping the Efficiency as a Constant of 0.25

Processors	2	4	8	16	24	32
2	1.0000	0.9419	0.6957	0.2204	0.0329	0.0090
4		1.0000	0.6677	0.2116	0.0316	0.0086
8			1.0000	0.3169	0.0473	0.0129
16				1.0000	0.1493	0.0408
24					1.0000	0.2732
32						1.0000

give an example of how program structures can affect the computing scalability.

6.4. Scalability and Program Locality

Two versions of Gauss Elimination programs, denoted as GE1 and GE2, are implemented on the KSR-1. GE2 exploits more processor locality than does GE1. The basic differences in the two programs are as follows. In an iteration of GE1, each processor is responsible for the elimination of a new row which is eliminated by another processor in the previous iteration. Therefore the processor must first access the new row remotely, then do the elimination. However, in GE2, the elimination of a row is statically scheduled onto a fixed processor, so that the number of remote accesses is reduced and the locality of the program is enhanced. In order to keep the efficiency E as a constant of 0.25, the problem size of the GE1 program was adjusted from $W = 59$ on 2 processors to $W = 3800$ on 32 processors. For the same efficiency constant, the problem size of the GE2 program was adjusted from $W = 55$ on 2 processors to $W = 1970$ on 32 processors and $W = 3317$ on 60 processors. Unfortunately GE1 could only be scaled up to 32 processors on the KSR-1.

The measured program latency curves plotted in Fig. 12 indicate that the program with better locality has significant lower average network delays.

The calculated scalabilities for GE1 and GE2 based on the measured program latency are listed in Tables IX and X, which show the effectiveness of the locality to the computing scalability.

6.5. The Effects of the Task Size in Dynamically Scheduling

Two versions of Quick Sort algorithms are implemented: QS1 with a task size of 5000 elements and QS2 with a task size of 2500 elements. In order to keep the efficiency E as a constant of 0.25, the problem size of the QS1 program was adjusted from $W = 4{,}866$ on 2 processors to $W = 9{,}753{,}184$ on 32 processors. For the same efficiency constant, the problem size of the QS2 program was adjusted from $W = 3483$ on 2 processors to $W = 53{,}769{,}503$ on 32 processors.

The measured program latency curves plotted in Fig. 14 indicate that the program (QS1) had significantly lower average network delays by increasing the task size.

The calculated scalabilities of the two programs based on the measured latencies are listed in Tables XI and XII. As we expected, QS1 is more scalable than QS2 due to the fact that QS1 uses a larger task size to reduce the network latency.

Comparing the scalabilities among the above three sets of algorithms, we can get the following scalability relations among these algorithms on the KSR-1:

$$APSP1 > APSP2 > QS1 > QS2 > GE2 > GE1.$$

Here ">" represents "more scalable than."

7. SUMMARIES AND CONCLUSIONS

We present the latency metric, an experimental method for measuring and evaluating the program and

TABLE XII
The Scalability of QS with Task Size of 2500 Elements by Keeping the Efficiency as a Constant of 0.25

Processors	2	4	8	16	24	32
2	1.00000	0.64500	0.15300	0.06170	0.02300	0.000474
4		1.00000	0.23800	0.09580	0.03570	0.000735
8			1.00000	0.40300	0.15000	0.003090
16				1.00000	0.37300	0.007670
24					1.00000	0.020600
32						1.000000

architecture scalability. We also address analytical relationships among the latency metric, and the isoefficiency function and isospeed metric. Using the latency metric, and other experimental methods, we have conducted an intensive experimental study of scalability of application programs on two network-based shared-memory multiprocessor systems, the Cerberus and the KSR-1. In this study, we quantitatively identify and evaluate the major performance bottleneck sources for parallel program scalability—nonlocal cache/memory searching/access, synchronization locks and barriers, cache coherence overheads, hot spots, and management of cache/memory localities. Current work includes developing a software tool using the latency metric for measuring and evaluating the program and architecture scalability.

ACKNOWLEDGMENTS

We thank all our colleagues at the High Performance Computing and Software Laboratory for many interesting and informative discussions on this research project. Qian Ma implemented and ran a few programs for part of the scalability measurements. George Butchee carefully read the paper and made helpful comments.

REFERENCES

1. Brooks, E. D. III, Darmohray, G. A., and Axelrod, T. S. *The Cerberus user manual*. Technical Report, Lawrence Livermore National Laboratory, 1991.
2. Chaiken, D., *et al.* Directory-based cache coherence in large-scale multiprocessors. *IEEE Comput.* **23,** 6(1990), 49–58.
3. Grama, A., Gupta, A., and Kumar, V. Isoefficiency function: a scalability metric for parallel algorithms and architectures. *IEEE Parallel Distrib. Technology* **1,** 3(1993), 12–21.
4. Gupta, A., and Weber, W. Cache invalidation patterns in shared-memory multiprocessors. *IEEE Trans. Comput.* **41,** 7(1992), 794–810.
5. Kendall Square Research, *KSR-1 Technology Background,* 1992.
6. Leighton, F. T. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann, San Mateo, CA, 1992.
7. Moler, C. Matric computation on distributed memory multiprocessors. In Heath, M. T. (Ed). *Hypercube Multiprocessors 1986*. SIAM, Philadelphia, 1986, pp. 181–195.
8. Singh, J. P., Weber, W-D., and Gupta, A. SPLASH: *Stanford parallel applications for shared-memory*. Technical Report, Department of Computer Science, Stanford University, 1991.
9. Sun, X., and Rover, D. T. Scalability of parallel algorithm-machine combinations. *IEEE Trans. Parallel Distrib. Systems* **5**, 6 (1994) 599–613.
10. Zhang, X., He, K., and Butchee, G. Execution behavior analysis and performance improvement in shared-memory architectures. *Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing*. IEEE Computer Society Press, Silver Spring, MD, December 1993, pp. 23–26.

Received May 1, 1993; revised February 11, 1994; accepted March 28, 1994

XIAODONG ZHANG is an associate professor of computer science and director of the High Performance Computing and Software Laboratory at the University of Texas at San Antonio. He has also been a visiting faculty member at Rice University and the Texas A & M University. He received his B.S. degree in electrical engineering from Beijing Polytechnic University, China, in 1982, his M.S. degree in Computer Science, and his Ph.D degree in computer science from the University of Colorado at Boulder, in 1985 and 1989, respectively. His research interest is primarily in the areas of parallel and distributed computation, parallel system performance evaluation, and numerical analysis for solving nonlinear equations and optimization problems. He serves on the Editorial Board of *Parallel Computing*. He is an ACM National Lecturer. He is a member of the *ACM*, the *IEEE Computer Society*, and the *SIAM*.

YONG YAN received his B.S. and M.S. degrees in computer science from Huazhong University of Science and Technology, Wuhan, China, in 1984 and 1987, respectively. He has been an associate professor of computer science there since 1990. Currently he is a visiting scientist at the High Performance Computing and Software Laboratory at the University of Texas at San Antonio. His research interest is in the areas of parallel and distributed computing, performance evaluation, operating systems, and algorithm analysis.

KEQIANG HE received his B.S. degree in computer science from Huazhong University of Science and Technology, Wuhan, China, in 1984, and his M.S. degree in computer engineering from Chunqing University, Chunqing, China, in 1987. He received his M.S. degree in computer science at the University of Texas at San Antonio in 1994. He is a recipient of a Southwestern Bell graduate fellowship in 1992 and 1993. He is currently employed as a system analyst at Cartotech Inc.