JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **18**, 231–241 (1993)

MIN-Graph: A Tool for Monitoring and Visualizing MIN-Based Multiprocessor Performance*

XIAODONG ZHANG

Computer Science Program, University of Texas at San Antonio, San Antonio, Texas 78249

NAGA S. NALLURI

American Airlines Decision Technologies, Dallas, Texas 75261-9616

AND

XIAOHAN QIN

Department of Computer Science and Engineering, University of Washington, Seattle, Washington 98195

A Multistage Interconnection Network (MIN) makes it possible to build large-scale shared-memory multiprocessor systems. To provide insight into dynamic system performance, we have developed an integrated data collection, analysis, and data visualization environment for a MIN-based multiprocessor system, called MIN-Graph. MIN-Graph is a graphical instrumentation monitor to aid users in investigating performance problems and in determining an effective way of exploiting the high performance capabilities of interconnection network multiprocessor systems. Our monitor measures, analyzes, evaluates, and displays the events, performance, and overhead of interprocessor communication, process scheduling, remote-memory access, network contention, and other processes on a MIN-based multiprocessor. The graphical monitor is X-window based and implemented on the BBN GP1000 and the BBN TC2000. © 1993 Academic Press, Inc.

1. INTRODUCTION

A multiprocessor with a Multistage Interconnection Network (MIN) can be constructed as a Non-Uniform Memory Access (NUMA) architecture. NUMA defines a shared-memory multiprocessor environment in which the times to access shared-memory from different processors are not identical. In a remote/local NUMA architecture, such as the BBN GP1000 and the TC2000, a particular processor can access some memory, such as its local memory, faster than other memory, such as the memory local to other processors. Since the memory modules are distributed and an interconnection network is used to connect a processor and its local memory with all other processors and their memory modules, this type of architecture can also support a distributed-memory multicomputer environment in which data sharing and communication are conducted by message-passing through the network. In a MIN-based architecture, a shared-memory environment is built through a distributed architecture—each processor has its local memory and is also able to access all other memory modules through a switching network. Therefore, a MIN-based architecture can scale upward to a large number of processors in a shared-memory multiprocessor design.

Complex interactions between many processors and memory modules through a MIN result in significantly larger space of possible performance behavior and potential performance bottlenecks. Programming a MIN-based multiprocessor system requires that users understand the performance characteristics of the system and be able to modify their programs and algorithms accordingly. A multiprocessor instrumentation should develop monitoring tools to aid users in measuring, evaluating, and tuning the parallel programs running on the parallel systems. Our performance visualization tool, the MIN-Graph, is based on software monitoring: the measured data are collected during the first phase of the monitor operation. The second phase, the data analysis, allows users to access the data. Finally, the third phase drives the graphical tool to display different performance activities.

In Section 2, we present programming models on a MIN-based architecture and the target machines, the BBN GP1000 and the TC2000. In Section 3, we give the design objectives and methods and briefly overview the

* This work has been supported in part by the National Science Foundation under research grants CCR-9008991, CCR-9102854, and instrumentation grant DUE-9250265, and the U.S. Air Force under Agreement FD-204092-64157. Part of the experiments were conducted on the GP1000 at BBN Advanced Computer Inc., and on the BBN TC2000 at Lawrence Livermore National Laboratory.

related work. The purpose of this section is to address why the MIN-Graph is unique for MIN-based architectures. The software structure and the events of interest are discussed in Section 4. Section 5 describes the instrumentation preprocessor, the issues of event measurement and analysis, and overhead reduction in the monitor. Section 6 presents the performance visualization for different system and application programs including spin–lock synchronization, sparse nonlinear system solving, and sequence pattern searching on the BBN GP1000 and TC2000. We also show how performance of the computational programs is improved with the aid of the MIN-Graph. The work is summarized in Section 7.

2. NUMA PROGRAMMING MODELS AND THE TWO TARGET ARCHITECTURES

2.1. Programming Models

Programming models on a MIN-based multiprocessor architecture can be classified into three types: distributed-memory, shared-memory with static scheduling, and shared memory with dynamic scheduling. Under the distributed programming model, each node is viewed as a complete computer supported by a processor, a local memory, and some I/O facilities physically on one board which connects to all other nodes in the system. An important factor in the efficiency of the distributed-memory model is the effectiveness with which data can be exchanged among its many nodes. The model of shared memory with static scheduling provides shared-memory accesses to all processors, with program code and private data stored in local memory. Programming requires partitioning the application across all the nodes at load time, which exploits the processor locality best but provides no dynamic process scheduling at run time. The shared-memory synchronization primitives such as barriers are supported for synchronizing processors at the end of a group of parallel tasks. The model of shared memory with dynamic scheduling implies that from a user's point of view all processors share a single pool of memory with different memory access times. A scheduling mechanism is supported to schedule the processes dynamically among the processors at run time.

2.2. The BBN GP1000

One of our target multiprocessor systems, the BBN GP1000 [2–4], based on the original Butterfly architectures, is a MIMD system and incorporates up to 256 Motorola 68020 processor nodes connected via a MIN. The network is composed of 4×4 switches to form a $p \log_4 p$ switching interconnection, where p is the number of processors connected. The bandwidth of each path of a switch is 32 Mb per second. Each processor node has a 68851 paged memory management unit for virtual memory processing. The memory of the machine is shared among all processors in this way—each processor node includes 4 MBytes of memory that can be accessed from any processor in the system via the network. Each memory location is physically local to its host processor, although globally accessible by any processor in the system. A memory access from a processor to its own memory through the direct path is called *local access*, and a memory access from a processor to other memory modules through the network is called *remote-memory access*. The time ratio between a local access and a remote access with no other processors active on the GP1000 is up to 1 : 15 depending on the types of the access [6, 20].

2.3. The BBN TC2000

The BBN TC2000 [5] is the latest and the most powerful member of the BBN family, which supports up to 512 Motorola 88100 processor nodes. The butterfly switch, composed of 8×8 switches, is used for the network. Each processor node includes 16 MBytes of memory that can be accessed from any processor in the system via the network. The bandwidth of each path of a switch is 38 Mb per second. Besides more powerful nodes and a larger scale system, the TC2000 has some other major differences from the GP1000. Each processor node in the TC2000 has a Motorola 88200 paged memory management unit for virtual processing. Each processor node provides a 32K byte code cache and a 16K byte data cache. The TC2000 avoids the cache-coherence problem by disallowing caching of shared data. This scheme caches private data, shared data that are read-only, and instructions, while references to modifiable shared data bypass the cache. In addition, an interleaved shared-memory scheme is supported as an option in order to reduce memory contention in computing.

3. WHY THE MIN-GRAPH IS UNIQUE FOR A MIN-BASED ARCHITECTURE

A number of parallel monitoring tools have been developed in a past few years. SHMAP [7], developed at the University of Tennessee to aid in the design, implementation, and understanding of matrix algorithms for parallel processing, is an example of a software monitor. SHMAP uses software to simulate hierarchical memory organization and different cache replacement policies. Trace code to record events of interest is inserted directly into the user's program. The events are automatically measured by the inserted code when the instrumented application programs run. Then the collected event data are analyzed and graphically displayed for studying memory access patterns, different cache replacement algorithms, and effects of multiprocessors on matrix algorithms. However, SHMAP does not directly monitor parallel programs on a shared-memory machine but uses simulations on a sequential machine. On the other hand, Reilly [17] developed a hardware monitor for the multiprocessor VAX system M31 designed and built by DEC Advanced VAX System Group. The M31 was designed to allow experimentation in the area of parallel program development and system design. To serve this purpose the M31's design includes support for an extensive performance monitoring system. The events of interest include fine grained events such as a cache hit or miss or a bus stall, system events such as process creation and process termination, and application events such as entry to or exit from a subroutine or access to a data structure. Initially, the M31 instrumentation system was to provide a completely noninvasive and passive hardware event collector. However, a number of problems such as additional complexity and very expensive cost have made the purely passive monitor unwieldy for software performance analysis. Malony and Reed [13], using the Intel iPSC/2 as a target distributed-memory architecture, have developed a hybrid performance instrumentation tool integrating data collection, analysis, and visualization in one box, called Hyperview. Three levels of instrumentation are developed: application instrumentation (by modifying Free Software Foundation's GNU C compiler to emit instrumented code), operating system instrumentation, and hardware monitoring. The hardware-assisted monitoring system is built up to circumvent the high overhead of software tracing and lack of a global clock with high resolution. PICL, a Portable Instrumented Communication Library developed at the Oak Ridge National Laboratory [10], is a software monitor mainly monitoring message-passing events. Since both the Hyperview and the PICL were originally built for hypercube architecture, which solely supports a message passing paradigm, the events of interest mainly focus on communications among processor nodes. It is possible that PICL can, as the authors claim, be implemented on any shared-memory or distributed shared-memory machines which support alternative message-passing programming paradigm. On pure shared-memory machines, there will be a significant performance penalty in this approach. On a MIN-based multiprocessor system, sources of major parallel processing overhead include imbalanced workload and scheduling and remote-memory accesses, since shared-memory programming models are commonly used. However, neither the Hyperview nor the PICL provides support for instrumenting remote-memory access and other related shared-memory activities on a MIN-based multiprocessor.

In contrast, our performance monitor is developed to monitor and visualize performance of a MIN-based multiprocessor. On a MIN-based architecture, communication between processors, unbalanced workload, scheduling, remote-memory accesses, and network contention make major contributions to parallel processing overhead. To help users address an algorithm's inefficiency and adjust the program accordingly, the MIN-Graph presents (1) the degree of parallelism (or concurrency) of a running program; (2) the workload in each processor; (3) message-passing communications among processors; (4) remote-memory accesses; and (5) network contention involved in computations.

4. SYSTEM STRUCTURE AND EVENTS OF INTEREST

We choose to use software monitoring because this approach does not depend upon any special hardware support, and is enough to detect those events required to present the parallel program status we want. In software monitoring, event detection is done by inserting trace code directly into the user's program before it runs. If extensive source code modifications are required, manual insertion is impractical. Thus we need a program preprocessor to automatically handle trace code insertion. These inserted trace codes are actually function calls which will record event data. All the trace routines are packed into one module to form a trace routine library. Later when the instrumented program is executed, performance data will be collected for analyzing its parallel behavior. After the program event trace is done, the performance results are presented to users. Since the complexity and quantity of data from performance measurements can be overwhelming, efficient representations of parallel performance are needed to allow users to identify bottlenecks easily and quickly. One such a technique is visualization, a method of transforming symbolic data into an intuitive and easy-to-read geometric form. The high information density of a graphical display makes it become almost an indispensable part of the software tools and environments.

MIN-Graph is composed of four parts: (1) an instrumentation preprocessor for event detection, (2) a trace routine library for event measurement, (3) a data analyzer for event analysis, and (4) a performance displayer for visualization. An *application program* becomes an instrumented program after going through the *instrumentation preprocessor*. The *instrumented program* outputs regular results executed by a MIN-based multiprocessor system. The collected *performance data* are studied by the *data analyzer*. Event measurement is actually done when a program runs with inserted tracing code and linked with the *trace routine library*. Finally the *analyzed data* are presented by the *performance displayer* on a graphics supported workstation.

After the system structure and trace methods are decided, the first question about program-based event trace is: What are the events of interest? The MIN-Graph has three basic classes of events:

• Process creation and termination: this event detection is used for tracing load balancing and task scheduling.
• Communication over the interconnection network: this event detection is used for tracing the network traffic and contention.
• Remote memory access: this event detection is used to trace the number of remote-memory accesses, hot spots, and network and memory contention.

5. EVENT INSTRUMENTATION, MEASUREMENT, AND ANALYSIS

5.1. An Instrumentation Preprocessor

The function of the instrumentation preprocessor is to insert tracing recorders into a user's application program to be monitored at compile time. In fact, the 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.

In many cases tracing codes are inserted before or after system calls which generate interesting events. It is efficient for the insertion to be done after the syntax analyzer reduces to a particular "function call" token. But care must be taken to guarantee the semantic consistency between the original parallel program and the instrumented program. As we can see, a function call must be an expression or part of an expression. To keep the program logic unchanged after tracing code has been inserted, we use the concept of the comma expression in C language. For detailed information about the preprocessor implementation, the interested reader may refer to [16].

If we only wanted to trace events generated by system calls, a lexical analyzer would be enough. Every system call can be handled as a "token" plus some mechanisms to recognize its parameters. However, the instrumentation for remote-memory access requires an analysis of data declarations and memory allocation specifications to obtain information about allocation of data to nonlocal memory modules. Therefore it is necessary to use Yacc to analyze data declarations and data allocations and to generate instrumentation codes. Sometimes information such as whether a memory access is a local access or a remote access may not be decided until runtime. But we are still able to statically insert tracing code by specifying parameters in the tracing recorder. Code insertion is invoked in the preprocessor whenever the syntax analyzer reduces to some possible remote allocated variables. When a program is running and a possible remote access happens, then a remote-memory access recorder is called. At this point the remote-memory access recorder will check to see if a real remote access happens, since it can know at this moment on which processor the process is running and on which memory module the access variable is allocated.

5.2. Event Measurement and Analysis

After a user's program is processed by the preprocessor, an instrumented program is produced. When an instrumented program runs, the inserted tracing routines automatically record the event data of interest and send them through a socket to another independent processor where data is analyzed.

One important aspect of a parallel program is the degree of parallelism, which defines the number of processors that are concurrently active. It is possible that one generates more than one process on a processor node (these processes may run under multiprogramming environment). However, at one point in time, only one process can be scheduled on that processor. The degree of an algorithm's parallelism can be obtained by analyzing event data for process creation and termination. Both events record the time they occur and the processor node IDs on which they are running. We maintain a dynamic data structure for the global status of multiprocessor activities.

A "network usage" structure gives a global picture of communication over the interconnection network among processes by explicitly specifying message-passing requests. Here network usage does not include remote-memory access, which implies the communication between processors and memory modules. This part of the monitor particularly supports the distributed-memory programming model to observe communication overhead. Message send and receive events are measured for analysis.

5.3. Overhead Reduction in the MIN-Graph

A key issue in a software monitor is the amount of overhead introduced by the instrumentation. The time spent generating performance data and space used storing the trace output are the overhead in software instrumentation. The overhead affects the behavior of a program by slowing down the execution speed on a MIN-based multiprocessor. The quality of a software monitor is mainly dependent on how well the overhead is handled and reduced. We will next briefly introduce the techniques we have used for reducing the time and space requirements of the run-time tracing code.

(1) *Minimize the Number of Inserting Points.* At each inserting point, trace functions will be called and trace output will be stored. Therefore, a major step to reduce the amount of inserted trace code and saved trace data is to minimize the number of inserting points. As previously described, we only choose necessary performance events for code insertions.

(2) *No Further Remote-Memory Accesses Involved in Tracing.* Since the trace code is directly inserted in user's program, certain distortions will be introduced. To minimize the distortion effect, only data local to the processor are recorded because remote data accesses could severely disturb the performance of the original parallel program. Event analysis and synthesis are used to obtain the global status of the parallel program by combining the sequence of event data.

(3) *Only Generate the Necessary Run-Time Trace Data.* In order to decrease the amount of trace data produced, we limit the trace size by recording only those events that cannot be determined statically or through event analysis and synthesis. For example, by only recording a remote-memory access event we can also obtain the other related events, such as the corresponding network routing path, which then does not need to be recorded at run-time.

(4) *Reduce the Trace Information if Possible.* In many computations, the sheer quantity of events recorded makes storing it impractical or impossible. In addition, many events are not useful for understanding insights of parallel performance. In general, poor performance of a program is caused by too many network transactions. For example, in the model of shared memory with dynamic scheduling, a poorly structured program can generate a huge number of remote-memory accesses in a very short period of time. The most important performance evaluation to the user is not when each remote access happens, but how it happens and how many are generated. We have two options for a user to choose. The first option is designed for a well-structured program with a reasonably limited number of remote-memory accesses, where all remote-memory access events are recorded at every moment when they happen. The second option is designed for those programs producing a large number of remote-memory accesses. The basic data structure for a remote-memory access includes *pid*, *timestamp*, *source_node*, and *destination_node*. If a processor node conducts remote-memory accesses to a destination node again and again, all the recorded data structures are the same except the time stamps. In the second data structure, the time clock is read in a period of time, and the number of remote-memory accesses are precisely counted in the period. This method greatly reduces the time spent in recording and the size of the trace output. The length of the trace period of reading the clock can be adjusted by a user to control the overhead of this type of trace in time and space.

(5) *Compress Trace Data.* A compression algorithm is applied to the trace output before it is written to a local memory buffer. However, the compression may add extra overhead, such as additional copying, CPU, and memory cycles. This overhead is a worthwhile investment when the recorded trace file is huge. Currently in MIN-Graph, data compression is an option for dealing with a large trace data set.

The overhead of the monitor can be measured by comparing the execution of the original program without inserting trace code and the execution of the instrumented program. We have measured the overhead with several application programs which will be described in the next section. The overhead of the MIN-Graph for a well-structured program ranges from 2 to 4 times over the normal execution time. For a poorly structured program, it ranges 10 to 20 times or more over the normal execution time.

6. PARALLEL PERFORMANCE DISPLAY AND TUNING

The visualization of the performance is presented in X-windows. The most difficult question in visualization is not how to do it but what is the best way for a user to assimilate as much critical information as possible from a quick look at the graphic presentations. The graphical performance presentation can give users a clear insight into the general behavior of the whole computing task, which is reflected by the degree of parallelism, processor's workload, network traffic, and remote-memory access.

6.1. *Monitoring and Visualizing Remote-Memory Access and Network Contention in Spin-Lock Synchronization*

In a MIN-based multiprocessor, all the memory modules are shared by all the processors through the network. Before a critical variable or a section is accessed by atomic instructions, a lock is set. This means other processors trying to enter the section must wait until the lock is released. How to wait for one's turn is the major issue for synchronization efficiency. The simplest approach is to spin ("busy–wait") for the lock, which may be implemented by a test-and-set hardware instruction. Obviously, spin–lock wastes processor cycles. Moreover, spin–wait may generate more traffic and consume more communication bandwidth in the network to slow other processors doing useful work, including the one working in the critical section. There are several variations of the spin lock. The basic idea is to set various delay functions after each test-and-set in each processor in order to reduce network and memory contention.

In order to reduce contention in the network and the memory module holding the lock, it would be more efficient that each processor busy-wait only on a locally accessible variable for the lock. Several queueing spinning algorithms have been proposed [1, 14, 21].

Various spin–lock algorithms for spin–locks have been implemented on the GP1000 and the TC2000 for performance evaluation [19]. These algorithms were monitored by the MIN-Graph. Here, we present some of the representative performance results. The basic operations for all the algorithms are that each processor requests lock acquisitions 10 times in a loop. The critical sections for all the experiments were set to 0. In addition, all experiments on both of the GP1000 and the TC2000 were under benchmark modes. Therefore, the systems were solely used for the measurements. The original experiments were run up to 128 processors on both of the BBN machines. Since we are interested in remote-memory accesses and network contention involved in each algorithm, we choose a medium size number of processors (40 processors) without losing generality. (Another reason is to reduce the sizes of the figures.)

Remote memory access patterns for a computation are formed by a two-dimensional array window, where row indices represent the processors in the system, and column indices represent the memory modules to be accessed by the processors in the system. The number in each element of the array (in ith row and jth column) represents the times of remote-memory access from processor i to memory module j. This window is dynamically updated at run time. The final snapshot reports the total number of remote-memory accesses and their distribution.

Network contention is defined as a conflict where more than one message needs to access the same portion of a path at the same time. The events of multiple memory accesses reaching same switches or same memory modules are recorded at different time period. In the contention graph presentation, the horizontal axis represents time, and the vertical axis represents the number of the switch or memory module conflicts happens at the time.

Figure 1 presents the memory access patterns and contention graphs of the spin-lock algorithms: test_and_set, test_and_set with static delay, test_and_set with exponential delay, and a distributed lock. The numbers of memory accesses are not printed in the memory access arrays because some of them have too long digits to place in the arrays. Although the memory access patterns of the three centralized algorithms are the same, the numbers of memory accesses and network contentions are different.

As we know, on a MIN-based multiprocessor, this simple spin–lock can cause a "hot-spot," delaying accesses to the memory module containing the lock location as well as to other memory modules. Processor 0 holds the lock, whose memory module becomes a hot spot because all the other processors try to access the lock through remote-memory accesses. (The memory access pattern array indicates that only the first column presents numbers for remote-memory accesses to processor 0).

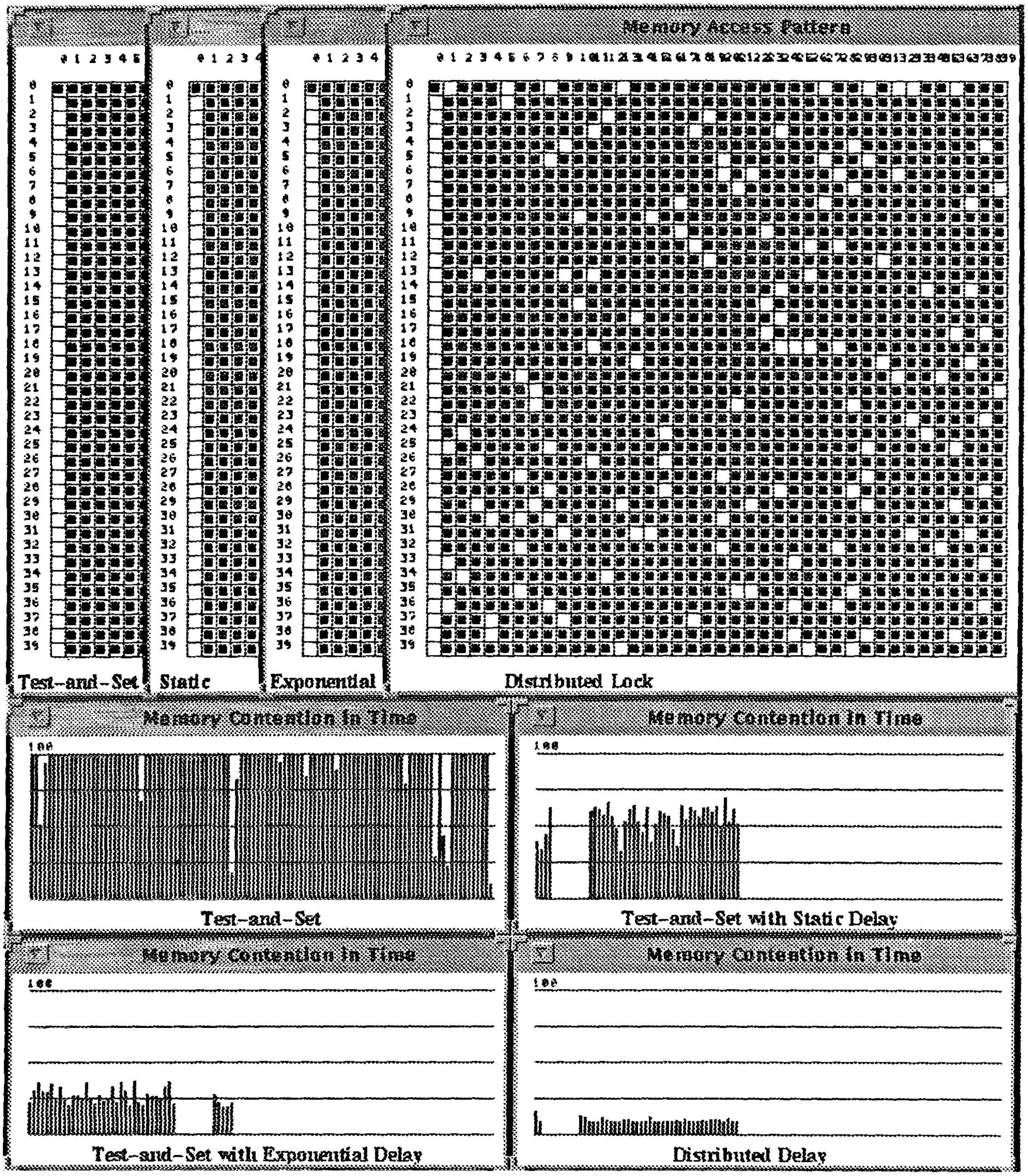
The other two variations are test_and_set with a static delay where a constant timing delay is applied after each polling to the lock, and test_and_set with an exponential delay where the timing delay to access the lock increases exponentially with every failed attempt to acquire the lock. These two locks still belong to the class of centralized algorithms because the lock is allocated in the globally shared-memory module. However, the delay functions reduce the number of remote-memory accesses to the lock. Therefore network transactions are reduced. For example, according to the three remote-memory access patterns presented by the MIN-Graph, the total number of memory accesses in test_and_set is 4682, in test_and_set with a static delay is 1741 (63% reduction), and in test_and_set with an exponential delay is 1012 (78% reduction).

The remote-memory access pattern of the distributed lock no longer centers to the first column. Rather than using a single scalar variable to represent the lock, this algorithm represents scalars as an array where each element in the array resides on a different processor. Each processor spins locally until the processor arriving the lock before it releases the lock and transfers the control to it. Therefore, processor 0 no longer serves as a "hot spot," and remote-memory accesses are scattered among all the processors. According to the memory access patterns monitored by the MIN-Graph, the total number of remote-memory accesses reduces to 764. Most importantly, the remote-memory accesses are scattered all over the memory modules.

The contention graphs of the four algorithms also clearly show effectiveness of the application of the delay functions and, especially the distributed algorithm. The distributed lock algorithm performs extremely well in terms of remote-memory access and network contention reductions.

6.2. *Monitoring a Parallel Nonlinear Optimization Program*

This program solves a large-scale and sparse nonlinear system of equations $F(x) = 0$, where $F \in R^n$, $x \in R^n$ and the Jacobian $J(x)$ is sparse. A triangular decomposition approach is to transform a sparse nonlinear system of equations into a block triangular structure so that the



Memory Access Pattern
Test-and-Set
Static
Exponential
Distributed Lock
Memory Contention in Time
100
Test-and-Set
Memory Contention in Time
100
Test-and-Set with Static Delay
Memory Contention in Time
100
Test-and-Set with Exponential Delay
Memory Contention in Time
100
Distributed Delay

FIG. 1. The memory access patterns and contention graphs of various spin-lock algorithms.

computations can be decomposed for parallel processing. Dennis, Martínez, and Zhang [8, 9] have developed several efficient parallel methods for solving a sparse nonlinear system of equations. We monitored one of the methods which was implemented on the GP1000.

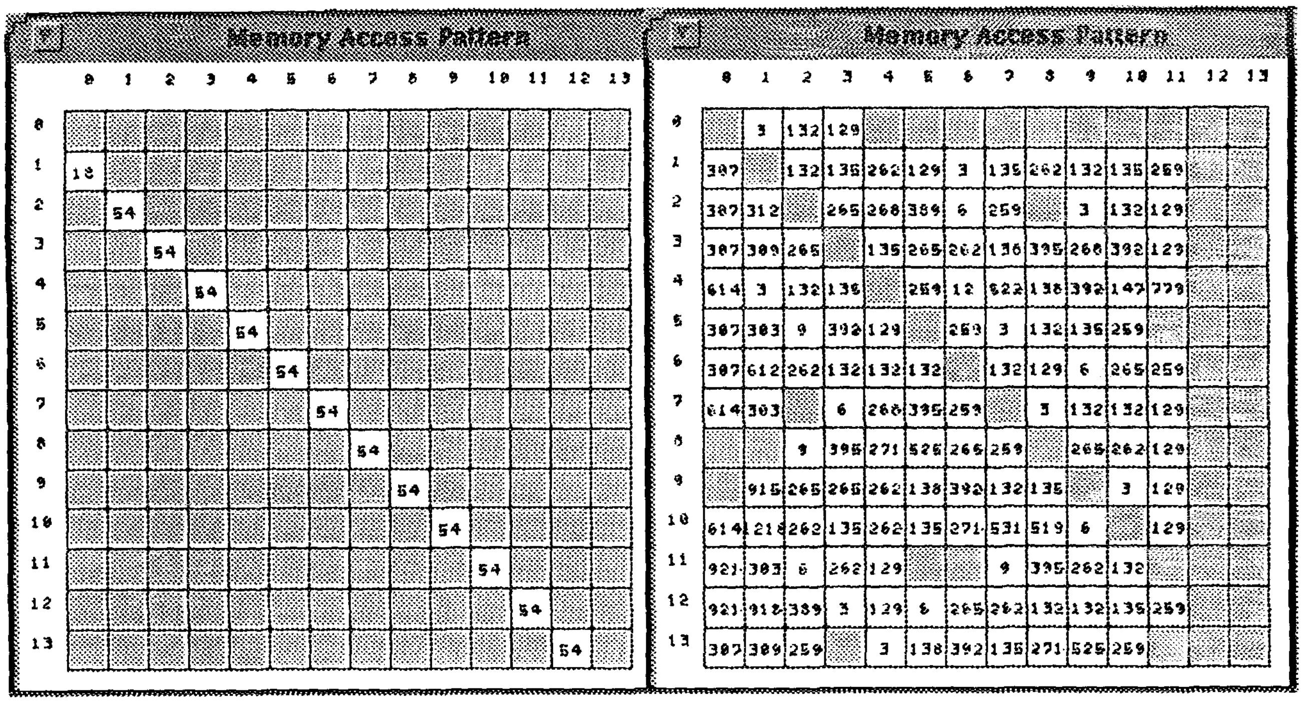
This method is a variation of the Gauss–Seidel–Newton method. It evaluates the diagonal blocks concurrently instead of successively as in the classical Gauss–Seidel–Newton method [15]. The number of variables of the problem we solved on the BBN GP1000 is 1,400. The problem is decomposed into 14 blocks, each of which has 100 variables. Each of the nonlinear subsystems can be evaluated and factored independently. We solve the problem by using the parallel Gauss–Seidel–Newton method.

The shared-memory model with static scheduling is applied for solving this problem, where the block variables are distributed among the 14 processors evenly. In each iteration, the block Jacobian matrices are evaluated and factored in each processor concurrently. Each block *i* is then used to calculate the Newton steps after obtaining the updated block variables from the previous blocks 1, ..., *i* − 1, followed by updating the variables. For simplicity of explanation, we assume that the number of processors is equal to the number of blocks in the system. The updated block variable exchanges take advantage of the block triangular structure so that each processor *i* uses a remote-read to obtain the updated block variables only from its neighbor processor *i* − 1 which contains all the variable blocks of 1, ..., *i* − 1. The processor containing the upper left corner block (block 1) does not conduct any remote-memory access. A barrier is set to sequence the updating process. The major overhead sources of this implementation are the remote-reads and the barrier.

Figure 2a gives the memory access pattern for solving the block triangular nonlinear system in a shared-memory model with static scheduling. In contrast to Fig. 2a, Fig. 2b presents the memory access pattern for solving the same problem in a shared-memory model with dynamic scheduling. The major difference is that many more remote-memory accesses are conducted in a random way. This program also takes much longer to finish, mainly because of more network transactions for remote-memory accesses and scheduling. The monitoring results from MIN-Graph indicate that dynamic scheduling is not necessary but generate more overhead for this computation.

6.3. *Visualizing Parallel Performance of Multiple Molecular Sequence Analysis Algorithms*

With the recent advances in biochemical techniques, DNA and protein sequencing has become routine and is a major tool of molecular biology and genetics research. In molecular sequence analysis, the problem of classifying nucleotide patterns and relationships between multiple sequences are of fundamental importance. Karlin *et al.* [11] described a "linked-list" algorithm to search for statistically significant long matching segments among multiple sequences. The idea of this algorithm has been refined, implemented, and applied to a data set of 386 DNA sequences from the intestinal bacteria *Escherichia coli* genome totaling over 1.43 million nucleotides [12]. This algorithm has been extended in two ways: to search for inverted complementary matches in DNA sequences,



Memory Access Pattern

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0														
1	12													
2		54												
3			54											
4				54										
5					54									
6						54								
7							54							
8								54						
9									54					
10										54				
11											54			
12												54		
13													54	

Memory Access Pattern

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0		3	132	129										
1	307		132	135	262	129	3	135	262	132	135	259		
2	307	312		265	268	389	6	259		3	132	129		
3	307	309	265		135	265	262	136	395	260	392	129		
4	614	3	132	135		259	12	522	138	392	147	779		
5	307	303	9	392	129		269	3	132	135	259			
6	307	612	262	132	132	132		132	129	6	265	259		
7	614	303		6	268	395	259		3	132	132	129		
8			9	395	271	525	266	259		265	262	129		
9		915	265	265	262	138	392	132	135		3	129		
10	614	218	262	135	262	135	271	531	519	6		129		
11	921	303	6	262	129			9	395	262	132			
12	921	912	389	3	129	6	265	262	132	132	135	259		
13	307	309	259		3	138	392	135	271	525	259			

a b

FIG. 2. The memory access patterns for solving a nonlinear triangular system: (a) static scheduling; (b) dynamic scheduling.

and to assemble significant core matches into alignment groups. Multiple sequences are concatenated into a single composite sequence, converting the problem of matching to the search for repeats in a single sequence. There are four main steps in the matching algorithm. First, the composite sequence of N letters is formed and represented as a sequence of integers from the domain of the input alphabet. A *linked list* array which connects

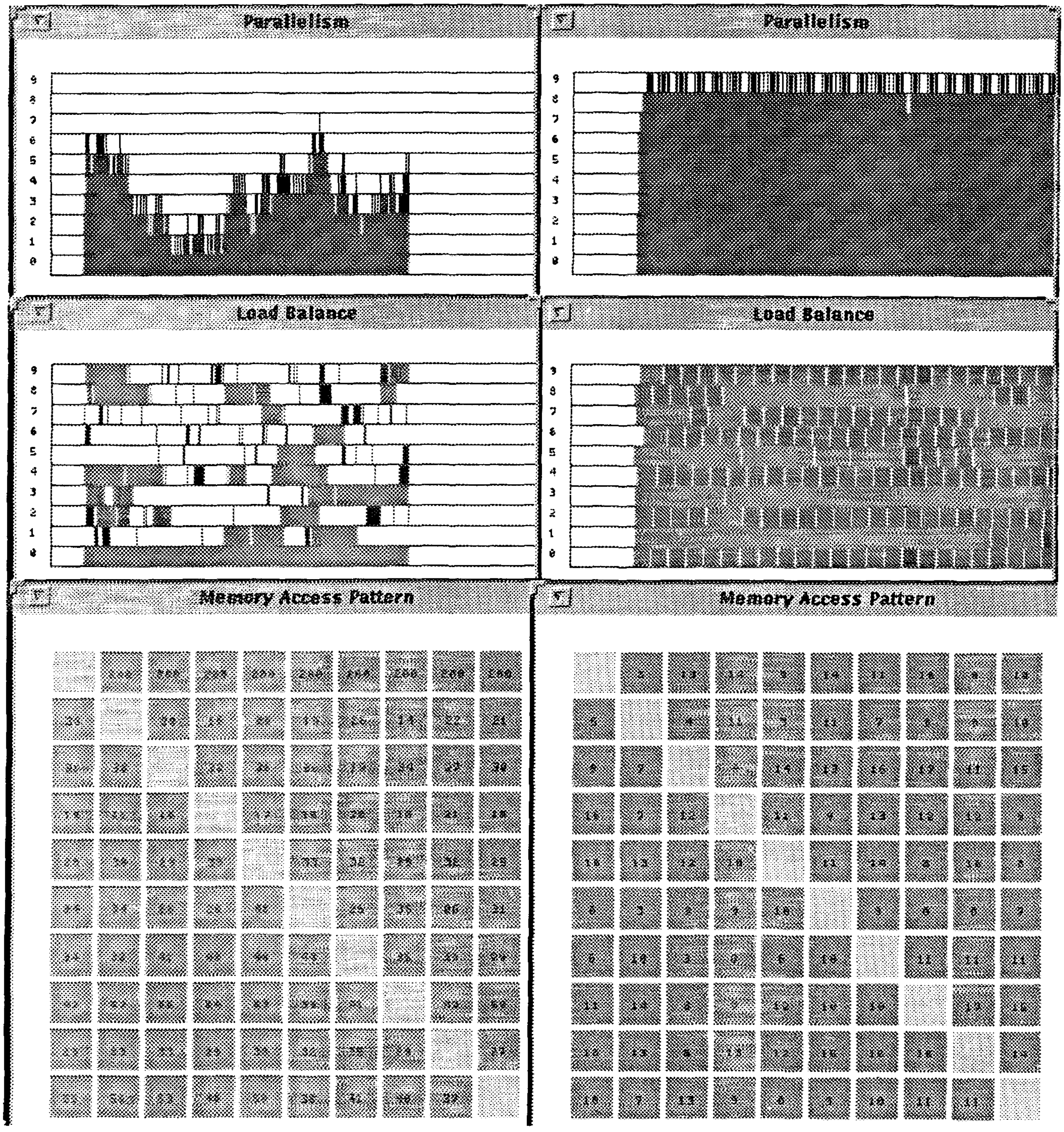


FIG. 3. (a) Performance results of a multiple molecular sequence analysis algorithm; (b) performance results of the modified program of the multiple molecular sequence analysis algorithm.

each occurrence of a k-word (a string of k consecutive letters) to the next occurrence of the same word in the composite sequence is constructed. Second, all reasonably long identity blocks, called core blocks, are located. Third, these core blocks are extended by flanking them with other identity blocks in both directions to form longer matches, allowing interruption by small error blocks. Fourth, maximally extended matches which meet the prescribed printing criteria will be produced in the output file.

This algorithm has been implemented by Srinivasan [18] on the BBN TC2000. The three phases of the matching algorithm—constructing the pointer array, finding the core blocks and extending core blocks with errors—have been developed in parallel forms. Figure 3 shows the final snapshots of workload windows and concurrency graph windows of two implementations monitored by the MIN-Graph. Both implementations apply the shared-memory model with dynamic scheduling. The first implementation (Fig. 3a) is more natural and easily adapted from the sequential algorithm. But the sizes of scheduled tasks are relatively small. The performance results from MIN-Graph shows that this implementation is not efficient: the concurrency graph level is not high and many processor idle gaps are presented in the workload window. This is because each processor spends more time waiting for tasks to be scheduled than computing the tasks.

Based on the advice from the MIN-Graph, the program is modified so that the sizes of the scheduled tasks are increased. Figure 3b reports the performance results, which show that the concurrency graph level is much higher, and processor idle time units are significantly decreased. According to the comparisons of the memory access patterns of the two programs presented by the MIN-Graph, the total number of remote-memory accesses decreases about 81% after the program is modified (from 6990 times to 1341 times). Of course the computing time is shortened as well.

7. SUMMARY

The key for MIN-based multiprocessor performance is to efficiently schedule tasks and distribute data among the processors through the interconnection network. First, this requires that shared data be placed in the processor node where they are expected to be used most (explore the best locality). Second, data need to be effectively distributed in order to reduce remote-memory access and network and memory contention. Third, the processor resources must be effectively used in a process of computation. Efficient and effective parallel processing has become increasingly dependent upon performance analysis and evaluation as the principal tool for explaining the behavior of multiprocessor systems. Our monitor provides performance data related to the above three aspects under distributed-memory and shared-memory programming environments. It gives users a better understanding of the effects from both the program and the interconnection network. Therefore, the effects may be reduced or even eliminated in tuning the parallel program. Current work includes monitoring several large-scale scientific computation programs on the GP1000 and the TC2000. Work continues on monitoring more real-world scientific application programs, and on installing the MIN-Graph in a public domain.

ACKNOWLEDGMENTS

MIN-Graph is a software product of many people's efforts. We are grateful to the National Science Foundation for providing a special supplemental grant in developing the tool. B. Bian made his contributions in the early design stage. B. Wang has implemented a user interface, and has been adding new functions to MIN-Graph. He will be a major contributor to the next version software. We thank W. Chan, P. Srinivasan, and W. Wu for providing their synchronization and scientific computing programs for us to test the software and tune their programs. Many thanks go to Neal Wagner for carefully reading the paper and contributing many valuable suggestions and comments. The anonymous referees provided numerous valuable suggestions for improving the paper.

REFERENCES

1. Anderson, T. E. The performance of spin-lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Systems* **1**, 1 (Jan. 1990), 6–16.
2. BBN Advanced Computer Inc. *Butterfly GP1000 Switch Tutorial* (1989).
3. BBN Advanced Computer Inc. *Inside the GP1000* (1989).
4. BBN Advanced Computer Inc. *Uniform System Approach* (1989).
5. BBN Advanced Computer Inc. *Inside the TC2000* (1989).
6. Bodin, F., *et al.* Performance evaluation and prediction for parallel algorithms on the BBN GP1000. *Proceeding of 1990 International Conference on Supercomputing*. ACM Press, New York (1990), pp. 401–413.
7. Dongarra, J., Brewer, O., Kohl, J. A., and Fineberg, S. A tool to aid in the design, implementation, and understanding of matrix algorithms for parallel processors. *J. Parallel Distrib. Comput.* **9**, 3 (June 1990), 185–202.
8. Dennis, J. E., Jr., Martínez, J. M., and Zhang, X. Parallel block triangular decompositions for solving large sparse nonlinear systems of equations. *Proceedings of the 5th SIAM Conference on Parallel Processing for Scientific Computing*, SIAM Press, Philadelphia (1992), pp. 168–173.
9. Dennis, J. E., Jr., Martínez, J. M., and Zhang, X. Triangular decomposition methods for solving reducible nonlinear systems of equations. *SIAM J. Optimization*, to appear.
10. Geist, G. A., *et al.* PICL, a portable instrumented communication library, C reference manual. Technical Report, ORNL/TM-11130, Oak Ridge National Laboratory (1990).
11. Karlin, S., Morris, M., Ghandour, G., Leung, M. Efficient algorithms for molecular sequence analysis. *Proc. Nat. Acad. Sci. USA* **85**, (1988), 841–845.

12. Leung, M., Blaisdell, B., Burge, C., and Karlin, S. An efficient algorithm for identifying matches with errors in multiple long molecular sequences. *J. Mol. Biol.* **221** (1991), 1367–1378.

13. Malony, A. D., and Reed, D. A. Visualizing parallel computer system performance. *In* Simmons, M., Koskela, R., and Bucher, A. (Eds.), *Instrumentation for Future Parallel Computing Systems*. ACM Press, New York (1990), pp. 59–90.

14. Mellor-Crummey, J. M., and Scott, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Systems* **9**, 1 (1991), 21–65.

15. Ortega, J. M., and Rheinboldt, W. C. *Iterative Solution of Nonlinear Equations in Several Variables*. Academic Press, New York (1970).

16. Qin, X. Modeling, Monitoring and Visualizing Parallel Performance on a MIN-based Multiprocessor. M.S. Thesis, University of Texas at San Antonio, Dec. 1991.

17. Reilly, M. H. *A Performance Monitor for Parallel Programs*. Academic Press, San Diego (1990).

18. Srinivasan, P. *Computational Algorithms for Multiple Molecular Sequence Analysis*. M.S. Thesis, University of Texas at San Antonio, Dec. 1991.

19. Zhang, X., and He, K., Evaluating synchronization effects to scientific computations on large scale shared-memory multiprocessors. *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, Philadelphia (1993).

20. X. Zhang and X. Qin. Performance Prediction and Evaluation for a NUMA Multiprocessor. *IEEE Trans. Software Engng.* **17**, 10 (1991), 1059–1068.

21. Zhang, X., and Wu, W. On measuring and evaluating synchronization and virtual memory performance of a MIN-based multiprocessor. *Proceedings of the 25th Hawaii International Conference of Systems Sciences*, Vol. 1. IEEE Comput. Soc. Press, New York (1992), pp. 218–224.

Accepted January 22, 1993

XIAODONG ZHANG received his B.S. in electrical engineering from Beijing Polytechnic University, China, 1982, and his M.S. and Ph.D. in computer science from the University of Colorado at Boulder, 1985 and 1989, respectively. From 1989 to 1992 he was an assistant professor of computer science at the University of Texas at San Antonio. He is currently an associate professor and chair of the Computer Science Program at the same university. He was a visiting scholar in the Center for Research on Parallel Computation, Rice University from 1990 to 1991. He is also a visiting member of the graduate faculty in the Computer Science Department at the Texas A&M University. 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 is a member of the ACM, the IEEE Computer Society, and the SIAM.

NAGA S. NALLURI received his B.S. in computer science from the B.M.S. College of Engineering at the Bangalore University, India, 1988, and his M.S. in computer science from the University of Texas at San Antonio in 1992. Currently he is a computer systems analyst in American Airline Decision Technology. His research interest is in the area of graphics and scientific visualization.

XIAOHAN QIN received her B.S. and M.S. in computer science from Fudan University, China, 1985 and 1988, respectively. She received a second M.S. degree in Computer Science from the University of Texas at San Antonio in 1991. She was a research associate at the same university in 1992. Currently she is working toward her Ph.D. in computer science at the University of Washington, Seattle. Her research interest is in the areas of parallel programming environment, parallel and distributed system and architecture design, and performance evaluation.