Comparative evaluation and case studies of shared-memory and data-parallel execution patterns [1]

Xiaodong Zhang [a,*] and Lin Sun [b]

[a] *Department of Computer Science, College of William and Mary, Williamsburg, VA 23187-8795, USA*
[b] *USWest Advanced Technology, Denver, CO 80503, USA*

Shared-memory and data-parallel programming models are two important paradigms for scientific applications. Both models provide high-level program abstractions, and simple and uniform views of network structures. The common features of the two models significantly simplify program coding and debugging for scientific applications. However, the underlining execution and overhead patterns are significantly different between the two models due to their programming constraints, and due to different and complex structures of interconnection networks and systems which support the two models. We performed this experimental study to present implications and comparisons of execution patterns on two commercial architectures. We implemented a standard electromagnetic simulation program (EM) and a linear system solver using the shared-memory model on the KSR-1 and the data-parallel model on the CM-5. Our objectives are to examine the execution pattern changes required for an implementation transformation between the two models; to study memory access patterns; to address scalability issues; and to investigate relative costs and advantages/disadvantages of using the two models for scientific computations. Our results indicate that the EM program tends to become computation-intensive in the KSR-1 shared-memory system, and memory-demanding in the CM-5 data-parallel system when the systems and the problems are scaled. The EM program, a highly data-parallel program performed extremely well, and the linear system solver, a highly control-structured program suffered significantly in the data-parallel model on the CM-5. Our study provides further evidence that matching execution patterns of algorithms to parallel architectures would achieve better performance.

Scientific Programming 7 (1999) 1–19
ISSN 1058-9244 / $8.00

1. Introduction

There are three major programming models for parallel computations: message-passing, shared-memory and data-parallel. In the message-passing model, each processor has its own local memory. Processors communicate through an interconnection network consisting of direct communication links connecting certain pairs of processors. The shared-memory model uses a global shared memory that can be accessed by all the processors through an interconnection network. This global memory can either be a physical memory bank, or a single address memory space connected by a set of distributed memory modules. Communications among the processors are accomplished through reading from and writing to the global shared memory. Process scheduling is performed by multiple threads of control. The data-parallel model supports simultaneous operations across large sets of data. This model provides a view of race-free, single thread of control and deterministic execution. It is well known that the message-passing model provides programming support at interconnection network levels, which is typically faster but difficult for an applications user to write programs. Both the shared-memory and data-parallel models provide high-level program abstractions, and simple and uniform views of network structures. The common features of the two models significantly simplify program coding and debugging for scientific applications. However, the easy programming view for each model is ob-

[1] This work is supported in part by the National Science Foundation under grants CCR-9102854 and CCR-9400719, by the U.S. Air Force under research agreement FD-204092-64157, by Air Force Office of Scientific Research under grant AFOSR-95-01-0215, and by a grant from Cray Research. Part of the experiments were conducted on the CM-5 machines in Los Alamos National Laboratory and in the National Center for Supercomputing Applications at the University of Illinois, and on the KSR-1 machines at Cornell University and at the University of Washington.

* Corresponding author. E-mail: zhang@cs.wm.edu.

tained at the cost of building a shared-memory/data-parallel system layer between a user and the interconnection networks. Performance comparisons between the data-parallel and the message-passing models have been conducted [6]. Comparisons between the shared-memory and the message-passing models have also been reported in a number of studies (see [7, 8], and [12]). An architectural comparison of data-parallelism, as implemented in the CM-2 and of vector processing, as implemented in the Cray Y-MP/8 is presented in [13].

All of the above cited programming model comparison studies, except [12] and [13], are supported by intensive simulations. It is necessary to use simulations to isolate implicit and random system effects for fair comparisons among the programming models. Instead of using simulations, our comparison studies are performed on real architectures. There are three main reasons for this. First, we do not intend to compare the shared-memory and the data-parallel programming models, but to compare their execution patterns. Therefore, using simulations along with some system assumptions would prevent us from observing implicit and random system execution effects, which are important performance issues and can be only captured on real architectures. Second, a multiprocessor architecture design generally targets support for a particular programming model, e.g., the KSR-1 for the shared-memory model, and the CM-5 for the data-parallel model. Performance evaluation of the computations using each model and its supported architecture would provide more comprehensive comparisons and implications of the study. Finally, for practical purposes, some complex architecture/model dependent phenomena could be understood only on the basis empirical observations of program executions.

Although both models share some common features for easy programming, the underlining execution and overhead patterns are significantly different between the two models due to their programming constraints, and due to different and complex structures of interconnection networks and system layers to support the two models. We implemented a standard electromagnetic simulation program (EM) and a linear system solver using the shared-memory model on the KSR-1 and the data-parallel model on the CM-5. Our objectives are to examine the execution pattern changes required for an implementation transformation between the two models; to study memory access patterns; to address scalability issues; and to investigate relative costs and advantages/disadvantages of using the two models for scientific computations. Finally our goals are to provide application designers with a better understanding of the two programming models and the architectural supports upon which their algorithms will have to run, and to provide architecture designers with a better understanding of the kinds of applications their machines will have to compute efficiently. Our results indicate that the EM program tends to become computation-intensive in the KSR-1 shared-memory system, and memory-demanding in the CM-5 data-parallel system when the systems and the problems are scaled. The EM program, a highly data-parallel program performed extremely well, and the linear system solver, a highly control-structured program suffered significantly in the data-parallel model on the CM-5. Our study provides further evidence that matching execution patterns of algorithms to parallel architectures would achieve better performance.

The organization of this paper is as follows. Section 2 discusses important issues that can result in performance differences between the two models. In Section 3, we introduce the electromagnetic simulation program and the linear system solver, and outline the architectures of the KSR-1 for the shared-memory model and CM-5 for the data-parallel model. The electromagnetic simulation program is data-parallel structured while the linear system solver is control-structured. Section 4 reports execution tracing and measurement results of the simulation program on both machines with detailed performance comparisons and evaluations. Section 5 studies execution patterns of the linear system solver on the two machines. We also address the scalability issues of the two models by using an experimental metric. Finally, Section 6 presents summaries and conclusions of this study.

2. Performance issues for shared-memory and data-parallel models

From a user point of view, both shared-memory and data-parallel models provide high-level program abstractions, and simple and uniform views of network structures. However, the two models exhibit the following different features in terms of programming views and constructions. These are important issues that can result in performance differences between the two models.

– *Multiple-thread control versus single-thread control*. The shared-memory model performs typical

MIMD operations, where control sequences are formed by dynamic and multiple threads. Synchronization of the threads in the shared-memory model must be explicitly expressed in programs. In the data-parallel model, a user is presented with the logical view that all processors execute a data-parallel program synchronously using a single program counter. A synchronization operation is performed automatically by the system at the end of each program instruction or each program block for single thread control. High Performance Fortran [4] builds on the single-thread data-parallel structures in Fortran 90 to support computationally intensive applications across a wide variety of high performance architectures.

– *Implicit communication versus explicit communication*. Communication in the shared-memory model is implicitly expressed by read/write operations in a non-uniform memory access (NUMA) form, while a user has control over data allocations by explicitly expressing the communication operations in a data-parallel program.
– *Complex data migration versus simple memory access patterns*. In the shared-memory model, the choice of interconnection networks to link processor nodes to cache/memory modules can make NUMA times vary drastically, depending upon the particular access patterns involved. With respect to the kinds of memory organizations utilized, shared-memory systems can be classified into the following three types in terms of data migration and coherence: non-cache-coherence NUMA (Non-CC-NUMA), cache-coherence NUMA (CC-NUMA) and cache only (COMA) architectures [16]. Cache/memory coherence protocols make data migration behavior dynamic and complex. In contrast, the data-parallel model provides simple memory access patterns, where a single sequence of instructions causes operations to be performed concurrently either on the full data sets or on selected sections. When a computation involves data items on different processors, interprocessor communication occurs. If the data layout generated by system software does not well match a program's communication pattern then performance will suffer.
– *Dynamic scheduling versus static scheduling*. In the shared-memory model, processors must be scheduled to individually fetch the program from a central memory, while in the data-parallel model, a control processor broadcasts the program to the processing nodes over a control network, and then the processors execute the program locally.
– *Comparative applications*. The shared-memory model favors parallelism from multiple thread control and scheduling which can asynchronously perform independent tasks among processors, while the data-parallel model favors parallelism from processing a large set of data using the same set of operations. In addition, the shared-memory model can also effectively support SPMD programs.

In summary, network overhead latency, that is, the delay caused by communication between processors and memory modules over the network in both the shared-memory model and the data-parallel model, is a major source of degraded parallel computing performance. However, execution behavior and system effects between the two models are significantly different because latency patterns are constructed differently in each model.

3. The application programs and two architectures

3.1. The electromagnetic simulation program

The electromagnetic (EM) scattering problem is an important application in microwave equipment, radar, antenna, aviation and electromagnetic compatibility design. This program simulates EM scattering from a conducting plane body. In the simulation model, a plane wave from a free space defined as region "A" in the application, is incident to the conducting plane. The conducting plane contains two slots which are connected to a microwave network behind the plane. Connected by the microwave network, the electromagnetic fields in the two slots interact with each other, creating two equivalent magnetic current sources in the slots so that a new scattered EM field is formed above the slots. In general the two slot will interact independent of the relationship outside region "A".

The well-known moment method [3] is used for the numerical model and for the simulation. First, the loaded slots are imitated. Second, an equivalent admittance matrix of region "A" is calculated using a pulse basis mode function expansion. Then intermediate results of excitation vector and coefficient vectors are obtained based on the first two computations. At this stage the resulting EM scattering field is thus simulated by computing a large linear system formed by a so called EM strength matrix. There are four parame-

ters representing characteristics of the equivalent magnetic current. These parameters are used as the numbers of mode function expansions (M and N), and the number of pulse functions (I and J) in the moment method. Another parameter used for final visualization purposes is the number of grid points for discretizing the EM scattering field. The parameters have direct effects on the computational requirements and simulation resolution used by the moment method.

This program carries out millions of iterations in calculating the admittance matrix of region "A",

$$\begin{bmatrix} [Y_{11}^a]_{I\times I} & [Y_{12}^a]_{I\times J} \\ [Y_{21}^a]_{J\times I} & [Y_{22}^a]_{J\times J} \end{bmatrix}. \tag{1}$$

Calculation of each element of the Y^a's in the above matrix involves solving Hankel Functions, Green's Functions and complex integrations. Since the computation is a full 3-D calculation, it is highly time consuming. For detailed information about the numerical method, the interested reader may refer to [3]. This application program has run on the CM-2 and the iPSC/860 [11].

3.2. Linear system solver

The first stage of the solver is to generate a linear system of equations

$$Ax = b \tag{2}$$

where A is a nonsingular $n \times n$ dense matrix. The second stage is Gauss elimination, which subtracts multiples of rows of A from other rows in order to reduce (2) to an upper triangular system. The last stage is to solve the linear system by backward substitution. In Section 5, we will study implementations of this linear system solver using the shared-memory model on the KSR-1 and the data-parallel model on the CM-5.

3.3. The KSR-1 system

The KSR-1 system [5], introduced by Kendall Square Research, is a cache coherent 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 of these rings (1088 processors). Each processor has a 32 MByte cache and a 0.5 MB subcache. Each processor node yields 40 MFLOPS peak floating-point rate.

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. A standard KSR-1 ring has 34 message slots, where 32 are designed for the 32 processors and the remaining two slots are used by the directory cells connecting to the next ring level. Each slot can be loaded with a packet, made up of a 16 byte header and a 128 byte subpage (the basic data transfer unit in the KSR-1). A processor in the ring ready to transmit a message waits until an empty slot is available. A single bit in the header of the slot identifies it as empty or full as the slots rotate through a ring interface of the processor.

3.4. The CM-5 machine

The last member of Thinking Machine's Connection Machine family is the CM-5 [15], a distributed memory MIMD multicomputer with up to 16K processing nodes. In its current implementation, a CM-5 node consists of a SPARC processor operating at either 32 MHz or 40 MHz, 32-Mbytes of memory, and an interface to the control and data interconnection networks. The SPARC processor is augmented with four vector units, each with direct parallel access to the node's main memory. This yields an aggregate memory bandwidth of 256 MB/sec per processing node with an observed 200 MB/sec bandwidth, and a 128 MFLOPS peak floating-point rate per processing node. In comparison with the KSR-1, the floating point operations on the CM-5 could be more than 3 times faster.

The parallel vector units on the CM-5 essentially make it a hybrid between vector and parallel architectures. Parallel variables are distributed between physical vector units and vector positions on a single vector unit. The memory layout of parallel variables onto the memory banks associated with the vector units is normally handled by system software, but may be overridden by the user. The CM-5 also provides a logically shared address space supported by physically distributed memory.

All communication between physical CM-5 nodes is via packet-switched message passing on either the data or control interconnection networks. The data network uses a fat-tree topology designed for high-bandwidth data traffic. The control network is a binary-tree topology designed for low-latency communication of shorter messages. Rapid global broadcast, synchronization and data reduction operations are performed by hardware in the control network.

3.5. Can we obtain fair and comparable performance results?

As we briefly discussed in the previous sections, the KSR-1 and the CM-5 are two different architectures. The number of processors available on the CM-5 is much higher than that on the KSR-1. Besides using different interconnection networks, each node on the CM-5 uses the higher clock rate Sparc processor. Obviously, the CM-5 is significantly more powerful than the KSR-1 in terms of total available computation cycles and memory bandwidth. Here the question is: can we still obtain fair and comparable performance results from the two machines for this study? We address the question from three aspects.

First, both systems are designed to be scalable up to more than one thousand processors through hierarchical interconnection networks. Performance comparisons based on scalability should be fair, because the scalability measurement is concerned with the relative overhead increment from both the program and the architecture and not from absolute cycle time, which is suitable for comparisons among different architectures.

Second, the KSR-1 and the CM-5 are two commercial shared-memory and data-parallel architectures. We do not simply compare the two architectures using the execution times but focus on comparing the two programming models and their execution patterns on the two machines. This study is relatively independent of the size of the system.

Finally, in comparing the performance of the two programming models on their supported architectures using the same program, it is important that the level of optimization be comparable on both machine being considered [1]. We optimized the program code in each programming model version by exploiting architecture features. The optimized program structure and its major source code will be studied in detail in the paper. Of course, we mainly relied on the compiler in the implementations, and did not make an additional effort to apply some "special tricks" to possibly make the code run more intelligently. The cache line size of the KSR-1 is 128 bytes. The large cache line has prefetching advantages as well as false-sharing disadvantage. It also means that although the KSR-1 has relatively high memory access latency, the communication bandwidth in cycles per bytes is large on the KSR-1. False sharing happens when more than one variable is located in one cache line, causing the cache line to be exchanged between processors even though the processors are accessing different variables. The false-sharing is not a concern for the two programs we evaluated on the KSR-1. Because the numbers of accesses to shared variables are limited in both programs.

4. Performance comparisons of the EM simulation program

We present performance comparisons of the EM simulation, with a data-parallel structured program using the shared-memory model on the KSR-1 and using the data-parallel model on the CM-5. The sequential version of the EM program spends about 90% of the total execution time on loops to calculate each element of the admittance matrix. This sequential program executed about 14 hours on a Sun SPARC 10 workstation.

4.1. Programming structures for each model

On the KSR-1 the calculation of each element of the admittance matrix in (1) of the simulation is parallelized. The computation is partitioned into a set of tasks which are distributed in the form of multiple threads among the processors in the system. Major numerical functions, such as Hankel and Green's functions, are evaluated concurrently in each processor. The results are then integrated to calculate the value of each element in the matrix. The concurrent calculations among different processors are coordinated by a synchronization barrier. The integration and update of globally shared variables are implemented using the KSR-1 atomic mutual exclusion primitives. Fig. 1 (left) presents the shared-memory programming model on the KSR-1. The data sets are stored in a single address space, each of which is associated with a processor node and physically distributed in the KSR-1 ring network. The globally shared data are moved among processor nodes upon access requests. Mutual exclusion operations are performed in a critical section. Synchronization of loop iterations is guaranteed by a barrier. For a given EM problem of $10 \times 10 \times 512 \times 512$, more than 100 million iterations are used in the computation. These iterations are distributed among the processors in the form of inner iterations in each node and outer iterations for the whole system. The entire execution control of data migration, cache coherence and synchronization is managed by a system scheduler. It is well known that variations of the program implementation on the KSR-1 could produce significant variations in performance results. Therefore, we opti-

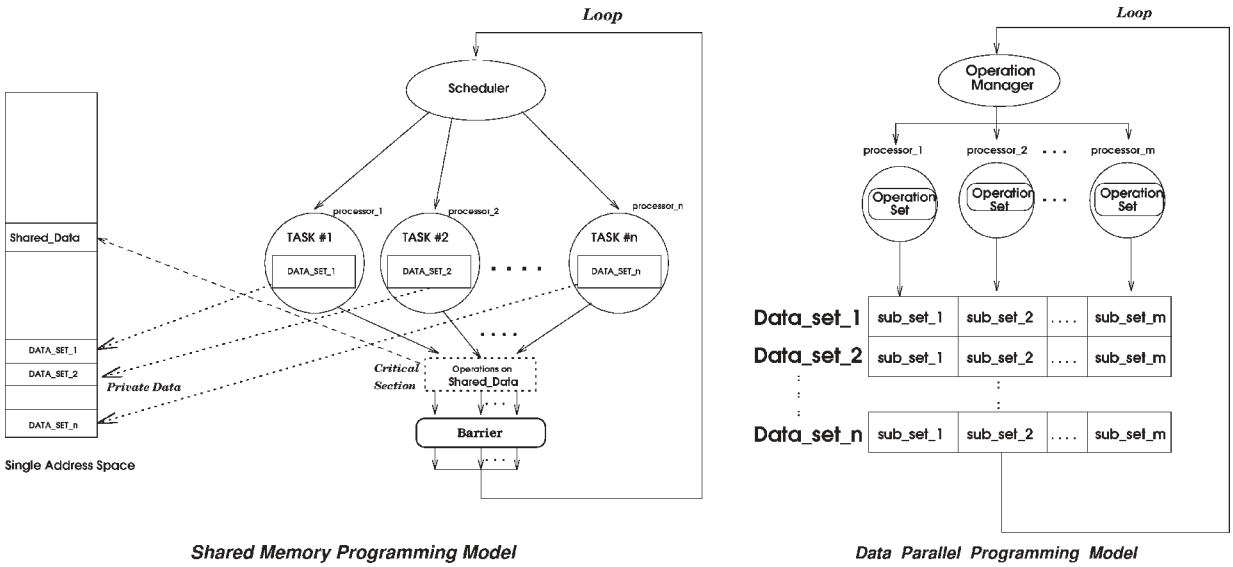

Fig. 1. The shared-memory model on the KSR-1 (left) and the data-parallel model on the CM-5 (right).

mized the program to eliminate all unnecessary data-dependencies, and aligned the global array to make global accesses to those elements with a minimum mutual exclusion requirement.

On the CM-5, the data sets are allocated to a large logic data bank in the system, and these data sets are physically distributed to each processor node. From a user point of view, each partitioned data set (array) is processed by a virtual processor. Each virtual processor performs a set of operations simultaneously on its assigned section of the data set. Thus, repeated operations in different iterations in the sequential program are now executed concurrently on the program's assigned elements by virtual processors on the CM-5. Communications among the virtual processors during execution are conducted either locally within a physical processor or between processors through the data network. Reduction operations for integrations of the simulation are performed through the control network. The data-parallel process control is handled by an operation manager which is called the "partition manager" in the CM-5. Fig. 1 (right) presents the data-parallel model for the simulation program on the CM-5.

4.2. Overall performance on the KSR-1 and the CM-5

The KSR-1 version of the simulation program is implemented with more than 1,000 lines of Fortran code, and the CM-5 version with less than 1,000 lines of CM Fortran. Here we report the overall performance of the program running on the KSR-1 and on the CM-5 with and without vector processing. Comparisons between the two models and computations based on their overhead patterns and scalabilities will be discussed in the following two sub-sections. Table 1 presents execution times from three versions of the program. The results indicate that the data-parallel implementation on the CM-5 is much more effective than the shared-memory implementation on the KSR-1 for computing this simulation. For example, using 64 processors, the execution took about 18 minutes on the KSR-1, but one minute on the CM-5 without using the vector unit, and only about 17 seconds with the vector unit. Using 64 processors, the CM-5 still showed a big potential to scale to a large number of processors for this fixed-size problem, but the KSR-1 seemed to have reached the minimum execution time. Regarding speedups, execution on the CM-5 without using the vector unit achieved the highest speedup, while execution on the KSR-1 had the lowest speedup among the three. The vector unit in each CM-5 node makes computation faster, and generates additional overheads as well. Here we briefly address the structure, power and overhead sources of using the vector unit on the CM-5.

CM Fortran programs can be classified into different execution models according to the way a program makes use of the hardware. This simulation program uses the global model, where a single program operates on arrays of data spread across all the processor nodes. The global model can be further divided into the Global SPARC Nodes Model and

Table 1
Execution time measurements of the simulation on the KSR-1 and the CM-5

Number of processors	KSR-1 (second)	CM-5 (second)	
		sparc	vu
32	1352.21	132.41	27.368
40	1237.53	N/A	N/A
50	1162.11	N/A	N/A
64	1063.92	61.26	16.8
128	N/A	33.91	10.37
256	N/A	18.40	5.789

the Global Vector-Units (VU) Model. In the Global SPARC Nodes Model, the partition manager serves as the control processor, and the SPARC nodes, are used for processing the multiple data sets. If the system has vector units, they just serve as memory controllers and do not participate in the processing. In the Global VU Model, the partition manager serves as the control processor, and the vector units provide the real power for processing. The SPARC processor nodes are invisible to the program, although they assist the vector units with OS services and communications.

On the CM-5, each vector unit is a memory controller and a computational engine controlled by a system interface. Vector units cannot fetch their own instructions; they merely react to instructions issued to them by the SPARC processor node. Each vector unit includes an adder, a multiplier, memory load/store, indirect register addressing, indirect memory addressing, and others. Every vector-unit instruction can specify at least one arithmetic operation and an independent memory operation. A vector unit can operate on two 64-bit data items together. Therefore, each CM-5 node can operate on eight 64-bit vectors simultaneously. In this EM simulation program, more than 78% of the total execution time is spent evaluating the specified function which involves complex arithmetic computation on a large set of data. This explains the superior execution performance of the vector unit Node Model version of the program.

When using the vector unit, all VU instruction fetching and control decisions are made by the SPARC processor node. The SPARC processor issues a vector instruction using the following procedure: it fetches the instruction itself from its data memory, calculates the special vector-unit destination address for issuing the instruction, and executes the store. An additional vector start-up overhead is required when a new vector operation is performed. The processing latency is a major overhead source of using the vector unit. However, there are two other advantages of using the vector unit besides the high speed. First, the latency of issuing vector instructions is independent of the number of processors. Second, the operations of issuing vector instructions by the SPARC node and vector processing operations in the vector unit can be overlapped after an operation pipeline is formed.

4.3. Comparative execution and overhead patterns

This comparative execution and overhead pattern study is based on the measurements, evaluations and analyses of the memory access characteristics, data communication, data movement, data locality, effects of cache coherence and other related effects from the KSR-1 and the CM-5 on this simulation application.

4.3.1. Computation-intensive versus memory-demanding executions

The simulation program using the shared-memory model is computation-intensive, which can be shown by the memory allocation arrangement. The input parameters of the program are $M \times N \times I \times J$, where I and J are the numbers of the expanding mode functions, and M and N are the numbers of the pulse functions used in the moment method. After partitioning the problem on the KSR-1, the memory requirement in each processor becomes $(M \times N \times I \times J)/p$, where p is the number of processors used. The major computation in each processor is to repeatedly evaluate different complex functions. A large requirement of CPU cycles dominate the numerical computation, while the memory space requirement is much less demanding. For example, for the simulation with problem size of $10 \times 10 \times 512 \times 512$, the memory allocation requirement for each processor node is less than half of one MB if 64 processors are used for the computation.

In contrast, the same computation on the the CM-5 becomes memory-demanding. The program is parallelized by transforming the data independent loops in the sequential program to a single sequence of operations. This transformation is done by duplicating the same variable in a loop q times, where q is the total number of iterations. The duplicated variables may be formed into an array, and are allocated among the nodes to be processed by a same set of operations concurrently. For the simulation with problem size of $10 \times 10 \times 512 \times 512$, the duplications of the variables expands the memory requirement to 3,600 MB. This requires more than 200 CM-5 memory modules to store all the variables, taking into account the additional memory space for the program and operating system in each node.

4.3.2. Comparative network delays and their sources

The major overhead of running a program on the KSR-1 comes from network latency, which is caused by synchronization, cache coherence, remote data accesses and other events involving network activities. All the performance data are collected by a hardware monitor called Pmon, 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. It also quantitatively describes the changes of network delay for an application program as the number of processors increases.

Fig. 2 (left) shows execution time distributions for the simulation program on the KSR-1, starting with 32 processors and ending with 64 processors. The total execution time consists of effective computation time, synchronization overhead, and cache miss latency. The cache miss latency measures accumulated network delay and CPU idle time caused by remote accesses and cache invalidations. For a fixed-size problem running on the KSR-1, the effective computing time decreases, while the overhead portions increase, both proportionally to the increment of the number of processors. For example, the effective computing time was about 40% of the total execution time on 32 processors, and it decreased to 23% with 64 processors. The measured execution and overhead patterns indicate that as the number of processors increases, the execution time eventually hits a minimum, after which adding processors can only cause the program to take a longer time to complete. This performance behavior is normal because the communication overhead of the architecture increases as the number of processors increases if the problem size is fixed.

The major overhead of running a program on the CM-5 also comes from network latency which can be quantitatively measured by accumulating I/O overhead, and latencies in the data network and the control network and between the partition manager and the processing nodes. The measured network latency data are collected by a software monitor called Prism on the CM-5. Prism is a Motif-based graphical programming environment within which users can develop and analyze programs written for the CM-5. Programs may be edited and compiled under Prism, then executed normally or step-by-step. To aid in program development, Prism provides a dbx-like debugger. To analyze program performance, Prism can collect data on execution time broken down by procedures or by lines of source code. For data-parallel programs, this data includes control processor time, vector unit processing time, communication time between the control processor and nodes, and data network communication time. Another useful Prism tool is the data visualizer, which displays the contents of large arrays or parallel variables in a variety of textual and graphical formats. Since Prism is still under construction, we also used the CM timer to measure and verify experimental results.

In order to obtain precise results, all experiments on both of the KSR-1 and the CM-5 were run under benchmark mode, so that the systems were solely used for these measurements.

In contrast to the executions on the KSR-1, Fig. 2 (right) presents completely different overhead patterns of the simulation using the data-parallel model on the CM-5 with/without VU support in each processor node. The simulation program was run up to 256 processors on the CM-5, and achieved good performance. The major execution difference between the KSR-1 and the CM-5 can be explained by the two different overhead patterns. The overhead portions on the CM-5 (the ratio between the overhead time and the total execution time) increase much more slowly than the ones on the KSR-1, which makes this program highly scalable on the CM-5. There are two main reasons for this. First, this memory-demanding program on the CM-5 requires a large memory allocation in each processor. Thus, to increase the number of processors in the computation will reduce the memory requirement burden in each node and reduce I/O latencies for page swapping between memory modules and secondary storage. Second, besides the data network for point to point data communication, the control network provides fast services for the synchronization and reduction operations of the computation. The latencies of both the data network and the control network caused in the computation are independent of the number of processors. If memory contention is reduced as the number of processors increases, the latencies of both may further decrease. Thus, the network bandwidth scales in proportion to the number of processor nodes. Table 2 presents latency measurements of I/O, the data network and the control network for executions of the program from 32 to 256 processors on the CM-5, which quantitatively support the above two reasons. For example, while increasing the number of processors from 32 to 256 on the CM-5 with the VU support, the I/O latencies de-

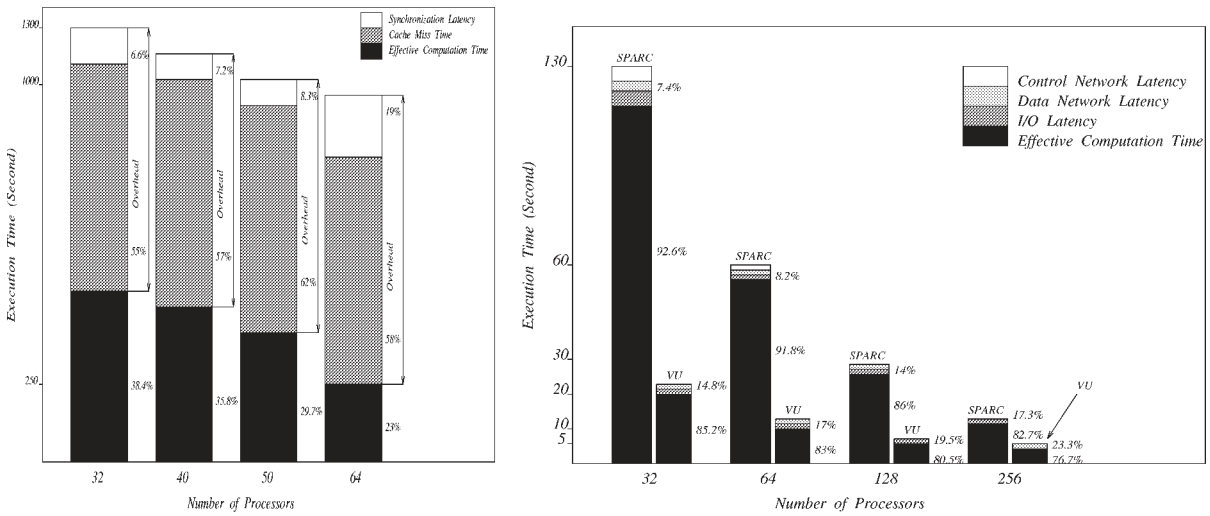

Fig. 2. Execution patterns on the KSR-1 (left) and on the CM-5 (right).

Table 2
Measurements of different types of network latencies on the CM-5

Number of	I/O latency (sec.)		Data Net. latency (sec.)		Control Net. (sec.)	
processors	SPARC	VU	SPARC	VU	SPARC	VU
32	2.54	1.32	2.54	2.14	4.93	0.70
64	1.93	1.00	1.10	0.93	2.46	0.35
128	1.49	0.98	0.46	0.50	1.25	0.19
256	1.25	0.57	0.24	0.29	0.64	0.11

creased from 1.32 to 0.57 seconds; the data network latencies decreased from 2.14 to 0.29 seconds; and the control network latencies decreased from 0.70 to 0.11 seconds.

4.4. Comparative program-architecture scalabilities

Scalability measures the ability of a parallel machine to improve performance as there are increases in the size of the application problem and in the number of processors involved. Overhead comprehensively measures all the lost cycles and bandwidths during a parallel execution, such as processor idle times, and the delay caused by communication between processors and memory modules over the network in a parallel system. As a major source of degraded parallel computing performance, the overhead forms a major obstacle to improve parallel computing performance and scalability. The metric proposed in [17] uses the overhead as a major factor to evaluate parallel computing scalability. We use the this metric to compare the scalabilities of programs on the KSR-1 and on the CM-5.

4.4.1. Latency sources and their measurements

There are three major latency sources forming overhead patterns inherent in algorithms and interconnection networks, namely, the memory reference latency, denoted as *ML*, the processor idle time, denoted as *IT*, and the parallel primitive execution overhead time, denoted as *PT*. The average latency in the latency metric is then defined as

$$L(W, N) = \frac{ML + IT + PT}{N}, \tag{3}$$

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

– *Memory reference latency* measures the delays caused by communication between processors and memory modules over the network. In a shared-memory system, this mainly comes from remote read and write accesses and corresponding cache coherence operations; while in a distributed

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

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

Processor idle time occurs in a sequential program due to memory access delays, overhead of page swapping and other I/O activities. Similar activities in parallel program execution will make the processor idle from time to time. The sequential delays should be eliminated. The processor idle time in a parallel program is measured as follows:

$$IT = NT_{\text{para}} - \sum_{i=1}^{N} T_i \tag{4}$$

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

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

$$ML = ML(N) - ML(1) \tag{5}$$

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

4.4.2. The latency metric

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

$$\text{scale}(E, (N, N')) = \frac{L_e(W, N)}{L_e(W', N')}. \tag{6}$$

We also call the metric in (6) an E-conserved scalability because the efficiency is kept constant.

In practice, the value of (6) is less than or equal to 1. A large scalability value of (6) means there are small increments in overheads 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 overheads and therefore a poorly scalable system.

The scalability metric of (6) will generate an upper triangular table, where each value represents the scalability value between any meaningful pair of processors. We compared the scalability between selected pairs of processors on the KSR-1 and the CM-5. This metric is concerned with the relative latency increment and not absolute cycle time, and thus can be used for scalability comparisons among different architectures.

Before measuring and evaluating the overhead, we need to experimentally determine the sizes of the problem (W and W'), for given system sizes (N and N') and for a given efficiency constant E. After that, the E-conserved overheads $L(W, N)$ and $L(W', N')$ can either be calculated or measured to determine the scalability. Fig. 3 gives the basic testing process to determine the problem size for a given efficiency running on a given number of processors.

4.4.3. Scalability evaluation

Keeping the efficiency at 80% for all the experiments by adjusting the problem sizes as the number of processors increases, we obtained average scalability measurements from 32 to 64 processors on the KSR-1 and the CM-5, and the measurements from 32 to 256 processors on the CM-5. Table 3 lists all the scalability values for the program running on both machines. The scalability results indicate that programs using the data-parallel model running on the CM-5 are more than 4 times more scalable than the program us-

Table 3
Scalability measurements of the EM simulation programs on the KSR-1 and the CM-5

	32–64 nodes			32–256 nodes	
	KSR-1	CM-5 SPARC	CM-5-VU	CM-5 SPARC	CM-5 VU
scale	0.23	0.89	0.83	0.78	0.70

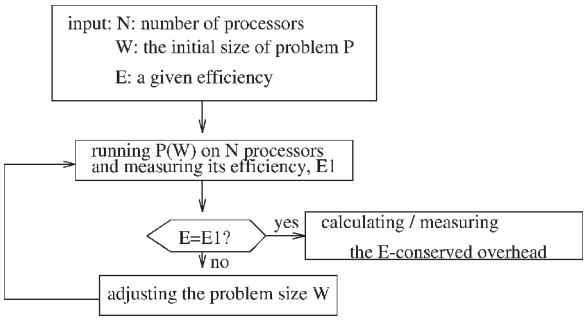

Fig. 3. The testing process to determine the problem size for a given efficiency constant running a given number of processors.

ing the shared-memory model on the KSR-1. This result is based on measuring all three programs from 32 to 64 processors on both machines. The scalability results obtained from 32–256 nodes show the high ability of the program to scale to a large number of processors on the CM-5.

5. Performance comparisons of the linear system solver

Our intensive measurements for execution patterns and scalability evaluation in the previous section have shown that the EM simulation program is an excellent candidate for data-parallel computing on the CM-5. The performance of the same program using the shared-memory model on the KSR-1 is also acceptable, considering the available resources. The last issue we want to address is: under what conditions will performance of a program using the data-parallel model be significantly degraded? For this reason, we implemented a parallel linear system solver using the shared-memory model on the KSR-1 and using the data-parallel model on the CM-5. This program includes three parts of the computation in sequence: forming a solvable linear system, transforming the coefficient matrix A into an upper triangular matrix using Gauss elimination, and solving the upper triangular system using backward substitution. The first part (forming the system) performs straightforward parallel operations for both the shared-memory and data-parallel models. Both Gauss elimination and backward substitution are fine-grained and control-structured algorithms with multiple synchronization points. The interested reader may refer to [14] for detailed information on parallel Gauss Elimination, and may refer to [10] for parallel backward substitution algorithms.

Fig. 5 presents the shared-memory implementation of Gauss elimination and backward substitution on the KSR-1. Matrix A of $n \times n$ order and vector b are initially divided into blocks of rows (m rows per processor) and physically distributed in each processor's memory which are globally shared. In Gauss elimination, processor 0 is dedicated to pivoting through remote accesses (lines 3 and 4). During this period of time, the remaining processors wait at the barrier (line 6). As soon as the pivoting operation is done for a column, multiple threads are used to perform subtractions of row elements simultaneously where remote accesses occur again (lines 8 and 10). For a CC-NUMA shared-memory system, the Gauss Elimination program would not involve any invalidations because no multiple data copies are generated. However, the KSR-1 is a COMA system. Duplicated copies are generated when remote reads are performed. In this program, when two rows are switched for pivoting, the two rows are duplicated in both processors. When the elimination starts, the write updates will cause invalidations on duplicated copies. False sharing also likely occurs during the invalidations. However, our tracing results (see Fig. 7 (left)) indicate that the number of invalidations for this program was not significant. Therefore, this program is still reasonable for this evaluation study.

Backward substitution is even more control-structured than Gauss elimination. While a processor calculates the solution for a diagonal variable in back order locally (lines 18 and 19), the remaining processors wait at the barrier (line 21). As soon as a solution is available, all the processors can perform their updating operations by using the variable's value simultaneously, where remote accesses are involved (line 23). Thus, the major factors to degrade performance in Gauss elimination and backward substitution are synchronization barriers and remote accesses. There are no shared variables in the program to cause cache invalidations after a write operation is performed. In both algorithms, par-

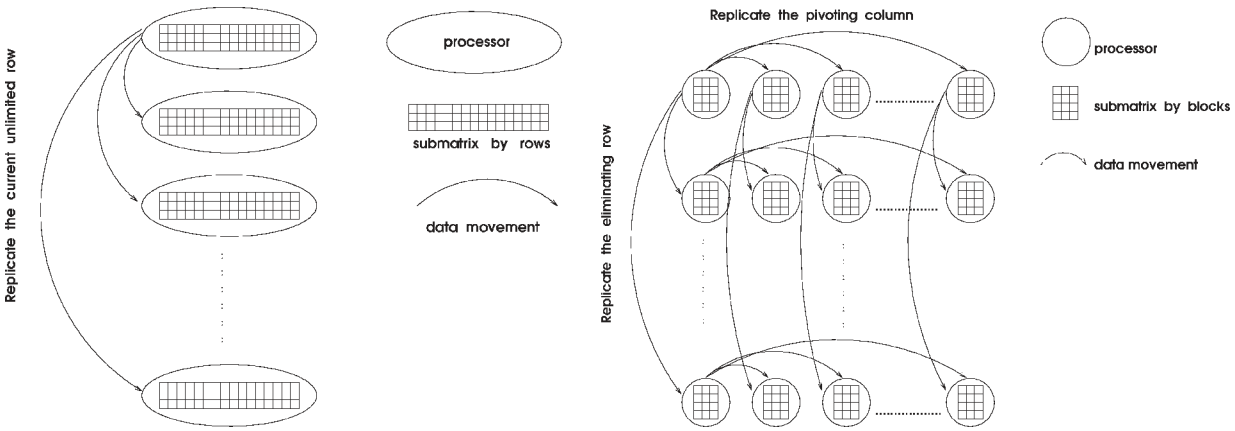

Fig. 4. Comparative memory layouts for the linear system solver using the shared-memory model on the KSR-1 (left) and using the data-parallel model on the CM-5 (right).

```
1. for k=1 to n do
2.     if id=0 then
3.       find pivot in the k-th column of A
4.       exchange pivot row and the k-th row so that A[k,k]=pivot
5.     endif
6.     BARRIER
7.     for j=k+1+id to n step p do
8.         c=A[j,k]/A[k,k]
9.         for i=k+1 to n+1 do
10.            s=A[k,i]
11.           A[j,i]=A[j,i]-s*c
12.        end i loop.
13.    end j loop
14.    B[k]=A[k,n+1]
15. end k loop
16. BARRIER
17. for m=n to 1 do
18.     if id=(m mod p) then
19.       B[m]=B[m]/A[m,n]
20.     endif
21.     BARRIER
22.     for q=1+id to m stedp p do
23.         B[q]=B[q]-A[q,m]*B[m]
24.     end q loop
25. end m loop
```

n: order of the matrix
p: number of processors
id: individual processor index
s: a local variable
c: a local array
A: the matrix stored in the shared memory
B: a vector stored in the shared memory

Fig. 5. Gauss elimination and backward substitution using the shared-memory model on the KSR-1.

allel operations decrease proportionally as the number of processed columns increase.

We implemented the linear solver using the data-parallel model on the CM-5. We compiled it using the CM Fortran compiler version 2.1.1-2 with code optimization option. We did not use any mathematical library functions in order to have a fair comparison. In this program, the parallel data layout was done manually to minimize interprocessor communications. The user's view of a data allocation, such as a vector or a two dimensional array, is called a "shape". Since the number of array elements is much larger than the number of node processors, array elements are grouped and distributed in each node. Each processor is further divided into the same number of virtual processors that will operate on each element. If a set of arrays in a

```
1. for k=1 to n do
2.     find pivot in i-th row , k-th column of A
3.     exchange the i-th row and k-th row of A
4.     replicate the k-th column n+1 times along the column direction to C
5.     C[ k:n, : ]=C[ k:n, : ]/pivot
6.     replicate the k-th row n times along the row direction to S
7.     A[ k:n, : ]=A[ k:n, : ]-C[ k:n, : ]*S[ k:n, : ]
8. end k loop
9. B=A[ : , n+1]
10. for m=n to 2
11.     B[ m ]=B[ m ]/A[ m, m ]
12.     replicate B[ m ] n times along the column direction to D
13.     B[ 1:m-1 ]=B[ 1:m-1 ]-A[ 1:m-1, m ]*D[ 1:m-1 ]
14. end m loop
```

n: order of the matrix

A: the matrix distributed among processors

C: a 2-D array that has the same shape as A

S: a 2-D array that has the same shape as A

B: the right hand side vector

D: a vector that has the same shape as B

pivot: a scalar variable

Note: A[k:n, :] means all the elements from k-th row to n-th row in A.

Fig. 6. Gauss elimination and backward substitution using the data-parallel model on the CM-5.

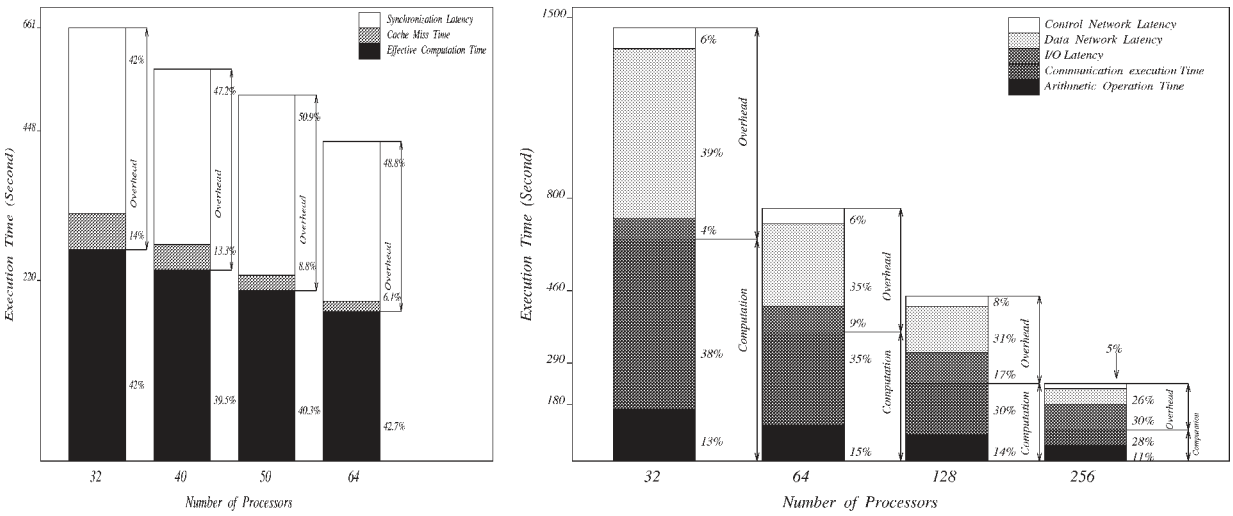

Fig. 7. Execution patterns of Gauss Elimination using the shared-memory model on the KSR-1 (left) and using the data-parallel model on the CM-5 (right).

computation are in the same shape, the operations upon each set of elements will be carried out simultaneously by its corresponding virtual processor. However, computation involving arrays in different shapes will cause heavy irregular interprocessor communications. A solution for this is to replicate the data so that the data sets operated on have the same shape. Replication of an array join calculation with a higher rank (different shape) array can be used to avoid heavy latency caused by irregular data movement. In our CM linear solver, we used the CM Fortran intrinsic function SPREAD which makes use both of the control and data network to carry out the replicating operations in a regular communication manner.

Fig. 6 presents the data-parallel implementation of Gauss elimination and backward substitution on the CM-5. The computing steps in bold fonts are operations involving interprocessor communications. First the control network is used to find the pivot by using its reduction operation (line 2). Our experiment

shows that the operations of finding pivots for all the columns took more than 10% of the communication time and took almost the entire latency time in the control network. The data network is used to exchange corresponding array elements across processors (line 3). Both the data and the control networks are used to replicate the portion of array A to multiple arrays in the same shape of A for more effective data-parallel operations (lines 4 and 6). To do the elimination, virtual processors in each physical processor operate on the corresponding array elements and save the result back to A without involving interprocessor communications (lines 7).

In backward substitution, the last variable is solved first (line 11). Then it is replicated to other processors so that the computation can be done concurrently by virtual processors (lines 12 and 13). The overhead comes from the broadcast communication of the solved variable in each iteration. Since the data size of the variable is small, the network latency is not significant. The latency sources for this linear system solver mainly come from reduction, broadcast operations to distribute pivot element and row, and replications of data. These operations cause significant amounts of activities in the data network and the control network.

The comparative memory layouts for the linear system solver using the shared-memory model on the KSR-1 and using the data-parallel model on the CM-5 are presented in Fig. 4, where the matrix is distributed by rows in the shared-memory model, and is distributed by block submatrices in the data-parallel model. In contrast to the EM simulation program, this linear system solver on the CM-5 has the following comparative features: control-structured versus data-parallel structured, low memory access and I/O demand versus high memory access and I/O demand, and dynamic processing versus static processing. The fine-grained and control-structured features of this program make the execution efficiency of this data-parallel implementation much lower than that of the EM simulation on the CM-5. In terms of execution, the linear system solver on the CM-5 can not take advantage of the powerful control network because only data communications are involved. On the other hand, the linear system solver does not require global variable invalidations in the shared-memory model on the KSR-1, which reduces the network contention and communication traffic.

Fig. 7 (left) presents comparative patterns of solving a linear system with 2,000 variables by using the shared-memory model on the KSR-1, and using the data-parallel model on the CM-5. As we expected, the performance on the KSR-1 is quite normal. Computation efficiency and speedups are acceptable. The latences caused by synchronization and cache misses increase as the number of processors increases.

In contrast, Fig. 7 (right) shows that the performance of the data-parallel program on the CM-5 is significantly degraded. First of all, the total execution time is unacceptably long. For example, using 32 processors, the CM-5 without using the vector units spent over twice amount of time as the 32 processor KSR-1 did. However, the CPU and the network speed of the CM-5 are much faster than the KSR-1. In the previous example of the EM simulation, the 32 processor CM-5 spent less than 1/10 of the time the 32 processor KSR-1 did. In addition, network latencies and overhead contribute about half of the total execution time on the CM-5 for the computation on different numbers of processors. The effective computation time is divided into two parts: arithmetic operations and communication instruction execution overhead. In contrast, the communication execution time is too trivial to be measured in the previous EM simulation program. This portion is quite significant in the linear system solver on the CM-5. The experimental results present an example of how a control structured algorithm implemented by the data-parallel model degrades its performance. Here we need to point out that the same linear system solver can be implemented much more efficiently using the message-passing library on the CM-5 [18].

In summary, the linear system solver would not fully take advantage of the data-parallel model on the CM-5. There are two reasons for this. First, the data replications to form array pairs for regular data operations significantly degrade the performance. Our experiments showed that about $2/3$ of the execution time was spent on operations related to data replication. Second, the number of data operations proportionally decreases in the programs of Gauss elimination and the backsolve. This data reduction nature generates high code block startup time and data movement overhead on the CM-5.

Keeping the efficiency to be 50% for all the experiments by adjusting the problem sizes as the number of processors increases, we obtained average scalability measurements of the linear system solver from 32 to 64 processors on the KSR-1 and the CM-5, and the measurements from 32 to 256 processors on the CM-5. Table 4 lists all the scalability values for the program running on both machines. These scalability results indicate that the programs using the data-parallel

Table 4
Scalability measurements of the linear system solver on the KSR-1 and the CM-5

	32–64 nodes			32–256 nodes	
	KSR-1	CM-5 SPARC	CM-5-VU	CM-5 SPARC	CM-5 VU
scale	0.37	0.76	0.73	0.72	0.70

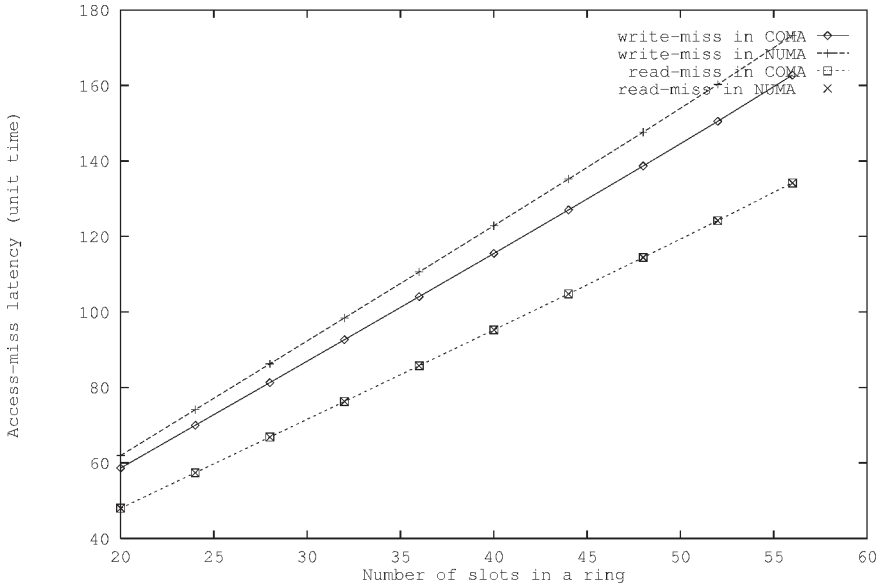

Fig. 8. Effects of changing the ring size to miss latencies.

model running on the CM-5 are still more scalable than the program using the shared-memory model on the KSR-1. This is because the network latencies of the CM-5 increase significantly less than that of the KSR-1 as the number of processors increases.

6. Implications and comparisons of the network architectures

We have shown that network latency forms a major obstacle to improve parallel computing performance and scalability on both the KSR-1 and the CM-5. The scalability results measured by the latency metric indicate that, for the same application program, data-parallel computing on the CM-5 has lower relative latency increments than shared-memory computing on the KSR-1. This is even true for the linear system solver, which is better suited to the shared-memory model. Implications of network latency changes in the two systems are the last part of this study. The purpose is to compare architecture and network support for shared-memory and data-parallel execution. We mainly investigate comparative network latency patterns as the KSR-1 and the CM-5 are scaled.

6.1. Network latency patterns versus KSR-1 ring system scaling

The KSR-1 is a typical MIMD multiprocessor where network latency in computing increases as the number of processors increases. There are two ways to scale the system: by increasing the number of processors by adding more slots in each ring, and by adding more rings in the system. Since both scaling methods increase the average access distance in computing, the network latency increases proportionally. To see this, we show two experimental results.

The first experiment was done on a simulated hierarchical ring to observe and compare the read- and write-miss latences of COMA and NUMA memory systems. The cache coherence protocol and the ring architecture in the simulation are constructed based on the KSR-1 model. Assume that the miss rate in each processor is uniformly distributed, and the rotation period of the ring is in unit time. Fig. 8 shows that, by increasing the number of slots in a ring, read-miss latencies in both COMA and NUMA systems have the same increasing curves, but the write-miss latency in NUMA increases slightly slower than that in the COMA due

to more frequent data movement in the COMA system.

The second experiment measured and compared the costs of maintaining coherent caches on the KSR-1 between one ring and two rings.

For more information about the latency analysis of ring-based multiprocessors, the interested reader may refer to [16].

Our experiments show that the maintenance of cache coherence bears no additional cost in the same ring, over and above the latency of the ring rotation itself in the KSR-1. This is because the ring rotation is clocked so that any and all actions that could possibly take place during a stop at each cell can be accomplished. Each stop of the ring rotation allows for the longest possible action to take place. Therefore, the rotation period must increase as the number of processors increases in the ring, and so does the network latency.

6.2. *Network latency patterns versus CM-5 fat-tree system scaling*

The bandwidth of each communication network channel changes differently as the number of processors changes in multicomputer networks. For example, a hypercube of arbitrary dimension can be made using a linear arrangement with connecting wires. The cube of each dimension is obtained by replicating the one of next lower dimension and then by connecting corresponding nodes. The higher dimensional hypercube is constructed by further connecting the bisections of the hypercube of the current dimension. A channel is a physical link between two directly connected nodes. It is made up of a bundle of wires consisting of wires for data bits and any necessary control bits. The number of channels across the bisection needed to construct a hypercube is $N/2$, where N is number of nodes in the hypercube. Therefore, the bandwidth of each channel proportionally decreases as the number of processors increases. This example shows that the bisection bandwidth is an important limitation in a multicomputer system.

The CM-5 data network is a 4-ary fat-tree. The network is composed of router chips, where each chip has an 8-bit-wide bidirectional link to each of its four child chips lower in the fat-tree, and four 8-bit-wide bidirectional links to its parent chips higher in the fat-tree. To route a message from one processor to another, the message is sent up the tree to the least common ancestor of the two processors, and then down to the destination. This network design provides many comparable paths for a message to take from a source processor to a destination processor. As it goes up the tree, a message may have several choices as to which parent connection to take. Since the CM-5 network construction has a constant bisection bandwidth, it has lower latency and higher throughput than variant network bisection structures such as high dimensional hypercubes.

Besides the bisection bandwidth, another important issue is the network usage efficiency, which directly affects how the network latency changes as the system is scaled. The message injection rate into the CM-5 network depends on the processor speed, which may limit the bandwidth of the processor-network links, but not the network hardware. The data network hardware speed on the CM-5 is 20 MB/s in each direction. The maximum message injection rate is calculated by

$$R_{\text{inj}} = \frac{K}{Q_{\text{send}} S}, \tag{7}$$

where K is the message packet size in bytes, Q_{send} is the communication protocol cost for sending a message in cycles, S is the processor speed in seconds/cycle. Substituting $K = 20$ (20 byte packet with 16 bytes of payload), $Q_{\text{send}} = 37$ (obtained from measurements [2]), and $S = 0.03$ (the clock rate of each Sparc processor node is 33 MHz) into (7), the maximum message injection rate, or the maximum bandwidth in each processor-network link on the CM-5 is 14.3 MB/s. Similarly, the maximum receiving rate is calculated by

$$R_{\text{rev}} = \frac{K}{Q_{\text{rev}} S}, \tag{8}$$

where Q_{rev} is the communication protocol cost for receiving a message in cycles. The receiving cost is higher, at 60 cycles per 16-byte packet, because the receiving operation needs both read and write. Substituting $Q_{\text{rev}} = 60$ and others into (8), the maximum receiving rate is limited to 8.8 MB/s.

There are three different data transmission rates involved in CM-5 data network communication: the injection rate, the data network bandwidth, and the message receiving rate. Since the injection rate is higher than the receiving rate on the CM-5, the network capacity is limited. The network capacity defines the number of message packets that can be injected without the receiver removing any. The measured network capacity for a variety of partition sizes of the CM-5 is reported in [2]. The CM-5 network capacity increases as the number of processors increases. This is because

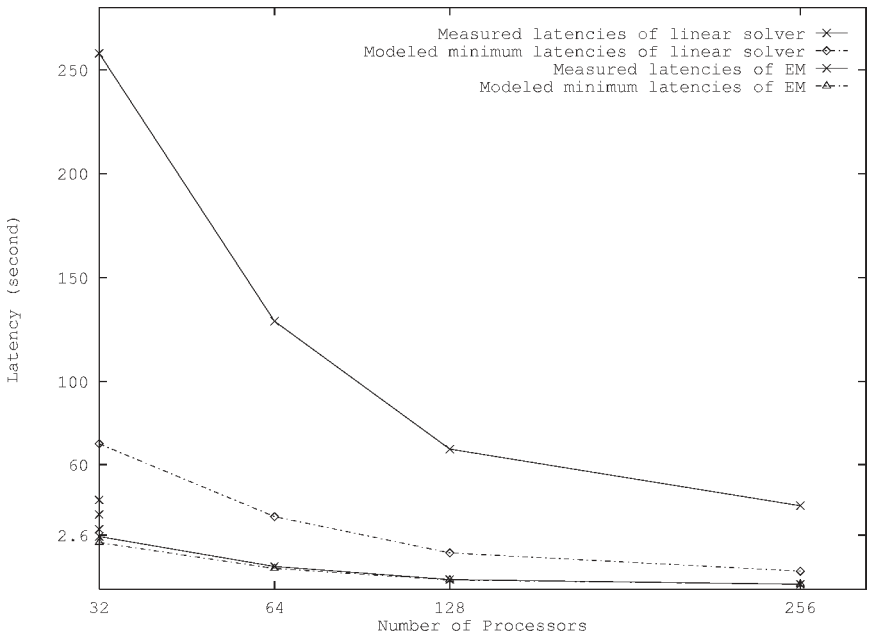

Fig. 9. Latency limits and latency changing patterns for the EM simulation and the linear system solver on the CM-5.

the chances of message collisions in the network are reduced as the network is scaled. For example, the CM-5 network capacity is 9.89 packets/node for a 16-processor partition, and increases to 11.3 packets/node for a 128-processor partition.

The *effective network bandwidth* defines an average data transmission rate constrainted by the three architecture transmission rates. The maximum effective network bandwidth, $B_{\text{max\_eff}}$, in theory, is the difference between the injection rate and the receiving rate ($Q_{\text{send}} - Q_{\text{rev}}$). In practice, it is not a constant, but is a function of the number of processors, which represents the maximum data transmission rate without any data contention:

$$B_{\text{max\_eff}}(p) = \frac{C(p)}{T}, \tag{9}$$

where $C(p)$ is the network capacity in Megabytes, which is a function of p, the number of processors, and T is time in seconds spent to transmit the data. The transmission time T is determined by the network bandwidth, B_{net} and the data size, $K(p)$:

$$T = K(p)/B_{\text{net}}, \tag{10}$$

where the data size, K is also a function of p, the number of processors. Substituting (10) into (9), the maximum effective bandwidth becomes

$$B_{\text{max\_eff}}(p) = \frac{C(p)B_{\text{net}}}{K(p)}. \tag{11}$$

For a fixed-size problem, increasing the number of processors would decrease the data size, $K(p)$, for each processor to transmit. Furthermore, increasing the number of processors would cause the network capacity, $C(p)$, to increase. Of course, the message traveling distance also increases as the number of processors increases. However, the negative effects caused by the distance increase on the CM-5 are trivial. This is because as the number of processors increases, the data-parallel operations supported by pipeline bits across the long wires are much less distance sensitive. The experiments in [9] show that message transmission latencies and bandwidths are independent of the partition size on the CM-5. The network latencies vary only slightly with the number of network levels crossed. Therefore, based on (11), as the number of processors increases, the effective network bandwidth of the CM-5 would generally increase, and the network latency would decrease. The minimum network latency can be determined by

$$L_{\text{min}} = \frac{K(p)}{B_{\text{max\_eff}}(p)}, \tag{12}$$

where $K(p)$ decreases and $B_{\text{max\_eff}}(p)$ increases as p increases.

Recall that the network latencies in the EM simulation on the KSR-1 significantly increased as the number of processors increased (see Fig. 2), and the network latencies in the linear system solver on the KSR-1 only moderately increased (see Fig. 7 (left)). As we have discussed in previous sections, the overhead patterns inherent in the two programs are different; thus higher overhead in the EM simulation caused a higher latency increment on the KSR-1 architecture.

In contrast, the CM-5 latency variations are fundamentally different. The minimum CM-5 network latency defined in (12) is computation dependent. The two minimum latency curves for the EM simulation and the linear system solver along with the measured latencies of the two computations on the CM-5 are plotted in Fig. 9. The number of packets of communications in both computations as run on various system partition sizes are collected by Prism, which was used for the calculation of the minimum network latencies. In the EM simulation, the measured latency curve is very close to the minimum latency curve. This indicates that the EM computation using the data-parallel mode on the CM-5 is highly effective, and fully takes advantage of the network architecture. There is a significant gap between the measured latency curve of the linear system solver and its minimum latency curve. The gap shrinks as the number of processors increases. This confirms two conclusion from our study. First, the linear system program suffers a great deal from using the data-parallel model on the CM-5. Second, since the CM-5 architecture has the unique network scaling feature for data-parallel computation, the network latency of a computation approached its ideal value as the number of processors increased. Fig. 9 also confirms that the network latencies of a computation regardless of being in favor of data-parallel or not, decrease as the number of processors increases. This is a fundamental reason why the CM-5 is highly scalable compared with other existing multiprocessor systems, such as the KSR-1.

7. Summary and conclusions

Here we summarize the performance implications and comparisons from our experiments for the execution patterns of the shared-memory and the data-parallel programs on the KSR-1 and on the CM-5.

- On the KSR-1, network latency caused by data migrations, cache coherence and process scheduling tends to increase as the number of processors increases. The computing efficiency may be maintained to a certain level if the sizes both of the problem and of the system are increased. The shared-memory on the KSR-1 expresses and executes a fine-grained and control-structured algorithm well.
- A computation-intensive program in the shared-memory KSR-1 system may often become a memory-demanding program in a data-parallel implementation on the CM-5, due to large data replications in each node for simultaneous data operations. The large memory allocation requirement may cause I/O bottlenecks in the computation. The I/O latency can be reduced by increasing the number of processors in the computation on the CM-5. However, up to a certain point of increasing the number of processors, the network latency may start to increase because the computation load in each processor is not high enough to keep all the processors busy all the time. Again, computing efficiency may be maintained at a certain level if both the sizes of the problem and of the system increase for both the KSR-1 and the CM-5.
- This work provides comparative views of execution patterns of the shared-memory and the data-parallel programs on the KSR-1 and on the CM-5. The results and comparisons are more-or-less system and architecture dependent. Therefore this approach has its limit in addressing global issues, such as comparisons among the programming models.

Acknowledgement

We are grateful to the anonymous referees for their comments and suggestions to improve the quality of the paper. We also thank the technical support for the CM-5 machines from Los Alamos National Lab and from the University of Illinois. We appreciate N. Wagner's careful reading of the paper and his helpful comments.

References

[1] D.H. Bailey, Twelve ways to fool the masses when giving performance results on parallel computers, *Supercomputer Review* (August 1991), 54–55.

[2] E.A. Brewer and B.C. Kuszmaul, How to get good performance from the CM-5 data network, in: *Proceedings of the 8th International Parallel Processing Symposium*, IEEE Computer Society Press, April 1994, pp. 858–867.

[3] R.F. Harrington, *Field Computation by Moment Methods*, Krieger Publishing Co., Malabar, FL, 1982.

[4] High Performance Fortran Forum, High Performance Fortran Language Specification Version 1.0, *Scientific Programming* **2**(1–2) (1993), 1–70.

[5] Kendall Square Research, *KSR-1 Technology Background*.

[6] A.C. Klaiber and J.L. Frankel, Comparing data-parallel and message-passing paradigms, in: *Proceedings of 1993 International Conference on Parallel Processing*, August 1993, Vol. II, pp. 11–20.

[7] A.C. Klaiber and H.M. Levy, A comparison of message passing and shared-memory architectures for data parallel programs, in: *Proceedings of the 21th Annual International Symposium on Computer Architecture*, April 1994, pp. 94–105.

[8] D. Kranz et al., Integrating message-passing and shared-memory: early experience, in: *Proceedings of 4th SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993, pp. 54–63.

[9] T.T. Kwan, B.K. Totty and D.A. Reed, Communication and computation performance of the CM-5, in: *Supercomputing '93*, IEEE Computer Society Press, November 1993, pp. 192–201.

[10] G. Li and T. Coleman, A new method for solving triangular systems on distributed-memory message-passing multiprocessors, *SIAM J. Sci. Statist. Computing* **10** (1989), 382–396.

[11] Y. Lu et al., Implementation of electromagnetic scattering from conductors containing loaded slots on the Connection Machine CM-2, in: *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, SIAM Press, 1993, pp. 216–220.

[12] C. Lin and L. Snyder, A comparison of programming models for shared memory multiprocessors, in: *Proceedings of the 1990 International Conference on Parallel Processing*, Vol. II, August 1990, pp. 163–170.

[13] O.M. Lubeck et al., The performance realities of massively parallel processors: a case study, *Supercomputing '92*, IEEE Computer Society Press, November 1992, pp. 403–412.

[14] C. Moler, Matrix computation on distributed memory multiprocessors, in: *Hypercube Multiprocessors 1986*, M.T. Heath, ed., SIAM, Philadelphia, PA, pp. 181–195.

[15] Thinking Machines Corporation, *The Connection Machine CM-5 Technical Summary*, 1993.

[16] X. Zhang and Y. Yan, Comparative modeling and evaluation of CC-NUMA and COMA on hierarchical ring architectures, *IEEE Trans. Parallel Distrib. Syst.* **6**(12) (1995), 1316–1331.

[17] X. Zhang, Y. Yan and K. He, Latency metric: an experimental method for measuring and evaluating program and architecture scalability, *J. Parallel Distrib. Computing* **22**(3) (1994), 392–410.

[18] X. Zhang, Y. Yan and K. He, Evaluation and measurement of multiprocessor latency patterns, in: *Proceedings of the 8th International Parallel Processing Symposium*, IEEE Computer Society Press, April 1994, pp. 845–852.