Single System Image: Need, Approaches and Supporting HPC Systems

RAJKUMAR
Operating Systems Group
Centre for Development of Advanced Computing
(A Scientific Society of Department of Electronics, Government of India)
2/1, Ramanashree Plaza, Brunton Road
Bangalore - 560 025, Karnataka, India
Email: raj@cdacb.ernet.in URL: http://cdacb.ernet.in/~raj

Abstract - High performance computing on proprietary or commodity hardware is gaining wide acceptance. For this to be practicable, it is important that systems provide a single system image at any one (or more) of the following levels: Hardware, Operating System, Message Passing Interfaces, Language/Compiler, or Tools. Single-system image greatly enhances the program's portability and at the same time increases the availability of a wide range of software (tools or applications). An operating system can exhibit single-system image by supporting a parallel file system, parallel commands, multi-user and multithreaded kernel for single/multi-user machine. The operating system should incorporate all these features without additional primitives or commands but having the same existing formats. This paper presents the needs and approaches for building a single system image, and a survey of systems supporting single-system image.

Keywords: Network, Parallel, Distributed, NOW, COW, Operating Systems, Single System Image

1. Introduction

Recently many computer systems supporting high performance computing have emerged. Each one of them is built using different architectures. They are classified based on how their processing elements are connected to memory subsystems and the distances between processing elements. This has lead to the emergence of High Performance Computing (HPC) systems: SMPs (Symmetric Multi-Processors—all processors are tied to common global memory), MPPs (Massively Parallel Processors—each node has its own local memory and they are interconnected using a proprietary switch), NOWs (Network of Workstations—multiple single user workstations are connected together using commodity networks), and COWs (Cluster of Workstations—multiple uniprocessor or SMP systems are connected together using a low-latency and high bandwidth interconnection network). Recent technology trends in high-speed/low-latency local area networks have lead to the convergence of hardware in MPPs and NOWs (COWs).

To move into the real era of high performance computing, the operating environment on these machines must provide a unified system view, which is popularly called as a Single System Image (SSI). The user need not be aware of the underlying system architecture to use these machines effectively. The operating environment must be familiar (provide the same look and feel of the existing system) and convenient to use. The user must be provided with the view of globalized file system, processes, and network. This allows the user to access system resources such as memory, processors, network, etc., transparently irrespective of whether they are available locally or remotely.

A single-system image can be provided at any one or more of the following levels:

- Hardware Level
- Operating System Level
- Message Passing Interfaces Level
- Language/Compiler Level
- Tools Level

An operating system can support single system image by supporting a parallel file system, parallel commands, multi-user and multithreaded kernel for single/multi-user machines. The operating system should incorporate all these features without the need

of additional primitives or commands but having the same existing formats.

The programming model offered on HPC machines must be consistent. That is, the programs written for MPPs must be portable to SMPs/NOWs/COWs with minimal or no change. This can be made possible with the availability of standard message passing interfaces such as MPI (Message Passing Interface) on different architectures. The clear distinction between shared and distributed memory systems is fading; the Distributed Shared Memory (DSM) systems allow us to use distributed memory systems similar to shared memory systems. The DSMs can be implemented either by software or hardware means.

Relative to traditional MPPs, a NOW offers the potential for better cost-effectiveness, increased scalability, and decreased hardware and software development time. The distinguishing point between NOWs and MPPs will soon be the operating system only. The development of operating systems supporting multiprocessor workstations, parallel architectures, and distributed client server systems is crucial for the success of high performance computer architectures.

Modern workstation operating systems have to provide the necessary support for efficient parallel program execution in an environment shared with sequential applications. The goal of these systems is to pool resources in a NOW to provide better performance for both parallel and sequential applications. To realize this goal, the operating system must support gang-scheduling of parallel programs, identify idle resources in the network (CPU, disk capacity/bandwidth, memory capacity, network bandwidth), allow for process migration to support dynamic load balancing, and provide support for fast inter-process communication for both the operating system and user-level applications.

Most of the operating systems supporting a single-system image are built as a layer on top of the existing operating systems and perform global resource allocation. This strategy makes the system quickly portable, tracks vendor software upgrades, and reduces development time. This shows new systems can be built quickly by mapping new services onto the functionality provided by the layer beneath.

The importance of parallel programming has to spread well beyond the traditional systems as many commercial HPC systems are available in the market. For this scenario to be realized, downward scalability is more important than upward scalability; that is, a parallel program written for an SMP or NOW must run efficiently on a single processor machine as well [1]. For example, a program written using Java programming language runs on any (uni/multi-processor) machine irrespective of its underlying architecture.

The remainder of this paper discusses: benefits of single system image; operating systems supporting a single system image such as Solaris MC, GLUnix, and SUPER-UX; parallel file systems; parallel tools; distributed shared memory systems; POSIX threads interface, message passing libraries, parallelizing compliers, architecture neutral programming language—Java, and IBM CICSPlex System Manager for MVS/ESA.

2. Benefits of Single System Image

The following are the benefits of supporting single system image on HPC systems:

- It frees the end-user from having to know where an application will run.
- It frees the operator from having to know where a resource (an instance of resource) is located.
- It does not restrict the operator or system programmer who needs to work-on a particular region; the end-user interface (hyperlink—makes easy to inspect consolidated data in more detail) can navigate to the region where a problem has arisen.
- There is reduction in the risk of operator errors, with the result that end-users will see improved reliability and higher availability of their systems.
- It allows to decentralize operations while maintaining control, or to centralize operations without the need to increase the skills required.
- It greatly simplifies system management operations; actions affecting multiple resources can be achieved with a single command, even where the resources are spread among multiple systems on different machines.
- It provides location independent message communication. Because SSI provides a dynamic map of the message routing as it occurs in reality, the operator can always be sure that actions will be performed on the current system.

- It knows and keeps track of the locations of all resources, so that there is no longer any need for system operators to be concerned with their physical location while carrying out system management tasks.

The benefits SSI available to operators also apply to system programmers. This reduces the time, effort and knowledge required to perform tasks, and allows current staff to handle larger or more complex systems.

3. Solaris MC: A Distributed Operating System for Clusters

Solaris MC (Multi-Computer) is a distributed operating system for a multi-computer, a cluster of computing nodes connected by a high-speed interconnect [2]. It provides a single system image, making the cluster appear like a single machine to the user, to applications, and to the network. The Solaris MC is built as a globalization layer on top of the existing kernel, as shown in Figure 1. It extends operating system abstractions across the cluster and preserves the existing Solaris ABI/API and hence runs existing Solaris 2.x applications and device drivers without modifications. The Solaris MC consists of several modules: C++ and object framework support, globalized processes and file system support, and networking [3].

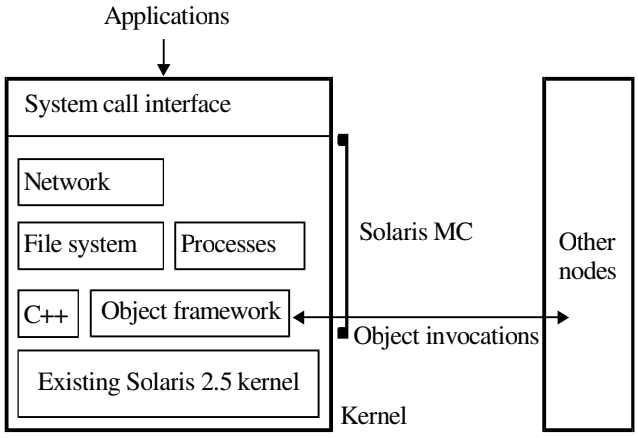

Figure 1: Solaris MC Architecture

The interesting features of Solaris MC include the following:

- Extends existing Solaris operating system
- Preserves the existing Solaris ABI/API compliance
- Provides support for high availability
- Uses C++, IDL, CORBA in the kernel
- Leverages Spring technology

The Solaris MC uses object-oriented framework for communication between nodes. The object-oriented framework is based on CORBA and provides remote object method invocations. It looks like a standard C++ method invocation to the programmers. The framework also provides object reference counting, notification to object server when there are no more references (local/remote) to the object. Another feature of the Solaris MC object framework is that it supports multiple object handlers.

A key component in proving a single system image in Solaris MC is a global file system. It provides consistent access from multiple nodes to files and file attributes. It uses caching for high performance. It uses a new distributed file system called ProXy File System (PXFS), which provides globalized file system without the need for modifying the existing file system.

The second important component of Solaris MC supporting a single system image is its globalized process management. It globalizes process operations such as signals. It also globalizes the `/proc` file system providing access to process state for commands such as `ps` and for the debuggers. It supports remote execution, which allows to start up new processes on any node in the system.

Another important component of Solaris MC supporting a single system image is its globalized networking and I/O. The Solaris MC allows to have one or more network connections and protocol processing can be carried out on any of the nodes, even if just one node has a network interface. It uses a packet filtering architecture to support networking capability.

4. SUPER-UX Operating System

NEC's SX Series supercomputers (SX-2, SX-3, etc.) use a common UNIX based operating system, SUPER-UX, which conforms to POSIX standards and also supports the popular BSD extensions [4]. It is enhanced extensively to provide the robust and reliable production environment required for supercomputing. The important features of SUPER-UX such as supercomputing enhancements, single-system image, supercomputing file system, batch subsystem, and networking are discussed in the following subsections:

Supercomputing Enhancements

The supecomputing enhancements of SUPER-UX include advanced I/O caching, extensive parallel processing support (such as HPF/SX, message passing-MPI/SX, etc.), and universal resource management facilities. Advanced features for accounting, check-point/restart, and optional automatic operation are provided.

The Single System Image

All configurations/models of the SX-4 Series supercomputers support multitasking and distributed message passing programing models. Irrespective of the configuration, SUPER-UX provides a single-point of entry to the system, a single point operation, and delivers a total system that is seamlessly integrated. The result of which is a single system image; it provides the flexibility necessary to configure a highly efficient system for any operational requirement.

Supercomputing File System (SFS)

The SFS provides capabilities suited to the demanding requirements of a true supercomputing environment. They include support for parallel I/O technologies, asynchronous I/O, bufferless I/O, flexible disk caching, contiguous block allocation strategies, and maximum file sizes which can exceed 2048 Terabytes in size. SFS can span multiple physical devices and device overflow is supported.

Batch Subsystem

The SUPER-UX batch subsystem is based on the widely used NQS (Network Queuing System). It is enhanced substantially to provide improved resource scheduling, job monitoring, and control.

Networking

The SUPER-UX is seamlessly integrated into different network environments (SX-4 can be connected to HiPPI, FDDI, and ATM networks). All UNIX compatible communication functions and protocols are supported including widely used network management (TCP/IP, DNS, SNMP) and EGP (Exterior Gateway Protocol) gateway. OSI standard support coupled with sockets and STREAMS interfaces, TLI (Transport Layer Interface) library, and X-Windows and Motif graphical interfaces, allows to integrate the SX-4 Series into any computing environment. In addition, it also supports NFS and DFS along with DCE, including Kerberos security enhancements for advanced network computing environments. NetAdmin software enables a single host management system (such as workstation) to control all networked host machines in addition to the SX-4 Series system.

5. GLUnix: A Global Layer Unix for NOW

The Berkeley's Network of Workstations (popularly called NOW supercomputer) harnesses the power of clustered machines connected via high-speed switched networks. It leverages commodity workstations and operating systems, and delivers performance comparable to MPPs. A unified system view on NOW is provided by a software system, called GLUnix. It is built as a layer on top of existing operating systems and performs global resource allocation on a network of workstations [5].

The GLUnix (Global Layer Unix) glues together individual UNIX operating systems to provide a single system image of the machines in a network. It strives to make all resources in a network of workstations are transparently available to each user. Thus all the idle processing power, memory, network capacity, and disk bandwidth is made available in a fair manner for both sequential and parallel applications [6]. To provide such functionality, GLUnix supports coscheduling of parallel programs, idle resource detection, resource migration, fast user level communication, remote paging, and fault tolerance (high availability).

The GLUnix enables efficient execution of parallel programs and improves the performance of memory or I/O intensive applications without modification of the base kernel. It is built using minimal set of standardized features of the underlying operating system and hence ensures its portability. It is portable to any system which supports inter-process communication, process signaling for scheduling and migration, and access to machine load statistics for idle resource detection; porting of GLUnix to new hardware platform is significantly easier.

GLUnix and LSF (Load Sharing Facility) Differences

There is not much functional difference between GLUnix and LSF, but they differ significantly in terms of targeted goals. Both detect idle resources for load balancing. GLUnix has better support for parallel programs and has the basis for a single system image; process id's in GLUnix are globally valid—the

user can kill a GLUnix process from any node. However, the fundamental difference is that GLUnix serves as the foundation for more advanced research studies into a wide range of areas. These include:

- Resource scheduling for integrating parallel and sequential applications
- Automated resource parameterization
- Mechanisms for layering systems extensions on top of existing operating systems
- Providing a single system image (SSI) as a layer on top
- Fault tolerance for a SSI
- Defining the user interface for a NOW

6. Parallel File System

Though the speed of most components of HPC systems has been steadily increasing, the I/O subsystem has not been keeping pace. This limitation of hardware can easily be overcome by implementing multiprocessor file system called parallel file system [7].

Most existing multiprocessor file systems use Unix-like file model. In this model, file is viewed as an addressable and linear sequence of bytes. Applications can issue requests to read or write data of contiguous subranges in that sequence of bytes. Access to these files can be improved by using parallel file system. A parallel file system typically declusters (i.e., scatters the blocks of each file across multiple disks) allowing parallel access to file. This parallel access reduces the effect of the bottleneck imposed by the relatively slow speed disk. Although the file is actually scattered across many disks, the underlying parallel structure of the file is hidden from the application. The syntax of commands used to access files in this way is the same as those accessing files using conventional Unix-like interfaces.

When data access is requested (when request size is larger than the declustering unit size), the file system accesses multiple disks in parallel and delivers higher bandwidth to the application, and possibly hides any latency caused by disk seeks. When there is a disparity between the request size and the declustering unit size, most of the individual requests generated by parallel applications may not be serviced in parallel. This limitation is overcome by providing applications with the ability to fully control this declustering according to their own needs.

7. Parallel Tools

Each node of any HPC system runs a separate copy of the operating system such as UNIX. It has introduced new problems in managing the user's environment. The familiar basic Unix commands (`ls`, `ps`, `cp`, `rm`, etc.) have to be provided to manage the user environment, since the user needs simple and familiar ways to: copy a file to the local file space of each node; find all processes running on all nodes; test a condition on all nodes; without getting swamped with output. On large machines (HPC systems) these commands are not useful unless they take advantage of parallelism in their execution.

Many HPC systems provide a full Unix environment on the entire system. This has many advantages, including providing a standard environment which the users are familiar with. The goals of these programs must include: familiarity to Unix users—they should have easy-to-remember names (e.g., `p`<unix-command-name>); scalability—they should be fast enough to use with the same regularity that users use `ls` and `ps`, regardless of the number of nodes; not generate too much output. An example system supporting parallel version of Unix commands is C-DAC's PARAM.

8. Distributed Shared Memory Systems

The efficient programming model on distributed memory systems is message passing, but programming using this model is hard compared to that available on shared memory systems. Shared memory systems offer simple and general programming model, but they suffer from scalability. An alternate cost effective solution is to build a DSM (Distributed Shared Memory) system which exhibits simple and general programming model of shared memory systems; and scalability of distributed memory systems. The DSM systems such as KSR1 offer physically distributed and logically shared memory, which is an attractive solution for large scale high-performance computing.

DSM systems can be implemented by using software or hardware solutions [8]. The characteristics of software implemented DSM systems are: they are usually built as a separate layer on top of the message passing interface; they take full advantage of the application characteristics; virtual memory pages, objects, and language types are units of sharing. The implementation can be achieved by: com-

piler implementation; user-level runtime package; operating system level (inside or outside the kernel). That is, a shared memory programming model for a distributed memory machine can be implemented either solely by runtime methods [9], by compile time methods [10], or by combined compile time and runtime approach [11]. A few representative software DSM systems are Munin, ThreadMarks, Linda, and Clouds.

The characteristics of hardware implemented DSM systems are: better performance (much faster than software DSM approaches), no burden on user and software layers (full transparency), fine granularity of sharing (cache block); extensions of the cache coherence schemes (snoopy or directory), and increased hardware complexity (not unreasonable). Typical class of hardware DSM systems are CC-NUMA (e.g.,Dash), COMA (e.g.,KSR1), and Reflective memory systems (e.g.,RMS).

9. POSIX Threads Interface

Threads are an emerging model for expressing concurrency on multiprocessor and multicomputer systems [12]. (Thread is defined as a piece of code that can execute in concurrence with other threads. And a program can have multiple threads executing simultaneously.) On multiprocessor systems, threads are primarily used to simultaneously utilize all the available processors and give the user an easy way to achieve parallel computation. However, on uniprocessor systems, threads are used to utilize system resources effectively by exploiting the asynchronous behavior (computation and communication overlap) of an application.

The standard interface for writing portable multithreaded programs is *pthreads* (POSIX Threads). The programs written using *pthreads* are portable across all machines irrespective of whether they are uni/multiprocessor systems.

10. Message Passing Libraries

The message passing libraries allow to write efficient parallel programs for distributed memory systems. They provide routines to accomplish send, receive, and other message passing operations. One of the most widely accepted standards for message passing is MPI (Message Passing Interface) [13]. MPI is a specification for message passing libraries, designed to be standard for distributed memory, message passing, and explicit parallel computing. This interface attempts to establish a practical, portable, efficient, and flexible standard for message-passing. MPI is available on most of the HPC systems including SMP machines.

11. Parallelizing Compilers: xHPF and spf

To use the computing power of parallel computers, applications have to be parallelized. Millions of lines of code are written for sequential machines and many programmers are not aware of parallel programming. Hence there is need for compilers which automatically parallelize the sequential code. There are many tools available. One such tool is *xHPF*, which is from Visual Numerics [14]. The xHPF compiler automatically parallelizes existing Fortran programs for a wide variety of distributed memory systems. Another tool, *spf*, which is from Visual Numerics automatically parallelizes existing Fortran programs for a wide variety of shared memory systems [14].

There is a need for programming systems that take a machine independent specification of parallelism from the programmer as an input and tailor the code to suit individual parallel target systems. It is also known that automatic parallelization to support machine independent programming is too hard. The *xHPF* compiler creates target code for distributed memory systems by leveraging several message passing protocols including PVM and MPI. The *spf* compiler creates target code for shared memory systems by leveraging POSIX threads. In both cases, the user just needs to a create configuration file which provides machine dependent details such as data alignment.

The standard message passing interface, MPI, is now available on both shared and distributed memory systems. Hence, the target code created by parallelization tools will run on any platform. This is how, a single system image can be achieved by using automatic parallelizing tools.

12. Java: An Architecture Independent Programming Language

Programming languages which allow to develop platform-independent applications will support SSI at language level. One such language, which is developed by SUN Microsystems is Java. Java is defined as

follows: It is a new, simple, object-oriented, distributed, portable, architecture neutral, robust, secure, multi-threaded, interpreted, and high-performance programming language [15]. Java is mainly intended for the development of object-oriented network based software for Internet applications. Its syntax is similar to C and C++, but it omits semantic features that make C and C++ complex, confusing, and insecure. It does not support some of the more difficult to use features of C++ such as pointers. It also features built-in safety mechanisms (like absence of pointers) which provide some level of security on network. Hence, Java as a logical successor to C++ can also be called as C++--++ (C-plus-plus-minus-minus-plus-plus i.e., remove some difficult to use features of C++ and add some good features).

Java is the first language to provide a comprehensive, robust, platform-independent solution to the challenges of programming for the Internet and other complex networks. Java features portability, security and advanced networking without compromising on performance. Java was initially designed to address the problems of building software for small distributed systems to embed in consumer devices. As such, it is designed for heterogeneous networks, multiple host architectures, and secure delivery. To meet these requirements, compiled Java code had to survive transport across networks, operate on any client, and assure the client that it is safe to run.

Java's future is promising. It is robust, object-oriented, and portable (source and executable byte code) i.e., Java's application byte code runs on any platform without any modification or recompilation; Java byte codes are interpreted by the Java Virtual Machine (JVM-the architecture neutral and portable language platform of Java) running on a local machine. Java integrates the flexibility of interpreted languages and power of compiled languages. Java comes bundled with a suite of classes for GUI (Graphical User Interface), multithreading, networking, file I/O, and the like. To add to this, APIs (Application Program Interface) for database access (Java DataBase Connectivity), more robust multimedia processing, and remote object access are being developed.

Java achieves superior performance by adopting a scheme by which the interpreter can run at full speed without needing to check the runtime environment. The automatic garbage collector runs as a low-priority background thread, ensuring a high probability that memory is available when required, leading to better performance. Applications requiring large amounts of computer power can be designed such that compute-intensive sections can be rewritten in native machine code as required and interface with the Java platform.

13. IBM CICSPlex System Manager for MVS/ESA

CICSPlex SM provides a real-time single-system image of all CICS regions and resources that make up the enterprise's transaction processing environment [16]. This enables you to avoid the problems of growing complexity by enabling management of the entire CICSplex (or any chosen subset) as easily as if it were a single CICS region.

By building an internal representation of the CICSplex topology, CICSPlex SM allows automatic routing of operational requests to all appropriate regions. This means that the operator no longer has to know the location of a CICS resource before working with it. The end user interface can be used to consolidate information into simplified status displays.

14. References

[1] Arvind. *The future of parallel programming*. Proceedings of the International Conference on Parallel Processing, India, 1994.

[2] Yousef A Khalidi, Jose M Bernabeu, Valda Matena, Ken Shirriff, Moti Thadani. *Solaris MC: A Multi-Computer OS*. 1996 USENIX Conference, January 1996.

[3] Ken Shirriff. *Nex Generation Distributed Computing: The Solaris MC Operating System*. Japan Talk, Solaris MC Research Group, Sun Microsystems Laboratories.

[4] NEC. *The SUPER-UX Operating Systems*. Supercomputer SX-4 Series, Technical Brochure, NEC Corporation, Japan.

[5] Amin Vahdat, Douglas Ghormley, and Thomas Anderson. *Efficient, Portable, and Robust Extensions of Operating System Functionality*. Computer Science Division, University of California, Berkeley, December 5, 1994.

[6] Remzi H Arpaci, Andrea C Dusseau, and at al. *The Interaction of Parallel and Sequential Workloads on a Network of Workstations*.

Computer Science Division, University of California, Berkeley.

[7] Nils Nieuwejaar and David Kotz. *The Galley Parallel File System*. Department of Computer Science, Dartmouth College, NH 03755-3510, Auguest 1996.

[8] Jalica Protic, Milo Tomasevic, and Velijko Milutinovic. *Distributed Shared Memory: Concepts and Systems*. Tutorial delivered at the International Conference on High Performance Computing, Seoul, Korea, April 28 - May 2, 1997.

[9] K Li and P Hudak. *Memory coherence in Shared Virtual Memory systems*. ACM TTOCS, November 1989.

[10] S Hiranandani, K Kennedy, and C. Tseng. *Compiling Fortran D for MIMD distributed memory machines*. CACM, August 1992.

[11] Sandhya Dwarkadas, Alan L, Cox, and Willy Zwaenepoel. *An Integrated Comple-Time/Run-Time Software Distributed Shared Memory System*. Operating System Review, December 1996.

[12] Rajkumar. *Multithreaded and Distributed Computing*. Tutorial delivered at the International Conference on High Performance Computing, Seoul, Korea, April 28 - May 2, 1997.

[13] Blaise M Barney. *The Message Passing Interface*. Tutorial delivered at the International Conference on High Performance Computing, Seoul, Korea, April 28 - May 2, 1997.

[14] Visual Numerics. *Parallel Processing Solutions*. Technical Brochure, Visual Numerics Inc., USA, http://www.vni.com, 1997.

[15] Sun. *The Java Language: A White Paper*. Sun Microsystems, 1996.

[16] IBM. *CICSPlex System Manager for MVS/ESA Version 1 Release 1*. `http://ncc.hursley.ibm.com/cicsplex/cpsm111.html`, 1997.

About the Author

Rajkumar, (born on April 14, 1971 in a small village Koutha (B), Bidar district, Karnataka, India) is a Member of the Technical Staff, Operating Systems Group, Centre for Development of Advanced Computing, Bangalore, India. He received the Bachelor of Engineering degree in Computer Science from the University of Mysore (JMIT) in the year 1992 and the Master of Engineering degree in Computer Science from the University of Bangalore (UVCE) in the year 1995. He was awarded the Dharma Ratnakara Memorial Trust *Gold Medal* for securing the I Rank in the year 1992. Besides being a visiting faculty for the Bangalore University, he is conducting the popular course on *Java* by Email which is broadcasted around the globe through the Internet. He has coauthored the books *Microprocessor x86 Programming* and *Object Oriented Programming with C++*.

Rajkumar lectured extensively on advanced technologies such as Parallel, Distributed and Multithreaded Computing, Client/Server Computing, Internet and Java all over India as a part of access'96 conference organized by the C-DAC and the Sun Microsystems. His research papers have appeared in National and International Conferences. In *High Performance Computing ASIA '97* Conference and Exhibition held at Seoul, Korea, he delivered the tutorial on Multithreaded and Distributed Computing. He serves as technical sessions chair on *HPCC Models for Graphics and Multimedia*, International Conference on Imaging Science, Systems, and Technology (CISST'97) and on *Unified System View by Software Means*, International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97) to be held at USA. He is on the Program Committee, CISST'97. His research interests include Programming Paradigms and Operating Environments for Parallel and Distributed Computing. He can be reached by the e-mail at `raj@cdacb.ernet.in` and by the URL `http://cdacb.ernet.in/~raj`.