A general-purpose scalable operating system: *SSS-CORE*

Takashi Matsumoto
Department of information science, University of Tokyo
7-3-1, Hongo, Bunkyo-ku, Tokyo, Japan 113
+81-3-5802-8829
tm@is.s.u-tokyo.ac.jp

Shigeru Uzuhara
AXE Inc.
24-2, Shimokamo, kitanonogami, Sakyo-ku
Kyoto, Japan 606
+81-75-724-1966
uzu@axe-inc.co.jp

Kei Hiraki
Department of information science, University of Tokyo
7-3-1, Hongo, Bunkyo-ku, Tokyo, Japan 113
+81-3-3812-2111 ex.4039
hiraki@is.s.u-tokyo.ac.jp

ABSTRACT

Recently, difference between a Massively Parallel Processor and a network of workstations (NOW) almost disappears from hardware point of view. However, current performance of NOWs is still much lower than that of MPPs because of huge overheads of operating systems. Furthermore, both MPPs and current NOWs are not general-purpose because they have not succeeded in giving users (1) a single system image with flexible resource allocation, or (2) multi-tasking environments with shared memory spaces.

SSS-CORE is a general purpose scalable operating system for NUMA parallel distributed systems. It provides very efficient multi-tasking environment with timesharing and space partitioning. Furthermore, it tries to allow each parallel application to achieve maximum performance by cooperation of user and kernel level resource allocation and scheduling and by offering low-latency high-throughput memory based communication facilities. SSS-CORE also provides a low-overhead mechanism that allows information transfer between kernel and user level.

KEYWORDS

operating system, workstation cluster, scalability, fast communication, memory-based communication

1 INTRODUCTION

For realizing a general-purpose computing environment on a large-scale multicomputers (MPP) and a network of workstations (NOW), performance of the system is often compromised by introduction of a multi-tasking system image, protection mechanisms and virtualization of system resources. Consequently, current parallel processing environment including NOWs and MPPs are mainly used as a back-end computer for numerical computations or as a distributed processing environment for almost independent single tasks. It is why various optimization technique in parallel processing and light-weight communication mechanisms require exclusive use of the system if the system is used as a parallel computer.

We have developed a general-purpose operating system kernel SS–CORE[1] since 1989. SS–CORE supports a multicomputer with Uniform Memory Access (UMA) architecture, where all the memory accesses from CPUs have approximately same latencies. Unfortunately, UMA architecture is not scalable and the number of processors is limited. A multicomputer with NUMA (Non-Uniform Memory Access) architecture is scalable because NUMA architecture does not have limitation of the number of CPUs in a system if a sufficient network is provided. a NOW, an MPP with distributed memory architecture and a distributed shared memory system are examples of NUMA systems.

SSS-CORE[2] is a general purpose scalable operating system for NUMA parallel distributed systems. The main objectives of SSS-CORE are (1) to offer a general-purpose single system image to users of MPPs and NOWs, (2) to exploit maximum performance on multiple parallel job environments from a hardware system including network communication, and (3) to give users a conventional UNIX-like and X-like API.

It has following features:

1. SSS–CORE supports distributed implementation such as NOWs,
2. realizes general-purpose multi-tasking environment by time-sharing and partitioning,
3. gives a single system image to every user,
4. makes a high-performance and fair multi-tasking environment by a brand-new resource allocation and scheduling mechanism using low-overheads communications between user programs and the kernel[2],
5. achieves high-bandwidth low-latency user-level communications and synchronizations with protection (Memory-based communication facility)[5],
6. supports efficient distributed shared memory (Asymmetric Distributed Shared Memory architecture)[5, 7], and
7. offers a resource information server for cooperating with optimizing compilers and application programmers.

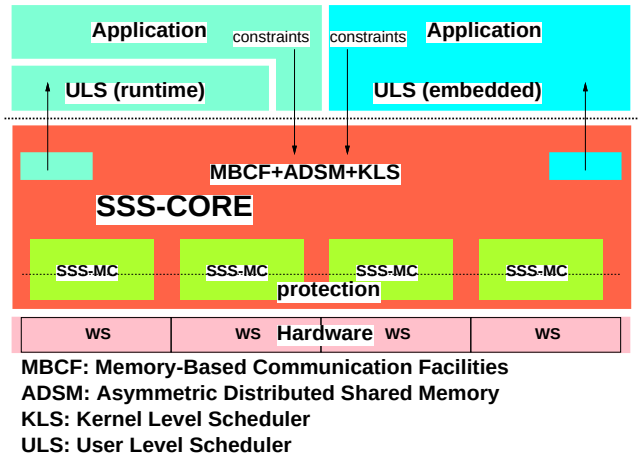

Figure 1: SSS-CORE basic architecture

2 Basic architecture of SSS-CORE

We show as the basic architecture of the SSS–CORE in Figure 1. The system consists of workstations. Each workstation is either a single processor system or an SMP. Each node has an SSS–MC (Micro Core), which is a simple node operating system that implements intra-node resource managements, device drivers, basic user interfaces and communication protocols (TCP/IP and UDP/IP) to external systems. Within the SSS–CORE system, both kernel programs and user programs communicate and synchronize with each other through the Memory-Based Communication Facilities. For programmers' or compilers' convenience of using an efficient distributed shared memory system, support mechanisms for the Asymmetric Distributed Shared Memory are presented by the SSS–CORE. The Kernel Level Scheduler realizes a global resource management and fair-share schedulings with users' constraints.

3 Memory-Based Communication Facilities

3.1 Outlines of MBCF

In general-purpose parallel and distributed systems, performance of the protected and virtualized user-level communications/synchronizations is the most crucial issue to realize efficient parallel execution environments. We proposed a novel high-speed user-level communications/synchronizations scheme "Memory-Based Communication Facilities (**MBCF**)" suitable for the general-purpose system with off-the-shelf communication-hardware. The MBCF realizes the direct remote-memory-accesses to user-task-spaces and offers programmers and compilers a global shared-memory space.

The MBCF emulate functions of the Memory-Based Processor (MBP)[8, 9] by software routines with various high-speed implementation techniques. Outline of the MBCF mechanism is as follows. Remote memory accesses are invoked by explicit system calls for MBCF functions. First a user-program prepares an MBCF packet in user-mode and executes the MBCF requesting system-call. Then in the system-call for requesting MBCF the kernel-level routine makes a inter-node communication-style packet and transmit it through a conventional network interface card(NIC). Finally the interrupt routine for receiving MBCF packet processes the packet and directly executes the remote access specified in the packet and returns a reply if necessary.

On the MBCF scheme, Protections and virtualizations in communications and synchronizations are replaced with those of memory accesses. This replacement makes high-speed implementations of the mechanisms feasible without using any additional hardware mechanisms. The MBCF only exploits the TLBs and/or the MMUs of the node processors in order to translate logical remote addresses.

3.2 High-speed implementation techniques

To make implementations of the MBCF as high-performance as possible, we apply many techniques on software engineering and exploit advanced architectural features of latest commercial processors. We list these techniques below.

- Direct accesses to target logical spaces
- On-memory-synchronizations
- Cache-conscious programming
- MBCF-dedicated requesting system-call
- MBCF-dedicated receiving interrupt routine
- Wide but fixed variety of MBCF functions

The above techniques do not assume any special-mechanisms of processors and they are applicable to all computers. The following mechanisms are useful for high-speed implementation of the MBCF, and they are implemented in most of latest advanced processors (e.g. SuperSPARC, UltraSPARC).

- TLB corresponding to the coexistence of multiple contexts
- Physical-address-tagged cache
- Light-weight context switching
- User-privileged memory-access capability in kernel mode
- Page aliasing capability of the MMU/TLB
- Registers dedicated for system-calls or interrupts

If some architectural mechanisms described above are absent from a processor, there are some performance degradations but the MBCF can be implemented with software emulations of the mechanisms.

Owing to the techniques of software engineering and advanced mechanisms of the node processors, the MBCF scheme is more flexible and less expensive than the message-passing-style system interface for communication.

3.3 Snoopy Spin Wait algorithm

To reduce overheads of user-level synchronizations, the Snoopy Spin wait (**SS-wait**)[3] algorithm is mainly used in the SS–CORE system and the SSS–CORE system. The SS-wait is a kind of spin wait algorithm using synchronization variables. If we use simple spin waiting for synchronizations, there are some risks to suffer starvation states or large loss of CPU time. To avoid these situations, the SS-wait exploits meta information on resource allocations, schedulings, and degree of progress of threads.

In SS-wait algorithm, first of all an application program checks a synchronization variable, and leaves the spin loop successfully if the variable represents the completion of the synchronization. If the variable shows the incompletion, the meta information is checked and possibility of large loss of CPU-time is calculated. If there is no possibility or small possibility, the program reenter the start point of the SS-wait. If there is big possibility or evidence of large CPU-time loss, the program switches the current context (process/thread) to one of other ready contexts through user-level scheduler and/or kernel-level scheduler. In the SS–CORE system, the meta information is stored in the central shared memory, and resource management information is shared with kernel (in a read and write mode) and application programs (in read-only mode). In the SSS–CORE system, since there is no central shared memory, another information distribution mechanism is required. Hierarchical mechanism of collecting and distributing information is implemented in the SSS–CORE using the MBCF. This mechanism efficiently multi-cast the meta information to nodes, and user programs can check it only with simple memory references as in SS–CORE.

SS–CORE and SSS–CORE (Scalable SS–CORE) are named after this "SS-wait".

3.4 Performance comparison with commercial MPP

Table 1 shows a performance comparison with user-level communication mechanisms of MPPs. We use workstation cluster of Axil 320 model8.1.1 (Sun SPARCstation20 compatible, 85MHz SuperSPARC x 1) machines with Fast Ethernet SBus Adapter 2.0 (100BASE-TX). The Ethernet system cannot guarantee the arrival of transmitted packets. The implementation of the MBCF using Fast Ethernet has a considerably complicated protocol which guarantees packet-arrival and fifoness of point-to-point communications, but buffering mechanisms can avoid performance degradations in usual unsaturated communications. Moreover cache-conscious programming also prevents the overhead of the complicated protocol from becoming large.

Entries in Table 1 except for MBCF and UDP / SUNOS are quoted from following papers. Figures without any marks are from [10]. Figures on SP–1/SP–2 (with ‡ mark) are from [11].

The implementation of the SSAM in the table have no mechanism to guarantee packet arrival or FIFO property. The practical SSAM would suffer larger overheads than the SSAM in the table.

Considering two points:

- As for the level of protections and virtualizations, the MBCF is the highest of all, and
- As for the performance of the raw communication hardware, the MBCF is the lowest of all,

figures in the table 1 show that methodology and implementation techniques of the MBCF/100BASE-TX are excellent and remarkable.

4 Kernel Level Scheduler

A parallel optimizing compiler generally assumes that system resources are used by a single application. Our goal is to run object codes efficiently in a multi-user and multi-process environment. In such an environment, a process can achieve higher speedup when allocated resources satisfy its requirements and preferences. In SSS–CORE, a process can use scheduling constraints to specify its requirements and preferences of the number of processors to use, communication cost between processors, memory access costs, and process migration. For example the *fixed processors constraint* expresses a constant number of processors a process requires. The *variable processors constraint* enables a process to be allocated variable number of processors.

Since resources are allocated to satisfy each process requirements, a mechanism must be arranged to coordinate fair sharing of resources. Fairness can be achieved by managing priorities according to the amount of used resources and strength of given scheduling constraints and scheduling in priority order. Aging priorities according to these terms realizes fairness.

[1]To guarantee packet arrival and fifoness, user-level MBCF-protocol is implemented using UDP/IP system-calls.

Machine	Peak bandwidth (Mbytes/s)	Round-trip latency(μs)
SP–1 + MPL/p	8.3 / 8.7‡	56 / 75‡
Paragon + NX	7.3	44
CM–5 + Active Message	10.0	12
SP–2 + MPL/udp	10.8‡	554.0‡
SS–20 cluster + SSAM (ATM 156Mbps)	7.5	52
SS–20 cluster + UDP[1]+ SUNOS4.1.4 (100BASE-TX)	5.5	800
SS–20 cluster + MBCF (100BASE-TX)	**11.2**	**49**

Table 1: Performance of MBCF

Priority is based on following values; (1)amount of used resources: U_r, (2)strength of scheduling constraints: R_c, (3)degree of constraints satisfaction: $S_c = 0$ *or* 1, (4)amount of wasted resources: W_r, (5)presence of waiting process: $f_w = 0$ *or* 1. The aging value of a process is computed from next expression.

$$aging = (U_r R_c S_c + W_r) f_w * t - C_r(1 - t)$$

Smaller the value, higher the priority. C_r is the aging coefficient. To prevent priority values from diverging, the sum of the aging values of processes that were running before the time slice is equally divided and distributed to waiting processes.

Under general environment where multiple processes execute simultaneously, system must be designed to bear situations when physical memories are exhausted. Selecting victim pages from those that are less frequently accessed and have less re-reference cost will enhance system performance.

In SSS–CORE, a kernel level scheduler allocates resources to each process by looking into the resource management tree. This way, resources that most fit for the use of a process can be allocated. The resource allocation within a process is left to user level scheduler, which freely re-allocate resources into internal threads [2].

5 Asymmetric Distributed Shared Memory

To execute shared-memory-based parallel programs efficiently in a system without hardware-remote-cache mechanisms, some software cache scheme must be performed by the OS and/or user codes. In the MBCF system, considering code optimizations for inter-node communications, the full user-level cache scheme (User-level DSM:**UDSM**), where the MBCF interfaces are directly used in user-level codes to maintain software-remote-caches, is better than OS-based software DSMs. In other words, the UDSM scheme is more suitable to exploit flexibilities of the MBCF for the optimizations of communication and execution than OS-based DSMs. However, in the UDSM case the user-level execution-code must explicitly maintain, check and modify software-controlled-cache tags. Up to now, neither processors are fast enough to neglect the overhead of handling software cache-tags nor optimizing compilers are sophisticated enough to hide and/or reduce it. Inter-node communications occur only at shared-write situations and in usual applications the number of shared-writes is much less than shared-reads. Considering these characteristics we introduce a brand-new remote cache scheme "Asymmetric Distributed Shared Memory (**ADSM**)" [5].

In conventional page-based (i.e. OS-based) DSMs, not only read-cache-misses but also shared-writes are supported by the TLB/MMU mechanisms of node processors using write-protection traps and page-fault traps. Though the ADSM is one of page-based cache schemes, only read-cache-misses are supported by the TLB/MMU mechanisms. For each shared-write in the ADSM scheme, a proper sequence of instructions which maintains the cache consistency of the system is inserted into the user-level execution-code by the optimizing compiler. In the MBCF system, the user-level code-sequences include the MBCF-dedicated system-calls and invalidate (or update) remote caches while modifying the local cache-states. Since the instructions for the consistency maintenances at shared-writes are explicitly inserted in the application codes, there is large room for various code optimizations. Strategy of handling shared-reads (read-cache-misses) and that of handling shared-writes are different. Therefore we call this scheme the "asymmetric" DSM.

The combination of the MBCF and the ADSM can realize an efficient distributed shared memory environment on Network of Workstations or distributed-memory multiprocessors without DSM-dedicated hardware.

6 SSS–CORE/NOW Ver.1.1

We are currently developing SSS–CORE/NOW Ver1.1 for arbitrary number of SPARCstation 20 connected by 10BASE-T or 100BASE-TX Ethernets. In order to im-

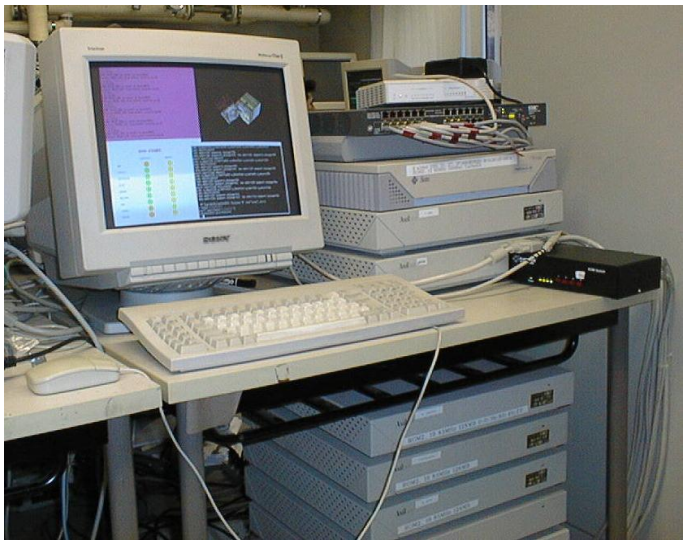

Figure 2: A view of SSS–CORE/NOW Ver1.1

plement features of SSS–CORE efficiently, SSS–CORE has not been developed by modifying existing operating systems such as UNIX and MACH, but is developed from scratch including boot codes [2] and device drivers.

The following list shows implemented features of SSS–CORE/NOW Ver.1.1.

- Multi-task (Time-Sharing System + Partitioning System)
- Memory Protection, Task Protection
- Two types of system-calls: light-weight system-call without context-switch and conventional system-call with a possible context-switch
- Communication protocols for external systems (UDP/IP, TCP/IP, TFTP, TELNET)
- Intra-node shared memory (page alias capability)
- Graphical console, keyboard input, mouse input
- Simple and kernel-integrated shell (batch-file execution, command arguments, environment variables, command-line editing, command retrieval, debugging support)
- SunOS emulations (outsourcing of file-related facilities)
- Memory-Based Communication Facilities (MBCF)
  - Virtualization of physical nodes, protection and security
  - Elastic memory barrier facility[12] (a kind of relaxed memory barrier)
  - Option of reporting a MBCF transaction result
  - WRITE, WRITE_F, READ, SWAP, etc.
  - Memory-Based FIFO
  - Memory-Based Signal
  - Hierarchical multi-cast and ack-combining[8] (update, invalidate, etc.)
  - present implementation with commodity hardware of Ethernet and/or Fast Ethernet, and coexistence with TCP/IP and UDP/IP
- Hierarchical mechanism of collecting and disclosing resource information
- Page management mechanism for Asymmetric Distributed Shared Memory: ADSM
- Kernel-level resource scheduler and manager which handles hints and constraints from users/applications (Now under implementing)
- A optimizing (cross) compiler for MBCF + ADSM, the compiler supports C language with PARMACS macros

[2] SSS–CORE/NOW Ver.1.1 executes on a disc-less workstations, and it is booted via network.

7 Conclusion

We have developed a general-purpose scalable operating system SSS–CORE, which shows that a high-performance computing system can be constructed by combination of workstations, off the shelf network hardware and an operating system dedicated to parallel processing.

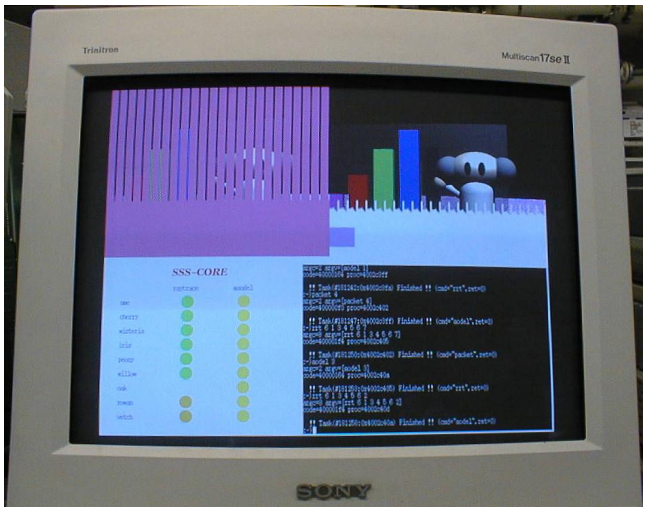

Figure 3: Screen of parallel ray-tracing

8 Descriptions of the demonstration

Demonstration of SSS–CORE shows following application programs running with SSS–CORE on a cluster of SPARCstation 20s.

- **Parallel ray-tracing**
  Pixel calculations are assigned to multiple nodes

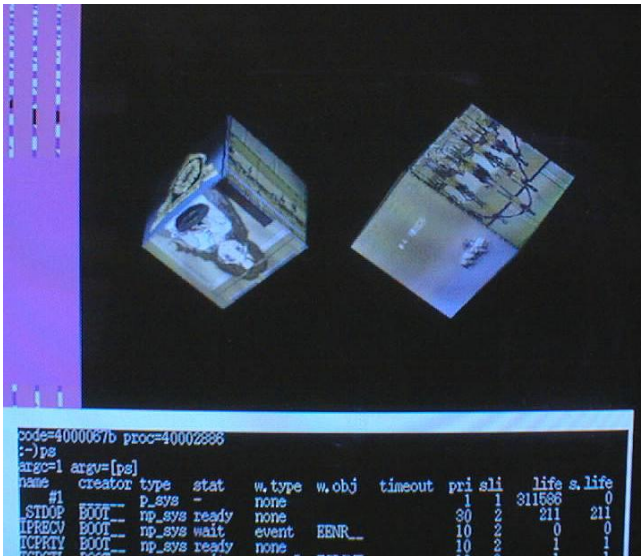

Figure 4: Screen of movies on flying cubes

in block-cyclic manner, then the results are transferred to a node with a display by MBCF (see Figure 3).

- **Movies on flying cubes**
  Each node replays two movies from cinepak format and does 3D transformations, then transfers pixel data to a node with a display by MBCF. Outputs from nodes are texture-mapped on two flying cubes. See Figure 4 [3] .
- **Multi-screen movie**
  Each node replays a movie from cinepak format and transfers pixel data to a display-node without any transformations. 8 movies are simultaneously replayed in a display for demonstrating high-bandwidth of MBCF.
- **Numerical computation using an optimizing compiler for SSS–CORE**
  A numerical application programmed with shared memory model is compiled by our optimizing compiler[7] for ADSM of SSS–CORE. LU-decomposition and FFT are used in the demonstration.

Acknowledgment

The work is supported by Advanced Information Technology Program (AITP) of Information-technology Promotion Agency (IPA), Japan. Authors thank Mr. Kenji Morimoto and Mr. Junpei Niwa for developments of demonstration applications on SSS–CORE.

REFERENCES

[1] T. Matsumoto, Y. Negishi, S. Uzuhara, and T. Moriyama: Classification of Parallel Programs Based on Grain Size Information and Its Application to a Multiprocessor Resource Management Scheme (in Japanese). *Proc. 7th Conf. of Japan Society for Software Science and Technology*, pp.133–136 (October 1990).

[2] T. Matsumoto and K. Hiraki: Resource management schemes of the general-purpose massively-parallel operating system: SSS–CORE (in Japanese). *Proc. 11th Conf. of Japan Society for Software Science and Technology*, pp.13–16 (October 1994).

[3] T. Matsumoto: Synchronization and Processor Scheduling Mechanisms for Multiprocessors (in Japanese). *IPS Japan SIG Reports*, Vol.89, No.99, ARC–79–1, pp.1–8 (November 1989).

[4] Y. Nobukuni, T. Matsumoto, and K. Hiraki: Resource Management Methods for General Purpose Massively Parallel OS *SSS–CORE. Proc. of International Symposium on High Performance Computing*, Springer-Verlag LNCS 1336, pp.255–266 (November 1997).

[5] T. Matsumoto, T. Komaarashi, S. Uzuhara, and K. Hiraki: The Asymmetric Distributed Shared Memory Using Memory-Based Communication Facilities (in Japanese). *Proc. of Computer System Symp.*, IPS Japan, pp.37–44 (November 1996).

[6] T. Matsumoto and K. Hiraki: Performance of Memory-Based Communication Facilities Using Fast Ethernet (100BASE-TX) (in Japanese). *Proc. of Computer System Symp.*, IPS Japan, pp.101–108 (November 1997).

[7] J. Niwa, T. Inagaki, T. Matsumoto, and K. Hiraki: Efficient Implementation of Software Release Consistency on Asymmetric Distributed Shared Memory. *Proc. of Int. Symp. on Parallel Architectures, Algorithms and Networks (I-SPAN'97)*, IEEE Computer Society Press, pp.198–201 (December 1997).

[8] T. Matsumoto and K. Hiraki: A Shared-Memory Architecture for Massively Parallel Computer Systems (in Japanese). *IEICE Japan SIG Reports*, Vol.92 No.173, CPSY 92–26, pp.47–55 (August 1992).

[9] T. Matsumoto, K. Nishimura, T. Kudoh, K. Hiraki, H. Amano, and H. Tanaka: Distributed Shared Memory Architecture for JUMP–1:A General-Purpose MPP Prototype (Invited Paper). *Proc. of Second Int. Symp. on Parallel Architectures, Algorithms and Networks (I-SPAN'96)*, IEEE Computer Society Press, pp.131–137 (June 1996).

[10] T. von Eicken, A. Basu, and V. Buch: Low-Latency Communication Over ATM Networks Using Active Messages. *IEEE Micro*, pp.46–53 (February 1995).

[11] M. Snir et al.: The Communication software and parallel environment of IBM SP2 *IBM Systems Journal*, Vol. 34, No. 2, (1995).

[12] T. Matsumoto and Hiraki, K.: Elastic Memory Consistency Models (in Japanese). *Proc. of 49th Annual Convention of IPS Japan*, Vol.6, 5K–3, pp.5–6 (September 1994).

[3] This program is based on xanim (PDS). Copyrights of animated pictures on the cubes belong to Nagoya TV Co. and Sunrise Inc.