Design and Implementation of OpenMP Library for λ Computing Environment

Keigo Goda[*], Mai Imoto[*], Ken-ichi Baba[†], Noriyuki Fujimoto[*] and Masayuki Murata[*]
[*]Graduate School of Information Science and Technology, Osaka University
[†] Cybermedia Center, Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
Telephone: +81-6-6879-4542
Email: [*]{k-gouda, m-imoto, fujimoto, murata}@ist.osaka-u.ac.jp, [†]baba@cmc.osaka-u.ac.jp

***Abstract*—Grid technology has been studied and developed by numerous researchers in recent years. Data in conventional grid environments are changed by using TCP/IP. However, as long as the architecture is based on packet switching, highly efficient computing is difficult to achieve. We thus propose a new architecture, the λ computing environment, where network switches and computing nodes are connected to one another with optical fibers, thereby offering high-performance computing by establishing an optical wavelength path between shared memories on computing nodes.**

We established the lambda computing environment using the AWG-STAR system, and designed a data-sharing structure for the OpenMP library, which is a parallel-computing programming language utilizing shared memory. Moreover, we evaluated its performance against existing parallel computing in a PC-cluster environment by executing OpenMP applications.

I. Introduction

The demand for large-scale computation such as that involved in gene-information analysis, image processing, and global-environment simulation that treats enormous volumes of data has recently been increasing. Research into grid-computing technology and high-speed data transmission have been actively pursued to satisfy these demands. TCP/IP is usually used for communications in grid-computing environments such as that in control messages and data exchanges between computing nodes. However, TCP/IP has various detrimental effects in these environments. For example, lost packets need to be retransmitted as some may be lost along the route from the source node to the destination node because of traffic congestion caused by the volume of data that TCP/IP itself transmits. Furthermore, the transmission bandwidth may be decreased by controlling the congestion.

New technology that enables high-speed and highly reliable communications is therefore needed to satisfy the demand for grid computing. Research into Wavelength Division Multiplexing (WDM) technology has been the main target of development, and IP over a WDM network has also been studied and developed to provide high-speed transmission on the Internet based on WDM technology. Moreover, standardization of the routing technology for the Internet, called GMPLS, which is a communications technology that uses more optical technologies for the lower layers than WDM technology, has also been advanced by IETF [1]. Research into optical packet switches based on optical technology has also begun, which is aimed at attaining actual IP communications in a photonic network.

However, many such technologies presuppose the current Internet technology. That is, an IP packet is treated as information units, and the target of research and development has become on how to carry it at high speed along a network. Therefore, as long as architecture based on packet-switching technology is being focused on, high-quality communications to all connections will be difficult to achieve and computing throughput on the grid environment will remain low.

We thus propose a new architecture that we call the λ computing environment, which has wavelength paths between computing nodes and optical switches to achieve high-speed and highly reliable communications in grid-computing environments [2]–[5]. We can attain high-speed and highly reliable data exchange or data sharing in the λ computing environment because computing nodes do not utilize the conventional TCP/IP network but establish wavelength paths as a communications channel in advance. We focus on the data communication method using high-speed network, so we target the single CPU machine as the computing node not but the multiple CPU machine as a cluster PC in this paper.

Related work [2]–[5] has reported the evaluation of architecture that has accomplished distributed-parallel computing in a λ computing environment. All these have presumed a shared-memory architecture for sharing of data, which is required for parallel computing. Nakamoto [3] proposed utilizing a virtual optical ring as a shared memory, taking the coherence between shared memory in the virtual ring and the caches of all computing nodes into consideration. Taniguchi [5] analyzed how the network topology and the method of controlling cache coherency influenced performance, by using a semi-Markov process. However, these studies only evaluated the environment by simulation and modeling.

We previously implemented a Message Passing Interface (MPI) library [6] and executed an MPI application, which is a library specification to pass messages via a network, share data, and synchronize processes. Although MPI is the de-facto standard for parallel computation, it is a library for processors that have no shared memories. However, there is another model of parallel computation that assumes shared memory between multiple processors. This model is more suited to the λ computing environment, because it can utilize the shared memory that we have considered for this.

Our aim was to execute an OpenMP application utilizing the shared memory in the λ computing environment, and implement the OpenMP library and data sharing structure. OpenMP is a standard specification of parallel computation for the shared-memory model [7]. We can parallelize programs by inserting comment statements or pragma statements into existing Fortran or C (C++) applications.

We utilized the AWG-STAR system developed by NTT Photonics Laboratory [8] as an instance of the λ computing environment. The AWG-STAR system is an information-sharing network based on WDM technology and data is transmitted through an Array Waveguide Grating (AWG) router, which processes wavelength routing. The AWG-STAR system offers a unique feature where the Shared Memory Board (SMB) on all nodes connected to the AWG router is provided as an extended memory from the computing node, and the data is automatically synchronized in nodes when the data is written on the SMB. We implemented the OpenMP library to utilize the SMB of the AWG-STAR system efficiently and evaluate performance.

The rest of the paper is organized as follows. Section II explains the λ computing environment. Section III discusses our design of the OpenMP implementation on the AWG-STAR system, and our evaluations of the environment are given in Section IV. We present our conclusion and directions for future work in Section V.

II. λ Computing Environment: New Distributed Computing Environment

We will first explain the λ computing environment that we propose as a new distributed-computing environment in this section and then the AWG-STAR system that we utilized to establish it. We will then describe how distributed and parallel computation in the λ computing environment are executed.

A. WDM Technology for λ Computing Environment

The λ computing environment is based on WDM technology. The computing nodes and optical switches that it is composed of are connected with optical fibers. One hundred or more wavelengths, which are expected to be 1000 or more in the future, are multiplexed in an optical fiber by WDM or DWDM (Dense WDM) technology and they provide a broadband communications line for computing nodes. WDM technology is usually considered to be a lower-layer technology that attains GMPLS and IP over a WDM network. We used WDM technology in this study to establish wavelength paths and utilize these as an exclusive communications line.

We can therefore accomplish high-speed and highly reliable data exchange or data sharing in a λ computing environment because the computing nodes do not utilize a conventional TCP/IP network but establish wavelength paths as an exclusive communications channel in advance. The details on the established wavelength paths are shown in Fig. 1.

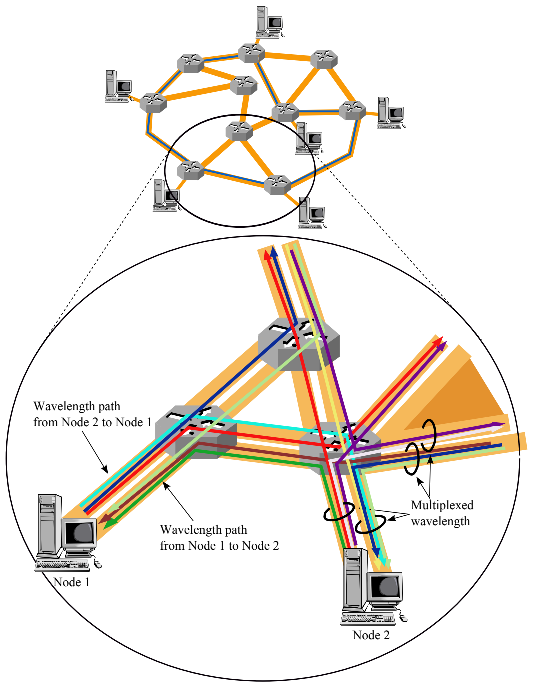

Fig. 1. Established Wavelength Paths

TABLE I
Assignment of Wavelengths in I/O Ports of AWG Router

Input \ Output	Port 1	Port 2	Port 3
Port 1	56	58	60
Port 2	58	60	62
Port 3	60	62	64

B. AWG-STAR system

1) Brief Overview of AWG-STAR system: The AWG-STAR system is a platform for an information-sharing network accomplished by WDM technology and wavelength routing using AWG routers. Computing nodes connected to the AWG router physically configure a star topology, but are logically a ring (see Fig. 2). The AWG router processes optical signals without transforming them into electrical ones, which provides high-speed transmission. All nodes are equipped with an SMB, which has a shared memory that can contain identical data at the same address over all nodes of the AWG-STAR system. While conventional systems need apparent instructions to transmit data, data in this system are automatically sent to the optical ring network when they are written on the SMB, and data on the other SMBs of all computing nodes are updated in real time. Furthermore, they only need to access their own SMBs to read data from the shared memory. This system achieves high-speed data sharing because it runs in the background at the hardware level.

2) Configuration of AWG-STAR System: The AWG router can configure a wavelength path by dynamically changing the wavelength of the optical signal. It has 32 input ports and 32

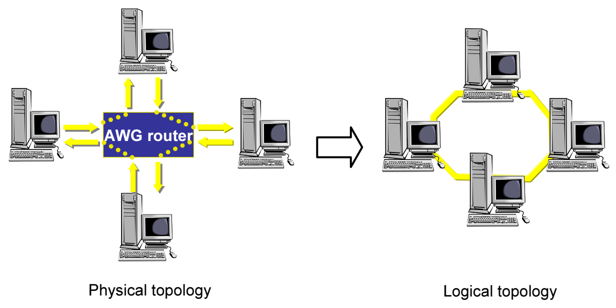

Fig. 2. Network Topology in AWG-STAR System

output ports, and which output port each incoming signal uses is determined by the wavelength. Table I shows a instance of how wavelengths are assigned. We can see, for example, that when wavelength 58 enters input port 2, the signal exits output port 1. The figure for the wavelengths, however, is specific to the AWG-STAR system.

3) Access to Shared Memory and Data Sharing: There are two ways to access the shared memory. The first is Direct Memory Access (DMA) using the SMB function, and the second is addressing with a pointer. Shared memory is on the SMB, which is connected to the computer with a PCI local bus. It therefore requires time to transmit data through the PCI bus, as well as to write to and read from the shared memory. This leads to the delay time from the CPU to the shared memory being slower than that to the local memory. Data also have to go around the optical ring to be updated on all the computing nodes' SMBs .

One control token is on the optical ring and the computing nodes share data by attaching the sending frame (address, data, control code, and CRC) to the control token. There are two patterns to update the shared memory; the first is where computing nodes write to their own shared memory, and the second is where they receive data updated from other computing nodes.

Computing Nodes Write to Own Shared Memory: Here, the computing nodes first write the data to be shared on their own SMBs. They next receive the control token and attach the sending frame to the end of the series of frames attached to the token. After that, the computing node passes the control token to the next node. When the token has finished going around the optical ring and all the other computing nodes have received the updated data, the node deletes the data from the control token. Figure 3 shows the model for this.

Computing Nodes Receive Data for Updating: A computing node checks if there are data to update that are attached to the control token after it is received. If there are, it reads the data and updates its own SMB and passes the control token to the next computing node. Figure 4 shows the model for this.

Figure 5 summarizes the operation to update data in the shared memory. One computing node writes on the shared memory and updates the data. It waits for the control token that the data are attached to. Other computing nodes receive the data attached to the control token and update their own

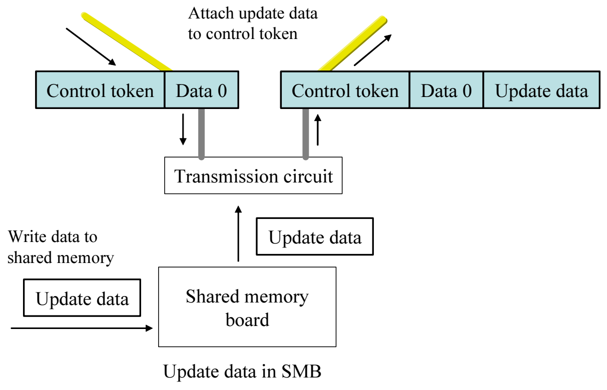

Fig. 3. Write Data to Shared Memory

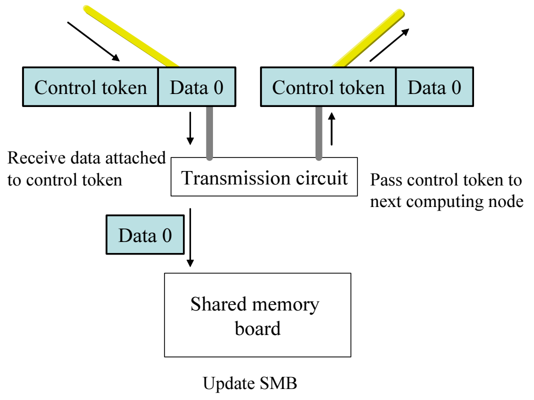

Fig. 4. Receive Data for Updating

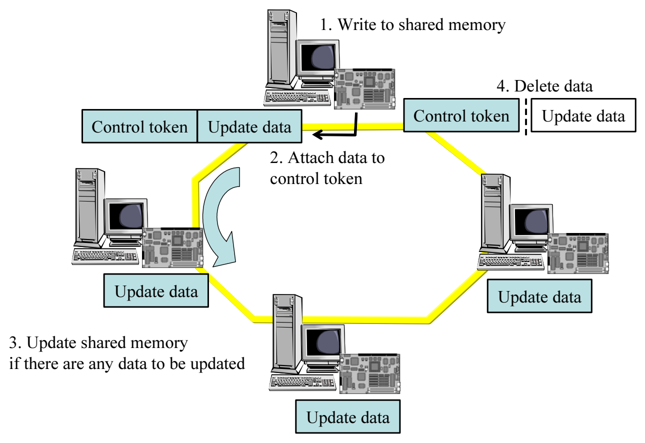

Fig. 5. Operation to Update Data

SMBs. The data are deleted when they are passed around the ring.

4) Access Delay Time for AWG-STAR System: The access delay time from the CPU to the shared memory is slower than that to the local memory, because of the delay time to pass through the PCI bus and to share data. Table II lists the specifications for the SMB and the access speed via the PCI bus. The data-sharing time consists of two factors: the first

TABLE II
SPECIFICATIONS FOR SMB

Transmission speed of optical ring	2.152 Gbps
Data size for every transmission	1 KByte
Processing time for frame transmission	500 ns
Maximum transmission rate to SMB	64 MBytes/s
Maximum transmission rate from SMB	80 MBytes/s

is the time to treat the control token, and the second is the propagation delay. The series for treating the control token (add/delete the sending frame and update the SMB) takes about 500 ns. The propagation delay in the optic fiber is 5 ns/m.

C. Distributed Parallel Computing and OpenMP

Distributed parallel computing models can be classified into two categories based on whether the computing environment has a shared memory. That is, the first model is distributed parallel computing with a distributed memory, and the second is that with a shared memory. Representative of the distributed-memory programming model is the Message Passing Interface (MPI), whose processes share and synchronize data with message passing. This means programmers must design when and which data are exchanged. OpenMP, on the other hand, which is representative of the shared-memory programming model, presupposes all CPU has a shared memory. Therefore, processes with the OpenMP application can exchange and share data without distributed-memory aware programming. Our research group has already implemented the MPI library for the λ computing environment [6]. Our aim was to execute the OpenMP application in the λ computing environment in this research.

OpenMP is one of the application programming interface (API) standards for shared-memory parallel programming, which supports multi-platforms in C/C++ and Fortran language. It consists of a set of compiler directives, library routines, and environmental variables that influences run-time behavior.

Programmers using OpenMP develop a parallel program by inserting compiler directives called OpenMP directives into the source code. An OpenMP directive is an instruction to express parallelism and data sharing and programmers tell the compiler to parallelize a specified part of the program through these directives. Therefore, they do not need to write the detailed behavior of a parallel program in the source code. Figure 6 shows a simple example of the OpenMP source code. The statement "pragma" on the 2nd line is an OpenMP directive. The loop part from the 3rd line is executed in parallel. All the other parts without any directives are executed sequentially.

The generation of the executable object of the parallel program by the OpenMP compiler consists of approximately two stages (see Fig. 7). The first involves transformation of the source code to an intermediate code. The OpenMP compiler interprets the OpenMP directive in the source code, and generates the intermediate code based on the information

```
double pi = 0.0;
#pragma omp parallel for reduction(+:pi)
for (i = 0; i < N; i++) {
    double x = (i + 0.5) * w;
    pi += 4.0 / (1.0 + x * x);
}
```

Fig. 6. Example of OpenMP Source Code

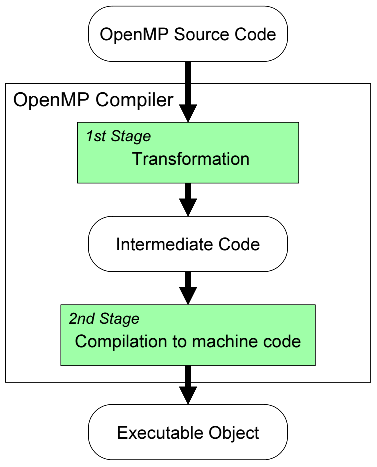

Fig. 7. Overview of OpenMP Compiler

provided with the directives. The intermediate code contains the specific code required for parallel processing. For example, library function calls for communications and synchronization are inserted. The second stage is the source-code compilation. The intermediate code is converted to an executable object with machine language.

As was previously discussed, programmers can develop their programs to parallelize easily because the compiler converts them according to their directives. In addition, there are no codes that depend on a specific computing environment in the source code of the parallel program. However, the OpenMP compiler must be implemented so that it is dedicated to each computing environment because the intermediate code depends on a specific computing environment including the hardware, OS, and communication library.

III. DESIGN OF OPENMP

We had to design an OpenMP compiler generating an intermediate code for the AWG-STAR system to implement OpenMP in the λ computing environment. OpenMP was originally designed for a shared-memory system such as that in a symmetric multiple processor (SMP), which shares a single memory with multiple processors in a computer. However, the AWG-STAR system has a different scheme for shared memory. That is, as multiple-computing nodes are interconnected via a network and the memory of each computing node is independent in the AWG-STAR system, their memories are treated as a single memory with specific functions.

We utilized distributed shared-memory (DSM) technology that enables all or part of the memory in computing nodes to be shared to construct a shared-memory system in a distributed system in which computing nodes are generally connected via a network. Most existing DSM technology uses a software-distributed shared memory (SDSM) that controls memories in all computing nodes as a single shared memory through software. The AWG-STAR system used in our study, on the other hand, has a DSM system that maintains the consistency of memories through hardware.

Our implementation of the OpenMP compiler was based on an existing OpenMP compiler called OMPi [9] for the SMP environment. We chose this compiler because its source code was available and the general idea behind it was simple. We needed to design it for the AWG-STAR system since OMPi was for the SMP environment. However, as the AWG-STAR system has a local memory in each computing node and a shared memory on the SMB, this configuration is similar to an SDSM system sharing part of the local memory of computing nodes. A related study on the implementation of OpenMP using an SDSM [10] was therefore useful for our own implementation regarding the issue of porting the environment from the SMP to the cluster.

When we implemented OpenMP on the AWG-STAR system, we needed to establish:

1) The execution mechanism for the parallel-execution section,
2) The data-sharing scheme, and
3) The synchronization primitives.

The methods we used to accomplish all three on the AWG-STAR system will be described in the sections that follow.

A. Execution Mechanism for Parallel Execution Section

It is necessary to achieve parallel processing utilizing multiple-computing nodes to use OpenMP on the AWG-STAR system.

There are sequential-execution and parallel-execution sections in OpenMP programs. Our method of implementation clearly distinguishes between a computing node called the master, which executes both sequential sections and parallel sections, from other computing nodes called workers, which execute parallel sections. That is, workers receiving a request from the master begin executing the parallel-execution section.

Both the master and workers in our method execute an identical executable object. However, as soon as the program is executed, the workers immediately go on standby, and wait while the master completes executing a sequential-execution section. When the master has finished executing the sequential-execution section, it requests the workers to start executing the parallel-execution section by utilizing barrier synchronization. Computing is accomplished in parallel with multiple-computing nodes. After the parallel-execution section has been executed, barrier synchronization is again conducted, and the workers again return to standby; the master then executes the sequential-execution section (see Fig. 8).

Barrier synchronization is a mechanism that synchronizes more than two computing nodes at the same time. The details on barrier synchronization are described in Section III-C.

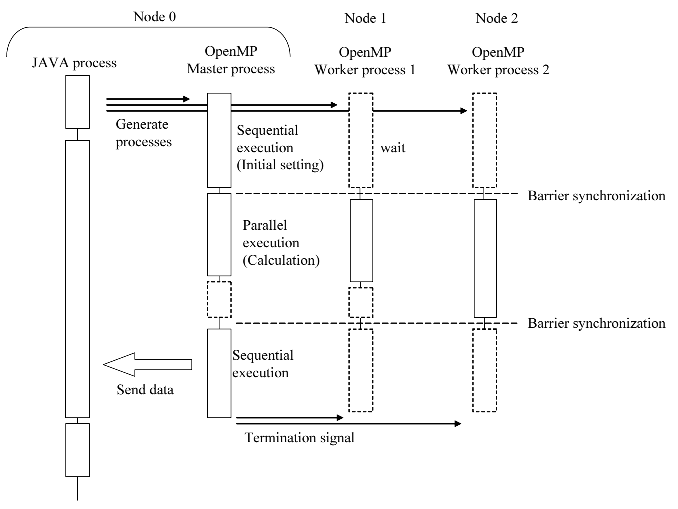

Fig. 8. Execution Sequence Model for OpenMP (Example of Application Processes in Sec. IV-C)

B. Data-sharing Scheme

The data sharing in OpenMP means the values of variables in the program are shared. In other words, it is possible to read from and write to variables between computing nodes executing parallel computing. In OpenMP, all variables declared on the outside of the parallel-execution region are implicitly shared. For example, the code in Fig. 6 shares variable `pi` between computing nodes while parallel processing is being executed.

A variable is shared with the SMB that the AWG-STAR system offers. In other words, this is achieved by allocating the memory area of variables to the shared memory on the SMB. Variable content is therefore automatically shared between computing nodes due to the features of the AWG-STAR system. The compiler replaces all references to the variables in the source code with the references of pointer variables, and sets the address of the memory area allocated on the shared memory to that of the pointer. Figure 9 shows an example of transformed variable references. The memory areas for the shared global variables are statically allocated at the start of the program. However, the memory areas for the shared local variables are dynamically allocated before the parallel-execution region has started execution, and released after execution has finished. Therefore, a mechanism to manage the dynamic allocation of memory is required.

We utilized a rule of our OpenMP compiler that frees the variables in reverse sequential order of allocation to implement this dynamic memory. That is, we reserved enough memory area in advance, and allocated it from the top. This implementation made it easy to control the allocation and de-allocation of memory because we could use the shared memory as a stack.

We had to hold the value that showed the last address of the allocated memory in adopting this stack structure. To achieve this, we used a rule of the OpenMP compiler where only the master requests the shared memory to be allocated or freed.

Local Variables		
double x;	⟶	double (*x);
int y[10];	⟶	int (*y)[10];
Global Variables		
double x;	⟶	double (*__G_x);
int y[10];	⟶	int (*__G_y)[10];

Fig. 9. Example of Variable References Transformed by OpenMP Compiler

The master increases or decreases the value of the tail address of the stack when functions for allocation or de-allocation are called in a program.

C. Synchronization Primitives

Shared data in a parallel program that is read and written by multiple-computing processes during parallel processing must be access-controlled by locks to prevent inconsistencies. A function for barrier synchronization is also required to guarantee the order of execution of dependent processing. It is necessary to provide these functions to achieve OpenMP. These are called synchronization primitives because they are not provided by the AWG-STAR system.

We therefore implemented a synchronization primitive on the AWG-STAR system. Synchronization between computing processes in the OpenMP program was achieved by calling our synchronization primitives from the intermediate code.

1) Lock Control Function: Programmers have to use a lock (exclusive) control function to maintain coherency in the shared variables. The critical sections in an OpenMP program are determined by OpenMP directives, and then exclusively controlled. Programmers label these critical sections to distinguish between them if there are more than one in a program. Our OpenMP compiler converts the labels into positive integers $(1, 2, \cdots)$ in intermediate code. We called the integers lock numbers, and distinguished between critical sections by using these.

One method of implementing the lock-control function was by preparing an index of lock status in the shared memory. That is, "unlocked" or "process i locked" was shown in the index in order of lock number. When process i was to lock a section, it first confirmed that the status of the lock number was "unlocked", and it next updated the status to "process i locked", which meant process i had locked the section.

This method has a drawback, however, because the lock steps in critical sections are not atomic and more than one process may enter one critical section at the same time. We adopted the master-worker approach to control the index that is in the shared memory to avoid this problem. Worker process i first confirms that the status of the lock number is "unlocked" to lock a section, and then requests the master process to lock it. When the master process receives the request, it reconfirms the index and changes the status to "process i locked". Last, the worker process obtains a locking acknowledgment from the master process. Up to one process can execute a section because processes enter a critical section in order of arrival of the requests.

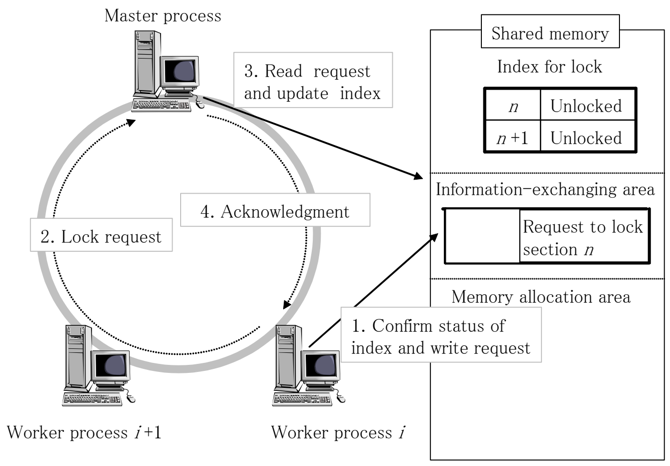

Fig. 10. Lock-control Function

We needed a function to transmit requests between master and worker processes to adopt this approach. We then utilized a signal function that the AWG-STAR system offers and part of the shared memory as an information-exchange area between processes. When worker process i makes a request to the master process, it writes the request to the ith information-exchange area and sends a signal to the master. As the master process receives the signal and notices that there is a request from process i, it reads the request from the ith area. Acknowledgments from the master to the worker are done in reverse order. This function is outlined in Fig. 10.

Processes leaving a critical section have to unlock it. They execute the same action as that of the lock to do this. That is, a worker process requests the master process to unlock the critical section, and the master updates the index in the shared memory.

To implement these, we prepared three threads in the master process, i.e., the "main thread", which executes the OpenMP application program, the "control thread", which controls the index in the shared memory, and the "signal-waiting thread", which receives signals from the worker processes. We adopted this approach because these three threads have to be computed in parallel. As the process generates the control thread when it starts, it is divided into the main and control threads. The signal-waiting thread is generated from the control thread. We will now describe the operation of the control and the signal-waiting threads (see Fig. 11).

The control and signal-waiting threads share information on the number of the process that sent a signal to the master process. The process number is stored in the variable `RequestProcess`. The signal-waiting thread constantly awaits signals from worker processes and updates the value of `RequestProcess` when it receives them. When the control thread notices that the value of `RequestProcess` has been updated, it reads data from the information-exchange area of that process. Data written in the information-exchange area of one process consists of two 32-bit data. The first 32 bits contains the kind of request. The written data is 0x0000FF00 if a lock is requested, and 0x0000FFF0 if an unlock is requested. The second 32 bits in the information-exchange area contains the lock number that the process wants to lock/unlock.

The control thread confirms the index where the request is to secure lock-number n from process i. The actual implementation of the index is an array of integers. Lock number n is unlocked when the nth integer is -1, and when it is an integer larger than zero, the process number for that figure is locked. Therefore, if the nth integer of the array is -1, it is changed to the number for i, which means the section is locked by process i, and the control thread sends a signal to that process. If critical-section n is not unlocked, the request is enqueued to the waiting list. This queue is a one-way list for each lock number, and one element on the lists includes the process number and the pointer for the next element. The head pointer of the list points to NULL if no process is waiting to be locked.

The control thread confirms from the queue of lock-number n whether there are any processes waiting to be locked where the request is to unlock lock-number n from process i. The control thread changes the index to "unlocked" if no processes are waiting. If there are, it enqueues the first process and changes the index to the number of that process. Last, it sends a signal to the process that is waiting for it.

It is too time consuming to send signals to the master process itself like other worker processes do when the main thread of the master process executing an OpenMP application wants to lock and unlock critical sections. To avoid this, the main-thread directory calls the series of functions described above except for sending signals when the master process locks/unlocks the critical section.

2) Method of Barrier Synchronization: Barrier are synchronized when a process needs to wait until all the processes reach the same break point. As previously described, this synchronization is not only called when the programmer writes it into the program, but it is also automatically inserted into the intermediate code.

We adopted the master-worker approach for synchronizing barriers as well as the lock function. When a process arrives at a barrier, it writes a notice of arrival to the information-exchange area, sends a signal to the master process, and enters the waiting state. After the master process receives signals from all worker processes, it sends signals back to them to release them from the waiting state. The worker processes that receive these signals resume execution.

We also implemented a method of controlling barrier synchronization in the control thread (see also Fig. 11) of the master process. When the control thread detects an update with a value of `RequestProcess`, and the data written in the information-exchange area is 0xFF000000, this means that the process that sent the signal has arrived at the barrier. The control thread manages the process that has arrived at the barrier with a local variable. If the variable indicates that all processes have arrived at the barrier, the control thread sends signals to all the worker processes. If there are processes that are not yet at the barrier, it only updates the variable. It is too time consuming to send a signal to the master process itself as well as the lock-control function. We therefore prepared a barrier flag to share information between the main and control threads. When the main thread executing an OpenMP application arrived at the barrier, it directly performed the series of functions previously described. If the master process was not the last process to arrive, it set the flag to true, which was changed to false when all the processes arrived.

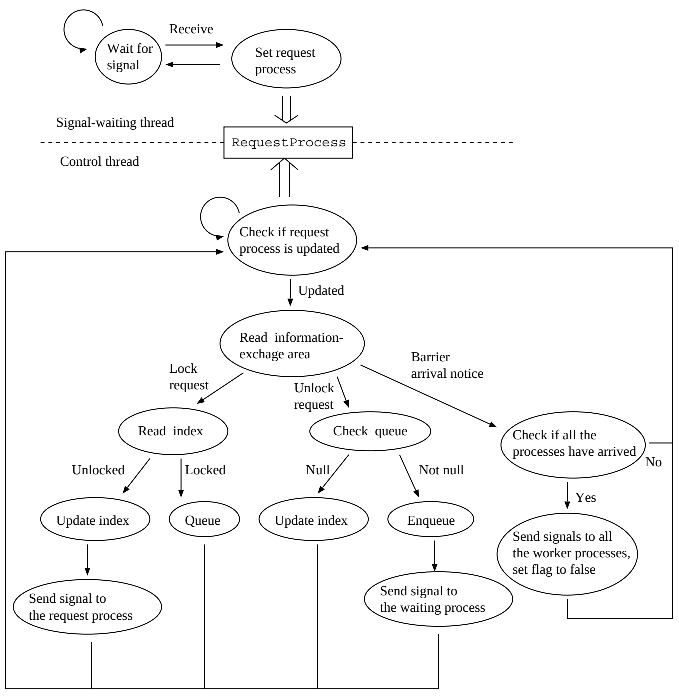

Fig. 11. Scheme for Lock Control and Barrier Synchronization

IV. Evaluation of Performance

We evaluated the performance of our OpenMP implementation in the λ computing environment by executing benchmark programs and the OpenMP application.

A. Environment for Evaluation

Table III lists the specifications for the computing nodes we used for evaluation. We use from one computing nodes to four ones, and all nodes have the same specifications. We changed the length of the optical ring depending on the number of computing nodes. That is, if we let the number of computing nodes be N, the length of the optical ring is $10N$m. We executed one OpenMP process in one computing node. This is because the AWG-STAR system does not enable multiple processes to be executed in a computing node.

We used cluster middleware called SCore [11] for comparison, which can accomplish parallel computing with OpenMP in an Ethernet environment. SCore offers an SDSM system called SCASH, achieving a shared memory virtualy in the distributed memory environment, and a dedicated OpenMP

TABLE III
SPECIFICATIONS FOR COMPUTING NODES

CPU	Xeon 3.06 GHz
Main memory	512 MB
Level-one cache	20 KB
Level-two cache	512 KB
NIC	Intel PRO/1000
PCI bus	64 bit/66 MHz
PCI transmission speed	533 MBytes/sec
OS	Redhat Linux 7.3
Compiler	gcc 2.96

TABLE IV
EXECUTION TIME FOR NPB 2.3 BT CLASS W

Environment	# of Nodes	Execution Time (s)	Speed-up Ratio
AWG-STAR	1	47930	1.00
	2	27322	1.75
	4	17172	2.79
SCore	1	25	1.00
	2	357	0.07
	4	430	0.06

compiler called Omni/SCASH [10]. We used a 1-Gbps Ethernet and each Ethernet cable was 10-m long.

B. Evaluation with Benchmarks

We selected BT and EP benchmarks from the NAS Parallel Benchmarks (NPB) 2.3 [12] and the Himeno benchmark [13] for our evaluation. We used the OpenMP C version of NPB 2.3 [14] which is ported by RWCP (Real World Computing Project). The size of the problem is XS ($128 \times 64 \times 64$) on the Himeno benchmark and Class W (BT: $24 \times 24 \times 24$ B! (BEP: 2^{25}) on the NPB 2.3. The frequency with which NPB 2.3 EP accesses the shared data is low, but the Himeno benchmark and NPB 2.3 BT access it relatively frequently.

Table IV and Fig. 12 show the results for the BT and EP of the NPB 2.3 and Table V shows the results for the Himeno

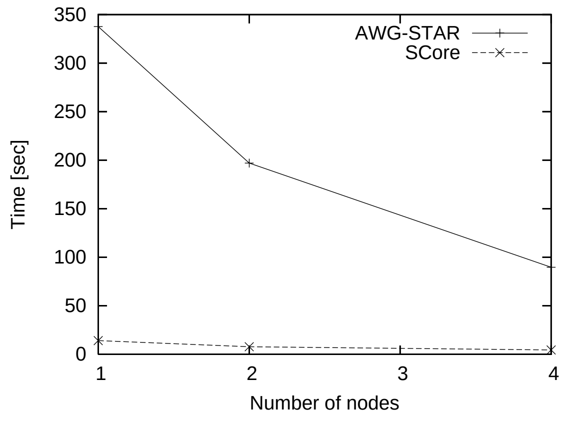

Fig. 12. Execution Time for NPB 2.3 EP Class W

TABLE V
RESULTS FOR HIMENO BENCHMARK

Environment	# of Nodes	MFLOPS	Speed-up Ratio
AWG-STAR	1	0.1074	1.000
	2	0.2073	1.930
	4	0.3760	3.501
SCore	1	393.9016	1.000
	2	5.8346	0.015
	4	4.0228	0.010

benchmark. Table IV and Fig. 12 show the execution time for the benchmark program, and Table V lists the performance in MFLOPS. SCore outperforms the AWG-STAR system, which did not achieve sufficiently high performance, in these tables and the figure. However, there was a tendency for performance to improve as the number of computing nodes increased. The problem with access to the shared memory may be why the AWG-STAR system performed poorly.

SCASH as a SDSM system attains a shared memory virtualy by exchanging the content of local memory between computing nodes via the network. SCASH acquires the latest data by communicating with computing nodes and copies these to the local memory if the data has been updated by other computing nodes when they access data on the shared memory. SCASH maintains the consistency of its data between computing nodes in this way. Although it takes a long time to acquire data, access to data that has already been acquired is fast.

On the other hand, the hardware keeps the data written in the shared memory of the AWG-STAR system the same. However, as we have to access this shared memory via the PCI bus from the CPU, we cannot read or write sufficiently fast with the current AWG-STAR system. Although access from the CPU to the local memory is possible at about 2 GB/s, the maximum access speed to the shared memory is only about 80 MB/s. All access to the shared data is slow since all shared data are read and written against the memory on the SMB.

Therefore, memory is a bottleneck in the AWG-STAR system and may affect general computing. The difference in performance in the experimental results with SCore was about 20-fold with four nodes in the NPB 2.3 EP, but it was about 40-fold in the NPB 2.3 BT. The difference in performance with SCore in the benchmark with high-access frequency to the shared data was greater than the difference in the benchmark with low-access frequency. The difference in performance with the Himeno benchmark was smaller than the others because of its wide range of memory access. Parallel programs on SDSM without data locality generally perform poorly because of frequent node-to-node communications.

C. Evaluation with OpenMP Application

We utilized a Mandelbrot set as a parallel-processing application and a GUI window to check the progress of executions.

1) Calculation of Mandelbrot set: The Mandelbrot set is defined as a set of all points of complex parameter c such that the sequence, $z_0 = 0$, $z_{n+1} = z_n^2 - c$ $(n = 0, 1, \cdots)$, does not escape to infinity. Mathematically, the Mandelbrot set is

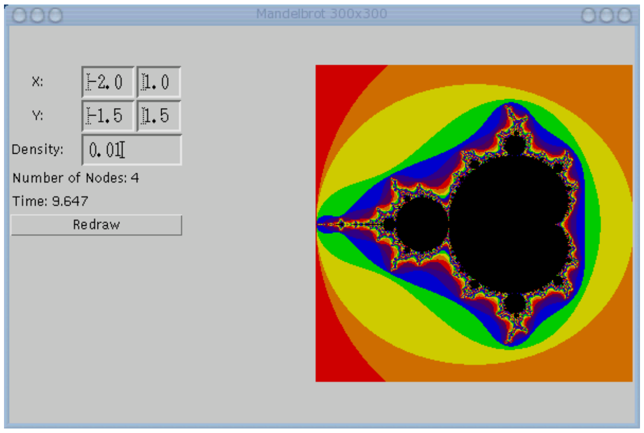

Fig. 13. Window Displaying Image of Mandelbrot Set

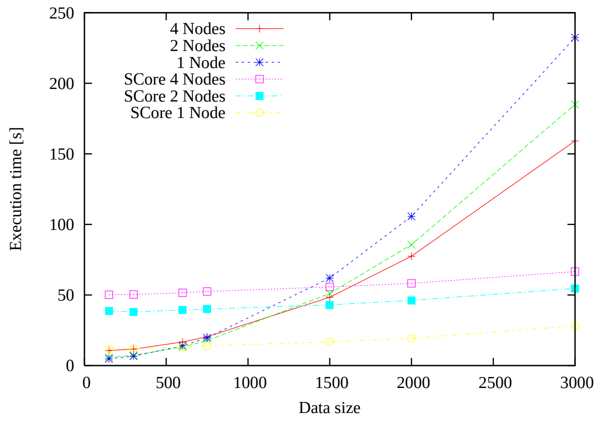

Fig. 15. Execution Time in AWG-STAR System and SCore

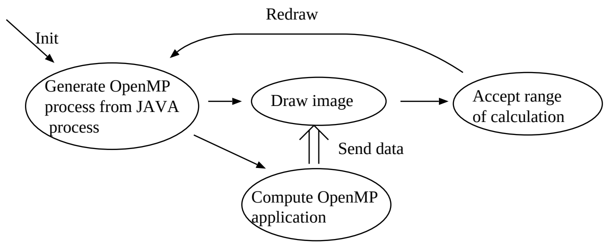
Init
Redraw
Generate OpenMP process from JAVA process
Draw image
Accept range of calculation
Send data
Compute OpenMP application

Fig. 14. State Transition Diagram for Drawing Mandelbrot-set Image

just a set of complex numbers. A given complex number, c, either belongs to M or it does not. A picture of the Mandelbrot set can be made by coloring all the points, c, that belong to M black, and all the other points white. The more colorful pictures that are usually seen are generated by coloring points not in the set according to how quickly or slowly the sequence, $|f_c^n(0)|$, diverges to infinity.

Simple parallelizing of the Mandelbrot set is done by dividing the complex plane into the number of processes, and each process calculates the area. Each computation is independent and only the calculation results at the end need to be collected. This means that the application is suitable for parallel computation as well as the λ computing environment because the overhead for data sharing/transmission is slight.

Figure 13 shows a window displaying the image of a Mandelbrot-set zoom sequence. Labels X and Y indicate the range of calculation (maximum and minimum), and the label density indicates the difference between each complex number. The image is redrawn by changing the number of the label and pressing the "Redraw" button. The window also shows the computation time.

Figure 14 is a state transition diagram of the application for drawing the Mandelbrot-set image. When the range of calculation is set on the Java GUI, it generates OpenMP processes that execute the Mandelbrot set. They calculate the Mandelbrot set and send the results to the Java process through a socket. The Java process draws the image after the results have been received to show the progress of calculation.

The Java process generates both master and worker OpenMP processes (see Fig. 8). The master OpenMP process starts sequential execution and the worker processes immediately call barrier synchronization. When the master process reaches the break point of the barrier method, which means that sequential execution has finished, all the processes start successive parallel executions. The Mandelbrot set is executed in the parallel executions, and barrier synchronization is also called at the end of the parallel executions. After that, the master process returns to sequential execution and sends the result to the socket.

2) Performance Evaluation with Execution Time: Figure 15 plots the execution time to calculate the Mandelbrot set in the AWG-STAR system and SCore. We used the total execution time of applications for the evaluation unlike the benchmark results in Section IV-B. We let the data size of the complex-number set to be calculated be $x \times y$, and executed it when the size was $x = y$. The horizontal axis of the figure indicates the value of x.

The execution time is shorter with few processes in the AWG-STAR system when there is a small amount of data. However, as the amount of data increases, the execution time shortens with many processes. This is a characteristic result with parallel computing. The time for an initial setting takes from 10 to 50 sec according to the number of computing nodes in SCore. The execution time not involving the initial setting is shorter than that for the AWG-STAR system, and there is little difference between the number of computing nodes.

3) Evaluation of Performance with CPU Usage: Figures 16 and 17 show the CPU usage during calculation with the four computing nodes. As these measurements are on a time scale of seconds, there are time lags between the computing nodes. The first 7 sec are spent in sequential execution by the master process. The master process during this time writes the array into the shared memory to initialize the array that stores the calculation results. Parallel execution by all the processes, including the worker processes, starts after the first sequential execution. They compute the Mandelbrot set in the parallel execution, and worker processes account for nearly

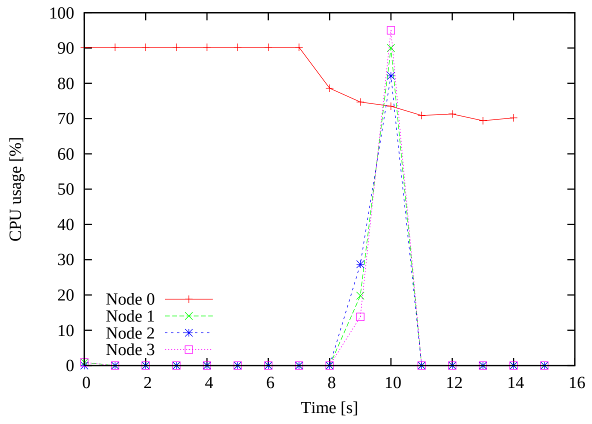

Fig. 16. Data Size 600x600

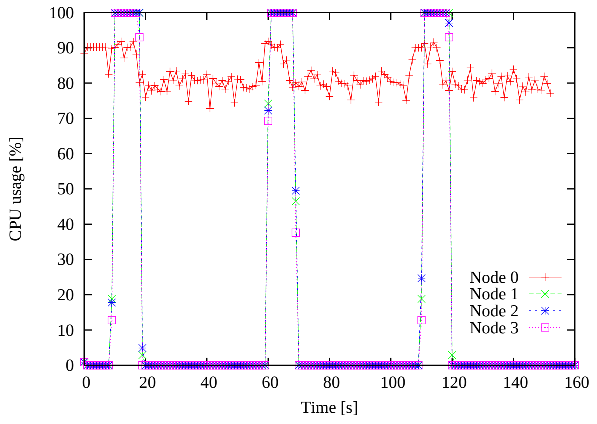

Fig. 17. Data Size 3000x3000

100% of CPU usage. The CPU usage by the main process remains at around 90%, which is not only that for the OpenMP process but also that for the Java process. The threads for the synchronization primitives are running in that node.

When the size of the data is 3000×3000, the calculation is divided into multiple computations because we set the size of the array to 1000×3000 in the program. The CPU usage for this is shown in Fig. 17. We can see that sequential execution takes a long time. The master process reads the calculation results from the shared memory during the sequential execution and sends them to the socket. The Java process receives the results from the master process of the OpenMP application, and draws an image of the Mandelbrot set. The overhead incurred for this series of processes is thought to be low access speed to the shared memory. The observed speed to read data from the shared memory is less than 1 Mbps. However, because the SCore utilizes software-distributed shared memory, there is no difference between the speed to read from the shared memory and that of local memory. This decreases the performance overhead, and the delay to treat a large volume of data remains short. The memory read time ratio to the execution time is 87% – 97% in these cases, so if the access speed to the shared memory will be improved, the execution time for applications is expected to become shorter.

V. Conclusion

We proposed a new architecture for distributed parallel-computing environments in this paper, the λ computing environment, which utilizes optical wavelength paths to interconnect shared-memory systems. We established the λ computing environment with the AWG-STAR system, and implemented the OpenMP shared-memory parallel programming standard on it. We found our environment was not able to achieve sufficiently high performance by evaluating it against benchmark programs and applying the Mandelbrot set to it. This is because the access speed for the memory of the AWG-STAR system was too slow.

NTT are currently developing the next version of the AWG-STAR system to improve its method of memory access to reach that of a shared memory. As we should then be able to access with almost the same speed as a local memory, we should be able to achieve better performance.

Acknowledgment

We would like to thank Dr. Akira Okada at NTT Photonics Laboratory for his helpful suggestions and support.

References

[1] E. L. Berger, “Generalized multi-protocol label switching (GMPLS) signaling functional description,” *IETF RFC3471*, Jan. 2003.

[2] H. Nakamoto, K. Baba, and M. Murata, “Shared memory access method for a λ computing environment,” in *Proc. of IFIP OpNeTec*, pp. 210–217, Oct. 2004.

[3] H. Nakamoto, K. Baba, and M. Murata, “Proposal and evaluation of realization approach for a shared memory system in λ computing environment,” in *Proc. of COIN2005*, pp. 90–95, May 2005.

[4] E. Taniguchi, K. Baba, and M. Murata, “Implementation and evaluation of shared memory system for establishing λ computing environment,” in *Proc. of OECC2005*, 5A2-3, pp. 20–21, July 2005.

[5] E. Taniguchi, “Design and evaluation of shared memory architecture for WDM-based λ computing environmnet,” Master’s thesis, Graduate School of Info. Sci. and Tech., Osaka University, Feb. 2006.

[6] M. Imoto, E. Taniguchi, K. Baba, and M. Murata, “Implementation and evaluation of MPI library with Globus Toolkit for establishing λ computing environment,” in *Proc. of IEICE APSITT2005*, pp. 421–426, Nov. 2005.

[7] OpenMP Architecture Review Board, “OpenMP Application Program Interface Version 2.5,” 2005.

[8] Y. Sakai, K. Noguchi, R. Yoshimura, T. Sakamoto, A. Okada, and M. Matsuoka, “Management system for full-mesh WDM AWG–STAR network,” in *Proc. of ECOC2001*, vol. 3, pp. 264–265, Sept. 2001.

[9] V. Dimakopoulos, E. Leontiadis, and G. Tzoumas, “A portable C compiler for OpenMP V. 2.0,” in *Proc. of the European Workshop on OpenMP (EWOMP03)*, Sept. 2003.

[10] M. Sato, H. Harada, and A. Hasegawa, “Cluster-enabled OpenMP: An openmp compiler for the SCASH software distributed shared memory system,” *Scientific Programming*, vol. 9, no. 2, pp. 123–130, 2001.

[11] “PC Cluster Consortium,” available at http://www.pccluster.org/.

[12] “NAS Parallel Benchmarks,” available at http://www.nas.nasa.gov/Resources/Software/npb.html.

[13] “Himeno Benchmark,” available at http://accc.riken.jp/HPC/HimenoBMT/.

[14] “OpenMP C versions of NPB2.3,” available at http://phase.hpcc.jp/Omni/benchmarks/NPB/.