# Master's Thesis

Title

# Design and Implementation of Synchronization Primitives for $\lambda$ Computing Environment

Supervisor

Professor Masayuki Murata

Author

Mai Imoto

February 14th, 2007

Department of Information Networking
Graduate School of Information Science and Technology
Osaka University

Master’s Thesis

**Design and Implementation of Synchronization Primitives for $\lambda$ Computing Environment**

Mai Imoto

## Abstract

Grid computing technology, which enables large scale computing has been studied and developed by numerous researchers in recent years. As we usually treat large-volume data in grid computing environment, we need the technology that enables the high speed and large scale transmission in a network. In conventional TCP/IP, however, it is difficult to achieve good performance because of overhead caused by packet processing and retransmission of lost packets.

In this thesis, we propose a new computing environment (which we refer to $\lambda$ computing environment) that provides a infrastructure for parallel computing among computing nodes distributed in the wide area. Network switches and computing nodes are connected each other with optical fibers in the $\lambda$ computing environment, thereby offering high-speed and reliable connection pipe among end computing nodes. Moreover, optical fibers are directly connected to each memory of computing nodes, so that a shared memory is constituted between the computing nodes. We propose to utilize this shared memory in the $\lambda$ computing environment for the parallel computation and expect that high-speed parallel computation in a wide-area distributed system can be achieved.

In this thesis, we establish the $\lambda$ computing environment using the AWG-STAR system which has a shared memory in hardware and aim to execute OpenMP application, the de-facto standard of the shared memory parallel computing. To execute parallel computation, we design and implement synchronization primitives and data sharing structure of the AWG-STAR system. Also, we implement an OpenMP application that graphically shows the calculation, and evaluate the system with it. The results show the advantage of parallel computation. However we cannot achieve the sufficient performance of application, because the speed to read data from the shared memory is very slow. This makes the performance of application low.

To resolve the bottleneck of the AWG-STAR system, a next version of the AWG-STAR system is begin developed. The next version differs from the current version in the control of the signals.

Thus we design synchronization primitives for the next version of the AWG-STAR system. Additionally, we propose a virtualization of the shared memory to execute a large-scale computation required larger memory than the available physical shared memory of the AWG-STAR system.

**Keywords**

$\lambda$ computing environment, Shared memory architecture, AWG-STAR system, Distributed parallel computing, OpenMP

## Contents

## List of Figures

## List of Tables

## 1 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. For example, one of the important issues to realize the grid computing is how to share and cooperate geographically/systematically distributed resources. To solve that problem, Globus Alliance has been developing Globus Toolkit, which is nowadays de-fact standard of the grid computing middleware. At the same time, researches into the high-speed data transmission have been also pursued. 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 existence of 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.

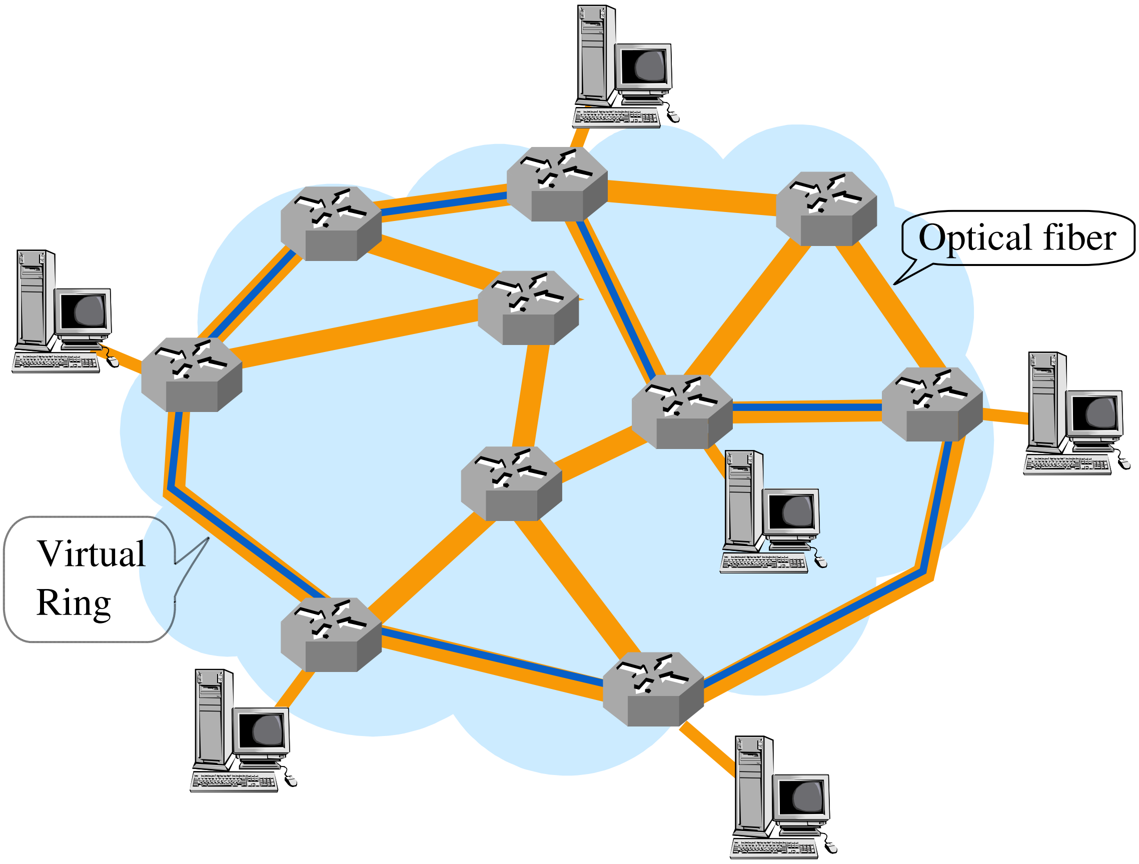


Figure 1: $\lambda$ computing environment

We thus propose a new architecture that we call the $\lambda$ 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 $\lambda$ computing environment because computing nodes do not utilize the conventional TCP/IP network but establish wavelength paths as a communications channel in advance (See Figure 1).

Related work [2-5] has reported the evaluation of architecture that has accomplished distributed-parallel computing in a $\lambda$ 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 $\lambda$ 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 $\lambda$ 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, 9] as an instance of the $\lambda$ 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 remainder of this paper is organized as follows. Section 2 explains the $\lambda$ computing environment and the parallel computing environment that we establish in this study. This is followed in Section 3 by the design and implementation of the synchronization primitives which are needed to execute OpenMP applications. The proposal and implementation is evaluated by an application are given in Section 4. We design the synchronization primitives for the new version of the AWG-STAR system in Section 5, and in Section 6 gives a conclusion and outlook on future work.

## 2 $\lambda$ Computing Environment: New Distributed Computing Environment

We will first explain the $\lambda$ 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 $\lambda$ computing environment are executed.

### 2.1 WDM technology for $\lambda$ Computing Environment

The $\lambda$ 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 $\lambda$ 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 Figure 2.

### 2.2 AWG-STAR System

#### 2.2.1 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 Figure 3). The AWG router processes optical signals without transforming them into electrical ones, which provides high-speed transmission. All nodes are equipped with a shared memory board (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

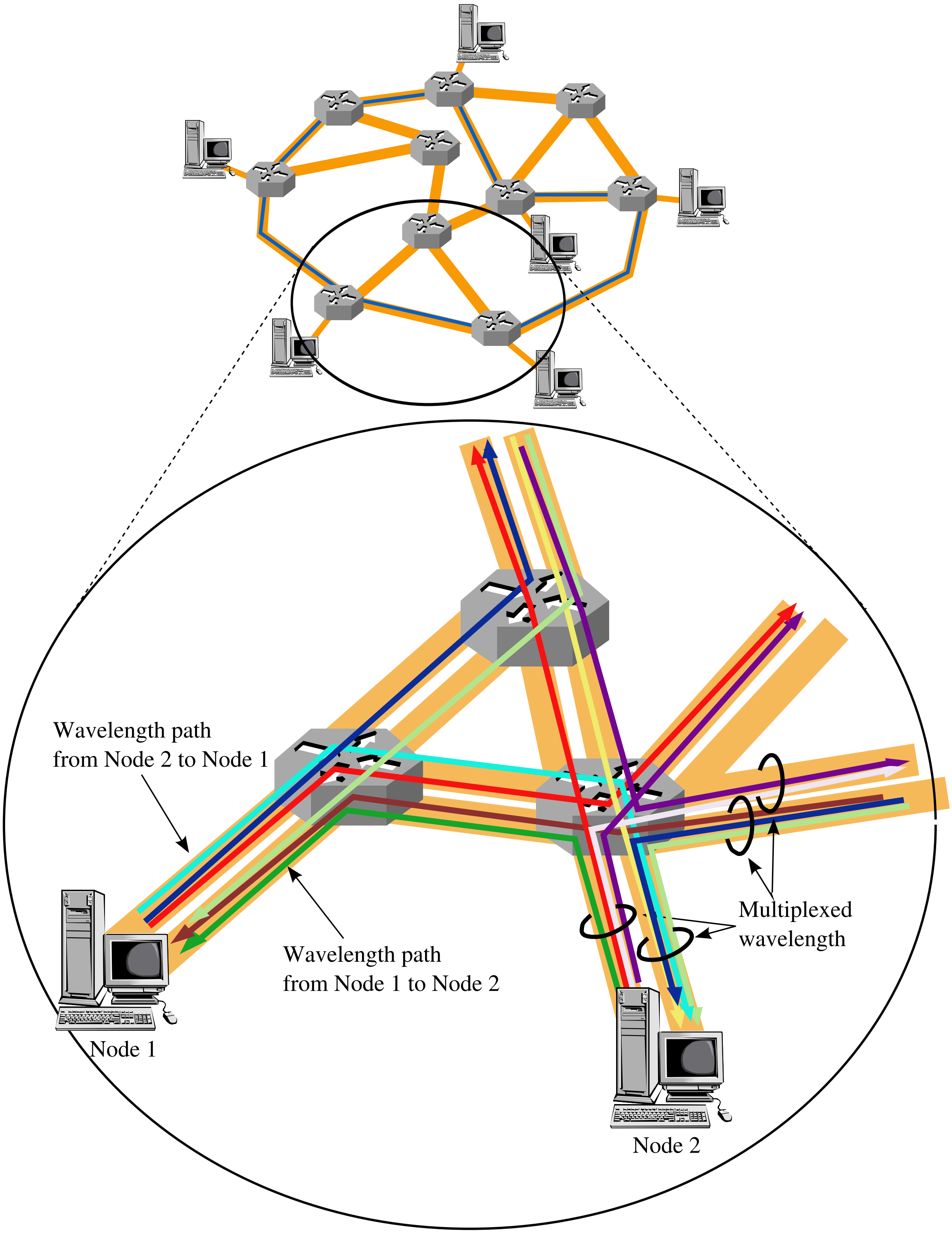


Figure 2: Established wavelength paths

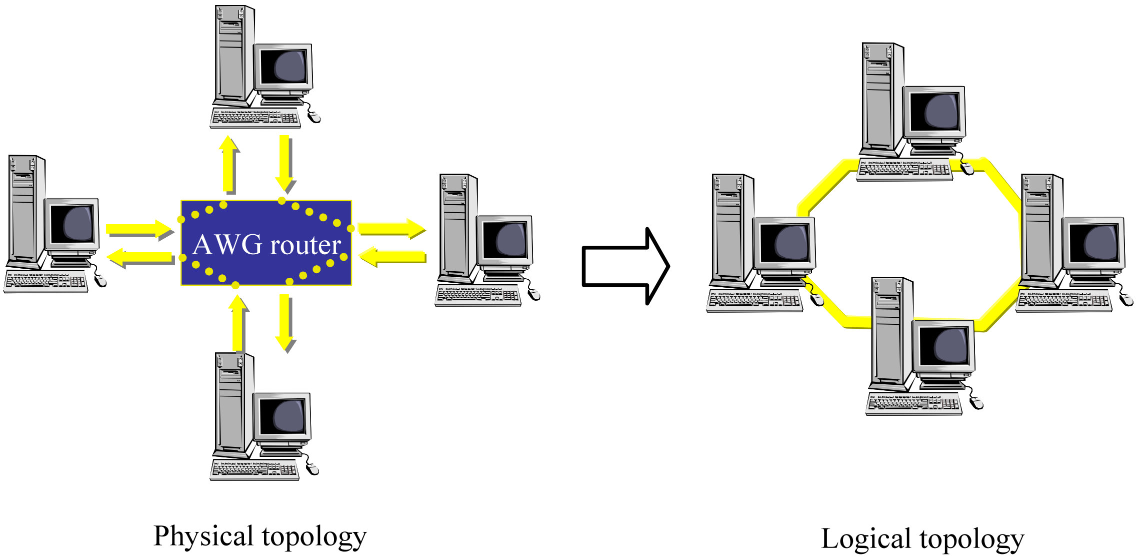


Figure 3: Network topology in AWG-STAR system

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.

### 2.2.2 Configuration of AWG-STAR system

We show the outline drawing of the AWG-STAR system in Figure 4. In this system, each computing node is connected to the AWG router through transponder with tunable lasers, and configure a star topology in physical but does a ring in logical. The system uses multi-mode fiber. Each computing node is equipped with a SMB containing shared memory.

The AWG router can configure a wavelength path by dynamically changing the wavelength of the optical signal. It has 32 input ports and 32 output ports, and which output port each incoming signal uses is determined by the wavelength. Table 1 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.

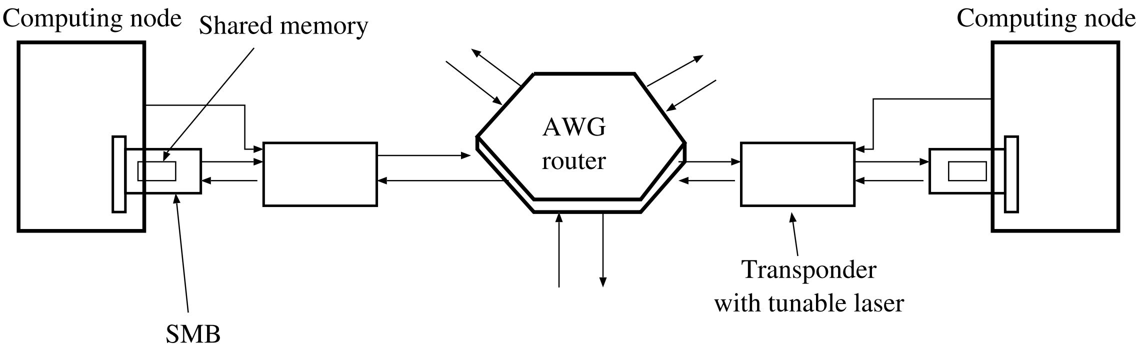


Figure 4: Configuration of AWG-STAR system

Table 1: Assignment of wavelength 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 |

### 2.2.3 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.

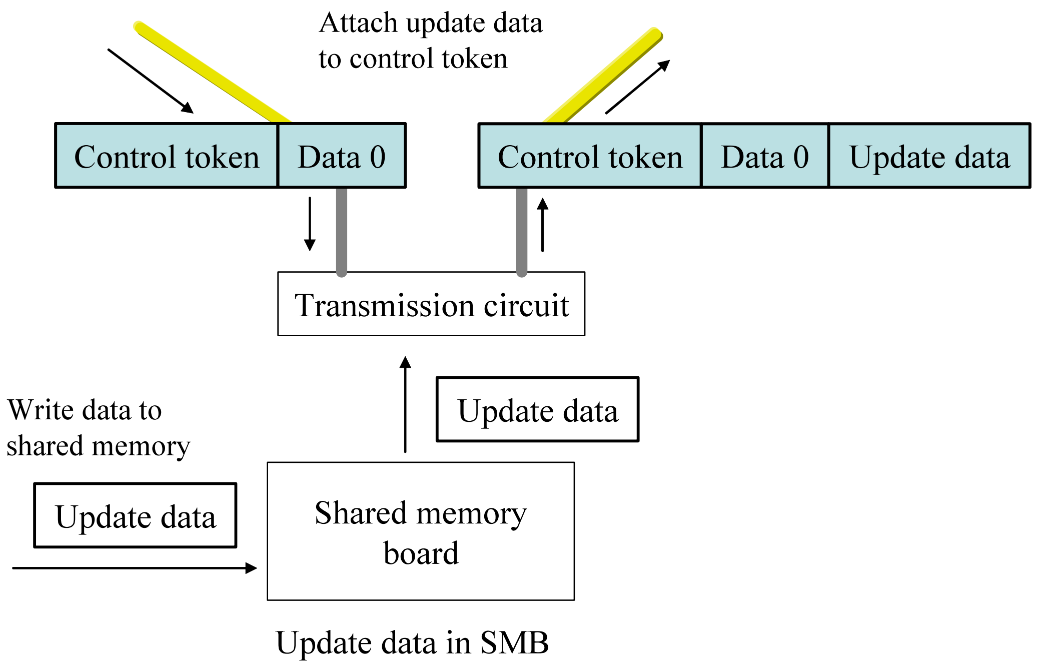


Figure 5: Write data to shared memory

### 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 5 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 6 shows the model for this.

Figure 7 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 SMBs. The data are deleted when they are passed around the ring.

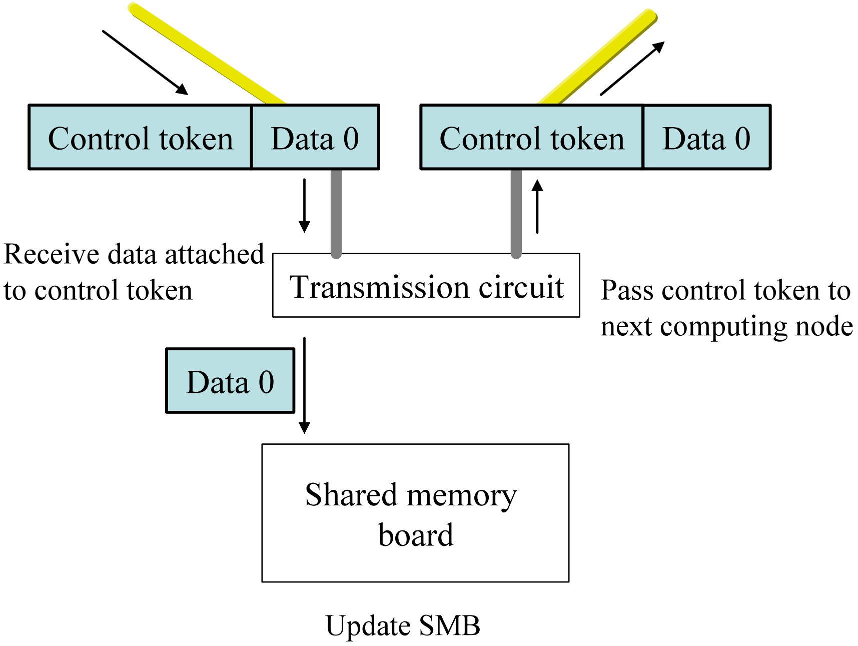


Figure 6: Receive data for updating

### 2.2.4 Access delay time for AWG-STAR system

In the AWG-STAR system, we have delay times to access a SMB and to share data with all nodes. 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 2 lists the specifications for the SMB and the access speed via the PCI bus. The data-sharing time consists of two factors: the first 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.

## 2.3 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

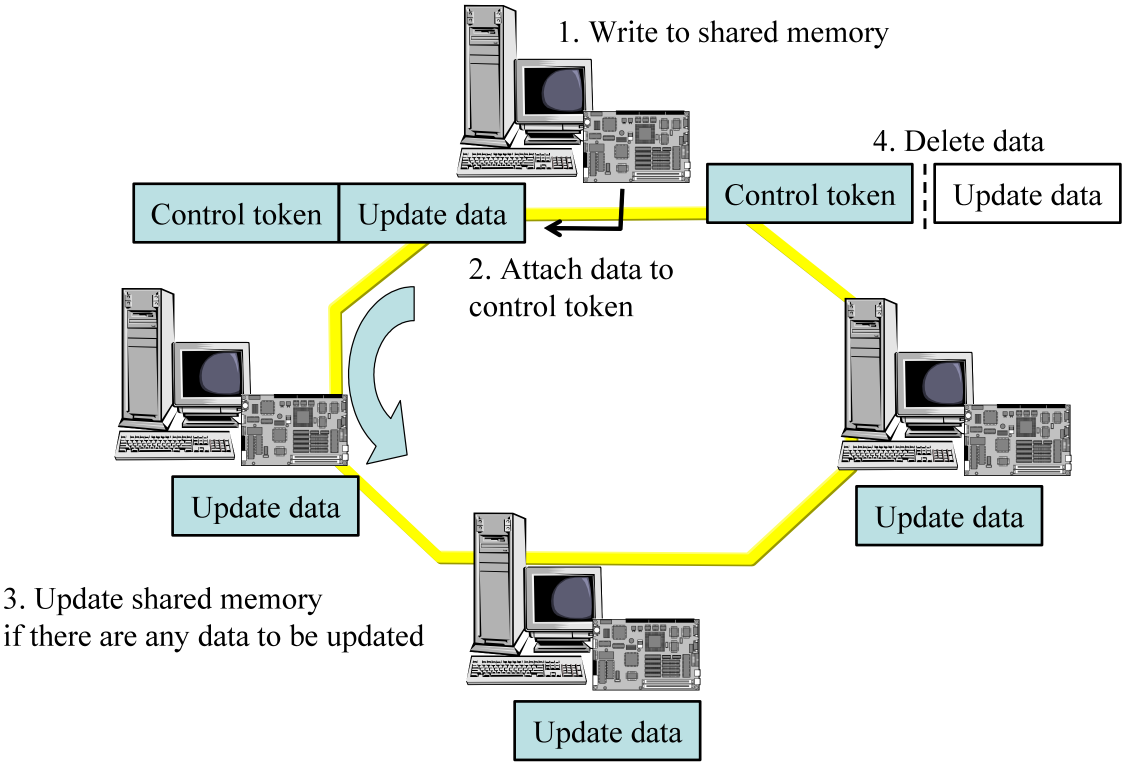

1. Write to shared memory
4. Delete data
Control token
Update data
Control token
Update data
2. Attach data to control token
Update data
Update data
3. Update shared memory if there are any data to be updated
Update data

Figure 7: Operation to update data

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 $\lambda$ computing environment [6]. Our aim was to execute the OpenMP application in the $\lambda$ computing environment in this research.

The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including UNIX platforms and Windows NT platforms. Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable and scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

Programmers using OpenMP add OpenMP directives to their programs to parallelize the appli-

Table 2: Specifications for SMB

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

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

Figure 8: Example of OpenMP source code

cation. Figure 8 shows the brief source code of the OpenMP program. `pragma` and the following statement on the second line is an OpenMP directive, and the successive `for` loop will be parallelized. That is, section without directives are executed sequentially by one process, and that with directives are executed with multiple processes. To execute as described above, OpenMP compiler generates intermediate code which is adapted for the computing environment, and intermediate code is compiled again into the machine-language program by the compiler of original programming language. We describe master process which executes the sequential execution, and do worker process which is other than the master process.

OpenMP compiler has a lazy consistency model for the shared memory, so that it can optimize the program such as transposing the memory access [10]. That is, all the processes may not have the same value for a shared variable on execution, but must do that when flush function is called or parallel execution is finished. This implementation leads programmers to utilize lock control function so as not to make inconsistent data.

### 2.4 Executing OpenMP in the AWG-STAR System

In this paper, we aim to execute OpenMP application in the AWG-STAR system. Utilizing the shared memory of the AWG-STAR system, there are two methods to write to the shared memory.

One method is to write directly when there is a write to shared variables. This method access shared memory every time, so it's not acceptable for the system where access delay time is large. Because it does not need to consider memory consistency, however, it is suitable for the AWG-STAR system in which all the computing nodes automatically get the same value for the shared variables. Another method to write to the shared memory is to buffer data in the local memory until flush function is called. This method decrease the number of accesses, so it is suitable for the system where the access delay time is large. But this method needs another consideration for the memory consistency between the computing nodes. In this paper, we adopt the former method to take advantage of the characteristic of the AWG-STAR.

# 3 Implementation of Synchronization Primitives

Figure 9 shows the protocol stack when we execute OpenMP applications. On executing in the AWG-STAR system, we need a compiler which generates intermediate code executable in the AWG-STAR system. The OpenMP compiler [11] implements parallel execution by multiple computing nodes and data sharing utilizing the AWG-STAR system. The OpenMP compiler also needs lock control function and barrier synchronization method, which are called synchronization primitives, and they are not provided by the AWG-STAR system. Additionally, we need a structure to check which area in the shared memory are already allocated or not. In this paper, we implement the methods which concern the functions of the AWG-STAR system, and provide them as a library to our OpenMP compiler [12] with hiding the inside implementation.

In this section, we describe the implementation of dynamic memory allocation, lock control function, and barrier synchronization method.

## 3.1 Dynamic Memory Allocation

A method for dynamic memory allocation is needed for the distributed parallel computing system. That is, when shared variables are declared in OpenMP program, the program has to know which address in shared memory it allocates them. This method also requires the function to free the shared variables which are not needed anymore. For example, a shared variable is declared in an OpenMP function. The compiler dynamically allocates the shared memory address for the variable when it executes the function, and it frees the address when the program goes out of the function. 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. 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. This implementation, however, does not inform worker processes about which shared variables are allocated where in the shred memory, but this communication is done in the OpenMP compiler.

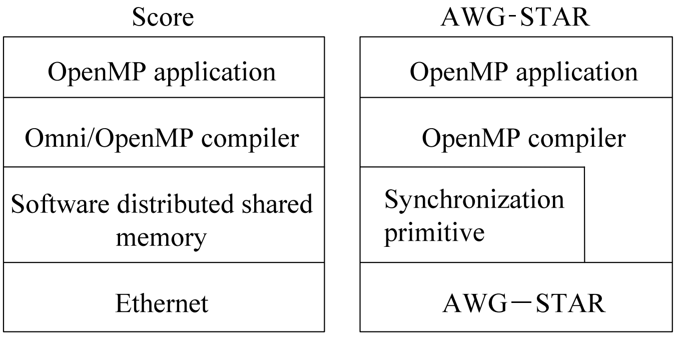


Figure 9: Protocol stack on executing OpenMP application

When we use this structure of memory allocation, the system cannot allocate larger area than it has reserved in advance. Then we propose additional design of shared memory virtualization and its memory allocation in Section 5.4.

### 3.2 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

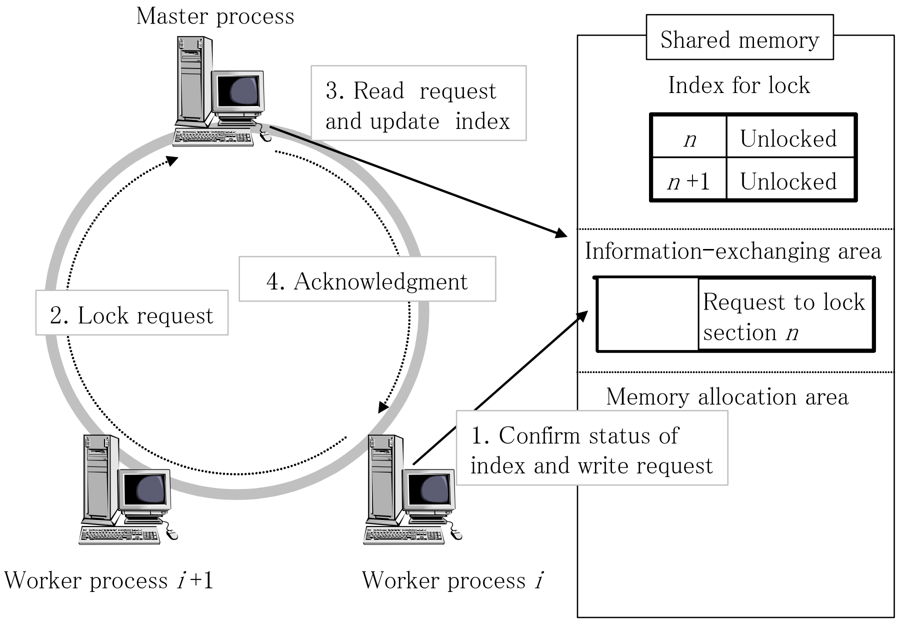


Figure 10: Lock-control function

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.

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 $i$th 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 $i$th area. Acknowledgments from the master to the worker are done in reverse order. This function is outlined in Figure 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 Figure 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 $n$th integer is $-1$, and when it is an integer larger than zero, the process number for that figure is locked. Therefore, if the $n$th 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 dequeues 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.

### 3.3 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 Figure 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.

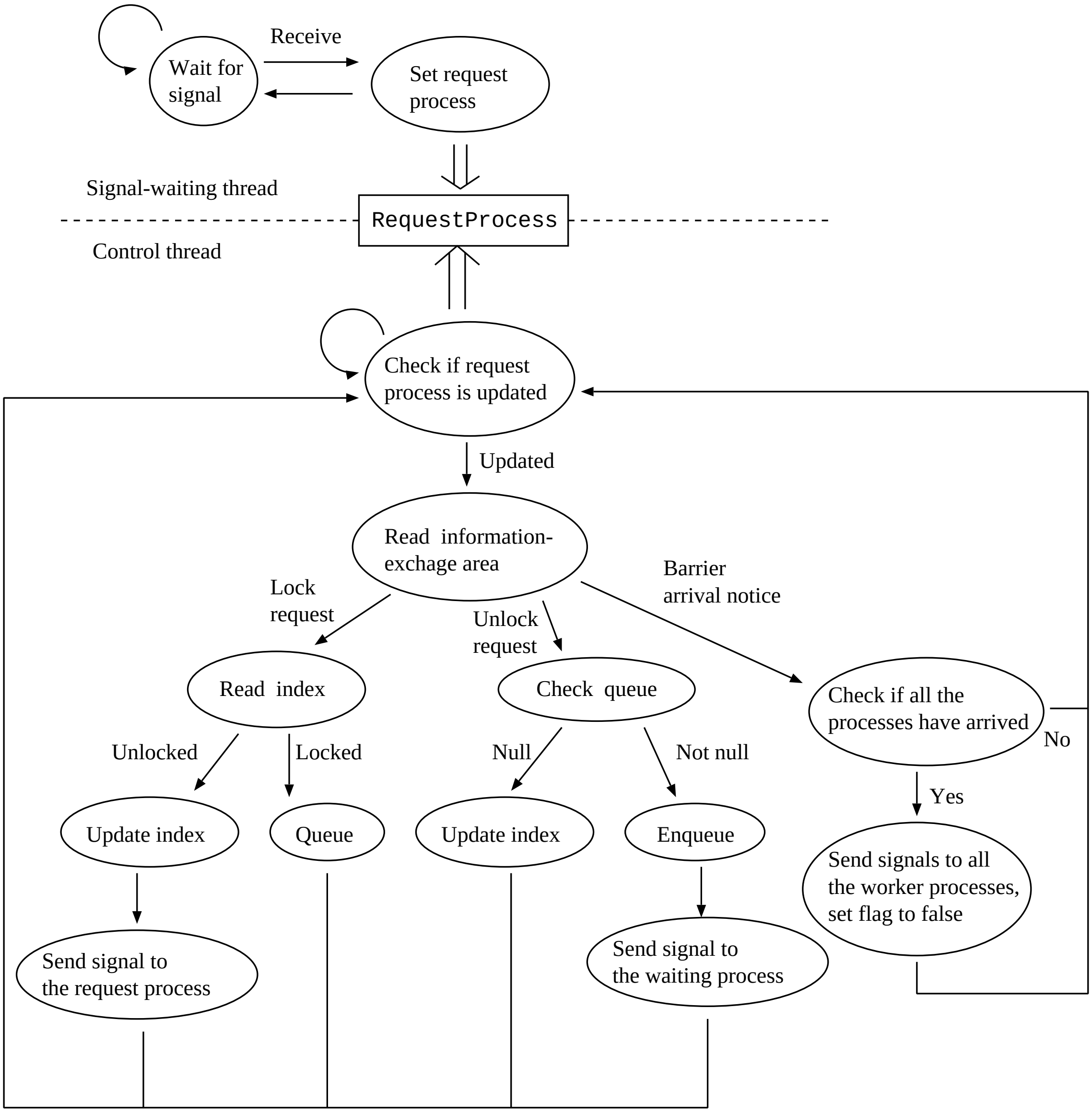


Figure 11: Scheme for lock control and barrier synchronization

## 4 Evaluation of Performance with OpenMP Application

In this section, we execute an OpenMP application and evaluate the performance. We utilize Mandelbrot set as a parallel processing OpenMP application and GUI window to see the progress of the execution.

### 4.1 Environment for Evaluation

Table 3 lists the specifications for the computing nodes we used for evaluation. We use from one computing node 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 $10N\mathrm{m}$. 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 [13] for a comparison, and the software distributed shared memory, SCASH [14], which utilizes Ethernet . In the case for the SCore, we use 1 Gbps Ethernet and the length of each Ethernet cable is 10 m long.

### 4.2 Application to Calculate 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 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 $\lambda$ computing environment because the overhead for data sharing/transmission is slight.

Table 3: Specifications for computing nodes

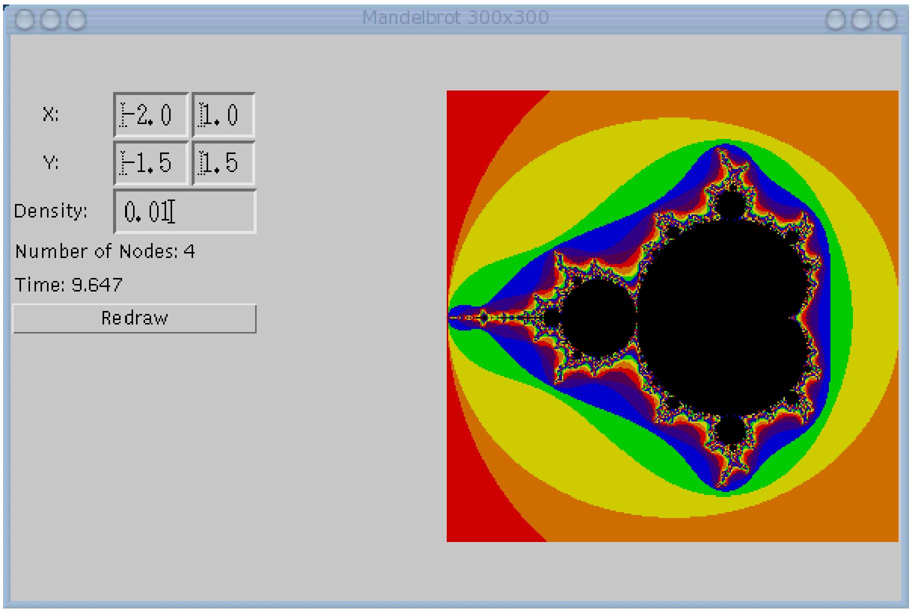
| | |
|---|---|
| CPU | Xeon 3.06 GHz |
| Main memory | 512MB |
| Level one cache | 20KB |
| Level two cache | 512KB |
| NIC | Intel PRO/1000 |
| PCI-bus | 64 bit/66MHz |
| PCI transmission speed | 533MBytes/sec |
| OS | Redhat Linux 7.3 |
| Compiler | gcc 2.96 |

### 4.2.1 Drawing computing result

Figure 12 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 13 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 performance of the JAVA and OpenMP processes are shown in Figure 14. The Java process generates both master and worker OpenMP processes. 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.



X: -2.0 1.0

Y: -1.5 1.5

Density: 0.01

Number of Nodes: 4

Time: 9.647

Figure 12: Window displaying image of Mandelbrot set

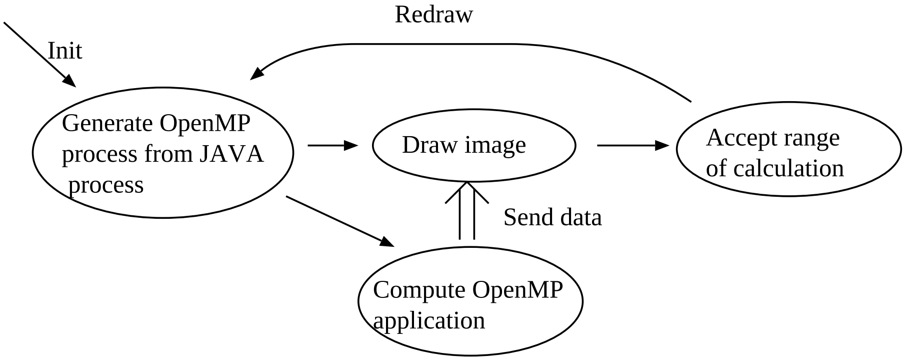

Redraw
Init
Generate OpenMP process from JAVA process
Draw image
Accept range of calculation
Send data
Compute OpenMP application

Figure 13: State transition diagram for drawing Mandelbrot set image

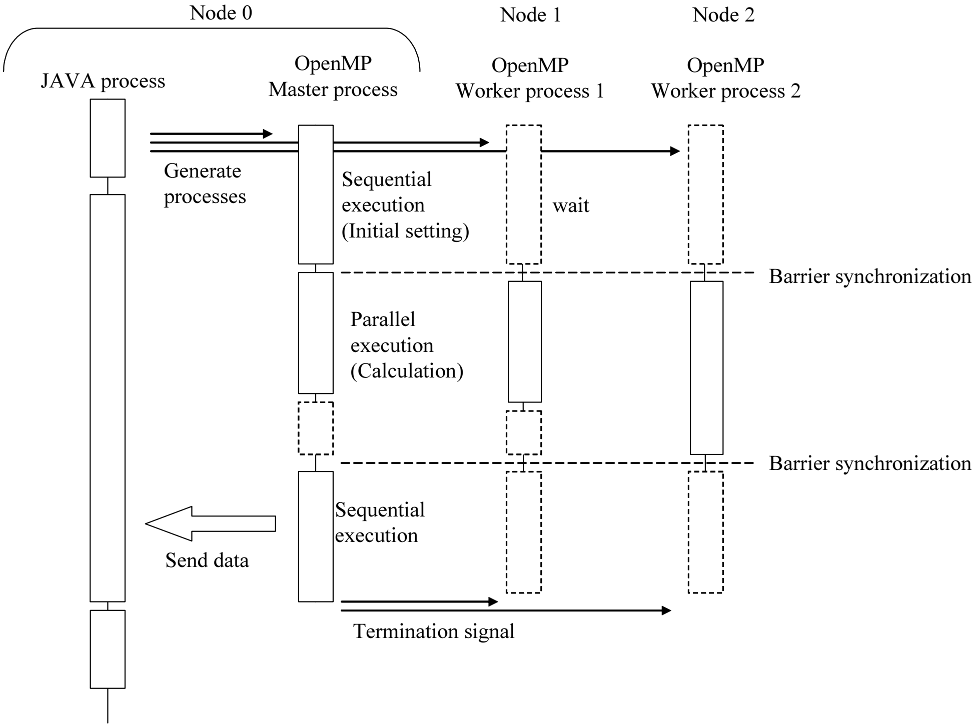


Figure 14: Performance of application processes

### 4.2.2 How to treat large data size

When the range of calculation is wide, or the density is high, the data size becomes large. In this case, two problems arise: one is the window displaying th image cannot fit within the display, and the other is the data size becomes larger than the size of memory. We describe the method to solve the problems in this subsection.

#### Thinning out the result data

We set the maximum data size to display 900×900, and if data size is larger than that, we thin out the result data to fit within the display. That is, when data size is smaller than 900×900, all the result are plotted, but when data size is larger than that, only equally-spaced result are plotted on the displaying window. For example, Figure 15 shows a part of the result of Mandelbrot set, whose whole result is larger than 900 × 900. We use only the encircled number for the drawing. The interval is changed according to the data size.

|  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| ④ | 5 | 5 | ⑤ | 5 | 6 | ⑥ | 7 |
| 5 | 5 | 5 | 5 | 6 | 6 | 7 | 10 |
| 5 | 5 | 5 | 6 | 6 | 7 | 9 | 15 |
| ⑤ | 6 | 6 | ⑥ | 7 | 7 | ⑨ | 12 |
| 6 | 6 | 6 | 7 | 7 | 8 | 10 | 15 |
| 6 | 6 | 6 | 7 | 8 | 9 | 11 | 31 |
| ⑥ | 6 | 7 | ⑨ | 11 | 12 | ⑫ | 40 |
| 6 | 6 | 7 | 9 | 14 | 41 | 16 | 20 |
| 6 | 7 | 8 | 8 | 10 | 28 | 18 | 22 |
| ⑦ | 7 | 8 | ⑧ | 9 | 11 | ⑫ | 14 |
| 7 | 7 | 8 | 9 | 9 | 10 | 12 | 13 |

Figure 15: Data to use

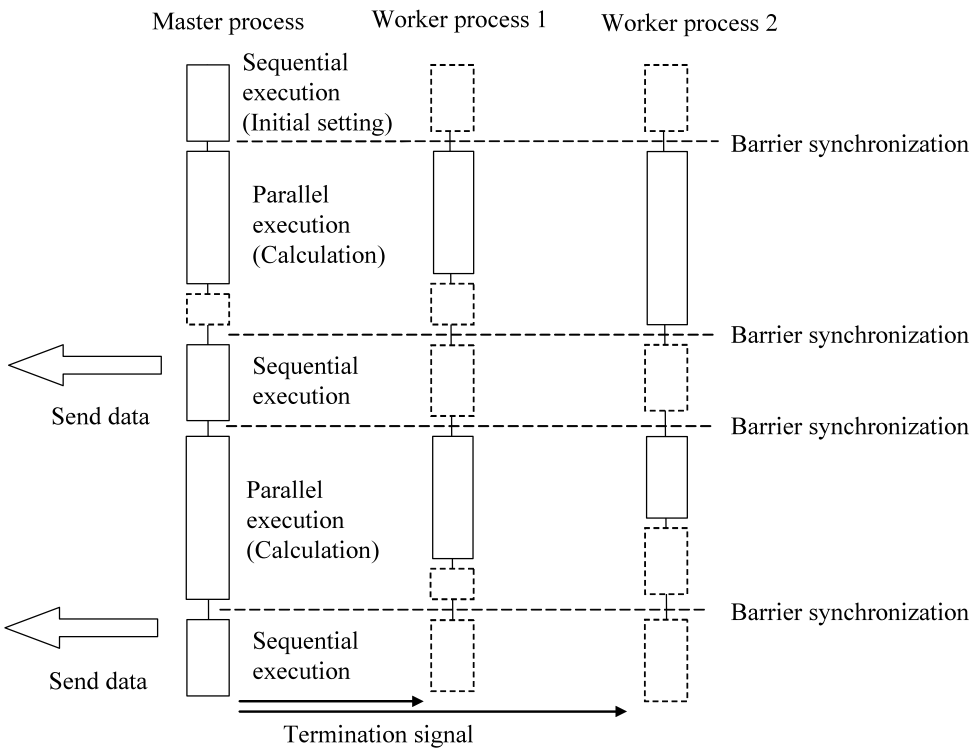


Figure 16: Multiple execution

**Multiple execution**

When the data size is larger than certain number, runtime error occurs. This is because large array in the program cannot be allocated in the local memory. Additionally, the shared memory size of the AWG-STAR system is only 512MB, so it is impossible to store all the data in it at one time. Therefore, we change the program to calculate the Mandelbrot set in some batches. For example, to calculate the program with data size $3000 \times 3000$, the area is divided into 3 area of $1000 \times 3000$, and one area is calculated in one parallel execution (see Figure 16).

## 4.3 Evaluation of Performance

### 4.3.1 Execution time

Figure 17 plots the execution time to calculate the Mandelbrot set in the AWG-STAR system, and the Figure 18 does in the SCore. 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 s 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.

### 4.3.2 CPU usage

Figures 19 to 22 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 s 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 100% of CPU usage. In Figure 19, because the data size is small and the calculation takes less than one second, it does not show high CPU usage. The reason the CPU

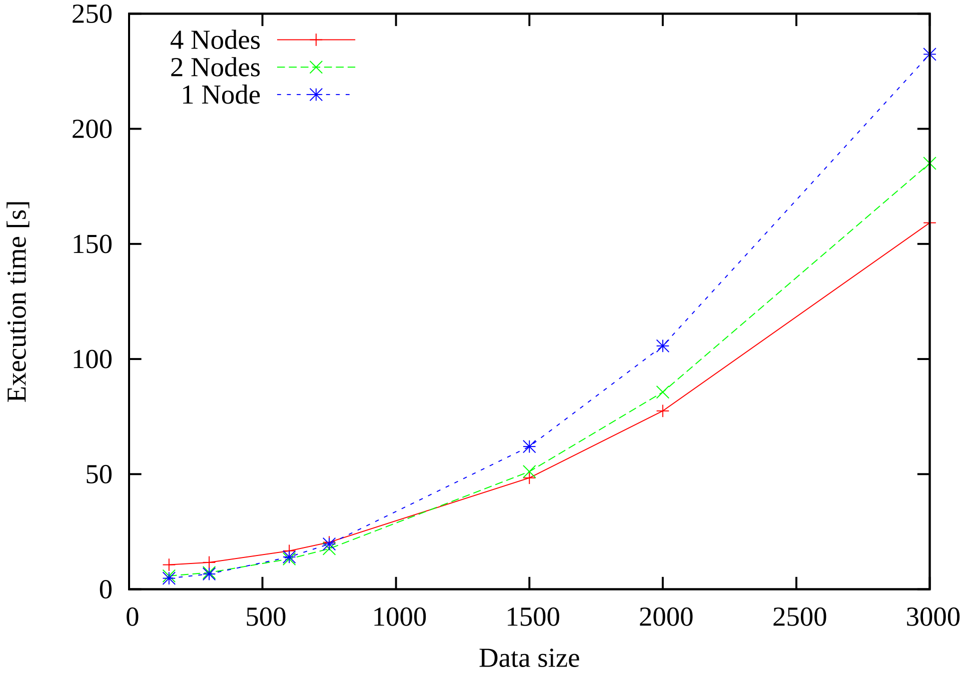


Figure 17: Execution time in AWG-STAR system

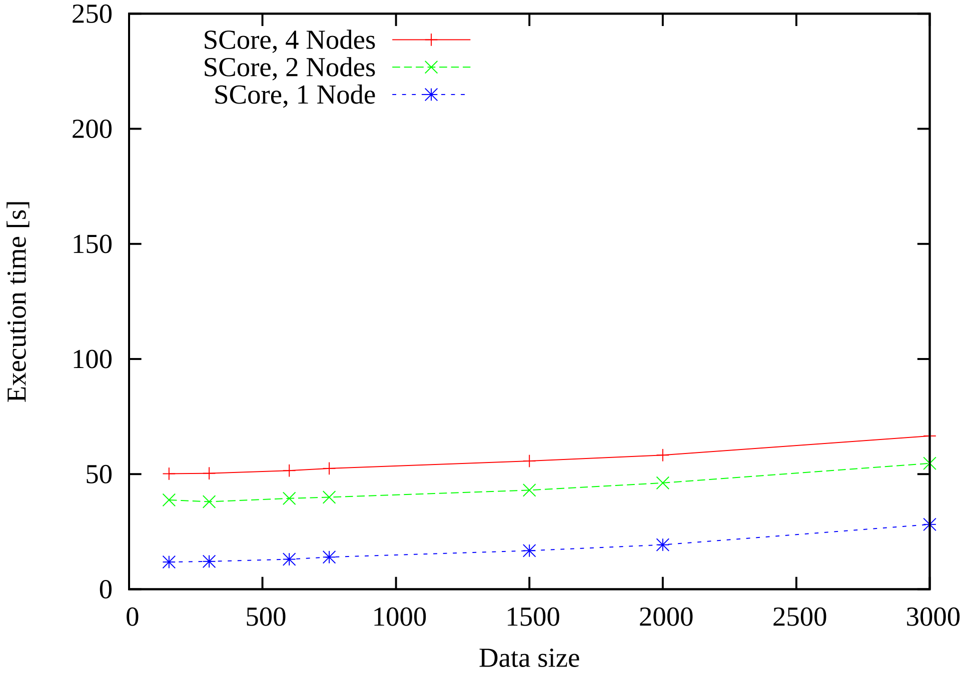


Figure 18: Execution time in SCore

usage of the main process remain around 90 % is that not only OpenMP process but also JAVA process and the operation for the synchronization primitives are performing in that node.

When the size of data are $1500 \times 1500$ and $3000 \times 3000$, the calculation is divided into multiple computations because we set the size of the array to $1000 \times 3000$ in the program. The CPU usage of that two cases are shown in Figure 21 and 22. 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 access speed to the shared memory will be improved in the next version of the AWG-STAR system, and the execution time for applications is expected to become shorter.

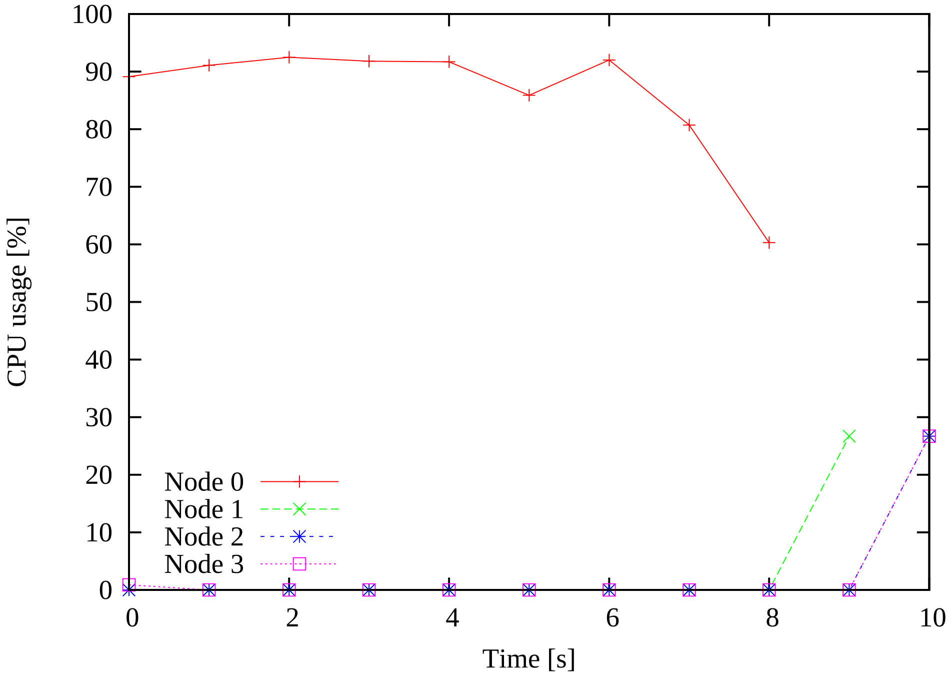


Figure 19: Data size 300x300

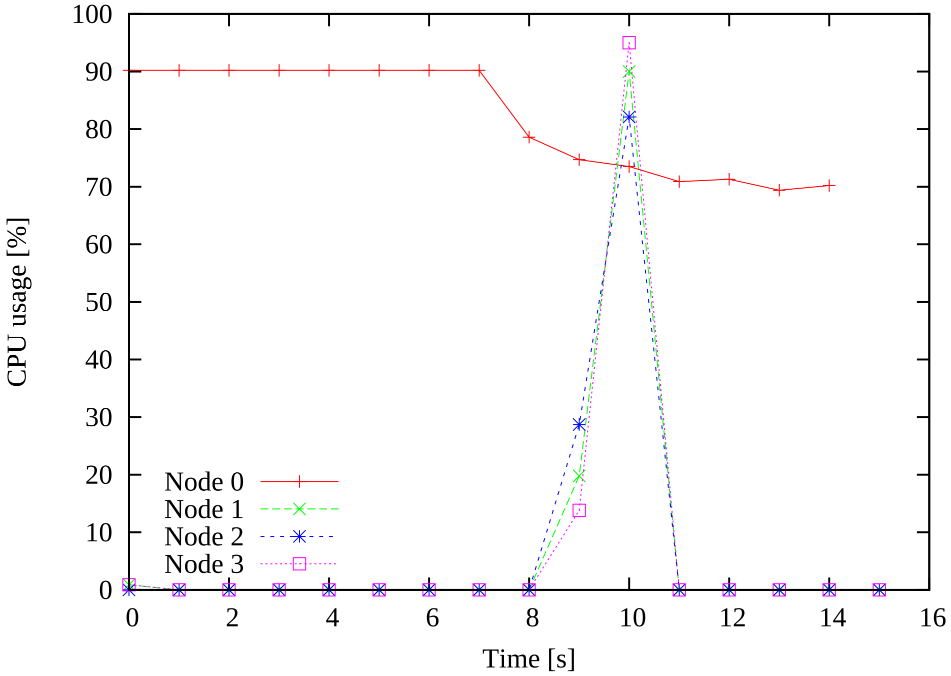


Figure 20: Data size 600x600

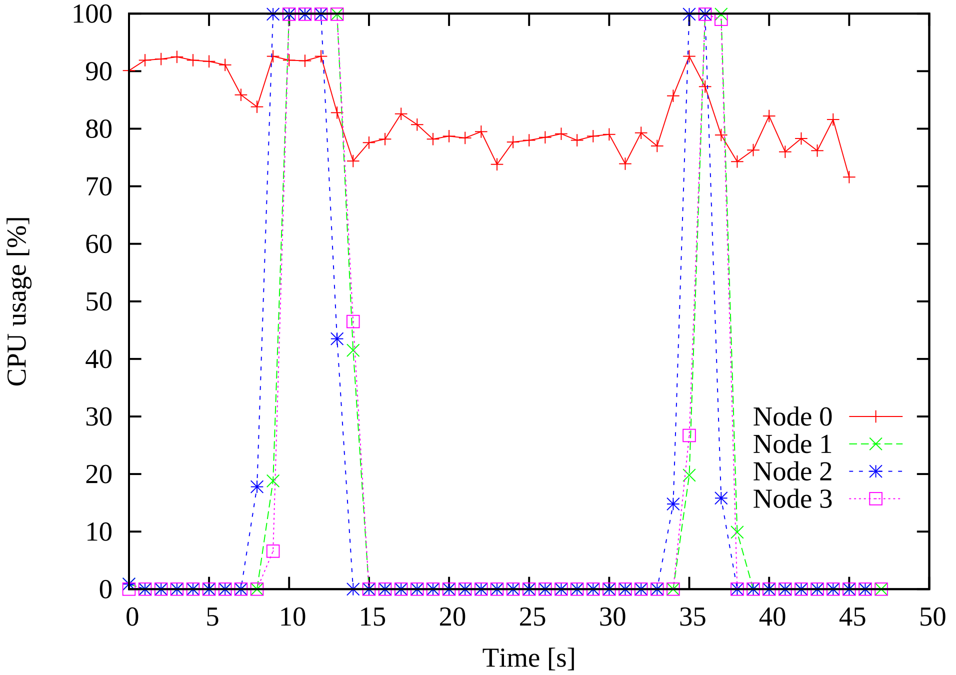


Figure 21: Data size 1500x1500

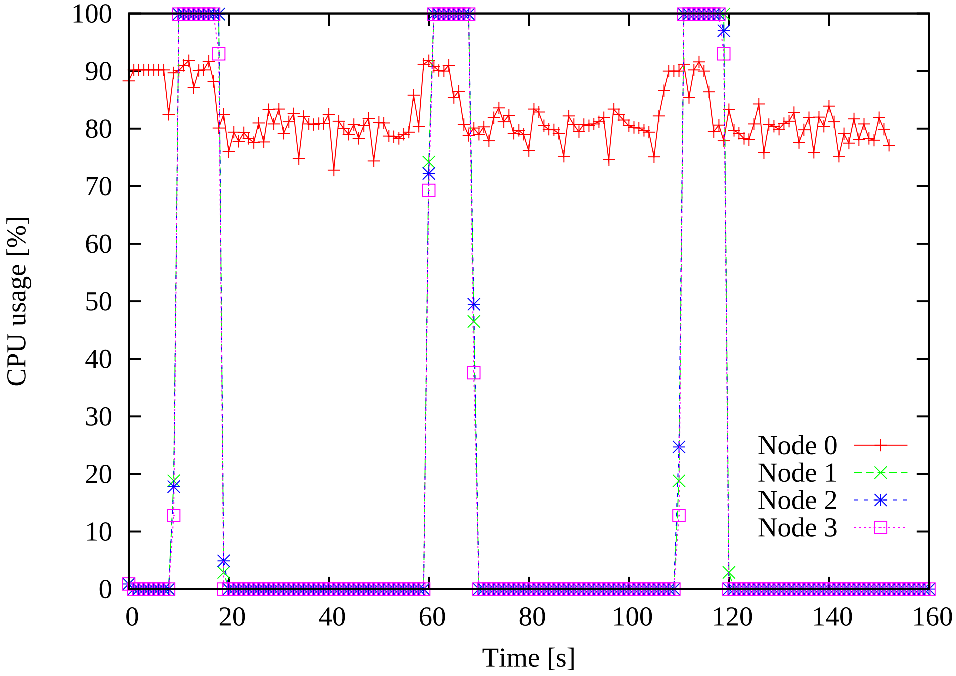


Figure 22: Data size 3000x3000

# 5 Design of the Synchronization Primitives for the Next version of AWG-STAR System

As we described in Section 4, the performance of the OpenMP application utilizing the current AWG-STAR system is not sufficient. One of the reasons is the low access speed to the SMB. Additionally, only one control token on the optical ring also keeps the performance low. To solve these problems, a new version of the AWG-STAR system is now developing by NTT Photonics Laboratory. In this section, we design the synchronization primitives for the next version of the AWG-STAR system

## 5.1 Brief Overview of the Next Version of the AWG-STAR System

Before describing the design of synchronization primitives, we explain brief overview of the new AWG-STAR system. There are two main differences between the current and the new: one is the architecture and the other is the signal. While the shared memory is on the SMB which is connected to the computer with PCI bus in the current version, the shared memory uses a part of the local memory in the next version. This makes the access speed to the shared memory equal to that to the local memory. The details of the architecture is described in [12].

The other change is a signals and a control token used in the system. In the current version of the AWG-STAR system, data to update the shared memory are transfered by attaching to one control token. Thus, expected waiting time for the control token is long, and more than one data are attached to it. In the next version, there are multiple control token on the optical ring. We call "free token" which data is not attached to, and "data token" which data is attached to. When a process updates data in the shared memory, it waits for the free token going around the ring and attaches the data to it. The signal is changed from "free token" to "data token" and updates other shared memories. Thus, one token is attached to only one data, and the waiting time to update the shared memory is expected to be shorter because of the multiple free tokens.

Additionally, there are another type of signal, which is not attached to a free token. Each node can send that kind of signal anytime, but the signal does not change data in the shared memory. There are three types of communication in this signal: point-to-point communication, multicast communication, and broadcast communication. As described in the next section, we utilized this signal on designing synchronization primitives.

Table 4: Frame configuration

| | Section name | Contents |
|---|---|---|
| MAC [47:0] | MAC address | Mac address of the SMB |
| TOKEN [2:0] | Token identification | 000: Free token<br>001: data token<br>011: point-to-point communication<br>100: Multicast communication<br>101: Broadcast token<br>110: Control Token |
| C_ST [4:0] | Control token number | Given number of control token |
| Node_ID [7:0] | Node number | Node number |
| Multi_G [3:0] | Multicast group | Figure that shows the multicast group |
| Act_Seq [3:0] | Action sequence | Sequence number of divided data |
| Packet_Cnt [7:0] | Packet count | Sequence number one computing nodes has sent |
| Addr [31:0] | Address | Offset address in the shared memory |
| Data | Data | Data or pad if the frame is smaller than 64 Byte |
| FCS | Frame check sequence | Frame check sequence |

**Frame configuration**

Table 4 shows the configuration of signal frame, including free token, data token, and other types of signal. The signal starts with the MAC address of the computing node which sent that signal, and next shows the token identification section showing the type of the signal. When the token identification section is 000 or 001, the signal is free token or data token used to change the data in the shared memory. When token identification section is 011, 100 or 101, the signal is not used to change the data but the communication between computing nodes. When the token identification section is 110, the signal is control token used for link control (not same control token as the current version), and the next 5 bits in the frame show the function of the control token section. These 5 bits of control token identification section are, however, ignored other than control token, so we set here the meaning of the synchronization primitive. Next node number section carries

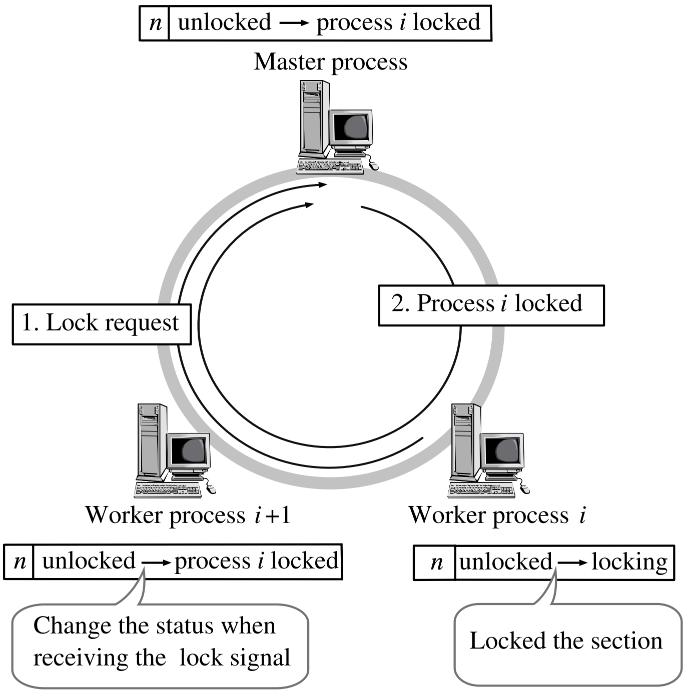


Figure 23: Method to get lock in master-worker style

that of the send node or receive node. Multicast group section, action sequence section, packet count section and address section are used for specific signals. In data section, it carries the data to update the shared memory if the signal is data token, or other data to control the communication. There is frame check sequence at the last of the signal.

## 5.2 Lock Control Function

We design two styles for the lock function. One is master-worker style, which is as same as the current version of the AWG-STAR system, and the other is distributed style. We utilize the same OpenMP compiler, so each critical section is distinguished by the lock number.

### 5.2.1 Master-worker style

In this style, all the processes have each lock index in their own local memory, but their indexes are updated only by the master process or the process which got the lock. To lock a section, the worker process $i$ confirms the status of the lock number is “unlocked”, and sends lock-request signal to the master process with point-to-point communication. When the master process receives the signal,

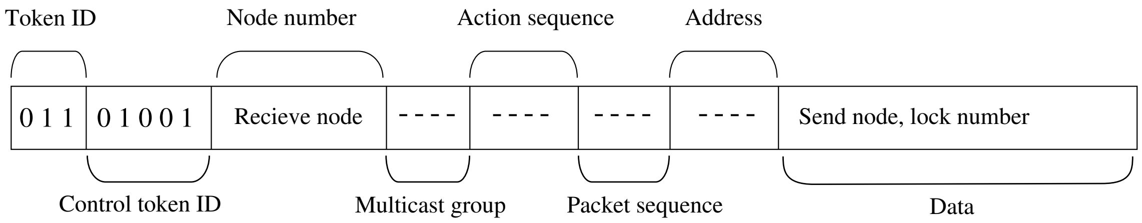


Figure 24: Lock request signal on master-worker style

it sends broadcast signal which makes all the processes to change the lock index to "process $i$ locked". The process $i$ can know that it got the lock. Figure 23 shows a series of function to get the lock.

When the process wants to unlock the critical section, it sends broadcast signal which means the section is now unlocked. All the processes update the index to "unlocked".

### Configuration of the signal

We set here the configuration of the signal. As described before, we use the control token ID section to identify the meaning of the signals. Also, the token ID section is determined by the type of the communication.

- Lock request signal (Figure 24)

  Because the signal is point-to-point communication, the control token ID section is 011, and we set the number 01001 for the control token ID section. The other required information is send and receive node and request lock number, so we carry receive node in the node number section, and send node and lock number in data section.

- Lock acknowledge signal (Figure 25)

  Because the signal is broadcast, the control token ID section is 101, and we set the number 01010 for the control token ID section. The other required information is the locked node and lock number, so we carry locked node in the node number, and lock number in data section.

- Unlock signal (Figure 26)

  Because the signal is broadcast, the control token ID section is 101, and we set the number

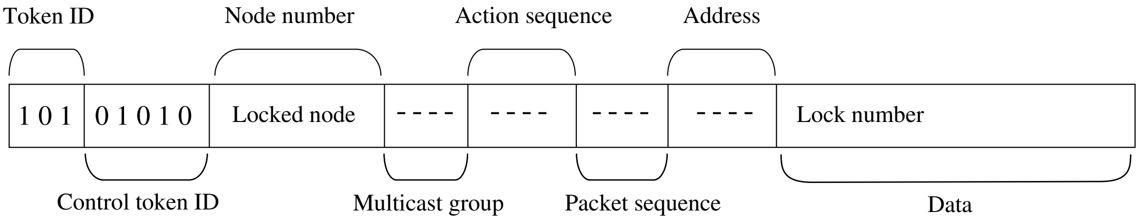


Figure 25: Lock acknowledge signal on master-worker style

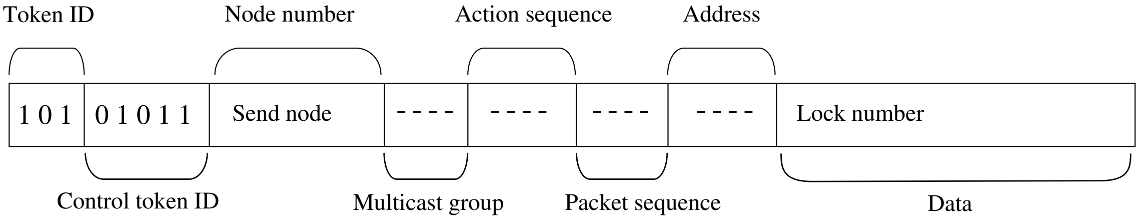


Figure 26: Unlock signal on master-worker style

01011 for the control token ID section. The other required information is the send node and lock number, so we carry send node in the node number, and lock number in data section.

### Duplication of the signal

In this method, earlier the signal arrived at the master process, earlier its process can get the lock. We consider here two of the processes send signals at almost same time. One process $i$ sends signal first, and another process $j$ does that secondly before it receives the broadcast signal from the master process, the master process receive two signals. It gives the lock acknowledgment to the process $i$, whose signal arrived at the master process earlier. After executing the critical section, process $i$ sends broadcast signal to unlock. The master process passes the lock acknowledgment to the process $j$ without any signals from the process $j$. Figure 27 shows the series of function when the signals are duplicated.

### Signal loss

In normal performance, the process which sent lock-request signal will receive a lock-acknowledgment signal from the master process, regardless of whether the process can get the lock. Therefore, in the case that the process does not receive any signals from the master process within definite

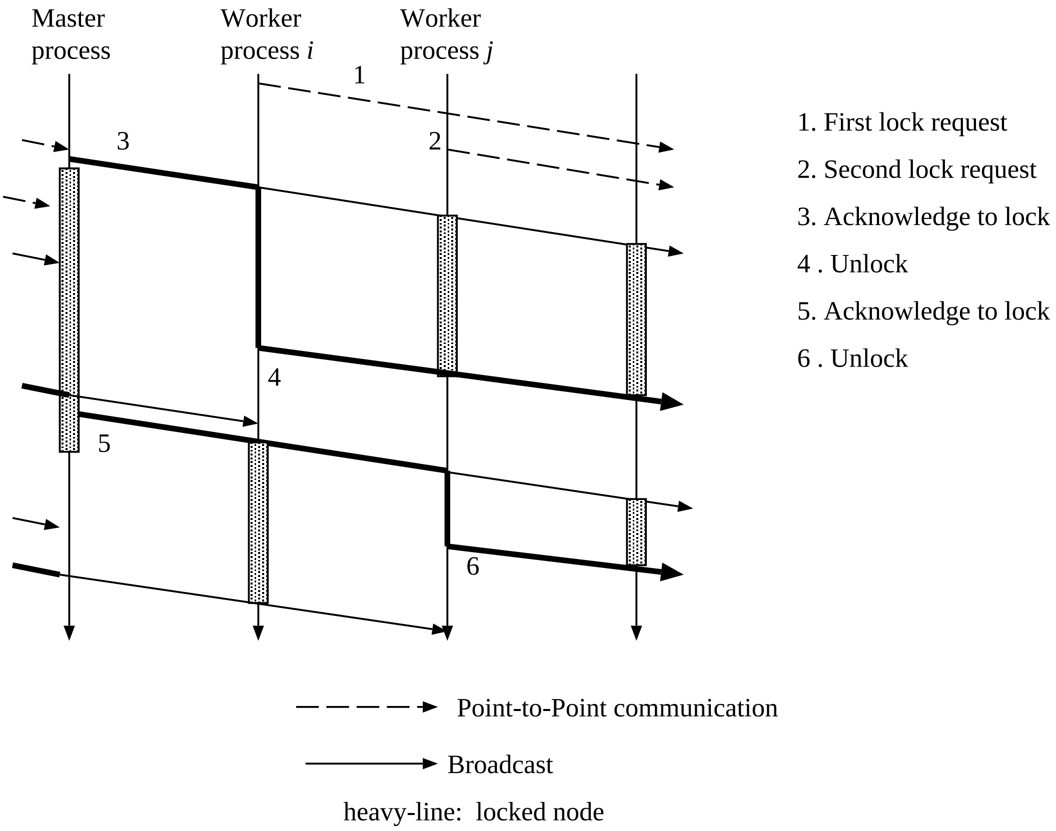


Figure 27: Duplication of signal on master-worker style

period of time, there must be a signal loss and the process sends the signal again.

Receiving lock-request signal, the master process sends lock-acknowledgment signal. Because this signal is broadcast, the master process can notice signal loss if the signal does not come back to itself, so it sends the signal again. The unlock signal is also broadcast signal, so the sending process does the same way.

### 5.2.2 Distributed style

In this style, we utilize a lock right, and only the process which is holding the lock right can lock the section. The processes pass around the lock right regardless of master or worker process, and each process has the lock index in its own local memory. The lock index has the status of whether it holds the lock right or not, as well as the three section status of “locking”,“another process locked” and “unlocked”.

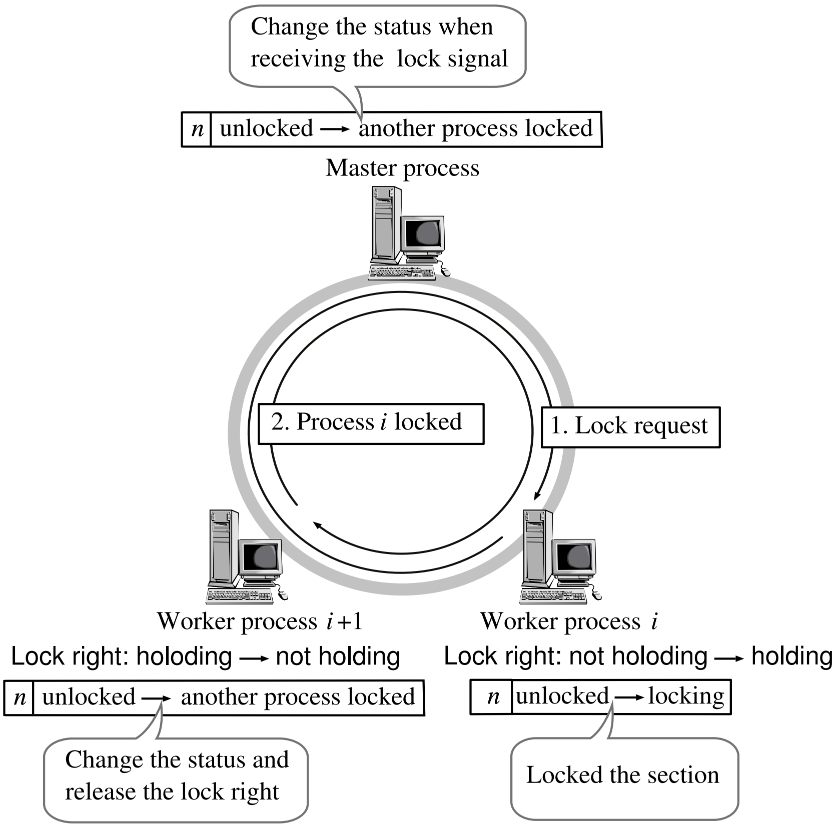

Change the status when receiving the lock signal
n unlocked → another process locked
Master process
2. Process i locked
1. Lock request
Worker process i+1
Lock right: holoding → not holding
n unlocked → another process locked
Change the status and release the lock right
Worker process i
Lock right: not holoding → holding
n unlocked → locking
Locked the section

Figure 28: Method to get lock in distributed style

To lock a section, confirming the status of its lock index is "unlocked", the process sends broadcast signal to request the lock right. The process which is holding the lock right, or locked the section last time, passes the lock right to the process by adding an acknowledgment to the broadcast signal from the requesting process, and changes the lock index status from "holding" to "not holding". The process which gets the lock updates the index status to "locking", and the processes other than that update it to "another process locked". Figure 28 shows a series of function to get the lock.

When the process wants to unlock the critical section, it sends broadcast signal which means the section is now unlocked. All the processes update the index to "unlocked", but the lock right is keep held by the same process until another process requests it.

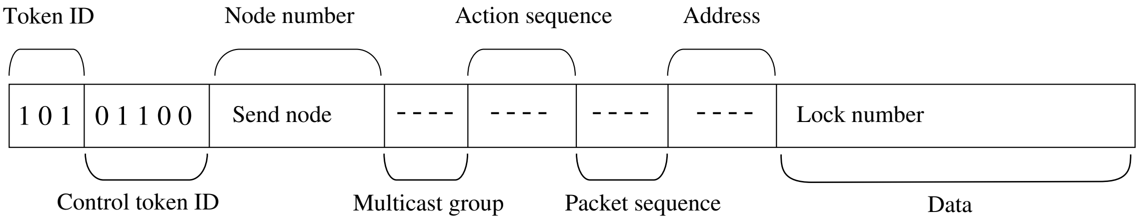


Figure 29: Lock request signal on distributed style

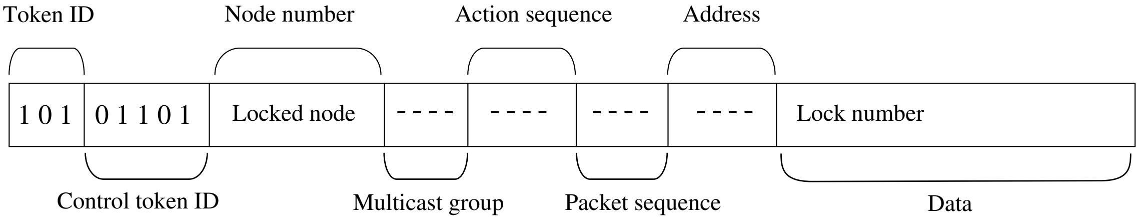


Figure 30: Lock acknowledge signal on distributed style

**Configuration of the signal**

- Lock request signal (Figure 29)

  Because the signal is broadcast, the control token ID section is 101, and we set the number 01100 for the control token ID section. The other required information is the send node and lock number, so we carry send node in the node number, and lock number in data section.

- Lock acknowledge signal (Figure 30)

  Because the signal is broadcast, the control token ID section is 101, and we set the number 01101 for the control token ID section. The other required information is the locked node and lock number, so we carry locked node in the node number, and lock number in data section.

- Unlock signal (Figure 31)

  Because the signal is broadcast, the control token ID section is 101, and we set the number 01110 for the control token ID section. The other required information is the send node and lock number, so we carry send node in the node number, and lock number in data section.

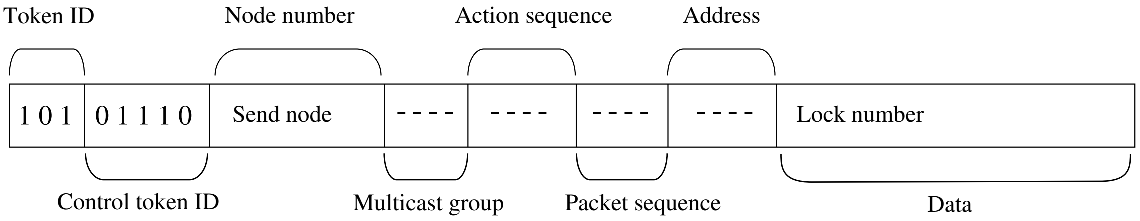


Figure 31: Unlock signal on distributed style

#### Duplication of the signal

In this method, earlier the signal arrived at the process holding the lock, earlier the process can get the lock. If one process $i$ sends request signal and another process $j$ does the same after that, process $i$ gets the lock and notices process $j$ also wants the lock right. After executing the section, process $i$ passes the lock right to process $j$ instead of sending unlock signal, and changes the status of index to "another process is locking". Process $j$ receives the lock right and returns ACK to process $i$. Figure 32 shows the series of function when the signals are duplicated. At the point of star mark, process $i$ can notice that process $j$ also request to lock.

#### Signal loss

In normal performance, the lock-request signal will come back to the process which the signal is sent by. Therfore, process can notice signal loss if the signal does not come back to itself, so it sends the signal again. The process which held the lock right has to pass it again.

When unlocking the section, it sends broadcast signal. In the case of signal loss, the process also can notice it and send the signal again. Because the signal to pass the lock right on the duplication is point-to-point communication, process which received signal returns ACK signal.

### 5.3 Barrier Synchronization Method

We propose also two styles for the barrier synchronization method, which are master-worker style and distributed style.

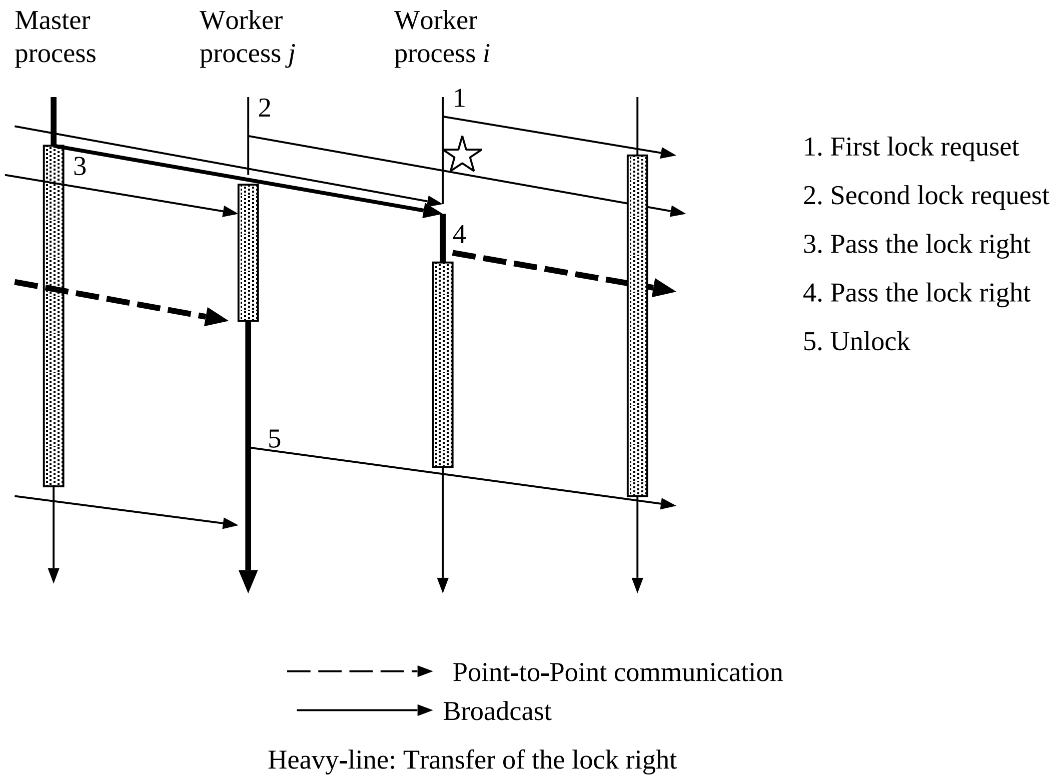


Figure 32: Duplication of signal on distributed style

### 5.3.1 Master-worker style

The process arriving at the barrier firstly sends a signal to the master process with point-to-point communication, receives ACK signal from the master process, and suspends the execution. When receiving signals from the worker processes, the master process return ACK signals for them. After all of the worker processes sent it to the master process, the master process sends broadcast signal which breaks the suspend of the worker processes. Receiving the signal, they restart the execution.

#### Configuration of the signal

- Barrier arrival signal (Figure 33)

  Because the signal is point-to-point communication, the control token ID section is 011, and we set the number 01111 for the control token ID section. The other required information is send and receive node, so we carry receive node in the node number section, and send node

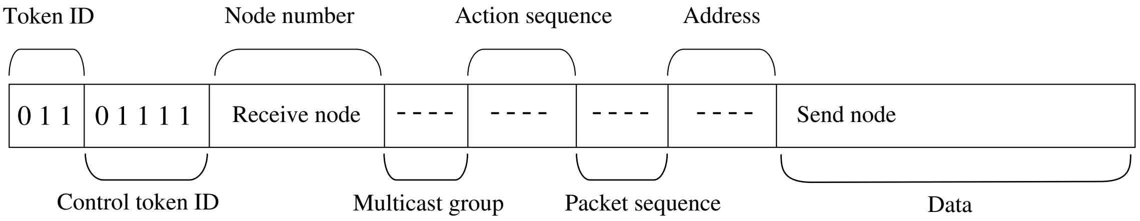


Figure 33: Barrier arrival signal on master-worker style

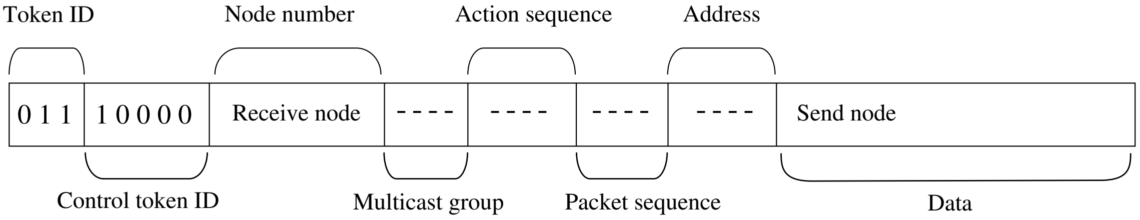


Figure 34: ACK signal on master-worker style

in data section.

- ACK signal(Figure 34)

  Because the signal is point-to-point communication, the control token ID section is 011, and we set the number 10000 for the control token ID section. The other required information is send and receive node, so we carry receive node in the node number section, and send node in data section.

- Barrier release signal (Figure 35)

  Because the signal is broadcast, the control token ID section is 101, and we set the number 10000 for the control token ID section.

### Signal loss

If the signal to notice arriving at the barrier or the signal to reply that is lost, the process which sends the signal knows that and sends it again. If the signal to break the suspend is lost, the master process knows that because the signal is broadcast, and the process sends it again.

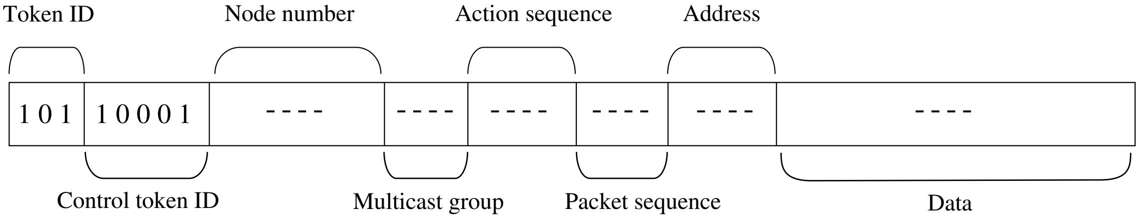


Figure 35: Barrier release on master-worker style

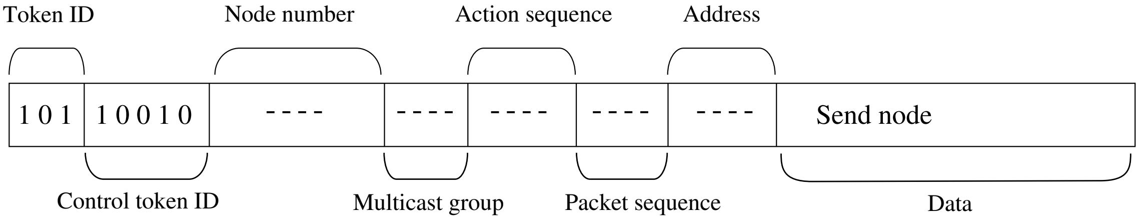


Figure 36: Barrier arrival signal on distributed style

### 5.3.2 Distributed style

In this distributed style, instead of setting the master process, we determine the process which arrives at the barrier at first as a temporary main process of that barrier. When arriving at the barrier, the process sends broadcast signal to inform other processes of it. If the process is the first one to do that, the signal sent by it returns to itself, and it becomes the main process. The process other than the main process also sends broadcast signal, but difference from the first is that it receives ACK signal from the main process. After all of the processes sent the signal to the main process, the main process sends broadcast signal which breaks the suspend of the other processes. Receiving the signal from the main process, they restart the execution.

#### Configuration of the signal

- Barrier arrival signal (Figure 36)

  Because the signal is broadcast, the control token ID section is 101, and we set the number 10010 for the control token ID section. The other required information is send so we carry send node in data section.

- ACK signal(Figure 37)

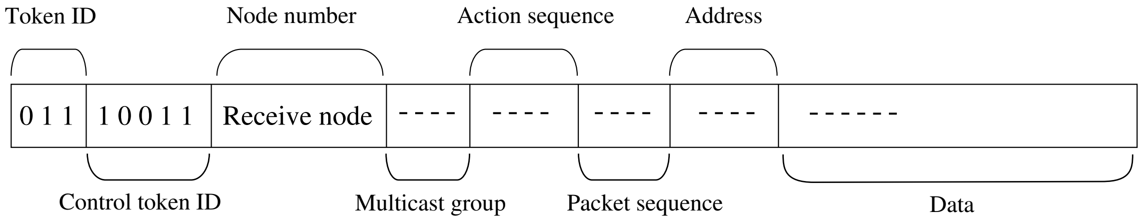


Figure 37: Barrier ACK signal on distributed style

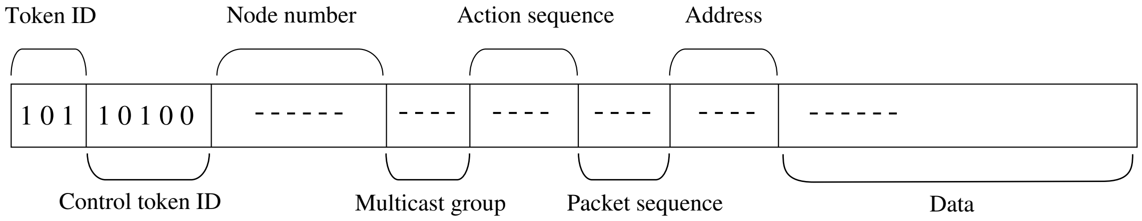


Figure 38: Barrier release signal on distributed style

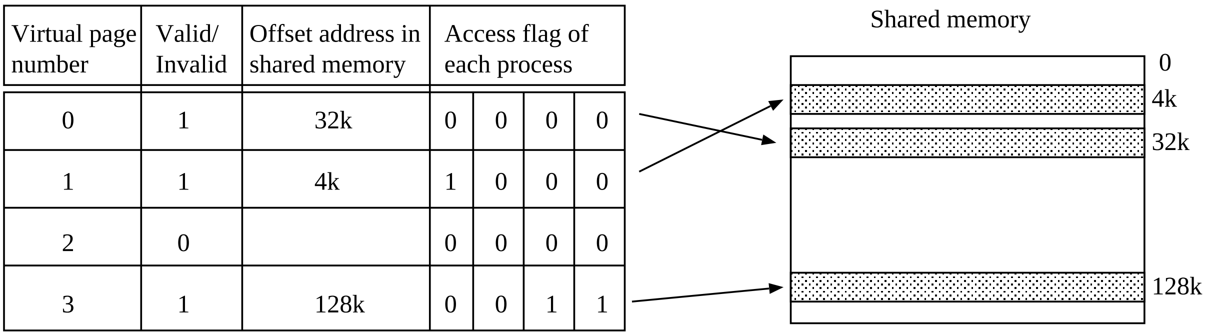
Because the signal is point-to-point communication, the control token ID section is 011, and we set the number 10011 for the control token ID section. The other required information is receive node, so we carry receive node in the node number section.

- Barrier release signal (Figure 38)

  Because the signal is broadcast, the control token ID section is 101, and we set the number 10100 for the control token ID section.

### Duplication of the first signal

We consider here more than one processes arrive at the barrier at the same time at first. Process $i$ and $j$ send broadcast signal before they receive the signal from the other one, both will recognize themselves as the master process becafuse they do not receive ACK signal. In this case, we determine the process $i$ is the master process if $i < j$. Therfore, if the process receives another barrier-arriving signal during its signal is going around the optical ring and the process number of the signal sent process is larger that its own, it sends the signal again. To do so, only one process is determined as the main process, and other process will receive ACK signal from the main process by sending the signal again.



| Virtual page number | Valid/ Invalid | Offset address in shared memory | Access flag of each process | | | |
|---|---|---|---|---|---|---|
| 0 | 1 | 32k | 0 | 0 | 0 | 0 |
| 1 | 1 | 4k | 1 | 0 | 0 | 0 |
| 2 | 0 | | 0 | 0 | 0 | 0 |
| 3 | 1 | 128k | 0 | 0 | 1 | 1 |

Figure 39: Page table and allocation

**Signal loss**

If the signal to notice arriving at the barrier or the ACK signal to reply that is lost, the process which sends the signal knows that, and sends it again. If the signal to break the suspend is lost, the main process knows that because the signal is broadcast, and the process sends it again.

### 5.4 Shared Memory Virtualization

As described in Section 3.1, the current implementation for dynamic memory allocation cannot use larger area than the AWG-STAR system provides, though the size of the system is not enough. Thus, we propose additional design of shared memory virtualization and its memory allocation.

#### 5.4.1 Page table and memory allocation

We divide the virtual shared memory into pages and manage them by a page table. The Figure 39 shows the page table and allocation of the pages. The pages are numbered in order from 1, and there is Valid/Invalid flag showing whether it exists in physical shared memory. If the flag is 1, it means the pages is in the physical shared memory that the AWG-STAR system provides, and its address is stored in the next entry as the offset address from the top of the shared memory. Furthermore, there are flags for each page if the processes executing OpenMP application are accessing to the area. This whole page table is stored in static area of the shared memory, and all the processes can access it. The master process mainly updates the table but the access flag of each process is changed by each assigned process.

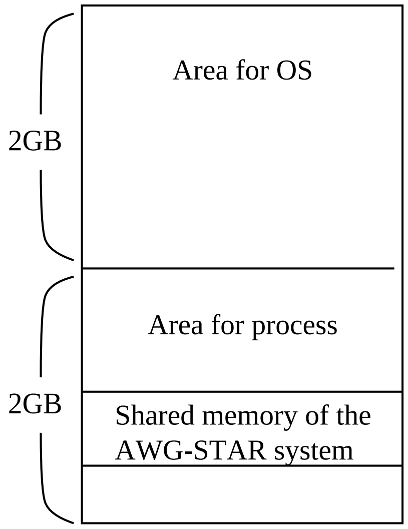


Figure 40: Address space mapping

### 5.4.2 Address Space Mapping

The address space mapping is shown in Figure 40. Because the address is 32 bit, the address space is 4GB, and 2GB of them is used for processes. When we use the AWG-STAR system, a part of the address space for the process is used as the shared address space.

The first area to allocate the shared variables is physical shared memory of the AWG-STAR system. In the same manner of the current implementation, we reserve the area of shared memory in advance, and allocate it from the top. In case when larger size of shared memory is needed, we firstly reserve one page in local memory for the swapping are. Next, we reserve another page and allocate that from the top. This reservation is done by calling the function of local memory allocation. In this way, we utilize the local memory as the virtual shared memory, so if we need more shared memory, we can use less local memory.

Appropriate size of one page is determined by a simulation or execution of an application. Small page size takes extra time to search the entry in the page table and many changes of access flag of each process. But the page size is too large, overhead of swapping becomes large.

### 5.4.3 Access to the shared memory

We consider here that a process accesses to a shared variable. The process knows its virtual shared address and also virtual page number. Thus, by accessing to the page table in the shared memory, the process can know the physical address in the shared memory or that the variable is not in the shared memory.

#### In case that the page is in the shared memory

Before the process accesses to the page, it changes the access flag in the page table from 0 to 1. And also, when it accesses to another page, it changes the flag to 0. Because of the locality of reference, it is considered that there are not so frequent update of the flag.

#### In case that the page is NOT in the shared memory

When the page to access does not exist in the physical shared memory, the page is needed to be swapped in. If the process is executed at the worker, the worker process sends the signal to the master process to request the page to be swapped in. Receiving the signal, the master process executes swapping and updates the page table. After the worker process notices the update of the page table, it accesses the page in the way of described above.

### 5.4.4 Swapping

#### Request to swap a page in

When a worker process requests to the master process of swapping a page in, it utilizes the Point-to-Point communication signal. Figure 41 shows the configuration of the request signal. The required information are the receive node which the master process is running and the request page. Because the signal is Point-to-Point communication, the token ID section is 011, and we set 11000 in the control token for this signal. The number of the worker nodes is carried in node number section and the request page number is done in data section.

#### Page swap operation

When the master process receives a request from a worker process, or it accesses to a page that does not exists in physical shared memory, it makes a page swap. The first thing to do is to decide

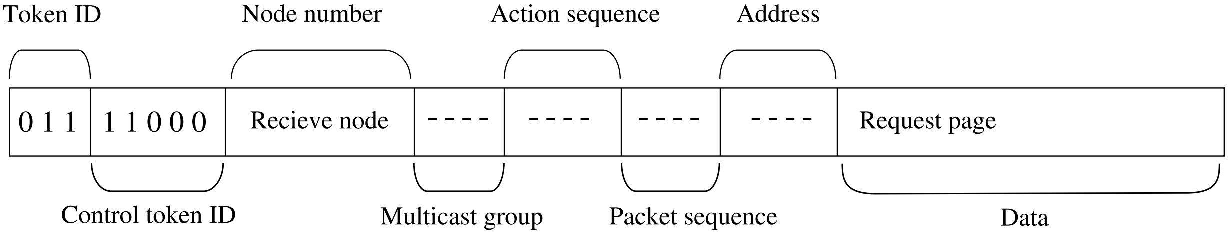


Figure 41: Swap request

which page in the shared memory is swaped out. Because the page to swap out must not be accessed by any processes, we choose one of the pages whose access flags in the page table are all 0.

Next, it changes the Valid/Invalid flag in the page table from 1 to 0, and copies the swap out page to the local memory area which are reserved for the swapping in advance. After that, it copies the swap in page from the local memory to the physical shared memory, and finally it updates the page table. The area for swapping is changed every time when swapping is operated.

### Overlapping on swap out and access

A problem occurs when the timing of swapping out a page and the access to that page overlaps. Figure 42 shows the case. A worker process sends a signal to change the page table when accessing the page, before the signal that notices the page will be swapped out. We prioritize here the signal of the master process over the worker process. Therefore, the worker process should check if there is a signal from the master process during its signal is going around the optical ring. If it receives the signal from the master process, it sets the access flag back to 0, and sends a signal to request to access the swapped out page.

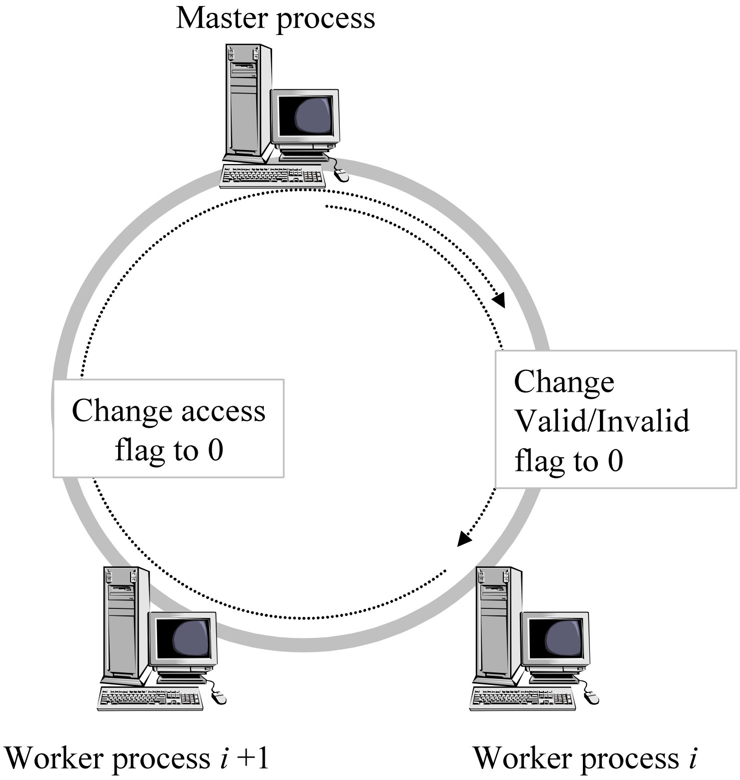


Figure 42: Overlapping on swap out and access

## 6 Conclusion

In this thesis, we proposed a new architecture of distributed parallel computing environment, the $\lambda$ computing environment, which utilizes the optical wavelength path for the interconnection of the shared memory system. We established the $\lambda$ computing environment with the AWG-STAR system so that to execute high performance parallel computation on it. We chose OpenMP shared-memory parallel programming as a parallel computation, and we designed and implemented the functions such as lock control function, barrier synchronization method and dynamic memory allocation for it. We described the design and implementation of these methods in Section 3. We adopted master-worker approach to implement synchronization primitives including lock control function and barrier synchronization method, and established stack structure for the dynamic memory allocation.

We implemented Mandelbrot set which graphically shows the calculation result to evaluate the performance of the system. The execution time of the application shows an advantage of parallel computation because the execution time shortens with many processes as the amount of data increases. CPU usage during calculation of Mandelbrot set shows, however, that the access speed to the shared memory of the AWG-STAR system is slow.

To resolve the bottleneck, NTT Photonics Labolatory are now developing the next version of the AWG-STAR system improved a memory access method to a shared memory. In this thesis we also designed the synchronization primitives and data sharing structure for it. We designed the synchronization primitives which utilized the signals for the AWG-STAR system so that to attain high speed synchronization. We proposed two approaches, master-worker style and distributed style, for both lock control function and barrier synchronization method. It is future work to assess which approach can attain higher performance. Lastly we designed we proposed a virtualization of the shared memory to execute a large-scale computation required larger memory than the available physical shared memory of the AWG-STAR system. We can expect better performance by implementing them on the next version of the AWG-STAR system.

## Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Masayuki Murata of Osaka University, who introduced me to the area of computer networks. His thoughtful leading and advice have been essential for not only my research activity but my career.

I am also deeply grateful to Associate Professor Ken-ich Baba at Osaka University for his appreciated guidance and encouragement. All of my works would not have been possible without his supports.

I am heartily grateful to Associate Professor Noriyuki Fujimoto at Osaka University for his creative suggestions and insightful comments. My thank also goes Dr. Akira Okada at NTT Photonics Laboratory for his useful advice and technical support. I also thank his colleagues for hearty welcome during my internship in their office.

I am also indebted to Associated Professor Naoki Wakamiya, Go Hasegawa, and Specially Appointed Associate Professor Kenji Leibnitz, and Research Associates Shin'ich Arakawa, Masahiro Sasabe, Yuichi Ohsita at Osaka University for their helpful comments. Finally I wish to express my warmest thanks to my friends in the Department of Information Networking of Osaka University.

## 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 $\lambda$ computing environment," in *Proceedings of IFIP Optical Networks and Technologies Conference (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 $\lambda$ computing environment," in *Proceedings of the forth International Conference on Optical Internet (COIN2005)*, pp. 90–95, May 2005.

[4] E. Taniguchi, K. Baba, and M. Murata, "Implementation and evaluation of shared memory system for establishing $\lambda$ computing environment," in *Proceedings of 10th OptoElectronics and Communications Conference (OECC2005)*, 5A2-3, pp. 20–21, July 2005.

[5] E. Taniguchi, "Design and evaluation of shared memory architecture for WDM-based $\lambda$ computing environmnet," Master's thesis, Graduate School of Informantion Science and Technology, 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 $\lambda$ computing environment," in *Proceedings of 6th Asia-Pacific Symposium on Information and Telecommunication Technologies*, 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 *Proceedings of 27th European Conference on Optical Communication 2001*, vol. 3, pp. 264–265, Sept. 2001.

[9] A. Okada, H. Tanobe, and M. Matsuoka, "Dynamically reconfigurable real-time information-sharing network system based on a cyclic-frequency AWG and tunable-wavelength lasers," in

*Proceedings of 29th European Conference on Optical Communication 2003*, vol. 4, pp. 978–979, Sept. 2003.

[10] J. Hoeflinger and B. de Supinski, "The OpenMP memory model," *Conference: Presented at: First International Workshop on OpenMP, Eugene, OR (US), 06/01/2005–06/04/2005*, 2005.

[11] V. Dimakopoulos, E. Leontiadis, and G. Tzoumas, "A portable C compiler for OpenMP V. 2.0," *Proc. of the European Workshop on OpenMP (EWOMP '03), Aachen, Germany*, Sept. 2003.

[12] K. Goda, "Design and implementation of OpenMP compier for the $\lambda$ computing environment," Master's thesis, Graduate School of Informantion Science and Technology, Osaka University, Feb. 2007.

[13] "PC Cluster Consortium," available at `http://www.pccluster.org/`.

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