Master's Thesis

Title

Design and Implementation of OpenMP Compiler for λ Computing Environment

Supervisor

Masayuki Murata

Author

Keigo Goda

February 14th, 2007

Department of Information Networking

Graduate School of Information Science and Technology

Osaka University

Master's Thesis

Design and Implementation of OpenMP Compiler for λ Computing Environment

Keigo Goda

Abstract

In recent years, the demand for the grid computing technology is arising, and it has been researched widely. In the existing grid computing environment, TCP/IP is usually used for communication between computing nodes. However, the packet processing of TCP/IP degrades network throughput and computing performance. Thus, we have proposed a new architecture called the λ computing environment, that has wavelength paths between computing nodes and optical switches, and provides these paths as a realization method to achieve high-speed and high-reliable communication in the grid computing environment.

In this thesis, we established λ computing environment using the AWG-STAR system, and we design and implement OpenMP application programming interface for the AWG-STAR system. The AWG-STAR system is an information sharing network system based on WDM technology which is developed by NTT Photonics Laboratory. We provide the OpenMP compiler designed for the AWG-STAR system. Our OpenMP compiler is based on the existing OpenMP Compiler called OMPi, and we modify OMPi and its runtime library to enable parallel computing on the AWG-STAR system. Our OpenMP compiler utilize the shared memory system provided by the AWG-STAR system for data sharing, and our runtime library controls parallel execution on computing nodes connected with the AWG-STAR system.

Next, we evaluate the performance of our implementation for the AWG-STAR system by executing the benchmark programs. However, as a result, current AWG-STAR system is not able to achieve comparable performance to existing parallel computing environment. On the AWG-STAR system, we must access the shared memory via the PCI bus from the CPU. Since the access speed of the PCI bus is much lower than local memory, the insufficient access speed of the shared memory degrades the performance of computing.

Currently, NTT Photonics Laboratory is developing the next version of the AWG-STAR system which targets the improvement of the memory access performance. We estimate the effectiveness of the improvement on the next version of the AWG-STAR system in this thesis. Our estimation shows that the improvement on the next AWG-STAR system will be significantly effective.

Keywords

λ Computing Environment
AWG-STAR System
OpenMP
Distributed Parallel Computing
Distributed Shared Memory

Contents

List of Figures

List of Tables

1 Introduction

In recent years, the demand for the grid computing technology is arising, and it has been researched widely. In the grid computing environment, we can utilize CPUs and storages of computing nodes interconnected by the network as a single virtual computer. Today, various applications of this technology are being studied. For example, they are the gene information analysis, image processing and the global environment simulation, and so on.

In the existing grid computing environment, TCP/IP is usually used for communication such as control messages and data exchanges between computing nodes. However, TCP/IP has some harmful effects in such environment. For example, some packets may be lost on the route from the source node to the destination node because of traffic congestion caused by own volume data transmission on the network. The retransmission of lost packets cause degradation of network throughput and computing performance on the grid. So, it is difficult to achieve high-performance in application of large-scale computing that treats large-scale data.

Thus, our research group proposes a new architecture called the λ computing environment [1–4], that has wavelength paths between computing nodes and optical switches, and provides these paths as a realization method to achieve high-speed and high-reliable communication in the grid computing environment. In λ computing environment, we can realize high-speed and high-reliable data exchanging or data sharing, because computing nodes utilize established wavelength paths as exclusive communication channel instead of conventional TCP/IP network.

Our research group recently focuses the study of the architecture of the shared memory system on the λ computing environment [1–4]. In [2], they propose the shared memory system utilizing the virtual optical ring and the cache coherency protocol for it. In [4], they analyze how the network topology and the coherency protocols influence the performance, by simulation and modeling using semi-Markov process.

To enable parallel computing on the λ computing environment, we require the programming environment. In this thesis, we establish the programming environment for the λ computing environment. In other words, we provide the programming interface for the parallel computing on the λ computing environment.

Today, the de-facto standard of the application programming interface (API) for parallel computing is the message passing interface (MPI). MPI is the specification of the library interface which provides routines to send and receive messages and to synchronize processors. MPI is usually used on the distributed memory environment like clusters or the grid computing environment, but, MPI is also supported on the shared memory and the non-uniform memory access (NUMA) architectures.

We already have implementation of MPI for the λ computing environment [5]. However, development of the parallel programs using MPI is difficult. Usually, the implementation of MPI is provided as the low-level communication library. Programmers must code the detail behaviour of the parallel programs like explicit message exchanging by themselves. In addition, the programming model of MPI assumes the environment which has no shared memories. It is possible to use MPI on the λ computing environment, but programming with MPI cannot take advantage of our shared memory system. So, we choose another programming model which is suitable for our computing environment.

OpenMP [6] is the alternate standard of the API, which targets the parallel computing on the shared memory environment. OpenMP is defined as the extension to the existing programming language. Currently, OpenMP is supported in many commercial and non-commercial compilers. OpenMP provides the higher-level API which abstracts underlying computing environment and provides the support for typical parallelizations. So, programmers can develop their parallel programs more easily than MPI.

In this thesis, as an instance of the λ computing environment, we utilize the AWG-STAR system developed by NTT Photonics Laboratory. The AWG-STAR system is an information sharing network system based on WDM technology. On the AWG-STAR system, the data is transmitted via the array waveguide grating (AWG) router, which processes wavelength routing. The AWG-STAR system provides the shared memory system, that is implemented as the shared memory board (SMB) which has shared memory on it. The SMB on each node is connected to the AWG router, and the shared data is automatically synchronized among the computing nodes.

In this thesis, we aim to establish the programming environment using OpenMP on the λ computing environment. We design and implement OpenMP for the AWG-STAR system, and we evaluate the performance of our implementation by executing the benchmark applications.

The rest of the thesis organized as follows. First, in Section 2, we describe the AWG-STAR system which we utilize in this study and OpenMP which we implement on it. We design our implementation of OpenMP in Section 3, and the detail of that implementation is described in

Section 4. Our implementation is evaluated in Section 5. In Section 6, we introduce the next version of the AWG-STAR system and discuss the effectiveness of the improvement on it. Last, we conclude this study in Section 7.

2 Distributed Parallel Computing using AWG-STAR System

2.1 AWG-STAR System

The AWG-STAR system which we utilize in this study is the information sharing network platform [7, 8]. It is realized by data transmission with the WDM technology and wavelength routing using the AWG router. By using the AWG router, we can establish a high-speed data transmission network, because it processes optical signals without optical/electrical conversion.

Each computing node connected with the AWG-STAR system is equipped with a shared memory board (SMB). The SMBs are directly interconnected via the wavelength path, and configure a ring topology in logical. The SMB has a shared memory and it is connected with the computing node via PCI bus. In the AWG-STAR system, the contents of this shared memory are automatically kept identical all over the nodes. No explicit instruction is required to transmit or receive data. We can exchange the data in real-time via an optical ring network by using this shared memory. This shared memory is mapped to the normal memory space. So, we can access to the memory without distinction between the shared memory of the AWG-STAR system and the local memory which the computing node originally has.

Figure 1 summarizes data sharing scheme using the AWG-STAR system. In the AWG-STAR system, the computing node can share the data with the other nodes by writing the data to the shared memory. On the optical ring, one control token is circulating. The data written to the shared memory is automatically sent to the optical ring network by attaching the data to the control token. When the computing node writes some data to the shared memory, the SMB updates own shared memory and waits for the arrival of the control token. When the SMB receives the control token, if the control token has the update data from the other computing nodes, each SMB updates own shared memory. Next, the SMB attach written data to the control token if it has written data, and the SMB passes the control token to the next node. As updating is performed in background by hardware, we can share the data at high speed. By reading values from own SMB, each computing node can get the shared data updated by other nodes.

2.2 OpenMP

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

The implementation of OpenMP is usually provided as the compiler called OpenMP compiler. In OpenMP, programmers develop a parallel program by inserting compiler directives called OpenMP directive into the source code. OpenMP directive is the instruction to express parallelism and data sharing. Via OpenMP directives, programmers tell the compiler to parallelize the specified part of the program. So, programmers need not write the detail behavior of the parallel program in the source code.

Figure 2 shows a simple example of the OpenMP source code. The statement “pragma” at the 2nd line is OpenMP directive. In this example, the loop part from the 3rd line is executed in parallel. All 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 2 stages (see Fig. 3). The 1st stage is transformation of source code. The OpenMP compiler interprets the OpenMP directive in the input source code, and generates the intermediate code based on the information provided from directives. The intermediate code is that specific code required for parallel processing is inserted. For example, library function-calls for communication and synchronization are inserted. The 2nd stage is source code compilation. The intermediate code is converted to the program object of machine language.

As remarked above, programmers can develop their programs to parallel easily because the compiler converts instead of them according to their directive comments. In addition, there are no codes which depends on the specific computing environment in the source code of the parallel program. However, because the intermediate code depends on specific computing environment including the hardware, OS, and the communication library, the OpenMP compiler must be implemented dedicatedly for each computing environment.

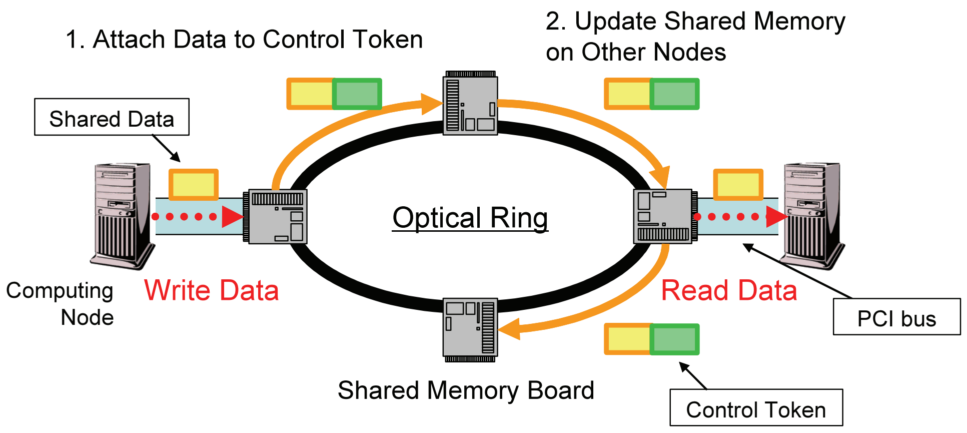

Figure 1: Data Sharing using AWG-STAR System

```
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);
}
```

Figure 2: Example of OpenMP Source Code

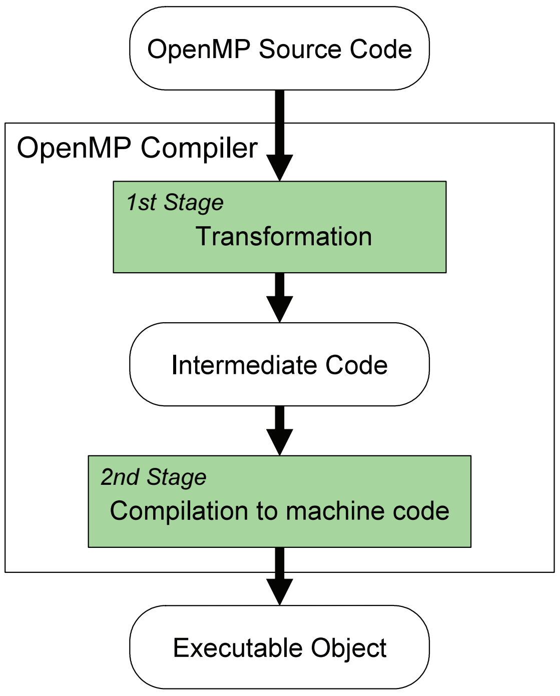

Figure 3: Overview of OpenMP Compiler

3 Design of OpenMP Compiler for AWG-STAR System

We have to design an OpenMP compiler generating an intermediate code for the AWG-STAR system to implement OpenMP in the λ computing environment. OpenMP is 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 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 utilize distributed shared memory (DSM) technology that enables all or part of the memory in each computing nodes to 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 the software. The AWG-STAR system used in our study, on the other hand, has a DSM system that maintains the consistency of memories through the hardware.

On the AWG-STAR system, we use two kinds of memory for distributed parallel computing. One is a local memory in each computing node originally and the other is a shared memory which is provided by a shared memory board (SMB) of the AWG-STAR system. 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 [9] is therefore useful for our own implementation method of OpenMP on the AWG-STAR system.

When we implement OpenMP on the AWG-STAR system, we need to establish:

(1) The execution mechanism for the parallel-execution section,

(2) The data-sharing scheme, and

(3) The lock function and barrier synchronization function including dynamic memory allocation

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

3.1 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 sections and parallel-execution sections in OpenMP programs. The implementation methods of OpenMP on the distributed parallel computing environment are roughly classified into two types in terms of the treatment of the sequential execution sections. One executes the sequential execution section with 1 computing node, and the other method does with all computing nodes. The former method clearly distinguishes the computing node called the master, which executes the sequential execution section from other computing nodes called worker. Workers receiving the request from the master, and execute the parallel execution section. Therefore, only the master always executes computation. In contrast, in the latter method always all computing nodes execute computation.

The latter method [10] is proposed in order to reduce the overhead which occurs on the beginning request of the parallel execution section in the former method. However, because the computing nodes are connected via the wavelength pass with low delay on the AWG-STAR system, this overhead does not cause a significant problem. Therefore we adopt the former method in our research.

In our method, 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. 4).

A request for execution of the parallel execution section is informed from the master to the workers by using the barrier synchronization on the AWG-STAR system. So, we do not need additional network with TCP/IP or other protocols. Barrier synchronization is a mechanism that synchronizes more than two computing nodes at the same time. Each computing node temporary suspends processing at the provided point of the program and waits until all computing nodes arrives at the same point of the program.

The summary of the mechanism of the parallel execution request is as follows.

(1) Initially, all workers are standby with the barrier synchronization function called.

(2) The master writes a memory address of the program code which workers should execute to the shared memory.

(3) The master calls the barrier function, and releases the standby state of workers.

(4) Workers start execution with specified address.

Parallel computation with multiple computing nodes is enabled by such a technique.

3.2 Data-sharing Scheme

The data sharing in OpenMP means the values of variables in the program is 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 at the outside of the parallel execution region are implicitly shared. For example, the code in the Figure 2 shares variable *pi* between computing nodes while parallel processing is executed.

A variable is shared with the SMB which the AWG-STAR system offers. In other words, it 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 5 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.

The OpenMP specifications has not only the shared variable but also the non-shared variable called the private variable. The private variables are explicitly declared in the parameter of the OpenMP directive by programmers. Our compiler does not modify the references of the private variables, and keeps it on the local memory. This is because there is no need to share values of these variables, and the access speed of the local memory is faster than the shared memory.

3.3 Lock and Barrier Synchronization

In the parallel program, shared data which is read and written by multiple computing nodes during parallel processing must be access controlled by the lock function to prevent inconsistency. And also, to guarantee the execution order of the processing, the barrier synchronization function is required. In order to execute the parallel program in OpenMP, it is necessary to provide these functions.

Therefore we originally made a synchronization primitives [11] for the shared memory and the signaling mechanism which we utilize the function offered by the AWG-STAR system. The synchronization between computing nodes in the OpenMP program is achieved by calling our synchronization primitives from the intermediate code which the compiler generates.

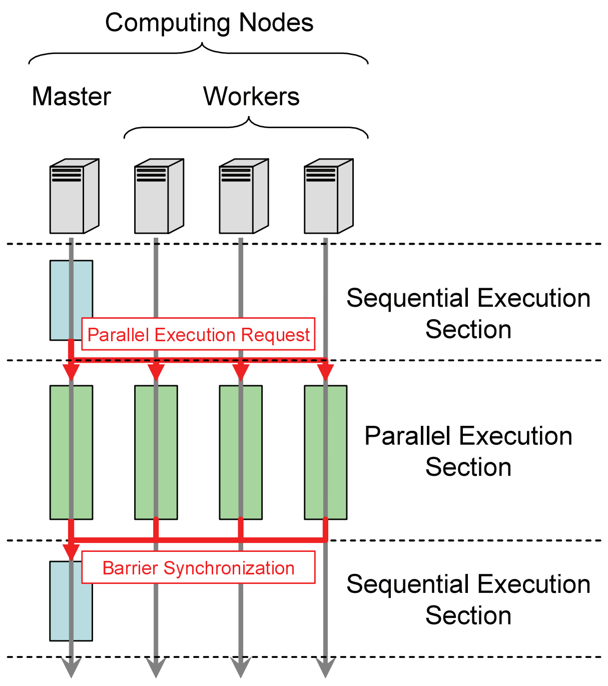
Computing Nodes
Master
Workers
Sequential Execution Section
Parallel Execution Request
Parallel Execution Section
Barrier Synchronization
Sequential Execution Section

Figure 4: Execution Model of OpenMP

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

Figure 5: Example of Variable Reference Transformation by OpenMP compiler

4 Implementation of OpenMP Compiler for AWG-STAR System

Our OpenMP compiler is implemented based on OMPi [12]. OMPi is the research OpenMP compiler for C language which is developed by University of Ioannina, and it supports the SMP environment with the POSIX threads library.

OMPi consists of 2 programs and the runtime library. `omipcc` is the front-end program of OMPi and the special linker for OpenMP programs. `ompi` is a C-to-C translator which generates the intermediate code of C language from input OpenMP source code. `libompi` runtime library coordinates parallel processing and manages the thread pool via the POSIX threads API.

The compilation sequence of the OpenMP source code using OMPi is shown in Figure 6. First, `ompi` transforms input OpenMP source code to the intermediate code. Second, the intermediate code is compiled to the object file by the back-end compiler. The back-end compiler is standard C compiler, and `cc` is used as the back-end compiler defaultly. Last, `ompicc` links object files with the runtime library, and creates an executable object. At the same time, `ompicc` generates initialization codes for shared global variables and links generated codes with object files and the runtime library.

We modify original OMPi to use computing nodes instead of threads for parallel execution. We implement the runtime library of OMPi to support the AWG-STAR system and modify the translator to allocate the memory areas of the shared variables on the shared memory of the AWG-STAR.

Our OpenMP compiler for the AWG-STAR system currently supports subsets of OpenMP features. Most of important OpenMP features are already implemented, but `ordered` directive, `copyprivate` clause, and some features are not implemented yet. The implementation status of OpenMP directives and clauses is summarized in Table 1.

4.1 Runtime Library

`libompi` runtime library is a set of functions which is called from the intermediate code generated by `ompi`. Figure 7 shows a common form of the intermediate code. The functions prefixed by `_omp_` are runtime library functions.

`libompi` controls execution of the parallel execution section. `ompi` translator extracts the parallel execution section in the OpenMP source code as the function like `main_parallel_0`

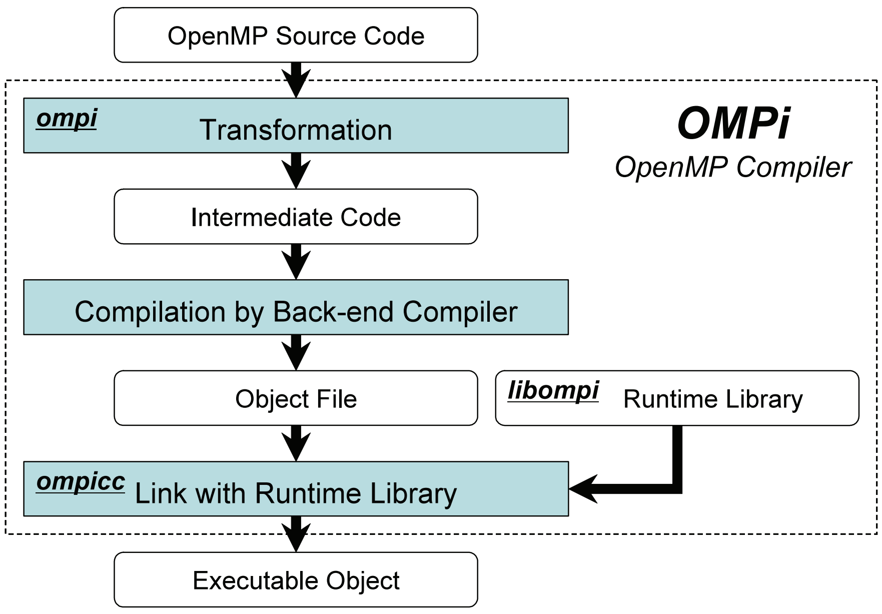

Figure 6: Compilation Sequence of OMPi

Table 1: Implementation Status of OpenMP Directives and Clauses

OpenMP Directive		OpenMP Clause	
parallel	√	if	√
for	√	num_threads	√
sections	√	schedule	√
section	√	ordered	
parallel for	√	nowait	√
parallel sections	√	default	√
single	√	shared	√
master	√	private	√
critical	√	firstprivate	√
barrier	√	lastprivate	
atomic	√	copyin	
flush	√	copyprivate	
orderd		reduction	√
threadprivate	√		

Original

```
int main(int argc, char **argv)
{
#pragma omp parallel
  {
    /* Parallel Excecution Section */
  }
}
```

Intermediate Code

```
int main (int argc, char **argv)
{
  _omp_initialize ();
  {
    _OMP_PARALLEL_DECL_VARSTRUCT (main_parallel_0);
    _omp_create_team ((-1), _OMP_THREAD, main_parallel_0,
                       (void *) &main_parallel_0_var);
    _omp_destroy_team (_OMP_THREAD->parent);
  }
}

void *main_parallel_0 (void *_omp_thread_data)
{
  int _omp_dummy = _omp_assign_key (_omp_thread_data);
  {
    /* Parallel Excecution Section */
  }
  return 0;
}
```

Figure 7: Basic Code Transformation by OMPi

in Figure 7. And, `ompi` inserts runtime library function-call `_omp_create_team` instead of it. `_omp_create_team` is a function to request worker threads to execute specified function.

Our runtime library has the same interface as original `libompi`, but uses computing nodes instead of threads for execution of parallel sections. Each computing node has the identification number called node ID. The computing node which has node ID 0 behaves as the master. The other computing nodes behave as workers.

The function `_omp_initialize` in original `libompi` initializes worker thread-pool. Our implementation of its function initializes the AWG-STAR system and the library of lock and barrier synchronization. And then, workers start infinite loop for waiting parallel execution request from master (see Fig. 8). The master exits this function and starts execution of the sequential execution section.

The function `_omp_create_team` is the implementation of the parallel execution request for the master. Figure 9 shows the overview of this mechanism. This function inform the address of the function to workers, which workers should execute. We reserve the memory area to inform the function address on the shared memory. We call this area execution information area (EIA). In function `_omp_create_team`, the master writes the address to EIA and calls barrier synchronization. Workers read the function address from EIA, and execute that function.

The function `_omp_destroy_team` is used to wait for workers. The master returns to execution of the sequential execution section after calling this function.

4.2 Code Transformation

Figure 10 is a code transformation example of shared variables. Local variables referred in the parallel execution section are declared inside function `main_parallel_0`. The private variable is simply declared as a local variable like variable `y`. The shared variable like variable `x` is declared as a pointer variable and is accessed through the pointer. The memory addresses of shared variables are shared by workers using an address table. In this example, structure `func_parallel_0_vars` is used to share the addresses of shared variables. We call it shared variable table (SVT).

These transformations are supported in original `ompi`. However, original `ompi` allocates shared variables to the local memory. So, we modified `ompi` to use the shared memory.

We use two functions for management of the shared memory area. `_ompiawg_malloc` is

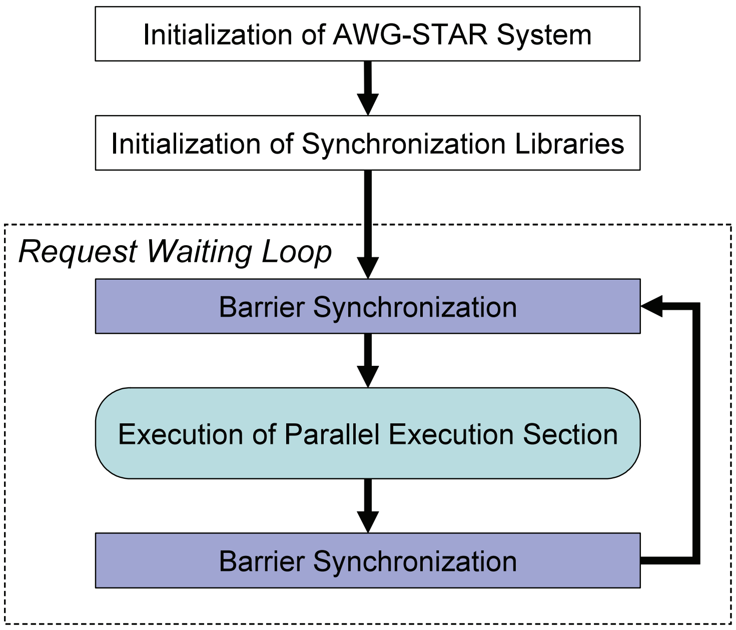
Initialization of AWG-STAR System
Initialization of Synchronization Libraries
Request Waiting Loop
Barrier Synchronization
Execution of Parallel Execution Section
Barrier Synchronization

Figure 8: Behaviour of Function `_omp_initialize` on Worker

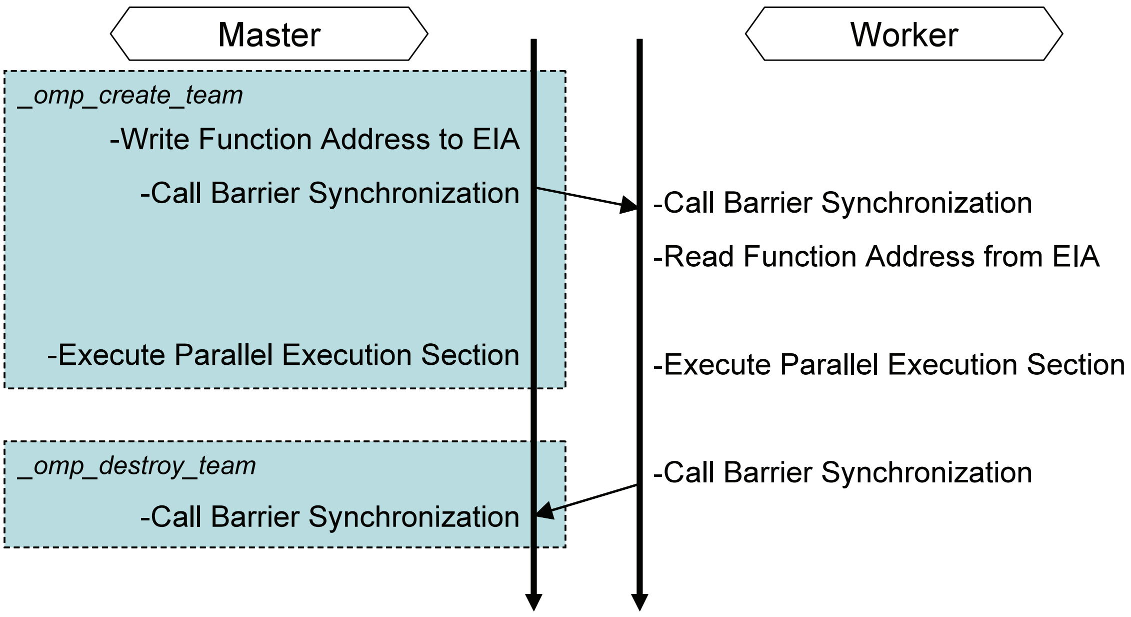
Master
Worker
_omp_create_team
-Write Function Address to EIA
-Call Barrier Synchronization
-Execute Parallel Execution Section
-Call Barrier Synchronization
-Read Function Address from EIA
-Execute Parallel Execution Section
-Call Barrier Synchronization
_omp_destroy_team
-Call Barrier Synchronization

Figure 9: Behaviour of Function `_omp_create_team` and `_omp_destroy_team`

Original

```
int g;

void func()
{
  int x, y;

#pragma omp parallel private(y)
  {
    y = x + g;
  }
}
```

Intermediate Code

```
typedef struct
{
  int (*x);
} func_parallel_0_vars;

int (*__G_g);

void func ()
{
  int x, y;
  {
    func_parallel_0_vars *func_parallel_0_var;
    func_parallel_0_var =
                _ompiawg_malloc (sizeof (func_parallel_0_vars));
    func_parallel_0_var->x = _ompiawg_malloc (sizeof (x));
    memcpy ((func_parallel_0_var->x), &(x), sizeof (x));

    _omp_create_team((-1), _OMP_THREAD, func_parallel_0,
                     (void *) func_parallel_0_var);
    _omp_destroy_team(_OMP_THREAD->parent);

    memcpy (&(x), (func_parallel_0_var->x), sizeof (x));
    _ompiawg_free (func_parallel_0_var->x);
    _ompiawg_free (func_parallel_0_var);
  }
}

void *func_parallel_0 (void *_omp_thread_data)
{
  int _omp_dummy = _omp_assign_key (_omp_thread_data);
  int (*x) = &(*((func_parallel_0_vars *)
                 (_OMP_THREAD->sdn->shared_data))->x);
  int y;
  {
    y = (*(x)) + (*(__G_g));
  }
  return 0;
}
```

Figure 10: Code Transformation for Shared Variables

used to allocate memory, and `_ompiawg_free` is to release memory. Only the master has a responsibility to memory management of the shared memory. The master allocates the memory areas for the shared variables, and inform the addresses of shared variables to workers. We also allocate SVT on the shared memory of the AWG-STAR system, and inform its address. An address of SVT is passed to workers by using the function `_omp_create_team`.

In the case of global variables, they are also accessed through the pointer and allocated on the shared memory like variable `g` in Figure 10. However, memory areas of the shared global variables are not allocated at compile-time. We allocate shared memory for them at run-time, because we must get the address of the shared memory from the device driver of the AWG-STAR system. `ompicc` generates the initialization code for the shared global variables, and the master executes it in the function `_omp_initialize`.

5 Performance Evaluation

We evaluate the performance of our OpenMP implementation in λ computing environment by executing benchmark programs. We uses cluster middleware called SCore [13] for comparison, which can accomplish parallel computing with OpenMP in the Ethernet environment. SCore offers an SDSM system called SCASH achieving a shared memory virtually in the distributed memory environment, and a dedicated OpenMP compiler called Omni/SCASH [9].

5.1 Environment for Evaluation

Table 2 and 3 list the specification of experimental system and specification of the AWG-STAR system. The computing nodes which are used for experiment are 4 nodes and all same specifications. In addition, in this experiment, assuming number of node computers as N, the length of optical ring network of the AWG-STAR system becomes $10N$m.

We select BT and EP benchmarks from the NAS Parallel Benchmarks (NPB) 2.3 [14] and the Himeno benchmark [15] for our evaluation. We use the OpenMP C version of NPB 2.3 [16] which is ported by RWCP. The size of the problem is XS ($128 \times 64 \times 64$) on the Himeno benchmark and Class W (BT: $24 \times 24 \times 24$, EP: 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.

5.2 Benchmark Results and Discussion

Table 4 and Figure 11 show the results for BT and EP of the NPB 2.3 and Table 5 shows the result for the Himeno benchmark. Table 4 and Figure 11 show the execution time for the benchmark program, and Table 5 lists the performance in MFLOPS. SCore outperforms the AWG-STAR system, which does not achieve sufficiently high performance in these tables and figure. However, there is a tendency for performance to improve as the number of the computing nodes increases. The problem with access to the shared memory may be why the AWG-STAR system performed poorly.

SCASH as an SDSM system attained a shared memory virtually 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 accessed data on 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 2GB/s, the maximum access speed to the shared memory is only about 80MB/s. All accesses 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 is about 20-fold with four nodes in the NPB 2.3 EP, but it is 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 is greater than the difference in the benchmark with low-access frequency. The difference in performance with the Himeno benchmark is 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.

Table 2: Specifications of Computing Nodes

CPU	Xeon 3.06 GHz
Main memory	512MB
L1 cache	20KB
L2 cache	512KB
NIC	Intel PRO/1000
PCI bus	64 bit/66MHz
PCI transfer rate	533MB/s
OS	Redhat Linux 7.3
Compiler	GCC 2.96

Table 3: Specification of Shared Memory Board of AWG-STAR system

Transmission rate of optical ring	2.152Gbps
Transfer data length	1KB
Frame processing delay	500ns
Maximum Writing speed of shared memory	64MB/s
Maximum Reading speed of shared memory	80MB/s

Table 4: Execution Time for NPB 2.3 BT Class W

Environment	Number 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

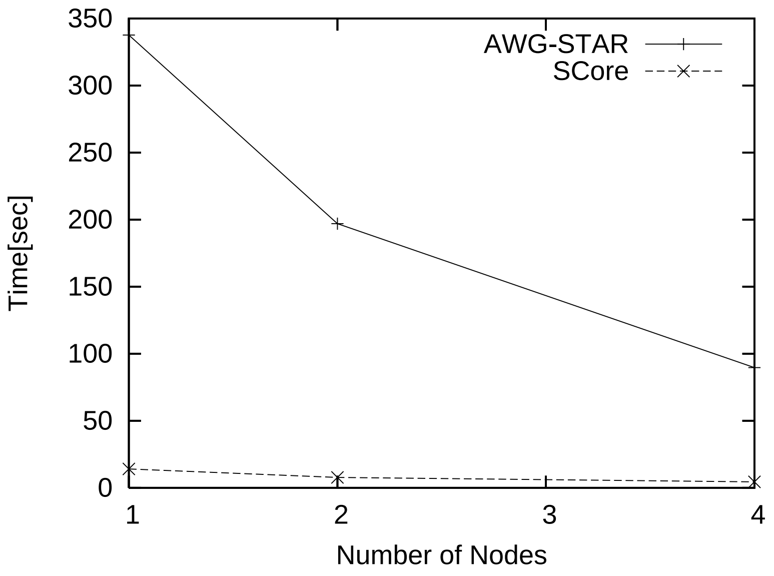

Figure 11: Execution Time for NPB 2.3 EP Class W

Table 5: Results for Himeno Benchmark

Environment	Number 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

6 Improvement on Next Version of AWG-STAR System

On the performance evaluation in Section 5, the current version of the AWG-STAR system does not achieve high performance enough. The major reason is that the access speed of the shared memory is low. To improve performance of the parallel computation on the AWG-STAR system, NTT Photonics Laboratory is developing the next version of the AWG-STAR system. On the next version, improvement of the memory access speed is mainly focused.

In this section, we introduce the overview of the next version of the AWG-STAR system and discuss effectiveness of the improvement on the next AWG-STAR system.

6.1 Overview of Next AWG-STAR System

The basic idea of the data sharing mechanism of the next AWG-STAR system is the same as the current version. The data written to the shared memory is automatically broadcasted via the optical ring network, and the shared memories of all nodes are updated. But, on the next version, the physical location of the shared memory is significantly different. And, the broadcasting mechanism using a control token is also improved.

On the current AWG-STAR system, the shared memory exists on the SMB connected with the PCI bus. As described above sections, the largest bottleneck is the access speed to the PCI bus. On the current version, all shared memory accesses are performed via the PCI bus. However the PCI bus is much lower bandwidth compared to the local memory, and is shared with the other devices. Therefore, the PCI bus degrades the performance of the shared memory in the case of both of read and write. But on the next version, we utilize a part of the local memory as the shared memory.

And also, the configuration of the hardware is different. Figure 12 and Figure 13 shows differences. The next version of the AWG-STAR system targets AMD64 platform. The computing nodes are equipped with the AWG-STAR interface board (AWG-STAR IF) which is connected to the computing node with HyperTransport. HyperTransport is the high-bandwidth, low-latency point-to-point link technology. The AWG-STAR IF is directly connected to the local memory via HyperTransport and a crossbar switch. On the current AWG-STAR system, the SMB is connected with PCI bus, but the AWG-STAR IF is connected with the higher bandwidth link. And, the current version of the SMB has the shared memory on it, but the AWG-STAR IF has no shared memory.

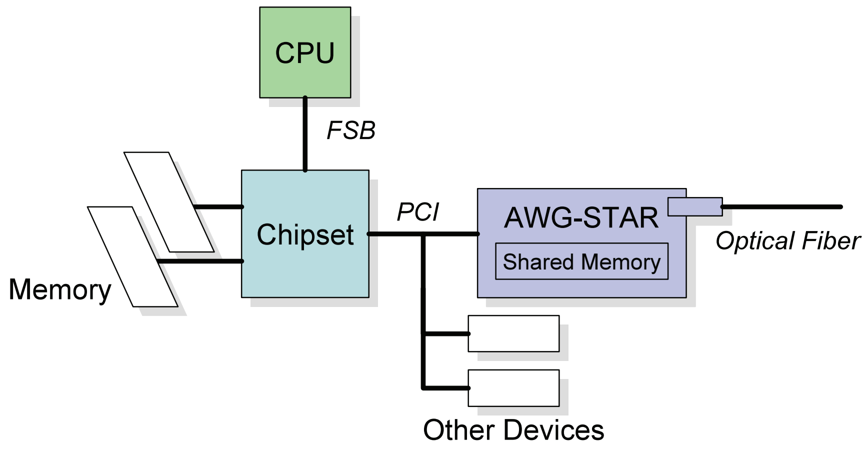

Figure 12: Configuration of Current AWG-STAR System

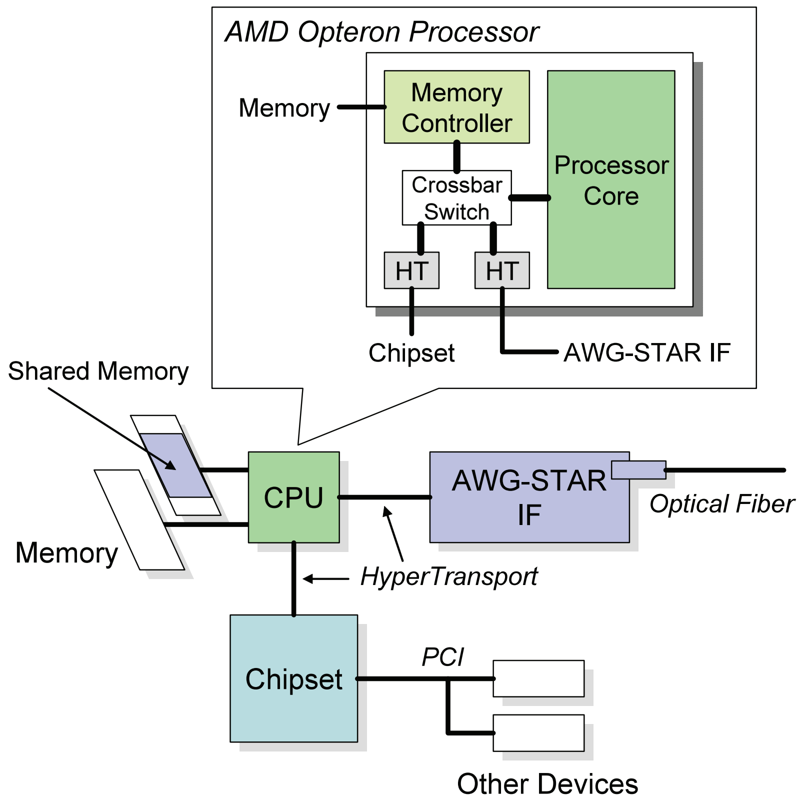

Figure 13: Configuration of Next AWG-STAR System

On the next version, the read performance of the shared memory is significantly improved. We can read from the shared memory at the same speed as the local memory, because the shared memory is the local memory in physical.

The write performance is also improved on the next AWG-STAR system. When the data is written to the memory area reserved as the shared memory, the device driver detects writing and sends written data to the AWG-STAR IF similarly to the current AWG-STAR system. However, written data is transmitted via HyperTransport instead of the PCI bus. So, the transmission time of written data is shortened. Additionally, the transmission time of the written data to the optical ring is shortened. On the current AWG-STAR system, to send the data via the optical ring, we must wait for the arrival of a control token. However, the next AWG-STAR system utilize multiple tokens. The master node of the next AWG-STAR system creates multiple tokens at some intervals continuously. When the AWG-STAR IF has the data which should be transmitted, it catches the free token and can transmit the data to the optical network immediately.

6.2 Performance Estimation

In this section, we discuss effectiveness of the improvement on the next version of the AWG-STAR system. At first, we estimate the total shared memory access time included the execution time of the benchmark program which we used in Section 5. Next, from estimated access time and expected performance of the next AWG-STAR system, we estimate how much the improvement on the next AWG-STAR system will affect the execution time. We choose NPB 2.3 EP as the target of our estimation. A result of this benchmark is shown in Figure 11.

To estimate the total shared memory access time, we require an access time of the shared memory of the current AWG-STAR system and the number of shared memory access in the benchmark program. So, we measure the average time which is needed to read or write single value on the share memory in the real application program. For measurement, we create a program which contains only the shared memory access code. Access times that we measure with the program are shown in Table 6. And, we count the number of the shared memory access in the intermediate code of NPB 2.3 EP, which is generated by our OpenMP compiler. At this time, we focus only the memory access of the master, because the execution time of the benchmark is measured on the master. Memory accesses performed by workers are not counted. Results are shown in Table 7.

From these results described above, we estimate the total shared memory access time. The total shared memory access time is defined as the product of the shared memory access time and the number of shared memory access. Table 8 shows the estimated total shared memory access time. Estimated results are almost equal to the execution time of the benchmark program. That means the execution time consists mostly of the shared memory access time. Especially, almost of them are reading from the shared memory. So, we found that the improvement of the read performance of the shared memory have a large impact on the computation performance.

As remarked in Section 6.1, the read performance of the shared memory is significantly improved on the next version of the AWG-STAR system. The read performance of the shared memory is same as the local memory. If a computing node has the local memory which can read at 2GB/s, we can also read the shared memory at 2GB/s. It is over 25 times the maximum reading speed of the shared memory of the current AWG-STAR system. Therefore, on the next AWG-STAR system, the execution time will be reduced about 25 times. So, it is thought that the improvement on the next AWG-STAR system will be significantly effective.

Table 6: Shared Memory Access Time of Current AWG-STAR System

Number of Nodes	Read (ms)	Write (ms)
1	0.005154996	0.001071544
2	0.005154996	0.001297516
4	0.005154996	0.002595032

Table 7: Number of Shared Memory Accesses in NPB 2.3 EP

Number of Nodes	Read	Write
1	67109910	29
2	33554966	29
4	16777494	29

Table 8: Total Shared Memory Access Time in NPB 2.3 EP

Number of Nodes	Read (ms)	Write(ms)
1	345951.3381	0.031074781
2	172975.7257	0.037627966
4	86487.91958	0.075255932

7 Conclusion

In this thesis, we established λ computing environment using the AWG-STAR system and, we design and implement OpenMP application programming interface for the AWG-STAR system. We develop our OpenMP implementation based on the existing OpenMP compiler, OMPi. We modify OMPi and its runtime library to enable parallel computing on the AWG-STAR system. Our OpenMP implementation utilizes the shared memory system provided by the AWG-STAR system for data sharing.

Next, we evaluate the performance of our implementation for the AWG-STAR system by executing the benchmark programs. However, as a result, current AWG-STAR system is not able to achieve comparable performance to existing parallel computing environment , because of insufficient access speed of the shared memory. Our estimation shows that the execution time of the OpenMP application consists mostly of the shared memory access time, and the largest bottleneck is the read performance of the shared memory. It is thought that the cause of low access performance of the shared memory is the PCI bus, because all accesses to the shared memory are performed via the PCI bus.

NTT Photonics Laboratory is currently developing the next version of the AWG-STAR system, and the performance of the memory access will be improved in that version. Therefore, we also estimate the effectiveness of the improvement on the next version of the AWG-STAR system in this thesis. Our estimation shows that the improvement on the next AWG-STAR system will be significantly effective.

As a future work, the performance evaluation of our OpenMP implementation on the next AWG-STAR system is required. Moreover, because our implementation currently does not implement all features of OpenMP, implementation of fully compliant OpenMP is also a future work.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Masayuki Murata at Osaka University for providing me this precious study opportunity as a master's student in his laboratory. His excellent guidance and thoughtful advice have provided an invaluable experience that will help me in my career.

I especially would like to express my deepest appreciation to my supervisor, Associate Professor Ken-ich Baba at Osaka University for his elaborated guidance, considerable encouragement and invaluable discussion that make my research of great achievement and my study life unforgettable.

I am very grateful to Associate Professor Noriyuki Fujimoto at Osaka University and Dr. Akira Okada at NTT Photonics Laboratory for their valuable cooperation in my experiments.

I am also indebted to Associate Professor Naoki Wakamiya and Assistant Professor Shin'ichi Arakawa at Osaka University for their helpful comments.

Finally, I want to say thanks to may friends and colleagues in the Department of Information Networking of the Graduate School of Information Science and Technology of Osaka University for their support. Our conversations and work together have greatly influenced this thesis.

References

[1] H. Nakamoto, K. Baba, and M. Murata, “Shared memory access method for a λ computing environment,” in *Proceedings of IFIP Optical Networks and Technologies Conference (OpNeTec)*, pp. 210–217, Oct 2004.

[2] H. Nakamoto, K. Baba, and M. Murata, “Proposal and evaluation of realization approach for a shared memory system in λ computing environment,” in *Proceedings of the forth International Conference on Optical Internet (COIN2005)*, pp. 90–95, May 2005.

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

[4] E. Taniguchi, “Design and evaluation of shared memory architecture for WDM-based λ computing environment,” Master’s thesis, Graduate School of Information Science and Technology, Osaka University, Feb. 2006.

[5] M. Imoto, E. Taniguchi, K. Baba, and M. Murata, “Implementation and evaluation of MPI library with Globus Toolkit for establishing λ computing environment,” in *Proceedings of 6th Asia-Pacific Symposium on Information and Telecommunication Technologies*, pp. 421–426, November 2005.

[6] OpenMP Architecture Review Board, *OpenMP Application Program Interface Version 2.5*, May 2005. `http://www.openmp.org`.

[7] 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*, vol. 4, pp. 978–979, Sep 2003.

[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*, vol. 3, pp. 264–265, Sep 2001.

[9] 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.

[10] S. J. Min, A. Basumallik, and R. Eigenmann, "Optimizing OpenMP programs on software distributed shared memory systems," *International Journal of Parallel Programming*, vol. 31, no. 3, pp. 225–249, 2003.

[11] M. Imoto, "Design and implementation of synchronization primitives for λ computing environment," Master's thesis, Graduate School of Information Science and Technology, Osaka University, Feb. 2007.

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

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

[14] "NAS Parallel Benchmarks," available at `http://www.nas.nasa.gov/Resources/Software/npb.html`.

[15] "Himeno Benchmark," available at `http://accc.riken.jp/HPC/HimenoBMT/`.

[16] "OpenMP C versions of NPB2.3," available at `http://phase.hpcc.jp/Omni/benchmarks/NPB/`.