Technical Report UEC-IS-2005-3, Version 2005-08-19
Graduate School of Information Systems, The University of Electro-Communications

Mocha Version 0.2: Yet Another Software-DSM System

Kenji Kise†, Takahiro Katagiri†, Hiroki Honda†, and Toshitsugu Yuba†
† Graduate School of Information Systems
The University of Electro-Communications

Abstract

Software distributed shared memory (S-DSM) provides an attractive parallel programming model. We are developing yet another S-DSM system, Mocha. Its design philosophy is to achieve good performance for a large PC cluster and to offer an easy to use S-DSM system. In the current version of Mocha, the first goal is mainly attained by reducing the acknowledgment (Ack) overhead.

We show that an acknowledgment is not necessarily required for each message transmission in the middleware layer. Then, a method to reduce the acknowledgment overhead for a page request is mentioned. The method is implemented in our S-DSM system Mocha Version 0.2.

We report the performance of Mocha Version 0.2 with several benchmark programs on a 32-node PC cluster.

1 Introduction

As an environment for parallel computation, cluster systems using general-purpose personal computers (PC clusters) are becoming popular. Because a PC cluster has no shared memory, the message passing model is used to develop applications in many cases. On the other hand, the shared memory is an attractive programming model for parallel computers. Software distributed shared memory (S-DSM) has been proposed to realize virtual shared memory on a PC cluster as the middleware layer software. Since the idea of S-DSM was proposed[1], several systems[2, 3, 4] have been implemented.

Some S-DSM systems, such as TreadMarks and JIAJIA, adopt the user datagram protocol (UDP) which does not provide reliable communication between the nodes. To detect a communication error and recover from it, therefore, an acknowledgment (Ack) is used for every message transmission. We show that an acknowledgment is not necessarily required for each message transmission in the middleware layer. Then, a method to reduce the acknowledgment overhead for a page request is mentioned. The method is implemented in our S-DSM system Mocha Version 0.2. We report the performance of Mocha Version 0.2 with several benchmark programs on a 32-node PC cluster.

2 Mocha: Yet Another S-DSM System

2.1 Design Philosophies and Implementation

Mocha is an S-DSM system being constructed with two design philosophies: (1) It achieves good performance especially for a PC cluster of many computation nodes. (2) It offers an S-DSM system easy to use as a parallel processing environment.

Mocha is a home based S-DSM, where each page is specified to a node by a user. Mocha is implemented to realize a simple and scalable S-DSM system by rewriting the JIAJIA with scope consistency[5]. The main difference between Mocha and JIAJIA is the reduction of acknowledge overhead in coherence control for obtaining good performance. The followings are points of the code simplification in order to increase readability: (1) JIAJIA's complicated functions for such as home migration and load balancing are removed. (2) Function interfaces are reorganized to make optimization of the source code. (3) The current version of Mocha supports Linux operating system only.

Table 1. The comparison of lines of code.

S-DSM	LOC	eLOC	lLOC
TreadMarks	2,717	2,306	2,029
JIAJIA	2,728	2,304	2,012
Mocha	1,161	981	819

Mocha is written in C and code size is small. The comparison of code lines of three S-DSM systems is summarized in Table 1. The data is obtained with Resource Standard Metrics Version 6.62. A line of code (LOC) is defined as a line that is not a comment or blank line within a source file. Effective line of code (eLOC) is defined as a LOC that is not a stand-alone braces or parenthesis. Logical line of code (lLOC) is defined as a code statement that ends in a semicolon. In every measure, the code lines of Mocha is less than half of JIAJIA and TreadMarks.

Table 2. The comparison of McCabe's cyclomatic complexity[6].

S-DSM	Total	Max	Average
TreadMarks	905	74	6.24
JIAJIA	710	53	5.30
Mocha	342	18	4.12

McCabe's cyclomatic complexity[6] is the most widely used member of a class of static software metrics. It measures the number of independent control paths of a program module. The total cyclomatic complexity (Total), the maximum cyclomatic complexity (Max) and the average cyclomatic complexity (Average) of three S-DSM systems are summarized in Table 2. The data is obtained with Resource Standard Metrics Version 6.62. A function of more than 20 cyclomatic complexity is defined as the complex and high risk module. Table 2 indicates that all functions of Mocha are less than 20 cyclomatic complexity. They are not so complex.

```
 1  #define OP_NULL       100 /* null:                */
 2  #define OP_EXIT       101 /* server: exit         */
 3  #define OP_GETP       110 /* server: get page     */
 4  #define OP_GETPGRANT  111 /* server: getp grant   */
 5  #define OP_DIFF       112 /* server: diff         */
 6  #define OP_DIFFGRANT  113 /* server: diff grant   */
 7  #define OP_BARR       114 /* server: barrier      */
 8  #define OP_BARRGRANT  115 /* server: barr grant   */
 9  #define OP_ACQ        116 /* server: aquire       */
10  #define OP_ACQGRANT   117 /* server: aquiregrant  */
11  #define OP_WAIT       118 /* server: wait         */
12  #define OP_WAITGRANT  119 /* server: wait grant   */
13  #define OP_INV        120 /* server: invalidate   */
14  #define OP_WTNT       121 /* server: w-notice     */
15  #define OP_REL        122 /* server: release      */
16  #define OP_BCAST      123 /* server: broadcast    */
```

Figure 1. The list of message types used by Mocha Version 0.2.

Figure 1 is the list of message types used by Mocha Version 0.2. If a page fault occurs in referencing the non-cached shared memory, the page request (getpage) acquires the page from a node that has the requested page. Mocha Version 0.2 reduces the acknowledgment overhead of messages GETP and GETPGRANT for the page request.

2.2 Omission of Acknowledgment Messages

Mocha is unique by using the concept of acknowledgement omission. Figure 2(a) shows how three messages, Msg-1, Msg-2, and Msg-3, are sent from node 1 to node 2. This is an example of conventional S-DSM communication. When the messages are received, node 2 on the receiving side immediately returns acknowledgment (Ack) messages Ack-1, Ack-2, and Ack-3 respectively.

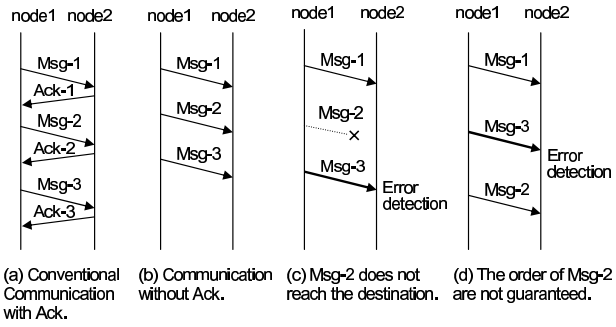

Figure 2. S-DSM Communication with and without acknowledgment.

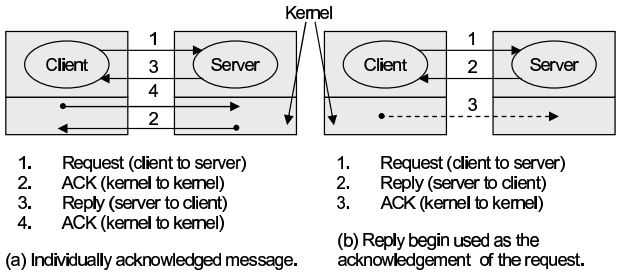

Figure 3. Communication in a client-server model [7].

Figure 2(b) shows a communication without acknowledgment. If all transmitted messages are received without error, the acknowledgment costs can be reduced as shown in (b). Omitting all acknowledgment messages means reducing the general send-receive message count in half. This is expected to enhance the communication performance, especially in applications with frequent messages. In the UDP communication, however, an error occurs by a certain frequency. The examples of communication errors are shown in Figure 2 (c) and (d). Msg-2 does not reach the destination node in (c), and the arriving order of Msg-2 and Msg-3 cannot be guaranteed in (d). Despite of these communication errors, S-DSM systems should be constructed to operate correctly.

As shown in Figure 2(b), Mocha improves S-DSM performance by omitting acknowledgment messages. Note that the idea of the acknowledgment omission is not novel. For example, reference [6] discusses a technique of omitting acknowledgments in a client-server model. Figure 3 is the communication used in this discussion. An acknowledgment for the request can be omitted by using a reply message as its acknowledgment. Also an acknowledgment for the reply shown as the broken line in Figure 3(b) can be omitted depending on the properties of the reply. Mocha is

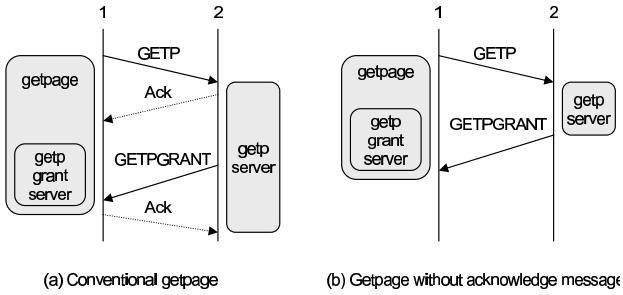

Figure 4. (a) The behavior of a page request. (b) Page request without acknowledgment.

the first S-DSM where the concept of the acknowledgment omission in a client-server model is applied.

2.2.1 Ack Omission for Page Request

Figure 4(a) shows the behavior of a page request. Suppose that node 1 requires a page and node 2 has the page. Node 1 calls the getpage function with the referenced memory address as a argument. The getpage function creates a GETP message corresponding to the page request and transmits it to node 2. On receiving this message, node 2 returns an acknowledgment (Ack) message as a reply. To meet the page request, node 2 then calls the getpserver function. This function packs the necessary page information in a GETPGRANT message and sends it to node 1. On receiving the GETPGRANT message, node 1 calls the corresponding getpgrantserver function. This function stores the received page information in an appropriate memory area and resets the global variable to terminate the getpage function. The getpage function waits in a busy wait state for the page to arrive. When GETPGRANT arrives, the getpage function exits from the busy wait state and continues the application processing.

The getpage function sends a GETP message and waits in a busy wait state. In this kind of page request processing flow, the exit of the getpage function from the busy wait state guarantees that no communication errors occurred in the two messages of GETP and GETPGRANT. If the function did not receive the GETPGRANT message within the timeout limit in the busy wait state, a communication error might have occurred. In this case, the GETP message is sent again and the system waits for the GETPGRANT message. The system should be designed to have no problems, even if the same message of GETP or GETPGRANT arrives several times. Using the GETPGRANT as an acknowledgment of GETP, acknowledgment of GETP or GETPGRANT can be omitted as shown in Figure 4(b).

Figure 5 shows the pseudo-code of the original getpage for which the acknowledgment is not omitted. Line 2 sets

```
1  void getpage(address_t addr){
2    getpwait=1;
3    generate_message(OP_GETP, addr);
4    send_message();
5    while(getpwait); /** busy wait **/
6  }
```

Figure 5. The pseudo-code of the original getpage for which the acknowledgment is not omitted.

```
1  void getpage(address_t addr){
2    getpwait=1;
3    for(i=0; i<GETPAGE_MAX_RETRY; i++){
4      generate_message(OP_GETP, addr);
5      send_message();
6      while(not_timeout() && getpwait); /*busy wait*/
7      if(getpwait==0) break;
8    }
9  }
```

Figure 6. The pseudo-code of the function getpage to be implemented in the Ack omission. The while loop in Line 6 finishes when the global variable getpwait has been reset or a timeout has occurred.

the global variable getpwait, Line 3 generates a page request message, and Line 4 transmits the generated message. When a message corresponding to the request arrives, the received page is stored appropriately and the global variable getpwait is reset. This finishes the while loop in Line 5 and terminates the page request function getpage.

Figure 6 is the pseudo-code of the function getpage to be implemented in the Ack omission. The while-loop in Line 6 finishes when the global variable getpwait has been reset or a timeout event has occurred. If the variable getpwait equals to zero in Line 7, the getpage is terminated because the requested page is assumed to have been received. Otherwise, the for-loop from Line 3 transmits the page request message again.

2.2.2 Implementation of Ack Omission

This section discusses the implementation of the Ack omission.

To identify a message type, the S-DSM system Mocha uses a char-type variable of 8 bits. As summarized in Figure 3, however, not all of the 8 bits are used because the number of message types is only 16. Therefore, the low-order 7 bits are used to indicate an message type and the highest-order bit is used as a flag to indicate whether the message requires acknowledgment. This bit is called the reliable_msg_flag. The system sends an acknowledgment only when reliable_msg_flag is set.

Figure 7 shows the pseudo-code of the message trans-

```
 1  #define MSG_MASK 0x80  /* 10000000 */
 2  void send_one_message(){
 3
 4    if(sendqh->op==OP_GETP ||
 5       sendqh->op==OP_GETPGRANT)
 6      sendqh->op = sendqh->op | MSG_MASK;
 7
 8    for(i=0; i<MAX_RETRY; i++){
 9      ret = sendto(message);
10      if(sendqh->op & MSG_MASK) return;
11
12      while(not_timeout())
13        if(FD_ISSET(fds[serverproc], &fds)!=0){
14          recvfrom(message);
15        }
16    }
17  }
```

Figure 7. The pseudo-code of the message transmission function send_one_message.

mission function send_one_message. If the message is GETP or GETPGRANT (Line 4 and 5), the reliable_msg_flag is set in Line 6. The sendto in Line 9 transmits the message. If the reliable_msg_flag is set, the return in Line 10 can terminate the transmission function immediately and start the next processing because there is no need to wait for the acknowledgment. Otherwise, it is necessary to wait for the acknowledgment message. The while loop from Line 12 to Line 15 waits for the acknowledgment and then terminates the send_one_message function.

The message receiving section checks the reliable_msg_flag of the received message. Like the conventional system, the system transmits an acknowledgment message only when this flag is not set.

We use the code shown in Figure 6 as getpage function to transmit a page request message (GETP). As a result of the parameter adjustment, the timeout interval is set to 40 ms. Therefore, if a page request message is sent again by a communication error, the overhead of 40ms is imposed on a system.

3 Performance Evaluation

This section evaluates the performance of S-DSM Mocha Version 0.2. A Mocha system that does not use the Ack omission is called the Mocha base here.

3.1 Environment

For the evaluation, we use a PC cluster where 32 personal computers are connected with a 48-port gigabit Ethernet switch (HP ProCurve Switch 3400cl-48). Each node is an SMP (symmetric multi-processor) type computer with two processors of Intel Pentium 4 Xeon (2.8 GHz) and 1 GB memory. Although each node is an SMP computer, only one process is assigned per each node. The system software of the cluster is SCore 5.6.1 by PC Cluster Consortium[8] constructed on RedHat Linux 7.3. The execution time of each benchmark program is calculated by the arithmetic mean of five measurements.

3.2 Benchmark Programs

As the benchmark programs, we use N-queens[9], LU (parallel dense blocked LU factorization, no pivoting), SOR (Red-Black Successive Over-Relaxation) and MM (Matrix Multiply). The object code of the benchmark programs is generated using GCC version 2.96 compiler with the optimization option O2.

As a parameter of N-queens, the problem size N=17 and the task allocation size of 8 is used. The elapsed time of the sequential version is 55.0 second.

As a parameter of LU, the matrix size of 1024×1024, and the block size of 8 is used. The element of the matrix is double precision. The elapsed time of the sequential version is 267 second.

As a parameter of SOR, the matrix size of M=4096 and N=4096, iterations=400 is used. The elapsed time of the sequential version is 98.1 second.

As a parameter of MM, the matrix size of 1280×1280 is used. The element of the matrix is double precision. The elapsed time of the sequential version is 9.14 second.

3.3 Performance Comparison

Figures 8 to 11 summarize the results with the number of nodes on the x-axis and the speedup on the y-axis. The speedup is normalized by the elapsed time of JIAJIA single node. There are results for JIAJIA Version 2.2, TreadMarks Version 1.0.3.3 (in Figures 10 and 11 only), and Mocha Version 0.2 using the Ack omission (Mocha) and not omitting the acknowledgment (Mocha base).

The LU benchmark result is shown in Figure 8. The almost optimal speedup is achieved by increasing the number of nodes in every S-DSM system since the traffic in LU is small compared with the calculation.

The N-queens benchmark result is shown in Figure 9. The speedup on the 16-node configuration is 11.0 for JIAJIA and 11.7 for Mocha. The mocha is 6% faster than the JIAJIA on the 16-node configuration. In the case of 32-node configuration, the performance of every S-DSM system drops extremely because of the lock and unlock contention.

The SOR benchmark result is shown in Figure 10. The speedup on the 32-node configuration is 15.5 for JIAJIA and 20.4 for Mocha. The Mocha is 31% faster than the JIAJIA on the 32-node configuration. Some parameters of Mocha are set up so that its performance may become the optimal on the 32-node configuration. Therefore, TreadMarks shows the best speedup on the configurations of less than 32-node.

Table 3. The average number of communication errors on Mocha for page request.

benchmark	2-node (2 CPU)	4-node (4 CPU)	8-node (8 CPU)	16-node (16 CPU)	32-node (32 CPU)
N-queens	0.00	0.00	0.40	1.40	1.80
LU	0.00	0.00	0.40	1.80	4.00
SOR	0.00	0.00	0.20	0.80	1.20
MM	0.00	0.00	0.00	1.60	6.20

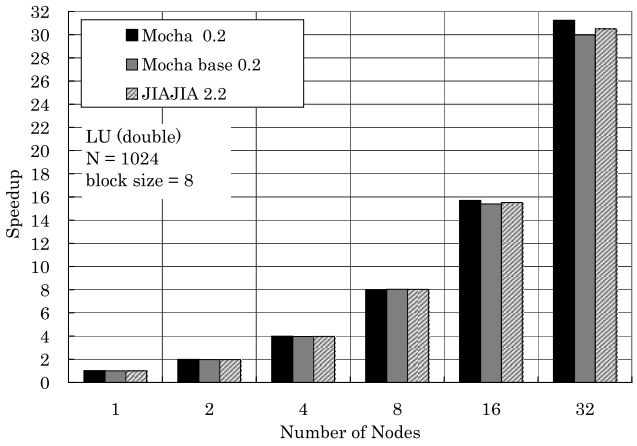

Figure 8. The performance comparison of the S-DSM systems. Benchmark is LU.

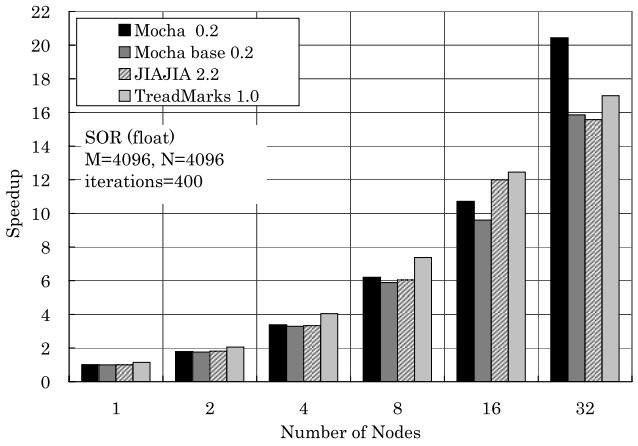

Figure 10. The performance comparison of the S-DSM systems. Benchmark is SOR.

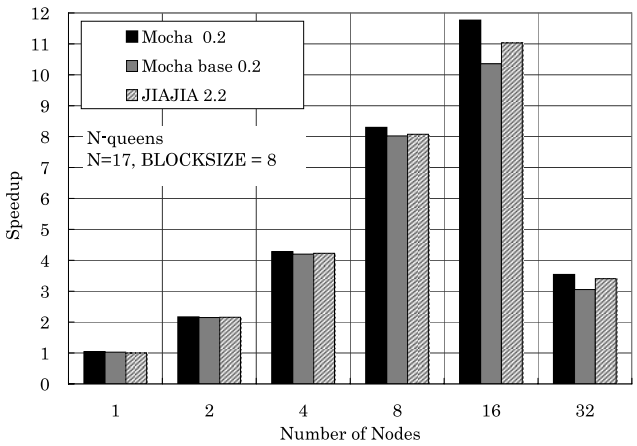

Figure 9. The performance comparison of the S-DSM systems. Benchmark is N-queens.

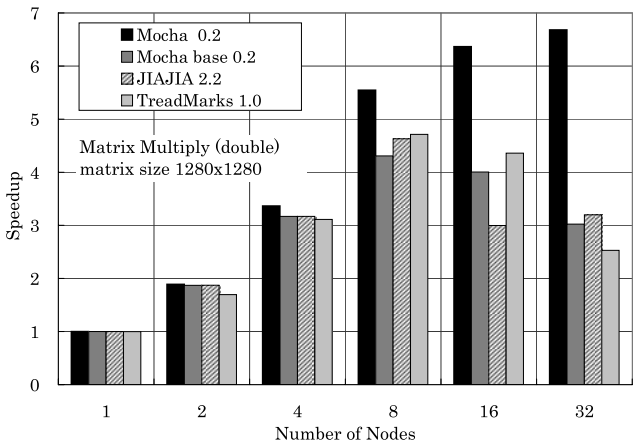

Figure 11. The performance comparison of the S-DSM systems. Benchmark is MM.

The MM benchmark result is shown in Figure 11. In MM execution, the message GETP and GETPGRANT account for most of the elapsed time of all communications. Therefore, the Ack omission for the page request produces remarkable effects. The conventional systems, JIAJIA, TreadMarks and Mocha base, do not show performance improvement where the configuration of more than 8 nodes. Even on the 16-node and 32-node configurations, however, Mocha using the Ack omission keeps performance improvement. Mocha achieves the speedup of 6.68 on the 32-node configuration. Compared with the 16-node Mocha base, the 16-node Mocha achieves a speedup as large as 58%.

Table 3 summarizes the number of errors (the sum of the communication errors at all nodes) for page request by the S-DSM system Mocha, which uses the Ack omission. The data is calculated by the arithmetic mean of five measurements. From Table 3, we see that the frequency of errors is very low.

From the evaluation results in this section, the following conclusion can be obtained. Mocha using the Ack omission achieves high performance in all benchmark programs except for SOR on small node configurations. Especially in a benchmark of high page transfer frequency, such as MM, Mocha achieves a drastic speedup as much as 58% on the 16-node configuration, compared with the conventional communication method of not omitting the acknowledgment.

4 Conclusions

We introduced the S-DSM Mocha Version 0.2. We show that an acknowledgment is not necessarily required for each message transmission in the middleware layer. Then, a method to reduce the acknowledgment overhead for a page request is mentioned. The method is implemented in our S-DSM system Mocha Version 0.2.

We reported the performance of Mocha Version 0.2 with several benchmark programs. From the evaluation results, the following conclusion can be obtained. Mocha using the Ack omission achieves high performance in all benchmark programs except for SOR on small node configurations. Especially in a benchmark of high page transfer frequency, such as MM, Mocha achieves a drastic speedup as much as 58% on the 16-node configuration, compared with the conventional communication method of not omitting the acknowledgment.

Acknowledgement

S-DSM Mocha Version 0.2 is implemented to realize a simple and scalable S-DSM systemby rewriting the JIAJIA. We would like to thank the JIAJIA authors and Center of High Performance Computing, Institute of Computing Technology, Chinese Academy of Sciences. This project is partially supported by Grand-in-Aid for Fundamental Scientific Research B(2) #16300004 from Ministry of Education, Culture, Sports, Science and Technology Japan.

A Installing and Running Mocha

The contents of the README.txt in a distribution file mocha-0.2r2.tgz are as follows.

```
------------------------------------------------
Mocha Version 0.2 Release Note,  2005-08-17
by Kenji KISE, UEC

------------------------------------------------
Mocha is an S-DSM system being constructed with
two design philosophies: (1) It achieves good
performance especially for a PC cluster of many
computation nodes. (2) It offers an S-DSM system
easy to use as a parallel processing environment.

Mocha Version 0.2 is free software. See the file
COPYING for copying permission.

------------------------------------------------
1.0 Package Organization

COPYING    : GNU GENERAL PUBLIC LICENSE Version 2
README.txt : this document file
mocha.c    : Mocha source code
jia.h      : JIAJIA's original header file
main.c     : sample application(Matrix Multiply)
Makefile   : make file to build executable

------------------------------------------------
2.0 Quick Start

------------------------------------------------
2.1 Edit the set_hosts function for your cluster

Please edit the function set_hosts in mocha.c to
adjust your cluster configuration.
The arguments of host_set are node-ID, node name,
and IP address. The first Node-ID must have the
value of 0.

This is the original set_hosts function.

void set_hosts(int *host_max){
  int i;
  *host_max = 32;

  for(i=0; i<32; i++){
    char name[512];
    char addr[512];
    sprintf(name, "comp%02d.score", i);
    sprintf(addr, "192.168.208.%d", 200+i);
    host_set(i, name, addr);
  }

  /*******************************************
  host_set(0, "comp00.score", "192.168.208.200");
  host_set(1, "comp01.score", "192.168.208.201");
  host_set(2, "comp02.score", "192.168.208.202");
  host_set(3, "comp03.score", "192.168.208.203");
  host_set(4, "comp04.score", "192.168.208.204");
  host_set(5, "comp05.score", "192.168.208.205");
  host_set(6, "comp06.score", "192.168.208.206");
```

```
  host_set(7, "comp07.score", "192.168.208.207");
  ******************************************/
}

If you have three hosts of node0.domain,
node1.domain, and node2.domain, the set_hosts
function looks like this.

void set_hosts(int *host_max){
  *host_max = 3;
  host_set(0, "node0.domain", "192.168.208.200");
  host_set(1, "node1.domain", "192.168.208.201");
  host_set(2, "node2.domain", "192.168.208.202");
}

If you have three hosts of node0.domain,
node1.domain, and node2.domain and want to run
two process per one node, the set_hosts function
looks like this.

void set_hosts(int *host_max){
  *host_max = 6;
  host_set(0, "node0.domain", "192.168.208.200");
  host_set(1, "node0.domain", "192.168.208.200");
  host_set(2, "node1.domain", "192.168.208.201");
  host_set(3, "node1.domain", "192.168.208.201");
  host_set(4, "node2.domain", "192.168.208.202");
  host_set(5, "node2.domain", "192.168.208.202");
}

-------------------------------------------------
2.2 Compilation

Just type make will generate an executable a.out.

-------------------------------------------------
2.3 Execution

The -Q option is available to specify the number
of nodes to be used.

This is the command to use two nodes.
% a.out -Q2

This is the command to use four nodes.
% a.out -Q4

-------------------------------------------------
2.4 The Ack omission

The -S option enables the Ack omission.
If you want to use four nodes and run with Ack
omission.
% a.out -Q4 -S

-------------------------------------------------
```

References

[1] Kai Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the International Conference on Parallel Processing (ICPP'88)*, volume 2, pages 94–101, 1988.

[2] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 94 Usenix Conference*, pages 115–131, 1994.

[3] M. Rasit Eskicioglu, T. Anthony Marsland, Weiwu Hu, and Weisong Shi. Evaluation of the JIAJIA Software DSM System on High Performance Computer Architectures. In *Proceedings of the Thirty-second Annual Hawaii International Conference on System Sciences-Volume 8*, page 8012. IEEE Computer Society, 1999.

[4] Benny Wang-Leung Cheung, Cho-Li Wang, and Kai Hwang. Migrating-Home Protocol for Implementing Scope Consistency Model on a Cluster of Workstations. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, 1999.

[5] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 277–287, 1996.

[6] Thomas J. McCabe. A Complexity Measure. *IEEE Transaction on Software Engineering*, SE-2(4):308–320, 1976.

[7] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall International Editions, 1992.

[8] http://www.pccluster.org/index.html.en/. PC Cluster Consortium.

[9] Kenji Kise, Takahiro Katagiri, Hiroki Honda, and Toshitsugu Yuba. Solving the 24-queens Problem using MPI on a PC Cluster. Technical Report UEC-IS-2004-6, Graduate School of Information Systems, The University of Electro-Communications, June 2004.