Using a Cluster as a Memory Resource: A Fast and Large Virtual Memory on MPI

Hiroko Midorikawa, Kazuhiro Saito
Department of Computer and Information Science, Seikei University, Tokyo, Japan
midori@st.seikei.ac.jp, dm08406@cc.seikei.ac.jp

Mitsuhisa Sato, Taisuke Boku
Graduate School of Systems and Information Engineering, University of Tsukuba, Ibaraki, Japan
msato@cs.tsukuba.ac.jp, taisuke@cs.tsukuba.ac.jp

***Abstract*—The 64-bit OS provides ample memory address space that is beneficial for applications using a large amount of data. This paper proposes using a cluster as a memory resource for sequential applications requiring a large amount of memory. This system is an extension of our previously proposed socket-based Distributed Large Memory System (DLM), which offers large virtual memory by using remote memory distributed over nodes in a cluster. The newly designed DLM is based on MPI (Message Passing Interface) to exploit higher portability. MPI-based DLM provides fast and large virtual memory on widely available open clusters managed with an MPI batch queuing system. To access this remote memory, we rely on swap protocols adequate for MPI thread support levels. In experiments, we confirmed that it achieves 493 MB/s and 613 MB/s of remote memory bandwidth with the STREAM benchmark on 2.5 GB/s and 5 GB/s links (Myri-10G x2, x4) and high performance of applications with NPB and Himeno benchmarks. Additionally, this system enables users unfamiliar with parallel programming to use a cluster.**

***Keywords*-virtual memory; software distributed memory; remote paging; cluster computing**

I. Introduction

Today, a 64-bit OS provides ample memory address space that is beneficial for applications using a large amount of data, such as databases and bioinformatics. Even in the current X86_64 architecture, up to 256 terabytes of address space is available. In a traditional virtual memory system, a page is swapped in/out of the local hard disk when the size of program-use memory exceeds that of physical memory on a computer. Because the bandwidth of network devices has become faster than that of hard disks, it is expected that the memory in other remote computers can be used to manage swapped pages efficiently. This paper proposes higher availability of a public cluster as a memory resource for sequential applications requiring very large address space. The authors previously proposed the Distributed Large Memory System (DLM) [1], which is virtual large memory distributed over cluster nodes using remote page swapping. This DLM, designed for sequential programs that need to access larger amounts of data beyond the local physical memory, is user-level software that does not need any special hardware or kernel modification. It runs as a user-level parallel program using distributed memory over a cluster, but it looks like a sequential program execution to users. In contrast to other related research [2]-[4], DLM operates independently from the OS swap system. Thus, setting DLM system parameters, such as the unit size of communication, is completely free from kernel parameters, which in turn exploits the highest performance of the processors and network. We also found that DLM shows more stable behavior than conventional kernel swapping [1].

DLM utilizes basic techniques, such as memory access detection by the SIGSEGV signal and page-based memory management that is traditionally used in software distributed shared memory (SDSM) for parallel processing [5]-[7]. However, DLM is tuned for sequential processing. It is free from an extra overhead to manage memory consistency among multiple calculation processes on different nodes as well as performance degradation caused by a false sharing when using large sizes of memory pages, which are critical for SDSM. Larger page size is efficient for communication, but it is important to balance the memory access locality contained in application programs with the per-page overhead of swapping even in DLM. The larger page size for swapping may cause unnecessary data transmission, which produces extra overhead for applications. We already investigated the influence of various sizes of pages for applications as well as the one-page swap overhead [1]. Moreover, DLM introduced thread technology in communication for efficient usage of multi-core processors.

The original implementation of DLM [1] used a signal interrupted I/O with a TCP socket in a communication thread. Even with such a general communication protocol on 10 Gbps Ethernet, the DLM achieved 380 MB/sec of remote memory bandwidth in the STREAM benchmark [10]. Other application benchmarks also performed higher in DLM than other research schemes using remote direct memory access (RDMA), fast low-level communication protocols, specially designed NIC hardware and/or kernel modification. Both on the 1G bps and 10 Gbps Ethernet, DLM outperformed traditional kernel swap systems using a local hard disk as the swap device.

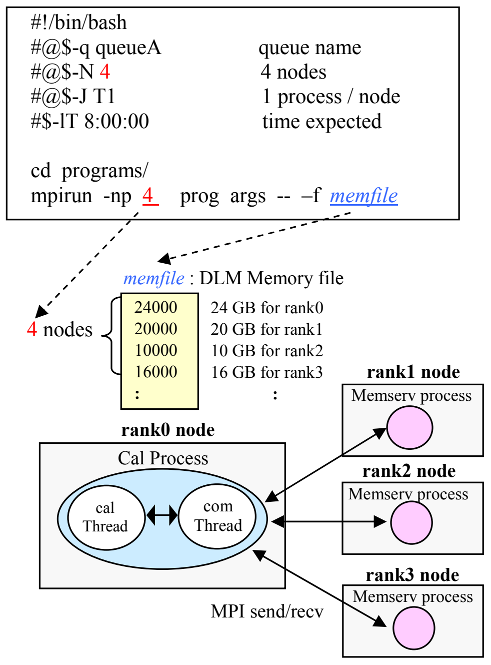
In this paper, the authors propose a new implementation of DLM which uses only the MPI instead of direct use of a socket or a specific protocol such as TCP/IP for inter-node communication. It is designed to be executable on general-purpose open clusters managed by the MPI batch queuing system, through which nodes for users are allocated dynamically by the batch scheduling system at each run.

MPI is a highly abstracted description for communication which is independent from the actual communication fabrics or protocols on lower layers in communication. Thus, it provides higher portability and availability for a wider range of cluster systems with various kinds of communication links and protocols. Recently, many MPI libraries have been directly implemented on high-performance communication fabrics, such as Myrinet and Infiniband, and these libraries also support the newest communication utilities, such as a network bonding to accelerate performance. Consequently, the first benefit of MPI-based DLM is high portability and high performance.

The second benefit is that it opens clusters for general users who have had no opportunity to use a cluster because they have no parallel application programs. Widely available open clusters are often managed with the MPI batch queuing system. DLM enables users who have sequential applications using large amounts of data to access a public cluster to take advantage of the large amount of available memory. Moreover, users do not need any knowledge of parallel programming in MPI and they are spared the great expense of purchasing a high-end, memory-rich general computer or dedicated software-based SMP machine such as vSMP [8]. The MPI-based DLM is available for clusters with batch systems as well as interactive systems. This feature is valuable for most actual HPC clusters in operation.

The user-level implementations of a memory server accessing remote memory have high portability but a lack of transparency compared to kernel-level implementations, in which the kernel swap device is changed implicitly. JumboMem [9], a user-level memory server, takes another approach to provide transparency. It provides a dynamic linkable shared object library and replaces memory-related functions, such as malloc, with newly implemented functions utilizing remote memory in the JumboMem address space. Although it is implemented at the user level, it is completely transparent to applications. However, this mechanism only functions on dynamically allocated memory data, meaning the static array which is common for large data usage cannot be distributed across a cluster. Moreover, such memory-related functions are used in system-level software with the processes fork, file access buffer handling, and mmap. Entire replacement of memory related functions to those for JumboMem address space causes undesirable side effects affecting performance and safety.

In DLM, a user can specify the data to be deployed over remote memories, allowing unspecified data to remain in local memory. DLM provides two user supports: a DLM library for hand translation and a DLM compiler for automatic translation. Users can easily translate their sequential C programs to DLM programs with DLM library functions. The DLM compiler automatically translates original static declarations of data arrays into dynamic data allocations and original accesses to data arrays into pointer-based accesses. The translated DLM program still appears as a sequential program, so users can implicitly access the remote memory without any knowledge of parallel programming in MPI.


```
#!/bin/bash
#@$-q queueA          queue name
#@$-N 4               4 nodes
#@$-J T1              1 process / node
#$-lT 8:00:00         time expected

cd  programs/
mpirun  -np  4    prog  args  --  –f  memfile
```

Figure 1. Example of the DLM runtime system, memfile and batch shell

As a result of using the STREAM benchmark [10], the remote memory bandwidth of MPI-based DLM achieves 493 MB/sec for Myrinet10G/2-NIC-bonding, and 613 MB/sec for Myrinet10G/4-NIC-bonding. In the NPB [12] and Himeno benchmarks [14], MPI-based DLM shows a slight slowdown when using larger memory rather than local physical memory. When only 4%-6% of the total data used in an application resides in local memory, most of the applications execute 1.7 - 4.0 times slower than in **normal execution**, which means 100% of the data resides in local memory. All experiments are done on the public open cluster with MPI batch queuing system on the T2K Open Supercomputer (University of Tokyo, Hitachi cluster HA8000.)

II. Overview of MPI-based DLM system

A. DLM on MPI batch queuing system

The runtime system of MPI-based DLM, DLM-MPI, is the same as that of the socket-based DLM, DLM-socket, as shown in Fig. 1. This figure also shows a typical execution shell for batch queuing systems. The DLM initialization function called by a user program on the local node (rank0) creates a communication thread (comThread), which communicates with both the calculation thread (calThread) and memory server processes. User program codes run only on the calThread on the rank0 node. Users usually specify the number of nodes according to the amount of remote memory they want to use, in a one-process-per-node fashion, even if a node has multi-core CPUs. Users can specify a memfile as an option, which includes a list of the available sizes of memory on each node, as shown in Fig. 1. Without memfile, the same amount of memory is allocated on each node of the DLM system by default. All communications between nodes use the MPI library functions instead of socket reads/writes with interrupted I/O using the SIGIO signal. The communications between comThread and calThread on rank0-node use SIGUSR signals and status flags.

The user-specified large data, called **DLM data**, are allocated not only in local node (rank0) memory but also in remote memory on memory server nodes (rank1-rankN) when the amount of local node memory is not sufficient for the DLM data. The users can allocate DLM data by dlm_alloc( ), which is similar to malloc(). When calThread accesses a data area allocated in a remote node memory, it is detected by SIGSEGV, and calThread requires the memory server process who possesses the data to send the data to calThread. After receiving the requested data, other data are sent back to the memory server node to maintain the amount of data in local memory under the pre-specified constant value in memfile. This data swapping between local and remote nodes is done in the unit **DLM pagesize**, which is a multiple of the pagesize defined in the OS kernel.

At the end of the program, function dlm_shutdown() is called when calThread finishes the program code execution, and then dlm_shutdown() calls MPI_Finalize(), which finalizes the remote and local processes. The end of the programs is detected by the batch queuing system.

B. Application interface for the DLM System

Users can easily translate their sequential C programs to C programs for the DLM system. There are two ways to translate user programs to DLM.

The first is hand translation using three DLM functions, dlm_startup( ), dlm_alloc( ) and dlm_shutdown( ), as shown in Fig. 2. Fig. 2 is an example of a DLM program, in which a 2-D matrix and vector are multiplied using DLM data arrays. In the first and last parts of the original program, dlm_startup() and dlm_shutdown(), respectively, must be inserted. The large amounts of data in programs can be allocated by dlm_alloc() as DLM data, which is then distributed over the remote memory on cluster nodes. Hand-translated programs are compiled by mpicc (MPI compiler) with the DLM-MPI library, libdlmmpi, as follows.

```
mpicc prog.c -o prog -ldlmmpi
```

Another way is to use the DLM-MPI compiler, which automatically translates original static declarations of data arrays into dynamic data allocations and original accesses to data arrays into pointer-based accesses, as shown in Fig. 2. Fig. 3 is a DLM program for the DLM compiler that is equivalent to that shown in Fig. 2. It is almost the same as an ordinary sequential C program except malloc( ) is replaced with dlm_alloc( ) and **dlm** is attached to static data declaration statements. This compiler translates DLM programs in a slightly different way from the old one [1], because the new dlm_startup( ) functions differently from the former dlm_init() used in DLM-socket.

In DLM programs, only user-specified DLM data can be placed in remote memory when the local memory does not have sufficient space. DLM-unspecified data are guaranteed to be in local memory. Although the programs shown in Fig. 2 and Fig. 3 appear to be sequential programs, users can implicitly access the remote memory without any knowledge of parallel programming in MPI.

```
//DLM Program example :  Matrix Vector Multiply
#include <stdio.h>
#define N 16384 // example: mem 2048MB + 32KB
double (*a)[N];  //  a pointer for 2-D array
double  *x, *y;  //  pointers for 1-D arrays

main(int argc, char *argv[])
{  int i,j;
   double temp;
   dlm_startup(&argc, &argv);

   a = (double (*)[N]) dlm_alloc(  N * N * sizeof(double) );
   x = (double *) dlm_alloc( N * sizeof(double) );
   y = (double *) dlm_alloc( N * sizeof(double) );

   for(i = 0; i < N; i++)      //  Initialize array a
         for(j = 0; j <N; j++) a[i][j] = i;

   for(i = 0; i < N; i++) x[i] = i;     // Initialize array x
   for(i = 0; i < N; i++){           // a[N][N]*x[N]=y[N]
         temp = 0;
         for(j = 0; j <N; j++)  temp += a[i][j]*x[j];
         y[i] = temp;
   }
   dlm_shutdown( );
   return 0;
}
```

Figure 2.   The sample of DLM program for hand-translating

```
//DLM Program example :  Matrix Vector Multiply
#include <stdio.h>
#define N 16384 // example: mem 2048MB + 32KB
dlm double a[N][N], x[N], y[N];  // DLM data declare

main(int argc, char *argv[])
{   int i,j;
   double temp;
   for(i = 0; i < N; i++)       //  Initialize array a
         for(j = 0; j <N; j++) a[i][j] = i;

   for(i = 0; i < N; i++) x[i] = i;     // Initialize array x
   for(i = 0; i < N; i++){           // a[N][N]*x[N]=y[N]
         temp = 0;
          for(j = 0; j <N; j++)  temp += a[i][j]*x[j];
         y[i] = temp;
   }
   return 0;
}
```

Figure 3.   The sample of DLM program for DLM-MPI compiler

III. Implementation of the MPI-based DLM System

A. DLM System Initialization and Finalization

In the former DLM-socket for interactive cluster systems, the dlm_init( ) function forks remote processes on memory server nodes according to the list of hostnames in the user-specified hostsfile [1]. In the new DLM-MPI, this procedure is eliminated and the initialization function, dlm_startup( ), only uses the MPI processes which are already forked on the nodes allocated by the MPI batch queuing system. In DLM-MPI, all processes execute the same program in SPMD fashion. Thus, all MPI processes call dlm_startup() first, in which MPI_Init(), MPI_Comm_size(), and MPI_Comm_rank() are called, to get their rank number and number of processes running, N.

In contrast to the former initialization function, dlm_init(), designed for DLM-socket [1], processes on memory server nodes never return from the dlm_startup() function while the user program code is running on rank0-node. After initialization of a DLM page table, the processes for the memory servers on rank1-rank(N-1) nodes do an infinite loop of waiting for messages from the client process on rank0 node. The processes finished after receiving a program-end message from the client process.

The main process, calThread, on rank0 node first parses the command line options and broadcasts parameters to the memory server processes. Then, it sets signal block masks and creates a communication thread, comThread, on the local node. A SIGSEGV handler is registered for calThread, and calThread initializes a DLM page table.

The finalization function, dlm_shutdown(), is only called by the calThread on rank0. It sends a program-end message to all remote memory server processes. After exchanging program-end messages, MPI_Finalize() is called. The memory server processes on the remote nodes exchange this final end message through comThread on rank0 and finalizes with MPI_Finalize() in the dlm_startup() function.

DLM data are managed with DLM pages and a DLM page table. Each entry of a DLM page table has a rank id indicating where a page is allocated, a page top address and an end address where data is allocated, and the remaining memory size in the page. In DLM system initialization, the cal process allocates a DLM page table with sufficient page entries which can afford the total amount of memory size denoted in the DLM memfile. The DLM page table itself is not swapped out to memory servers.

B. Priority of the DLM data allocation node

When DLM data are allocated, the necessary number of DLM pages is mapped in local memory on rank0 node first, up to the specified memory size in the first line of the DLM memfile in Fig. 1. If local memory is not sufficient for mapping all DLM data, the remaining data are mapped in the remote memory of the next rank-numbered node. If the server node memory for the current allocation is not sufficient, the remains of the DLM data are always allocated in the next rank-numbered node in ascending order.

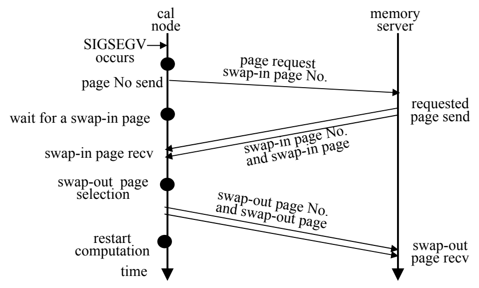


Figure 4. Serial protocol

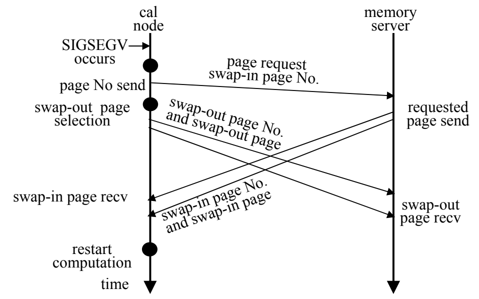


Figure 5. Cross protocol

C. DLM Page Swapping Protocol

When a user program accesses an unmapped non-local DLM data area, a SIGSEGV signal handler is invoked to retrieve the required DLM page from the memory server. The handler also swaps a local DLM page with a remote DLM page if necessary. DLM-MPI provides the memory resource management for memory free as well as memory allocation. Thus, if some memory free calls, dlm_free(), are done before the SIGSEGV occurs, page swapping is not always necessary. It is possible to map the requested page in local memory without swapping out the other page.

Basically, page swapping consists of three procedures: (1) sending a page request; (2) receiving a swap-in page; and (3) sending the swap-out page. The swapping protocol is classified into two types according to the different order of procedures: (1)(2)(3) for the serial protocol in Fig. 4, and (1)(3)(2) for the cross protocol in Fig. 5. The former DLM-socket employs the cross protocol, and the cal process immediately sends a page request accompanied with a swap-out page to reduce the number of sends. The cal process adopts a very simple page swap selection algorithm: a round-robin page selection over address (called ARR in this paper) to send a page request immediately for rapid return from the SIGSEGV handler. A memory server reads the header containing a requested page number first and immediately

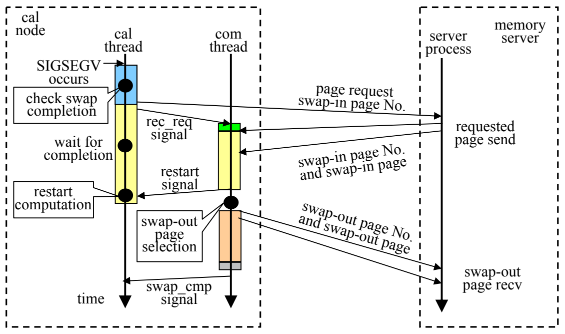


Figure 6. Aggressive serial protocol for MPI_THREAD_SERIALIZED

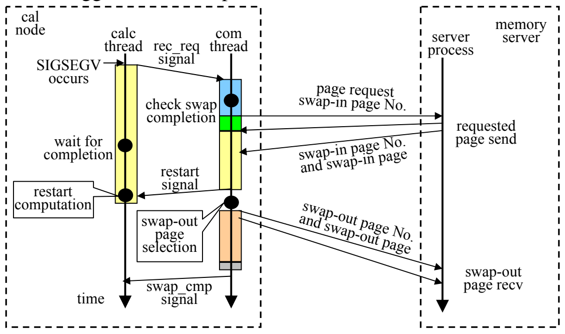


Figure 7. Aggressive serial protocol for MPI_THREAD_FUNNELED

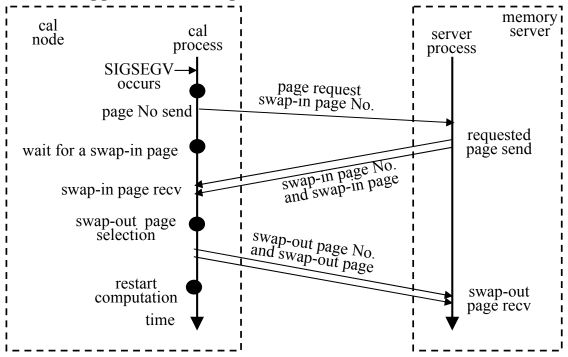


Figure 8. Conservative serial protocol for MPI_THREAD_SINGLE

sends back the requested page to the cal process. Then, the server reads the swap-out page from the cal process.

The page replacement algorithm is a critical factor for the performance of remote memory paging. Therefore, DLM-MPI employs an aggressive serial protocol (AS protocol), shown in Fig. 7, to utilize more time for sophisticated page replacement. In the AS protocol, calThread sends a requested page number directly to a memory server process immediately after SIGSEGV occurs in order to eliminate the communication overhead. The memory server process sends the requested page as soon as possible after receiving the requested page number. Then, comThread receives the page, maps the page and notifies calThread. As calThread returns from the SIGSEGV handler and restarts the program, comThread selects the swap-out page and sends it to the

Table I
OPEN CLUSTER WITH BATCH QUEUING SYSTEM

	T2K Open Supercomputer, HA8000
Machine	HITACHI HA8000-tc/ RS425
CPU	AMD QuadCore Opteron 8356(2.3GHz) 4 CPU/ node
Memory	32 GB/ node (936 nodes), 128 GB/ node 16nodes)
Cache	L2 : 2 MB/ CPU (512 KB/ Core), L3 : 2 MB/ CPU
Network	Myrinet-10G x 4, (5 GB/s full-duplex) Myrinet-10G x 2, (2.5 GB/s full-duplex)
OS	Linux kernel 2.6.18-53.1.19.el5 x86_64
Compiler	gcc version 4.1.2 20070626 cc: Hitachi Optimizing C mpicc for 1.2.7
MPI Lib	MPICH-MX (MPI 1.2)

memory server. The AS protocol introduces concurrency in the serial procedure for efficient page swapping. While the former page swapping procedure is executing, calThread is suspended to send a new page request as it waits for completion.

To implement DLM with MPI, the thread support level of the MPI library is an important factor for selecting a protocol. DLM has no limitation for MPI library selection, but it is necessary to select the appropriate protocol for the thread support level of MPI provided by the user's system. MPI defines four thread support levels: 1) MPI_THREAD_SINGLE, 2) MPI_THREAD_FUNNELED, 3) MPI_THREAD_SERIALIZED, and 4) MPI_THREAD_MULTIPLE. The highest level, 4), is not yet widely available, so only the lower three levels are realistic selections. In serial protocols, the AS protocol in Fig. 6 is expected for fast swapping, but it requires the level 3). For the two other lower levels, another AS protocol in Fig. 7 and the conservative serial protocol (CS protocol) in Fig. 8 can be used. The CS protocol can be used for any level of MPI, so it has the highest portability; however, it may require the longest time for swapping. In this paper, AS and CS protocols are used for single thread applications. For multi-thread applications, Fig. 6 and Fig. 7 are applicable with the level 4) and the level 2) respectively.

IV. MPI-BASED DLM PERFORMANCE ON AN OPEN CLUSTER

A. Experimental Setting

In this paper, all experiments are conducted on a public open cluster with MPI batch queuing system; that is, the T2K Open Supercomputer shown in Table I. In this environment, DLM-MPI uses two types of nodes with different network performance: 5 GB/s links (Myri-10G[11] x 4, BONDING=4) and 2.5 GB/s links (Myri-10G x 2, BONDING=2). According to the performance evaluation [1], the 1 MB DLM pagesize is used for the measurements in this paper. In this experiment, an original simple ARR

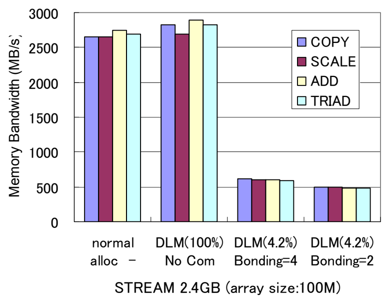


Figure 9. Local and remote memory bandwidth (STREAM 100M elements Array Size (2.4 GB))

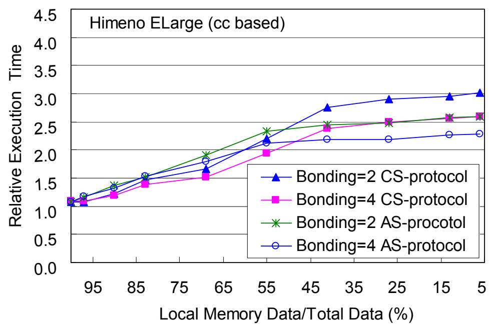


Figure 10. Himeno benchmark relative exec time for AS and CS protocols

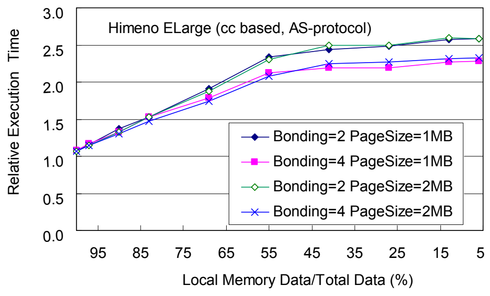


Figure 11. Himeno benchmark relative exec time for different DLM pagesizes

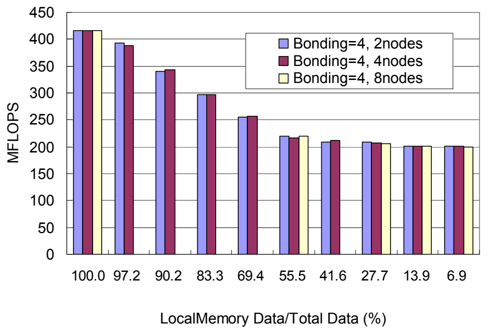


Figure 12. Himeno benchmark performance on multiple nodes, AS protocol

page replacement scheme is used. Also, the following are defined: **local memory ratio** is the ratio of the size of data in local memory to the total size of data used in a program; **bonding=2** represents network bonding with 2 Myri10G NICs; and **bonding=4** is that with 4 Myri10G NICs.

B. STREAM benchmark: Remote memory bandwidth

The remote memory access bandwidth in the DLM-MPI system is measured using the STREAM benchmark[10].It outputs sustainable memory bandwidths in application levels for each kernel. STREAM originally used static array declarations, but a modified dynamic data allocation version is used to measure local memory bandwidths. We chose an array of 100 M elements (2.4 GB) and an array of 1 G elements (24 GB) for comparison between the remote and local memory bandwidth. The local memory bandwidth is approximately 2.7 GB/s, which is almost the same as the memory bandwidth of DLM with 100% of data in local memory, as shown in Fig. 9 and Table II. The remote memory bandwidth achieves 493 MB/sec for bonding=2, and 613 MB/sec for bonding=4 on 2.4 GB arrays. Table II also shows the remote memory bandwidths of larger arrays, 96 GB and 144 GB using 4 nodes (25 GB/node, bonding=2) and 8 nodes (20 GB/node, bonding=4) on DLM.

C. Himeno benchmark

This benchmark measures the speed of major loops for solving Poisson's equation [14]. It uses multiple loops of iterations and is known as a heavy memory access program. Although it outputs performance in MFLOPS, the values are translated into relative execution times. The ELARGE size, 513x513x1025, 15 GB, is used here.

Fig. 10 shows the relative execution times for normal program execution, which uses only local memory. The horizontal axis represents the local memory ratio and shows the results for different swap protocols and network link speeds. Even if the local memory ratio is only 6.9%, which means 93.1% of the data are on the remote memory, the relative execution times are 2 to 3 times longer compared to that of the normal execution (local memory ratio 100%). At a 6.9% local memory ratio, the CS protocol is 17% slower than the AS protocol in bonding=2 and is 19% slower in bonding=4.

Fig. 11 shows the case of using 2 MB DLM pagesize for two kinds of network, bonding=2 and bonding=4. There are no significant differences in performance between the 1 MB and 2 MB pagesize in this application. The network with bonding=4 has 13% higher performance at the maximum than the network with bonding=2 when the communication traffic becomes heavy at the lower local memory ratio.

Fig. 12 shows absolute performances in MFLOPS when 15 GB of the array data are distributed over multiple nodes (2 to 8). Fig. 12 shows the same performance among the number of nodes used for the DLM when the local memory ratio is the same. The performance independence from the

Table II
LOCAL AND REMOTE MEMORY BANDWIDTH ON STREAM BENCHMARK

Memory Size	Array Size	Network Bonding	Local mem Ratio %	COPY MB/sec	SCALE MB/sec	ADD MB/sec	TRIAD MB/sec	Memory Configuration	Memory Bandwidth Type
2.4 GB	100 M	-	100.00	2657.95	2655.63	2746.98	2695.07	normal malloc	Local mem bandwidth
2.4 GB	100 M	4	100.00	2829.35	2692.67	2893.66	2820.04	2.4 GB-0.1 GB 2 nodes	DLM Local mem bandwidth
2.4 GB	100 M	4	4.20	613.81	607.38	596.70	593.98	0.1 GB-2.4 GB 2 nodes	DLM Remote mem bandwidth
2.4 GB	100 M	2	4.20	493.70	492.42	483.29	481.99	0.1 GB-2.4 GB 2 nodes	DLM Remote mem bandwidth
24 GB	1 G	-	100.00	2660.21	2660.27	2750.64	2696.38	normal malloc	Local mem bandwidth
24 GB	1 G	4	4.35	593.79	578.25	579.82	579.42	1 GB-24GB 2 nodes	DLM Remote mem bandwidth
24 GB	1 G	2	4.35	479.03	466.90	470.25	469.86	1 GB-24GB 2 nodes	DLM Remote mem bandwidth
96 GB	4 G	2	27.30	443.17	495.43	464.78	463.87	25 GB x 4 nodes	DLM Remote mem bandwidth
144 GB	6 G	4	14.56	524.88	520.75	514.33	519.05	20 GB x 8 nodes	DLM Remote mem bandwidth

number of memory servers is shown in other applications.

The experiments for larger data, the 1025x1025x2049 float and double arrays, which are not defined in the original benchmark, are conducted using multiple nodes with 20 GB memory/node with bonding=4. The performances of the float array (112 GB) and double array (241 GB) achieve 179.8 MFLOPS and 88.77 MFLOPS, respectively, with a local memory rate of 8.19%.

D. Time analysis of AS swap protocol

The time component in the AS protocol of the 1 MB DLM page is investigated using the Himeno Benchmark. Fig. 13 and Fig. 14 show the time components of the AS swap protocol in the calThread and comThread for Himeno MIDDLE size for two local memory ratios (high and low) and three different communication types. The communication types are 10 Gbps TCP/IP socket (IP on Myrinet, Myri-10G 1 NIC), 2.5 GB/s MPICH-MX (bonding=2), 5 GB/s MPICH-MX (bonding=4). In socket-10 Gbps, DLM uses only MPI_Init and MPI_Finalize at the first and the last part, and other communications between the cal process and memory servers are done by the TCP/IP socket on the Myri-10G on the same T2K cluster system. The T2K does not provide a NIC bonding facility for socket use.

The color of the time components in these figures corresponds to the color of procedure components shown in Fig. 6. The lower violet-colored component, sigwait (previous), in the calThread in Fig. 13 represents the suspending time from sending a new page request while the previous swap-out page sending is executing. This waiting time becomes longer when the local memory ratio is small in the same communication type, because a program causes more swapping at a low local memory ratio and the possibility of overlap of the previous swap-out and new swap becomes higher. For a similar reason, the waiting time becomes longer in lower-speed communication types with the same local memory ratio.

The next yellow-colored time component, sigwait (swap-in), in the calThread of Fig. 13 corresponds to the swap-

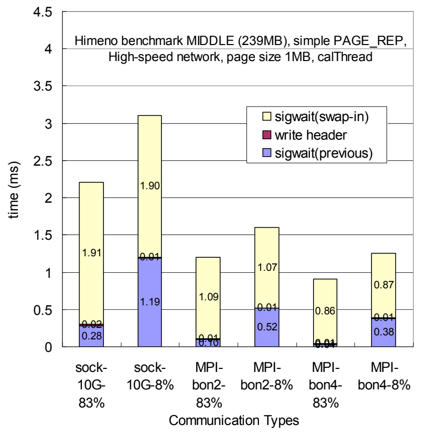


Figure 13. HimenoM time component of AS protocol in calThread

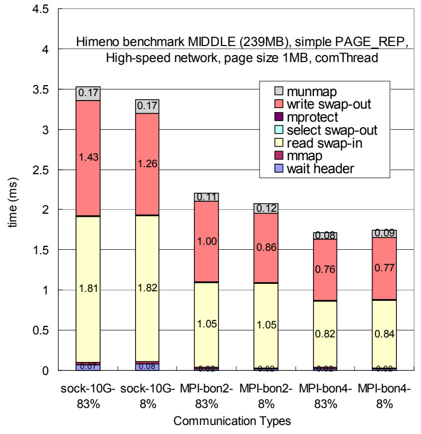


Figure 14. HimenoM time component of AS protocol in comThread

in page reading time from the memory server. It also corresponds to the yellow-colored time component, read swap-in, in the comThread of Fig. 14 . As shown in the calThread in Fig. 13, the page read (swap-in) time in MPI-MX(bonding=2) is 59% - 56% of those in the socket-10 Gbps, and the time in MPI-MX (bonding=4) is 78% - 82%

of that in MPI-MX(bonding=2). These results show that 4-NIC bonding is not very efficient for MPI-MX.

The estimated 1 MB DLM page read performance values from these time components are 526 MB/s for socket-10 Gbps, 935 MB/s for MPICH-MX bonding=2 and 1149 MB/s for MPICH-MX bonding=4. According to a simple loopback test program using MPI_Recv and MPI_Send, the 1 MB message read time from the communication link is 1.73 ms for bonding=2 and 1.57 ms for bonding=4. In a comparison of these values, the swap-in times in the DLM calThread are slightly smaller because part of the DLM page has already arrived when the header read (wait header in the comThread) finishes.

E. *NAS parallel benchmark*

The C version of sequential programs, IS, CG, MG, FT and BT, from NPB2.3-omni-C [13] are also used for DLM evaluation. The maximum data size class in this NPB, CLASS-C, is used. The newest release, NPB3.3, provides larger data classes, CLASS D and CLASS E, but the original NPB programs are written in Fortran, except IS. We use NPB2.3-omni-C [13] for CG, MG, FT and BT with CLASS C and NPB3.3 for IS with CLASS D, which is the maximum size for IS. NPB is a benchmark for measuring the performance of parallel processing, so it is not necessarily suitable for DLM evaluation, even though it is well-known and familiar to HPC researchers and has various applications with different memory access patterns.

Table III shows the size of the data, the type of data allocation, dynamic or static, and the number of data in static and dynamic allocation in each original program. In CG and BT, the number of iterations is shrunk to reduce the evaluation times. It also shows the normal execution times of programs where all data are allocated dynamically in the local memory.

Fig.15 - 21 show the relative execution times to the normal execution time. The horizontal axis represents the local memory ratio. NPB measures and outputs the time of a core calculation, such as iterative calculation over a large data array, but a program usually has a data initialization and a data verification in the first and last parts of the program, respectively. In this experiment, the dominant large data in original NPB programs are specified as DLM data for distribution across the memory server nodes. The DLM data size is almost the same as the whole data size used in the programs, because it is dominant. The local memory ratio is calculated based on the whole size of the program data, but all of the DLM data are not always used in the core calculation. Some of the benchmark performance does not slow down in the core calculation, even if the local memory ratio becomes smaller, as shown in IS of Fig. 15. Also, the usage rate of DLM data in the core calculation depends on the applications.The ratio of execution times of a core part to the other parts depends on both the applications and the number of iterations of the core part. Some applications consume more time for their verification than for their core calculation. Therefore, in this experiment, we focus on the core calculation time and memory accesses even though these are influenced by the ratio of DLM data used in a core calculation.

IS is an integer sort which requires three integer arrays of size $2^{27}$for CLASS-C, 1.6 GB and $2^{31}$for CLASS-D, 51 GB. IS-C uses a relatively small data size while IS-D requires a very large data size, but both show the same tendency in performance. Fig. 15 shows no slowdown and a flat performance over the local memory ratio, which is higher than 40.7%. In IS, the data used in the core calculation are roughly half of the whole DLM data, and they are already in local memory. Thus, no swaps are needed. The relative execution time increases to 1.45 (174 K swap counts) when the local memory ratio is between 30.5% and 4.06%. It becomes 4.34 (2,475 K swap counts) when the local ratio becomes 2.0%.

CG-C requires 14 different types and sizes of arrays, 1.2 GB in total. The relative execution time is approximately 1.1, when the local memory ratio is higher than 40%. CG has the same characteristic as IS; it uses only partial DLM data in the core calculation, which causes no swaps at a higher local memory rate. However, once the local memory ratio becomes lower than 40%, the execution time increases 3 to 4 times, as shown in Fig. 16. The swap count is 55 K at 36% and 111 K at 9% for each local memory ratio. There is not much performance difference between network speed of bonding=2 and bonding=4. The CS protocol measurement values at higher local memory ratio include some noise. The CS protocol is 20% slower than the AS protocol at 4.17% of the local memory ratio.

MG-C is a 3-D multi-grid algorithm for 512x512x512 used to obtain an approximate solution to the discrete Poisson problem. All data are dynamically allocated and a huge number of malloc or dlm_alloc functions are called, for example, 974,107 times in CLASS-C. The relative execution times increase according to the local memory ratio, from 1.19 at 100% to 3.55 and 3.8 at 5.7%, as shown in Fig. 17. The swap count gradually increases as the local memory ratio decreases; 134K at 73%, 204K at 44% and 310K at 15%. The network with bonding=4 has a 19% higher performance than that of bonding=2 at the maximum.

FT-C is a 3-D FFT calculation for 512x512x512, which requires three 3-D arrays of complex numbers, 7 GB. The iterative calculation has three different directions of data accesses with discrete gaps. Swap counts are 64K at 92% and 130K at 76% - 46%. FT uses most of the DLM data in the core calculation. The relative execution time in the AS protocol is 1.16 at 100% and 1.69 (bonding=4) and 1.86 (bonding=2) at 19.38% for each local memory ratio. Although Fig. 18 shows a relatively small performance degradation in comparison with MG and CG, the perfor-

Table III
NAS PARALLEL BENCHMARKS USED IN THE EXPERIMENT

	CG-C	IS-C	MG-C	BT-C	FT-C	IS-D
Size Parameter	15000	2**27	512**3	162**3	512**3	2**31
DLM Data Size (GB)	1.156	1.610	3.581	5.080	7.014	51.538
Data Type in original	static	static	malloc	static	static	static
Num of Data and malloc	14	4	974,107	16	8	4
Iteration Shrink Ratio	10/75	-	-	10/200	-	-
Normal Exec Time (malloc) cc (sec)	87.42	42.27	217.11	226.66	573.47	1809.03

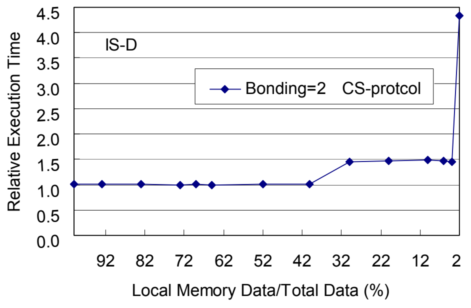


Figure 15. NPB IS-D (NPB3.3) relative execution time

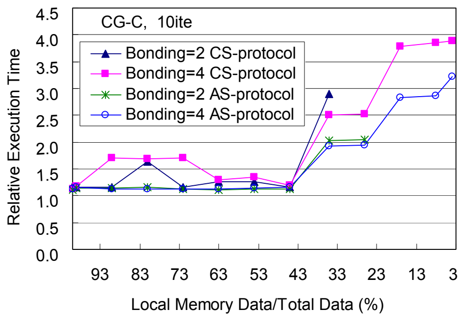


Figure 16. NPB CG-C relative execution time

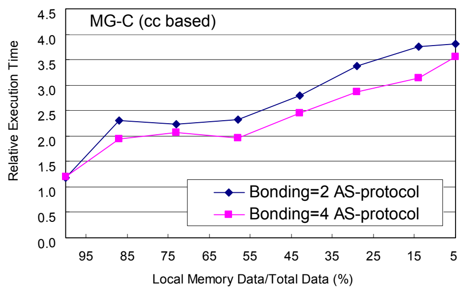


Figure 17. NPB MG-C relative execution time, AS protocol

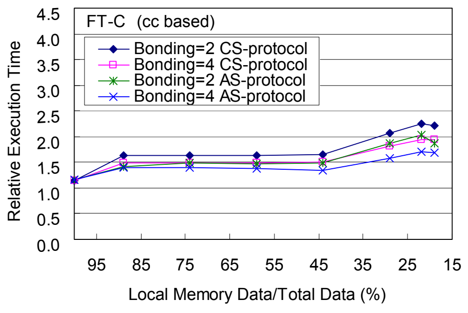


Figure 18. NPB FT-C relative execution time

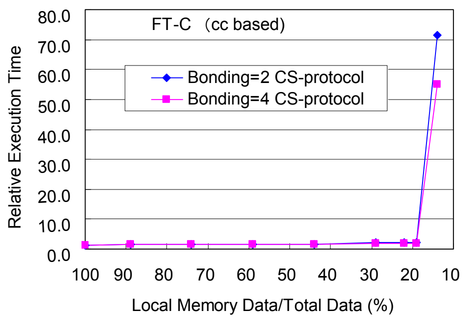


Figure 19. NPB FT-C relative execution time, AS protocol

mance drastically drops at 14.89% of the local memory ratio, at which point the execution time becomes 54.1 (bonding=4) and 71.2 (bonding=2). This causes a large swap count, so the working set of access data in the iterative procedure may increase beyond the size of local memory.

BT-C is a program solving 3-D compressible Navier-Stokes equations for simulated CFD applications. It uses 162x162x162, 5 GB of a block-tridiagonal of 5x5 blocks, and solves sequentially along each dimension. As with FT-C, the performance of BT-C does not degrade much when the local memory ratio is greater than 60%. The relative execution time is 1.18 on 100% and 1.94 (bonding=4) and 2.14 (bonding=2) on 61.84%, as shown in Fig. 20. When the local memory ratio is less than 60%, the relative execution time increases drastically to 31.0 (bonding=4) and 47.46 (bonding=4), as shown in Fig. 21.

This may be one of the reasons for the slowdown. In such applications, once the working set of memory access in one iteration exceeds the size of local memory, the number of swaps grows rapidly and causes prohibitively long execution time.

The BT with smaller CLASS A and B shows a similar tendency to that of CLASS C. The smaller class performance degradation ratio is bigger than CLASS C. In BT-A, the relative execution time grows to 518 for bonding=2 when the local memory ratio is 51% in the CS protocol. In the case

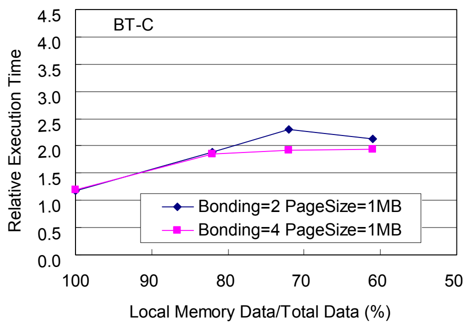


Figure 20. NPB BT-C relative execution time, AS protocol

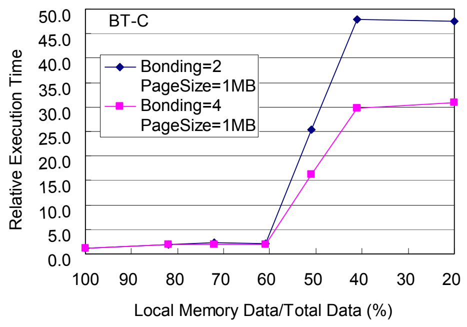


Figure 21. NPB BT-C relative execution time at lower local memory ratio, AS protocols

of using a currently developing page replacement algorithm, the relative execution time is reduced to 280 for bonding=2 and 220 for bonding=4. However, swap counts are reduced from 3,771K to 2,114K.

BT uses 16 multi-dimensional arrays from 3-D to 6-D and accesses many array elements in each iteration, so it is different from the Himeno benchmark, which also uses seven 4D-arrays but accesses only about 30 elements of an array in each iteration. In the Himeno benchmark, all DLM data are used in core calculations and the memory access is high, but the working set in one iteration is relatively small compared to the whole data.

V. Conclusion

This paper proposed using a cluster as a memory resource for sequential applications requiring very large address space. The MPI-based Distributed Large Memory System (DLM) provides a fast and large virtual memory on widely available open clusters managed with a batch queuing system. The proposed system exploits higher portability, which in turn enables the use of higher performance communication links. Swap protocols for each MPI thread support level are also proposed and investigated. The results revealed that some benchmark programs show relatively small performance degradation, 1.7 to 4.0 times slower than normal execution, even if only 4%-6% of the total data is in the local memory. It was also shown that an application program, Himeno benchmark, works on 241 GB distributed memory using 12 nodes.

We are now developing an efficient page replacement algorithm for remote page swapping in DLM. According to the initial evaluation, it gains a speedup of 10% to 40% in some applications.

Future work includes the following. 1) A detailed analysis on memory access locality in applications, and the relationship between a working set and page size. 2) Development of an effective and low-cost user-level page replacement algorithm. 3) Investigation of efficient swap protocols in cross/serial types, MPI thread-support level, socket versus MPI, and the design for multi-thread user programs.

Acknowledgment

The authors wish to thank Prof. Munenori Kai at Seikei University for his support of our research and Mr. Hiroya Matsuba at the University of Tokyo for his technical advice on the use of the T2K supercomputer and MPICH-MX. This work was supported by JSPS KAKENHI 21500062.

References

[1] H.Midorikawa, M.Kurokawa, R.Himeno, M.Sato, ”DLM: A Distributed Large Memory System Using Remote Memory Swapping over Cluster Nodes”, IEEE Cluster Computing, pp.268-273, 2008
[2] S. Liang, R. Noronha, and D. K. Panda, Swapping to Remote Memory over InfiniBand: An Approach using a High Performance Network Block Device, IEEE Cluster Computing, 2005
[3] Pavel Mache, Linux Network Block Device, (1997) http://www.xss.co.at/linux/NBD/ http://nbd.sourceforge.net/
[4] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel, ”Nswap: A network swapping module for Linux clusters,” in Euro-Par’03 [Online]. http://www:cs:swarthmore:edu/ newhall/europar03:pdf
[5] Michael Stumm and Songnian Zhou, ”Algorithms Implementing Distributed Shared Memory”, IEEE Computer, pp.54-64, 1990
[6] Bill Nitzberg and Virginia Lo, ”Distributed Shared Memory: A Survey of Issues and Algorithms”, IEEE Computer, pp.52-60, 1991
[7] H.Midorikawa, U.Ohashi,H.Iizuka, ”The Design and Implementation of User-Level Software Distributed Shared Memory System: SMS - Implicit Binding Entry Consistency Model - ”, Proc. of IEEE Pacific Rim Conf. on Communications Computers and Signal Processing, pp.299-302, 2001
[8] (2009) ScaleMP web site [Online] http://www.scalemp.com/
[9] Scott Pakin and Greg Johnson, ”Performance Analysis of a User-level Memory Server”, IEEE International Conference on Cluster Computing, pp.249-258, 2007
[10] (2009) STREAM Benchmark web site [Online] http://www.cs.virginia.edu/stream/ref.html
[11] (2009) Myri-10G, Myricom web site [Online]. http://www.myri.com/
[12] (2009) NPB (NAS Parallel Benchmarks) web site [Online]. http://www.nas.nasa.gov/Resources/Software/npb.html
[13] (2009) NPB2.3-omni-C web size [Online]. http://phase.hpcc.jp/Omni/benchmarks/NPB/index.html
[14] (2009) Himeno Benchmark web site [Online]. http://accc.riken.jp/HPC/HimenoBMT/index.html