A C compiler for Large Data Sequential Processing using Remote Memory

Shiyo Yoshimura, Hiroko Midorikawa
Graduate School of Science and Technology, Seikei University, Tokyo, Japan
E-mail:dm106231@cc.seikei.ac.jp, midori@st.seikei.ac.jp

Abstract

Prevailing 64bit-OS enables us to use a large memory address space in computer programming general. However, the actual physical memory becomes the limitation in utilizing it fully. When a program requires more memory than available physical memory in a computer, a traditional virtual memory system performs the page swap between a local hard disk and physical memory. Here, with the recent development in high-speed network, remote-memory access via networks becomes faster than accessing a local hard disk. We built the Distributed Large Memory System (DLM) to access vast remote memories in networks. The DLM is designed as a user-level software for high portability. The DLM provides a very large virtual memory using remote memories distributed over cluster nodes. This paper proposes a newly designed C compiler for the DLM. It provides an easy programming interface to use the abundant memory of the DLM with existing sequential programs, instead of developing parallel programs.

1. Introduction

In recent years, we can use a large memory address space from the programs running on 64bit-OS. However, the physical memory of a computer becomes a limitation for such memory use. Ordinary, when a program requires more memory than physical memory in a computer, memory pages are swapped in/out a hard disk. However, accessing remote memories in network-connected computers becomes faster than accessing local hard disks, because of the recent development of high-speed networks.

Researchers who simulate a scientific numerical problem usually develop a sequential program first and validate it with small scale problems. Then, they simulate large scale problems. To deal with large scale problems, the programs sometime require more memory than a local memory in one computer. In such cases, the sequential programs have to be converted to parallel programs that utilize large memories on multiple nodes in a cluster. However, developing a parallel version of the programs is not an easy task for people who are not familiar with parallel programming. Moreover, it will impose them to pay extra costs for debugging and validating a parallel version of programs. Additionally, not all sequential program models can be converted to parallel ones because of the nature of original simulation models. In these cases, some users prefer to run existing sequential programs using large memory distributed over cluster nodes, even if the execution time becomes slower than the time of a parallel version of the programs.

Because of these reasons, the Distributed Large Memory (DLM) System [1], which was a virtual large memory system distributed over cluster nodes, was developed. The DLM system is designed for sequential programs that need a large amount of data beyond the local physical memory. It was reported that the DLM system using remote memory achieved higher performance of program executions compared to the kernel swap system with a local hard disk. In this paper, we propose a DLM compiler, which enables us to use rich memory distributed over multiple nodes of a cluster with existing sequential programs. It also eliminates the extra cost for developing parallel programs.

To use remote memory for a sequential program, there are two ways: kernel-level implementations and user-level implementations.

Kernel-level implementations have limited portability because they ordinary require a special hardware and/or kernel modifications. Kernel-level implementations usually replace a traditional swap device, a hard disk, with remote memory. It was reported that changing a swap device to remote memory often caused performance degradation in page swapping [1]. One of the reasons is that the swap system of a traditional OS is usually tuned to a hard disk. Another reason is unstable behavior in remote communication under the lack of memory when a swap daemon is initiated. However, kernel implementation gives complete transparency to a user.

It means there is no need to change programs for using a remote memory.

User-level implementations are designed independent from OS kernel and swap daemon, and run as a user-level program. They have high portability, but it imposes users to suit their programs to the APIs provided by the implementations. User-level implementations generally achieve a high communication performance than kernel-level implementations, because they are executed without initiating swap daemon.

The JumboMem [2], the one of the user-level implementations, improved user transparency. It was achieved by providing a dynamic linkable shared object library and replacing memory-related functions, such as malloc, with newly implemented JumboMem functions, which utilize remote memory in the JumboMem address space. It realizes perfect user transparency. There is neither need to modify user programs, nor to recompile existing binary programs. However, there are two problems. First is that JumboMem only supports dynamic memory allocation functions, and it does not support static array declarations, which are commonly used in many numerical programs. The second problem is that all malloc functions are replaced with JumboMem functions which use remote memories. It sometimes causes significant problems, e.g. I/O buffer memory for file access might be allocated in remote memory, which must be always allocated in local memory.

The DLM system is a user-level software to achieve high portability and performance. It resolves two problems occurred in JumboMem. First, the DLM provides the API that can support both dynamic memory allocation and static array data. The second, users can distinguish two types of data, data allocated in both remote memory and local memory and data always exist in local memory. To improve low user transparency in user-level implementation, the DLM compiler is proposed.

2. The DLM System

2.1. The DLM System Overview

Fig.1 shows an overview of DLM system. The DLM system runs a sequential user program at Cal Host node. The DLM system automatically allocates data in remote memory of Mem Server node, when a user program needs more memory than the size of a local memory. When the user program accesses certain data in remote memory, the DLM system swaps in the page which contains the data to Cal Host node, and swaps out the other pages to Mem Server node. The unit of swapping is *DLM page size* that is a multiple of the OS page size. General protocol, TCP/IP or MPI, is used in the DLM system. So it can run on a wide variety of high-speed communication media like 10Gbps Ethernet, Infiniband, and Myri10G. It looks like a sequential program execution for users, but actually it runs as a user-level parallel program using distributed memory over a cluster.

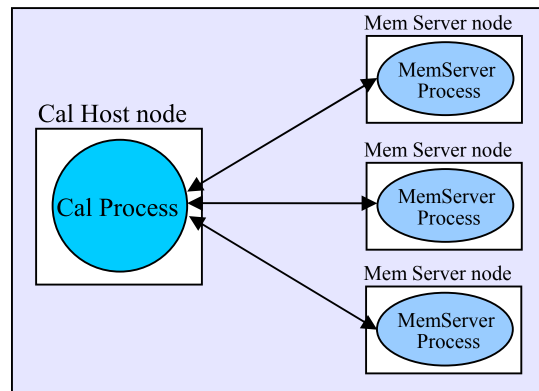


Fig. 1 Example of the DLM system

2.2. Program Interface of the DLM System

The proposed interface is designed to alleviate a user's load of rewriting programs. The knowledge of parallel programming is unnecessary to use the DLM system.

As previously mentioned, the DLM supports two types of memory allocations, both a static array declaration and a dynamic memory allocation for remote memory. Users can specify large data, called *DLM data*, which are allocated not only in local memory but also in remote memory when the amount of local memory is not sufficient for the *data*.

Generally in C programs, global variables and static variables are allocated in static data area, local variables are allocated in stack memory area, and dynamically allocated data by *malloc* function are allocated in heap memory area. However, in the DLM system, the *DLM data* are always allocated in heap memory area of local/remote memory irrespective of a global static array or a local variable. This enables us to use a large amount of data, not limited by local memory size or a compiler.

The DLM programs are identical to ordinary C programs, except attaching *dlm* before *the DLM data* declarations. The *dlm* is introduced as one of the storage specifiers in C grammar, like *extern* and *static*. DLM's API has 2 features as follows:

- A user can distinguish the *DLM data* from ordinary data using *dlm* specifier. The first line in Fig.2 represents an ordinary data declaration, which allocates data in local memory only. The second line represents the *DLM data* declaration, which allocates data in a local memory and/or a remote memory on memory servers.
- A user can specify 2 types of the *DLM data*. The first line in Fig.3 represents a static array declaration of the *DLM data*. The second line represents a dynamic memory allocation of the *DLM data*.

```
int a[100][10] ; →allocate in a local memory only
dlm int b[1000][1000] ; → allocates in local
                          memory and/or
                          remote memory
```

Fig. 2 Ordinary data and the *DLM data* declarations

```
dlm double c[1000][1000] ;
                    →static array declaration
c =(double(*))dlm_alloc(1000*1000*sizeof(double));
                    →dynamic memory allocation
```

Fig. 3 Static and dynamic *DLM data* allocation

Fig.4 shows a DLM sample program, which calculates median values of multiple array data. The array *a*(①) at the beginning of the program includes 10 integer arrays, which has 10G elements in each array. The main function randomly assigns integer numbers to the array *a*. In the *median* function, local array variable *b*(②) is created, and one of the integer array of *a* is copied to *b*. Then, *qsort* function sorts the array *b* and the *median* function returns the median of the array *b*. The essence of the program is preserving the original order of the array *a* by copying one of the array *a* to the array *b* at every time when *median* is called.

The size of local array variable *b* at ② in Fig.4 is 40GB. In an ordinary C program, it is allocated in a stack memory, so the program execution is usually restricted by the size of local memory and the size of stack memory area. On the other hand, the *DLM data* are always allocated to heap memory area in local memory and/or remote memory, even if they are declared as local variables in programs. So the program using large data specified as the *DLM data* is hardly limited in an actual execution by the available local memory size or a kernel memory layout.

```
#include <stdio.h>
#include <stdlib.h>
#include <dlmm.h>
#define NUM 10
#define LENGTH (10*(1L << 30)) // 10GB
dlm int a[NUM][LENGTH]; // 400GB ①

int median(long int num) {
  dlm  int b[LENGTH]; // 40GB   ②
  long int j, ans;
  for ( j = 0; j < LENGTH; j++)
    ③  b[j] = a[num][j]; ④
  qsort(b,LENGTH,sizeof(int),compare_int); ⑤
  ans = b[LENGTH/2]; ⑥
  return ans;
}

int main ( int argc, char *argv[])
{
  long int i, j;
  for ( i = 0; i < NUM; i++)
    for ( j = 0; j < LENGTH; j++)
          ⑦ a[i][j] = rand();
  for ( i = 0; i < NUM; i++)
    printf("median[%d] = %d\n", i, median(i));
  return 0;
}
```

Fig. 4 A sample of the DLM program

3. Structure of the DLM Compiler

The DLM compiler is designed to have 2 parts for high portability. The first part includes a general C preprocessor and a DLM translator, *dlmpp*. The second part is a general C compiler.

Fig.5 shows procedures in the DLM compiler. First, the DLM compiler converts a DLM program including *dlm* specifiers to an ordinary C program by *dlmpp* translator. It performs 3 tasks: (1) insert DLM library functions, (2) transform the *DLM data* declarations to ordinary C pointers, and (3) rename the variables accessing to the DLM data by considering their scopes.

The next, the DLM compiler creates an execution program from the C program by a gcc compiler with the *dlm* library. Fig.6 shows an example of compile command, *dlmc*, which makes an execution program (prg1) from a DLM program (prg.c) with *dlm* library.

The DLM system is also available by manually rewriting programs using *dlm* functions and executing them with an ordinary C compiler. However, the DLM compiler gives a great benefit for users by saving the time bothering to rewrite the programs.

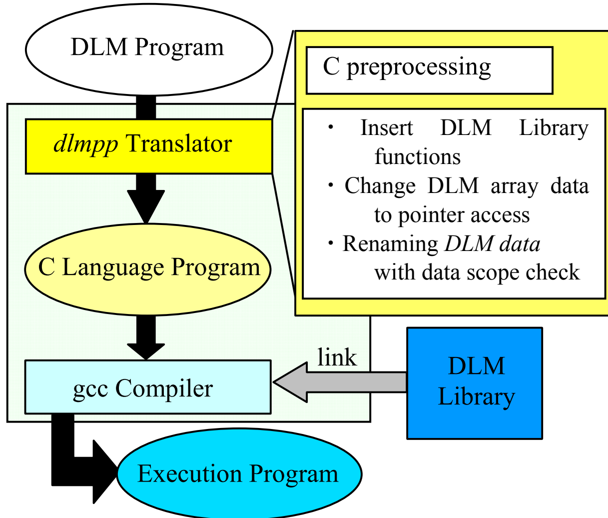


Fig.5 Components of DLM compiler

```
dlmc   prg.c   –o   prg1   -ldlm
```

Fig. 6 A DLM compile command example

The *dlmpp* translator translates Fig.4 to Fig.7 (header files introduced by C preprocessor are omitted). Following procedures are performed when translating DLM programs to C language programs by the DLM compiler.

- Insert the dlm_startup function after variable declarations of the main function. The dlm_startup activates the DLM system. It creates memory server processes at memory server nodes and setups communication between memory server processes and a calculation process. (See Fig.7 - ⑨)
- Insert the dlm_shutdown function before all return statements in the main function. The dlm_shutdown terminates the DLM system. It finishes the communication with memory servers and shut down memory server processes. (See Fig.7 - ⑫,⑬)
- Change DLM array data declarations to pointer base declarations. (See Fig.4 - ①,② → Fig.7 - ①,②). Insert dlm_alloc functions of the *DLM data* after the dlm_startup function(See Fig.7 – ③,⑩). Rename all of the *DLM data* variables to *_dlm_variable name_block number* (See Fig.4 - ③,④,⑤,⑥,⑦ → Fig.7 - ④,⑤,⑥,⑦, ⑪).
- Insert dlm_free function at the last parts of all blocks and functions, if *dlm* variables are declared in the blocks and/or functions. (See Fig.7 - ⑧)

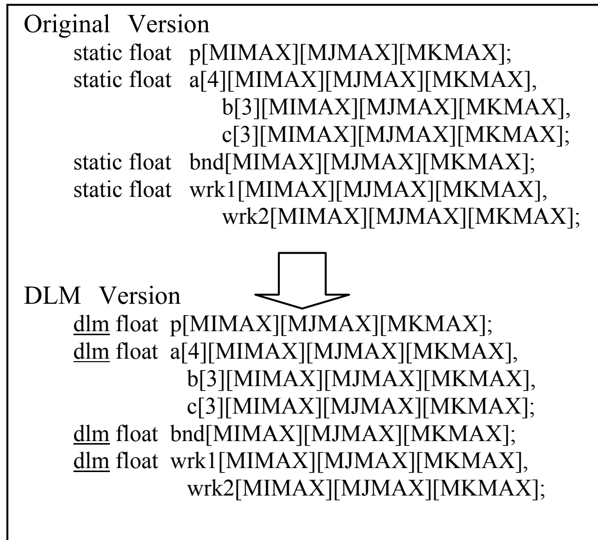
```
              . . . . . . . . . . . . .
int (*__dlm_a_0)[(10*(1L<<30))]; ①


int median(long int num) {
 int (*__dlm_b_1); ②
 long int j, ans;
 __dlm_b_1 = ( int (*))dlm_alloc((10*(1L<<30))*sizeof( int )); ③
 for ( j = 0; j < (10*(1L << 30)); j++)
      ④__dlm_b_1[j] = __dlm_a_0[num][j]; ⑤
 qsort(__dlm_b_1,(10*(1L << 30)),sizeof(int),compare_int); ⑥
   ans = __dlm_b_1[(10*(1L << 30))/2]; ⑦
   dlm_free(__dlm_b_1); ⑧
   return ans;
}
int main ( int argc, char *argv[])
{
   long int i, j;
   dlm_startup(&argc, &argv); ⑨
__dlm_a_0 = ⑩
( int(*)[(10*(1L<<30))])dlm_alloc(10*(10*(1L<<30))*sizeof( int ));
   for ( i = 0; i < 10; i++)
      for ( j = 0; j < (10*(1L << 30)); j++)
             ⑪__dlm_a_0[i][j] = rand();
   for ( i = 0; i < 10; i++)
 printf("median[%d] = %d\n", i, median(i));
      dlm_shutdown(); ⑫
      return 0;
      dlm_shutdown(); ⑬
}
```

Fig. 7 A C program converted by *dlmpp* translator

4. DLM programs for DLM System

In this section, we show some examples of rewriting actual programs for the DLM system. Basically, what users have to do for converting existing sequential programs to DLM programs is only 2 or 3 things: (1) inserting *dlm.h* at the top of the program, (2) attaching *dlm* before large data declarations, and (3) replacing malloc with dlm_alloc.

Fig. 8 shows all modified parts for Himeno Benchmark [3]. In the original program, large data are declared as global static variables. Since the source file is only one, *static* is unnecessary. Here only 4 modifications are required (Fig.8).



Original Version

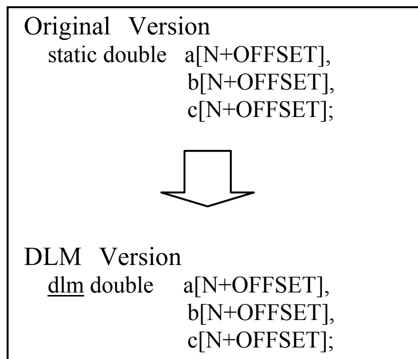
```
static float  p[MIMAX][MJMAX][MKMAX];
static float  a[4][MIMAX][MJMAX][MKMAX],
                 b[3][MIMAX][MJMAX][MKMAX],
                 c[3][MIMAX][MJMAX][MKMAX];
static float  bnd[MIMAX][MJMAX][MKMAX];
static float  wrk1[MIMAX][MJMAX][MKMAX],
                 wrk2[MIMAX][MJMAX][MKMAX];
```

DLM Version

```
dlm float  p[MIMAX][MJMAX][MKMAX];
dlm float  a[4][MIMAX][MJMAX][MKMAX],
              b[3][MIMAX][MJMAX][MKMAX],
              c[3][MIMAX][MJMAX][MKMAX];
dlm float  bnd[MIMAX][MJMAX][MKMAX];
dlm float  wrk1[MIMAX][MJMAX][MKMAX],
              wrk2[MIMAX][MJMAX][MKMAX];
```

Fig. 8 Modification of Himeno Benchmark

Fig.9 shows the case of STREAM Benchmark [4], where only one modification is required.



Original Version

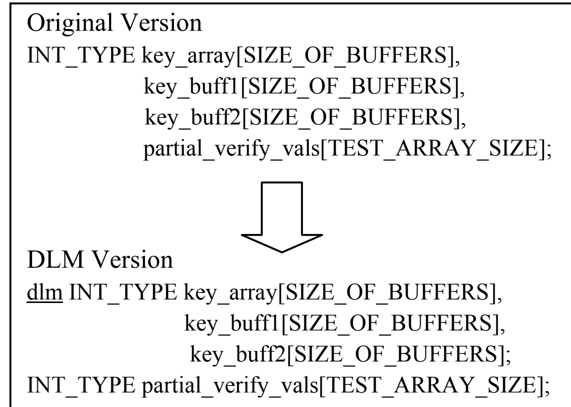
```
static double  a[N+OFFSET],
               b[N+OFFSET],
               c[N+OFFSET];
```

DLM Version

```
dlm double  a[N+OFFSET],
            b[N+OFFSET],
            c[N+OFFSET];
```

Fig. 9 Modification of STREM Benchmark

Fig.10 shows the modified parts of IS in the NAS Parallel Benchmark [5]. In the IS, the first 3 arrays are declared as *DLM data*, while the last one is declared as normal because its size is very small.



Original Version

```
INT_TYPE key_array[SIZE_OF_BUFFERS],
         key_buff1[SIZE_OF_BUFFERS],
         key_buff2[SIZE_OF_BUFFERS],
         partial_verify_vals[TEST_ARRAY_SIZE];
```

DLM Version

```
dlm INT_TYPE key_array[SIZE_OF_BUFFERS],
             key_buff1[SIZE_OF_BUFFERS],
             key_buff2[SIZE_OF_BUFFERS];
INT_TYPE partial_verify_vals[TEST_ARRAY_SIZE];
```

Fig. 10 Modification of NPB IS.B

In numerical simulations, static array declarations are often used for large data. Without the DLM compiler, users have to translate original static declarations of data arrays into dynamic data allocations (dlm_alloc) and convert all the original data-array accesses into pointer-based accesses. The DLM compiler reduces the rewriting costs of programs by users to its minimum. Only attaching *dlm* to the existing sequential programs is sufficient for using a remote memory for large data.

5. The DLM performance

This section shows one of benchmark performances using the DLM system. The experiments are conducted on a public open cluster of the T2K Open Supercomputer HA8000 cluster with the MPI batch queuing system [6] (Table 1).

Table 1 Environment of Experiment

	T2K Open Supercomputer, HA8000
Machine	HITACHI HA8000-tc/RS425
CPU	AMD QuadCore Opteron 8356(2.3GHz) 4CPU/ node
Memory	32GB/node (936 nodes), 128GB/node (16nodes)
Cache	L2 : 2MB/CPU (512KB/Core), L3 : 2MB/CPU
Network	Myrinet-10G x 4, (40Gbps) bonding4 Myrinet-10G x 2, (20Gbps) bonding2
OS	Linux kernel 2.6.18-53.1.19.el5 x86_64
Compiler	gcc version 4.1.2 20070626, Hitachi Optimizing C mpicc for 1.2.7
MPI Lib	MPICH-MX (MPI 1.2)

The benchmark used here is Himeno Benchmark [3], which measures the speed of major loops for solving Poisson's equation. It uses multiple loops of iterations and is known as a heavy memory access program.

Fig. 11 shows the performance of the Himeno Benchmark of ELARGE size, 513x513x1025 float array (15GiB). The benchmark outputs the performance in MFLOPS, but the values here are translated into the relative execution time.

The horizontal axis of Fig. 11 represents the ratio of *local data size/total data size* (*local memory ratio* L) used in the benchmark program. Note that (1-L)% of the total data resides in remote memory, while the remaining L% in local memory. The performance with the DLM system is measured by limiting the size of available local memory in Cal Process node. The vertical axis of Fig.11 represents a relative execution time in the DLM system compared with an ordinary execution time without the DLM. In an ordinary execution, a program uses only local memory, where the *local memory ratio* is 100%. The performance of ELARGE in an ordinary execution is 415 MFLOPS.

Fig.11 shows the case of using 1MB *DLM page size* for bonding=4 (40Gbps) network. According to Fig. 11, even if the *local memory ratio* becomes 6.9%, which means 93.1% of the total data resides in remote memory, the benchmark execution time becomes 2.3 times longer than the ordinary execution time.

We also evaluate the newly created larger data version of the Himeno Benchmark with XLARGE, 1025x1025x2049 float array (112GiB). 6 nodes and 40Gbps network is used in this experiment, where 20GiB local memory in each node is used for execution. In this benchmark, a normal execution using only local memory is impossible because physical memory size in one node is limited 32GiB. Here the absolute performance is 179.4 MFLOPS, where the *local memory ratio* is 17.4%.

Moreover, a larger data size version, XLARGE-d with 1025x1025x2049 double array (2 25GiB) is created for evaluation. This experiment uses 12 nodes. The size of local memory for each node and network bandwidth used here are the same as the previous experiment. The absolute performance becomes 88.8 MFLOPS, where the *local memory ratio* is 8.1%.

These experiments show that existing sequential programs can be used for large-scale problems beyond the limitation of local physical memory size.

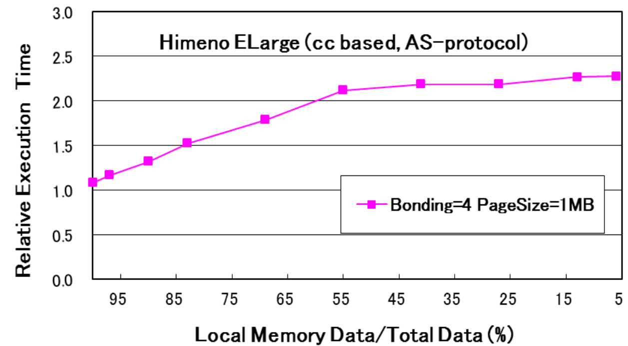


Fig.11 Performance of DLM with Himeno Benchmark

6. Conclusion

The DLM system makes easy to use remote memories in a cluster without parallel programming. In the previous DLM system, users had to rewrite existing programs by inserting dlm functions, such as dlm_startup, dlm_alloc and dlm_shutdown, and modifying array data to pointer-based data. The proposed DLM compiler relieves users from these annoying program rewrites. With this compiler, users are only required to attach *dlm* before the *DLM data*.

The proposed compiler will make people to process large data easily by accessing a cluster with sequential programs, and it will save time for rewriting a sequential program considerably.

References

[1] H. Midorikawa, K. Saito, M. Sato, T.Boku "Using a Cluster as a Memory Resource: A Fast and Virtual Memory on MPI", *Proc. of IEEE International Conference on Cluster Computing*, IEEE cluster 2009, 2009-09, page 1-10.

[2] S. Pakin, G. Johnson, "Performance Analysis of a User-level Memory Server", *Proc. of IEEE International Conference on Cluster Computing*, IEEE cluster 2007, 2007-09, pp. 249-258.

[3] Himeno Benchmark website [Online], http://accc.riken.jp/HPC/HimenoBMT/downloadtop.html, 2011-5

[4] STREAM Benchmark website [Online], http://www.cs.virginia.edu/stream/ref.html, 2011-5

[5] NPB (NAS Parallel Benchmarks) website [Online], http://www.nas.nasa.gov/Resources/Software/npb.html, 2011-5

[6] T2K Open Supercomputer, HA8000, http://www.cc.u-tokyo.ac.jp/service/ha8000/, 2011-5