Design and Evaluation of Page-swap Protocols for a Remote Memory Paging System

Hikari Oura, Hiroko Midorikawa, Kenji Kitagawa, Munenori Kai
Graduate School of Science and Technology
Seikei University
Tokyo, Japan
dm166201@cc.seikei.ac.jp, midori@st.seikei.ac.jp

***Abstract*— A remote memory paging system called a distributed large memory (DLM) has been developed, which uses remote-node memories in a cluster, as the main memory extension of a local node. The DLM is available for out-of-core processing, i.e., processing of large-size data that exceeds the main memory capacity in the local node. By using the DLM and memory servers, it is possible to run multi-thread programs written in OpenMP and pthread for large-scale problems on a computation node whose main memory capacity is smaller than the problem data size. A page swap protocol and its implementation are significant factors in the performance of remote memory paging systems. A current version of the DLM has a bottle-neck in efficient page swapping because all communication managements between memory servers and the local computation node are allocated to one system thread. This paper proposes two new page swap protocols and implementations by introducing another new system thread to alleviate this situation. They are evaluated by a micro-benchmark, Stream benchmark, and a 7-point stencil computation program. As a result, the proposed protocol improves the performance degradation ratio, i.e., the performance using the DLM divided by the performance using only the local memory, from 57% in the former protocol to 78% in stencil computation, which processes data whose capacity is four times larger than the local memory capacity.**

Keywords— out-of-core; remote memory; paging; swap; protocol; page swap; large memory;

I. Introduction

There is an increasing demand for the use of large capacity memory for large-scale scientific and engineering computations; however, there are limitations to the main memory size that can be loaded into a single computer owing to hardware limitations and power consumption. The most common technique to resolve this is using distributed memories over nodes in a cluster by converting existing programs to MPI (Message Passing Interface) programs. The MPI is widely used to process large-scale data, although it is not always possible to convert the original programs to MPI programs, which are parallel processing programs based on the distributed memory model. In some cases, algorithms, libraries, and programs were originally designed under shared memory model, and it is very difficult to convert them to MPI programs. In other cases, program conversion is possible but its development cost is too high, and it takes to a long time for users to accept such programs.

The remote memory paging provided by DLM [1] is an effective option for solving this issue. The DLM provides a virtual large memory by remote memory paging using distributed memories in multiple computers connected to a high-speed network. The DLM supports multi-thread programs [2] written in OpenMP and pthread. By using the DLM, it is possible to run multi-thread programs for large-scale problems on a computation node whose main memory capacity is smaller than the problem data size.

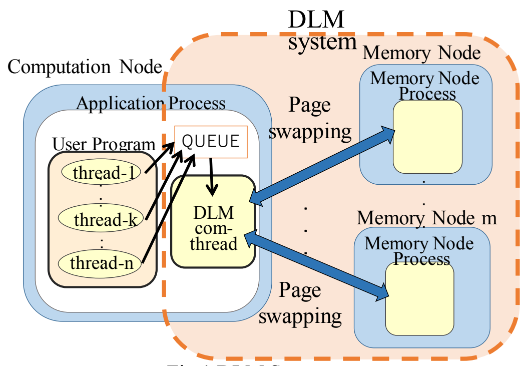


Fig 1 DLM Structure

Fig. 1 shows an overview of the DLM system. The system consists of one computation node and several memory server nodes. A user program runs on the computation node, and memory server nodes provide their memory to the user program. Users can allocate data size that is larger than the local memory capacity using the dlm_alloc() function. The function automatically allocates data area in remote memories in memory servers if the remaining local memory capacity is not sufficient for data. In the computation node, a DLM system thread called communication thread (com-thread) is created when a user program calls the dlm_startup() function. The com-thread is responsible for communication between a computation node and memory nodes and manages page requests from user threads in a request queue.

When one of the threads in a user program accesses data which is not in the local memory in the computation node but in a remote memory node, segv signal is generated. In a segv handler, the user thread puts a page request into the request queue and waits for the page fetch from a memory node. The com-thread extracts this page request from the queue, identifies the memory node that has the requested page, and sends out a page request to the memory node. Then, the com-thread gets the page from the memory node, and sends back another page in the computation node to the memory node as a swap-out page. Next, it temporarily suspends all other user threads. After copying the sent page to the appropriate user address space, it restarts all user threads. The page swapping in the DLM is performed in a unit of the DLM page size that can be defined by users. The user address space of a user program is managed by the DLM page table. Each page table entry contains memory nodes that possess the page.

A page swap protocol and its implementation are significant factors in the performance of remote memory paging systems. The current version of the DLM has a bottle-neck in efficient page swapping because all communication and page swap managements between memory servers and a local computation node are allocated to one system thread—the com-thread. This paper proposes two new page swap protocols and implementations by introducing another new system thread to alleviate this situation. The first one introduces the receiver thread, while the second one introduces the page swap thread. They are evaluated by a micro-benchmark, Stream benchmark, and a 7-point stencil computation program. As a result, the second protocols improve performance degradation for all the programs above. The paper also clarifies the problem in the first protocol.

II. Background and Related works

The DLM provides a large capacity of virtual memory using distributed memories on multiple nodes; however, it does not support distributed parallel processing using CPUs on multiple nodes, unlike in MPI parallel programs. It makes it easier to port multi-thread programs and algorithms designed on a shared memory model. In the DLM, multiple nodes in a cluster are regarded as memory resource not as CPU resource. Program execution on a single node using remote memory has a demerit in terms of the execution time, which takes longer than using only local memory. However, it gives us a merit in terms of seamless and easy porting from newly designed algorithms and programs examined using small local memory to those using the DLM to confirm their effectiveness in a realistic large-scale problem. The DLM is for users who prefer a longer execution time using remote memory to converting their programs to MPI programs while paying significant costs.

The systems based on the partitioned global address space (PGAS) model [6–10] also realize a large address space using distributed memories such as the DLM. Although PGAS systems provide a global view of shared data, most these systems have limits in accessible areas of shared data on remote nodes. Typically, only sleeve areas in stencil computation are accessible by only neighboring nodes. An underlying node-communication scheme in many PGAS systems employ one-sided communications, such as GET/PUT, in GASNET [11], and MPI, where it is necessary to pre-register accessible data areas for communication. Moreover, APIs implemented in PGAS systems are very different from those in OpenMP and pthreads. In contrast, the DLM has no limit in accessing remote data and provides almost the same APIs as those in shared memory model programs.

Another example, which realizes global address space using multiple nodes, is software-based distributed shared memory systems (SDSM) [12–16] that provide not only a global view of shared data but also full accessibility to global data in remote nodes. However, parallel processing with full accessibility to global data on multiple nodes requires some relaxed memory consistency models by performance reasons, it makes SDSM programs different from programs based on a shared memory model. Besides SDSM, remote memory accessing systems [17, 18] that are similar to the DLM were proposed; although such systems, except ArgoDSM [15], employ obsolete implementation such as socket-based communication on Ethernet, and do not support multi thread execution.

In addition to these systems, there are several high-end systems supporting single address space. The ScaleMP [19] unifies conventional Linux servers with specialized software configuration to a single address space system. The recently announced “The machine” by HPE [20, 21] targets big-data processing. It is expected to provide 160 TiB address space with state-of-the-art memory devices. However, such expensive systems are only acceptable to limited users. The DLM is widely available to realize a large virtual memory at low cost for general servers without special memory devices or large-capacity NVMe flash SSDs.

The page swap protocols in this paper focus on how to efficiently implement page swap communication among multiple threads that are dynamically created and terminated in user program execution. The investigation of page replacement policies in the DLM is out of the scope of this study, but was carried out in our earlier study [3]. All experiments in this paper use a basic CLOCK-like replacement policy. Recent MPI implementations support a higher MPI thread support level—MULTIPLE; however, it is well known that MPI performance with MULTIPLE support level is degraded compared with the default support level, FUNNELED or SERIALIZED. Thus, increasing the number of DLM system threads can possibly degrade the performance of MPI communication and increase the overhead of mutual exclusion among system threads.

III. Swapping Protocol Introducing Receiver thread

A. Receiver thread

In the current DLM shown in Fig. 1, all page requests generated by multiple user threads are processed by only one thread, the com-thread, in a step-by-step manner. The com-thread sends a page-request message to a memory node, receives requested page from the memory node (swap in), copies page to user data area, and sends back one of the other local page (swap out) sequentially for each page request generated by many user threads. It is considered as the biggest performance bottleneck in page swapping. To solve this problem, another system thread called the receive-thread (recv-thread) and another request queue called the external-request-queue are introduced as shown in Fig. 2. The recv-thread is responsible for receiving pages sent from memory nodes. Other procedures in page swapping, i.e., send page request, page copy to user data, and swap out, are still managed by the com-thread. After the recv-thread receives a page from a memory server, it inserts a page-apply request in the external-queue. The com-thread processes both requests in the external-request-queuewhere the recv-thread inserts the request, and the internal-request-queue where user threads insert requests. Introducing the recv-thread allows the computation node to simultaneously send requests to each memory node and receive data from memory nodes, which decreases the overhead incurred for page swapping. The next section describes a protocol called the R58, which we used in this implementation.

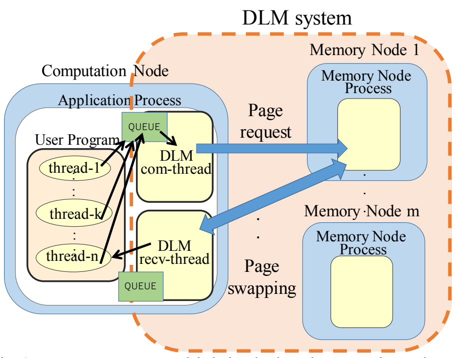


Fig 2 DLM structure which includes the Receiver thread

TABLE I ENVIRONMENT I

CPU	Intel® Xeon® CPU E5-2687W 0 @ 3.10GHz 2CPU × 8core/node
Memory	64GB/node
Cache	L2 256KiB L3 20MiB
Network	Infiniband FDR
OS	CentOS 7.1.1503 (Core) Linux 3.19.5
Compiler	gcc version 4.8.3
MPILib	MVAPICH2 version 2.0.1

B. Swap Protocol R58

Swap Protocol R58 which incorporates the receiver thread is detailed below. The unrevised DLM is referred to as Swap Protocol R3.

1) *When a user thread generates segv signal, the thread inserts a page request to the internal-request-queue in the segv signal handler.*

2) *The com-thread checks the internal-request-queue and sends a page request to the memory node who has the requested page.*

3) *The memory node processes the page request and sends the page to the computation node.*

4) *The recv-thread receives the page from the memory node in the page receive-buffer and put a page-apply request into the external-request-queue.*

5) *The com-thread extracts the page-apply request from the external-request-queue, temporarily suspends all user threads, and copies the page to the user data area.*

6) *The com-thread sends one page in the computation node to a memory node as a swap-out page*

```
#define ENUM ((unsigned long int) (1L<<28))
int main(int argc, char *argv[]){
dlm_startup(&argc, &argv);
array = (unsigned long int *) dlm_alloc(
        sizeof(unsigned long int) * ENUM); //2GB alloc
#pragma omp parallel for
  for (i = 0; i < ENUM; i++) array[i] = i;

for( j = 0; j < 3; j++)
#pragma omp parallel for
   for (i = 0; i < ENUM; i+=(1L<<17))    //data access per 1MB
     if (array[i] != i)   return 1;
dlm_shutdown();
return 0;
}
```

Fig 3 Micro Benchmark

7) *The com-thread then restarts the user threads.*

C. Micro benchmark Performance Evaluation of Protocol R58

We created a micro benchmark like the one in Fig. 3 to evaluate the performance of swap protocol R58 which incorporates the recv-thread. The micro benchmark sets aside 2GB array within the DLM data area, accesses the array in units of page sizes after it initializes all of the elements of the array with element numbers, and checks to see if the figures are properly initialized. That is to say,

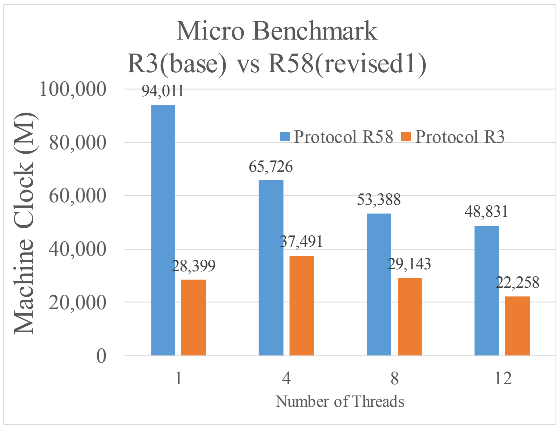


Fig 4 Execution time for R58 and R3's micro benchmarks

in addition to continuing contiguous sequential writing using multiple threads, the micro benchmark conducts non-contiguous readings of the DLM page units. The latter-half of reading is a particularly grueling test—a vast amount of page requests is sent out via multiple threads.

The operation experiment environment is shown in TABLE I. The computation node's DLM data area was set at 800 MiB, the memory node's DLM data area were set at 6000 MiB, and 1 memory node was used and initialized with a page size of 1MiB.

The results of the experiment are shown in Fig. 4. The right bar shows the execution time for R3 and the left bar shows the execution time for R58. The x-axis shows the number of threads used for the computation and the y-axis shows the execution time.

After introducing the R58 swap protocol, the execution time for 1 thread increased approximately 2.75 times more when compared to the R3 execution time. In order to identify the places significantly extending the execution time, we set the 5 following measurement points. We sought out the averages for each measurement point for all page requests occurring within the program.

1) *From right after a segv signal occurs to the point where a com-thread calls upon the internal-request-queue (A1-A2)*

2) *From the point where the com-thread requests a page up to the point where the recv-thread gets the page (A2-A3)*

3) *From the point where the page is received and put into the external-request-queue up to the point where the com-thread is called upon (A3-A4)*

4) *From the point where the com-thread sends out the swapped page up to the point where threads are restarted (A4-A5)*

5) *From the point where the threads restart up to the point where all the page swaps are completed*

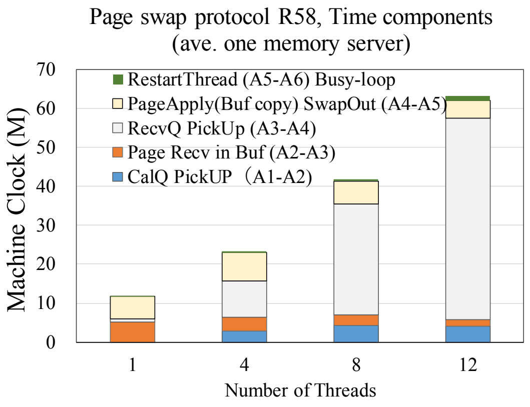


Fig 5 Execution time for each measurement point under R58

Fig. 5 is a graph that captures the execution time for each measurement point under the R58 Swap Protocol for each initialized thread. The x-axis shows the number of threads used for computation and the y-axis shows the execution time.

Under the R58 Swap Protocol, as depicted in this graph, a page was received from memory node in 3) after it is put into the external-request-queue, the time it takes for the page to be called upon by the com-thread for just 1 thread was 7% of the execution time, but when 12 threads were used they occupied approximately 80% of the execution time. Furthermore, the time up the point from where the 1) com-thread extracted a request from the internal request queue increased as the number of threads increased. This phenomenon occurred because under the R58 Swap Protocol the extraction of requests from the internal-request-queue and the external-request-queue occur in tandem out of a principle of fairness, so it is difficult to say that there is efficiency in the process when there is bias in the number of request for both queues. In order to address this issue, we tested out several revisions to the queue extraction scheduling. However, we realized that effectively adapting the process to a dynamically fluctuating number of page requests and a fluctuating number allocation requests for pages being received would be difficult. Therefore, improving the efficiency of the R58 Swap Protocol—leaving the overall control of the DLM to com-thread and creating thread to only receive requests—would be a challenge.

IV. Protocol Using Page-Swapping Recv-Thread

A. *Swap Protocol R77*

The results of the performance evaluation experiment which utilized micro benchmarks from the R58 Swap Protocol showed us that having a recv-thread receive pages, and putting a page allocation request into the external-request-queue after having the com-thread check the external-request-queue and process the request in particular took a fair amount of time. So, we constructed the R77 Swap Protocol to move the DLM page swapping process from the com-thread to the recv-thread. The steps for the R77 Swap Protocol are written below.

1) *When a user thread generates segv signal, the thread inserts a page request to the internal-request-queue in the segv signal handler.*
2) *The com-thread checks the internal-request-queue and sends a page request to the memory node who has the requested page.*
3) *The memory node processes the page request and sends a page to the computation node.*
4) *The recv-thread temporarily suspends user threads and directly receives a page to the user data area*
5) *The recv-thread checks the page table and sends one page held within the computation node to a memory node as a swap-out page*
6) *The recv-thread restarts the user thread*

The special characteristic of the R77 Swap Protocol is that when the recv-thread receives a page, the page does not go through a buffer—it is directly received by the memory and the recv-thread executes the page swapping. Therefore, this protocol is able to swap pages without processing the external-request-queue. Thus far, com-thread could only access data DLM's internal data; however, under the R77 Swap Protocol, recv-thread could access the data as well. As a result, we also installed an exclusive control structure for internal data.

B. Micro benchmark Performance Evaluation of the R77 Protocol

We conducted an evaluation experiment using the micro benchmarks utilized by the R58 Swap Protocol to check R77 Swap Protocol's performance. To compare R77 with R58, we set the 4 following measurement points and sought out the average for each measurement point.

1) *From right after a user thread causes a segv occurs to the point where com-thread calls upon a request from the internal-request-queue (A1-A2)*

2) *From the point where the com-thread sends out a page request to the point where recv-thread receives a page (A2-A3)*

3) *From the point where the recv-thread suspends user threads, receives a page, sends a swapped-out page to the memory node up to the point where the recv-thread restarts the user threads (A4-A5)*

4) *From the point where the user threads are restarted up to the point where page swapping is completed (A5-A6)*

Fig. 6 shows the performances of both R77 and R3. The y-axis shows the execution time and the x-axis shows the number of slides used for computation. When compared with R3, R77 has an execution time of 1.25 times faster for 1 thread and R77 achieved approximately a 1.3 times faster for 12 threads. When compared with R58, R77 was able to run at 1/4 of the total execution time for one the thread and 1/3 of the total time for initializing 12 threads.

Fig. 7 is a graph of execution time during each measurement point for each implemented thread in regard to R77. The x-axis shows the number of threads used for calculation and the y-axis shows each time component. This graph shows that the under R77 the wait time—80% of which was occupied by the internal and

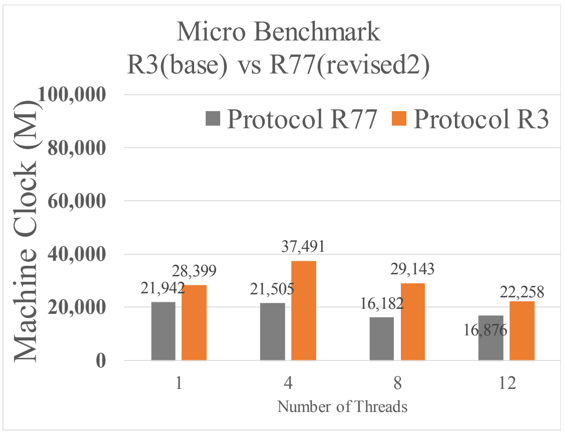


Fig 6 Execution time for R77 and R3's micro benchmarks

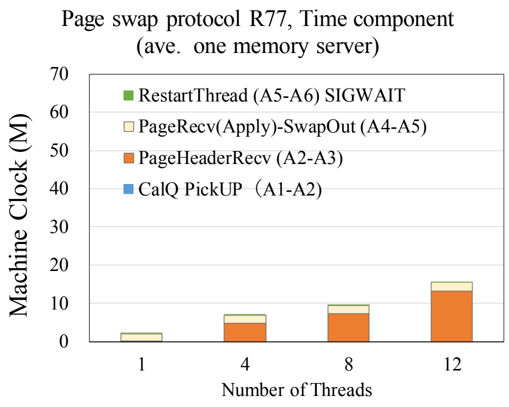


Fig 7 Execution time for each measurement point under R77

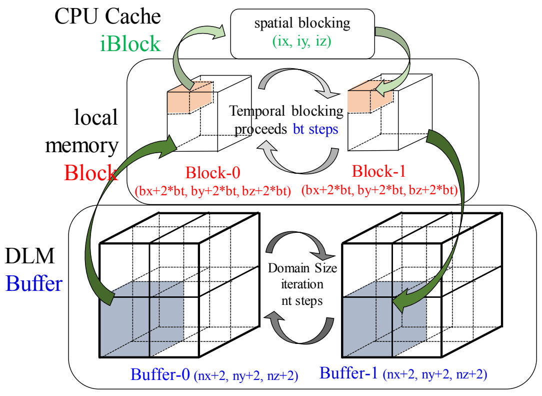
TABLE II ENVIRONMENT II

CPU	Intel® Xeon® CPU E5-2687W v3 @ 3.10GHz 2CPU × 10core/node
Memory	128GB/node
Cache	L2 256KiB L3 25MiB
Network	Infiniband FDR
OS	CentOS 7.1.1503 (Core) Linux 3.19.5
Compiler	gcc version 4.8.3
MPILib	MVAPICH2 version 2.0.1

external-request-queues using the R58 protocol (A3-A4 of Fig 5—has disappeared. However, 2) has taken more time. Under protocol R77, the fact that the recv-thread continues to swap pages for each memory node is efficient, but the fact the recv-thread sends and receives pages required for 1 page swap, there is a performance bottleneck. In a case where multiple memory nodes exist, even if other pages are already sent to the memory node, the nodes are not able to process them right away.

V. Performance Evaluation Using Applications

In section IV, we investigated the effect of the revised protocol on actual application programs. We utilized the 7-point stencil computation that used temporal blocking [4] and Stream [5] for this experiment. TABLE I was used in Stream and TABLE II was used in the stencil computation in relation to operating environment.


CPU Cache
iBlock
spatial blocking
(ix, iy, iz)
local memory
Block
Temporal blocking proceeds bt steps
Block-0
(bx+2*bt, by+2*bt, bz+2*bt)
Block-1
(bx+2*bt, by+2*bt, bz+2*bt)
DLM
Buffer
Domain Size iteration nt steps
Buffer-0 (nx+2, ny+2, nz+2)
Buffer-1 (nx+2, ny+2, nz+2)
Fig 8 Memory layout for Stencil computation

A. *7-Point Stencil Computation that Utilizes Temporal Blocking*

The 7-point stencil computation is one of the most basic grid computations and computes with 1 point of data that renewed and 6 points of data that adjoin that point. The 7-point computation renews all the points of data and repeats these over a multiple time steps. As shown in Fig. 8, the stencil computation holds two blocks that show the amount stored in computation nodes, and stores buffers in the computation node and memory nodes.

In this experiment, we executed a 7-point stencil computation that used both space blocking and temporal blocking. The problem size ranged from 64GiB to 512 GiB and we measured them using 16 threads. The computation node's DLM data area was set at 120 GiB, the DLM data area for the memory nodes was set at 120 GiB, and the problem size ranged from 64 GiB to 256 GiB for 3 memory nodes. The 512 GiB problem size used 5 memory nodes. Measurements were carried out with a 1 MiB page size.

Fig. 9 is a graph depicting the relationship between the problem size and Effective Mflops/s for the 7-point stencil computation which introduced both the R58 and R77 Swap Protocols. The x-axis shows the problem size and the Y-axis is Effective Mflops/s.

All data is held within the computation node at the 64 GiB problem size so there is no difference in performance. All data is not contained within the computation node for computations with problem sizes 128 GiB or larger. With this computation, R77 has between 1.3 to 1.5 times more Effective Mflops/s when compared with R58. Under the R58 protocol, performance decreases as the problem size increases. When comparing R58 at 64 GiB problem size versus 512 GiB problem size, only approximately 54% of the performance is achieved at the 512 GiB problem size. However, when comparing the 64 GiB problem size with the 512 GiB problem size under the R77 protocol, 78% of the performance is achieved at the 512 GiB problem size.

Fig. 10 is a graphic depicting the relationship between the problem size and Effective Mflops/s for the 7-point stencil computation run under the R3 and R77 Swap Protocols. The x-axis shows the problem size and the Y-axis is Effective Mflops/s.

Protocol R3's performance is close to 1.4 times the performance of R77 when run with a 64 GiB problem size (remote memory not used). When run with a

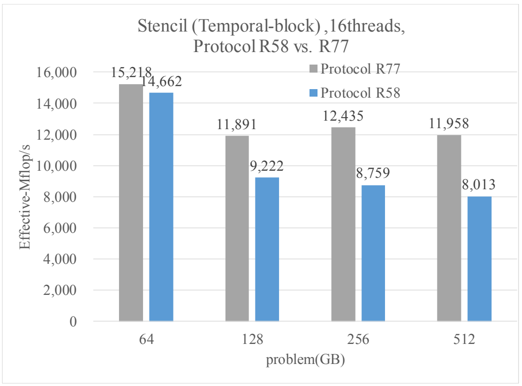


Fig 9 Effective Mflops/s for the 7-point stencil computation under R58 and R3

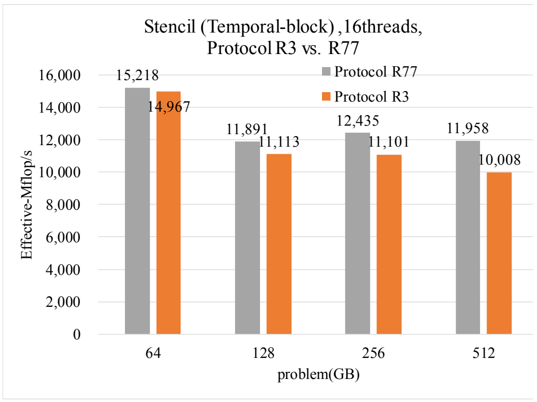


Fig 10 Effective Mflops/s for the 7-point stencil computation under R3 and R77

problem size of 128 GiB or greater that uses remote memory, Protocol R77's performance greatly exceeds R3's performance. When run with a problem size of 512 GiB, Protocol R77 has approximately 1.2 times the performance when compared with R3. Under Protocol R3, performance decreased to 66% when comparing a 512 GiB problem size with 64 GiB problem size (with only local memory access available). Under these same conditions, Protocol R77 was able to achieve 78% of R3's performance. This occurs because page requesting and swapping is done independently by the com-thread and the recv-thread. Thus, numerous memory nodes are able to be processed simultaneously.

B. *Stream Benchmark*

Stream [5] is a benchmark for investigating memory access bandwidths. We allocated 3 arrays—that are used in the Stream computation—within the DLM data area and modified those arrays to carry out the computation. The DLM data area for the computation node was set at 56 GiB. The DLM data area for the memory nodes was set at 120 GiB. There was 1 computation node, 4 memory nodes, the page size was set at 8MiB and we used 14 threads for carry our measurements for the computation. The total size for the 3 arrays was restricted to a range between 48 to 512 GiB and measured. In actuality, the 48 GiB array did not use remote memory and only accessed the local memory of the computation node.

Fig. 11 shows Stream results under the R77 protocol. The y-axis shows bandwidth expressed in a logarithm and the x-axis shows the total amount of secured arrays. With 48 GiBs, where 3 arrays are stored in the DLM data area, there is no difference in performance in relation to processing type. However, with 64 GiB and greater, there are great differences in terms of performance among the 3 processes excluding copy. When comparing the performance of an array size of 64 GiB with an array size of 48 GiB, performance drops to 59%. This is the greatest performance loss to occur even when compared with the decrease between Protocol R3 and Protocol R58. In terms of performance, R77 is 500 MB/s higher than R58.

Fig. 12 shows Stream result under Protocol R58. The copy performance achieved the same level of performance as Protocol R3 when operating under an array size of 48 GiB with only local access. However, when compared to an array size of 48 GiB, an array size of 64 GiB performed at 67%. When operating under Protocol R3, the performance for an array size of 64 GiB achieved 80% of what was achieved when only accessing local memory (the performance of an array size of 48 GiB). Thus, we see that there is a major performance loss in terms of memory node communication under Protocol R58. The performance for an array size of 168 GiB or greater maintained 600 MB/s.

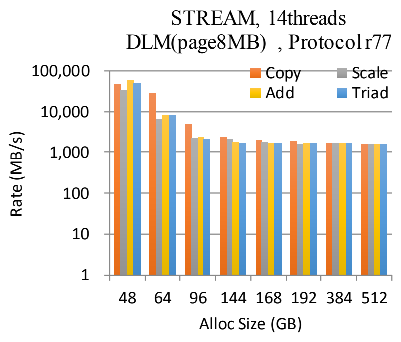


Fig 11 Results of Stream implementation under R77

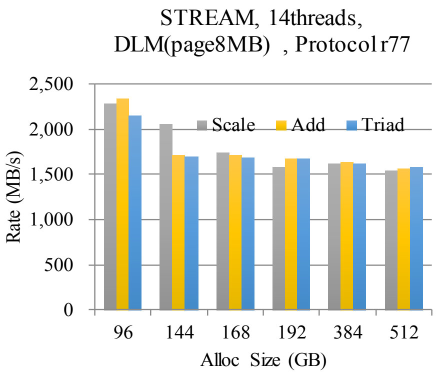


Fig 13 Results of Stream implementation under R77(2)

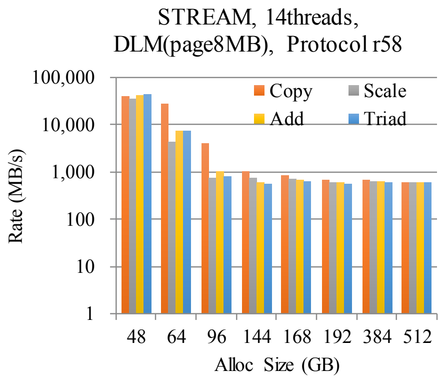


Fig 12 Results of Stream implementation under R58

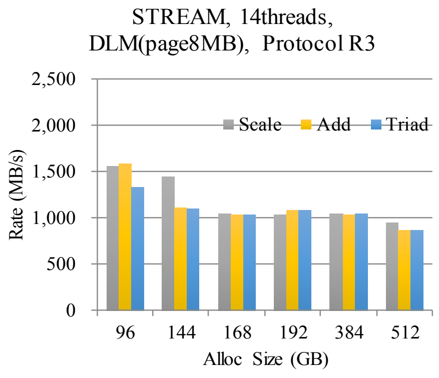


Fig 14 Results of Stream implementation under R3

Fig. 13 is a graph showing the results of Stream with a problem size of 96 GiB or greater under Protocol R77. The y-axis shows the bandwidth and the x-axis shows the problem size. The performance for an array size of 168 GiB or more was maintained at 1600 MB/s or greater. We see that this has better communication efficiency that the R3 and R58 Protocols referenced below. When compared with the R3 Protocol, the R77 Protocol is approximately 600 MB/s better in terms of performance with an array size of 512 GiB.

Fig. 14 is a graph showing the results of Stream with a problem size of 96 GiB or more under Protocol R3. A performance of 1000 MB/s is maintained when Protocol R3 is implemented with an array size of 168 GiB or more which is 4 times the size of the DLM data area that holds the computation node.

VI. Conclusion

In this research, we conducted comparative analysis using Stream and a 7-point stencil computation which utilized temporal blocking, and implemented and designed 2 new communication protocols to increase the communication efficiency of the DLM computation node and the memory node. As a result, we realized that Protocol R77, which approximately 1500 MB/s was achievable as remote memory access bandwidth. In the future, we plan on newly developing effective copy operations (GET, PUT) etc. between the data fixed in the local memory, the data extracted to the remote memory, and the system used to gather and send multiple pages in order to further improve efficiency.

References

[1] Hiroko. Midorikawa, Kazuhiro Saito, Mitsuhisa Sato, Taisuke Boku: "Using a Cluster as a Memory Resource: A Fast and Large Virtual Memory on MPI", Proc. of IEEE cluster2009, 2009-09, Page(s): 1-10

[2] Hiroko Midorikawa, Yuichiro Suzuki, and Masatoshi Iwaida: "User-level Remote Memory Pagingfor Multithreaded Applications", proc.of IEEE/ACM International Symp. on Cluster, Cloud and the Grid ComputingCCGrid2013,pp.196-197,2013-5(DOI10.1109/CCGrid.2013.63)

[3] Kazuhiro Saito, Hiroko Midorikawa, Munenori Kai,; "Page Replacement Algorithm using Swap-in History for Remote Memory Paging", proc. of IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, pp.533-538, 2009-8 (DOI: 10.1109/PACRIM.2009.5291315)

[4] Hiroko Midorikawa, Hideyuki Tan and Toshio Endo:"An Evaluation of the Potential of Flash SSD as Large and Slow Memory for Stencil Computations", Proceedings of the 2014 International Conference on High Performance Computing and Simulation (IEEE HPCS2014) (ISBN 978-1-4799-5311-0), pp.268-277, 2014-7 IEEE-HPCS2014

[5] John D. McCalpin :"STREAM: Sustainable Memory Bandwidth inHigh Performance Computers"

http://www.cs.virginia.edu/stream/

[6] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edo Apra. "Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit", International Journal of High Performance Computing Applications , (2006),20 (2): 203–231

[7] PNNL Global Arrays Toolkit http://hpc.pnl.gov/globalarrays/

[8] Jinpil Lee and M itsuhisa Sato. ``Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems," The 39th international Conference on Parallel Processing Workshops (ICPPW10), pp.413-420, San Diego, CA, Sep. 2010.

[9] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren ,"Introduction to UPC and Language Specification" ,CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[10] Chapman, Barbara, et al. "Introducing OpenSHMEM: SHMEM for the PGAS community." *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. ACM, 2010.

[11] D. Bonachea and J. Jeong GASNet: A Portable High-Performance Communication Layer for Global Address-Space Languages, CS258 Parallel Computer Architecture Project, Spring 2002.

[12] Keheler, Pete; Cox, Alan; Dwarkadas, Sandhya; Zwaenepoel, Willy . "Treadmarks: Distributed shared memory on standard workstations and operating systems". USENIX Winter. 1994: 23–36

[13] Weiwu Hu,Weisong Shi,Zhimin Tang."JIAJIA: A software DSM system based on a new cache coherence protocol",HPCN-Europe 1999: High-Performance Computing and Networking pp 461-472,1999.

[14] Midorikawa,H.,Ohashi,U., Iizuka,H.: "The Design and Implementation of User-Level Software Distributed Shared Memory System: SMS - Implicit Binding Entry Consistency Model -", Proceeding of IEEE Pacific Rim Conference on Communications Computers and Signal Processing, pp.299-302, 2001-08.(DOI: 10.1109/PACRIM.2001.953582)

[15] ArgpDSM https://www.it.uu.se/research/project/argo

[16] Kaxiras, Stefanos, et al. "Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory." Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing. ACM, 2015.

[17] Michael R. Hines, Jian Wang, and Kartik Gopalan,,"Distributed Anemone: Transparent Low-Latency Access to Remote Memory", In Proc. of the International Conference on High Performance Computi*ng (HiPC) , pp.18-21,2006*

[18] Pakin, Scott, and Greg Johnson. "Performance analysis of a user-level memory server." *Cluster Computing, 2007 IEEE International Conference on*. IEEE, 2007.

[19] ScaleMP ,http://www.scalemp.com/

[20] Chen, Fei; Gonzalez, Maria Teresa; Viswanathan, Krishnamurthy; Cai, Qiong; Laffite, Hernan; Rivera, Janneth; Mitchell, April; Singhal, Sharad ."Billion node graph inference: iterative processing on The Machine" ,HPE-2016-101 ,December 22, 2016

[21] The Machine HPE https://www.labs.hpe.com/the-machine