Analyzing MPI-3.0 Process-Level Shared Memory: A Case Study with Stencil Computations

Xiaomin Zhu,[*] Junchao Zhang,[†] Kazutomo Yoshii,[†] Shigang Li,[‡] Yunquan Zhang,[‡] Pavan Balaji[†]

[*]Shandong Computer Science Center (National Supercomputer Center in Jinan), China,
Shandong Provincial Key Laboratory of Computer Networks, China, *zhuxm@sdas.org*
[†]Argonne National Laboratory, USA, {*jczhang, yoshii, balaji*}*@anl.gov*
[‡]Institute of Computing Technology, Chinese Academy of Sciences, China, {*shigangli.cs, yunquan.cas*}*@gmail.com*

***Abstract*—The recently released MPI-3.0 standard introduced a process-level shared-memory interface which enables processes within the same node to have direct load/store access to each others' memory. Such an interface allows applications to declare data structures that are shared by multiple MPI processes on the node. In this paper, we study the capabilities and performance implications of using MPI-3.0 shared memory, in the context of a five-point stencil computation. Our analysis reveals that the use of MPI-3.0 shared memory has several unforeseen performance implications including disrupting certain compiler optimizations and incorrectly using suboptimal page sizes inside the OS. Based on this analysis, we propose several methodologies for working around these issues and improving communication performance by 40-85% compared to the current MPI-1.0 based approach.**

***Keywords*-MPI-3.0, process shared memory, intranode communication, stencil, multicore**

I. Introduction

The Message Passing Interface (MPI) [1] is the de facto parallel programming model in high-performance computing. It provides a message-passing application program interface on distributed-memory machines. There are a few implementations of the MPI specification. Some are open-sourced and freely available.

As we enter into the "big data" age, communication is becoming increasingly important. Moreover, communication is expected to make up a large part of power consumption in the future [2]. Thus, the implementation and optimization of MPI are even more critical.

MPI-3.0 standard recently introduced shared memory interfaces for intranode communication. In this paper, we use five-point stencil computations, which is also used in previous work [3] [4], as a case study to show the usage, overhead, and efficiency of MPI-3.0 shared memory. We introduce how to use MPI-3.0 shared memory to implement five-point stencil. Also, we analyze the overhead and find that the use of MPI-3.0 shared memory has several unforeseen performance implications including disrupting certain compiler optimizations and incorrectly using suboptimal page sizes inside the OS. We then propose several methodologies for working around these issues to get rid of the overhead. As a result, the communication performance is improved by 40-85% compared to the current MPI-1.0 based approach, which is 10-60 % compared to the previous work. Our contributions in this paper include:

1) Analysis of the effect of using MPI-3.0 shared memory on compiler optimizations, and methodologies to overcome the negative effect.
2) Analysis of the overhead of computation over shared memory, including cache misses / memory conflicts due to data access pattern, page faults / Translation Lookaside Buffer (TLB) data misses due to page size. Solutions to these issues are presented.

In this paper, we use MPI-3.0 to indicate the shared-memory implementation and use MPI-1.0 to represent the `MPI_Send/Recv` implementation for intranode communication. The remainder of this paper is organized as follows. In Section II, the background of this paper is introduced, and then the implementation details are given in Section III. In Section IV we analyze the overhead and give solutions to avoid them. Section V shows the improvement on communication from using shared memory. Related work is introduced in Section VI, followed by conclusions and discussions of future work in Section VII.

II. Background

A. Stencil

Stencil computation is a fundamental algorithm used in many applications. It involves a large number of iterations, in each of which the value of every element in a matrix is updated using values of its neighbors (e.g., for a 2D five-point stencil, each element has four neighbors: up, down, left and right), as shown in Figure 1. In a parallel distributed memory implementation, the local matrix on each process is usually augmented to include buffers to store values of remote elements on neighbor processes. These buffers are called ghost area. Figure 1 shows a matrix partitioned by four processes by row, as well as the ghost area. We can see that to update element A we need the value of element B, which belongs to another process. In order to use the value of B, communication is carried out before updating, and the value of B is copied to the local memory space, shown as B' in the figure.

B. Multicore, Many-Core, and Intranode Communication

In recent years, many vendors have tried to improve the performance and efficiency of processors by increasing the number of cores per CPU, and multicore CPUs are ubiquitous and equipped on almost all supercomputers. With the increase

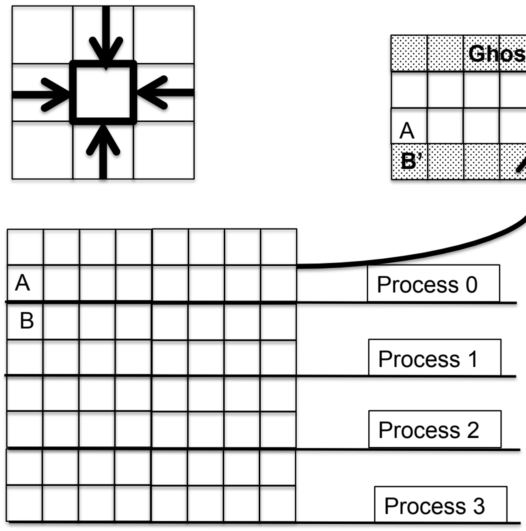


Fig. 1: Element update and ghost area of five-point stencil

in cores in a processor, communication among intranode cores plays an increasingly important role. Indeed, currently 50% of the communication of most distributed applications is due to intranode communication [5].

However, MPI abstracts the programming environment of distributed-memory systems and assigns different ranks to processes inside the same shared-memory node. This operating system-level separation of processes forces MPI to perform unnecessary memory copying for communications within shared-memory nodes, which is suboptimal.

To address this situation, MPI implementors have focused on optimizing the intranode communication in the most general communication functions, `MPI_Send/Recv`, and some optimizations have been integrated into the current MPI implementations. For example, the Network Interface Card (NIC) loop-back without injection to the network method [6] is used if the NIC detects that the destination rank is on the same physical node. Nevertheless, two copy operations are necessary for each message. The sender copies from its private send buffer into a first-in-first-out (FIFO) queue or shared-memory region, and the receiver copies data out into its private receive buffer.

Clearly, a message-passing model based on such a memory copy approach is still suboptimal in terms of memory, energy, and time [7]. Therefore, other approaches for reducing the memory copy times have been proposed. For example, a one-copy method was proposed in [6][8]; and recently a zero-copy technique was proposed in [7], which passes a pointer from the sender to the receiver instead of copying the data. These optimizations help improve the performance of communication. However, data inside the same node can use hardware shared-memory support, and they should be shared spontaneously without any data copy. Moreover, although the zero-copy method was implemented, it has yet been integrated into any MPI implementations; and it has an overhead for buffer allocation and freeing for each message.

In the recent MPI-3.0 standard [1], process-level shared-memory interfaces are introduced as a part of the improved one-sided communication, i.e., remote memory access (RMA) interfaces. With such interfaces, processes in a shared-memory domain (usually a node) can access each other's memory with direct load/store instructions. This approach offers the possibility that programmers can have true zero-copy intranode communication (We actually focus on doing no communication at all and instead use the shared memory capabilities) if the data structure of the algorithm allows direct access to remote memory without copying data to the structure first. Note that MPI-3.0 RMA has two pieces. The first is put/get like operations for data movement. The second is process-level shared memory which we are focusing on. It does use the same base infrastructure (window creation and epoch capabilities), but not the put/get operations.

There are some specific kernel support for direct memory access in some systems, and on traditional Linux, LiMIC [6] and KNEM [9] kernel modules also work. However, with MPI-3.0's new shared memory interfaces, we get this capability in a portable manner.

C. *MPI-3.0 Shared Memory*

Our objective is to use MPI-3.0 shared-memory interfaces to replace `MPI_Send/Recv` for intranode communication without any data copy. The way to use shared-memory is as follows.

1) Communicator Split: Since MPI-3.0 shared memory works only for processes in the same node, we need to distinguish intranode processes from internode ones. Calling `MPI_Comm_split_type(comm,split_type,..., newcomm)` with `split_type = MPI_COMM_TYPE _SHARED` serves our needs. It splits the communicator `comm`, and returns a shared-memory communicator of which the current process is a member through the output parameter `newcomm`. Then communication among processes belonging to the same communicator can be implemented by the following MPI-3.0 shared-memory functions.

2) Memory Allocation: We can use the new shared-memory communicator `newcomm` in the `MPI_Win_allocate _shared(...,info,newcomm, baseptr, win)` function to allocate memory. It is a collective function, and returns the address of the memory in argument `baseptr`. Please see the MPI-3.0 standard [1] for details. The memory allocated in such a way allows direct accesses from another process, in either load or store instructions, with a pointer returned by the windows query function shown next.

3) Windows Query (pointer query): The pointer query function is called `MPI_Win_shared_query(win, rank,..., baseptr)`. The last parameter `baseptr` is an output argument pointing to the shared memory of the target process, which can be used to access the remote memory as if it was local. If the shared memory is allocated contiguously, one process can even access another process's shared memory based on offset to its private pointer initialized in the `MPI_Win_allocate_shared` function.

4) Synchronization and Data Consistency: Using shared memory for communication generally needs an explicit synchronization, which otherwise is implicitly implemented in the `MPI_Send/Recv` mechanism. Programmers are responsible for the synchronization, in other words, the data consistency of the two communicating processes. This is implemented by `MPI_Win_sync` and `MPI_Barrier` functions. `MPI_Win_sync` synchronizes the private and public window copies, and `MPI_Barrier` synchronizes the processes sharing the memory. `MPI_Barrier` is easy to use, but it requires synchronization among all shared-memory processes. Arguably, process synchronization can be implemented in other ways, such as shared flag checking, synchronizing only with necessary processes.

III. Implementation

We use the MPI-3.0 shared memory functions introduced above to implement the five-point stencil computation. Since our goal is zero-copy and direct load/store, we remove the ghost region, which is in fact a receive buffer. Our approach enables the data of the target process to be accessed directly with the pointer to the shared memory, and the memory consumption is $bx \times by$ instead of $(bx + 2) \times (by + 2)$ (for partition on both axes), where bx and by are height and width of the matrix, respectively.

This approach does add programming difficulties and complexity, since the element update computation is not uniform and particular treatment must be paid to the border elements that demand shared-memory access. The pseudo-code is shown in Algorithm 2, following Algorithm 1 of the MPI-1.0 version. As shown in the numbered lines, the algorithms differ in memory size, matrix offset, communication, and computation. In the MPI-3.0 algorithm, we divided the whole matrix into several parts with partition along the y-axis; then each process has only "north (up)" and "south (down)" neighbors, so the top and bottom row should be handled separately. If the partition is done on both axes, the four corner elements need to access the memory of two other processes.

IV. Analysis and Optimization

We implemented the algorithm with the objective of better communication performance; this improvement will be shown in next section. Here we describe the results of profiling the computation part, namely, the element update part; and we present solutions we found to avoid the overhead.

We conducted overhead analysis of this section and performance evaluation of next section on three platforms (Fusion, Blues and MIC) at Argonne National Laboratory, whose configurations are shown in Table I. Also, we used a workstation, whose configuration is shown in the last row of Table I, for the page size related analysis.

Specifically, we found a performance gap between the MPI-1.0 and MPI-3.0 versions, especially when the matrix size was large. We did a series of profiling studies and determined that three factors were contributing to the performance gap: compiler optimizations, cache misses, and page faults.

Algorithm 1: MPI-1.0 implementation

```
Data:
    Matrix size : $bx, by$
    Iteration: $times$
Result:
    Heat Value: $heat$
begin
      $heat \leftarrow 0.0$;
1     $double\ *mem \leftarrow malloc(2 \times (bx+2) \times (by+2) \times sizeof(double))$;
      $double\ *aold \leftarrow mem$;
2     $double\ *anew \leftarrow mem + (bx+2) \times (by+2)$;
      for $iter \leftarrow 0$ to $times - 1$ do
          /* Some Intialization to $aold$ */
          /* Communication */
3         $MPI\_Send/Recv$;
4         for $i \leftarrow 1$ to $bx + 1$ do
              for $j \leftarrow 1$ to $by + 1$ do
                  $anew[i,j] = \frac{anew[i,j]}{2.0} + \frac{(aold[i-1,j]+aold[i+1,j]+aold[i,j-1]+aold[i,j+1])}{8.0}$;
                  $heat+ = anew[i,j]$;
          $Swap(anew, aold)$;
      return $heat$;
```

To address these factors, we abstracted the algorithm to a simpler one, which just assigns a value to each element of an array instead of the stencil computation. The code is shown in Algorithm 3. The only difference between the MPI-1.0 and MPI-3.0 implementations is the memory-allocating methods, numbered 1 and 2 in the algorithm. We did the profiling on the simple code first, and then extended it to the complex stencil computation.

A. Compiler Optimizations

The experiments in this part were carried out on Blues. Note that for the simple code, we allocated memory of the same size in codes of MPI-1.0 version and MPI-3.0 version, which is different from the stencil code because of the removal of the ghost area. In the MPI-3.0 code, the shared memory address is got by the local process with `MPI_Win_allocate_shared`, and by other remote processes with pointers returned by `MPI_Win_shared_query`. In the MPI-1.0 code, the memory is allocated by `malloc`. Our initial expectation was that compiler should generate the same binary code and have the same performance. However, in practice we found that even with the same memory size, the two memory allocation methods have quite different performance impacts on the same computation. We predicted this difference was partially due to compiler optimizations.

With further analysis, we found the prototype of the `malloc` function in glibc has a special attribute decoration,

TABLE I: Evaluation Platforms

Name	CPUs in a Node and Type	Memory Bandwidth	Cores	Linux Kernel	MPI Version
Fusion	2 × 4-core Intel Xeon E5540 @ 2.53GHz	25.6 GB/s	8	2.6.18	MVAPICH2-2.0[10] with Intel icc 13.1.3
Blues	2 × 8-core Intel Xeon E5-2670 @ 2.60GHz	51.2 GB/s	16	2.6.32	MVAPICH2-2.0 with Intel icc 13.1.3
MIC	1 × Intel Xeon Phi 7120a @ 1.238 GHz	352 GB/s	61	2.6.38	Intel MPI5.0 [11] with gcc 4.4.7
Workstation	2 × Intel Xeon CPU X5650 @ 2.66 GHz	32 GB/s	24	3.11.0	MPICH3.1.3 [12] with gcc 4.8.1

Algorithm 2: MPI-3.0 implementation

```
Data:
    Matrix size : $bx, by$
    Iteration: $times$
Result:
    Heat Value: $heat$
begin
    $heat \leftarrow 0.0$;
    $double * mem$;
1   $MPI\_Win\_allocate\_shared(2 \times bx \times by \times sizeof(double), ......., mem, .....)$;
    $double * aold \leftarrow mem$;
2   $double * anew \leftarrow mem + (bx \times by)$;
    for $iter \leftarrow 0$ to $times - 1$ do
        /* Some Intialization to aold */
        /* No Communication here */
3       $MPI\_Win\_sync$;
        $MPI\_Barrier$;
4       $Computing$ :
            $\rightarrow Update\ first\ row$
            $\rightarrow Update\ middle\ rows$
            $\rightarrow Update\ last\ row$
        $Swap(anew, aold)$;
    return $heat$;
```

Algorithm 3: A simpler code with memory allocated either with MPI-1.0 *malloc* or MPI-3.0 $MPI\_Win\_allocate\_shared$

```
Data:
    Size : $n$
Result:
    Time cost: $time$
begin
    $heat \leftarrow 0.0$;
    $double *\ mem$;
    /* Select either MPI-1.0 or MPI-3.0
       memory allocation method */
1   MPI-1.0: $mem \leftarrow malloc(n \times sizeof(double))$;
2   MPI-3.0: $MPI\_Win\_allocate\_shared(n \times sizeof(double), ..., mem, ...)$;
    $time$ -= $currenttime$;
    $double\ temp = 0.0$;
    for $iter \leftarrow 0$ to $n - 1$ do
        $mem[iter] \leftarrow (double)iter$;
        $temp$ += $mem[iter]$;
    $time$ += $currenttime$;
    $print(temp)$;
    return $time$;
```

`__attribute__((__malloc__))`. This is a GCC extension and is supported by both GNU and Intel compilers. It tells the compiler that a function is malloc-like, i.e., that the pointer P returned by the function cannot alias any other pointer valid when the function returns, and moreover no pointers to valid objects occur in any storage addressed by P. Using this attribute can improve optimization. Functions like `malloc` and `calloc` have this property because they return a pointer to uninitialized or zeroed-out storage. Functions like `realloc` do not have this property, as they can return a pointer to storage containing pointers. In contrast, `MPI_Win_allocate_shared` returns the memory allocated through the `baseptr` argument instead of the return value, and hence does not have this attribute. For safety, the compiler has to assume that the returned pointer might alias other valid pointers, and even more the pointer might point to other pointers. Thus the compiler is conservative at optimizations. To overcome this defect, we found a simple approach is to add the `restrict` keyword provided by the C Standard to the pointer passed to `MPI_Win_allocate_shared`, as shown in the bottom of Algorithm 4. The `restrict` keyword conveys the compiler the same information that the pointer does not alias other valid pointers. For the simple code, we also found another approach (shown in the middle of Algorithm 4), that is to declare a second pointer (`mem2`) after shared memory allocation and assign `mem` to `mem2`, which guarantees `mem2` will never point to itself, and thus the compiler can optimize the `for` loop.

With the Intel icc-13.1.3 compiler on Blues, we verified the above analysis by passing `-vec-report6` and `-par-report` to icc. Without the `restrict` keyword or the alias `mem2`, icc reported "loop was not vectorized: existence of vector dependence", and pointed out the reason was the flow and anti-dependence on the `mem` variable in the loop body. With the `restrict` keyword or the alias `mem2`, the compiler reported "LOOP WAS VECTORIZED". Vectorization has a big performance impact. We tested the simple and stencil codes with or without these two approaches with one process. The performance results are show in Figure 2, where the execution time are normalized to that of the MPI-3.0 version. We can see the huge improvement (up to 42%)

Algorithm 4: Decoration of shared memory pointers

```
Data:
    Item size : $n$
Original begin
    $double * \ mem$;
    /* shared memory allocation,
       simplified afterwards                    */
    $MPI\_Win\_allocate\_shared(n \times sizeof(double), ..., \&mem, ...)$;
    for $iter \leftarrow 0$ to $n-1$ do
        $mem[iter] \leftarrow (double)iter$;
Alias begin
    $double * \ mem$;
    $MPI\_Win\_allocate\_shared(...\&mem...)$;
1   $double * mem2 \leftarrow mem$;
    for $iter \leftarrow 0$ to $n-1$ do
        $mem2[iter] \leftarrow (double)iter$;
restrict begin
2   $double * \ restrict \ mem$;
    $MPI\_Win\_allocate\_shared(...\&mem...)$;
    for $iter \leftarrow 0$ to $n-1$ do
        $mem[iter] \leftarrow (double)iter$;
```

with usage of the `restrict` keyword or alias variables. The performance improvement with the stencil code is not as large as the simple code, likely because the stencil computation is more complex than the simple code. In a word, the `restrict` keyword and the alias approach do help compiler optimizations for some relatively simple and regular computations.

However, we found a gap still existed between the MPI-3.0 and MPI-1.0 versions in the memory initialization part. Note that the initialization time was not included in data shown in Figure 2. This gap also existed in the computation part (i.e., the loop body), though it was small. We will analyze the gap in the following sections.

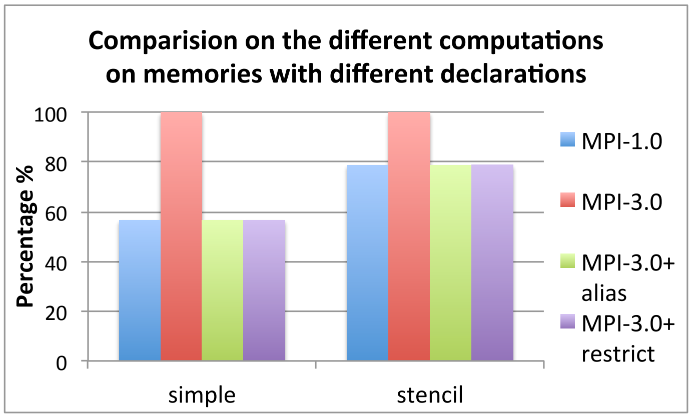


Fig. 2: Performance comparison with/without restrict keyword and alias

B. Cache Misses and Memory Bank conflicts

Another factor we think responsible for the performance gap is cache misses and memory bank conflicts. These two factors are related to data size, data layout, and data access patterns. According to the data access pattern of the five-point stencil, we can easily get the data access stride for one process and among all processes. Since code in the MPI-3.0 version removes the ghost area, the data access pattern is different from that in MPI-1.0 version. For example, for a matrix of size $m \times n$, the relative offsets of the left, right, up and down neighbors of an element are $-1, 1, -(m+2)$ and $(m+2)$ in the MPI-1.0 code, whereas for the MPI-3.0 code, they are $-1, 1, -m$ and m, respectively. In terms of parallel data access pattern, we can see that the stride between two processes are multiples of $m \times n$ for MPI-3.0 and multiples of $(m+2) \times (n+2)$ for MPI-1.0.

It is better if the data access stride is not multiple of the cache line size; otherwise, it may generate more cache misses. Also, it is better that different processes do not access the same memory bank simultaneously. Since the cache miss and memory bank conflict are different with different size, we profiled different matrix sizes. In theory the performance of MPI-1.0 and MPI-3.0 should be the same as a whole. But for almost all cases, MPI-1.0 outperforms MPI-3.0 (shown in the next subsection). Note that for stencil problems, many data padding and re-layout techniques (e.g. [13] [14]) have been proposed for reducing cache miss and memory bank conflict; further discussion is out the scope of this paper.

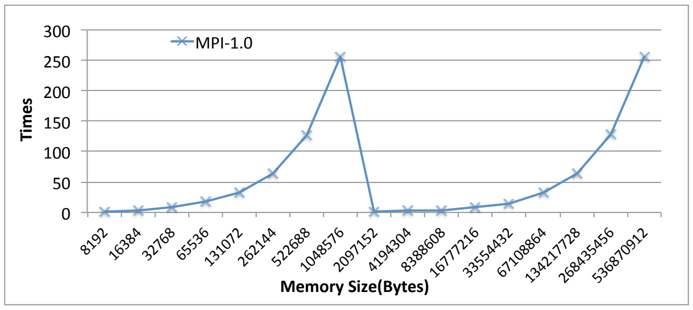
C. Page Sizes

MPI-3.0 shared-memory functions allocate shared memory by the `MPI_Win_allocate_shared` function, and this is the only difference from the regular memory allocation method. In the aforementioned profilings, we found that this affected the computation performance in both the simple code and the stencil code when the allocated memory size was over 2 MB. We therefore turn our focus to page sizes, which will affect page faults and TLB data misses (TLB misses for short).

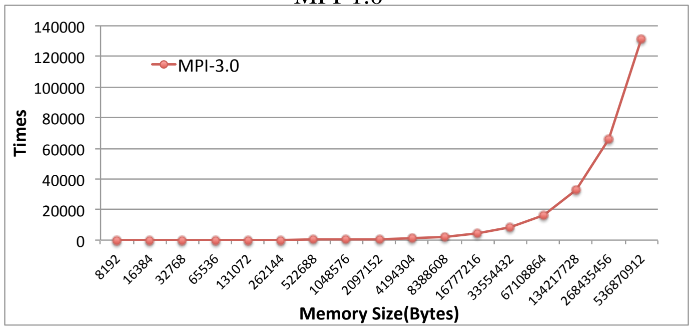
Note that, we did profiling on Blues to find out what the problem is in the "Profiling with the Simple Code" part below. While the "Solutions" and "Performance Improvement" parts were conducted on the Workstation listed in Table I, which we have root privilege as required by some experiments.

1) Profiling with the Simple Code: Using the built-in `perf` command and PAPI [15] we profiled the number of page faults and data TLB misses. We found that the MPI-3.0 version experienced more page faults and TLB misses than the MPI-1.0 version.

To count the page faults and data TLB misses accurately, we used the simple code as before. In addition, we used the underlying `mmap` function directly to implement the abstracted simple code. The only difference is that `malloc` calls anonymous `mmap` whereas MPI-3.0 shared memory calls file-backup `mmap` via `/dev/shm`. We then separate the first-time access to the memory from the subsequent accesses in order to see page faults, which are shown in Figure 3. We can see the trend of page faults with increasing memory size. An obvious breakpoint is shown in the figure when the memory size is 2 MB for MPI-1.0, while for MPI-3.0 it increases steadily.

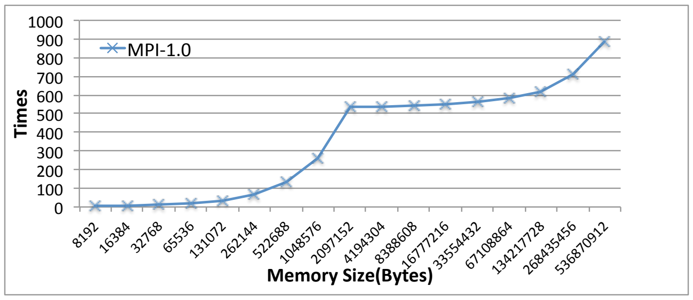


MPI-1.0

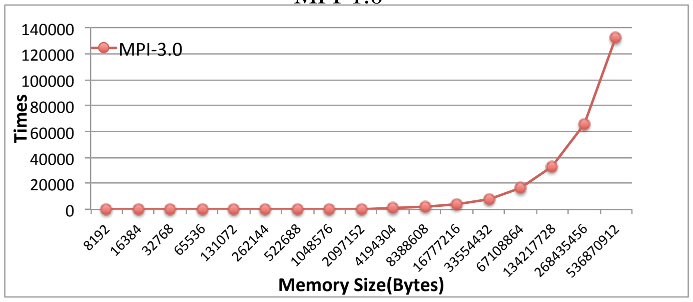


MPI-3.0

Fig. 3: Page faults with increasing memory size



MPI-1.0



MPI-3.0

Fig. 4: Data TLB misses with increasing memory size

2) Explanation: Based on the profiling results, especially the breakpoint at memory size 2 MB, we conjectured that the cause was the different page sizes. Our subsequent analysis confirmed this, as follows.

MPICH shared-memory functions use `mmap` to allocate memory by default, while `malloc` also calls `mmap` internally for larger buffer sizes. [1] The `mmap` function only creates a new range within the process's virtual address space; it does not immediately allocate physical memory unless the MAP_POPULATE flag is specified. When the user-space program attempts a write access on an unpopulated page for the first time, the processor raises a page fault interrupt and the Linux kernel allocates a physical memory page and installs a page table entry into main memory, which usually costs a couple of thousands CPU cycles. A page table is used for the virtual-to-physical address translation. Loading a page table every time from main memory for the address translation is impractical, so TLB caches translation information. Similar to other cache mechanisms, cache misses, so-called "TLB misses", occur, which influences on memory performance. In addition to the default page size such as 4 KB on x86, the memory management unit in a modern processor usually supports larger page sizes such as 2 MB which can reduce the number of page faults and TLB misses, thus it improve memory performance. In relatively newer Linux kernels with proper kernel configurations, users can use large page sizes either explicitly or transparently.

With these configurations, newer Linux kernels can transparently back up a virtual memory range with huge pages. This feature is called transparent huge pages (THP). No source code modification or linking with libraries is required, while it has no guarantee that THP can always back up a virtual memory range with huge pages. THP is used by default if `/sys/kernel/mm/transparent_hugepage/enabled` is set to "always". Also, it needs to increase the amount of shmem permitted per segment in `/proc/sys/kernel/shmmax` and increase the total amount of shared memory in terms of page number in `/proc/sys/kernel/shmall`. However, THP does not work with file-backup `mmap`, so MPI-3.0 shared-memory functions can use only the regular page size(4 KB). Hence, the page size is different, as are the page faults and TLB misses.

3) Solutions: MPICH uses `mmap` function for shared memory (this is configurable; an alternative in Linux is `shmget`) allocation; however, the `mmap` is not anonymous but file-backup and can not use THP. In order to use huge pages with MPICH, we need more system settings, including setting the hugepage number in `/proc/sys/vm/nr_hugepages` and setting the group id of huge page in `/proc/sys/vm/hugetlb_shm_group`. These two can also be done by the `sysctl` command by the root, or they can be set in the configuration file `/etc/sysctl.conf`.

After we finished the above settings, we modified the source code of MPICH 3.1.3 to use huge pages for the MPI-3.0 shared memory. Users can use the huge page support in the Linux

[1]the `malloc` calls the `brk` for smaller buffer sizes. The threshold value to switch between `mmap` and `brk` is managed in a C library.

kernel by using either the `mmap` system call or standard SYSV shared-memory system calls (`shmget`, `shmat`). Both of them are supported in MPICH and can be controlled by a configure option.

First, we implemented the version with `shmget` and found that huge pages can be used. However, the amount of space allocable with SYSV is limited. If the user tries to allocate too much space, it will fail.

Then we turned to file-backup `mmap`. We mounted the hugetlbfs explicitly and modified the source code of MPICH. In `src/util/wrappers/mpiu_shm_wrappers.h`, the path should be the mounted hugetlbfs; In `src/mpid/ch3/channels/nemesis/src/ch3_win_fns.c`, the `PAGESIZE` variable is changed from 4096 (4 KB) to 2097152 (2 MB). Some other accessory modifications (removal of write operation to the file and removal of munmap function) are necessary.

4) Performance Improvement: With the above modifications to system settings and MPICH source code, the computation performance of MPI-3.0 version was improved in terms of page faults and TLB misses.

The number of the page faults is reduced to the same level as the MPI-1.0 version when the requested memory size is over 2 MB. This page fault overhead is once, while the TLB misses overhead is multi times, which occurs during the computing over the memory and affects the computing performance. Then we used the time consumption to evaluate it, and selected 30 different matrix sizes with stride 2. The profiling involved two processes, and the performance is shown in Figure 5. We can see that generally MPI-3.0 with huge pages performed as same as MPI-1.0 which uses huge pages implicitly. Also, the gap between MPI-1.0 and MPI-3.0 without huge pages is plotted.

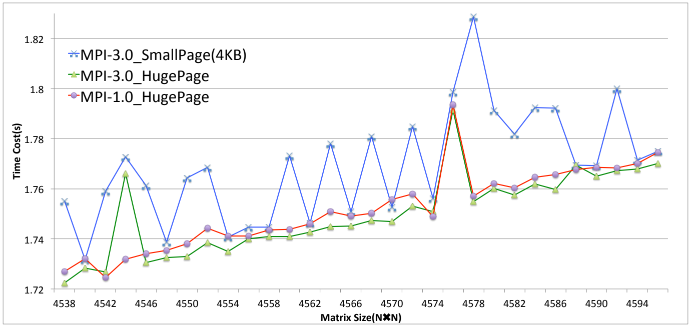


Fig. 5: Computation performance comparison between MPI-1.0 and MPI-3.0

V. Communication Performance Evaluation

We did the evaluation on the first three platforms shown in Table I, with three cases: MPI-1.0, MPI-3.0, and OneCopy. OneCopy was used in [4], which also used five-point stencil as a case study with one data copy. The communication performance is shown in Figure 6. Note that for the MPI-3.0 version, since no data copy is needed, the communication time is the time for synchronization. We can see that with MPI-3.0 we can get an improvement of about 40% to 60% on Fusion, 70% on Blues, and 50% to 85% on MIC, compared with MPI-1.0 version. Compared with the OneCopy method, the improvement is from 10% to 60%.

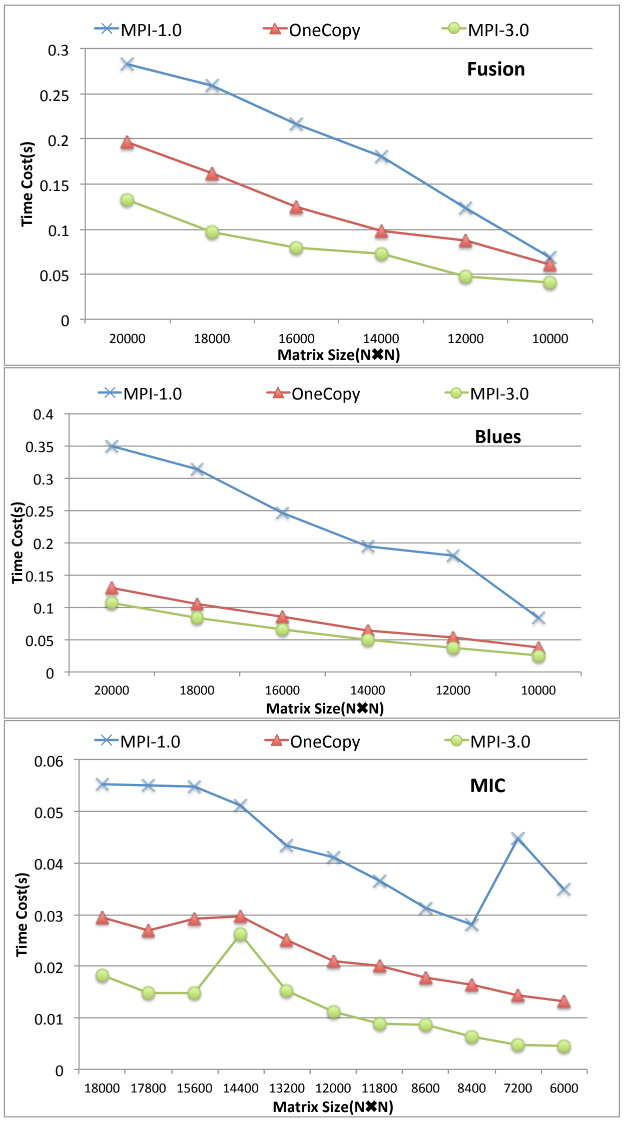


Fig. 6: Communication improvement on three platforms

VI. Related Work

Previous work on using MPI-3.0 shared memory to improve communication is discussed by Hoefler et al. [3][4]. However, their method needs one data copy. They introduced the related overheads but did not analyze them in such depth as ours.

Some methodologies on optimizing intranode communications are proposed, such as the NIC-level loop-back without message injection into the network, which has been integrated in some MPI implementations. But with this optimization, two copies are needed. Some one-copy methods are proposed in [6][8]. A zero-copy method is proposed in [7], but it has an overhead of buffer allocation and freeing for each message and has not been integrated into MPI implementations. Another option is to use MPI + threads, that is, to spawn multiple threads within an MPI process on a node, so that threads can share each other's memory. However, a drawback is that it requires multi-threading support from MPI, which complicates MPI implementations and often adds considerable overhead to communication. The drawbacks of thread-based MPI and multi-threaded MPI applications are analyzed in [2], and these make the MPI-3.0 shared-memory functions even more important, as it leverages the shared memory as well as avoids the overhead caused by multithreading.

VII. Conclusion and future work

In this paper, we introduce how to use new MPI-3.0 shared-memory functions to optimize intranode communication. We analyze the related overheads and give solutions for reducing them. With these solutions, the computation over the shared memory has the same performance as with locally allocated memory. With regard to communication performance, we can see an improvement over 40% and up to 85% compared with MPI-1.0 version.

We leave dynamic and automatic detection and usage of huge pages in MPICH as future work.

Acknowledgment

This work is partially supported by the Promotive research fund for excellent young and middle-aged scientists of Shandong Province under Grant BS2013DX028, the State Key Program of National Natural Science of China under Grant No.61432018, and the National Natural Science Foundation of China under Grant No. 61272136. Additionally, this material was based upon work supported in part by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

References

[1] "MPI: A Message-Passing Interface Standard," http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, Sep. 2012.

[2] A. Friedley, G. Bronevetsky, T. Hoefler, and A. Lumsdaine, "Hybrid MPI: efficient message passing for multi-core systems," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 18.

[3] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, "Mpi+ mpi: a new hybrid approach to parallel programming with mpi plus shared memory," *Computing*, vol. 95, no. 12, pp. 1121–1136, 2013.

[4] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. W. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, *Leveraging MPIs one-sided communication interface for shared-memory programming*. Springer, 2012.

[5] L. Li, X. Zhang, J. Feng, and X. Dong, "mplogp: a parallel computation model for heterogeneous multi-core computer," in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*. IEEE, 2010, pp. 679–684.

[6] H.-W. Jin, S. Sur, L. Chai, and D. K. Panda, "Limic: Support for high-performance mpi intra-node communication on linux cluster," in *Parallel Processing, 2005. ICPP 2005. International Conference on*. IEEE, 2005, pp. 184–191.

[7] A. Friedley, T. Hoefler, G. Bronevetsky, A. Lumsdaine, and C.-C. Ma, "Ownership passing: efficient distributed memory programming on multi-core systems," in *ACM SIGPLAN Notices*, vol. 48, no. 8. ACM, 2013, pp. 177–186.

[8] W. Huang, M. J. Koop, and D. K. Panda, "Efficient one-copy MPI shared memory communication in virtual machines," in *Cluster Computing, 2008 IEEE International Conference on*. IEEE, 2008, pp. 107–115.

[9] B. Goglin and S. Moreaud, "Knem: A generic and scalable kernel-assisted intra-node mpi communication framework," *J. Parallel Distrib. Comput.*, vol. 73, no. 2, pp. 176–188, Feb. 2013.

[10] The Ohio State University, "MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE," http://mvapich.cse.ohio-state.edu, 2014.

[11] Intel Corporation, "Intel MPI library," http://software.intel.com/en-us/intel-mpi-library, 2014.

[12] Argonne National Laboratory, "MPICH — High-Performance Portable MPI," http://www.mpich.org, 2014.

[13] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan, "Data layout transformation for stencil computations on short-vector simd architectures," in *Compiler Construction*. Springer, 2011, pp. 225–245.

[14] J. Jaeger and D. Barthou, "Automatic efficient data layout for multithreaded stencil codes on cpu sand gpus," in *High Performance Computing (HiPC), 2012 19th International Conference on*. IEEE, 2012, pp. 1–10.

[15] "PAPI," http://icl.cs.utk.edu/papi/.

Government License