Can Streaming SIMD Non-Temporal Instructions Benefit Intra-node MPI Communication on Modern Multi-core Platforms?

Hao Wang, Miao Luo, Krishna Kandalla, Sayantan Sur, Dhabaleswar K. Panda
Department of Computer Science and Engineering, The Ohio State University
{wangh, luom, kandalla, surs, panda}@cse.ohio-state.edu

Abstract

Modern multi-core platforms are evolving very rapidly. Scientific applications are leveraging this growth in computing power to continually increasing the computational complexity of problems that can be solved. These applications are typically written using the Message Passing Interface (MPI), the pre-dominant parallel programming model. Currently, multi-core platforms have multiple sockets with each socket housing a multi-core processor chip. There are various levels of cache hierarchy and sharing in this platform. Cores in the same socket may have private L1 and L2 caches and share L3 cache, while cores on different sockets do not share any caches. MPI processes are placed on one process per core. These processes communicate using user-level shared memory. This involves memory copies, to and from shared buffers. Thus, making performance of memory copy routines critical to overall performance. The Streaming SIMD Extensions (SSE) provide instructions that can perform streaming and non-temporal memory moves. Utilizing these instructions can improve memory bandwidth and reduce cache pollution. The impact of these instructions may depend on the levels of shared cache hierarchy between the communicating processes.

In this paper, we investigate whether SSE non-temporal instructions are beneficial for intra-node MPI communication. We design a framework for optimizing memory copy functions within MVAPICH2, a popular implementation of MPI on InfiniBand. We investigate six different possible memory copy functions using different instructions and integrate them into MVAPICH2. We present performance investigation on latest multi-core architectures: AMD Barcelona, Intel Nehalem, and AMD Magny-Cours. Our investigation reveals that using non-temporal instructions for communicating processes located on different sockets can significantly improve bandwidth at 64K Byte messages by 60% and 13% for the multi-pair bandwidth benchmark, and improve NAS benchmark and HPCC benchmark from 2% to 14% on AMD Barcelona and Intel Nehalem architectures.

I. Introduction

Over the past several decades, scientific applications have proved their insatiable need for more computation power. Most scientific applications attempt to simulate a wide range of natural phenomena from cosmology, molecular dynamics, weather prediction, etc. In order to reach the level of accuracy and length of simulations, application scientists increase their problem size and resolution and these applications consume tens of thousands of processor cycles on large scale supercomputing systems, each year. Over the past few years, we have witnessed a dramatic change in compute node architecture. Due to power requirements and heat dissipation issues, individual processors cannot be made arbitrarily fast by increasing clock frequency. As a result, processors have evolved to have multiple computation cores operating at relatively lower clock frequencies. Recently, AMD Magny-Cours [1] compute platforms with up to 24 cores per node have been released. The compute platform typically consists of multiple sockets, with each socket housing a multi-core chip. These chips contain multiple cores inside. Current generation commodity processors typically have 8-12 cores, with many more planned for the future. Compute cores that reside within a socket may share caches. For example, in the Intel Nehalem architecture [2], the cores that are on the same socket share L3 caches, while L1 and L2 caches are private for each code. Cores that do not live in the same socket may not share any caches at all.

The Message Passing Interface (MPI) is the most popular parallel programming model among scientific applications. Typical MPI implementations use IPC shared memory for intra-node communication. This is done at user-level by copying the message into a shared memory buffer and then copying it back into the destination process memory. Although, this scheme involves two copies, this is the preferred method for a very large percentage of MPI users. Production systems, such as those in supercomputer centers, have traditionally avoided OS-kernel assisted approaches [3], [4] even though they avoid one copy in the message passing process due to security concerns and to remove the chance of entire nodes failing when kernel components are buggy,

as opposed to the MPI process aborting when bugs are encountered in user-level. Due to this continued interest in user-level techniques, we make it the focus of our paper.

Based on the node architecture, and the communication pattern, cores with various levels of shared cache hierarchy may communicate. In the case of cores sharing caches, message passing may be optimized by leveraging the caches, by avoiding memory bus transactions. However, in the case where caches are not shared, it may not be optimal to pollute the caches further by occupying them with intermediate shared memory buffers. Additionally, there are multiple types of processor instructions that are available on modern multi-core machines. The streaming SIMD instructions in the SSE instruction set provide a whole range of memory copy operations that may be individually more efficient in certain cases. For example, non-temporal moves, which bypass cache hierarchies may be more efficient when the communicating cores do not share caches. On the other hand, temporal moves may be more efficient in a shared cache environment.

Depending upon patterns of communication and MPI usage characteristics by scientific applications, the answer may differ. To the best of our knowledge, there has been no contemporary study on the impact of specific memory copy instructions on MPI performance on multi-core machines. There are several open research questions that need answering:

1) Can standard memory copy routines provided by C compiler libraries provide near optimal performance for inter-socket communication (no cache sharing)?
2) Can advanced SSE2 non-temporal instructions be utilized to implement a special memory copy function for inter-socket communication? What are the performance advantages?
3) Can the potential performance advantages be always realized on different modern multi-core systems?
4) Can the benefits be observed at the MPI level with an enhanced set of micro-benchmarks and application kernels?

In this paper, we investigate whether non-temporal instructions, such as those provided by SSE, are beneficial for intra-node message passing, especially in the case where there are no levels of caches that are shared. We design a framework for optimizing memory copy functions within MVAPICH2, a popular implementation of MPI on InfiniBand on multi-core platforms. We investigate six different possible memory copy functions using different instructions and integrate them into MVAPICH2. We present a thorough performance investigation on latest multi-core architectures: AMD Barcelona, Intel Nehalem, and AMD Magny-Cours. Our investigation reveals that usage of non-temporal instructions for communicating processes that are located on different sockets can significantly improve bandwidth. Performance can be improved by 60% and 13%for 64K Byte message sizes over the default shared memory implementation for the multi-pair bandwidth benchmark on Barcelona and Nehalem architectures.

The rest of the paper is organized as follows. In Section II we provide background information about the topics discussed in this paper. Then in Section III, we describe the design of our framework of optimal memory copy in MVAPICH2. Experimental results are presented and discussed in Section IV. Related work in this field is presented in Section V. Finally, we conclude the paper in Section VI and discuss future directions.

II. Background

In this section, we provide requisite background about the topics discussed in this paper. We begin our discussion with streaming SIMD extensions and the modern multi-core platforms used in this paper. Then, we discuss user-level intra-node message passing techniques employed by state-of-the-art MPI implementations.

A. Streaming SIMD Extensions (SSE)

Streaming SIMD instruction set was proposed in the late 1990s as an extension to the x86 instruction set. The motivation behind this extension was to provide advanced instructions that could use novel processor features to optimize many applications. These instructions applied to integer operations, floating point operations, branching, and data moving. The single instruction multiple data (SIMD) aspect improves the utilization of processor, cache and memory subsystems by leveraging data parallelism. In this paper, we focus exclusively on the data moving operations or `mov` instructions.

There are multiple variations of `mov` instructions available that move 32-bits, 64-bits and 128-bits of data using one instructions. When multiple words can be moved together, the memory bus bandwidth usage is maximized. Aligned moves are particularly bandwidth efficient as they reduce the number of bus transactions. These instructions obey the cache hierarchy. Peak performance is typically observed when data is moved between caches.

In a multi-processor system implementing various levels of cache hierarchy, one of the major performance considerations is to maximize cache performance by reducing cache pollution. In order to provide opportunities for software to prevent cache-pollution, various versions of non-temporal instructions are provided (32-bit, 64-bit, 256-bit). These instructions bypass caches, and have the potential of improving performance in the cases that caches are not shared, and the amount of data being moved is too large to fit in destination cache anyways. i.e. reduce cache thrashing.

Although compilers are able to generate assembly code with more fancy SSE instructions usage, compilers are hard

to generate instructions considering cache sharing between processes due to lack of the run time information. The availability of various instructions for moving depending on different case scenarios provides an opportunity to optimize memory copy functions in MPI libraries. We will discuss it in detail in Section III.

B. Modern Multi-core Platforms

The increasing demand for computational cycles is being met by the emergence of modern multi-core architectures. Current generation high-end servers are comprised of several sockets with each socket comprising of 4 to 8 compute cores. The number of compute cores are constantly on the rise with experimental chip-sets already offering about 64 cores per socket [5]. We expect to see several main-stream systems offering such capabilities in the near future, as well. Owing to such trends, we believe that the performance of communication operations between processes that are on the same compute node is bound to play a key role in scaling scientific applications. As the number of compute cores increases, the aggregate compute power offered by the compute node improves. However, we need to note that more resources will invariably be shared across several processing units. Current generation multi-core architectures rely on technologies such as the QuickPath Interconnect(QPI) [6] and the HyperTransport [7] to deliver low latency and high bandwidth between different sockets. Next generation systems will necessarily use a complex network within a socket to provide connectivity between the compute cores. Most current generation architectures involve some degree of cache sharing between processes that are on the same socket, while processes belonging to different sockets do not share cache and could have different memory controllers, as well. It is hence necessary to consider the subtle differences introduced by the various hierarchies while designing a high-performance message passing library that ensures low communication latencies for all types of intra-node communication operations.

C. User-Level Shared Memory Message Passing Techniques

With the advent of multi-core processors, the amount of communication within a node is rapidly increasing. Typical MPI implementations utilize IPC shared memory message passing for intra-node communication. In this mechanism, a circular pool shared memory buffers is created prior to message passing. When a process wants to send a message to another process on the same node, a buffer is consumed from the circular pool. The message is then copied to the shared memory buffer, from which the receiver again copies it to the destination buffer. Thus, there are two copies involved in this method. The interaction with the cache hierarchy in a two-socket quad-core processor system is shown in Figure 1. We consider the inter-socket communication with sufficiently large message sizes, concentrating on the last-level cache, L3. In the first step, the source buffer is brought into the L3 cache. Then it is copied to the shared memory buffer that also in the cache in the second step. This cache line is then evicted to memory, when the receiver attempts to read the message in the third step. The shared memory buffer is then copied to the receiver's cache and finally placed into the destination buffer in the fourth step. Current MPI stacks, MVAPICH2 [8], MPICH2 [9] and OpenMPI [10] follow such a technique.

III. Design and Implementation

In this section, we discuss the design and implementation of our design of cache hierarchy aware non-temporal memory copy instructions. We describe the framework in MVAPICH2.

A. Design Considerations

As discussed in Section II-C, user-level shared memory based copy techniques involve two memory copies. There can be two types of cache sharing relationships. One is when two communicating cores share the last level of cache. Secondly, the communicating cores may share no caches at all. The two memory copies used in this method are: copy from send buffer to shared buffer, or copy from shared buffer to receive buffer.

When two communicating cores share a cache, the performance of the two memory copy technique is almost completely determined by the cache performance. The cache performance depends on cache size (compared to size of messages being exchanged), cache replacement policy and coherence policy. The best performance is expected when both sender and receiver buffers, along with intermediate shared memory buffer fit within the cache hierarchy. In-fact, when caches are shared, non-temporal copies can in-fact adversely affect performance. When two communicating cores do not share a cache, non-temporal instructions have the possibility to improve performance. From the point of view of cache utilization, the data in shared buffer illustrated in Figure 1 shouldn't be put into cache either on sender side or on receiver side due to the following reasons:

1. The shared memory created and managed by MPI library is not visible to applications. It cannot be used by applications directly.
2. The shared memory is usually organized as a ring buffer and used with a circular fashion. Individual cache lines will not be reused before the entire ring of buffers is used. This is especially true in a large share memory pool that exceeds the L3 cache size.
3. The sender side issues write operations and the receiver

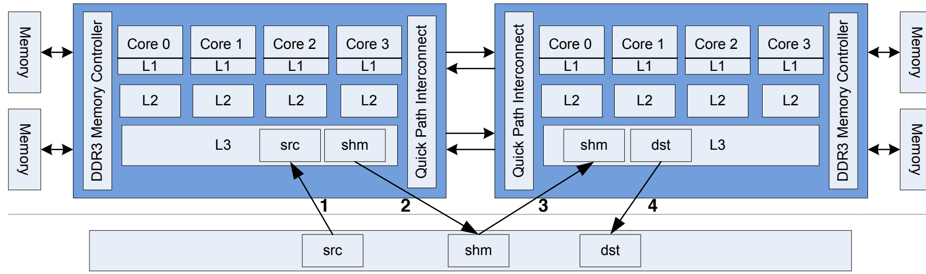


Fig. 1. Memory accesses for User-Level Shared Memory Implementation

side issues read operations. The shared memory data in cache will become invalid before it can be reused by read operations.
4. The data in shared memory may also compete for cache with application's data that could potentially be reused, which could result in application data being evicted from the cache, leading to higher conflict misses.

In Section III-B, we implement different memory copy routines and verify whether non-temporal instruction can improve performance when two communicating cores don't share a cache in Section IV.

B. Memory Copy Instruction

In order to understand the performance characteristics of different move instructions for intra-node communication on the modern multi-core processors, we use the following move instructions [11] to implement six different memory copy functions.

1. 32 bit move: original x86 instruction `mov` and `movsb`;
2. 64 bit move: MMX instruction `movq`;
3. 128 bit move: SSE2 instruction `movdqa` and `movdqu`;
4. 32 bit non-temporal move: SSE2 instruction `monti`;
5. 64 bit non-temporal move: SSE instruction `movntq`;
6. 128 bit non-temporal move: SSE2 instruction `movntdq`, and SSE4.1 instruction `movntdqa`.

For situations in which the communicating processes are on different sockets and do not share caches, we investigate non-temporal move instructions. The main rationale of using the non-temporal instruction is to minimize cache pollution. We also evaluate the performance impact of choosing move instructions with different widths on MPI benchmarks and applications performance, particularly those that are sensitive to bandwidth. We use four non-temporal move instructions: `MOVNTI` (32 bit), `MOVNTQ` (64 bit), `MOVNTDQ` (128 bit), and `MOVNTDQA` (128 bit). `MOVNTI`, `MOVNTQ`, and `MOVNTDQ` move data from registers to memory (write to memory); and `MOVNTDQA` move data from memory to registers (load from memory). The `MOVNTDQA` instruction is used to implement non-temporal load and others are used to implement non-temporal store.

1) Non-temporal Store: Algorithm 1 illustrates the designs associated with a non-temporal memory copy function with `MOVNTDQ` instruction, which has input parameter **n** as original data size need to be copied, **dst** as the destination address and **src** as the source address.

Although there are sixteen 128 bit registers, xmm0-xmm15 that can be used to implement the 128 bit non-temporal move operation, registers xmm8 through xmm15 cannot used since they are accessible only in 64-bit operating mode, and one more byte is required to store the instruction itself. For example, 4 bytes are required for "`MOVDQA` (%rsi) %%xmm0", but 5 bytes are required for "`MOVDQA` (%rsi) %%xmm8". In the memory copy function amd64_cpy_128_nt_store(), registers xmm0 to xmm7 are used. It can copy 128 bytes in one loop (128 bit per register x 8 registers). At the beginning of the Algorithm 1, if the destination address is not aligned with the cache line size, we copy the data in two steps. We first copy the first few bytes of the data with another memory copy operation amd64_cpy_small which copy small amounts of data with the `MOVSB` and `MOVSQ` instructions, such that the remainder of the data is aligned. We then copy the rest of the data, as described in Algorithm 1. We also use the `PREFETCH0` instruction twice to fill 128 bytes of data into the cache at the start of the operation. In each iteration of the loop, 128 bytes of data are fetched into the cache, by pipeling the `MOVUPS` and the `MOVAPS` instructions with the `PREFETCHT0` instruction. The `MOVUPS` and the `MOVAPS` instructions are used to load aligned or non-aligned data into the xmm register, respectively. The `MOVNTDQ` instruction is used to store data from the xmm register into the 128 bit memory location without cache pollution. Finally, the `SFENCE` instruction is used to maintain ordering across different write operations. Since the data in the shared-memory buffers by-pass the L3 cache, our proposed copy operation could lead to lower conflict misses and better cache availability for applications.

2) Non-temporal Load: We also design another function amd64_cpy_128_nt_load() with the `MOVNTDQA` instruction for the MPI receive operation to minimize the cache pollution on the receiver side due to the shared buffers. Unlike the `MOVNTDQ` instruction, `MOVNTDQA` is only effective for

Algorithm 1 MOVNTDQ memory copy

```
if n > 128 then
  if dst unaligned with cache line size then
    j = unaligned_data_size;
    amd64_cpy_small(dst, src, j);
    n -= j;
  end if
  i = n >> 7; n = n - (i<<7);
end if
prefetcht0;
if i>0 then
  if src misaligned then
    for i>0 do
      prefetcht0;
      movups; movntdq;
      src += 128; dst += 128
      i -= 1;
    end for
  else
    for i>0 do
      prefetcht0;
      movaps; movntdq;
      src += 128; dst += 128
      i -= 1;
    end for
  end if
  sfence;
end if
if n>0 then
  amd64_cpy_small(dst, src, n);
end if
```

Write Combining (WC) memory type [11]. However, most systems rely on the Write Back(WB) mechanism and the memory type can be changed only in the kernel mode.

In order to enable the MOVNTDQA instruction, a kernel module is implemented, where two kernel interfaces of Page Attribute Table (PAT) [12] set_memory_wc() and set_memory_wb() are called to change the memory type. This kernel module is called to change a large piece of shared buffer from WB to WC in MPI_Init() if amd64_cpy_128_nt_load() is set in MPI environment variable; and is called in MPI_Finalize() to change the memory type back. However, we don't include its performance data in Section IV, since the kernel module required, which is out of the scope of this paper.

C. Dynamic Memory Copy Routines Selection

In order to enable memory copy routines selection for different processes in the run-time, we design and implement a selection framework that considers the cache hierarchy and sharing. During MPI_Init(), we utilize the Portable Hardware Locality (hwloc) [13] interface to generate a hierarchical topology tree. When MPI processes are bound to different cores, depending on the hwloc tree, MPI library detects cache sharing or no cache sharing between cores. This step is implemented for all intra-node communication channels. Finally, MPI environment variables are used to choose between different memory copy routines at run-time.

IV. Experimental Evaluation

A. Experimental Testbed

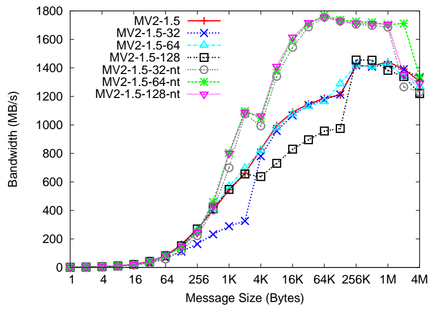
The experimental testbed consists of three different types of multi-core processors. The first platform is an AMD Barcelona machine with four sockets. Each socket is a Quad-Core AMD Opteron 8350 Processor with 2 MB shared L3 cache. The second platform is an Intel Nehalem machine with two sockets. Each socket is a Quad-Core Intel Xeon E5530 Processor with 8 MB shared L3 cache. The third platform is an AMD Magny-Cours machine with two sockets. Each socket is two Hex-Core AMD Opteron 6174 Processors each with 12 MB shared L3 cache.

RedHat Enterprise Linux Server 5 with kernel version 2.6.18-164 is used on all machines along with Open Fabrics Enterprise Distribution (OFED) version 1.5.1. We used MVAPICH2-1.5 as our MPI library. We utilized GCC version 4.1.2 for the purposes of our evaluation on micro-benchmarks. Intel Compiler version 11.1 is used for all application level evaluation. For micro-benchmarks, OMB in Section IV-B and IMB in Section IV-D, performance with GCC and ICC are the same, as time is communication dominated. The MVAPICH2 code base and micro-benchmarks are compiled with GNU compiler in these two sections. For applications in Section IV-E, MVAPICH2 code base and applications are all compiled with Intel compiler.

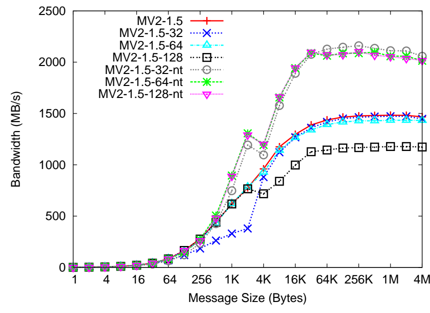
In the following figures, we refer to the default MPI library with no memory copy optimizations as MV2-1.5. We analyze the performance of different temporal memory routines based 32, 64 and 128 bit move instruction and we refer to these as MV2-1.5-32, MV2-1.5-64 and MV2-1.5-128, respectively. We refer to the different non-temporal memory schemes that we have used as MV2-1.5-32-nt, MV2-1.5-64-nt and MV2-1.5-128-nt, respectively.

B. OSU Microbenchmarks

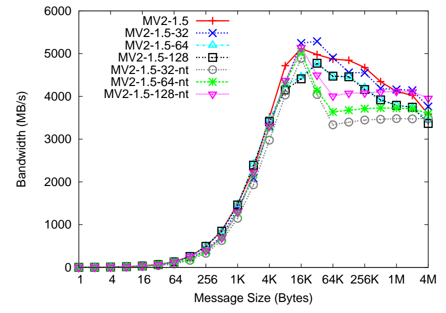
We ran various micro benchmark level tests from OMB [14] suite. We used the single-pair and multi-pair latency and bandwidth tests. The single-pair tests osu_latency and osu_bw were run with two processes across two sockets on three types of processors; the multi-pair tests osu_multi_lat and osu_mbw_mr were run with four pairs of processes across two sockets on Barcelona processor and Nehalem processor and six pairs of processes on Magny-Cours processor, using all cores on two sockets. As a result,



(a) Barcelona

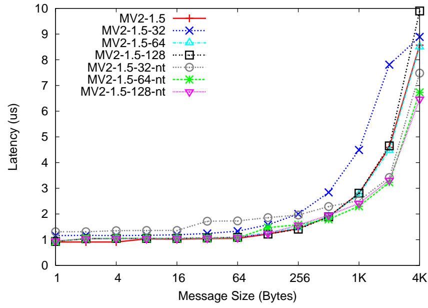


(b) Magny-Cours

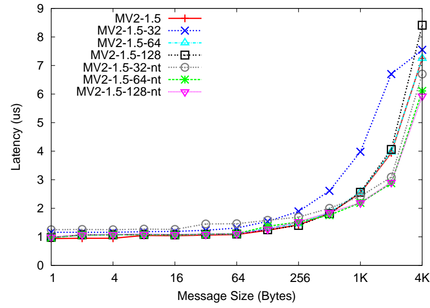


(c) Nehalem

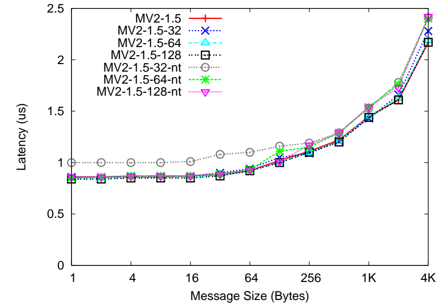
Fig. 2. Bandwidth performance



(a) Barcelona

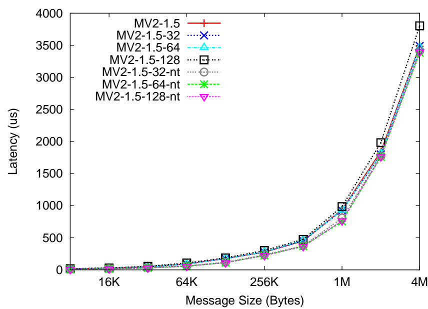


(b) Magny-Cours

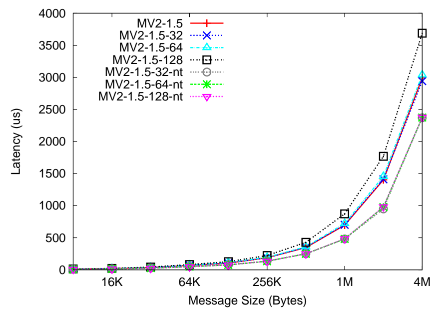


(c) Nehalem

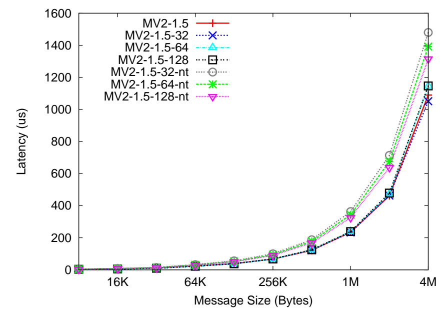
Fig. 3. Small Message Latency Performance



(a) Barcelona

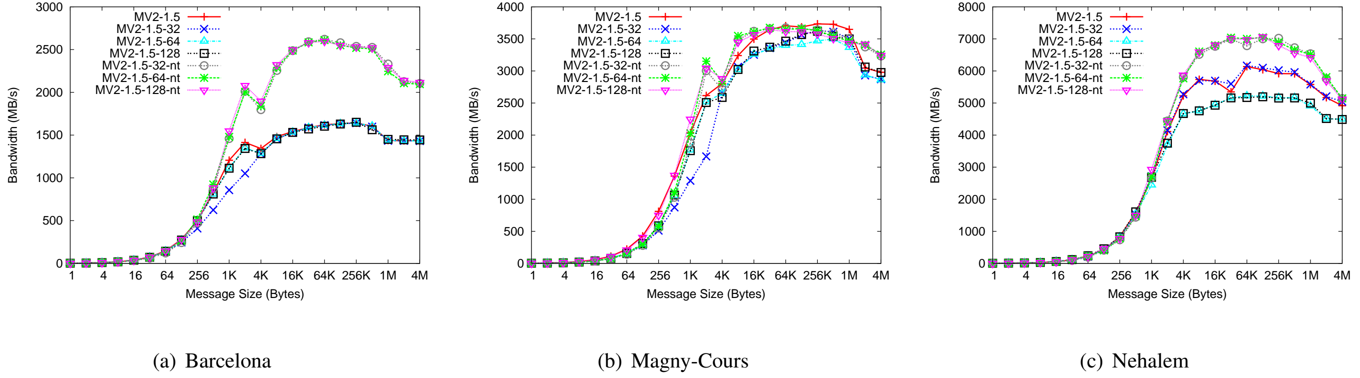


(b) Magny-Cours



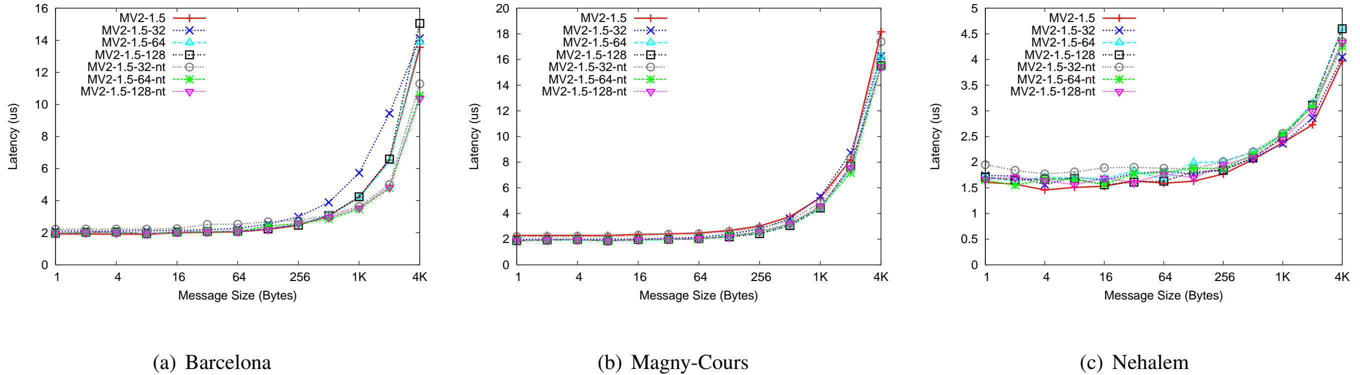
(c) Nehalem

Fig. 4. Large Message Latency Performance



(a) Barcelona (b) Magny-Cours (c) Nehalem

Fig. 5. Multi-Pair Bandwidth Performance



(a) Barcelona (b) Magny-Cours (c) Nehalem

Fig. 6. Multi-Pair Latency Small Message Performance

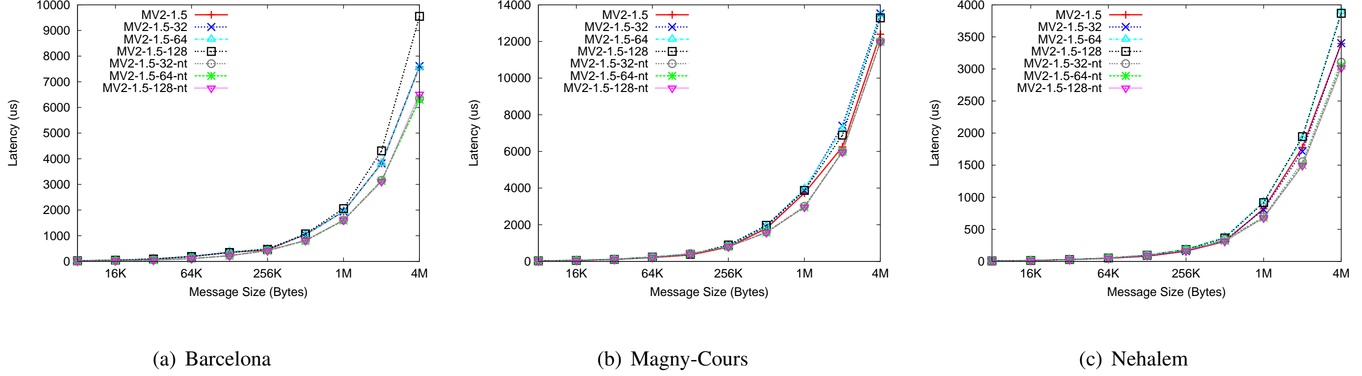
all MPI channels use no cache sharing communication. In OMB micro benchmarks, the send buffer and the receive buffer are allocated as 4MB in the beginning and aligned with the page size. For each data size, the corresponding size data from the start of the send buffer will be send out, and it will repeat default loop time to get the stable number.

1) Bandwidth Performance: Figure 9 and Figure 5 show the performance of single-pair and multi-pair bandwidth benchmarks separately on different platforms.

On Barcelona processor, non-temporal memory copy routines have better performance compared with standard memcpy() function, which is illustrated as "MV2-1.5" in all the figures, and temporal memory copy routines. The non-temporal routine "MV2-1.5-128-nt" achieves up to 48% improvement for single-pair bandwidth testing (Figure 2(a)) and 60% improvement for multi-pair bandwidth testing (Figure 5(a)), compared with standard memcpy() function. The performance improvement comes from the data send and receive mode: on the send side, the data in the send buffer will be sent in each loop; after the first loop, the memory has been filled into sender side L3 cache; in the followed loops, the data in the L3 cache will be reused. Due to cache contention between the send buffer and the shared buffer, when non-temporal memory copy minimizes the shared buffer data into the L3 cache, the send buffer data has more possibility kept into cache and reused in the next loop.

On Magny-Cours processor, as on Barcelona processor, non-temporal memory copy routines have better performance. Figure 2(b) shows that "MV2-1.5-128-nt" can get 51% improvement for single-pair bandwidth testing. However, for multi-pair bandwidth performance as shown in Figure 5(b), non-temporal only achieve similar bandwidth as standard memcpy() function. One possible reason is that: we use die 0 and die 3 of Magny-Cours in the experiments, and there is x8cHT (Hyper Transport) connection between these two dies instead of x16cHT for dies intra socket. When all six cores are used in multi-pair tests, it is hard to improve bandwidth considering the upper bandwidth limitation 4GB/s which is tested on our Magny-Cours processor with STREAM benchmark [15] even after we upload the latest BIOS version.

On Nehalem processor, non-temporal routines improve multi-pair bandwidth peroformance in Figure5(c). "MV2-1.5-128-nt" gets up to 31% performance. But for single-pair test, Figure2(c) illustrates that non-temporal routines can't



(a) Barcelona (b) Magny-Cours (c) Nehalem

Fig. 7. Multi-Pair Latency Large Message Performance

get better performance. We will explain this phenomenon in Section IV-C, with a discussion including cache miss number and efficient bandwidth utilization issue on Nehalem processor.

2) Latency Performance: Figure 3, Figure 4, Figure 6, and Figure 7 show the performance of single-pair and multi-pair latency for the small message and the large message respectively.

Similar to bandwidth performance, on Barcelona system, "MV2-1.5-128-nt" can achieve 18% for single-pair test and 17% for multi-pair test at 1MB message. However, for small message less than 1KB, 32 bit non-temporal memory copy adds an 0.7 mirco-second extra overhead.

On Magny-Cours processor, for non-tmporal routines, we observe the improved latcny for one-pair test, but there is no significant improvement for multi-pair test, as same as multi-pair bandwidth test.

Also similar to bandwidth performance, on Nehalem processor, non-temporal memory copy performs even worse for single-pair benchmarks.

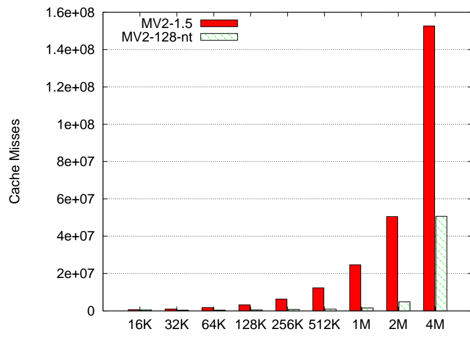
C. Measuring Cache Miss and Bandwidth Usage

Our work focuses on the intra-node communication, so cache effect and bandwidth requests play an important role, especially for non-temporal memory copy routines. Considering the performance variation on Nehalem processor, we use the Linux tool perfmon2 [16] to measure L3 cache misses and bandwidth request for the single-pair and multi-pair bandwidth test. Linux kernel version is upgraded to 2.6.30 to support perfmon tool in this section. The test has three pair processes for multi-pair bandwidth test to avoid sampling disturbed on core 0. Since we don't change memory copy on the receive side, we don't compare cache effect on this side. On the send side, it is the send buffer data that filled and reused in L3 cache will affect the performance; so the read cache miss and bandwidth usage are measured. We use the following events to get the bandwidth usage: 1) `UNC_QMC_NORMAL_READS.ANY`: Read requests for quickpath memory controller; 2) `UNC_QMC_WRITES.FULL.ANY`: Full cache line write requests to DRAM; 3) `CPU_CLK_UNHALTED`: Number of cycles during which the processor is not halted. We name V_read for `UNC_QMC_NORMAL_READS.ANY`, V_write for `UNC_QMC_WRITES.FULL.ANY`, CLK for `CPU_CLK_UNHALTED`. The bandwidth usage can be calculated from this formula: $(V_{read} + V_{write}) * cache_line_size/CLK$

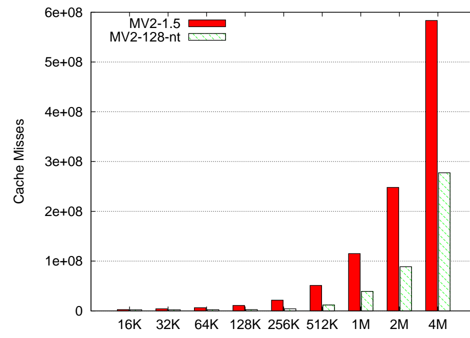
Figure 8 compares L3 read cache miss number and figure 9 compares bandwidth usage with varying message sizes. Figure 9 illustrate the bandwidth usage for "MV2-1.5" can be up to 3.5GB/s in osu_bw test and 6.2GB/s in osu_mbw_mr test. On Nehalem processor, the most inter sockets bandwidth with STREAM benchmark is 9GB/s. When bandwidth usage is at a low level, as illustrated in Figure 9(a), which is 3.5GB/s in osu_bw and smaller than 40% peak bandwidth, the read cache miss number cannot determine the performance, since there isn't bandwidth contention to fill data into cache. As a result, in osu_bw, even though "MV2-1.5-128-nt" has lower cache misses, there is not much performance compared with "MV2-1.5". When bandwidth usage is large enough, as illustrated in Figure 9(b), which is 6.2GB/s in osu_mbw_mr and larger than 65% peak bandwidth, the reduced read cache misses improve the performance significantly.

D. Intel MPI Benchmarks

For IMB benchmarks, we run Allgather, Alltoall, and Allreduce benchmarks to test collectives interfaces performance. We use 16 processes on Barcelona processor and Magny-Cours processor, and 8 processes on Nehalem processor. MVAPICH2 offers flexible CPU mapping schemes that allow users map specific process to specific core, based on the application characteristics. The "bunch" scheme involves mapping processes as close as possible, such that they are on the same socket and can share caches. The "scatter" scheme involves binding MPI process with rank

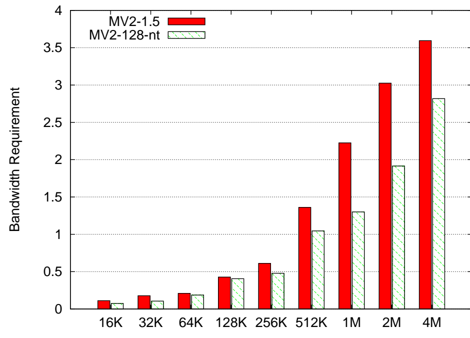


(a) Bandwidth

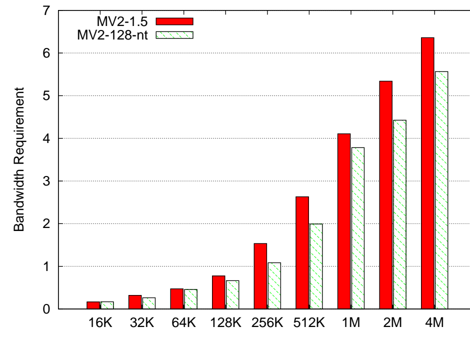


(b) Multi-pair Bandwidth

Fig. 8. Read Cache Miss Analysis

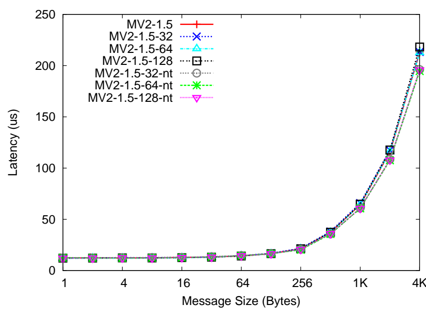


(a) Bandwidth

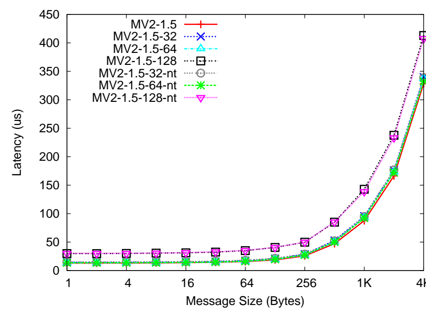


(b) Multi-pair Bandwidth

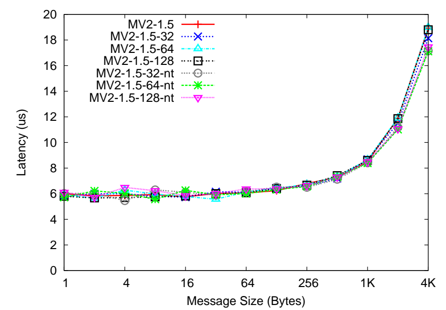
Fig. 9. Bandwidth Requirement and Effectiveness



(a) Barcelona

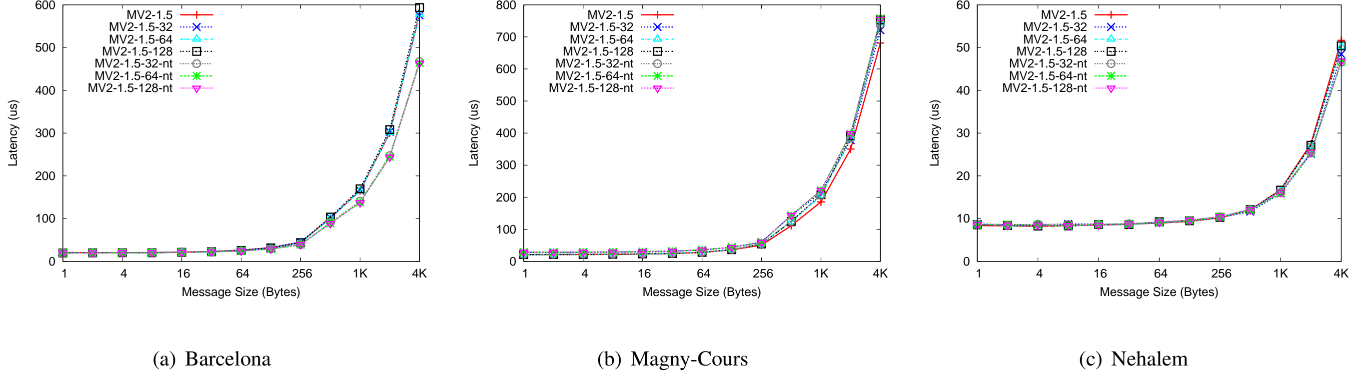


(b) Magny-Cours



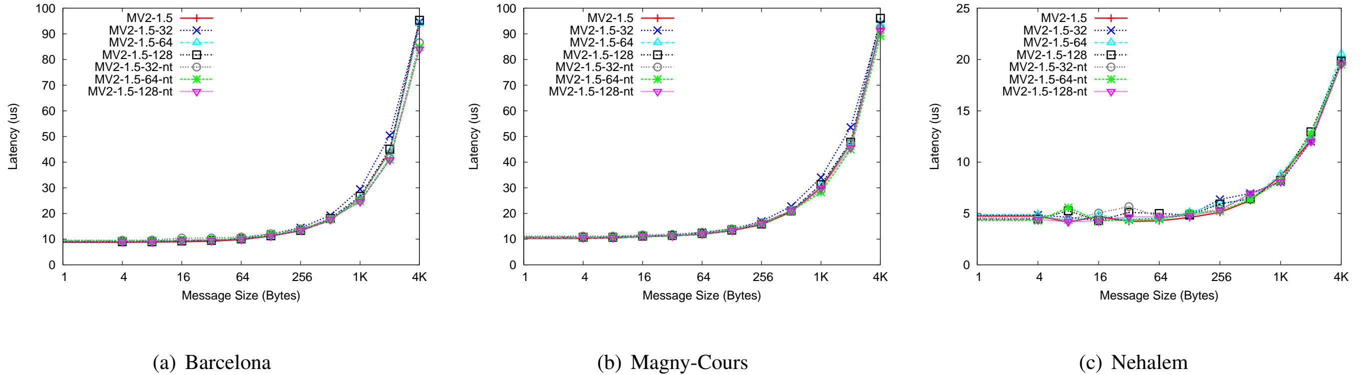
(c) Nehalem

Fig. 10. Allgather(Scatter) Small Message Latency



(a) Barcelona (b) Magny-Cours (c) Nehalem

Fig. 11. Alltoall(Scatter) Small Message Latency



(a) Barcelona (b) Magny-Cours (c) Nehalem

Fig. 12. Allreduce(Scatter) Small Message Latency

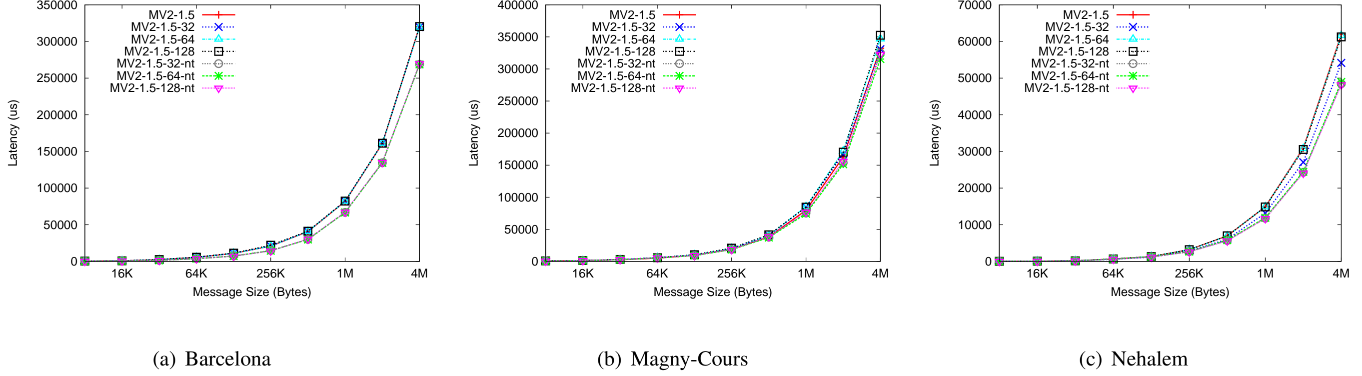
i to the socket ($i\%numberofsockets$). This ensures that neighboring MPI processes get mapped to different sockets and do not share cache. We would like to note that these CPU mapping schemes are different from the MPI_Scatter operation in the MPI Standard. Since the figures for the small message size illustrate the simillar performance comparing as those for the large message size, we only focus on those for the large data size in the evaluation part of this sub section.

1) Allgather: Figure 13 illustrates the performance of Allgather benchmark for the large message size on the three platforms. From Figure13(a) and Figure13(c), we observe that "MV2-1.5-128-nt" achieves up to 15% and 22% improvement compared with "MV2-1.5".

The performance improvement by non-temporal memory copy is determined by collectives interfaces' implementation and processes binding policy. For example, Barcelona processor includes 4 sockets each hosting 4 cores. In MVAPICH2, Allgather uses Bruck [17] algorithm for small messages: in step k ($0 <= k < \lceil lgp \rceil$), rank i process receives a message from rank $(i + 2^k)$ process and sends a message to rank $(i - 2^k)$ process; and Allgather uses Ring based algorithm for large messages: in each step, rank i process gets a message from rank $(i+1)$ process and sends a message to rank $(i - 1)$ process. When "scatter" policy is used, rank i is bound to socket j where j is modulo of i over 4. For small messages, in the first two steps, rank i communicates with the processes located in other sockets, and in the last two steps, rank i communicates with the processes in the same sockets. So, non-temporal memory copy improve the performance not obviously; and we don't show collectives small message figures in this section. For large message, in each step, rank i communicates with rank $(i + 1)$ and rank $(i - 1)$ located in other sockets; and all communications are inter sockets communications, always being improved by non-temporal memory copy.

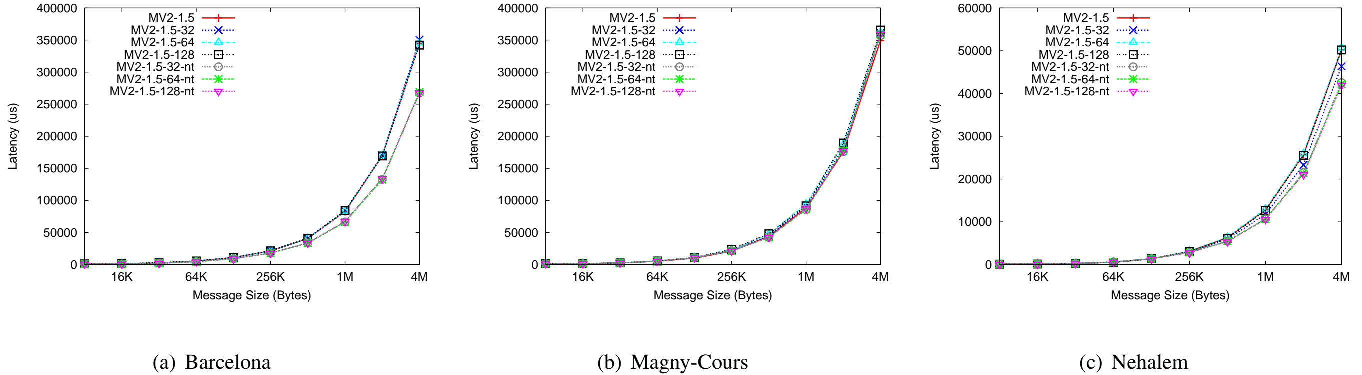
Figure 13(b) shows that non-temporal memory copy can only achieve 2% improvement, which is consistent with the multi-pair latency performance in Section IV-B: due to 16 processes scatted into 4 dies, x8cHT is used. We will do further investigation on Magny-Cours processor in the future.

2) Alltoall: In Figures 14, we compare the performance of MPI_Alltoall with different meomory copy routines for large messages. In MVAPICH2, we use the pair-wise exchange algorithm for large messages for MPI_Alltoall, in



(a) Barcelona (b) Magny-Cours (c) Nehalem

Fig. 13. Allgather(Scatter) Large Message Latency



(a) Barcelona (b) Magny-Cours (c) Nehalem

Fig. 14. Alltoall(Scatter) Large Message Latency

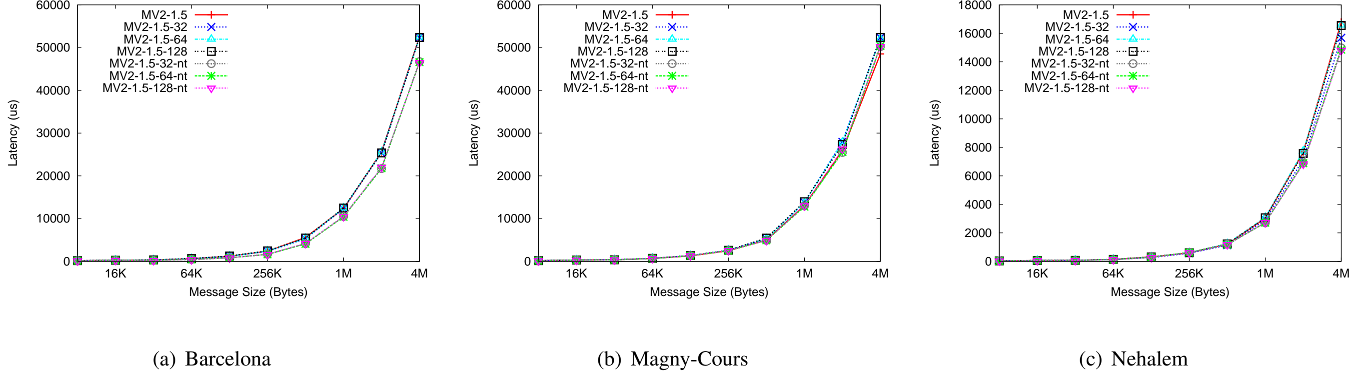
which a processes with rank i sends and receives distinct messages from process with rank $(iXORj)$, in iteration j of the exchange. The number of inter-socket exchanges performed by each process remains the same whether we use the "bunch" CPU mapping or the "scatter" mapping and we did not see any difference in our experiments. For uniformity, we have chosen to demonstrate the performance comparison with the "scatter" mapping across the different architectures. In Figure 14(a), we observe that "MV2-1.5-128-nt" can deliver about 30% improvement at 16KB data size and about 20% improvement at 4MB data size on Barcelona processor. In Figure 14(c), we observe up to 16% improvement of "MV2-1.5-128-nt" on Nehalem processor. We observe that the benefits are much smaller on Magny-Cours processor in Figure 14(b), which is similar to the results that we have observed with this platform, thus far.

3) Allreduce: In MVAPICH2, we use the multi-core aware shared-memory based algorithm for small messages. For medium and larger messages, we use a point-to-point based algorithm that first performs the "reduce-scatter" operation, followed by an "Allgather" exchange to ensure that all the processes have the globally reduced data buffer. We use the recursive-doubling algorithm for both these operations. As indicated in Section IV-D1, some of the exchanges occur within the same socket, while the rest of them involve inter-socket exchanges. Hence, we expect to see some performance benefits with the inter-socket exchange being optimized with non-temporal memory copy routines. In Figure 14(a) and Figure 14(c), we observe up to 20% improvement in latency on Barcelona processor and up to 10% improvement on Nehalem processor.

E. Performance with Application Kernels

We choose several benchmarks from NAS benchmarks suite and HPCC benchmark suite to evaluate non-temporal memory copy routines comparing with standard memcpy() function. The results are shown in Table I, which is divided into three sections for three different platform: Barcelona, Mangy-cours and Nehalem. To save space, we fill column of "1.5" the actual executaion time or bandwidth number for the standard memcpy() function. In "32nt", "64nt" and "128nt", we show the improvement percentage by 32 bit, 64 bit and 128 bit non-temporal memory copy routine.

1) NAS Benchmarks: The NAS Parallel Benchmarks (NPB) [18] are a small set of application kernels designed to help evaluate the performance of parallel supercomputers.



(a) Barcelona (b) Magny-Cours (c) Nehalem

Fig. 15. Allreduce(Scatter) Large Message Latency

		NAS			PTRANS		RandomRing	
		IS	CG	FT	Lat	Bw	Lat	Bw
Barcelona	1.5	4.1	68.86	64.49	0.61	1.34	3.11	0.23
	32nt	8.78%	3.40%	3.66%	2.58%	-0.17%	0.84%	14.82%
	64nt	8.78%	3.53%	3.85%	3.60%	1.91%	2.59%	14.66%
	128nt	8.54%	2.92%	4.12%	2.61%	1.40%	1.88%	14.23%
Magny-cours	1.5	3.46	37.38	49.48	0.84	0.98	3.10	0.21
	32nt	1.45%	0.27%	-2.36%	6.24%	4.71%	-1.96%	6.51%
	64nt	1.73%	0.51%	-1.46%	5.97%	4.30%	3.36%	4.62%
	128nt	1.45%	-0.88%	-1.64%	5.42%	3.12%	3.56%	6.65%
Nehalem	1.5	3.22	43.67	63.25	2.00	0.42	1.93	0.85
	32nt	4.35%	0.16%	1.45%	2.12%	3.37%	-0.99%	-3.00%
	64nt	4.66%	0.27%	0.51%	1.47%	4.39%	2.20%	1.91%
	128nt	4.35%	0.18%	1.39%	2.11%	7.26%	1.17%	3.53%

TABLE I. Application evaluation

We focus on CG, FT and IS for our application evaluation since these application kernels spend significant portion of their time in MPI communication and use large messages.

From Table I, we can observe that, on Barcelona and Nehalem, the non-temporal memory copy routines can perform over the standard memcpy() function in most of the situations, especially for IS benchmark, where collective operations take a large portion in communication stage. Also, on Magny-cours system, though non-temporal memory copy routine can achieve same or slightly better performance for IS and CG benchmark, for FT benchmark, it loses about 2% performance. We are investigating the reason.

2) HPCC Benchmarks: HPCC is a benchmark suite which is used to examine the performance of HPC architectures that stress different aspects of HPC systems involving memory and network in addition to computation [19]. We utilized two benchmarks inside HPCC testing suite to measure the performance of non-temporary memory copy function: PTRANS and RandomRing (based on b_eff benchmark).

PTRANS measures the rate of transfer for larges arrays of data from multiprocessor's memory. The size of matrix for PTRANS is 20480 and the blocking factor is 80. From the result table, we can find out that the best latency result comes on Magny-cours with 32 bit non-temporal memory routine and the best bandwidth result comes on Nehalem with 128 bit non-temporal memory routine.

Random Ring measures latency and bandwidth using a random ordering where communication takes place between processes with adjacent ranks in the communicator. For Random Ring bandwidth, non-temporal memory copy routine can achieve about 14% better performance on Barcelona, about 6% on Magny-cours and about 3% on Nehalem, which is consistent with micro-benchmark results that we have discussed in Section IV-B. But for latency performance, the non-temporal memory copy routine can only achieve -2% to +3% improvement.

V. Related Work

Over the past several years, several researchers have been studying the performance of communication inside a node. As the multi-core phenomena has gathered more momentum, this topic is getting more and more attention.

Subramoni et. al. have studied the performance differences of various memory moving instructions for intra-socket and inter-socket communication in [20]. The study focuses on only basic communication costs, as opposed to performing a thorough analysis of various methods in the context of MPI communication, as is done in this paper. In [21], Chai et. al. present a technique to reduce memory utilization for shared memory message passing, and improve performance for large messages on NUMA platforms. Our work is directly applicable to this past work by enhancing the core memory copy routines that were utilized by Chai, et. al. In [22], Buntinas et. al examined large intra-node message transfers and utilized 32-bit non-temporal instructions. Non-temporal instructions are also explored in the context of MP_Lite in [23] by Chen et. al. In [24] Borg has explored the choice of memory copy routines in a dynamic fashion. In this paper, compared with [22] and [23], we extend the concept and use more optimal quad-word non-temporal instructions, and we provide a dynamic memory copy routines selection mechanism. Compared with [24], we also highlight the dynamic memory copy routines selection due to architectural advances, the choice of optimal memory copy function is determined at run-time based on process-core-cache location. Compared with all these related work, as far as we are aware, we are the first to optimize MPI receiver side cache utilization through SSE4.1 non-temporal load instruction, and the first one to do a thorough performance investigation on the latest multi-core processors.

VI. Conclusion and Future Work

In this paper, we presented performance analysis of a state-of-the-art MPI messaging stack, MVAPICH2 enhanced with streaming SIMD non-temporal memory copy instructions. We performed on our design and evaluation on modern multi-core platforms with multiple levels of cache hierarchy. We have designed a framework for choosing optimal memory copy routines based on cache hierarchy shared by communicating cores. Our evaluation reveals that performance can be improved by 60% and 13% for 64K message sizes over the default shared memory implementation for the multi-pair bandwidth, and by 2% to 14% for NAS benchmark and HPCC benchmark on AMD Barcelona and Intel Nehalem architectures.

We intend to continue working in this direction. The memory copy routines developed in this paper will be integrated into the public MVAPICH2 releases. We also plan to evaluate the impact of our designs on large scale performance runs of real applications.

VII. Acknowledgements

This research is supported in part by U.S. Department of Energy grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; National Science Foundation grants #CNS-0403342, #CCF-0702675, #CCF-0833169 and #CCF-0916302; grants from Intel, Mellanox, Cisco, QLogic, and Sun Microsystems; Equipment donations from Intel, Mellanox, AMD, Appro, Chelsio, Dell, Fujitsu, Fulcrum, Microway, Obsidian, QLogic, and Sun Microsystems.

References

[1] AMD Magny Cours. http://blogs.amd.com/work/tag/magny-cours/.
[2] Intel, "Intel Nehalem." http://www.intel.com/technology/architecture-silicon/next-gen/.
[3] H.-W. Jin, S. Sur, L. Chai, and D. K. Panda, "Lightweight Kernel-level Primitives For High-performance MPI Intra-node Communication Over Multi-core Systems," in *Cluster 2007*, 2007.
[4] High Performance Runtime Systems for Parallel Architectures, LaBRI, INRIA Bordeaux - Sud- Ouest, "KNEM: High-Performance Intra-Node MPI Communication." http://runtime.bordeaux.inria.fr/knem/.
[5] Tilera. http://www.tilera.com/products/processors/TILEPRO64.
[6] Intel Corporation. http://www.intel.com/technology/quickpath/.
[7] Hyper-Transport. http://www.hypertransport.org/.
[8] MVAPICH2: High Performance MPI over InfiniBand /10GigE /i-WARP and RDMAoE. http://mvapich.cse.ohio-state.edu/.
[9] MPICH2: High Performance and Widely Portable MPI. http://www.mcs.anl.gov/research/projects/mpich2/.
[10] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, (Budapest, Hungary), pp. 97–104, September 2004.
[11] "Intel 64 and IA-32 Architectures Software Developers Manual Volume 2A: Instruction Set Reference." http://www3.intel.com/Assets/PDF/manual/253666.pdf.
[12] Venkatesh Pallipadi and Suresh Siddha, "PATting Linux," in *Proceedings of the Linux Symposium*, (Ottawa, Ontario, Canada), July 23rd26th, 2008.
[13] "Portable Hardware Locality (hwloc)." http://www.open-mpi.org/projects/hwloc/.
[14] OSU Micro-benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.
[15] "STREAM: Sustainable Memory Bandwidth in High Performance Computers." http://www.cs.virginia.edu/stream/.
[16] "perfmon2: the hardware-based performance monitoring interface for Linux." http://perfmon2.sourceforge.net/.
[17] J. Bruck, C.-T. Ho, S. Kipnis, and D. Weathersby, "Efficient algorithms for all-to-all communications in multi-port message-passing systems," in *SPAA '94: Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, (New York, NY, USA), pp. 298–309, ACM, 1994.
[18] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks," *The Int. J. of Supercomputer Applications*, vol. 5, pp. 63–73, Fall 1991.
[19] HPC Challenge Benchmarks. http://icl.cs.utk.edu/hpcc/.

[20] H. Subramoni, F. Petrini, V. Agarwal, and D. Pasetto *Computer Architecture Letters, title=Intra-Socket and Inter-Socket Communication in Multi-core Systems*, vol. 9, pp. 13 –16, jan. 2010.

[21] L. Chai, A. Hartono, and D. K. Panda, "Designing Efficient MPI Intra-node Communication Support for Modern Computer Architectures," in *IEEE Conference on Cluster Computing*, 2006.

[22] D. Buntinas, G. Mercier, and W. Gropp, "Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication Subsystem," in *Proceedings, 13th European PVM/MPI Users' Group Meeting*, (Bonn, Germany), September 2006.

[23] X. Chen and D. Turner, "Efficient message-passing within smp systems," in *Proceedings, 10th European PVM/MPI Users' Group Meeting*, (Venice, Italy), September 2003.

[24] Ø. Borg, "Dynamic Selection of MPI Intra-copy Routines Based on Program Characteristics," Master's thesis, NTNU - Trondheim Norwegian University of Science and Technology, 2006.