2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

Analysis of MPI Shared-Memory Communication Performance from a Cache Coherence Perspective

Bertrand Putigny*, Benoit Ruelle†, Brice Goglin*
* *Inria Bordeaux – Sud-ouest, France*
† *Bordeaux Polytechnic Institute, France*
Bertrand.Putigny@inria.fr, Benoit.Ruelle@ipb.fr, Brice.Goglin@inria.fr

***Abstract*—Shared memory MPI communication is an important part of the overall performance of parallel applications. However understanding the behavior of these data transfers is difficult because of the combined complexity of modern memory architectures with multiple levels of caches and complex cache coherence protocols, of MPI implementations, and of application needs.**

We analyze shared memory MPI communication from a cache coherence perspective through a new memory model. It captures the memory architecture characteristics with micro-benchmarks that exhibit the limitations of the memory accesses involved in the data transfer. We model the performance of intra-node communication without requiring complex analytical models. The advantage of the approach consists in not requiring deep knowledge of rarely documented hardware features such as caching policies or prefetchers that make modeling modern memory subsystems hardly feasible.

Our qualitative analysis based on this result leads to a better understanding of shared memory communication performance for scientific computing. We then discuss some possible optimizations such as buffer reuse order, cache flushing, and non-temporal instructions that could be used by MPI implementers.

I. Introduction

The performance of MPI communication in parallel scientific applications is often a key criteria for the overall software performance. Communication tuning has often been investigated for achieving better performance. Indeed, most MPI implementations adapt their communication strategies to the underlying architecture and to the operation parameters. For instance processes running of the same node communicate through shared memory instead of through the network interface.

Communication inside nodes usually relies on two memory copies across a shared-memory buffer. These copies involve cache coherence mechanisms that have an important impact on the actual performance of memory transfers. Unfortunately, MPI implementations tune shared memory communication strategies based on metrics that rarely take caches into account, merely by considering their sizes. Tuning of shared memory communication actually requires understanding the performance implications of cache coherence. Apprehending this impact can be cumbersome because modern memory architectures are increasingly complex, with multiple hierarchical levels of shared caches.

We propose a method based on memory micro-benchmarks to help understanding this impact through a qualitative analysis. Relying on benchmarks to provide memory hierarchy insights avoids building very complex analytical models that are platform dependent. Our model improves productivity by remaining the same across different platforms. It only captures the low-level (and hidden) details of the architecture during measurements. Based on this model, the knowledge of buffer states in caches, and micro-benchmarks, we show that tuning shared memory communication depending on the cache coherence protocol can be a source of optimization ideas. This can ease prototyping of communication strategies through shared memory.

The remainder of the paper is organized as follows. Section II presents the scope and context of our work. Section III describes how cache coherence is actually involved in shared-memory MPI communication and how our model lets us analyze it. The model is then evaluated in Section IV and it leads us to discuss several possible optimization ideas. Related work is finally described in Section V before concluding.

II. Optimizing intra-node MPI communication

A. On the Importance of Intra-node Communication

Intra-node communication had been involved in scientific computing long before the rise of multicore processors and many-cores nodes. Dual-processor servers were already considered to have one of the best performance ratios ten years ago. For instance, Thunderbird, one of the last very large clusters based on single-core processors, reached the fifth rank of the Top500 (http://top500.org) in 2005 while using two single-core processors per node.

Since the advent of multicore processors, more than 75% of the Top500 are now clusters of dual-processor nodes with at least 4 cores per processor, making intra-node communication even more critical to the overall performance. Indeed, the increasing importance of locality in modern servers causes most applications to communicate more with their local neighbors. Users are advised to map processes according to their affinities so as to benefit from cache-sharing, intra-node communication or reduced network distance as much as possible.

Most modern parallel applications still communicate through the *de facto* standard, MPI. And it is expected that a wide amount of data transfer between processes happens inside a single node, or even inside a single processor socket under a shared cache. This makes shared-memory communication increasingly critical to the overall parallel application performance. Moreover, coprocessors such as the Intel Xeon Phi may again emphasize this trend since they can run tens of MPI processes each.

978-1-4799-4116-2/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPSW.2014.139

B. Too many Configuration Options

Intra-node communication has been the subject of many research works since the advent of the MPI standard twenty years ago. Multiple data transfers strategies have been proposed [1], including relying on the external network interface, on specific network drivers, on custom operating system features [2], or on user-level techniques such as shared buffers and pipelining. This was still an active research area recently through platform-independent direct-copy mechanims such as LiMIC [3] and KNEM [4], and the inclusion of *Cross Memory Attach* [5] in the Linux kernel.

One common issue with all these strategies is: How to select the right one? None of them is the best for all communication patterns because small messages, large messages and collective operations do not have the same performance behavior and needs. Hardware characteristics such as the cache size has been used as a basic way to infer message size thresholds for switching between strategies [6], [3], [7]. However this appeared not to work properly for collective operations where memory contention becomes a critical factor [8]. Moreover some of these strategies also require the tuning of their internal parameters such as the pipeline chunk size.

Some MPI implementations such as Open MPI offer many configuration options for tuning intra-node communication. However proper tuning requires deep understanding of the implementation (how it actually transfers data), of the application (how it manipulates buffers, whether it needs overlap, etc.), and of the hardware memory architecture behavior (how memory accesses are implemented). It means that most users cannot actually tune this software, and even many developers unless they have all this knowledge.

Most users therefore do not complain that some implementations such as MPICH2 do not offer many explicit tuning options. However users often assume that MPI is properly tuned internally. Unfortunately this is hardly feasible given the aforementioned software and hardware complexity. Most MPI implementations just use hardwired default thresholds that were chosen five years ago. With tens of cores and much larger caches in modern platforms, the old defaults are likely far from optimal on todays platforms. Users may therefore complain about the observed performance and competitors may easily find cases where a carefully-tuned case-specific change improves performance.

C. The Need for a Better Understanding of Intra-node Communication

Proper automatic tuning of intra-node MPI communication strategy is very difficult because it depends on many factors: Is the transfer running alone on the machine or is it part of a large parallel communication scheme causing contention? Does the application want overlap? Does the hardware efficiently support these needs? Depending on the answers to these questions, the performance of a communication strategy may vary significantly.

We believe that cache coherence is the key to understanding these behaviors. Cache effects are often used as the easy cause of complex behaviors in memory-bound codes, especially shared-memory communication, without actually explaining them for real. Indeed the characteristics of caches (and of the cache coherence protocols that assembles them) is hidden in the hardware and rarely fully documented. Therefore cache coherence causes effects that cannot be easily modeled or even explained. Indeed we show later in this article that even modeling basic data transfers such as memory copies is difficult.

We want to tackle this problem with a new innovative approach that uses micro-benchmarks as a way to capture the complex behavior of the memory architecture. This idea was initially developed for predicting the behavior of memory-bound applications [**?**]. We present in the next sections how we use it for analyzing and better understanding shared-memory-based intra-node MPI communication.

III. How Cache-Coherence matters to Intra-node MPI Communication

Shared-memory MPI communication requires memory copies between the memory and caches of two cores. The cache-coherence protocol is therefore heavily involved. Most modern HPC platforms use a protocol that is a variant of MESI [10]. Each cache line is in one of the following states:

Modified: The cache line has been modified, and the data in memory is consequently stale. The cache holding this cache line is the only one to hold it.

Exclusive: Data in the cache line is not present in any other cache. This cache line is clean (i.e. the version in main memory is the same).

Shared: This cache line is shared with other caches: other caches can also hold the same data. The cache line is clean (memory and other caches hold the same value).

Invalid: This line is not present in this cache.

Intel and AMD platforms actually use the MESIF[1] and MOESI[2] variants. We detail in this section how the MESI protocol handles shared-memory MPI communication.

A. Anatomy of Intra-node Communication Memory Accesses

Shared-memory MPI communication uses an intermediate buffer that is shared between the sender and receiver processes. The sender process writes the message to the shared buffer before the receiver process reads it. As described on Figure 1, every byte in the transferred message therefore sees the following cache states:

[1]The *Forward* state reduces the traffic by having a single shared-copy reply to bus requests.

[2]The *Owned* state allows direct sharing of dirty cache-lines instead of first requiring a write-back to memory. See Section IV-E.

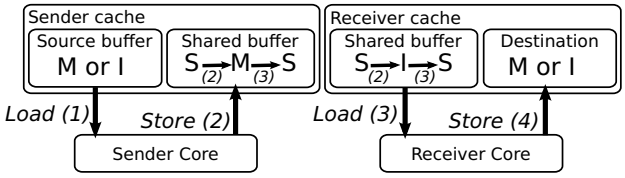


Figure 1. Cache state transitions for the source, shared and destination buffers of both sender and receiver cores during the memory accesses involved in a shared-memory-based data transfer.

1) The sender reads the data from its memory. Temporal locality implies that it may have been generated (written) recently. If so, this step is a *Load Hit* from a local Modified cache-line. If not available in the local caches anymore (either because it was generated a long time ago or because something else filled the caches in the meantime), this is a *Load Miss* that goes up to main memory.
2) The sender then writes the data to the shared buffer. That buffer was used by prior transfers. It is therefore usually available in the local cache as well as in the cache of another core. This is a *Store Hit* to a local Shared cache-line. The cache-line gets evicted from the remote caches and goes to the Modified state in the local caches.
3) The receiver reads the shared-buffer from the sender core. This is a *Load Miss* from a remote Modified cache-line. The remote cache line gets copied in the local caches and both copies switch to the Shared state (this explain the state before step 2).
4) Finally the receiver writes the data to its receive buffer. If the target buffer was recently used, this is a *Store Hit* (usually to a local *Modified* cache-line). Otherwise it is a *Store Miss* to main memory.

Most modern MPI implementations follow this model. MPICH2 [11] and Open MPI [12] both allocate one large buffer shared between all local processes. It is then divided into one set of fixed-size buffers (chunks) per sender. It means that each process always reuses the same buffers for all transfers, even toward different destination processes. We will see in Section IV-E that this reuse behavior is actually relevant on recent AMD platforms. Other strategies exist for various kinds of communication (for instance dedicating one larger buffer to each directed connection, etc.), but we will focus on this one when describing our model.

When the message is larger than fixed-size buffers, multiple ones are used and a pipeline protocol makes sure the receiver can read previous buffers while the sender fills the next ones. MPICH2 uses 64 kB *cells* while Open MPI uses 32 kB *fragments*[3] by default. As depicted in Table I, this pipelined model means that there may be 4 concurrent memory accesses during a single transfer: Sender and receiver cores can execute their own copy in parallel. Each copy involves loads and stores that can be executed in parallel by modern cores. We will analyze the actual parallelism in Section IV-A).

[3]Open MPI uses a smaller first fragment so that the receiver can prepare the receiving of the next fragments before they actually arrive.

Table I
MEMORY ACCESS PARALLELISM DURING A PIPELINED TRANSFER WHEN THE MESSAGE IS DIVIDED INTO 3 CHUNKS AND THE PROCESSOR CAN EXECUTE ONE LOAD AND ONE STORE IN PARALLEL.

Time step	Sender Core	Receiver Core
1	Load + Store (chunk #1)	
2	Load + Store (chunk #2)	Load + Store (chunk #1)
3	Load + Store (chunk #3)	Load + Store (chunk #2)
4		Load + Store (chunk #3)

B. Impact of the Cache Coherence on Memory Access Performance

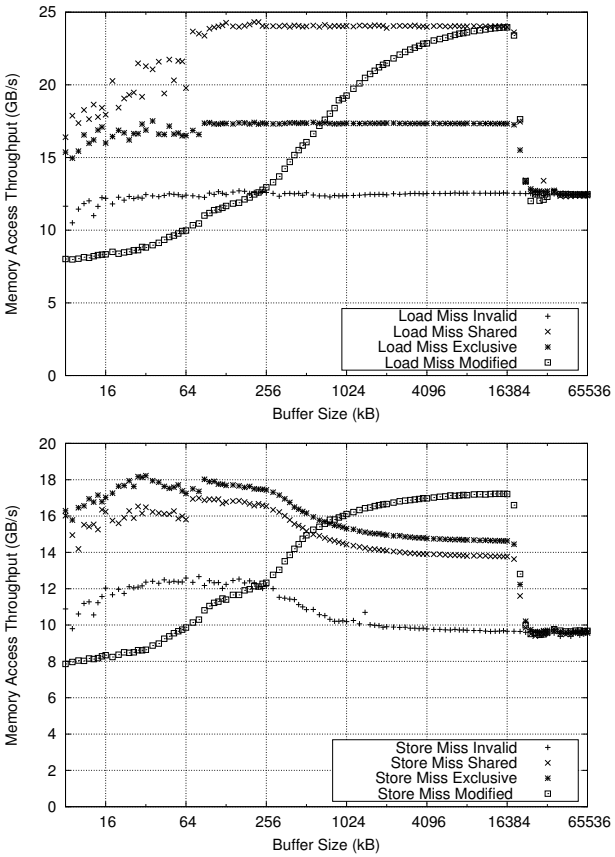


Figure 2. Load miss and store miss access throughput depending on the target buffer size and cache state, inside a 2 GHz 8-core Intel Xeon E5-2650 socket.

Figure 2 presents the memory access throughput when the target buffer is not in the local caches (load or store *miss*) but may be in some other caches in the socket. It shows that the

observed read throughput may vary by a factor of 2 while the write throughput may vary by a factor of 2.5 depending on the buffer size and on the caching state. This result reveals that caches and cache-coherence protocols have a deep impact on the application performance. Difference of similar magnitude can be observed in case of cache hits.

Counting the number of cache misses is a widely used technique for analyzing or even predicting performance, either for shared-memory communication [6], [13] or in a more general profiling context [14]. However, the above graphs show that this number cannot actually be used as a reliable source for accurate performance prediction because neglecting the caching state is not acceptable. We intend to take this information into account for a better model and understanding of MPI shared-memory communication.

We explained in the previous section that a transfer is made of 4 memory accesses that are executed in parallel. The figure shows that the performance of each of them is deeply related to the involved cache states. We are now going to explain how to combine them to model the overall transfer performance.

C. Modeling Communication by Combining Micro-benchmarks

The idea behind our memory model is that modern memory architectures make accurate analytical models too difficult because many hardware features, such as prefetchers or cache coherence implementations, are too complex and often not documented. We therefore chose to hide them inside the output of micro-benchmarks that capture the actual behavior of the hardware without having to understand and describe it for real. These benchmarks are run using our mbench framework.[4] It offers easy ways to setup memory buffers to specific cache states and compute the corresponding memory access throughputs for different numbers of threads. Measuring the memory throughput for different buffer sizes also capture the performance and sizes of each level of cache, which allows us to explicitly ignore them in the model.

Benchmark outputs are then combined to rebuild the application memory access pattern and predict its behavior with respect to scalability, buffer sizes, etc. This rebuilding considers involved buffers in the MESI cache-coherence protocol so as to find out the right performance of each buffer access. Given that most HPC architectures use a variant of the MESI protocol, we expect the model to match a wide range of HPC platforms.

The model targets memory-bound applications, i.e. where memory access is the key performance criteria and cannot be overlapped significantly with computation. More details can be found in [**?**]. We use it here for a better understanding of MPI shared memory communication based on the memory pattern described in section III-A. Each step translates into a benchmark output as listed in Table II.

Table II
TRANSITIONS INVOLVED IN OUR MODEL FOR EACH TRANSFER STEP.

Step	Core	State transition
1	Sender	Load Hit Modified if recently generated, Load Miss Modified otherwise
2	Sender	Store Hit Shared
3	Receiver	Load Miss Modified
4	Receiver	Store Hit Modified if recently used, Store Miss Modified otherwise

Given a message of size M and a maximal pipeline chunk of size C, there are $n = \lfloor M/C \rfloor$ chunks of size C_i (usually the first and/or last chunks are smaller than C if M is not an exact multiple of C). The overall transfer time is estimated to

$$T = S(C_1) + \sum_{i=2}^{n} max\Big(S(C_i), R(C_{i-1})\Big) + R(C_n) \quad (1)$$

where S and R are the times to copy a chunk on the sender and receiver side respectively. When there is a single chunk, the sender and receiver times are added: the overall time is a sequential aggregation of both sides. When there are many chunks[5], the first and last terms can be neglected, and the overall duration is the maximum of the sender and receiver copy times.

Finally our model allows us to estimate S and R from our benchmark outputs. Since modern processors can execute one load and one store at the same time, the duration of a copy should be the maximum of the corresponding loads and stores. For instance, copying from a recently generated source buffer to the shared buffer takes the maximum of a *Load Hit Modified* and a *Store Hit Shared* duration. Given that the throughput of these operations vary with the buffer size (because of caches), we still have to discuss whether each single chunk is copied at the throughput predicted for its own size (C_i) or for the entire message size (M). We chose the later because all chunk operations are executed consecutively and therefore cause memory and cache contention just like if the entire message was manipulated at once.

For instance if the source and destination buffers have been used recently, S and R are estimated to

$$S(C_i) = \frac{C_i}{min(LHM(M), SHS(M))} \quad (2)$$

$$R(C_j) = \frac{C_j}{min(LMM(M), SHM(M))} \quad (3)$$

where LHM, SHS, LMM and SHM are the benchmark-measured throughputs for a *Load Hit Modified*, *Store Hit*

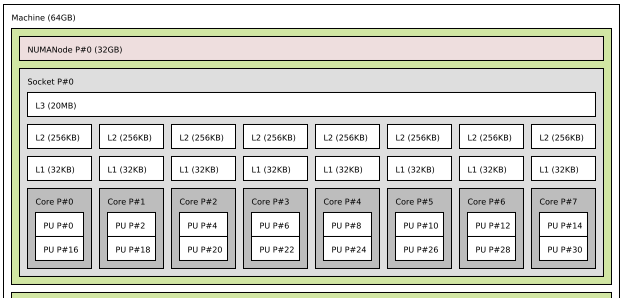
[4]The mbench framework is available for download from https://github.com/bputigny/mbench. Only the most relevant benchmark outputs are included in the paper.

[5]A 1 MB message uses 32 chunks in Open MPI and 16 in MPICH2.

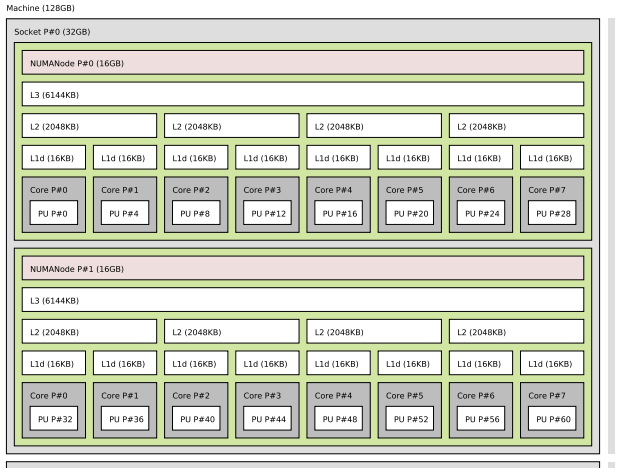
Shared, *Load Miss Modified* and *Store Hit Modified* respectively.

IV. Performance Evaluation and Optimization Hints

We now evaluate our model and use it to discuss some optimization ideas based on the impact of cache-coherence protocols on shared-memory MPI communication.



(a) Intel Xeon *Sandy Bridge* E5-2650.



(b) AMD Opteron *Bulldozer* 6272.

Figure 3. One socket of each kind of node in the evaluation platform.

Our evaluation platform is summarized in Figure 3. It consists in two kinds of nodes. The first contains two 8-core 2 GHz Intel Xeon E5-2650 processors (Sandy-Bridge micro-architecture, 16 cores total, a single Hyper-Thread used per core). The second kind is made of four 16-core 2.1 GHz AMD Opteron 6272 processors (Bulldozer micro-architecture, 64 cores total). CPU frequency scaling as well as Intel Turbo Boost and AMD Turbo CORE technologies were disabled during tests so that the CPU and memory absolute performance does not vary.

A. Evaluation of the Model

To evaluate the model presented in Section III-C we compare its prediction with the performance of an experiment. However our model only predicts the performance of the actual data transfer while MPI implementations add a lot of control code (such as eager message management, rendezvous messages, synchronization) around it. We therefore designed a synthetic experiment that only mimics the data transfer within the Open MPI 1.7 implementation (32 kB pipeline chunks). The performance behavior is similar, but the synthetic program gets higher performance thanks to the removal of the Open MPI control overhead.

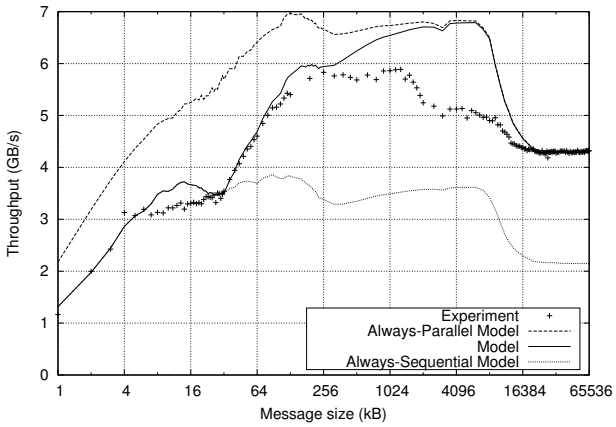


Figure 4. Comparison of the benchmark-based prediction model, the sequential model, the parallel model, and the actual shared-memory transfer. Intel platform.

Figure 4 presents the performance prediction of the model between 2 cores inside the same Intel socket. The top line is the *parallel prediction* which means both sender and receiver copies are executed fully in parallel. This is the asymptotic prediction for large messages. The bottom line is the *sequential prediction* which means copies are performed sequentially by the cores. This is the behavior for small messages when there is a single chunk.

As explained in Section III-C, the prediction model is a mix of these two cases transitioning from one to the other between 32 kB (single chunk) and 4 MB (128 chunks) message sizes. We observed that our model accurately predicts the performance except between 256 kB and 16 MB where the actual experiment is slower. These sizes corresponds to buffers that go into the L3 cache. We explain our misprediction by the fact that the L3 is shared between the two involved cores. It causes contention and capacity misses that our benchmark-based memory model does not really take into account accurately. However, our model works well when the message fits in L1 and L2 cache and in main memory.

One thing that makes our model hard to apply is the difficulty to predict the performance of memory copies that are involved on both sides and accumulated in the analytic formula (S and R functions). Figure 5 presents the prediction of one of the individual memory copies involved in the data transfer. It questions the pre-supposed ability

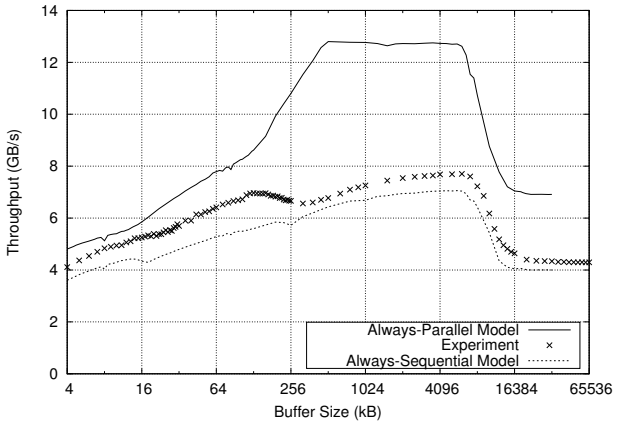


Figure 5. Benchmark-based prediction of the receiver-side memory copy performance. The source buffer was recently written by another core (*Load Miss Modified*) while the destination buffer was recently used locally (*Store Hit Modified*). Intel platform.

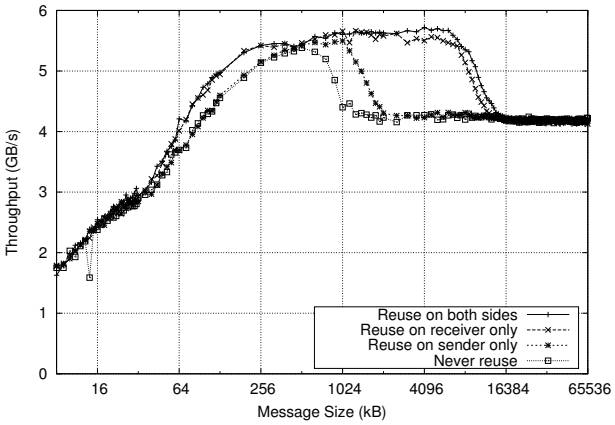


Figure 6. Impact of buffer reuse on IMB Pingpong throughput with Open MPI 1.7.3. IMB was modified to support buffer reuse on one side without the other. Intel platform.

of the processor to perform one load and one store in parallel as explained at the end of Section III-A. Up to 128 kB messages (inside the L1 and L2 private caches), the observed throughput is the parallel bandwidth reduced by 20%. However, for larger messages, in L3 and in main memory, we only measure only 10% above the sequential throughput while the parallel one would be twice higher. Again, this is related to contention in the shared L3 cache and on the memory channels, which do not optimally support heavy parallel loads and stores.

To summarize, our memory model can predict the performance of data transfer, assuming memory copy performance is understood, except when the shared L3 and main memory disturb parallel access performance. This shows why understanding shared-memory communication performance is always difficult: current memory architectures cannot be easily modeled, too many hidden hardware parameters are involved. Overall, we predict the performance behavior but not the absolute value very accurately. Fortunately, this is enough to analyze that behavior and discuss possible optimization hints in the next sections.

B. Impact of Application Buffer Reuse

One common source of mis-understanding of shared-memory MPI communication performance is the reuse (or not) of application buffers in multiple iterations. As explained in Section III, this changes the involved MESI cache states and causes individual memory access performance to vary significantly. It makes performance comparison meaningless when it is not clear whether the same buffers were reused multiple times. Some benchmarks [15] always reuse the same buffer while others such as IMB [16] have options to configure/avoid this reuse. We now look deeper at the actual impact of buffer reuse on the overall transfer time.

Figure 6 compares the throughput depending on buffer reuse on both sides. We observe that the receiver buffer state is much more important than the sender. Unfortunately this result is not convenient for application tuning because locality is easier to maintain on the send side: the application can usually send the data as soon as it is ready, while it often does not receive exactly when it needs it immediately.

The receiver buffer state is more important than the sender because the receiver-side is slower. Therefore improving the sender locality to improve its transfer side will not significantly improve the overall transfer time. Indeed our micro-benchmarks reveal that the receiver side memory accesses (steps 3 and 4 on Table II) hardly pass 15 GB/s for large messages, while the sender side (steps 1 and 2) often achieves close to 20 GB/s.

This imbalance between send and receive side copy durations could be a reason to switch to variable-size pipeline chunks as previously proposed for InfiniBand communication [17]. Unfortunately existing MPI implementations require deep intrusive changes before we could experiment this idea.

C. Load Miss of Sender-written data

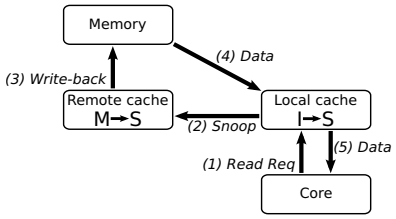


Figure 7. Anatomy of a *Load Miss Modified* (step 3) in the MESI protocol.

We now focus on one of the transfer step that matters to the overall performance: when the receiver loads data that

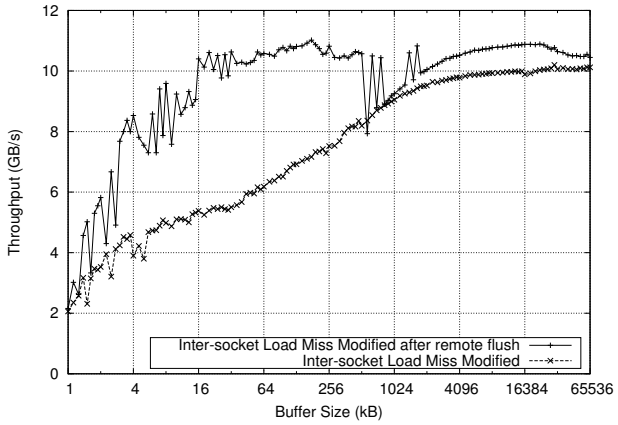


Figure 8. Impact of a flush of modified data on the performance of reading from another core, on the Intel platform.

was previously written by another core (*Load Miss Modified*, step 3). The usual problem with this transition in the MESI protocol is that the remotely-modified data has to be written back to memory before it can be shared by both cores (see Figure 7). If a cache is shared between the cores, the write-back is not actually required. If no cache is shared, for instance when processes run on different sockets, the write-back is required, and Figure 8 confirms that it is expensive: An explicit flush of the remote copy increases the local *Load Miss Modified* by 10% to 100%.

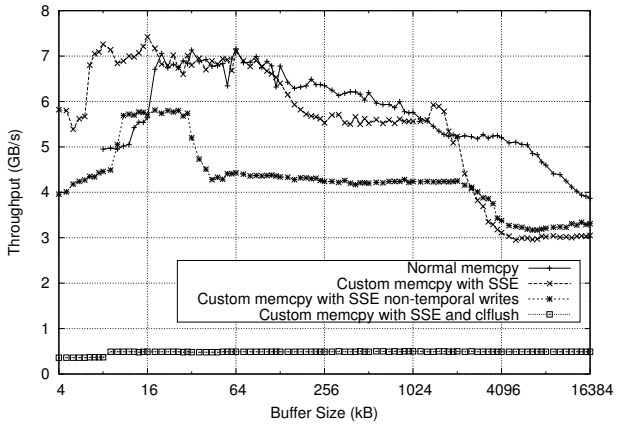


Figure 9. Impact of non-temporal stores and manually flushing on the performance of the sender write step 2, on the Intel platform.

One optimization would consist in moving this expensive remote write-back from the receiver load (step 3) back to the sender store (step 2), by anticipating it using one of the following ideas:

1) The sender could explicitly flush these cache-lines, e.g. with `clflush` x86 instructions. Unfortunately, this severely slows down the sender copy as depicted on Figure 9.

2) The sender could use a larger number of buffers so that the first buffers are automatically evicted when last ones are used. Unfortunately, current processors have very large caches that would require hundreds of buffers for this to work[6]

3) The sender could use non-temporal store instructions to directly reach main memory. This idea has often been considered in the past but very rarely used in production. Figure 9 shows that our custom copy with non-temporal writes is only about 30% slower than the usual copy, so the idea looks indeed interesting. Thus we modified Open MPI to perform a non-temporal store during step 2. However Figure 10 reveals that it actually divides the overall performance by a factor of 2. We could not explain this phenomenon. Unfortunately the behavior of non-temporal instructions with respect to cache-coherence protocol implementations is not widely documented.

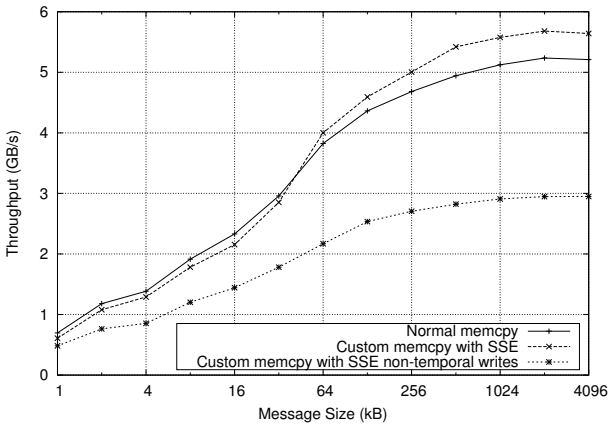


Figure 10. Impact of non-temporal stores in the sender write step 2 on the performance of IMB pingpong between 2 cores on different sockets, on the Intel platform, with a modified Open MPI 1.7.3.

Still, one has to keep in mind that moving the write-back to the sender-side may have the undesirable effect of moving the bottleneck from the receiver to the sender. It is therefore important to make sure that we do not slow the sender down too much. One idea that we are looking at is to force the write-back only when the sender is waiting for the receiver to progress: Once the sender filled all shared-buffers, it may have to wait until the receiver gives some of them back, it may therefore start manually flushing with `clflush` in the meantime.

D. Rewrite of Receiver-read data

We now focus on the other critical transition, when the sender writes to a buffer that was previously used (*Store Hit Shared*). The remote copy has to be invalidated before

[6]640 and 192 buffers of 32 kB are needed on our Intel and AMD platforms respectively.

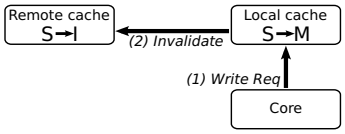


Figure 11. Anatomy of a *Store Hit Shared* (step 2) in the MESI protocol.

the local copy can switch from Shared to Modified (see Figure 11). Fortunately some modern processors such as Intel Xeon E5 feature a directory in their L3 cache so as to filter such invalidation requests when it is known that there are no other copies. So a former remote flushing could reduce the overhead.

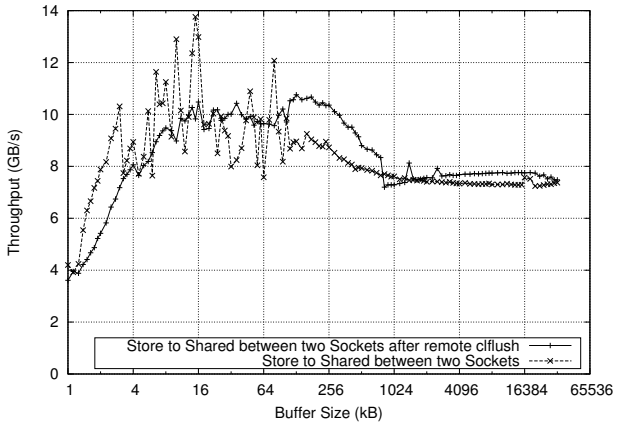


Figure 12. Impact of remote flushing on the performance of a local *Store Hit Shared* on the Intel platform.

Figure 12 shows that this idea could indeed improve performance by 5-10% for significant buffer size (larger than the 64 kB L1, we do not know why the graph is not smooth for smaller buffers). So one could think of adding some flushing on the receiver side. However, as discussed in the previous section, this would slow down the receiver bottleneck even more.

Another problem to consider here is that flushing instructions such as `clflush` may also flush lines out of other core caches that are below a higher-level inclusive shared cache, which would further degrade performance. For instance it would flush out all copies inside the entire Intel socket on our platform because the L3 is inclusive. On AMD, only the L2 is *mostly*-inclusive. This idea should therefore only be considered when the MPI implementation knows for sure that the involved cores do not shared an inclusive cache.

To summarize, optimizing the *Store Hit Shared* state (2) is hardly feasible in the context of the MESI protocol. However we have to revisit this result in next section due to certain characteristics of MESI variant implementations.

E. Shared-buffer Reuse Order and MOESI Protocol

AMD platforms use the MOESI protocol that was (notably) designed to ease sharing of modified data. This feature looks very interesting in our study because step 3 needs to read a remotely modified buffer. MOESI avoids the aforementioned write-back to memory by allowing immediate sharing of these dirty cache-lines with other cores. The original modified lines switch to the new *Owned* state (that is responsible for doing the write-back to memory eventually) while the shared copies go to *Shared* state. Unlike MESI where both sender and receiver copies are in the same Shared state after step 3, MOESI therefore introduces an asymmetry.

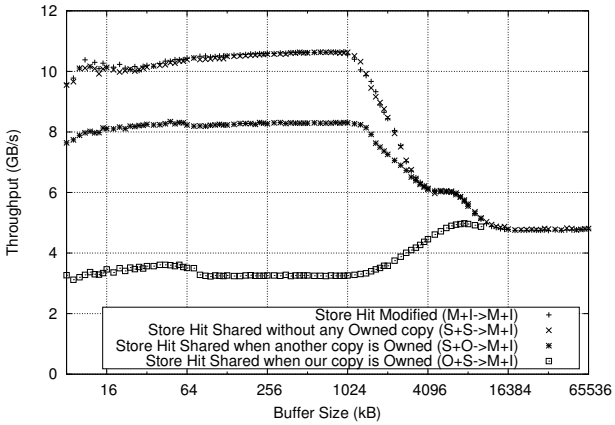


Figure 13. Store Hit performance depending on Shared, Owned and Modified state, inside a shared L3 cache, on AMD platform.

When a new transfer occurs through this shared-buffer, one of these asymmetric copies switches to Modified again during step 2 while the other gets invalidated. Given that the Modified state is similar to Owned (and not to Shared), one would expect that transitioning from Owned to Modified would be at least as quick as transitioning from Shared to Modified. Surprisingly Figure 13 shows the contrary: It is much faster (3x inside a socket, and 4x between sockets) to write to the Shared copy rather than the Owned one. We assume that a write-back always occurs when a cache-line leaves the Owned state and raises a non-documented phenomenon in this MOESI implementation.

This unexpected behavior leads to another unexpected result on Figure 14: On our AMD platform, data transfers are faster when shared-buffers are used in alternating direction (5 to 50% faster). This behavior seems very specific to AMD current micro-architecture *Bulldozer*. Intel nodes and some older AMD hosts (*Barcelona* micro-architecture) do not show such an asymmetric performance depending on

buffer reuse direction[7].

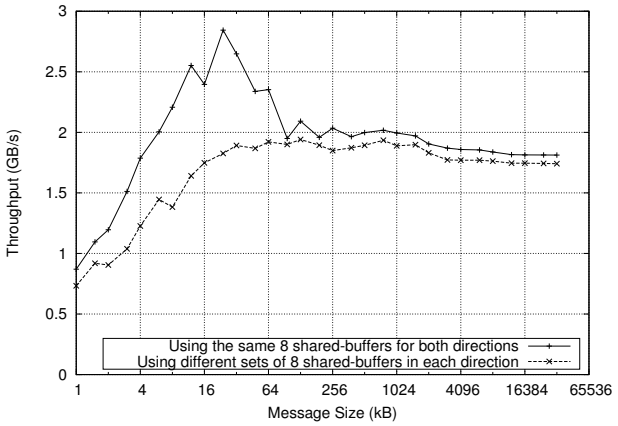


Figure 14. Performance of shared-memory data transfer depending on buffer reuse direction, inside a shared L3 cache, on AMD platform.

This result confirms the interest of our idea of hiding hardware complexity inside micro-benchmark outputs: Extracting the performance behavior from this black-box is much easier on modern platform that trying to formally understand and model the hardware.

V. Related Work

It is well known that tuning MPI implementations is difficult. Many configuration options are available and some of them even target conflicting use cases with respect to point-to-point operations vs collectives, blocking vs non-blocking, caching for intra-node communication, etc. When predicting a good configuration is not feasible, auto-tuning may be used to adapt the software to specific application needs. The OPTO framework [18] tests all possible configuration combinations so as to automatically find the best one. Machine learning was also proposed as an alternative method [19]. A training tool finds out important characteristics of the platform before matching them with specific application needs.

Our approach is rather a qualitative approach that tries to understand cache-related issues instead of blindly finding the best tuning for specific applications. One common way to evaluate intra-node communication performance is to look at cache misses [13]. However we explained in Section III-B that this is hardly a reliable piece of information. This paper gives some basic optimization hints to application developers while we rather focus on MPI implementers, those that are experts in the problems considered here.

The only work that is really close to our research mostly focuses on Xeon Phi accelerator cards [20]. However only synchronization issues (concurrent polling on shared receive queues) and small messages (up to 8 kB) are modeled. Our feeling is that modern memory architectures have a performance that is far too complex for such analytical models because of heavy and hardly-understandable behaviors when switching from L1 to L2, L3 or even main memory, or when looking at parallel accesses. This is why we hide this complexity inside micro-benchmark outputs.

[7]Intel nodes actually show a small performance difference as well, possibly because the MESIF protocol also breaks the symmetry between Shared copies (the *Forward* copy is the only one that replies to bus requests).

VI. Conclusion and Future Work

As intra-node communication becomes increasingly important for scientific application performance, the need to better understand and tune these data transfers raises. We believe that modern memory subsystems are too complex to be modeled accurately because of many hardware characteristics such as prefetchers and coherence protocols. We presented a new approach that models shared-memory communication performance by combining the individual throughput of micro-benchmarks based on the knowledge of buffer states in caches and of hardware cache coherence protocols. It allows us to predict the behavior of these transfers with a satisfyingly accuracy. It also shows that memory access parallelism and contention in shared caches and memory buses have complex behavior that explain why shared-memory communication performance is hard to tune and to model analytically.

The model and the predicted behavior was then used as the input for discussing optimization hints for MPI implementations. We identified two individual memory accesses within the overall data transfer that could be modified by adding some preflushing of cache-lines or non-temporal stores. Their limiting factors were described in current hardware cache coherence protocols. We then showed that the MOESI cache-coherence protocol of AMD platforms exhibits unexpected constraints on buffer reuse. It led us to improving communication performance by forcing shared-buffer reusing in alternating direction.

One drawback of our work is that we have not yet implemented all these ideas in MPI implementations. Some of then were only tested in synthetic benchmarks. The reason is that they require very intrusive changes in the way shared-buffers are managed internally in both Open MPI and MPICH2. We will then experiment these ideas on real applications. One may argue that cache-related optimizations may exhibit different behaviors on benchmarks and real applications. Fortunately our optimization hints target the accesses to the shared buffers without increasing the overall cache pollution, and they do not modify the way application buffers are actually involved. Additionally it should be noted that our memory model already works for several real-world application kernels [9].

We are looking at other architectures such as ARM processors that use MESI-based cache-coherence protocols. Our model should also be refined to better predict the

absolute performance by better understanding memory access parallelism and contention as well as better modeling capacity misses. Then we are looking at more complex communication patterns such as collective operations to see if our parallel benchmarks can explain the behavior of these operations at scale on large nodes. Finally we are trying to model other intra-node communication schemes such as direct-copy between processes through the kernel or copy offload to better understand when each strategy should be enabled.

VII. Acknowledgments

This work is supported by the STIC-AmSud SEHLOC project, and by Inria as part of the internship of Benoit Ruelle.

References

[1] D. Buntinas, G. Mercier, and W. Gropp, "Data Transfers between Processes in an SMP System: Performance Study and Application to MPI," *Parallel Processing, 2006. ICPP 2006. International Conference on*, pp. 487–496, Aug. 2006.

[2] R. Brightwell, T. Hudson, and K. Pedretti, "SMARTMAP: Operating System Support for Efficient Data Sharing Among Processes on a Multi-Core Processor," in *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2008*. Austin, TX: ACM Press, Nov. 2008.

[3] L. Chai, P. Lai, H.-W. Jin, and D. K. Panda, "Designing An Efficient Kernel-level and User-level Hybrid Approach for MPI Intra-node Communication on Multi-core Systems," in *Proceedings of the IEEE International Conference on Parallel Processing (ICPP-2008)*. Portland, Oregon: IEEE Computer Society Press, Sep. 2008.

[4] B. Goglin and S. Moreaud, "KNEM: a Generic and Scalable Kernel-Assisted Intra-node MPI Communication Framework," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 73, no. 2, pp. 176–188, Feb. 2013.

[5] C. Yeoh, "Cross Memory Attach," 2010, http://lwn.net/Articles/405284/.

[6] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud, "Cache-Efficient, Intranode Large-Message MPI Communication with MPICH2-Nemesis," in *Proceedings of the 38th International Conference on Parallel Processing (ICPP-2009)*. Vienna, Austria: IEEE Computer Society Press, Sep. 2009, pp. 462–469.

[7] T. Ma, G. Bosilca, A. Bouteiller, and J. J. Dongarra, "Locality and Topology aware Intra-node Communication Among Multicore CPUs," in *Proceedings of the 17th European MPI Users Group Conference*, ser. Lecture Notes in Computer Science. Stuttgart, Germany: Springer, Sep. 2010.

[8] S. Moreaud, B. Goglin, D. Goodell, and R. Namyst, "Optimizing MPI Communication within large Multicore nodes with Kernel assistance," in *CAC 2010: The 10th Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2010*. Atlanta, GA: IEEE Computer Society Press, Apr. 2010.

[9] B. Putigny, B. Goglin, and D. Barthou, "A Benchmark-based Performance Model for Memory-bound HPC Applications," in *Submitted to the 2014 International Conference on High Performance Computing Simulation (HPCS 2014)*, Bologna, Italy, Jul. 2014.

[10] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," *SIGARCH Comput. Archit. News*, vol. 12, no. 3, pp. 348–354, Jan. 1984.

[11] D. Buntinas, G. Mercier, and W. Gropp, "Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication Subsystem," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface: Proc. 13th European PVM/MPI Users Group Meeting*, Bonn, Germany, Sep. 2006.

[12] 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, Sep. 2004, pp. 97–104.

[13] S. Pellegrini, T. Hoefler, and T. Fahringer, "On the Effects of CPU Caches on MPI Point-to-Point Communications," in *Proceedings of the 2012 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Sep. 2012, pp. 495–503.

[14] A. Pesterev, N. Zeldovich, and R. T. Morris, "Locating Cache Performance Bottlenecks Using Data Profiling," in *Proceedings of the European conf. on Computer systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 335–348.

[15] "OSU Micro-Benchmarks," http://mvapich.cse.ohio-state.edu/benchmarks/.

[16] "Intel MPI Benchmarks," http://software.intel.com/en-us/articles/intel-mpi-benchmarks/.

[17] A. Denis, "A high performance superpipeline protocol for infiniband," in *Proceedings of the 17th International Euro-Par Conference*, ser. Lecture Notes in Computer Science, no. 6853. Bordeaux, France: Springer, Aug. 2011, pp. 276–287.

[18] M. Chaarawi, J. M. Squyres, E. Gabriel, and S. Feki, "A tool for optimizing runtime parameters of open mpi," in *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 210–217.

[19] S. Pellegrini, J. Wang, T. Fahringer, and H. Moritsch, "Optimizing MPI Runtime Parameter Settings by Using Machine Learning," in *EuroPVM/MPI*, ser. Lecture Notes in Computer Science, vol. 5759. Espoo, Finland: Springer, Sep. 2009, pp. 196–206.

[20] S. R. Garea and T. Hoefler, "Modeling Communication in Cache-Coherent SMP Systems - A Case-Study with Xeon Phi," in *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. ACM, 06 2013, pp. 97–108.