On the Effects of CPU Caches on MPI Point-to-Point Communications

Simone Pellegrini
University of Innsbruck
Innsbruck, Austria
spellegrini@dps.uibk.ac.at

Torsten Hoefler
University of Illinois
Urbana-Champaign, IL, USA
htor@illinois.edu

Thomas Fahringer
University of Innsbruck
Innsbruck, Austria
tf@dps.uibk.ac.at

Abstract—**Several researchers investigated the placing of communication calls in message-passing parallel codes. The current rule of thumb it to maximize communication/-computation overlap with early binding. In this work, we demonstrate that this is not the only design constraint because CPU caches can have a significant impact on communications. We conduct an empirical study of the interaction between CPU caching and communications for several different communication scenarios. We use the gained insight to formulate a set of intuitive rules for communication call placement and show how our rules can be applied to practical codes. Our optimized codes show an improvement of up to 40% for a simple stencil code. Our work is a first step towards communication optimizations by moving communication calls. We expect that future communication-aware compilers will use our insights as a standard technique to move communication calls in order to optimize performance.**

Keywords-**MPI; CPU Cache; Code Transformations**

I. Introduction

Programming distributed memory architectures to maximize performance is difficult. One of the main reason is the low abstraction level of many primitives provided by the message passing paradigm. Basic send/receive routines are indeed found in almost every distributed library or language (e.g. MPI [1]). When writing a parallel program, the programmer needs to take care not only of the semantic correctness of it but also about the positioning of the communication routines within the code. This problem has been investigated in the past and the common practice, in MPI, is to schedule nonblocking communication operations as early as possible in order to maximize communication/computation overlap [2].

However there are many architectural aspects which may play a role in determining a good program point to place send or receive calls. For example, the request management overhead of asynchronous routines – which are commonly used to hide communication costs – may penalize performance for small message sizes. Runtime systems for distributed memory libraries (e.g. MPI [1] and UPC [3]) usually employ optimizations to hide many architectural details to the user code. For example long messages may be split into smaller chunks to enable pipelining [4]. Oppositely, when too many short messages are sent, the runtime system may try to coalesce information into larger messages reducing the injection rate [5], [6]. Optimizations done at runtime are highly effective since the system is fully aware of the underlying architecture. However, most of the decision have already been made by the programmer in the source code and therefore, at this stage, is often too late to overcome performance bugs. For these reasons, production codes are usually hand-tuned for particular target architectures.

In this paper we study the impact of CPU cache on MPI communication routines. Indeed, in order to hide network latency, MPI libraries aggressively use *buffering*. For example, when small messages are sent, the MPI library does not wait for the receiver process to be ready-to-receive, instead MPI buffers the message data on the receiver side (often called *eager send*). MPI not only uses the main memory for buffering reasons but also for allowing efficient communication of MPI processes allocated on the same multi-core machine. Intra-node communication is performed by means of shared memory (SM) transfer layers which are provided by all major MPI implementations [7], [8]. Because buffering is implemented using the main memory, it is subject to cache hierarchies, and thus the reason for our study.

We measure, with a synthetic benchmark, the differences in terms of execution time, for point-to-point operations performed when the data being sent is fully loaded into the CPU cache or not. We repeat the experiment with multiple configurations, i.e., intra-node and inter-node. In the same way we measure the impact of those point-to-point communication routines on the application cache by accessing application data, previously loaded into the cache, right after the communication is performed. From the gathered data we derive a set of rules and guidelines which can be

This research has been partially funded by the Austrian Research Promotion Agency under contract nr. 824925 (OpenCore) and under contract nr. 834307 (AutoCore).

utilized to transform the input program for improved cache utilizations and thus performance. To the best of our knowledge, this aspect has been largely neglected until now. Work in literature focuses on quantifying the impact of local memory on communications [9]. Those works are principally concerned with *non-regular* data types which involve expensive packing/unpacking operations and optimizing the way the MPI library handles them; whereas our work focuses on contiguous data and how the impact of communication routines can be exploited, by a programmer or a compiler, to optimize the input code.

Experiments show that a send, or receive, operation can be up to 25% faster if the data is already in the CPU cache. Furthermore, cache pollution generated by communication routines can negatively impact on the application performance if not carefully placed. Indeed, send and receive operations can invalidate the content of the message buffers if they have been preloaded into the cache. Based on the guidelines derived by our benchmark data, we propose a communication/cache-aware code transformation which, when manually applied to a 3-point stencil code, it improves code performance up to 40% for specific message sizes. Our transformation always shows a positive effect on performance for messages which are smaller than the last level cache size.

The contributions of the paper are multiple:

- It presents a benchmark to measure the performance of send/receive operations with different configuration of the data cache;
- It derives, from gathered data, a set of optimization guidelines which can be used to tune an input program for improved cache behaviour;
- It demonstrates the efficacy of the derived optimization strategies by applying them to a 3-point stencil code.

II. Analyzing MPI Cache Behaviour

In order to highlight the effects of CPU caches on MPI communication routines we wrote a synthetic benchmark. The main goal of the MPI cache benchmark is to capture differences in terms of execution time between communication routines with multiple configurations of the CPU cache and additionally, to measure their impact on the application cache. In doing so, we also collect the value of several performance counters using the PAPI library [10]. Many benchmarking suites for MPI exist in literature[11], [12]. Coddington et al. wrote a survey of benchmarking tools for MPI's point-to-point communication [13]. However none of those is designed to capture cache behaviour of MPI routines. Some of the tools, e.g. MPIBench[12] and SKaMPI[11], provide options to pre-load messages into the cache before performing the communication but they do not provide a way to precisely capture the level cache pollution caused by MPI communication routines. The benchmark code which has been developed for this purpose follows the guidelines for reproducibility of measurements described in [14]; the code is publicly available at [15]. Beside the execution time the benchmark takes care of registering the values of multiple PAPI performance counters which will be used to understand low level implementation details of the underlying MPI library.

The benchmark is split into two scenarios, `SCN1` and `SCN2`, which are further described in this section. Because of space limitations in [15].

A. Scenario 1 – `SCN1`

`SCN1` studies the behaviour of single MPI send/receive routines. With this benchmark we are interested in capturing the behaviour, in term of performance, of two basic MPI routines, i.e. `MPI_Send` and `MPI_Recv`, considering different states of the data cache. We therefore perform a ping-pong operation with three different initial cache states. In the first case, `INV`, we make sure all the content in the cache is wiped out and none of the data elements being sent or received are present into any of the CPU caches. The second cache configuration, `EXCL`, entirely pre-loads into the cache the message data right before the communication is performed. Data elements are only read which means the corresponding cache lines are in the "exclusive" state. In the last cache configuration, `MOD`, cache lines are preloaded in the "modified" state.

B. Scenario 2 – `SCN2`

In the second scenario, referred as `SCN2`, we want to capture the level of cache pollution caused by send/receive communication routines. This is obtained by measuring the time, together with other performance counters, required to traverse the array containing the message data previously exchanged in the ping-pong operation. This is again done considering multiple configurations of the cache. In `INV`, we start by cleaning the caches, we then perform the message exchange and, upon competition of the send/receive, data is traversed and the measurement is performed. In the second configuration, `PRE`, we pre-load the message data into the cache before performing the message exchange. It is worth noting that, in both cases, the code for which we perform the measurements does not contain any communication statements. Obtained data is compared with the values measured while traversing the message buffer without previously performing any communication. Also in this case we consider two cache configurations, i.e., cache is invalidated before

System name	LEO3	VSC2
Max # of nodes	162	1.314
Sockets per node	2	2
Cores per Socket	6	8
Core Architecture	Intel Xeon X5650	AMD Opteron 6132 HE
Clock Frequency	2.67GHz	2.2GHz
L1 cache	32KB + 32KB	64KB + 64KB
L2 cache	256KB (private)	512KB (private)
L3 cache	12MB (shared by 6)	2x6MB (shared by 4)
Symmetric Multi-Threading (SMT)	Disabled	NA
Memory per Node	24GB DDR3	32GB DDR3
Interconnection	Infiniband 4x QDR	Infiniband 4x QDR
Kernel Version	2.6.32	2.6.32
Open MPI version	1.5.5	1.5.4
SM module	OpenMPI default	KNEM[16]

Table I: Experimental target architectures.

the array elements are accessed, `BASE_INV`, or the array is fully pre-loaded into the cache before being traversed, `BASE_PRE`.

We repeat the experiment with two different process allocations in order to test intra-node and inter-node point-to-point communications. This is obtained by allocating the two MPI processes respectively on different computing nodes or on the same multi-processor machine. In both cases, the use of affinity settings ensures the MPI processes are bound to a specific core of distinct CPUs. This is done in order to take full advantage of the CPU cache and avoid conflicts which arise when multiple processes share the same last level cache.

C. Hardware Platforms

We evaluated the code on 2 computing platforms summarized in Table I. The LEO3 cluster system consists of 162 compute nodes (with a total of 1944 cores). All nodes are connected through an Infiniband 4x QDR high speed interconnect. Each node contains two Intel Xeon CPU based on the Nehalem architecture where Hyper Threading (HT), or 2-fold SMT, has been disabled from the BIOS. The Vienna Supercomputing Cluster 2 (VSC2) is a HPC system which consists of 1.314 nodes, with 2 AMD Opteron processors each, for a total of 21.024 CPU cores. CPU cache layout for the two system is also summarized in Table I. These are both production clusters and the measurements have been taken while the clusters were fully operational, therefore we expect some noise to show up in the measurements. In order to reduce it we repeat each measurement 100 times and take the median.

D. MPI Communication Protocols

The cache benchmark treats the underlying MPI library as a black box. This allows us to make considerations which are not biased towards a particular feature of an MPI implementation. However, MPI libraries are very complex and in order to be able to correctly interpret the gathered data, implementation details cannot be completely neglected. Indeed every MPI library exposes several "knobs" which can be used to tune the performance of a particular application on the underlying target platform [17], [18]. One of the most relevant threshold for point-to-point communication is the so called "eager limit". The eager protocol is not standardized by the MPI specification, however it is an implementation technique utilized by all MPI implementations. Every message exchanged between peer processes is subject to this protocol. MPI libraries typically use (at least) two algorithms, *eager* and *rendezvous*. When the size of the transmitted message is smaller than the specified threshold value, the message (together with an MPI header) is eagerly sent to the receiver. For larger messages the *rendezvous* protocol is utilized instead, i.e. the sender process sends a ready-to-send message (RTS) to the receiver and blocks waiting for the acknowledgment, the clear-to-send (CTS), from the matching receive. The eager protocol is useful when latency is important because it avoids CTS/RTS round-trip overhead. However it requires additional buffering at the receiver side. Rendezvous protocols are typically used when resource consumption is critical.

For example, the Open MPI library [7] uses multiple protocols, detailed in [19]. In the case of eager send, the behaviour is the same as described before. The rendezvous protocol however enables better latency hiding. When the communication is performed over RDMA-enabled networks (such as an OpenFabrics-based network, e.g. InfiniBand) the protocol is divided into three phases. In the first phase the RTS message is sent to the receiver, while the sender is waiting for the CTS message, it starts "registering" the rest of the large message with the OpenFabrics network stack. Since the registration is slow the process is pipelined so that registration latency is hidden.

In shared-memory, the rendezvous protocol can use several implementation mechanism which have been presented over the last decade because of the increasing relevance of multi-core systems. Most shared-memory message passing implementations, such as Nemesis [20] device in MPICH2 and the SM component in Open MPI, depend on a double buffering memory scheme. An extra memory buffer is pre-allocated as an exchange zone between processes. Communication between the processes is performed using the so called *copyin/copyout* semantics (CICO). The sender process copies from the message buffer into the shared memory and in the same way the receiver reads it out and copies into the receiver buffer. In order to reduce latency, the copy happens in a pipelined way.

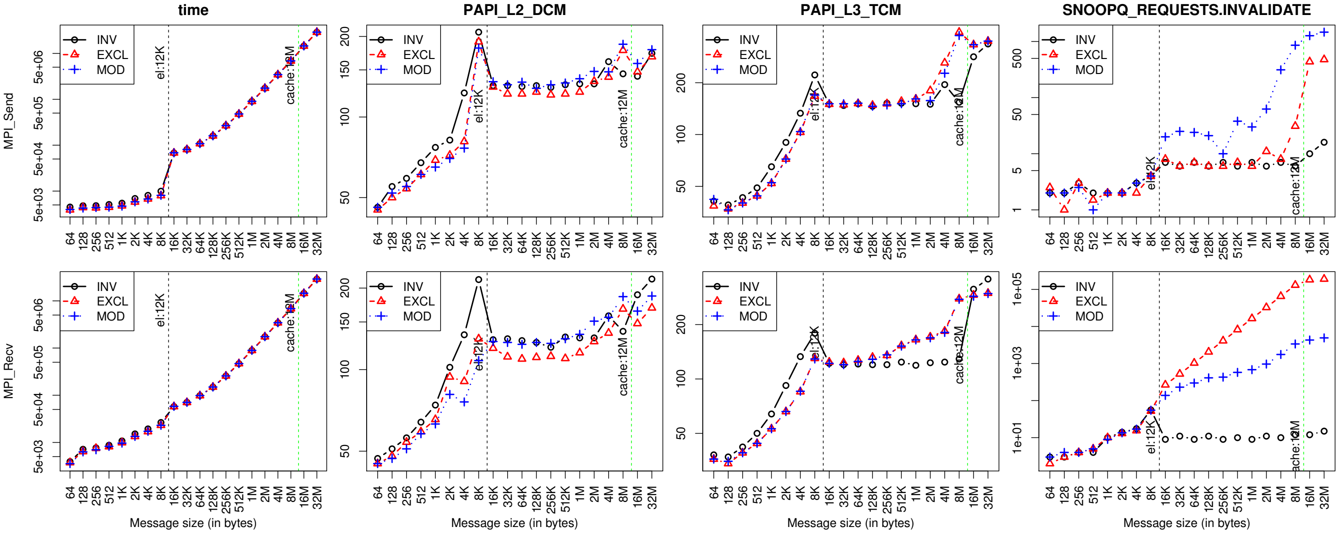


Figure 1: LEO3 Inter-node – SCN1 – Send/Receive

However approaches exist, such as KNEM [16], which via a kernel extension, allows the direct copy from the sender to the receiver buffer. This mechanism has the advantage to eliminate the additional memory copy and therefore both reduces latency and cache pollution.

We perform our measurements using the default settings provided by the chosen MPI library. We use Open MPI with the default eager limit, which is set by default to 12 KiB for communication over Infiniband and to 4 KiB for intra-node communication, on both systems. In the LEO3 cluster we used the default shared memory provided by the Open MPI library which is based on the CICO mechanism. On the VSC2 cluster shared memory communications are performed using the KNEM kernel extension.

III. Benchmark results

In this section, the data gathered by running our cache benchmark for cluster architectures, listed in Table I, is shown. For space limitations, we show values of the PAPI performance counters only for the LEO3 architecture.

A. Inter-node communication – Infiniband

Figures 1 and 2 depict the values obtained by the two benchmark scenarios (SCN1 and SCN2) using inter-node communication, over Infiniband.

1) SCN1: Figure 1 shows several performance counters associated with the MPI_Send operation, in the first line, and MPI_Recv, in the second line, using the three cache configurations: INV, EXCL and MOD. The first column shows the execution time which, in order to be as precise as possible, is expressed in terms of number of CPU clock cycles. Differences in terms of the execution are barely noticeable. However, we can see that having data preloaded into the cache (as in EXCL and MOD) reduces the amount of L2 data cache misses (PAPI_L2_DCM counter in the second column) up to the eager limit, this is visible especially at the receiver side where buffering happens. Indeed, the two routines have a reduced execution time, which reaches its peak of around 20% for messages of 8KiB, when the message data is preloaded into the cache.

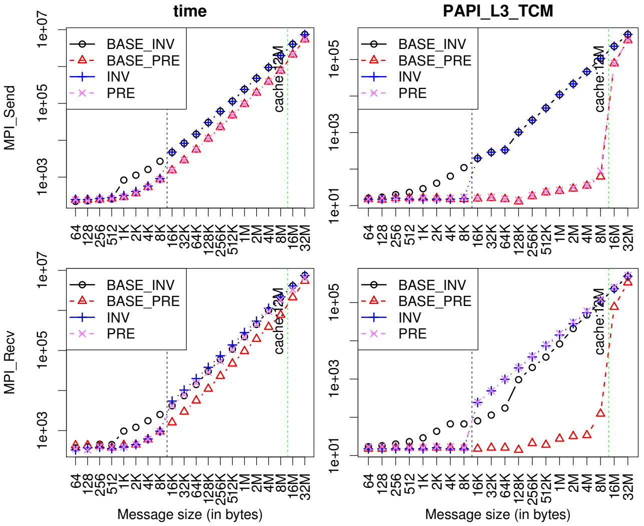


Figure 2: LEO3 Inter-node – SCN2 – Cache Pollution

After the eager threshold is exceeded we still have better behaviour of L2 cache however we notice an increase of L3 cache misses (PAPI_L3_TCM hardware counter) which is similar for the EXCL and MOD cache states. While the reduced cache misses in L2 cache are constant for increasing message sizes, L3 cache misses proportionally grows with the message size.

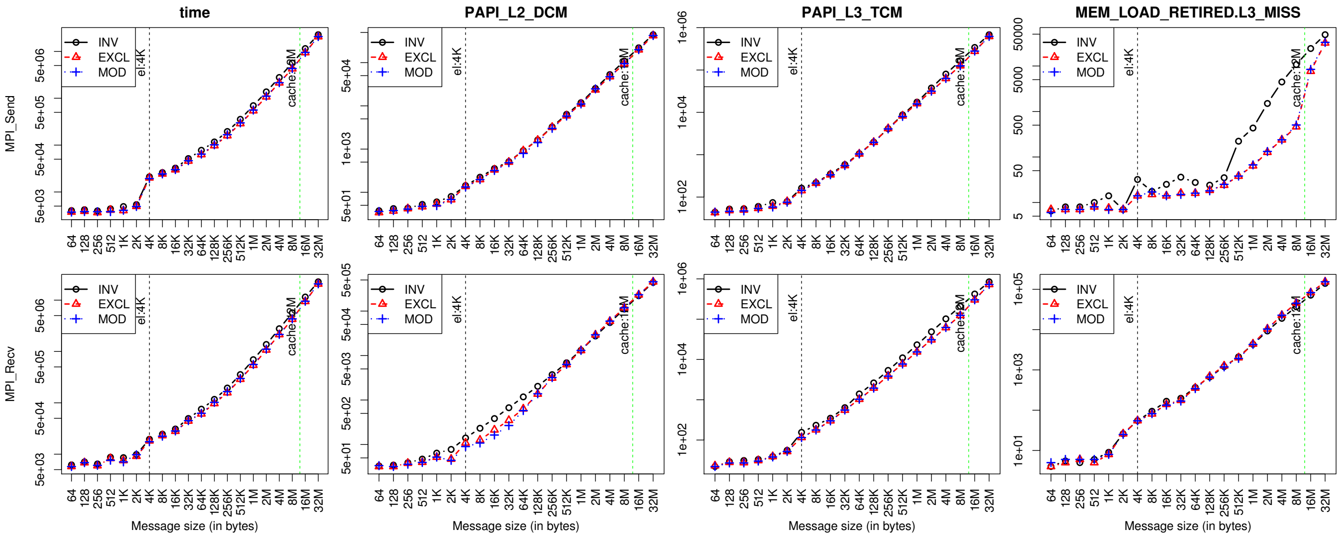


Figure 3: LEO3 Intra-node – SCN1 – Send/Receive

To better understand the reason for this we show another performance counter, in the last column of Figure 1, which depicts the number of *snoop* invalidation requests addressing the CPU. It can be noticed that during the rendezvous protocol, the number of invalidation requests increases considerably if the message data is preloaded into the cache. This is more marked for the receiver as the NIC driver updates the message buffer in main memory and therefore eventual dirty copies in the cache need to be invalidated.

2) *SCN2:* The measurements for the second scenario, SCN2, are depicted in Figure 2. As previously stated, this benchmark measures the performance resulting from accessing the message buffer right after being sent/received. We keep performance values for BASE_INV and BASE_PRE as a upper and lower bound for what we expect to be the performance from this scenario. Interesting is the number of L3 cache misses, in the case of the sender process, we notice that accessing the data after the send operation (INV) causes the same amount of misses measured for BASE_INV. This means the send operation does not pollute the application cache. However this is not true for messages which are smaller than the eager limit. In that case there are no L3 cache misses for both INV and PRE configurations.

Major differences between sender and receiver happen beyond the eager threshold. In PRE, while at the sender side the amount of cache misses is comparable with the one measured for the BASE_PRE configuration; the receiver behaviour is instead similar to the BASE_INV case. Indeed, the receive operation invalidates the entire L3 cache (as suggested by the memory bus snoop operations shown in Figure 1) and accessing the received elements costs as many memory operations as accessing it from a completely invalid cache (BASE_INV). Additionally, loading the data after the receive routine causes more misses than the BASE_INV configuration (which should be the performance upper-bound). Unfortunately we could not find a reasonable explanation for this. The increased amount of L3 cache misses has also a significant impact on the execution time which for INV and PRE is slightly higher than BASE_INV. In our opinion, the reason for this is consequence of the *memory pinning* operation performed by the MPI library. Also it is worth saying that the same kind of behaviour has been observed at the sender side when the data is preloaded in a "modified" state. In that case, the send operation invalidates all the preloaded cache lines and therefore accessing

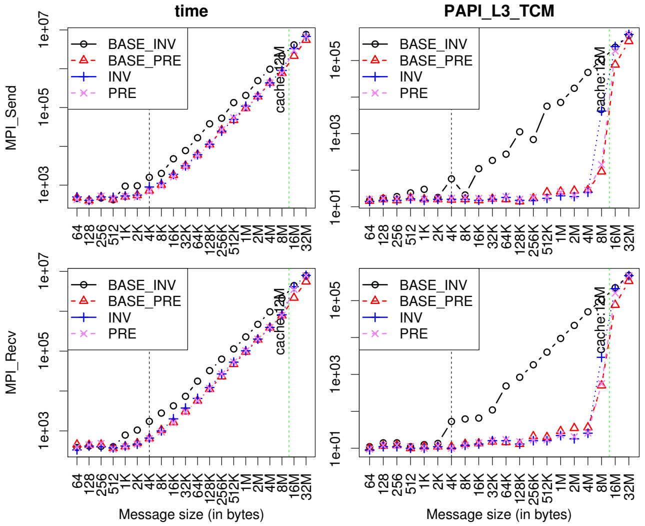


Figure 4: LEO3 Intra-node – SCN2 – Cache Pollution

the buffer data after the communication routine is slower.

B. Intra-node Communication – SM

In Figures 3 and 4, the data obtained for shared memory configuration for the LEO3 cluster is shown.

1) SCN1: Figure 3 depicts the measurements for SCN1. In this case we observe overall a much higher number of cache misses since the actual data exchange between the two MPI processes happens in shared memory. However, for the sender process, we see only small differences among the three configurations. We show the value of the MEM_LOAD_RETIRED:L3_MISS performance counter which proves the advantage, i.e. reduced number of memory load misses, due to fact of having the message buffer available in the cache. At the receiver side instead, we observe a smaller number of both L2 and L3 cache misses for messages up to the last level cache size. Overall, the performance of MPI routines is improved when data is preloaded into the cache and the gain reaches its peak, around 25%, before the cache size is exceeded. As already stated, in this machine shared memory communication is performed using a CICO mechanism. Because this transfer between sender and receiver is done using a shared buffer, which for the Open MPI library is of 32 KiB, only a portion of the data cache gets polluted during the transfer.

2) SCN2: This is visible in Figure 4. Differently from what observed for inter-node communications, in shared memory the message buffer is fully loaded into the cache for both INV and PRE configurations. However while the amount of L3 cache misses for PRE, BASE_PRE and INV is almost the same up to 4 MiB, at 8 MiB we start seeing a gap between the three configurations. The amount of cache pollution is higher at the sender side since the difference in terms of cache misses between PRE and BASE_PRE is noticeably higher than the receiver side. This is unexpected since the data transfer from the user buffer to the shared memory segment should be implemented using non temporal move instructions (e.g. MOVNTDQ), which avoids the target address to be loaded into the CPU cache. However, this penalty happens only for message sizes which are larger than half of the last level cache size.

IV. Considerations and Optimization Guidelines

From the output of the MPI cache benchmark we derive, in this section, a set of intuitive rules to find a good placement for send/receive communication statements which better exploit the properties of the CPU caches. We divide our consideration into three subsections applying to specific ranges of the transmitted data, i.e., (*i*) from 1 byte up to the eager limit, (*ii*) from the eager threshold up to the last level cache size and (*iii*) beyond the available cache size.

A. From 1 Byte to the Eager Threshold

When the eager protocol is utilized, messages are transfered to the NIC using a memcpy() operation which has the side-effect of loading the content of the send buffer into the CPU cache. Therefore if the transmitted data is accessed right after the send operation, the data will be still available in one of the CPU caches. Additionally the memcpy() routine also benefits from having the source and target buffers preloaded into the cache. However, the input program could present dependencies which does not allow this transformation to be applied. In such situation, the sent/received data should be accessed immediately after the communication routines or as late as before the message buffer content gets kicked out from the data caches.

RULE *1:* **For messages up to the eager limit, it is always preferable to perform the communication when the message data is cached. Received data should be immediately accessed.**

B. From the Eager Limit to the Last Level Cache Size

We now consider the second message range, from the eager limit up to the cache size. In this range intra-node and inter-node communication differ and we treat them separately.

1) Inter-node communication: As far the communication statement is concerned, we observe an increase in the number of L3 cache misses which is proportional to the message size, Figure 1. However the overall number of cache misses is small that the execution time is not affected by it. More interesting considerations can be done for Figure 2. At the sender side, we noticed no cache pollution caused by the send operation. Therefore we expect no changes in the application performance from changing send statements placement.

However things change dramatically at the receiver side. The receive operation invalidates all the preloaded cache lines in the case the message data was preloaded into the cache. Additionally, because of the memory pinning, utilized by the rendezvous protocol in Open MPI, loading the received data right after the communication statement has a negative impact on performance. A similar behaviour was also observed for the sender process when the data is preloaded in a "modified" state as discussed in Section III-B.

RULE *2:* **Avoid to access the transfered data immediately after the communication routines. If possible, perform all the computation on the message buffer before issuing a send operation. At the receiver side,**

```
for(unsigned iter=0; iter<MAX_ITERS; ++iter) {

  MPI_Sendrecv(&A[0][0], COLS, MPI_DOUBLE, bottom, 0,
         &A[ROWS-1][0], COLS, MPI_DOUBLE, top, 0,
         MPI_COMM_WORLD, MPI_STATUS_IGNORE);

  for(unsigned i = 0; i<ROWS-1; ++i)
    for(unsigned j = 0; j<COLS-1; ++j)
      tmp[i][j]=A[i][j]+(A[i+1][j]+A[i][j+1])/4;

  double** ttemp=A; A=tmp; tmp=ttemp; // swap arrays
}
```

Listing 1: 3-point stencil code

delay the access to the receiver buffer by overlapping other computation.

2) Intra-node communication: For shared memory communication we notice a reduction of L2 and L3 cache misses, Figure 3, which is proportional to the size of the message being transferred. This has positive effects on the execution time which reaches a maximum improvement, of around 25%, both for sender and receiver processes, for 8 MiB messages. For shared memory communications, both the send and the receive routines populate the cache with the content of the message buffer and in the case the data is preloaded before the communication routine, the cache lines will not be invalidated. However, when the CICO mechanism is utilized, cache pollution may occur for large messages.

RULE *3:* **Access the message data after the communication statements, if the data is not already loaded into the cache, when the message size is smaller than** *LAST_LEVEL_CACHE_SIZE*/2 **bytes. If the data is already into the cache, perform all the computation before invoking any communication routine.**

C. Beyond the Last Level Cache Size

Beyond the cache line the behaviour of our benchmarks tend to converge, therefore no meaningful optimization rule can be defined. However, large messages can be divided into smaller chucks using a well known MPI code transformation referred in literature as software pipelining or message strip mining [21]. If the splitting size is chosen accordingly, the cache effects can be enabled. However this aspect is orthogonal to the argumentation of this paper and we are set to explore it in future work.

V. A Case Study: 3-point Stencil

Following the optimization guidelines derived in the previous section we manually tuned a 3-point stencil code which encodes a pattern commonly utilized in many HPC codes. A common way of parallelizing such stencil operation in MPI is shown in Listing 1. The code has a communication statement at beginning of the loop which exchange the first and last row of a 2-dimensional matrix which is updated by the following stencil computation. It is worth noting that while the receive operation must be performed before the last iteration of the `i` loop, the send operation has no dependencies and can be therefore issued at any program point, but before the swap procedure. We derive two versions of the stencil code depicted respectively in Listings 2 and 3.

```
for(unsigned iter=0; iter<MAX_ITERS; ++iter) {

  for(unsigned j = 0; j<COLS-1; ++j)
      tmp[0][j]=A[0][j]+(A[1][j]+A[0][j+1])/4;

  MPI_Sendrecv(&A[0][0], COLS, MPI_DOUBLE, top, 0,
         &A[ROWS-1][0], COLS, MPI_DOUBLE, bottom, 0,
         MPI_COMM_WORLD, MPI_STATUS_IGNORE);

  for(unsigned i = 1; i<ROWS-1; ++i)
    for(unsigned j = 0; j<COLS-1; ++j)
      tmp[i][j]=A[i][j]+(A[i+1][j]+A[i][j+1])/4;

  double** ttemp=A; A=tmp; tmp=ttemp; // swap arrays
}
```

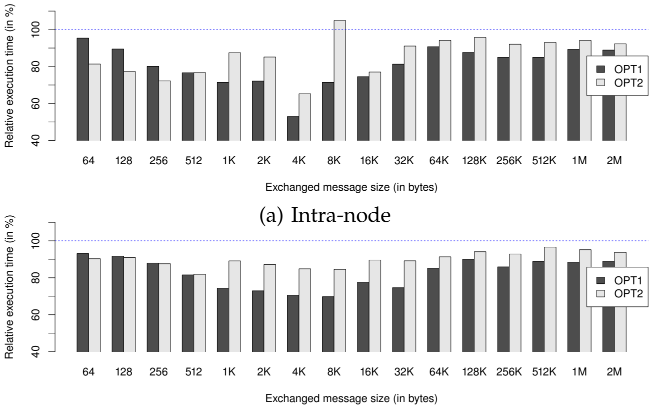
Listing 2: Tuned 3-points stencil code (*OPT*1)

Based on the our observations, the code has a bad cache behaviour as the array elements being sent, which are in a "modified" state, are accessed right after the communication statement, therefore after being kicked out from the CPU cache (when the rendezvous protocol is utilized). In order to optimize this aspect we can rewrite the code by moving the communication right after the first iteration of the loop is performed. This has two advantages: (*i*) it makes sure the matrix rows which are going to be sent/received are freshly loaded into the cache; (*ii*) avoid to access the received data right after the communication routine. The transformed code is depicted in Listing 2, we refer to this code version as `OPT1`.

The `OPT1` code version can however be further improved for the receive operation. As a matter of fact, the received data is not immediately consumed but accessed only in the last iteration of the stencil loop. This could not be optimal for messages which are smaller than the eager limit. For optimizing this aspect we can derive a second code version which utilizes the received data immediately after the data is available in the receiver buffer. This is obtained by reversing the order of execution of the stencil code. We traverse the 2-dimensional matrix from `ROW-3` backwards until the first row. In this way we make sure to have the send data already into the cache. We then perform the communication and successively complete the stencil by updating the last row. We refer to this code version as `OPT2`, the code is depicted in Listing 3. It is worth noting that in this version, the receiver buffer may not be into the cache before the message exchange if the entire problem does not fit into the cache.

```
for(unsigned iter=0; iter<MAX_ITERS; ++iter) {

  for(long i = ROWS-3; i>=0; --i)
      for(long j = COLS-2; j>=0; --j)
          tmp[i][j] = A[i][j] + 1/4*(A[i+1][j]+A[i][j+1]);

  MPI_Sendrecv(&A[0][0], COLS, MPI_DOUBLE, top, 0,
         &A[ROWS-1][0], COLS, MPI_DOUBLE, bottom, 0,
         MPI_COMM_WORLD, MPI_STATUS_IGNORE);

  for(unsigned j = 0; j<COLS-1; ++j)
      tmp[ROWS-2][j]=A[ROWS-2][j]+
                    (A[ROWS-1][j]+A[ROWS-2][j+1])/4;

  double** ttemp=A; A=tmp; tmp=ttemp; // swap arrays
}
```

Listing 3: Tuned 3-points stencil code (*OPT*2)



(a) Intra-node

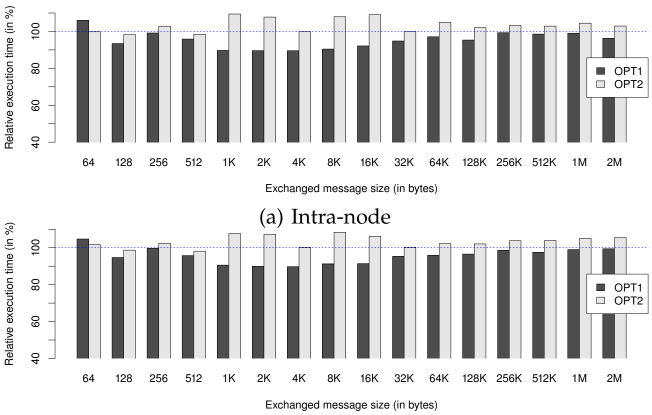
(b) Inter-node

Figure 5: LEO3 – Evaluation of tuned 3-point stencil application code

A. Evaluation

We evaluated the three versions of the stencil code on the two clusters described in Table I. Each version has been executed multiple times with different problem sizes using two different process allocations, i.e. intra-node and inter-node. We ran the stencil code using two MPI processes to correlate the outcome with the results gathered by the cache benchmark. We measured the execution time of each code versions and used the value of the median obtained from 100 repetitions of the program.

Figure 5 shows the execution time of code versions OPT1 and OPT2 relative to the baseline solution, i.e. 1 for the LEO3 cluster. The x axis refers to the size of the message (in bytes) being exchanged by the stencil computation in every iteration. As expected, the OPT2 version has better performance for small message sizes reaching, for shared memory, a performance improvement of around 20% for 256 bytes messages. For larger messages OPT1 has a better performance reducing the execution time of the stencil code by 40%. For larger message, the advantage becomes smaller as the communication/computation ratio diminishes.



(a) Intra-node

(b) Inter-node

Figure 6: VSC2 – Evaluation of tuned 3-point stencil application code

Figure 6 shows the results for the VSC2 cluster for both intra- and inter-node communications. Also on this machine, OPT2 has an advantage over the original stencil code for very small message sizes. However, for larger messages this version is noticeable slower. The OPT1 version, on the contrary, is faster for both inter- and intra-node communication. However, the measured performance improvement is contained, i.e. around 10%. We believe the bad performance of the OPT2 version is due to the reversed access of array elements which may inhibit the CPU prefetcher from correctly determine the data access pattern.

Overall, the tuned code is faster on both architectures. We demonstrate how our simple guidelines, derived from the LEO3 cluster, are portable to different architectures. However the experiment also shows how sensible the performance might be because of peculiarities of the underlying hardware.

VI. Conclusions and Future Work

In this paper we studied, using a synthetic benchmark, the impact of CPU caches on MPI communication statements and conversely the effects of MPI routines on the application cache. We described interesting findings regarding implementation details of the MPI library and we derived three simple optimization guidelines which can be used to tune MPI programs for improved cache utilization.

We followed and apply our optimization rules to a simple stencil code showing a performance improvement of up to 40%. To some extent, we demonstrated that performance is portable among different architectures. However, experiments showed that the details of the underlying CPU architecture may prefer different styles of optimizations. Therefore we expect, in the future, that by combining the semantics of MPI communication routines and the knowledge of the underlying architecture, the code transformations here proposed can be automatically applied by an MPI-aware compiler.

Acknowledgments

This work was supported by the Austrian Ministry of Science BMWF as part of the UniInfrastrukturprogramm of the Research Platform Scientific Computing at the University of Innsbruck. Furthermore, the computational results presented have been achieved in part using the Vienna Scientific Cluster (VSC).

References

[1] MPI Forum, "Message Passing Interface (MPI) Forum Home Page," http://www.mpi-forum.org/ (Dec. 2009).

[2] T. Fahringer and E. Mehofer, "Buffer-safe communication optimization based on data flow analysis and performance prediction," in *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques (PACT'97)*. IEEE Computer Society Press, 1997, pp. 189–200.

[3] UPC Consortium, "UPC Language Specifications, v1.2," Lawrence Berkeley National Lab, Tech Report LBNL-59208, 2005.

[4] G. M. Shipman, T. S. Woodall, G. B. andRich L. Graham, and A. B. Maccabe, "High performance RDMA protocols in HPC," in *Proceedings, 13th European PVM/MPI Users' Group Meeting*, ser. Lecture Notes in Computer Science. Bonn, Germany: Springer-Verlag, September 2006.

[5] M. J. Koop, T. Jones, and D. K. Panda, "Reducing connection memory requirements of mpi for infiniband clusters: A message coalescing approach," in *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, ser. CCGRID '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 495–504.

[6] W.-Y. Chen, C. Iancu, and K. Yelick, "Communication optimizations for fine-grained upc applications," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 267–278.

[7] "Open MPI," http://www.open-mpi.org.

[8] "MPICH2," http://www.mcs.anl.gov/mpi/mpich2.

[9] K. W. Cameron and X.-H. Sun, "Quantifying Locality Effect in Data Access Delay: Memory logP," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '03. Washington, DC, USA: IEEE Computer Society, 2003.

[10] K. London, S. Moore, P. Mucci, K. Seymour, and R. Luczak, "The PAPI Cross-Platform Interface to Hardware Performance Counters," in *Department of Defense Users Group Conference Proceedings*, 2001, pp. 18–21.

[11] R. Reussner, P. Sanders, and J. L. Träff, "SKaMPI: a comprehensive benchmark for public benchmarking of MPI," *Sci. Program.*, vol. 10, no. 1, pp. 55–65, Jan. 2002.

[12] D. Grove and P. Coddington, "Precise MPI Performance Measurement Using MPIBench," in *In Proceedings of HPC Asia*, 2001.

[13] P. Coddington, "Comparison of MPI Benchmark Programs on Shared Memory and Distributed Memory Machines (Point-to-Point Communication)," *International Journal of High Performance Computing Applications*, vol. 24, no. 4, pp. 469–483, 2010.

[14] W. Gropp and E. L. Lusk, "Reproducible measurements of mpi performance characteristics," in *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. London, UK, UK: Springer-Verlag, 1999, pp. 11–18.

[15] S. Pellegrini, "The MPI Cache Benchmark," https://github.com/motonacciu/mpi-cache-bench (May. 2012).

[16] H.-W. Jin, S. Sur, L. Chai, and D. K. Panda, "Lightweight kernel-level primitives for high-performance mpi intranode communication over multi-core systems," in *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, ser. CLUSTER '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 446–451.

[17] S. Pellegrini, T. Fahringer, H. Jordan, and H. Moritsch, "Automatic tuning of mpi runtime parameter settings by using machine learning," in *Proceedings of the 7th ACM international conference on Computing frontiers*, ser. CF '10. New York, NY, USA: ACM, 2010, pp. 115–116.

[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] T. Woodall, R. Graham, R. Castain, D. Daniel, M. Sukalski, G. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, and A. Lumsdaine, "TEG: A high-performance, scalable, multi-network point-to-point communications methodology," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 303–310.

[20] D. Buntinas and G. Mercier, "Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem," in *Proceedings of the International Symposium on Cluster Computing and the Grid*. IEEE Computer Society, 2006, pp. 521–530.

[21] A. Danalis, K.-Y. Kim, L. Pollock, and M. Swany, "Transformations to parallel codes for communication-computation overlap," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, ser. SC '05, Washington, DC, USA, 2005.