Interferences between Communications and Computations in Distributed HPC Systems

Alexandre Denis
alexandre.denis@inria.fr
Inria Bordeaux - Sud-Ouest
Bordeaux, France

Emmanuel Jeannot
emmanuel.jeannot@inria.fr
Inria Bordeaux - Sud-Ouest
Bordeaux, France

Philippe Swartvagher
philippe.swartvagher@inria.fr
Inria Bordeaux - Sud-Ouest
Bordeaux, France

ABSTRACT

Parallel runtime systems such as MPI or task-based libraries provide models to manage both computation and communication by allocating cores, scheduling threads, executing communication algorithms. Efficiently implementing such models is challenging due to their interplay within the runtime system. In this paper, we assess interferences between communications and computations when they run side by side. We study the impact of communications on computations, and conversely the impact of computations on communication performance. We consider two aspects: CPU frequency, and memory contention. We have designed benchmarks to measure these phenomena. We show that CPU frequency variations caused by computation have a small impact on communication latency and bandwidth. However, we have observed on Intel, AMD and ARM processors, that memory contention may cause a severe slowdown of computation and communication when they occur at the same time. We have designed a benchmark with a tunable arithmetic intensity that shows how interferences between communication and computation actually depend on memory pressure of the application. Finally we have observed up to 90 % performance loss on communications with common HPC kernels such as `CG` and `GEMM`.

CCS CONCEPTS

• **Networks** → *Network performance analysis.*

KEYWORDS

HPC, MPI, memory contention, processor frequency, runtime system

ACM Reference Format:
Alexandre Denis, Emmanuel Jeannot, and Philippe Swartvagher. 2021. Interferences between Communications and Computations in Distributed HPC Systems. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3472456.3473516

1 INTRODUCTION

Doing in parallel communications and computations (with non-blocking MPI calls or even more complex systems such as task-based runtime systems) is an increasing trend to get higher performances. As already mentioned in [10, 12], we have observed in some application executions that, sometimes, when computations and communications are executed side by side, communications are slower than nominal performances and computations can also be degraded.

Since possible interactions between communications and computations, and especially the impact on communication performances, are not well detailed in the literature (but only mentioned), we propose, in this paper to study the possible causes of these interferences and measure their impact on both communication and computing performances. We investigate the impact of processor frequencies, memory contention and the use of a task-based runtime system. Since we target HPC systems, we considered only fast networks (InfiniBand and OmniPath) as well as inter-node communications.

This paper presents the following study. We measured the impact of frequency scaling on communications. We also study the impact, in the case of CPU-bound applications and memory-bound ones, on communication bandwidth and latency. Moreover, we study the effect of data locality and thread mapping on the interference between computation and communication. Further, we introduce a benchmark with tunable arithmetic intensity to observe how the application memory pressure penalizes the performance of communications. We also study the communication performance degradation caused by the use of a task-based runtime system. For all possible presented interferences, we measure their impact on both communication and computing performances. Finally we studied two important HPC kernels: conjugate gradient (`CG`) and matrix multiply (`GEMM`) that exhibit different computation vs. memory access ratio (arithmetic intensity). We have observed that the degradation of communication performances increases with the number of cores involved in the computation and can be up to 90 %.

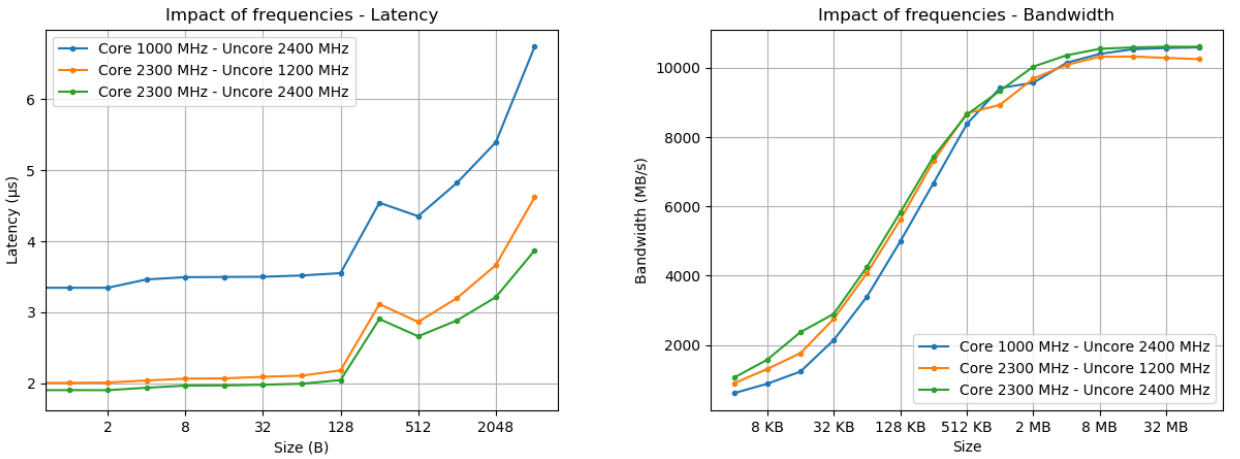
The rest of the paper is organized as follows: section 2 introduces our methodology, describes the benchmarks and the used clusters, next sections 3, 4 and 5 study respectively the impact of hardware frequencies on communications, the impact of memory contention on computations and communications, and the impact of a task-based runtime system on communications. In section 6, we measure communication performances in common computational kernels. Section 7 presents related works and section 8 finally concludes.

2 METHODOLOGY

Our goal is to measure performances of communications and computations when they are run side by side. To achieve this, we have designed a multithreaded and parallel benchmark using MPI+OpenMP. One thread is dedicated to communications (it submits communication instructions and ensures MPI progression) and

ICPP '21, August 9–12, 2021, Lemont, IL, USA

ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3473516



(a) Impact of constant frequencies on network latency

(b) Impact of constant frequencies on network bandwidth

Figure 1: Impact of constant frequencies on network performance on `henri` nodes

other threads do computations. This communication benchmark performs ping-pongs to measure network latency and bandwidth.

2.1 Benchmarking protocol

We need to compare performances of communications and computations when they are executed alone and when they are executed together. Therefore we decomposed our benchmark into the following steps:

(1) Computation without communication
(2) Communication without computation
(3) Computation with side by side communication

Computations and communications use different data and hence are completely independent. The majority of plots in this article compares performance of communications and computations when they are executed separately or simultaneously. The former are represented by plain curves and the later by dashed curves. Curves represent median value of the results obtained with several runs and background areas are delimited by the first and the last decile of these results.

Communications and computations are done in dedicated threads, in the same process. Each thread (computing ones and communicating one) is bound to a different core to stabilize performances and ensure reproducibility. In the remaining of the paper, we will call *computation cores* the cores that execute the computation threads and *communication core* the core that executes the communication thread.

We use the same metrics as NetPIPE [18]: *latency* represents the duration of a communication (time elapsed between the beginning of `MPI_Send` and the end of `MPI_Recv`, *i.e.* half ping-pong) and *bandwidth* is the obtained network throughput by dividing the size of the transmitted data by this *latency*. In this paper, unless stated otherwise, when we do not mention data size, *latency* is measured on 4 bytes of data (one `float`), and *bandwidth* is the asymptotic value, evaluated for 64 MB message size. Buffers used for ping-pongs are recycled to mimic standard applications that update internal data step by step and to take benefit of *registration cache* [20] mechanism. We chose to measure communication performances with ping-pongs for their simplicity, they require only few parameters to be analyzed and we are targeting applications with only point-to-point communications; analyzing also collective communications would be beyond the scope of this article. In the same way, when many threads make MPI communications, it brings many other considerations we do not want to explore in this study.

2.2 Cluster characteristics

We ran our own benchmark suite[1] on several clusters with different characteristics: from small experimental clusters to large production ones. Since results are generally similar on all tested clusters, we present only results obtained on `henri` nodes and mention eventual differences on other clusters. `henri` nodes are dual Intel Xeon Gold 6140 at 2.3 GHz with 36 cores split across 4 NUMA nodes and 96 GB of RAM and equipped with InfiniBand ConnectX-4 EDR. `bora` nodes are dual Intel Xeon Gold 6240 at 2.6 GHz with 36 cores split across 2 NUMA nodes and 192 GB RAM, and equipped with Intel Omni-Path 100 series network. `billy` nodes are dual AMD Zen2 Rome EPYC 7502 at 2.5 GHz with 64 cores split across 8 NUMA nodes with a total of 128 GB of RAM and equipped with InfiniBand ConnextX-6 HDR. `pyxis` nodes are dual Cavium-ARM ThunderX2 99xx at 2.5 GHz with 64 cores split across 2 NUMA nodes with 256 GB of RAM and equipped with InfiniBand ConnectX-6 EDR. Hyperthreading is disabled on `henri` and `bora` nodes.

`henri` nodes run Linux 5.7.7 with Debian 10, gcc 9.3 and Intel MKL 2019.5. We have physical access to `henri` nodes, thus we can change specific settings, especially in the BIOS.

We show results obtained with MadMPI, the MPI interface of NewMadeleine [3] ; we observed similar results with other MPI

[1] Available on https://gitlab.inria.fr/pswartva/memory-contention

implementations, such as OpenMPI 4.0 but they are not reported due to lack of space.

3 IMPACT OF FREQUENCIES

In this section, we study the impact of frequencies on communication performance. To avoid overheating and minimize energy consumption, processors change their frequencies depending on the processor load. This *dynamic frequency scaling* feature of modern processors has a direct impact on computing performance. Since computation may cause changes in processor frequencies, we assess, in this section, whether these frequency variations also have an impact on communications.

We consider two kinds of frequencies: core and uncore. The core frequency impacts computation units and L1 and L2 caches[2]. The uncore frequency [11] concerns last level cache and the memory controller[3]. We measure the impact of these two frequencies independently by setting them to a constant value for all cores and sockets. We evaluate network performances for the two extremes of the permitted ranges of frequencies: 1000-2300 MHz for core frequency and 1200-2400 MHz for uncore frequency.

3.1 Impact of frequencies on only communications

We performed ping-pong benchmarks to measure network latency and bandwidth in function of core and uncore frequencies. Since we study only the impact of frequencies, the ping-pong benchmark relies only on an MPI library and no other runtime. No computation is done at the same time.

Concerning the core frequency, as seen on Figure 1a, the network latency is lower when the frequency is higher: 1.8 μs with 2300 MHz *vs* 3.1 μs with 1000 MHz. We explain this as follows. The network latency is comprised of hardware latency (time to move the data over the wire) and software overhead (time for software to process the communication operation, the *o* of the *LogP* model [6]). Hence, with a lower core frequency, the software overhead is higher. Variations of the CPU frequency do not affect the network bandwidth (Figure 1b), except for the fixed overhead of latency that impacts slightly the bandwidth for small messages. It is explained by the fact that large messages are transferred through DMA, without going through the CPU, thus their speed is unaffected by CPU frequency.

Conversely, the uncore frequency has no impact on the latency (the difference when changing only the uncore is negligible regarding the difference when changing only the core frequency: +5% *vs* +72%) but has a small impact on the bandwidth (10.5 GB/s with 2400 MHz *vs* 10.1 GB/s with 1200 MHz).

3.2 Impact of frequency variations caused by computations

We now observe network performances when one core executes the communications (using the ping-pong benchmark) and other cores are executing CPU-bound computations: a computing benchmark counts in a very naive way the number of prime numbers in an interval. This forces the CPU to execute instructions which do not require any memory access (the algorithm uses only few integer variables).

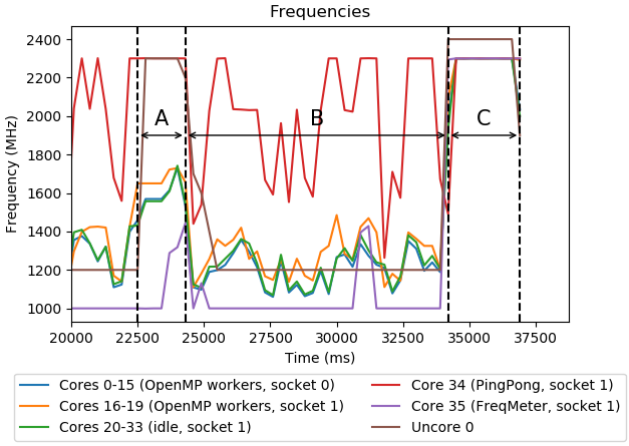
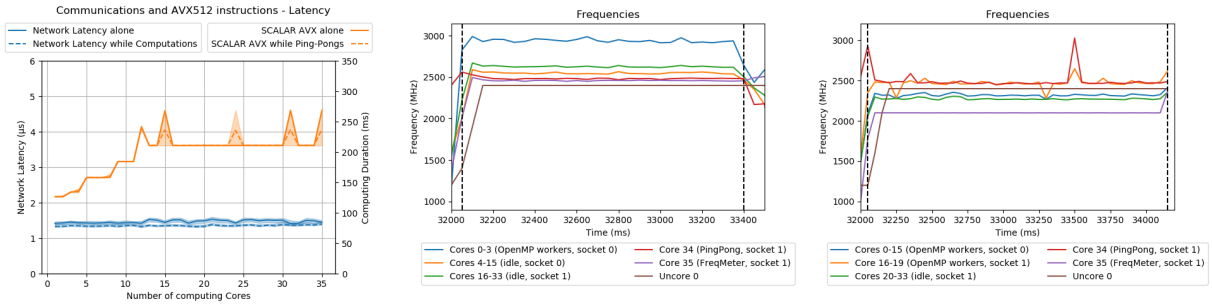


Figure 2: Frequency variations during (A) only communications, (B) sleep and (C) communications and computations, with 20 computing cores on `henri` nodes

In Figure 2, we plot the frequencies of the different cores when (A) only communication is performed, (B) all cores are idle and (C) communications are performed while 20 cores[4] are executing the compute-bound benchmark. We see that all cores have a higher frequency when computations and communications are done at the same time as when communications are executed alone. We have also measured the bandwidth and the latency when communications and computations are done side by side: the network bandwidth is very slightly improved when computation is done at the same time (9097 MB/s *vs* 9063 MB/s), as the network latency (1.52 μs *vs* 1.7 μs). As the CPU frequency of communication core is the same in case (A) and (C) we conclude that the communication latency is not related only by the frequency of the core doing these communications: when other cores increase their frequency, it improves the communication latency.

However, these results seem to be hardware-dependent: on `bora`, the network bandwidth has a wide deviation[5] and computations are highly impacted by the communications when there are more than 15 computing cores. The computing duration jumps from 183 ms to 236 ms: the computation is slowed down when it starts using the socket performing communication. Network latency is constant and duration of computations done along the latency benchmark is also constant regardless of the number of computing cores, exactly like on `henri` nodes.

[2]We use the `userspace` governor and the `cpupower` tool to set the constant core frequency we wish, otherwise the `performance` governor is used.
[3]We use Likwid to get and set the uncore frequency.
[4]We have observed that the performances of both computations and communications are constant regardless the number of computing cores: the cores perform all the same computations.
[5]We observed this behaviour on other clusters equipped with Intel Omni-Path networks.



(a) Network latency

(b) Frequency variation with 4 computing cores and communications

(c) Frequency variation with 20 computing cores and communications

Figure 3: Impact of AVX512 computations on network latency on `henri` nodes with turbo-boost

3.3 Impact of AVX instructions on frequencies

Computing cores can use wide vector instructions such as those from the AVX family [13]. Although these instructions allow reaching better computing performance, they force the cores executing them to reduce their frequency because these instructions consume more power [8]. The core frequency is further reduced when there are more cores executing AVX instructions at the same time. On the other hand, with *turbo-boost*, if only few cores execute AVX instructions, these cores can increase their frequency.

We study here if computing cores doing AVX instructions can have an impact on the frequency of the core executing the communication thread and thus change the communication latency. In our experiment, each computing cores do the same amount of computation: a set of multiple AVX512 floating instructions (weak scalability). Since the range of frequency variation is higher when turbo-boost is enabled, we show only this case here (results are similar when turbo-boost is disabled).

As expected, computations are faster with only few computing cores (Figure 3a). With only 4 computing cores (Figure 3b), computing cores work at 3 GHz and computations last 135 ms and with 20 computing cores (Figure 3c), their frequency is 2.3 GHz and computations last 210 ms (lowered core frequency increases computing duration). In both cases, the frequency of the communication core is stable at 2.5 GHz and is not negatively impacted by cores executing AVX instructions: no matter the number of computing cores, the network latency is always slightly better when computations are done at the same time (1.33 μs *vs* 1.49 μs) of computations. This is consistent with what we have observed in section 3.2 on the same machine: the uncore frequency (constant regardless the number of computing cores) has no effect on network latency, while a higher core frequency can improves latency. On `bora` nodes, computation and communication performances with AVX instructions are the same as those observed without AVX instructions.

To summarize, cores executing AVX instructions do not impact the frequency of the core executing the communications and thus communication performances are not affected.

3.4 Conclusion on the impact of frequency variations

In conclusion, we have observed that the core CPU frequency impacts communication latency: the higher the frequency, the lower the latency. To a lesser extent, uncore frequency slightly impacts the communication bandwidth. Computations can change the frequency of the cores executing them, but do not change the frequency of the core executing communications, even with AVX instructions, and hence do not impact the communication performance. On the contrary, latency is slightly better when computations are made at the same time.

Since changing the frequency settings (especially for uncore frequency) needs an access to the BIOS, we could perform these experiments about frequencies only on machines we have a physical access to. As explained in section 2.2, this concerns only `henri` nodes.

4 MEMORY CONTENTION

Data moving from memory to the CPU and data moving from memory to the NIC are actually using the same memory bus. Hence, in this section, we study the interaction between memory accesses used for computations and communications over the network, to check whether there may be contention between memory accesses for computation and network.

4.1 Benchmarking memory contention

To see what happens when memory contention occurs, we produce memory contention with the STREAM benchmark suite [16]. In this paper, we use the following two STREAM kernels that perform simple arithmetic on large arrays:

`COPY` copy each element of an array to another one: $b[i] \leftarrow a[i]$

`TRIAD` multiply each element of an array by a constant, add it to the element of another array and store the result in another one: $c[i] \leftarrow a[i] + C \times b[i]$

These kernels are memory-bound, causing high pressure on the memory bus. Moreover to really produce memory contention, we allocate memory on a single NUMA node, to increase the traffic on the memory bus between cores belonging to different NUMA nodes. The loop iterating over these arrays is parallelized on available

computing cores with OpenMP. The performance of the computing benchmark is measured using the memory bandwidth *per core* (hence higher is better). For communication we execute a ping-pong benchmark (see section 2.1) in its own thread. Such communication benchmark is run alongside STREAM in a separate thread to measure the impact of interferences.

4.2 Impact of memory contention

In the execution reported in Figure 4, memory is allocated on the NUMA node where the NIC is also connected, communication thread is bound to the last core of the other NUMA node, and computing threads are bound to cores respecting the order of the logical core numbering. Figure 4a shows that network latency is impacted by the STREAM operations when there are at least 22 computing cores and this impact can double the regular latency when all available cores are computing. However, STREAM operations are not impacted by the ping-pong benchmark. The network bandwidth is impacted sooner, from 3 computing cores (Figure 4b). When all available cores are computing, the network bandwidth is reduced by almost two third from the regular network bandwidth. Memory bandwidth measured by the STREAM benchmark is lower when network bandwidth is measured at the same time as when network latency is measured, which is expected because one bandwidth ping-pong transfers more data than a latency ping-pong (64 MB *vs* 4 B). On `bora` nodes, the network bandwidth is impacted, but later: from 20 computing cores; latency gives similar results. Results on `billy` and `pyxis` nodes are similar to those observed on `henri` nodes.

4.3 Impact of thread and data placement

Here, we study the impact on the performances of the data locality and the communication thread mapping. To do so, we bind the communication thread to a core either on the same NUMA node where the NIC is plugged (*near the NIC*) or on the other NUMA node (*far from the NIC*). Similarly, we explicitly allocate the memory (used for computations and communication) either on one or the other NUMA node. It is known [17] that placement may have an impact on communication performance; we check whether contention worsens this phenomenon.

On `henri` nodes, Figure 4 shows results for data near the NIC and communication thread far from the NIC and Figure 5 shows results for other placement schemes. Results on other clusters were similar. When the communication thread is bound near the NIC, the latency increases as soon as we use more than 6 computing cores, but then stays around 2 μs (even if the error range is higher). When the communication thread is far from the NIC, the latency increases considerably from 25 computing cores to double its value. Without computations at the same time, latency is slightly better when communication thread is bound near the NIC (1.39 μs *vs* 1.67 μs). As expected, because small messages are sent from the CPU to the NIC (using programmed I/O), thus if the communication thread is closer to the NIC, the latency is better.

Bandwidth curves have generally the same shape wherever the communication thread is bound. When data is located near the NIC, bandwidth decreases steadily when the number of cores increases. When data is located far from the NIC, bandwidth drops much more abruptly. Since large messages are sent through DMA, the crucial factor is data placement; when data is closer to the NIC, we observe better overall bandwidth and less impact of memory contention than when data is far from the NIC.

In all configurations, latency benchmarks do not impact the STREAM benchmark performance, but bandwidth benchmarks do: STREAM loses at most 25 % (with 5 computing cores) of its performance when executed side by side of communications.

To sum up, placement of communicating thread and locality of memory has an impact on network performances and on the memory contention. When the communication thread is far from the NIC, latency suffers more from contention on the memory bus. When data is far from the NIC, network bandwidth suffers more from contention. Moreover, in all configurations, transmitting large messages on the network increases the impact of contention on the computations. The Table 1 summarizes the impact of data and communication thread placements.

4.4 Impact of transmitted data size on memory contention

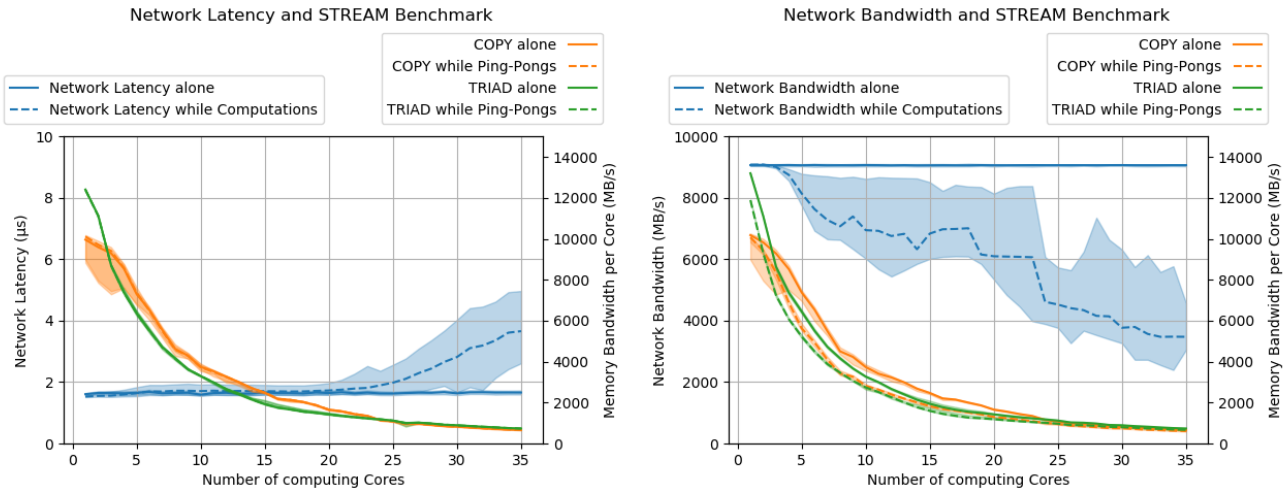
In this section, we study the impact of message size on the contention on the memory bus. We have observed in the previous section that bandwidth benchmarks are more impacted by contention than latency benchmarks. Moreover, communication libraries can exhibit different behaviours according to the size of data to transmit (*e.g.* switch from *eager* to *rendez-vous* protocol). We measure STREAM and network performances with varying message size transmitted through the network, so as to know which message sizes cause a performance drop. We performed this benchmark with two different numbers of computation cores: 5 cores, which is the point where STREAM is the most impacted by communications; and 35 cores, where the network bandwidth is the most impacted by STREAM, as we saw on Figure 4b.

With 5 computing cores (Figure 6a), communications begin to be degraded with a message size of 64 KB, but STREAM begins to be impacted sooner, from 4 KB transferred over the network. With 35 computing cores (Figure 6b), communications begin to be degraded sooner than with 5 computing cores: from 128 B and STREAM is impacted from 4 KB as well. Results were similar on other clusters, when there is a small computing cores or when all available cores are computing.

On the whole, a large number of computation cores causes a high memory pressure which impacts communication for a wide range of message sizes on the network. Conversely, large messages exchanged through the network cause enough traffic on the memory bus to impact computation even with only 5 cores.

4.5 From CPU- to memory-bound applications

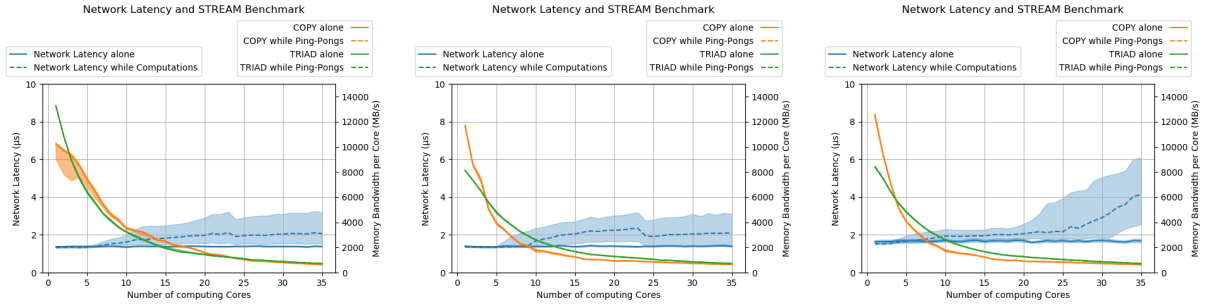
Real-world applications are usually not full CPU-bound or full memory-bound, but somewhere between these two extremes. Previous sections showed that CPU-bound applications have almost no impact on communications, but memory-bound ones do. Hence we modified the `TRIAD` algorithm in the STREAM benchmark to be able to tune the memory pressure, so as to see how the variation of this pressure degrades the network performance.



(a) Network latency

(b) Network bandwidth

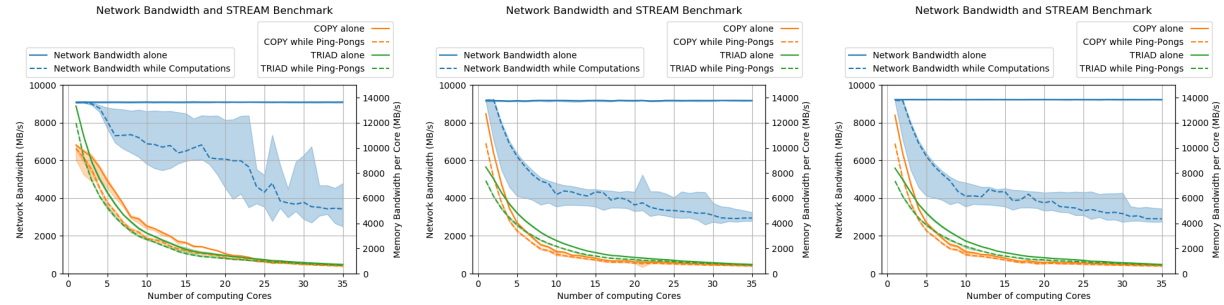
Figure 4: Memory-bound computations and network performances on `henri` nodes



(a) Network latency, data: near the NIC, MPI thread: near the NIC

(b) Network latency, data: far from the NIC, MPI thread: near the NIC

(c) Network latency, data: far from the NIC, MPI thread: far from the NIC



(d) Network bandwidth, data: near the NIC, MPI thread: near the NIC

(e) Network bandwidth, data: far from the NIC, MPI thread: near the NIC

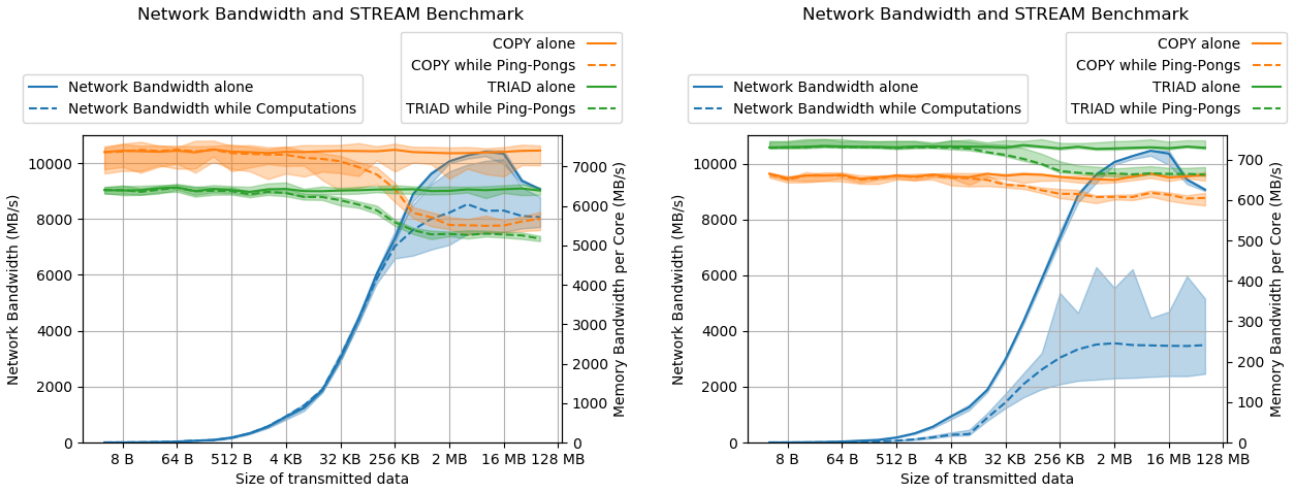
(f) Network bandwidth, data: far from the NIC, MPI thread: far from the NIC

Figure 5: Impact of communication thread placement and data locality on `henri`

The memory pressure that computation causes is expressed as the *arithmetic intensity*: as used by the roofline model [21], it is defined as the number of flops per byte of moved data. In practice, to make the arithmetic intensity of our benchmark tunable, we added a simple loop in STREAM, repeating the operation on each item of the array, before moving to the next item. By changing the number of repetition on each item, the arithmetic intensity varies: with few repetitions the program moves rapidly from one item of the array to the next one (memory-bound) and with many repetitions, it spends more time before accessing the next item (CPU-bound). We call this number of iterations per item in the array a *cursor*: changing

Table 1: Summary of impact of data and communication thread placement related to NIC location

Placement			Network performances		Computations
Data	Comm. thread	Figure	Latency	Bandwidth	STREAM `COPY`/`TRIAD`
Near	Near	5a, 5d	Increases slightly, starting from 6 computing cores	Decreases steadily	Impacted only with
Near	Far	4a, 4b	Increases highly, starting from 25 computing cores	Decreases steadily	high quantity of data
Far	Near	5b, 5e	Increases slightly, starting from 6 computing cores	Decreases abruptly	transmitted through the
Far	Far	5c, 5f	Increases highly, starting from 25 computing cores	Decreases abruptly	network (see section 4.4)



(a) 5 computing cores

(b) 35 computing cores

Figure 6: Impact of size of communicated data on `henri` nodes

the cursor value thus makes the benchmark progressively moving from being memory-bound to CPU-bound.

Results of this benchmark on `henri` nodes are depicted in Figure 7. We observe that high levels of memory pressure cause a huge performance drop for the network. When the arithmetic intensity is lower than 6 flop/B, the memory pressure has an impact: in the latency benchmark (Figure 7a), the network latency doubles and the computing duration is constant, which is the confirmation that it is actually memory-bound, and unaffected by the small messages on the network. In the same range in the bandwidth benchmark (Figure 7b), the network bandwidth drops by 60 % and the computation is slowed down by 10 % because of interference with network operations which use large packets in this benchmark.

For arithmetic intensity higher than 6 flop/B, the program becomes CPU-bound, the memory pressure decreases: communication performance gets back to its nominal value and computation time is unaffected by network interference. On `billy`, the boundary between memory- and compute-bound programs is at 20 flop/B: it is also visible on both computation and communication performances, but the network bandwidth becomes not impacted by computations only when the arithmetic intensity is higher than 70 flop/B.

4.6 Conclusion on memory contention

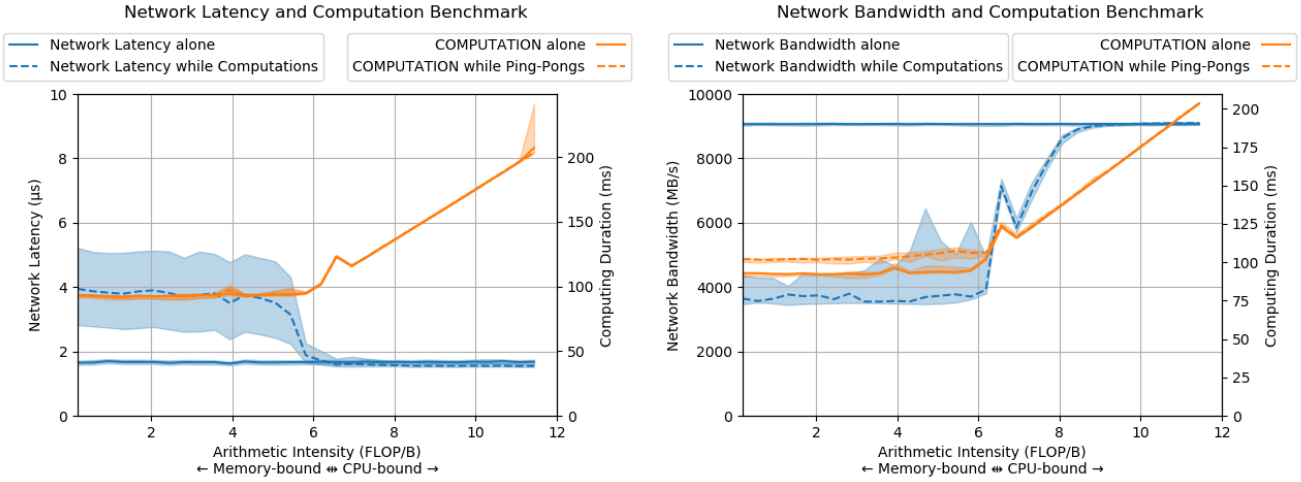
Contention on the memory bus, caused by data movement between main memory and cores or NIC, can have a strong impact on performances. We have shown that the impact depends on several factors: (1) the placement of the communication thread and the data locality, since contention amplifies the impact of NUMA effects on the network; (2) the size of the messages transferred over the network, since larger messages cause a higher impact on computation performance; and (3) the arithmetic intensity of the code executed by computing cores, since code with low arithmetic intensity (thus high memory pressure) have a higher impact on network performance.

5 RUNTIME SYSTEM IMPACTS ON COMMUNICATIONS

In this section, we study the impact of a *task-based* runtime system on communication performance. Such runtime systems are commonly used to program high performance applications and feature threads for executing communication alongside with thread(s) to execute communications.

5.1 Task-based runtime systems

A task-based runtime system uses a computational model where applications are modeled by a task graph. The runtime system



(a) Network latency **(b) Network bandwidth**

Figure 7: Impact of memory pressure on network performance on `henri` nodes

executes this task graph by (1) respecting the dependencies of the graph, (2) transmitting the data between tasks and (3) scheduling the tasks among the available resources. When such runtime system works on distributed memory, it also handles data transfers between hosts by exchanging messages (*e.g.* with an MPI library).

In this work, we focus on the StarPU [1, 2] runtime system. Computations are made by StarPU's threads executing tasks, called *workers*. On a host, StarPU uses the resources as follows. Among all the available cores, one is reserved for the communication thread and another one is reserved for the main thread (which submits tasks to the scheduler of the runtime system). By default, one worker thread is bound per remaining core.

We measure here the impact caused by StarPU on communication performances by executing a ping-pong benchmark, written with the StarPU API instead of plain MPI.

5.2 Runtime system overhead

Using the StarPU API for the communications adds extra software layers on the path of messages that have to go through message lists, be processed by a worker and then by the communication thread. These mechanisms add an overhead to communication performances: in StarPU, the latency is increased by 38 μs on `henri` nodes, by 23 μs on `billy` nodes and by 45 μs on `pyxis` nodes. This latency difference is also noticeable on network bandwidth benchmark for messages smaller than 64 MB.

5.3 MPI thread and data placement

Within StarPU, a thread is dedicated to communications and makes them progress. This thread is usually bound to a dedicated core (similarly to what we did in section 4.3). The issue of memory locality and communication thread placement is still present when memory is directly allocated by workers, namely by different cores. If there are workers on all available cores and memory allocation

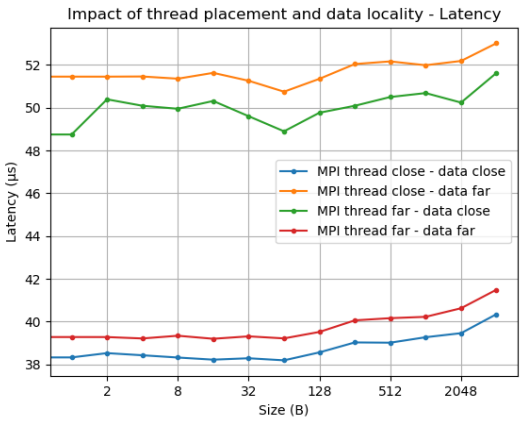


Figure 8: Impact of data locality and thread placement on network latency with StarPU on `henri` nodes. *close* means on the same NUMA node as the NIC and *far* means on the other NUMA node.

uses the *first-touch* strategy, memory will be allocated on different NUMA nodes. Therefore, the performance of the transfer for those messages should depend on where the memory was allocated regarding the NIC, as observed previously.

The Figure 8 shows the network latency overhead explained previously, but mainly that the most important for the network latency is that data to transfer and the communication thread are on the same NUMA node. That is expected, because for small messages, if the communication thread needs a remote NUMA access to get the data to send, it adds some delay to the latency. StarPU does not impact more the network bandwidth than previously observed.

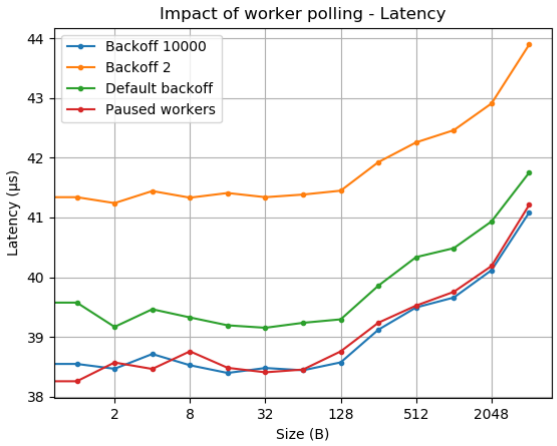


Figure 9: Impact of polling workers on network latency on `henri` nodes

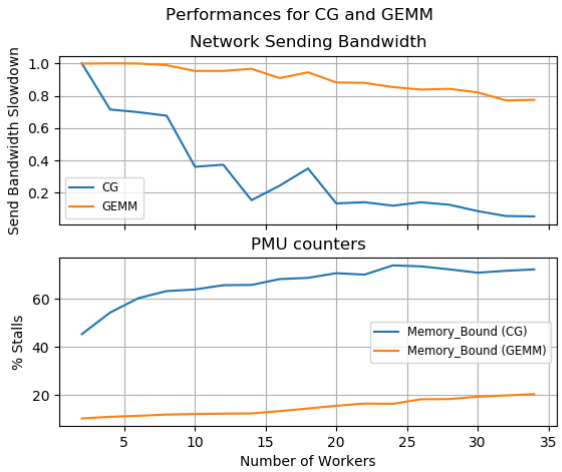


Figure 10: Network performances and hardware counter values of conjugate gradient execution on `henri` nodes

5.4 Worker polling

The scheduler stores tasks submitted to the runtime system in a list. When a worker finishes a task execution, it consults this list to get the next task to execute. To be reactive enough, workers wait actively for tasks, this mechanism is called *busy waiting* or *polling*: if the list is empty, the worker waits a moment by executing a number of `nop` instructions, and then tries again to get a task. The number of `nop` is defined by an *exponential backoff* algorithm: it is doubled after each unsuccessful poll until a maximum is reached. The number is reset to its minimum when the worker finally gets a task. The maximum number of `nop` instructions can be defined by the user. With a small number, workers will be very reactive when a new task is pushed to the list (the task will start sooner). However, it produces traffic on the memory bus, because this list of tasks is shared among all workers. Worker polling can be interrupted by pausing workers.

To study the impact of polling done by workers on communications, we implement a benchmark with a ping-pong on network running without any task to execute, hence workers are constantly polling for new tasks. We executed the benchmark with default configuration (the default maximum number of `nop` instructions is 32), with a huge backoff (10000: workers poll rarely), with a small backoff (2: workers poll very frequently) and with paused workers (they are not polling at all). Figure 9 shows that polling workers have an impact on communication latency: the latency is higher when workers poll more often. A long period between two polls is equivalent to paused workers and does not impact the latency. We explain this result by the increased traffic induced by workers accessing the list of tasks and locking mechanisms. However polling workers have no impact on communication performances on `billy` and `pyxis` nodes, probably because of different mechanisms to handle locking.

5.5 Conclusion on runtime system impact

Our experiments show that task-based runtime system (here StarPU) can negatively impact the communication performance (especially latency) because of the longer software stack messages have to go through and due to aggressive worker polling.

6 USE-CASES: COMPUTATIONAL KERNELS

To measure interferences between communications and computations in real computation codes, and especially the impact of computations on communications, we executed a dense conjugate gradient (`CG`) and a dense general matrix-matrix multiplication (`GEMM`), both built with StarPU, using the Intel MKL BLAS library and distributed on two `henri` nodes (2 MPI processes are enough to see the interferences and simplify the analysis). Using the profiling utility provided by the communication library, we measured the time spent to send data over the network. This gives a sending network bandwidth: the network bandwidth as perceived by the sending node, not taking into account the time to receive data on the receiving node. We also used `pmu-tools`[6], a tool built on the top of the Linux `perf` program to read CPU performance counters, to evaluate the memory pressure caused by the computations. Regardless of the number of computing cores, the execution parameters are the same: matrix sizes and/or number of iterations, hence the amount of network communications is also the same.

The Figure 10 depicts measured values according to the number of computing cores. All curves are the average of values obtained on the two MPI processes. The top plot represents the normalized bandwidth for network sends and the bottom one plots the proportion of execution time when the CPU was stalled on accesses to memory. This figure shows that, the more there are computing cores, the more cores are spending time to access the memory and hence affects negatively the sending bandwidth, as observed previously with our micro-benchmarks. `CG` is more affected by this effect than `GEMM`. This is because `CG` is more memory bound than

[6] https://github.com/andikleen/pmu-tools

GEMM: with the maximum number of workers, 70 % of stalls are caused by memory accesses, while it is only 20 % with GEMM. As seen previously, these different memory pressures affect differently the network send bandwidth: with GEMM, communications lose at most 20 % of performances, while with CG, the loss is up to 90 %.

To sum up, common computational kernels, such as CG and GEMM, can have a significant impact on communications executed at the same time as computations. This impact depends on the arithmetic intensity of the executed kernels.

7 RELATED WORKS

A lot of research is done about the impact of CPU frequency scaling, mostly to save energy. However most of these works consider communication phases as a good opportunity to reduce CPU frequency, because communications would be less CPU-intensive. In our work, we want to reach maximal performance of communications.

Liu *et al.* [15] studied power consumption of RDMA communications. They noticed that RDMA consumes less CPU cycles and memory bandwidth than TCP. Moreover, CPU frequency has almost no effect on RDMA performance, unlike TCP. Their work focuses on power consumption and only on communications.

In order to save energy, Lim *et al.* proposed [14] to decrease CPU frequency in communication phases of executed programs. They observed that reducing frequency does almost not degrade communications. However they only use Ethernet-100, which does not behave the same as high-performance networks.

Sundriyal *et al.* applied [19] DVFS (*dynamic voltage and frequency scaling*) and CPU throttling techniques during collective communications to reduce energy consumption. They accepted a communication performance loss of 10 % and changed behavior of only the communication core, not the whole machine.

Regarding memory contention, previous works focus mainly on impact of memory contention on computation and tend to neglect communication performances.

Memory contention caused by communications and computations is observed by Chai *et al.* [5]. They did not measure the impact of this contention.

Balaji *et al.* studied [4] CPU load and memory traffic caused by communications with TCP/IP over 10 Gbps Ethernet and with RDMA over 10 Gbps InfiniBand. They did not discuss the interaction with simultaneous memory-bound computations.

A theoretical model of the memory bandwidth sharing between computing and communicating threads was made by Langguth *et al.* [12]. Although they considered communications and computations are executed simultaneously, in their model, when communications end before computation, computation gets again all the available bandwidth and *vice-versa* when computation ends before communications. In most programs using StarPU, there are continuously communications and computations, hence there is always an interaction between these two tasks and one task can hardly get the whole memory bandwidth.

NiMC (*Network-induced Memory Contention*) is introduced by Groves *et al.* [10]: they studied the memory contention generated by network communication on a set of applications with and without RDMA. However they only considered the performance of computation, not the performance of the network communications. The solutions they proposed are already implemented in our benchmark (using a dedicated core, offloading RDMA transfers) or penalize communications (reducing network bandwidth to reduce memory bandwidth usage).

Gropp *et al.* proposed [9] to improve the *postal* model, commonly used to model ping-pong performances, by taking into account the number of MPI processes accessing the NIC at the same time. It is not applicable to our work since in our benchmarks only one thread handles all communications done by a host. However a similar modeling of the memory bandwidth sharing between communications and computations could be possible.

Other tools [7] benchmarking performances of simultaneous computations and communications exist, but they rather focus on the ability of MPI libraries to make communications progress when they are overlapped by computations.

All in all, our work sets oneself apart by evaluating performances of both computations and communications when they are executed side by side, by measuring which message size and arithmetic intensity causes contention and by evaluating interferences caused by task-based runtime systems.

8 CONCLUSION

Doing parallel computation side by side with communications is one of the key features of task-based runtime systems and MPI libraries to achieve high-performance. However, such feature can have side effects that actually degrade performance. The goal of this paper is therefore to actually measure and study the interference between computation and communication. We report the following findings. Frequency variations caused by computing cores have little impact on communications. However, memory contention caused by memory-bound computing programs and network transfer of big chunks of data has a strong impact on both computation and communication performances. This impact depends on the placement, the arithmetic intensity of the program executed by computing cores and the amount of data transferred across the network. Moreover, using a task-based runtime system can also penalize communications, just with the library overhead, but also with internal mechanisms like polling on task queues. Communication thread placement, data locality and node topology (to which NUMA node the NIC is the closest) also impact performances. We observed the penalty on communications also in the execution of common HPC kernels such as conjugate gradient and matrix multiplication programs. These preliminary detailed results are necessary to be aware of these behaviours, and before being able to present solutions.

As future works, we would like to better understand origins of these interferences to predict and quantify them. To avoid these interferences, task-based runtime systems could select (automatically) the optimal number of workers which reduces memory contention and maximizes performances for the whole program execution, or change dynamically the number of workers if there are identifiable communication phases. The task scheduler could try to give tasks to workers in a way to minimize data movements between NUMA nodes and to take into account communications involving data used

by these tasks. Future work also includes considering the impact of data movements between main memory and GPUs.

ACKNOWLEDGMENTS

This work is supported by the Agence Nationale de la Recherche, under grant ANR-19-CE46-0009.

This work is supported by the Région Nouvelle-Aquitaine, under grant 2018-1R50119 *HPC scalable ecosystem.*

Experiments presented in this paper were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see https://www.plafrim.fr/).

This work was granted access to the HPC resources of CINES under the allocation 2019-A0060601567 attributed by GENCI (Grand Equipement National de Calcul Intensif).

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

The authors furthermore thank Emmanuel Agullo, Brice Goglin, Amina Guermouche and Samuel Thibault for their help and advice regarding this work.

REFERENCES

[1] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Thibault. 2017. Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. *IEEE Transactions on Parallel and Distributed Systems* (2017). https://hal.inria.fr/hal-01618526

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198. https://doi.org/10.1002/cpe.1631

[3] Olivier Aumage, Elisabeth Brunet, Nathalie Furmento, and Raymond Namyst. 2007. NewMadeleine: a Fast Communication Scheduling Engine for High Performance Networks. In *Workshop on Communication Architecture for Clusters (CAC 2007)*. Long Beach, California, United States. https://hal.inria.fr/inria-00127356

[4] Pavan Balaji, Hemal Shah, and D.K. Panda. 2004. Sockets vs RDMA Interface over 10Gigabit Networks: An In-depth analysis of the Memory Traffic Bottleneck. (01 2004).

[5] L. Chai, Q. Gao, and D. K. Panda. 2007. Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*. 471–478. https://doi.org/10.1109/CCGRID.2007.119

[6] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. 1993. LogP: Towards a Realistic Model of Parallel Computation. *SIGPLAN Not.* 28, 7 (July 1993), 1–12. https://doi.org/10.1145/173284.155333

[7] Alexandre Denis and François Trahay. 2016. MPI Overlap: Benchmark and Analysis. In *International Conference on Parallel Processing (45th International Conference on Parallel Processing)*. Philadelphia, United States. https://hal.inria.fr/hal-01324179

[8] Mathias Gottschlag and Frank Bellosa. 2019. Reducing AVX-Induced Frequency Variation With Core Specialization. In *The 9th Workshop on Systems for Multi-core and Heterogeneous Architectures*. Dresden.

[9] William Gropp, Luke N. Olson, and Philipp Samfass. 2016. Modeling MPI Communication Performance on SMP Nodes: Is It Time to Retire the Ping Pong Test. In *Proceedings of the 23rd European MPI Users' Group Meeting* (Edinburgh, United Kingdom) *(EuroMPI 2016)*. Association for Computing Machinery, New York, NY, USA, 41–50. https://doi.org/10.1145/2966884.2966919

[10] T. Groves, R. E. Grant, and D. Arnold. 2016. NiMC: Characterizing and Eliminating Network-Induced Memory Contention. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 253–262. https://doi.org/10.1109/IPDPS.2016.29

[11] David L Hill, Derek Bachand, Selim Bilgin, Robert Greiner, Per Hammarlund, Thomas Huff, Steve Kulick, and Robert Safranek. 2010. The Uncore: A Modular Approach To Feeding The High-Performance Cores. *Intel Technology Journal* 14, 3 (2010), 30–49.

[12] J. Langguth, X. Cai, and M. Sourouri. 2018. Memory Bandwidth Contention: Communication vs Computation Tradeoffs in Supercomputers with Multicore Architectures. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. 497–506. https://doi.org/10.1109/PADSW.2018.8644601

[13] Gregory Lento. 2014. Optimizing performance with Intel advanced vector extensions. *online, September* (2014).

[14] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal. 2006. Adaptive, Transparent Frequency and Voltage Scaling of Communication Phases in MPI Programs. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. 14–14. https://doi.org/10.1109/SC.2006.11

[15] Jiuxing Liu, Dan Poff, and Bülent Abali. 2009. Evaluating High Performance Communication: a Power Perspective. *Proceedings of the International Conference on Supercomputing*, 326–337. https://doi.org/10.1145/1542275.1542322

[16] John McCalpin. 1995. Memory bandwidth and machine balance in high performance computers. *IEEE Technical Committee on Computer Architecture Newsletter* (12 1995), 19–25.

[17] Stéphanie Moreaud and Brice Goglin. 2007. Impact of NUMA Effects on High-Speed Networking with Multi-Opteron Machines. (11 2007).

[18] Quinn Snell, Armin Mikler, and John Gustafson. 1996. NetPIPE: A Network Protocol Independent Performance Evaluator. *IASTED International Conference on Intelligent Information Management and Systems* 1 (06 1996).

[19] Vaibhav Sundriyal, Masha Sosonkina, and Zhao Zhang. 2013. Achieving Energy Efficiency during Collective Communications. *Concurrency and Computation: Practice and Experience* 25 (10 2013). https://doi.org/10.1002/cpe.2911

[20] Hiroshi Tezuka, Francis O'Carroll, Atsushi Hori, and Yutaka Ishikawa. 1998. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. 308 – 314. https://doi.org/10.1109/IPPS.1998.669932

[21] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.