MPI Overlap: Benchmark and Analysis

Alexandre Denis, François Trahay

MPI Overlap: Benchmark and Analysis

Alexandre DENIS
Inria Bordeaux – Sud-Ouest, France
Alexandre.Denis@inria.fr

François TRAHAY
CNRS Samovar
Télécom SudParis
Université Paris-Saclay, France
Francois.Trahay@telecom-sudparis.eu

***Abstract*—In HPC applications, one of the major overhead compared to sequentiel code, is communication cost. Application programmers often amortize this cost by overlapping communications with computation. To do so, they post a non-blocking MPI request, perform computation, and wait for communication completion, assuming MPI communication will progress in background.**

In this paper, we propose to measure what really happens when trying to overlap non-blocking point-to-point communications with computation. We explain how background progression works, we describe relevant test cases, we identify challenges for a benchmark, then we propose a benchmark suite to measure how much overlap happen in various cases. We exhibit overlap benchmark results on a wide panel of MPI libraries and hardware platforms. Finally, we classify, analyze, and explain the results using low-level traces to reveal the internal behavior of the MPI library.

I. INTRODUCTION

On the path towards exascale, one of the main challenges to face is to reduce the cost of communication. For that, improving the raw performance of the network is not enough, and hiding the communication latency is critical [1].

From the application point of view, hiding the communication latency can be viewed as exploiting two hardware resources in parallel: after initiating the communication, while the Network Interface Card (NIC) transfers data through the network, the CPU performs a part of the application computation. Therefore, the program does not waste computing resources while waiting for the end of the communication.

However, in order to effectively reduce the cost of the communication, this solution requires that communications actually progress in the background during the application computation. The effectiveness of the overlapping thus depends on the strategy of the library. In the case of MPI, if the MPI implementation does not provide an overlapping strategy, the whole communication may happen when waiting for the communication completion, *after* computation. In this case, the communication is serialized with the computation and the benefits of overlapping is void.

In this paper, we propose to assess the effectiveness of the overlapping strategy implemented in an MPI library. We present state of the art techniques for making communication progress in the background. We then identify the parameters that may affect the performance of overlapping, and we propose a benchmark suite for evaluating the overlapping strategies in various cases. We present results of this benchmark suite for a wide panel of MPI libraries and hardware platforms. Finally, we analyze low-level traces that reveal the internal behavior of the MPI library so as to explain the benchmark results.

In short, this paper makes the following contributions:

- We propose a metric for measuring the capacity of an MPI library to overlap communication and computation, regardless the pure communication performance.
- We propose a benchmark suite for evaluating the overlapping strategy of an MPI implementation in various real-life situations.
- We evaluate the overlapping performance of 12 different versions of MPI, running on 8 clusters and supercomputers of various types.

The rest of the paper is organized as follows. Section II presents state of the art techniques for overlapping communication and computation and we identify the parameters that may affect the performance of overlapping. In Section III, we present a metric for measuring the performance of overlapping as well as the various test cases of the benchmark suite. Section IV reports the experimental results we obtained. In Section V, we analyze the experimental results and we inspect low-level traces that depict the internals of some of the tested MPI libraries. Section VI presents related works and Section VII concludes.

II. COMMUNICATION AND COMPUTATION OVERLAP

In this Section, we present state of the art techniques for overlapping communication and computation, and we identify the main parameters that may have an impact on the effectiveness of overlapping.

A. Non-blocking communications

The MPI standard defines several ways to exchange messages between processes [2]. The blocking primitives (such as `MPI_Send` or `MPI_Recv`) may block until the end of the communication (or until the message to send has been copied to an internal buffer). Using these blocking primitives in an application may result in spending a lot of time waiting for the completion of communications. Instead of wasting CPU time with blocking primitives, the application developer can use non-blocking primitives (*e.g.* `MPI_Isend`, `MPI_Irecv`, etc.).

Non-blocking primitives only initiate the communication and return to the application that can continue its computation during the progression of the communication. The application can check for the completion of the communication using

MPI_Wait, or MPI_Test. The benefits of non-blocking communication for the execution time of applications have been extensively studied [3], [4], [5].

However, the MPI specification does not guarantee that the communication progresses during the computation [2]. An MPI implementation may process all of a communication during the MPI_Wait. In this case, the communication is not overlapped with the computation, which roughly boils down to the behavior of blocking primitives.

In addition to point-to-point non-blocking primitives that were introduced in the first version of the MPI standard, collective communications can also be performed in a non-blocking way since MPI 3. In this paper, we focus only on point-to-point non-blocking communication; extending our work to collective non-blocking communication is part our future work.

B. Communication progression implementation

To have non-blocking communications progress, several methods are used by MPI libraries. It is relevant to know the different mechanisms to determine what to benchmark.

As permitted by the MPI specification [2], an MPI library may have no mechanism for background progression and have communication progress only inside MPI_* calls. Such a strategy is compliant and widespread. It assumes the application code calls MPI_Test on a regular basis in the computation. We will not consider applications crammed with random calls to MPI_Test in this paper.

The second approach, known as *NIC offloading*, relies on hardware being able to make communication progress thanks to an on-board NIC processor or a DMA accelerator (I/O AT). Most contemporary high-performance network technologies involve such a processor, especially those based on RDMA such as InfiniBand. However, MPI being higher level than bare metal (rendez-vous, datatypes, matching), protocols have to be carefully designed [6], [7] to have the NIC processor make MPI communications actually progress.

A similar case is kernel-based progression. In kernel space, *tasklets* [8] are small tasks to execute work asynchronously in the background. Network drivers that involve kernel — mostly TCP/Ethernet — are built with tasklets and their communications progress asynchronously in the background even when no system call is in progress.

Finally, user-space explicit progression mechanisms, either using threads [9] or tasks that mimics tasklets [10], [11], may be used to have progression independent from the limited hardware capabilities.

C. Overlap significant features

To match patterns used by applications and to detect corner cases caused by the mechanisms described in previous Section II-B, we distinguish the following features to be tested with regard to overlap:

- *side of overlap* — The most commonly supported and tested pattern for MPI overlap is the combination of a non-blocking MPI_Isend and computation. Since receive may behave differently, we have to test the symmetric case, with a non-blocking MPI_Irecv of contiguous data posted before computation on the receiver side. Moreover, asymmetric mechanisms may be used for progression, especially on RDMA where a remote read or write is used depending on which side is available to drive the transfer. Thus it is relevant to test cases where both sides overlap, *i.e.* MPI_Isend on sender side and MPI_Irecv on the receiver side at the same time.
- *non-contiguous data* — Hardware-based or kernel-based progression is limited by the capabilities of hardware or kernel programming interface, in particular to describe data layout. Since typical MPI applications use derived datatypes to send data that is not contiguous in memory, and since these derived datatypes are usually not understood by hardware or kernel (but may be converted to *iovecs* in some cases), it is wise to test non-contiguous datatypes, in addition to a contiguous data block.
- *CPU overhead* — One common way to measure overlap consists in measuring the total transfer time with and without overlapping computation. While this method is relevant to evaluate communication speed, it says nothing about computation speed, or whether communication utilizes CPU cycles that disturb computation. We propose to measure the time for the full sequence on a single end, to check whether overlap has actually happened or communication has delayed computation.
- *multi-threaded computation* — On multicore systems, computation phases are likely to be multi-threaded. Having computation on a single or all cores may have an impact on communication progression, especially on thread-based and kernel-based systems. Thus we need to benchmark overlap with multi-threaded computation to check whether it disturbs progression or not. However, multi-threaded computation introduces synchronization overhead and jitter, so we don't want *all* the benchmarks to be multi-threaded to keep good precision in other tests and separation of concerns.

III. AN MPI OVERLAP BENCHMARK

In this Section, we propose a new benchmark able to measure how much overlap actually happen in various cases.

A. Challenges for benchmark

To design an overlap benchmark that gives reliable and relevant results, we are facing the following challenges.

Exploration space. All existing benchmarks for overlap use either a fixed computation time [6], [9], [12], [13] or a fixed message size [7], [14], [15], [16]. However, depending on the overlapping strategy used by the MPI library, a measurement done with any of the parameters being fixed exhibits only a local behavior, that may or may not represent the global behavior of the library. Thus we propose to explore the full 2-D parameters space (message size $\times$ computation time).

Variability. Performance is likely to vary from one measurement to another. We choose to use the *median* value computed from several round-trips instead of the more common *average*, since it eliminates the influence of outliers mostly caused by process scheduler artifacts. Thus we use high-resolution timers and measure the time for each round-trip, so as to be able to compute the median round-trip time that will be used for graphs.

Uniform timing. We compare performance from various MPI libraries that are likely to use different methods for `MPI_Wtime` implementation. To have uniform timing across MPI libraries, we don't use `MPI_Wtime` and instead always use `clock_gettime`.

Barrier skew. Process skew [12] is a well-know issue for MPI benchmarks, where all processes are not synchronized so that we measure not only the wanted feature but the drift between processes. This issue is usually solved by inserting a barrier before each round. However, our benchmark is so fine grained that using an `MPI_Barrier`, where we don't know which one of the processes will be unlocked first, and is likely to vary from one MPI library to another, is not sufficient. We use an explicit synchronization based on send and receive operations, so that we control which one of the process is ready first; we ensure that the *receiver* will always be ready first.

Uncertainty on receiver side. On the receiver side, we control only when the receive operation is posted, but not precisely when communication actually begins. The barrier skew is typically in the order of magnitude of the network latency. We should add the transmission delay from sender to receiver. Thus actual communication typically begin 2 latencies after the round start time, so in the overlap benchmark, computation may have started before actual communication. This leads to *optimistic* results with actual overlap being worse than what is measured. However, with the considered durations for computation compared to latency on high performance network, this effect may be considered as negligible.

B. Metric

As a unified metric, easy to analyze, and usable across the full range of message sizes and computation times, we define an *overhead ratio*, that is a normalized measurement of the overhead compared to perfect overlap.

To build the ratio, we first define the *overhead*, noted Δ as the difference between the *measured* time $T_{measured}$ and the ideal time $T_{overlap}$ when the overlap would have been perfect. This ideal time is computed theoretically as

$$T_{overlap} = max(T_{comm}, T_{comp})$$

with T_{comm} the pure communication time without computation, and T_{comp} the pure computation time. Thus we obtain the overhead:

$$\begin{aligned}\Delta_{measured} &= T_{measured} - T_{overlap}\\ &= T_{measured} - max(T_{comm}, T_{comp})\end{aligned}$$

This overhead value is an absolute time, that depends on the network speed and computation time and is therefore hard to interpret. We propose to *normalize* it relative to the overhead obtained in the *serialized* case, *i.e.* the overhead we would obtain without overlap. The transfer time of these operations when serialized is:

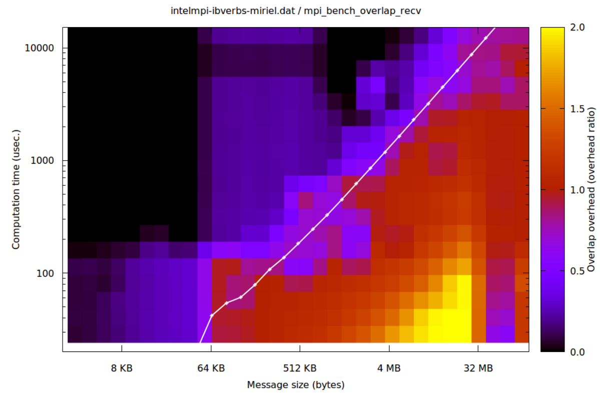


Fig. 1. Example 2-D graph output (Intel MPI, *ibverbs*, receive-side overlap benchmark). The color is the overhead ratio; the white line is T_{comm} for this MPI library, marking the limit between zones where computation is either shorter or longer the communication.

$$T_{serialized} = T_{comm} + T_{comp}$$

and thus the overhead for this worst case is:

$$\begin{aligned}\Delta_{serialized} &= T_{serialized} - T_{overlap}\\ &= T_{comm} + T_{comp} - max(T_{comm}, T_{comp})\end{aligned}$$

simplified as:

$$\Delta_{serialized} = min(T_{comm}, T_{comp})$$

We define the overhead ratio as

$$ratio = \frac{\Delta_{measured}}{\Delta_{serialized}}$$

and thus the final form we get is:

$$ratio = \frac{T_{measured} - max(T_{comm}, T_{comp})}{min(T_{comm}, T_{comp})} \quad (1)$$

We verify that in the ideal case with perfect overlap, we get $ratio = 0$ and in the serialized case, we get $ratio = 1$. We notice that the ratio is not bounded. In particular, it may be greater than 1 in case overlap tentative leads to slower transfer than manually serializing communication and computation. It is however usually non-negative except in case there was some imprecision in measuring T_{comm}.

C. Visualization

To visualize results, we draw a bi-dimensional *heat map* of the ratio defined in equation 1, with message size on the X axis and computation time on the Y axis, as illustrated by the example in figure 1.

The parameters space is explored every power of $\sqrt{2}$ in each direction, so as to get twice as many samples as powers of 2, and represented as a heat map, without interpolation nor smoothing. For each couple of computation time and message size, we run the benchmark 50 times and retain the median value.

Additionally, we draw T_{comm}, the pure communication time without computation. It materializes the border between two zones where communication is longer than computation (bottom right), and computation longer than communication (top left). It corresponds to which side of the min and max in equation 1 are used.

The metric is built so as to be easy to grasp for the reader: interpretation remains uniform across the full range of parameters. On all graphs, we have the following convention:

- black: $ratio = 0$, perfect overlap;
- purple: $ratio$ close to 0, some overlap happen, but not perfect;
- red: $ratio = 1$, no overlap, computation and communications are serialized;
- yellow: $ratio > 1$, overlap tentative made things slower then serialized. Values higher than 2 are cropped to keep the same scale on all graphs.

D. Benchmarks description and implementation

In order to measure the features listed in section II-C using the defined metric, we propose the following list of benchmarks. They run on two nodes, noted node 0 and 1. Time measurements are performed on node 0.

Sender-side overlap: node 0 posts an `MPI_Isend` for a contiguous data block, performs the given amount of computation, calls `MPI_Wait`, wait for the ACK; the other side receives data with a blocking call then sends the 0-byte ACK. $T_{measured}$ is defined as the time for the full sequence on node 0 minus a 0-byte latency.

Receiver-side overlap: node 0 sends a 0-byte CTS, then posts an `MPI_Irecv` for a contiguous data block, performs the given amount of computation, then calls `MPI_Wait`; the other node sends contiguous data with a blocking call. $T_{measured}$ is defined as the time for the full sequence on node 0 minus a 0-byte latency.

Both-sides overlap: node 0 posts an `MPI_Isend` for a contiguous data block, performs the given amount of computation, calls `MPI_Wait`, then posts an `MPI_Irecv` for a contiguous data block, performs the given amount of computation, then calls `MPI_Wait`; node 1 does the same, with receive before send. $T_{measured}$ is defined as half the time for the full sequence on node 0.

Non-contiguous: we use the same benchmark as *sender-side overlap* except that data uses a derived datatype, defined as a vector of blocks of 32 `MPI_CHAR` with a stride of 64 bytes. For this benchmark, T_{comm} is the transfer time with blocking calls using the same derived datatype.

CPU overhead: node 0 posts an `MPI_Isend` for a contiguous data block, performs the given amount of computation, call `MPI_Wait`, without ACK; the other node receives data with a blocking call. $T_{measured}$ is defined as the time for the full sequence on node 0. For this benchmark, T_{comm} is the time used by a blocking `MPI_Send`.

N threads load: both nodes run N threads performing some computation in the background, and *blocking* communications in a dedicated communication thread. Node 0 posts a blocking send, then a blocking receive; the other node, in reverse order. $T_{measured}$ is defined as half the time for the full sequence on node 0. This benchmark requires `MPI_THREAD_FUNNELED` support from MPI library. Since pathological behaviors with this benchmark involve thread scheduling issues, with penalties in the order of magnitude of miliseconds (kernel scheduler time slice), the scale for the colors is displayed with a *logarithmic scale*. The Y-axis is for the number of threads, not for computation duration, since computation runs in the background all the time. We run tests with a number of computation threads up to the number of *processing units* returned by `hwloc`.

Base benchmarks: to measure T_{comm} needed to compute our metric, we run first a round of benchmarks to get reference values for ping-pong with contiguous data, derived datatype, and processor time on sender-side.

Benchmarks *sender-*, *receiver-* and *both-sides overlap* measure how communication progresses in the presence of computation. Benchmark *CPU overhead* measures how computation speed is impacted in the presence of communication– mostly the cost of `MPI_Isend` and `MPI_Wait`. Benchmark *N load* measures how blocking communication progresses in the presence of computation in other threads. Benchmarks for *non-contiguous*, *CPU overhead* and *N load* all use a scheme with only sender-side overlap. It could be combined with receiver-side or both-sides overlap, but we believe that testing these features with sender-side overlap is enough to evaluate the impact of these features while keeping a small enough number of graphs as output.

The full benchmark suite takes from 10 minutes to 2 hours, depending on the network speed and the actual overlap. Then a script processes the data and automatically generates the graphs. The benchmark has been released as open source and is available for download [17]. All the graphs presented in this paper are direct output from the released tool.

IV. EXPERIMENTAL RESULTS

In this Section, we present experimental results obtained with our benchmark on various MPI libraries on a large scope of hardware.

We have run the benchmark on a variety of Linux clusters with portable MPI libraries: cluster `william` nodes are dual-Xeon ES-2650 equipped with ConnectX-3 *InfiniBand* FDR (MT27500); cluster `jack` nodes are dual-Xeon X5650 equipped with ConnextX *InfiniBand* QDR (MT26428); cluster `mistral` nodes are dual Xeon E5-2670 equipped with *InfiniBand* QDR; cluster `miriel` nodes are dual-Xeon E5-2680 equipped with TrueScale *InfiniBand* QDR; cluster `inti` nodes are dual-Xeon E5-2698 equipped with Connect-IB *InfiniBand* QDR (MT27600).

On these machines, we have run benchmarks with multiple versions of OpenMPI [18], multiple versions of MVAPICH [7], MPICH, Intel MPI, MPC [19], and latest version of our own MadMPI [10].

Furthermore, we have run the benchmark on supercomputers using the vendor-supplied MPI library, namely on: `bluewaters` [20], Cray XE with Cray Gemini network, using Cray MPI derived from MPICH, at NCSA; `K computer`

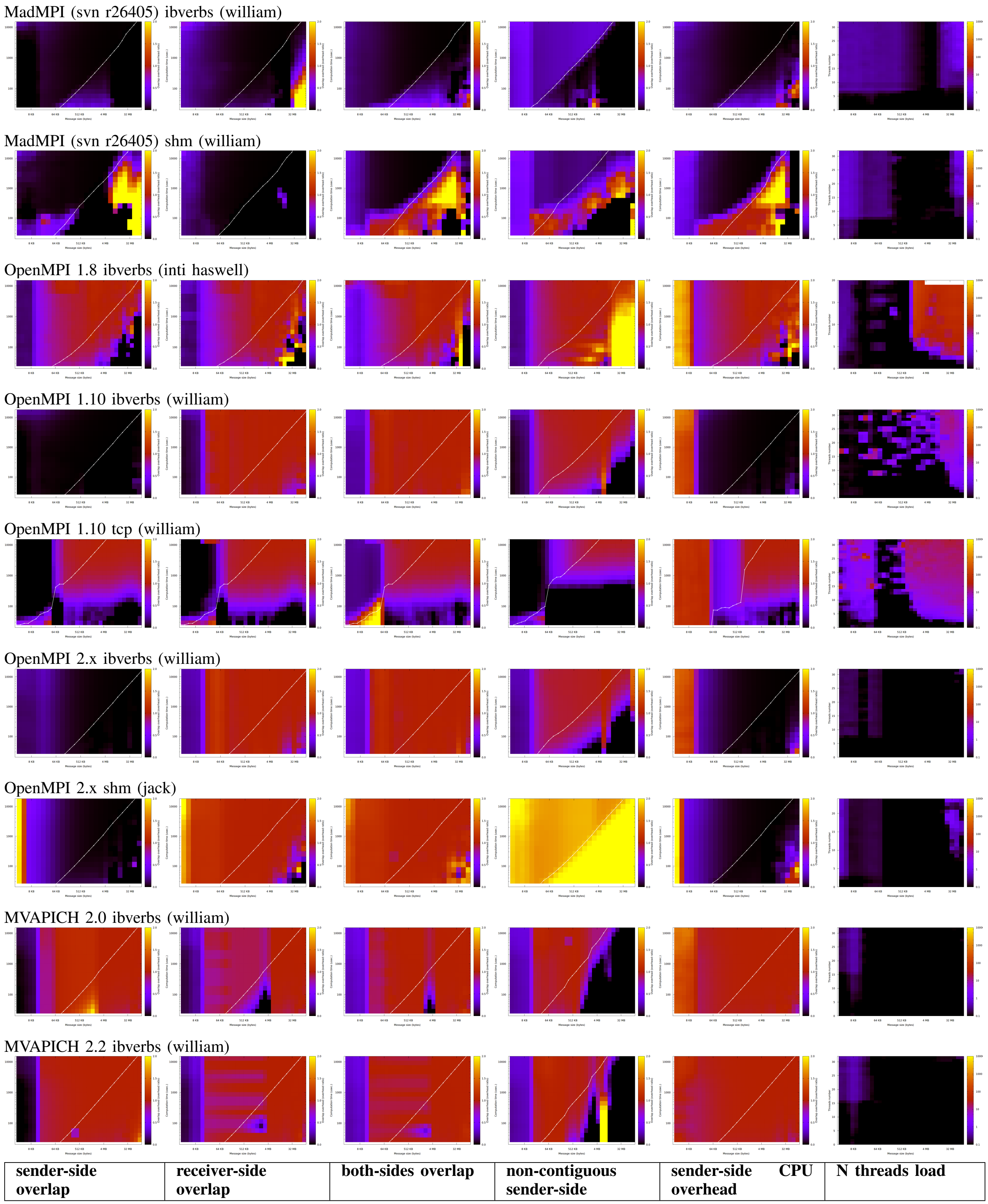


Fig. 2. Overlap benchmark results (1/3)

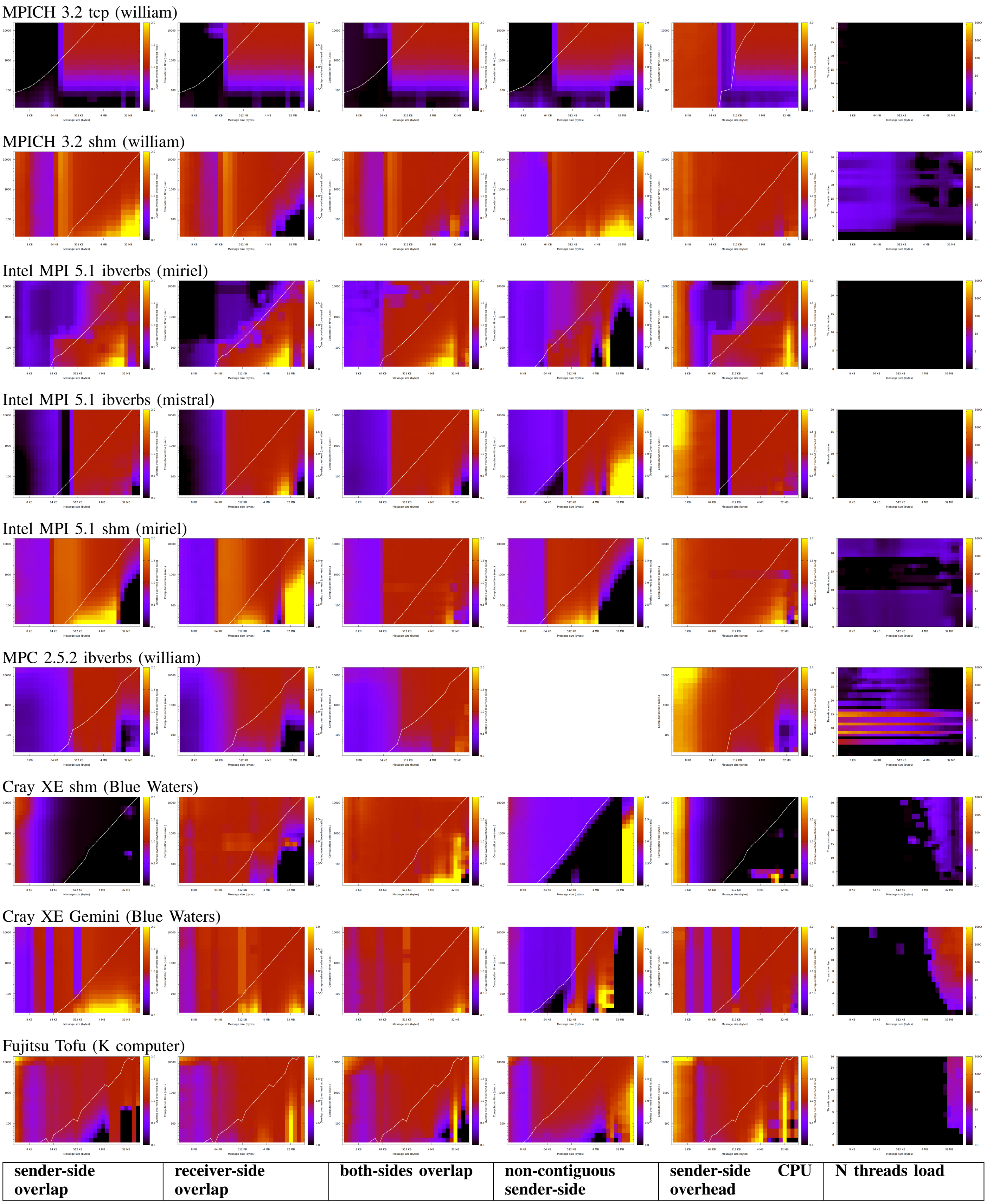
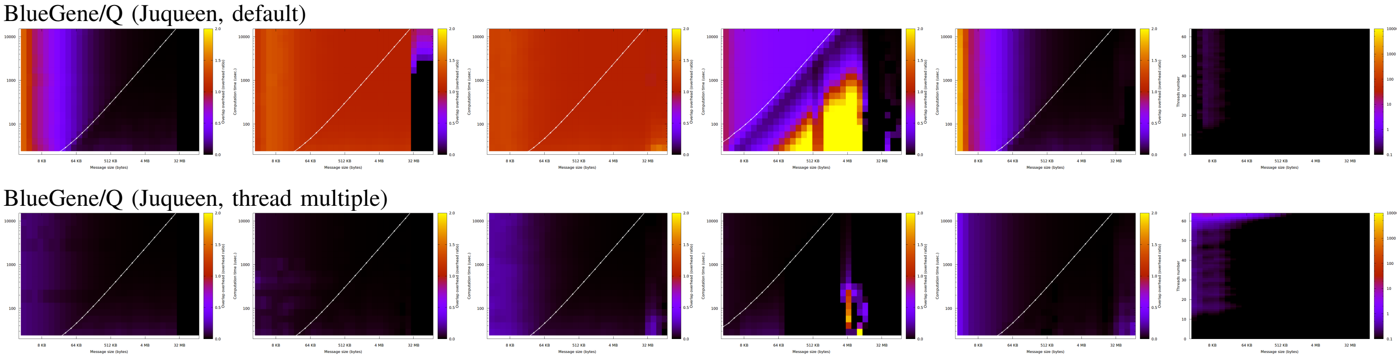


Fig. 3. Overlap benchmark results (2/3)



sender-side overlap	receiver-side overlap	both-sides overlap	non-contiguous sender-side	sender-side CPU overhead	N threads load

Fig. 4. Overlap benchmark results (3/3)

by Fujitsu, with SPARC64 VIIIfx nodes and TOFU network, using the vendor-supplied MPI derived from OpenMPI, at RIKEN; JUGENE, IBM BlueGene/Q, with IBM MPI, at Forschungszentrum Jülich, using both default tuning and *thread multiple* mode as recommended in the IBM Redbook for better overlap.

Benchmark results are depicted in Figures 2, 3, and 4. Each row of graphs is an MPI library on a given machine. Each column is a benchmark from the suite, as defined in Section III-D. We analyze the results in the following Section.

V. ANALYSIS

In this Section, we analyze the results obtained for the benchmarks. We first observe and discuss the results, then we trace low-level networking primitives to really understand what happens under the hood. We prefer real-world observation rather than code analysis to capture the real behavior, which might be different to what is expected from reading code.

A. Classification of results

On the resulting graphs, we observe very different results between MPI libraries, networks, or even between versions of the same MPI library. However we can distinguish some classes of behaviors.

Threaded libraries. Threaded MPI libraries such as MadMPI and IBM MPI on BlueGene/Q exhibit an overall good overlap. MadMPI gets a much better result on *InfiniBand* than on shared memory; this is not surprising given that inter-process shared-memory is not optimized, MadMPI being optimized for a single multi-threaded process per node.

Non-threaded libraries. Non-threaded MPI libraries rely on NIC offloading which cannot guarantee overlap in all cases. Overlap only happens on sender-side (not in all cases), and for small messages on receiver side.

Rendez-vous threshold. Even where the graph is mostly red, *i.e.* in cases no overlap happen, there is usually a stripe on the left with better overlap, that correspond to small message size. This is due to the *rendez-vous threshold*: small messages are sent in eager mode, which is easier to overlap using NIC offloading, while larger messages are sent in *rendez-vous mode* which is harder to overlap, since the MPI library should be called to handle the rendez-vous reply. It it especially true for OpenMPI, MVAPICH, Intel MPI, and MPC.

TCP. On TCP with small computation time, overlap is always good. A close look to the graphs shows that this time threshold corresponds to the round-trip time on TCP/Ethernet. With small computation time, the computation has ended before the sender receives the rendez-vous reply. This behavior is not observed on *InfiniBand* since the latency is much lower and such low computation times are not displayed.

Shared memory. For all results on shared-memory, overlap is significantly worse than on networks. This is due to the fact that no NIC offloading is available; transfer only progresses thanks to CPU.

Non-contiguous datatypes. In most cases, overlap using a non-contiguous datatype is much worse than when using a contiguous data block. It is explained by the inability of the NIC to offload communications with such types. However, in most cases there is a small zone at the bottom right (large packets, short computation time) where overlap overhead is low: this is due to non-contiguous datatypes being sent using a pipelined copy, the first chunk being overlapped. The result is missing for MPC because of a crash; the bug has been forwarded upstream to MPC developpers.

CPU overhead. The CPU overhead benchmark measures CPU usage on sender-side for overlapping non-blocking send. For MPI libraries that do not overlap on the sender-side, unsurprisingly we get an overhead ratio of 1 on this benchmark. Even in cases where overlap happen, for small messages sent in eager mode, we observe a significant CPU overhead due to the cost of `MPI_Isend` itself (*e.g.* small messages for OpenMPI 1.10 on `william`, Intel MPI on `mistral`).

Multi-threaded computation. The N-load benchmark involves blocking communications and computation in others threads. It shows thread scheduling and priority issues. We observe such issues, with performance orders of magnitude slower (color map uses *logarithmic scale* for this benchmark), for older and current version of OpenMPI, but fixed in the upcoming 2.x branch, and for MPC (current 2.5.2 version) due to a thread placement bug that will be fixed in next version. We observe that the *presence* of computation threads has an impact on performance, but the precise number of threads, and their placement on real cores or hyper-threads do not make any

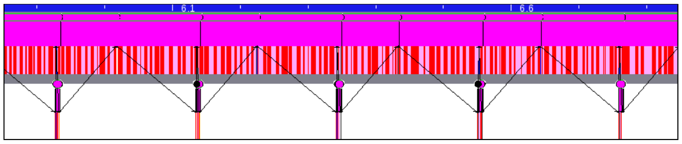


Fig. 5. OpenMPI 1.10, sender-side overlap, 500 KB message, 200 μs computation, receiver on top, sender at bottom. Shown excerpt is 1 ms.

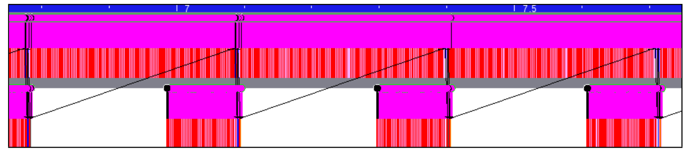


Fig. 6. MVAPICH 2.2a, sender-side overlap, 500 KB message, 200 μs computation, receiver on top, sender at bottom. Shown excerpt is 1 ms.

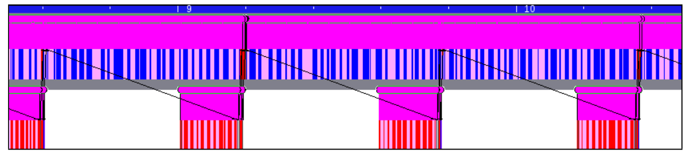


Fig. 7. OpenMPI 1.10, receiver-side overlap, 500 KB message, 200 μs computation, sender on top, receiver at bottom. Shown excerpt is 1 ms.

difference except in some pathological cases (OpenMPI 1.x).

B. Interleaved MPI and InfiniBand traces

To analyze precisely the behavior of MPI implementations when running the benchmark, we collected execution traces. The traces were generated using the EZTrace framework for performance analysis [21]. When running an application, EZTrace intercepts the calls to a set of functions and records time-stamped events for each call to these functions. The recorded events are stored in a trace file that can be analyzed *post-mortem*.

In order to analyze how an MPI implementation interacts with the network, we used two EZTrace [22] plugins: the `mpi` plugin that traces the application calls to the MPI API (*e.g.* `MPI_Isend`, `MPI_Wait`, etc.), and the `ibverbs` plugin that traces the calls to the *ibverbs* API (*e.g.*, `ibv_poll_cq`, `ibv_post_send`, `ibv_post_recv`, etc.) performed by the MPI implementation. Since these plugins rely on the MPI implementation and the *ibverbs* library for actually performing the communication, EZTrace does not change the behavior of the communication. The impact of the trace collection is thus limited to a light overhead (approximately 100 ns per function interception) caused by the event recording mechanism. The events corresponding to both the MPI and *ibverbs* API issued from any thread of the application are stored in the same trace files. Analyzing the resulting traces allows to understand the strategy of an MPI implementation for overlapping communication and computation. For instance, the traces show when a message is submitted to the NIC, when a queue is polled for completion, which thread polls the network, etc.

C. Traces examples and analysis

To precisely explain the performance obtained with our benchmarks, we have run some interesting cases with traces that we present in this Section. All traces use the same color convention for states: white block is computation; red block is `MPI_Wait`; blue block is `MPI_Send`, green block is `MPI_Isend`; pink block is `ibv_poll`. MPI messages are depicted with black arrows. Events are represented with half-height sticks surmounted with a bullet, with the following colors: black is *ibverbs* `ibv_post_*`; pink is *ibverbs* `ibv_poll`. Interactive exploration of traces is needed to get all details associated with events (data size, type of event, zoom), but traces exhibited here should be enough to grasp the global behavior of MPI libraries.

Sender-side overlap. Figure 5 is a trace for sender-side overlap on OpenMPI 1.10 with large message. We can see that the sender spends most of its time in computation (white), with almost no wait time. Interactive exploration of the trace reveals that the sender sends a rendez-vous request, then the receiver performs an RDMA *read*, hence the overlap on the sender-side.

Sender-side, no overlap. Figure 6 is a trace for sender-side overlap benchmark on MVAPICH 2.2a, large message. No overlap actually happens, we observe that network transfer and computation are serialized. Trace exploration reveals that a rendez-vous request is issued immediately, then computation is started, then RDMA write for data and packet notification (using send) are issued from the `MPI_Wait` after computation, hence the `ibv_post_send` event (in black) at the beginning of the `MPI_Wait`. On the sender-side (node at bottom), we clearly see alternating computation (white) and send while waiting (red block, covered with pink polling events).

Receiver-side, no overlap. Figure 7 is a trace for overlap attempted on receiver-side with OpenMPI 1.10, large message. We observe that no overlap actually happens, operations are serialized: the receiver is busy with computation when the rendez-vous request arrives; it notices the request and issues the RDMA *read* only when it reaches `MPI_Wait` after computation.

Both-sides overlap. Figure 8 shows a successful overlap on both sides with MadMPI, large messages. Since the library is threaded, 2 lines (threads) per node appear on trace. Top line is the application thread; and the 2nd line is `pioman` *idle thread* [10]. We observe that computation is done in the application thread, and communication progresses mostly in the other thread, which leads to perfect overlap.

Non-contiguous datatype. Figure 9 is a trace for sender-side overlap with a non-contiguous datatype for OpenMPI 1.10. We observe that almost no overlap happen. The library sends data in pipeline with 64 KB chunks. Only the first chunk

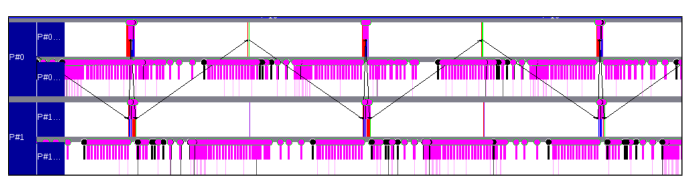


Fig. 8. MadMPI, both-sides overlap, 500 KB message, 200 μs computation, 3 lines per node (1 application thread, 1 `pioman` thread). Shown excerpt is 1 ms.

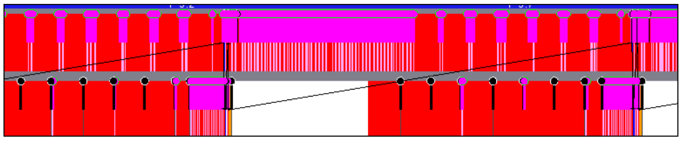


Fig. 9. OpenMPI 1.10, non-contig sender-side overlap, 500 KB message, 200 μs computation, receiver on top, sender at bottom. Shown excerpt is 1 ms.

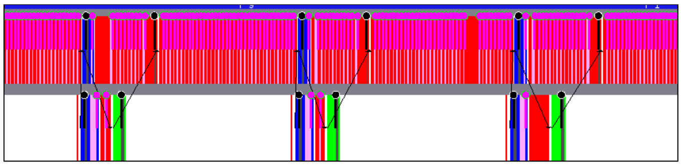


Fig. 10. MVAPICH 2.2a, sender-side overlap overhead, 8 KB message, 20 μs computation, receiver on top, sender at bottom. Shown excerpt is 100 μs.

is posted before computation (overlapped), the other chunks get posted in the `MPI_Wait` after computation; hence, the `ibv_post_send` events (in black) during the `MPI_Wait`.

CPU overhead. Figure 10 is a trace for sender-side CPU overhead with MVAPICH 2.2a, short message (8 KB) sent in eager mode. CPU usage for `MPI_Isend` is represented as green blocks. Deep exploration reveals that blocking `MPI_Send` (trace not shown) for such messages utilizes 1 μs of CPU time; non-blocking `MPI_Isend` (green blocks on Figure 10) take 2 μs, hence the overhead ratio of 1 measured in the benchmark.

We have seen that our interleaved MPI and *ibverbs* traces are a powerful tool to investigate why and how overlap works or not. We can check the frequency of polling, observe the internal protocol used by a given MPI library on *InfiniBand*, and understand the benchmark results. We actually used ourselves these tools to diagnose and fix progression-related bugs in MadMPI.

D. Discussion

The results presented in this paper show that most modern MPI implementations still fail to efficiently overlap computation and communication. The only MPI implementations that successfully hide communication (MadMPI, IBM MPI) rely on threads for making communication progress in the background. The other libraries that rely on NIC offloading are only capable of overlapping small messages (in eager mode) of contiguous data on the sender-side. Thus, most MPI implementations do not overlap efficiently in several cases:

- large messages sent using a rendez-vous protocol
- overlapping on the receiver side
- messages that consist of non-contiguous datatypes

VI. Related works

Some work related to what we present in this paper has already been done by people working on benchmarks and people working on overlapping in MPI libraries. None of them includes an exploration of the 2-D parameters space, nor benchmarks with derived datatypes, sender-side overhead, or multi-threaded computation load.

In the domain of MPI benchmarking, overlapping is an often overlooked issue. Intel MPI Benchmarks (IMB) [13] is a benchmark widespread in the MPI community. It comes with an overlap benchmark for MPI-I/O and non-blocking collectives, but not for point-to-point communications. The metric they use is an *overlap ratio*, quite similar but not identical to our *overhead ratio*.

Mpptest [15] stresses the importance of measuring overlap. However, they use a fixed message size and contiguous data. COMB [14] is a benchmark for assessing MPI overlap that shows the *CPU availability*, by actually measuring the time spent in `MPI_Wait` and `MPI_Test` A framework [16] dedicated to overlap has been proposed. It use fixed message size of 10 KB and 1 MB, and measures the time spent in `MPI_Wait`.

People working on implementing overlap in MPI libraries usually have their own benchmarks [6], [7], expressed as an overlap ratio measured with a fixed computation time.

All these works (except MPI-I/O overlap in IMB) are based on an overlap ratio that is hard to interpret for the reader since it depends on network performance. Their ratio is typically less than 100 % even with perfect overlap when computation is shorter than communication, and it always converges asymptotically to 1 even when no overlap happen. We believe our metric independent from network performance is easier to grasp.

Moreover, our 2-D graphs with full range of both computation time and message size, both-sides benchmarks, non-contiguous benchmarks, multi-threaded benchmarks, large survey of various MPI libraries regarding overlap, and analysis with low-level traces have not been done before.

VII. Conclusion

MPI application programmers try to amortize the cost of communications by overlapping them with computation. To do so, they use non-blocking communication primitives and assume communication progresses in the background. However, depending on MPI libraries and networks, this assumption may not be true.

In this paper, we have proposed tools to assess and analyze the real behavior of MPI libraries regarding overlap. We have released an MPI overlap benchmark, that uses a metric that does not depend on network performance, and checks the impact of various features on overlap (sender/receiver-side, contiguous/non-contiguous data, total transfer time v.s. CPU overhead, single thread v.s. multi-thread). We have shown benchmark results for various MPI libraries on various machines— clusters and supercomputers. We have observed that there are very few cases where overlap actually happens. Finally we have proposed a trace framework to analyze communication progression in MPI libraries on *InfiniBand*.

The tools proposed in this paper have been publicly released [17], [22] as open-source software. They may be useful for application programmers to discover the progression properties of the MPI library on their machine. These tools may be a great help for MPI library authors to detect and diagnose pathological behavior with regard to overlap.

In future works, we plan to extend the benchmark with more tested features. In particular, we have not tested cases with multiple non-blocking requests pending at the same time, with multiple requests bound to the same destination or to different destination nodes, and all schemes with more than two nodes. Another feature that would benefit from more thorough benchmarking is multi-threading. We plan to have OpenMP-based benchmarks, where we check the impact of non-blocking communication on OpenMP performance, and the case of multi-threaded communication, *i.e.* `MPI_THREAD_MULTIPLE`. Finally, we will extended our approach to non-blocking collective operations.

ACKNOWLEDGMENT

Experiments presented in this paper were carried out using the PLAFRIM experimental testbed, being developed under the Inria PlaFRIM development action with support from LABRI and IMB and other entities: Conseil Régional d'Aquitaine, FeDER, Université de Bordeaux and CNRS (see https://plafrim.bordeaux.inria.fr/).

Experiments presented in this paper were carried out using the *inti* cluster at CEA.

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

Thanks to François Tessier and Emmanuel Jeannot who ran the benchmarks on Blue Waters. The access to the machine was done in the context of the Joint Laboratory for Extreme Scale Computing (JLESC).

Thanks to Benedikt Steinbusch who ran the benchmarks on machine "JUQUEEN" at Forschungszentrum Jülich. This collaboration was facilitated by the Joint Laboratory for Extreme Scale Computing (JLESC).

Thanks to Balazs Gerofi from RIKEN, who ran the benchmarks on the K computer.

REFERENCES

[1] J. Dongarra *et al.*, "The international exascale software project roadmap," *International Journal of High Performance Computing Applications*, vol. 25, no. 1, pp. 3–60, 2011.

[2] Message Passing Interface Forum, "MPI: A message-passing interface standard," Knoxville, TN, USA, Tech. Rep., 2015.

[3] J. Sancho, K. Barker, D. Kerbyson, and K. Davis, "Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM New York, NY, USA, 2006.

[4] S. Potluri, P. Lai, K. Tomko, S. Sur, Y. Cui, M. Tatineni, K. W. Schulz, W. L. Barth, A. Majumdar, and D. K. Panda, "Quantifying performance benefits of overlap using mpi-2 in a seismic modeling application," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 17–25. [Online]. Available: http://doi.acm.org/10.1145/1810085.1810092

[5] G. Hager, G. Schubert, T. Schoenemeyer, and G. Wellein, "Prospects for truly asynchronous communication with pure mpi and hybrid mpi/openmp on current supercomputing platforms."

[6] M. J. Rashti and A. Afsahi, "Improving communication progress and overlap in mpi rendezvous protocol over rdma-enabled interconnects," in *High Performance Computing Systems and Applications, 2008. HPCS 2008. 22nd International Symposium on*. IEEE, 2008, pp. 95–101.

[7] S. Sur, H. Jin, L. Chai, and D. Panda, "RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM New York, NY, USA, 2006, pp. 32–39.

[8] M. Wilcox, "I'll do it later: Softirqs, tasklets, bottom halves, task queues, work queues and timers," in *Linux.conf.au*. Perth, Australia: The University of Western Australia, January 2003.

[9] T. Hoefler and A. Lumsdaine, "Message progression in parallel computing-to thread or not to thread?" in *Cluster Computing, 2008 IEEE International Conference on*. IEEE, 2008, pp. 213–222.

[10] A. Denis, "pioman: a pthread-based Multithreaded Communication Engine," in *Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Turku, Finland, Mar. 2015. [Online]. Available: https://hal.inria.fr/hal-01087775

[11] F. Trahay and A. Denis, "A scalable and generic task scheduling system for communication libraries," in *Proceedings of the IEEE International Conference on Cluster Computing*. New Orleans, LA: IEEE Computer Society Press, Sep. 2009. [Online]. Available: http://hal.inria.fr/inria-00408521

[12] T. Hoefler, T. Schneider, and A. Lumsdaine, "Accurately Measuring Overhead, Communication Time and Progression of Blocking and Nonblocking Collective Operations at Massive Scale," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 25, no. 4, pp. 241–258, Jul. 2010.

[13] Intel, "Intel mpi benchmarks 4.1," https://software.intel.com/en-us/articles/intel-mpi-benchmarks/.

[14] W. Lawry, C. Wilson, A. Maccabe, and R. Brightwell, "Comb: a portable benchmark suite for assessing mpi overlap," in *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, 2002, pp. 472–475.

[15] W. Gropp and E. Lusk, "Reproducible measurements of mpi performance characteristics," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, J. Dongarra, E. Luque, and T. Margalef, Eds. Springer Berlin Heidelberg, 1999, vol. 1697, pp. 11–18. [Online]. Available: http://dx.doi.org/10.1007/3-540-48158-3_2

[16] A. Shet, P. Sadayappan, D. Bernholdt, J. Nieplocha, and V. Tipparaju, "A framework for characterizing overlap of communication and computation in parallel applications," *Cluster Computing*, vol. 11, no. 1, pp. 75–90, 2008.

[17] A. Denis, "MadMPI benchmark," http://pm2.gforge.inria.fr/mpibenchmark/.

[18] R. L. Graham, T. S. Woodall, and J. M. Squyres, "Open MPI: A Flexible High Performance MPI," in *The 6th Annual International Conference on Parallel Processing and Applied Mathematics*, 2005.

[19] M. Pérache, P. Carribault, and H. Jourdren, "Mpc-mpi: An mpi implementation reducing the overall memory consumption," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 16th European PVM/MPI Users Group Meeting (EuroPVM/MPI 2009)*, ser. Lecture Notes in Computer Science, M. Ropo, J. Westerholm, and J. Dongarra, Eds. Springer Berlin Heidelberg, 2009, vol. 5759, pp. 94–103. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03770-2_16

[20] B. Bode, M. Butler, T. Dunning, W. Gropp, T. Hoefler, W.-m. Hwu, and W. Kramer, "The blue waters super-system for super-science," *Contemporary High Performance Computing Architectures*, 2012.

[21] F. Trahay, F. Rue, M. Faverge, Y. Ishikawa, R. Namyst, and J. Dongarra, "EZTrace: a generic framework for performance analysis," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Newport Beach, CA, May 2011.

[22] "EZTrace," http://eztrace.gforge.inria.fr/.