# Benefits of MPI Sessions for GPU MPI applications

Maxim Moraru, Adrien Roussel, Marc Pérache, Hugo Taboada, Christophe Jaillet, Michaël Krajecki

# Benefits of MPI Sessions for GPU MPI applications

Maxim MORARU
maxim.moraru@univ-reims.fr
Université de Reims Champagne Ardennes, LICIIS
51097 Reims, France

Adrien ROUSSEL
adrien.roussel@cea.fr
CEA, DAM, DIF, LRC DIGIT
F-91297 Arpajon, France
Université Paris-Saclay, CEA, Laboratoire en Informatique Haute Performance pour le Calcul et la simulation
Bruyères le Châtel, France

Hugo TABOADA
hugo.taboada@cea.fr
CEA, DAM, DIF
F-91297 Arpajon, France
Université Paris-Saclay, CEA, Laboratoire en Informatique Haute Performance pour le Calcul et la simulation
Bruyères le Châtel, France

Christophe JAILLET
christophe.jaillet@univ-reims.fr
Université de Reims Champagne Ardennes, LICIIS, LRC DIGIT
51097 Reims, France

Marc PÉRACHE
marc.perache@cea.fr
CEA, DAM, DIF, LRC DIGIT
F-91297 Arpajon, France
Université Paris-Saclay, CEA, Laboratoire en Informatique Haute Performance pour le Calcul et la simulation
Bruyères le Châtel, France

Michael KRAJECKI
michael.krajecki@univ-reims.fr
Université de Reims Champagne Ardennes, LICIIS, LRC DIGIT
51097 Reims, France

## ABSTRACT

Heterogeneous supercomputers are now considered the most valuable solution to reach the Exascale. Nowadays, we can frequently observe that compute nodes are composed of more than one GPU accelerator. Programming such architectures efficiently is challenging.

MPI is the defacto standard for distributed computing. CUDA-aware libraries were introduced to ease GPU inter-nodes communications. However, they induce some overhead that can degrade overall performances. MPI 4.0 Specification draft introduces the MPI Sessions model which offers the ability to initialize specific resources for a specific component of the application.

In this paper, we present a way to reduce the overhead induced by CUDA-aware libraries with a solution inspired by MPI Sessions. In this way, we minimize the overhead induced by GPUs in an MPI context and allow to improve CPU + GPU programs efficiency. We evaluate our approach on various micro-benchmarks and some proxy applications like Lulesh, MiniFE, Quicksilver, and Cloverleaf. We demonstrate how this approach can provide up to a 7x speedup compared to the standard MPI model.


*EuroMPI '21, September 08–10, 2021, Garching near Munich, Germany*

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

## CCS CONCEPTS

• **Computer systems organization → Heterogeneous (hybrid) systems**; • **Networks → Network performance analysis**.

## KEYWORDS

HPC, hybrid systems, multi-GPU computing, CUDA-Aware communication libraries, MPI 4.0 Sessions

**ACM Reference Format:**
Maxim MORARU, Adrien ROUSSEL, Hugo TABOADA, Christophe JAILLET, Marc PÉRACHE, and Michael KRAJECKI. 2021. Benefits of MPI Sessions for GPU MPI applications. In *EuroMPI '21: Proceedings of the 28th European MPI Users's Group Meeting, September 08–10, 2021, Garching near Munich, Germany*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

## 1 INTRODUCTION

While the HPC community sets the objective to reach exascale computing by 2020-2022, the performance rise of such systems is tightly bound to their energy consumption. Furthermore, power requirements are very high, necessitating hundreds of megawatts [25]. Energy efficiency is one of the most challenging problems on the road to exascale.

There were important efforts in optimizing energy efficiency by using different leverages: optimize CPU energy by changing clock frequency dynamically (Adagio [31], COUNTDOWN [5], Forestmn [13]); distribute job's power budget across processors efficiently (GEOPM [8], PShifter [12]); optimize other hardware components (cooling system, hard drive, memory); etc.

The current solutions favor the usage of hybrid architectures based on accelerators. Indeed, such architectures are not only able to improve the computing power but also help to reduce the electrical consumption of the nodes. Thus, a contemporary computing

system relies typically on a combination of accelerators and general-purpose CPUs.

The interest in using hybrid architectures increased significantly during the past years. This trend is readily apparent in the TOP500 history: the number of heterogeneous supercomputers in TOP500 passed from 19 in 2011 to 140 in 2019 [3]. One of the most popular accelerators are NVidia GPUs, used in a broad range of industries: computational finance, weather, data science, machine learning, medical imaging life science, energy physics [34] etc. By analyzing the November 2020 TOP500 list we can discover that three supercomputers from the TOP5 are based on NVidia GPUs (Summit, Sierra, Selene). We can also notice that the most energy-efficient system on the Green500 is the new NVidia DGX SuperPOD.

With the increasingly larger problem sizes to solve, there is an obvious need for scalable GPU computing. Oliver Fuhrer et al. [10] leveraged 4888 GPUs to run a climate model on the full Piz Daint supercomputer. Another example of using large-scale GPU computing is distributed deep learning domain [19, 32]. Hiroaki Mikami et al. [26] used 3456 Tesla V100 GPUs in order to train a ResNet-50 model on the ImageNet dataset.

Distributed GPU programming assumes the use of communication libraries, which handle data transfers between compute nodes. GPU accelerators have their own memory and all data transfers must be managed differently from the CPU ones. Hence, CUDA-Aware libraries were introduced to ease GPUs inter-node communication. These libraries are able to handle CPU and GPU buffers simultaneously and offer different optimization in both cases. The drawback of using CUDA-Aware features is that we have to check the buffer locality each time we perform a data movement. This test adds an overhead on the application's critical path and could have an important impact on performances. Results presented in this paper show that the overhead can even achieve up to 700% for some proxy applications.

The main targets of our work are hybrid MPI applications which have a part of their kernels running on the GPUs and the other part on the CPUs. Thus, the communication library must handle CPU to CPU, CPU to GPU, and GPU to GPU memory transfers in the same program. This applies to kernels that are not yet ported to GPUs or that perform badly on these architectures. LLNL Monte Carlo Transport codes [24] is a representative example of such an application. These applications could achieve better performances with a good CPU/GPU load balancing and thus are optimized with an hybrid applications programming model. Another good example is the IO part of scientific computing codes. IO often relies on dedicated libraries like HDF5 [9], netCDF [18]. These libraries are optimized for CPU. The current MPI design does not allow to dissociate CPU and GPU components. Consequently, the need to test the buffer locality adds a significant slowdown on the MPI critical path. Taking into consideration this context, there is an obvious need of removing this unneeded overhead.

This paper aims to solve this problem by making the following contributions:

- Study the overhead's behavior on some MPI micro-benchmarks and proxy applications.
- Provide a portable solution based on the new MPI 4.0 Sessions.
- Propose a method to integrate this solution into the MPI standard.

The rest of this paper is organized as follows. The next section gives an overview about related work and background. Sections 3 and 4 describe our approach and its integration in the MPI standard. In section 5, we present our experimental protocol, evaluate the benefits of using MPI Sessions and discuss about results. The last section presents our conclusions and future work.

## 2 BACKGROUND AND RELATED WORK

In this section we briefly outline the basic concepts of the MPI CUDA-Aware libraries and the new features introduced in the MPI 4.0 Release Candidate. We also discuss about some previous approaches that aimed at optimizing GPU communications.

### 2.1 MPI CUDA-Aware libraries

CUDA-Aware MPI libraries are a set of MPI implementations which can directly handle GPU buffers: MPC [28], MVAPICH2 [27], Open-MPI [11], etc. The user does not need to explicitly stage GPU buffers through host memory, allowing the library to optimize GPU memory transfers. Depending on the buffer size and the hardware topology it is possible to pipeline messages or to use GPUDirect technology. All these optimizations are transparent for the user and ease the development of Hybrid Application. As a balance, CUDA-Aware MPI libraries add an overhead on the critical path as they must handle buffers differently depending on whether they reside in host or device memory.

### 2.2 Unified Virtual Addressing (UVA)

Unified Virtual Addressing (UVA) is a memory address management system, introduced in CUDA 4.0. Figure 1 shows the way UVA is working on a system with multiple GPUs. At runtime initialization, the virtual address range of the application is partitioned into two areas: the CUDA-managed VA range and the OS-managed VA range. All the CUDA pointers are within a certain range which always falls within the first 40 bits of the process VA space. Hence, the location of a buffer can be determined based on the Most Significant Bits (MSBs) of its address. Technically, this operation can be performed by calling the CUDA Driver API function "*cuPointerGetAttribute()*". There is no runtime equivalent as this call sequence would introduce unneeded overhead.

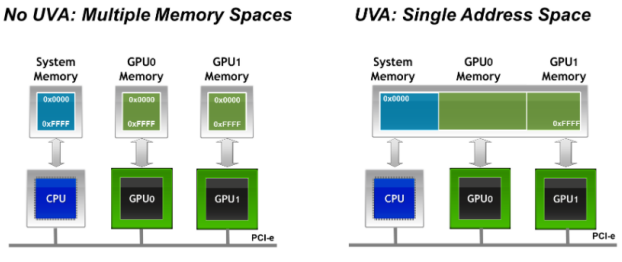


**Figure 1: Unified Virtual Addressing (figure taken from [17])**

### 2.3 MPI 4.0

Recently, the MPI Forum has published a draft version of the MPI 4.0 Standard. This draft is the release candidate for the MPI 4.0 Specification and will be considered for ratification during December 2020 [2]. Among the most significant additions of this new specification, we can find the MPI Sessions, Persistent Collectives, Partitioned Communication and Topology Solutions. All these new concepts have a big potential in optimizing hybrid MPI+GPU applications. In this paper, we focus on the benefits of MPI Sessions.

The Release Candidate introduced MPI Sessions extensions to the MPI standard. This new paradigm is aimed at solving MPI World limitations. Currently, MPI cannot be initialized more than once and cannot be initialized from different application components without a priori knowledge or coordination. The Sessions Model offers an isolated environment that can be initialized and freed multiple times during the same run. Furthermore, it is also possible to allocate and free MPI resources for a specific component of the application. These new functionalities have even bigger advantages for MPI CUDA-Aware libraries, as it is possible to initialize MPI GPU features only for specific libraries or portions of code and consequently avoid the overhead of unnecessary verifications.

The purpose of this article is to bring a solution based on MPI Sessions. Our goal is to minimize the overhead induced by GPUs in an MPI context, and in conclusion to obtain efficient CPU+GPU programs.

### 2.4 MPI Attribute–based Approach

Since the introduction of GPUs as general-purpose accelerators, many papers have studied the ways to optimize the inter-node GPU communication [14, 29, 33]. However, there are only few articles that evaluate the overhead associated with CUDA-Aware communication libraries.

Aji et al. presented an extensible framework for data movement in accelerator-based systems. In [4] they present the consequence of testing buffer locality before each MPI transfer but their work is limited to point-to-point intra-node communications and does not provide an exhaustive analysis of this overhead. Instead, they bring a solution to this problem. They use MPI communicator and datatype attributes to indicate buffer locality to the MPI library. Technically, this operation consists in setting a specific attribute to a datatype or communicator via the *MPI_type_set_attr()/MPI_Comm_set_attr()* function. This attribute indicates the communication pattern and is checked by the MPI library at runtime. Thus, *cuPointerGetAttribute()* can be replaced by a simple call to *MPI_type_get_attr()* or *MPI_Comm_get_attr()*. While this approach allows to considerably improve performance, there is still a small overhead on the MPI critical path. Furthermore, the user has to manually instrument each MPI datatype tag, which can be tedious. Aji et al. also present a limitation of using communicator attributes. This approach does not allow to move data from the GPU on one process to host memory of another process.

Best to our knowledge, there is no previous work on a solution build around MPI Sessions.

## 3 CONTRIBUTION

In this section, we present our contributions. Firstly, we clarify the importance of removing *cuPointerGetAttribute()* from the MPI critical path by evaluating its overhead on several micro-benchmarks and HPC applications. Secondly, we bring a lightweight solution based on the new MPI Sessions concept. Our goal is to highlight the benefits of MPI sessions on CUDA-Aware libraries and GPU hybrid applications.

### 3.1 *cuPointerGetAttribute* overhead analysis

Even if there was a significant effort to improve the performance of *cuPointerGetAttribute()*, this operation adds an overhead on the MPI critical path and can have a significant impact on the overall application performance. In order to evaluate this overhead, we instrumented the well-known Intel MPI Benchmarks [6]: before each MPI routine we test if the buffer resides in CPU or GPU memory. All these tests are unneeded as we always transfer host memory. The purpose of this experimental protocol is to evaluate the overhead of transferring CPU memory with a CUDA-Aware MPI library, depending on the MPI routine, buffer size, and process number. We also consolidate these results on more representative HPC applications thanks to four proxy applications. Performance evaluations will be discussed in section 5.

All these experiments are aimed at evaluating the potential performance improvement of using MPI Sessions.

### 3.2 Sessions based approach

As we could see from the previous sections, the CUDA-Aware communication libraries are able to automatically detect the buffer locality. This operation is done before each memory transfer and thus slowdown the application. To avoid this overhead, we need to provide the required information to the communication library. Hence, the test will no longer be necessary.

This operation can be performed by associating to each MPI Session a specific communication pattern: CPU to CPU, GPU to GPU, CPU to GPU, or GPU to CPU. Therefore, inside a specific Session data would always move to/from GPUs and to/from CPUs. The sessions number depends on the desired communication pattern. For example, it is possible to have a session for CPU - CPU memory transfers and a second session for GPU - GPU communication. Though, by creating four sessions we can cover all communication patterns presented above. This approach eliminates the *cuPointerGetAttribute()* from the critical path.

Our solution is similar to the Communicator-based approach presented in [4]. The difference is that we apply this concept to the MPI Sessions Model, which has some important advantages:

- the overhead can be fully removed,
- there are no limitations for data movement,
- there is no need to introduce new programming habits, as this approach fits perfectly in the new MPI Sessions Model.

## 4 INTEGRATION INTO THE MPI STANDARD

In the new MPI Sessions Model, the *MPI_Init()* function is replaced by *MPI_Session_init()*. The last one takes in parameters a *MPI_Info* object which allows the user to designate the MPI resources to be allocated. The MPI 4.0 Release Candidate describes how this

```
static MPI_Session lib_shandle = MPI_SESSION_NULL;

const char comm_key[] = "mpi_communication_pattern";
const char comm_value[] = "MPI_CPU_TO_GPU";


MPI_Info sinfo = MPI_INFO_NULL;
MPI_Info_create(&sinfo);

MPI_Info_set(sinfo,comm_key,comm_value); /*setting the communication pattern*/

int rc = MPI_Session_init(sinfo,MPI_ERRORS_RETURN,&lib_shandle);
```

**Figure 2: Example of CUDA-Aware MPI implementation initialization**

model can be used to change the application's thread support level. Creating different MPI Session handles allows using different thread support levels for different components of an application.

The MPI 4.0 standard mentioned that *MPI_Info* object can be used as well to specify the MPI implementation-specific resources. Therefore, it is possible to use *MPI_Info* to indicate the communication pattern for a specific Session. Technically, this operation can be performed by adding a new (key, value) pair to the info object. Our approach can be integrated into the new MPI standard by simply adding a new key called *mpi_communication_pattern* allowing us to specify the communication pattern.

Figure 2 presents a simple example inspired from the MPI 4.0 Draft Specification [2], which shows how to use MPI Sessions to optimize GPU communications. In this example, we initialize an MPI Session and associate it to a specific communication pattern. All the communicators derived from this session will perform only CPU to GPU transfers. Thus, all send buffers will reside in host memory and all receive buffers in device memory. Having this information, *cuPointerGetAttribute()* can be completely removed from the critical path.

# 5 APPLICATIONS/BENCHMARKS

In this section, we describe our experimental setup and discuss the obtained results. We split our experimental protocol into two phases: analyze the overhead on some micro-benchmarks and confirm the observed tendency on real HPC applications.

## 5.1 Experimental setup

First of all, we chose to analyze the *cuPointerGetAttribute()* overhead on some MPI micro-benchmarks. One of the most popular candidate for this purpose is the Intel MPI benchmarks suite (IMB) [6] which provides an efficient way to measure the performance of MPI routines. We decided to study some MPI1 functions from the IMB-MPI1 component, as these routines are the most encountered ones in real HPC applications. The main goal of these experiments is to evaluate the impact of *cuPointerGetAttribute()* on the MPI1 routines, according to some important criteria: MPI function type, process number and buffer size.

Secondly, we validate the obtained micro-benchmarking results on representative HPC applications. Our goal is to highlight the impact of using MPI Sessions in a real HPC context.

## 5.2 Experimental context

All the experiments were carried out on the ROMEO petaflop system hosted at Université de Reims Champagne Ardennes, France. ROMEO is a Sequana X1000 cluster with 115 nodes, all equipped with Intel Skylake 6132 processors. Each node has two sockets, each equipped with one processor having 14 cores. A part of the nodes are also equipped with four NVidia P100 GPUs, interconnected via NVlink.

The main characteristic of our cluster is the use of the Bull eXascale Interconnect (BXI) developed by Atos, which is a 100Gb/s proprietary interconnec systemt. The BXI network scales up to 64k nodes and is composed of two separate ASIC components, a Network Interface Controller (NIC) and a switch. The NIC implements in hardware the Portals 4 communication primitives, and thus completely offload the communication from the host processor. [7]. The BXI switch has 48 ports which allows building larger systems with fewer elements. Additionally, fewer hops in the data path reduce the communication latency and limit congestion points.

The versions of the libraries we used for our tests are as followed: CUDA 11.2, OpenMPI 4.0, and BXI 1.7 with a BXI Host Channel Adapter v1.2. As we have a proprietary interconnect, we also have a specialized OpenMPI library that use the native Portals 4 interface.

## 5.3 Micro-Benchmarking Evaluation

We executed the Intel MPI micro-benchmarks with 2, 28, and 56 processes, and with buffer sizes going from 0 bytes to 4 megabytes. Our study is limited to the most common MPI-1 Benchmarks (Alltoall, Allreduce, Allgather, Bcast, Gather, Reduce, Scatter, PingPing, PingPong), which includes the most encountered MPI functions in realistic HPC applications. We also take care to disable turbo boost and bind MPI ranks to cores. In the following sections, we discuss separately collective and peer-to-peer benchmarks results.

*5.3.1 Micro-benchmarks variability.* As described in [15] the MPI micro-benchmarks have a big variability. Indeed, some run-times have an error of approximately 10% and even bigger.

We are aware of this behavior, and we took care to specify the min and the max for all our results. As well, we took it in account while analyzing the obtained graphs.

*5.3.2 MPI collectives.* Figures 3,4 shows our results concerning the MPI collectives. It presents the overhead and the mean logarithmic execution time with the associated bar error. The overhead is described as the ratio between a CUDA-Aware execution and a basic one. A ratio closed to 1 means that the *cuPointerGetAttribute()* overhead is negligible. This appears when the execution time of a benchmark is much more important than the time needed for a data locality test. The time of an MPI function increases with the buffer and world size, while the time of calling *cuPointerGetAttribute()* is constant and does not depend on the transferred buffer size. Therefore, the impact became insignificant from a certain point.

By analyzing Figures 3,4 we can notice that the overhead can be observed only for two MPI ranks executions with relatively small buffers. Moreover, it becomes insignificant for the MPI collectives with 28 and 56 ranks. The big overhead at the beginning of each

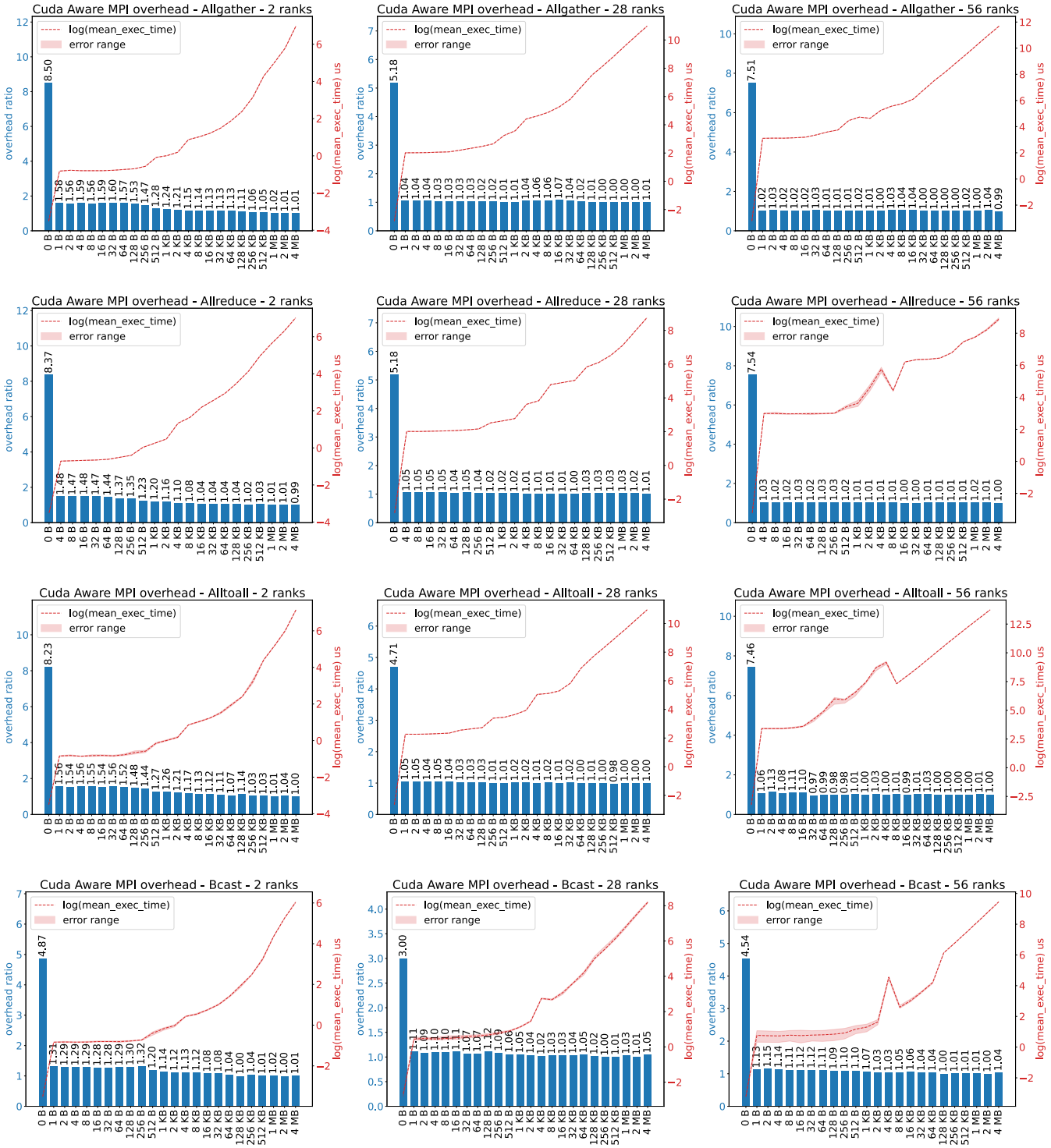


**Figure 3: Overhead induced by the data locality test before each MPI routine.** $overhead\_ratio = \frac{t\_cuPointerGetAttribute}{t\_basic}$, **where *t_cuPointerGetAttribute* is the mean time of one execution with runtime checks (cuPointerGetAttribute) and *t_basic* is the mean time of a basic run (Allgather, Allreduce, Alltoall, Bcast)**

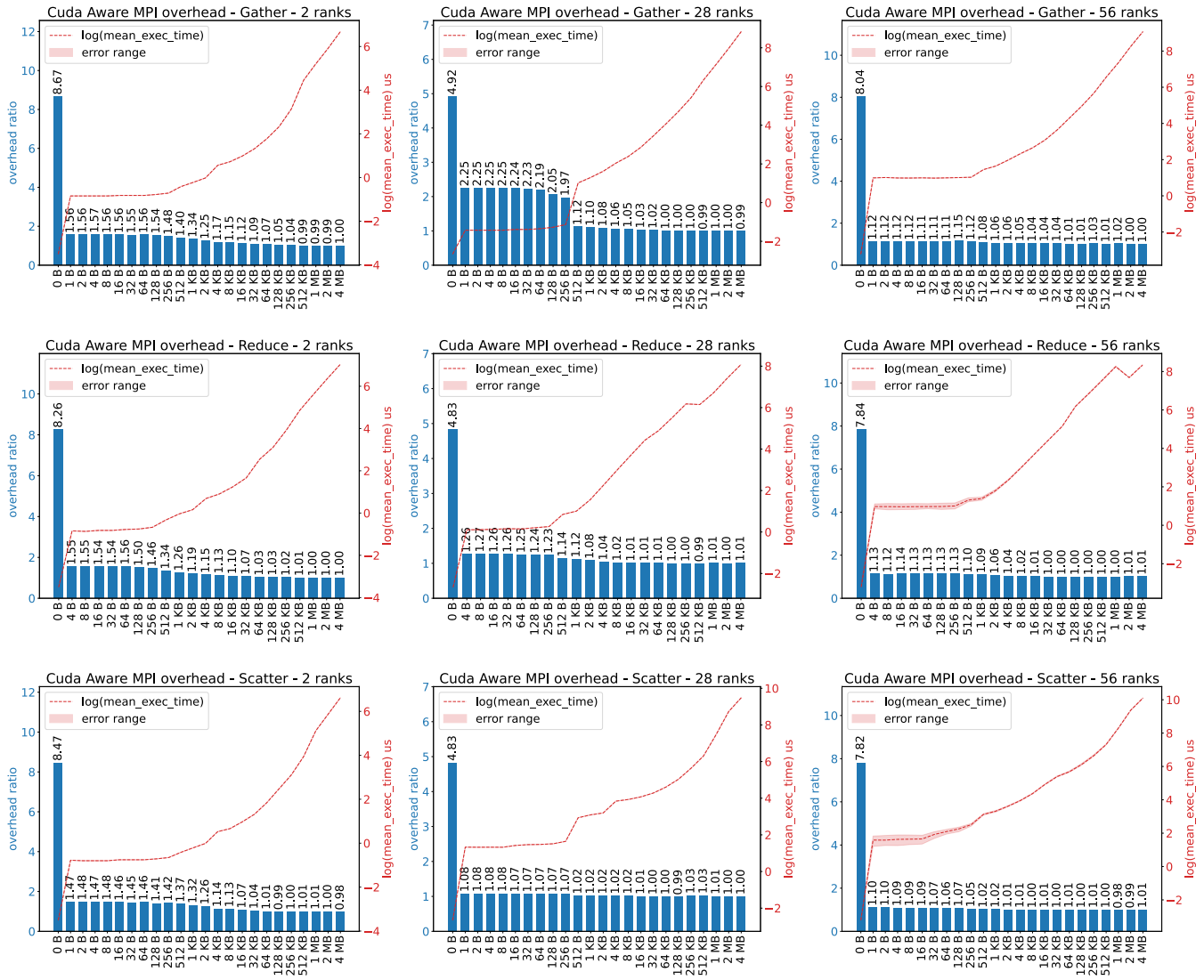


**Figure 4: Overhead induced by the data locality test before each MPI routine.** $overhead\_ratio = \frac{t\_cuPointerGetAttribute}{t\_basic}$**, where *t_cuPointerGetAttribute* is the mean time of one execution with runtime checks (cuPointerGetAttribute) and *t_basic* is the mean time of a basic run (Gather, Reduce, Scatter)**

collective is due to the specific MPI behavior: a collective return immediately if the count parameter is equal to zero.

The collectives with 2 processes were instrumented to better understand the behavior of *cuPointerGetAttribute()*. We consider these results non-relevant for a real HPC context as we target large-scale applications. We can notice an important overhead for Scatter, Gather, Bcast, Reduce even with 28 and 56 ranks. However, we can not consider these results because of their important standard deviation. Thus, we can conclude that in most of the cases *cuPointerGetAttribute()* will not have a significant impact on the collective performance.

*5.3.3 MPI P2P.* Figure 5 shows our point-to-point results. We can notice that the overhead is more important for this communication pattern. However, it becomes negligible for larger buffers, which is the same behavior we observed for collective communications.

Since point-to-point communications are largely used in many HPC applications, this overhead can lead to significant performance degradation.

## 5.4 Applications

The obtained results with MPI micro benchmarks are not sufficient to conclude on the MPI Session Model benefits. In order to verify these performance improvements, we need to study its behavior

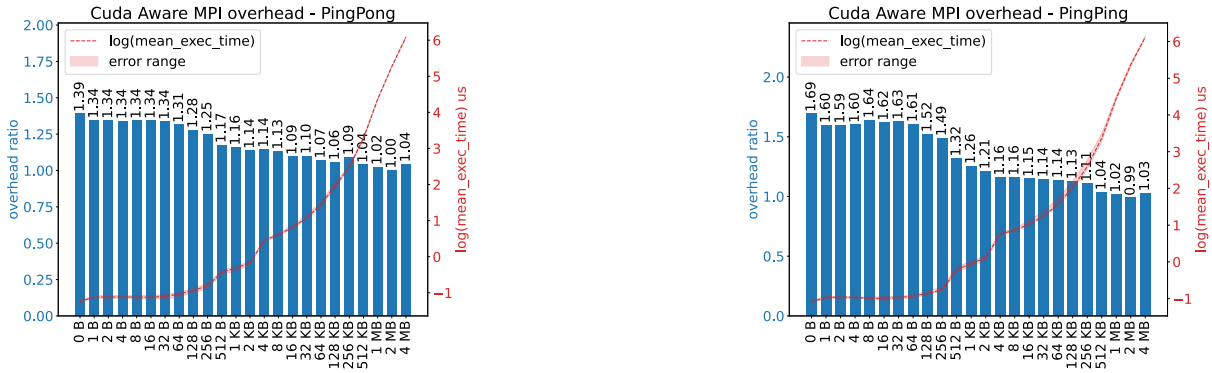


**Figure 5: Overhead induced by the data locality test for P2P MPI routines.** $overhead\_ratio = \frac{t\_cuPointerGetAttribute}{t\_basic}$**, where *t_cuPointerGetAttribute* is the mean time of one execution with runtime checks (cuPointerGetAttribute) and *t_basic* is the mean time of a basic run**

into real MPI applications. We chose four mini-applications for this purpose: LULESH [1], MiniFE [23], Quicksilver [30], CloverLeaf [21]. Mini-apps provide a smaller full featured program which allows to explore performance tuning techniques in less time. Therefore, all the observations are also applicable on full applications. We consider these applications to be the best candidates to highlight the importance of CUDA-Aware overhead. In the previous sections, we saw that the Session advantages are more accentuated for peer-to-peer communications, and the four applications rely on a such communications pattern.

Table 1 presents a brief description of the four applications. We list in this table all the MPI routines used in each application, in order to better understand their communications pattern.

In order to evaluate the Sessions's benefits we developed a `LD_PRELOAD` MPI library which performs a *cuPointerGetAttribute()* test before each MPI routine and for each buffer. We also took care to initialize the CUDA context with *MPI_init()*. We then executed several times each application with and without this library. Thus, it was possible to evaluate the overhead associated exclusively to buffer checks. In fact, MPI CUDA-Aware libraries can also present an overhead associated to the registry cache and other implementation specific resources. At the end we compare the total execution time of the two runs, which we present as the ratio between a run with `LD_PELOAD` and a basic one: $ratio = \frac{t\_cuPointerGetAttribute}{t\_basic}$. All tests was done on CPU versions of these applications.

As for the micro-benchmarks we disabled the turbo boost and bound MPI ranks to cores. The standard deviation for all applications is between 1% and 2%, meaning that our results are fairly reliable.

*5.4.1 LULESH.* LULESH was initially designed as one of five challenge problems for the DARPA UHPC program. It has since evolved and become a representative mini-app of a simplified 3D Lagrangian hydrodynamics on a unstructured mesh.

Our evaluation was performed on the version 2 of LULESH [16] which adds multiple regions and variable cost functions. We decided to test our approach with 27 ranks, with no load balancing and for the flowing problem size: 10, 20, 30, 40, 50.

Figure 6 presents our results. The number above each pair of bars represent the overhead ratio. We can observe an important overhead for the two smallest problem sizes, 10 and 20. This is due to a short execution time compared to the number of MPI routine calls. For the larger problem sizes the overhead is totally hidden.

In order to understand this behavior we profiled LULESH. We computed the number of times an MPI routine is called in a period of time. Thus we can compute the number of buffer checks (calls to *cuPointerGetAttribute*). We consider this metric more relevant than the total number of MPI calls as it can be applied on both proxy and real HPC applications. In fact, the overhead can be insignificant for applications with a big number of MPI calls but longer execution time.

Figure 7 describes the LULESH MPI call intensity, which corresponds to the checks number per second. We can notify a correlation between the MPI call intensity and the associated slowdown. Therefore, the overhead is weakly observable when the intensity decrease.

The MPI calls intensity must be taken as a rough metric. We saw in the previous sections that the CUDA-Aware overhead depends on several factors: process number, buffer size, communication pattern. Thus, we can consider this metric only in a particular context: same communications pattern, buffer and process number.

*5.4.2 MiniFE.* MiniFE is a proxy applications for unstructured implicit finite element codes, which comes from the Mantevo project of Sandia National Laboratories. MiniFE executes the whole finite elements phases: generation, assembly and analysis. The physical domain is a 3-D box and it is possible to define each (*x,y,z*) dimension size.

We decided to fix the $z$ dimension to 10 for all the runs. We also set the $x$ dimension equals to the $y$ one, which took the following values: 1000, 2000, 3000, 4000, 5000. This pattern was inspired from the MniFE github [23].

Figure 8 shows MiniFE results. We can notice a significant overhead even for longer runs. By analyzing Figure 9 it is possible to discover the same correlation with MPI calls intensity, as in the case

| Proxy Application | Description | Collectives | P2P |
|---|---|---|---|
| LULESH2.0 | hydrodynamics on an unstructured mesh | Allreduce, Reduce | Isend, Irecv |
| MiniFE | unstructured implicit finite element | Allreduce, Allgather, Bcast, Reduce | Send, Irecv |
| Quicksilver | simplified dynamic monte carlo particle transport problem | Allreduce, Bcast, Gather, Reduce | Isend, Irecv |
| CloverLeaf | cartesian grid method for the compressible euler equations | Allreduce, Reduce | Isend, Irecv |

**Table 1: Brief characterization of proxy applications**

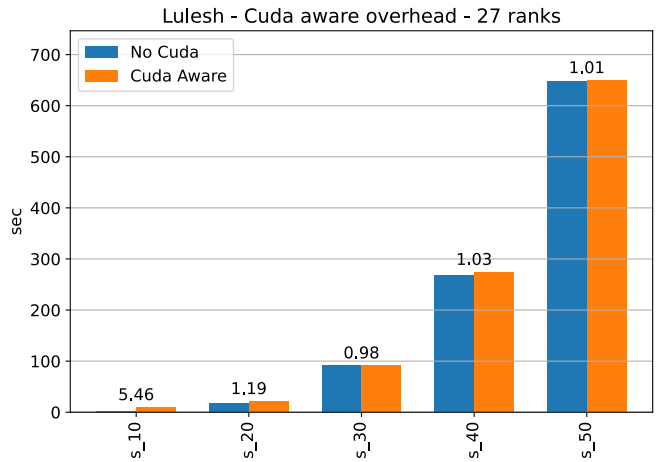


**Figure 6: Lulesh overhead analysis**

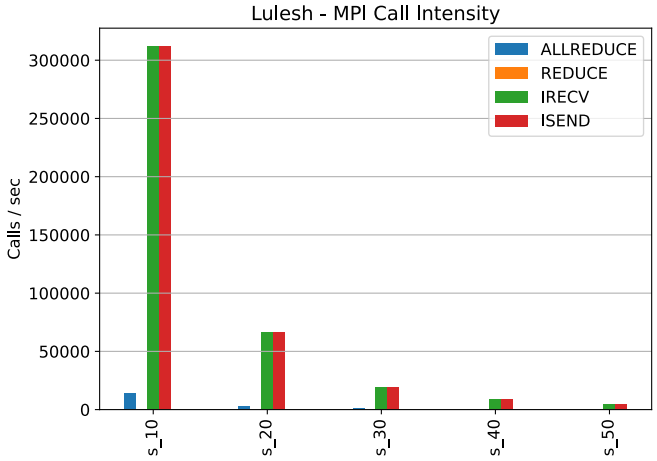


**Figure 7: LULESH - the number of times an MPI Routine is called during a period of time**

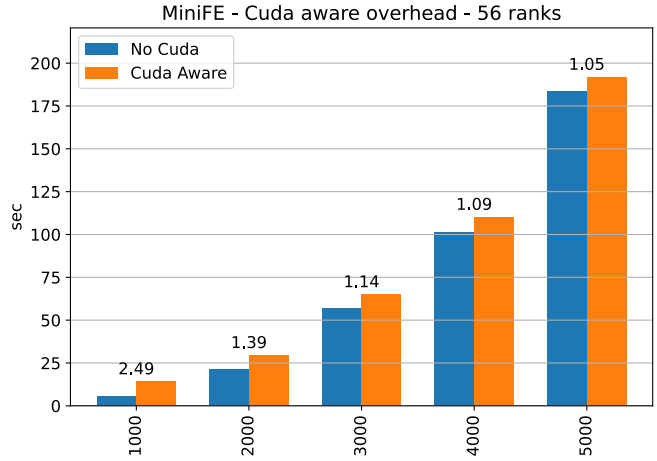


**Figure 8: MiniFE overhead analysis**

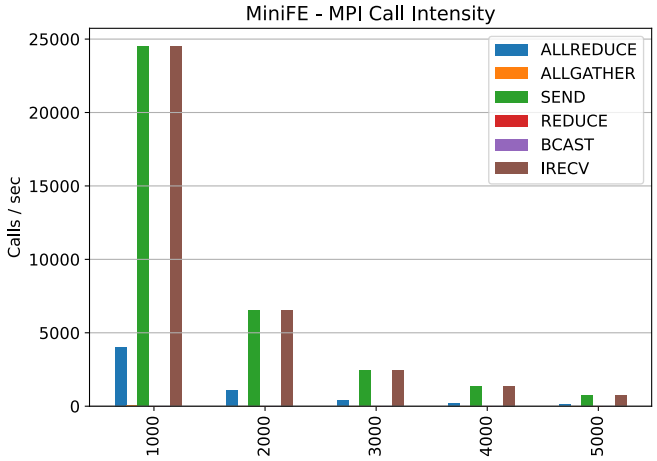


**Figure 9: MiniFE - the number of times an MPI Routine is called during a period of time**

of LULESH. However, comparing to MiniFE the LULESH's MPI intensity is much more important for less overhead. This is because of a different communication pattern, as we mentioned in section 5.4.1. Estimated the overhead based on the MPI communications is quite tedious and can be subject of an entire article.

*5.4.3 Quicksilver.* Quicksilver is a proxy application that solves a simplified dynamic Monte Carlo particle transport problem. It is used as a prototype to test potential programming models and design options for Mercury [20].

Quicksilver can be instrumented by command line options and an input file. The Quicksilver git [23] provides several examples

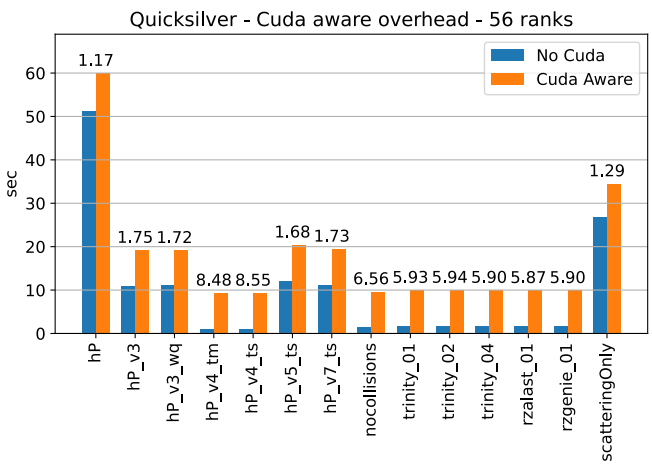


**Figure 10: Quicksilver overhead analysis**

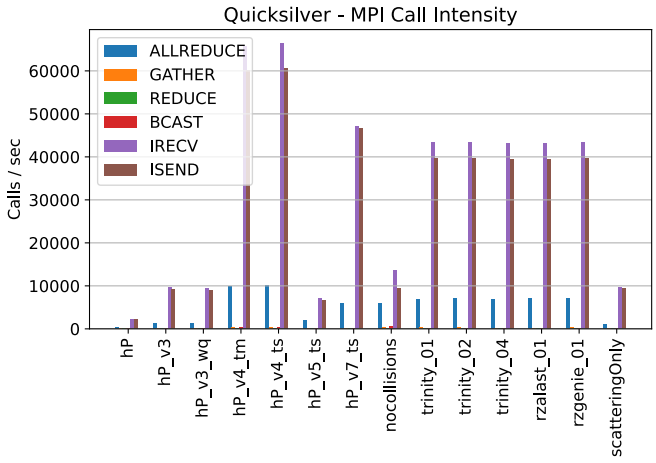


**Figure 11: Quicksilver - the number of times an MPI Routine is called during a period of time**

of input files. Thus, we choose to evaluate our approach on these input files. The full list used for our experimental protocol is the following:

- homogeneousProblem.inp (hP)
- homogeneousProblem_v3.inp (hP_v3)
- homogeneousProblem_v3_wq.inp (hP_v3_wq)
- homogeneousProblem_v4_tm.inp (hP_v4_tm)
- homogeneousProblem_v4_ts.inp (hP_v4_ts)
- homogeneousProblem_v5_ts.inp (hP_v5_ts)
- no.collisions.inp (nocollisons)
- quicksilver_aprun_trinity_01.sh (trinity_01)
- quicksilver_aprun_trinity_02.sh (trinity_02)
- quicksilver_aprun_trinity_04.sh (trinity_04)
- quicksilver_slurm_rzalast_01.sh (rzalast_01)
- quicksilver_slurm_rzgenie_01.sh (rzgenie_01)
- scatteringOnly.inp (scatteringOnly)

Figure 10 shows Quicksilver results. There is a significant overhead for all test cases that we instrumented. These results demonstrate the real need of using MPI Sessions.

Figure 11 shows the MPI calls intensity depending on different routines. We can highlight the same trend as in previous cases. However, the Quicksilver's communications pattern is strongly dependent on the input file. Therefore this metric can not be appropriate for these test cases (see "nocollisons" for a counter example).

*5.4.4 CloverLeaf.* CloverLeaf is a mini-application that solves the compressible Euler equations of compressible fluid dynamics in two spatial dimension. As the previous proxy applications, CloverLeaf is often used as a representative benchmark for a real HPC context.

This application can be instrumented via an input file. We evaluate the CUDA-Aware overhead on four input files which are recommended on the CloverLeaf's github [22].

Figure 12 depict CloverLeaf results. As for LULESH we have a pronounced overhead for smaller runs.

By analyzing Figure 13 we observe an incompatibility between the MPI intensity and the associated overhead. This is can be observed by comparing *bm* with *bm_short* In this particular case, this disproportion is because of the buffer sizes. Indeed, *bm_short* performs memory transfers with smaller buffers.

## 6 CONCLUSIONS AND FUTURE WORK

In this paper we addressed performance issues in hybrid CPU/GPU applications in the MPI context. We diagnose that the detection of buffer locality can induce a large overhead in the MPI critical path. This overhead (up to 700%) is visible on micro-benchmarks and representative HPC applications. In order to remove this overhead, we introduce a new methodology based on the upcoming MPI 4.0 Sessions to distinguish the part of code or the libraries that require CUDA-aware MPI from the part of code or the libraries that are using CPU-only buffers. This method also allows dealing with hybrid messages (CPU to GPU or GPU to CPU).

We have implemented a library prototype to evaluate the gain we can expect on micro-benchmarks and representative applications thanks to our methodology. It allows to gain up to 88% on representative applications and on micro-benchmarks. Thanks to our evaluations on micro-benchmarks we were able to characterize the configuration of MPI function (type of communication, communicator size and buffer size) that are the most impacted by the CUDA-aware support and thus require to be optimized. We extend our evaluation with representative application thanks to proto-applications. We analyzed four types of applications and illustrate the benefit of our methodology on each of them.

Our approach is fully portable and does not depend on the network technology. In fact, it allows transferring useful information to the communication library, which can be used to improve hardware specific MPI libraries. As long as the NIC and the GPU are connected through a PCI Express, the communication library need to ensure that the GPU's memory is visible on the PCIe BARs before each GPU

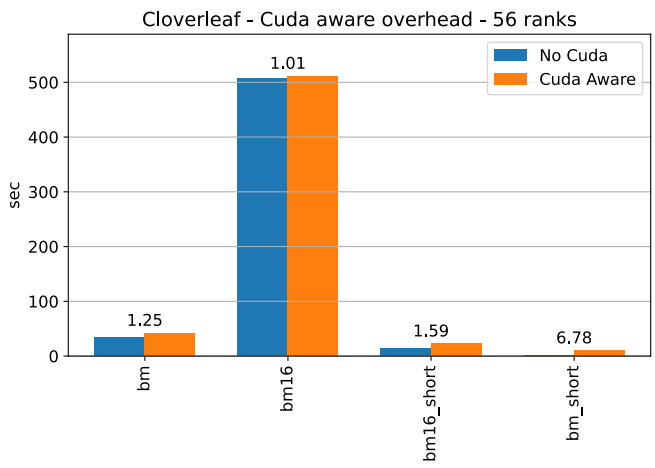


**Figure 12: CloverLeaf overhead analysis**

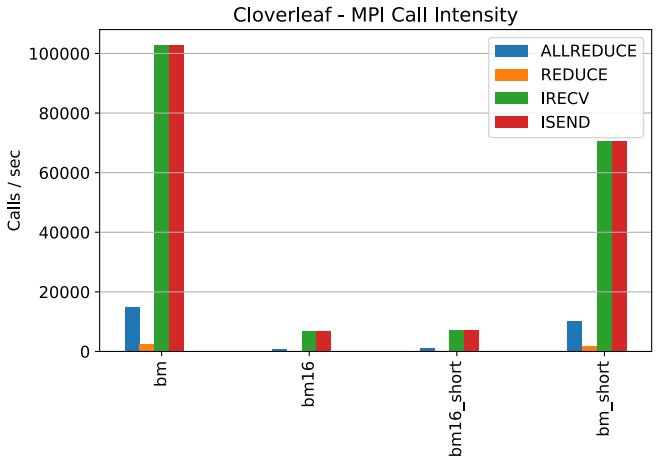


**Figure 13: Cloverleaf - the number of times an MPI Routine is called during a period o time**

transfer. Our model showed an important performance improvements for NVidia GPUs, but it can be also used for optimizing AMD and the new Intel GPUs. Indeed, AMD's Heterogeneous-compute Interface for Portability (HIP) has the same Unified Virtual Addressing environment as CUDA. Furthermore, the new Intel GPUs are also based on the PCIe technology, meaning that the communication library has to differentiate CPU from GPU buffers.

Even if we did not implement the methodology presented in real MPI implementations, it can be easily done as it is similar to the support of thread-multiple in MPI implementation thanks to Sessions.

In order to improve the performances of hybrid applications, this work should be extended to analyze other MPI 4.0 new functionalities such as persistent collectives or partitioned communications. In the context of persistent collectives, as the communication buffers are kept between MPI calls, we can optimize the MPI implementation to avoid useless *cuPointerGetAttribute* function calls. Regarding partitioned communications, we expect to have similar behavior to regular MPI function calls. Eventually, an extensive analysis of data locality of `MPI_GUIDED` features has to be done to use this new feature to add GPU support in the communicator decomposition and consider GPU as a real independent compute resource; just like a CPU is.

## ACKNOWLEDGMENTS

Present results have been obtained within the frame of DIGIT, Contractual Research Laboratory between CEA, CEA/DAM-Île de France center and the URCA.

We would like to thank Romeo HPC Center for supplying all necessary hardware and software resources for our experiments. We must also thank Julien JAEGER for helping us to identify the most important MPI 4.0 features for optimizing GPU applications. We would also like to extend our gratitude to the ATOS BXI team who provided us with important support in developing a CUDA-MPI library on BXI interconnect.

## REFERENCES

[1] [n.d.]. *Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory.* Technical Report LLNL-TR-490254. 1–17 pages.

[2] [n.d.]. mpiForumSite. https://www.mpi-forum.org/. Accessed: 2021.

[3] 2020. An Analysis of System Balance and Architectural Trends Based on Top500 Supercomputers. (8 2020). https://doi.org/10.2172/1649132

[4] Ashwin M. Aji, Pavan Balaji, James Dinan, Wu-chun Feng, and Rajeev Thakur. 2013. Synchronization and Ordering Semantics in Hybrid MPI+GPU Programming. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*. 1020–1029. https://doi.org/10.1109/IPDPSW.2013.256

[5] Daniele Cesarini, Andrea Bartolini, Piero Bonfà, Carlo Cavazzoni, and Luca Benini. 2018. COUNTDOWN: A Run-Time Library for Application-Agnostic Energy Saving in MPI Communication Primitives *(ANDARE '18)*. Association for Computing Machinery, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3295816.3295818

[6] Intel Corporation. [n.d.]. Intel MPI Benchmarks. https://software.intel.com/content/www/us/en/develop/documentation/imb-user-guide/top.html. Accessed: 2021.

[7] S. Derradji, T. Palfer-Sollier, J. Panziera, A. Poudes, and F. W. Atos. 2015. The BXI Interconnect Architecture. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. 18–25. https://doi.org/10.1109/HOTI.2015.15

[8] Jonathan Eastep, Steve Sylvester, Christopher Cantalupo, Brad Geltz, Federico Ardanaz, Asma Al-Rawi, Kelly Livingston, Fuat Keceli, Matthias Maiterth, and Siddhartha Jana. 2017. Global Extensible Open Power Manager: A Vehicle for HPC Community Collaboration on Co-Designed Energy Management Solutions. In *High Performance Computing*, Julian M. Kunkel, Rio Yokota, Pavan Balaji, and David Keyes (Eds.). Springer International Publishing, Cham, 394–412.

[9] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. 2011. An Overview of the HDF5 Technology Suite and Its Applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases* (Uppsala, Sweden) *(AD '11)*. Association for Computing Machinery, New York, NY, USA, 36–47. https://doi.org/10.1145/1966895.1966900

[10] O. Fuhrer, T. Chadha, T. Hoefler, G. Kwasniewski, X. Lapillonne, D. Leutwyler, D. Lüthi, C. Osuna, C. Schär, T. C. Schulthess, and H. Vogt. 2018. Near-global climate simulation at 1 km resolution: establishing a performance baseline on 4888 GPUs with COSMO 5.0. *Geoscientific Model Development* 11, 4 (2018), 1665–1681. https://doi.org/10.5194/gmd-11-1665-2018

[11] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. 2004. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, 97–104.

[12] Neha Gholkar, Frank Mueller, Barry Rountree, and Aniruddha Marathe. 2018. PShifter: Feedback-Based Dynamic Power Shifting within HPC Jobs for Performance. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing* (Tempe, Arizona) *(HPDC '18)*. Association for Computing Machinery, New York, NY, USA, 106–117. https://doi.org/10.1145/

3208040.3208047
[13] J. Halimi, B. Pradelle, A. Guermouche, and W. Jalby. 2014. FoREST-mn: Runtime DVFS beyond communication slack. In *International Green Computing Conference*. 1–6. https://doi.org/10.1109/IGCC.2014.7039158
[14] K. Hamidouche, A. Venkatesh, A. A. Awan, H. Subramoni, C. Chu, and D. K. Panda. 2015. Exploiting GPUDirect RDMA in Designing High Performance OpenSHMEM for NVIDIA GPU Clusters. In *2015 IEEE International Conference on Cluster Computing*. 78–87. https://doi.org/10.1109/CLUSTER.2015.21
[15] Sascha Hunold and Alexandra Carpen-Amarie. 2015. MPI Benchmarking Revisited: Experimental Design and Reproducibility. *CoRR* abs/1505.07734 (2015). arXiv:1505.07734 http://arxiv.org/abs/1505.07734
[16] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes*. Technical Report LLNL-TR-641973. 1–9 pages.
[17] Jiri Kraus. [n.d.]. Nvidia Developer Blog. https://developer.nvidia.com/blog/introduction-cuda-aware-mpi. Accessed: 2021.
[18] Jianwei Li, Wei keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. 2003. Parallel netCDF: A High-Performance Scientific I/O Interface. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. 39–39. https://doi.org/10.1109/SC.2003.10053
[19] Shijian Li, Robert J. Walls, and Tian Guo. 2020. Characterizing and Modeling Distributed Training with Transient Cloud GPU Servers. arXiv:2004.03072 [cs.DC]
[20] LLNL. [n.d.]. Mercury web site. https://wci.llnl.gov/simulation/computer-codes/mercury. Accessed: 2021.
[21] Andy Mallinson, David Beckingsale, Wayne Gaudin, Andy Herdman, John Levesque, and Stephen Jarvis. 2013. CloverLeaf: Preparing Hydrodynamics Codes for Exascale.
[22] Mantevo. [n.d.]. CloverLeaf github. https://github.com/UK-MAC/CloverLeaf. Accessed: 2021.
[23] Mantevo. [n.d.]. miniFE Finite Element Mini-Application. https://github.com/Mantevo/miniFE. Accessed: 2021.
[24] M S McKinley, R Bliele, P S Brantley, S Dawson, M O'Brien, M Pozulp, and D Richards. 2019. Status of LLNL Monte Carlo Transport Codes on Sierra GPUs. (4 2019). https://www.osti.gov/biblio/1559415
[25] P. Messina. 2017. The Exascale Computing Project. *Computing in Science Engineering* 19, 3 (2017), 63–67. https://doi.org/10.1109/MCSE.2017.57
[26] Hiroaki Mikami, Hisahiro Suganuma, Pongsakorn U.-Chupala, Yoshiki Tanaka, and Yuichi Kageyama. 2018. ImageNet/ResNet-50 Training in 224 Seconds. *CoRR* abs/1811.05233 (2018). arXiv:1811.05233 http://arxiv.org/abs/1811.05233
[27] D. Panda, K. Tomko, K. Schulz, and A. Majumdar. 2013. The MVAPICH Project: Evolution and Sustainability of an Open Source Production Quality MPI Library for HPC.
[28] Marc Pérache, Hervé Jourdren, and Raymond Namyst. 2008. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In *the 14th International Euro-Par Conference (LNCS, Vol. 5168)*, Springer (Ed.). Las Palmas de Gran Canaria, Spain, 78–88. https://doi.org/10.1007/978-3-540-85451-7_9
[29] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda. 2013. Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs. In *2013 42nd International Conference on Parallel Processing*. 80–89. https://doi.org/10.1109/ICPP.2013.17
[30] D. F. Richards, R. C. Bleile, P. S. Brantley, S. A. Dawson, M. S. McKinley, and M. J. O'Brien. 2017. Quicksilver: A Proxy App for the Monte Carlo Transport Code Mercury. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 866–873. https://doi.org/10.1109/CLUSTER.2017.121
[31] Barry Rountree, David K. Lowenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. 2009. Adagio: Making DVS Practical for Complex HPC Applications *(ICS '09)*. Association for Computing Machinery, New York, NY, USA, 460–469. https://doi.org/10.1145/1542275.1542340
[32] Siddharth Samsi, Andrew Prout, Michael Jones, Andrew Kirby, Bill Arcand, Bill Bergeron, David Bestor, Chansup Byun, Vijay Gadepally, Michael Houle, Matthew Hubbell, Anna Klein, Peter Michaleas, Lauren Milechin, Julie Mullen, Antonio Rosa, Charles Yee, Albert Reuther, and Jeremy Kepner. 2020. Benchmarking network fabrics for data distributed training of deep neural networks. (08 2020).
[33] Rong Shi, Xiaoyi Lu, Sreeram Potluri, Khaled Hamidouche, Jie Zhang, and Dhabaleswar K. Panda. 2014. HAND: A Hybrid Approach to Accelerate Non-contiguous Data Movement Using MPI Datatypes on GPU Clusters. In *2014 43rd International Conference on Parallel Processing*. 221–230. https://doi.org/10.1109/ICPP.2014.31
[34] D. vom Bruch. 2020. Real-time data processing with GPUs in high energy physics. *Journal of Instrumentation* 15, 06 (jun 2020), C06010–C06010. https://doi.org/10.1088/1748-0221/15/06/c06010