# Trade-offs in Context Identifier Allocation in MPI

George Bosilca
bosilca@icl.utk.edu

Thomas Herault
herault@icl.utk.edu

Jack Dongarra
dongarra@eecs.utk.edu

Innovative Computing Laboratory
The University of Tennessee
Knoxville, TN, USA

## ABSTRACT

The notion of communicators is one of the most central concepts in the Message Passing Interface (MPI), allowing the library developers to contextualize their message exchanges, and scope different algorithms to well-defined groups of processes, and allowing the MPI implementation to specialize its matching and deliver messages in the right context. Due to its collective nature, the communicator identifier allocation is directly linked with the performance of the collective communication implementation on the execution platform. We propose a scalable algorithm for communicator identifier allocation, that minimize the amount of data involved in the operation. This allows for faster context allocation implementations, that are capable of taking advantage of the hardware accelerated collective where available. Additionally, we present 3 communicator storage strategies, and their implementations in the context of Open MPI. We explore the performance of these new algorithms, compare them with algorithms available in other MPI implementations, and discuss their trade-offs.

## 1. INTRODUCTION

Central to the Message Passing Interface (MPI), communicators *bring together multiple concepts* to provide the appropriate scope for all communication operations in MPI. The concepts behind the notion of communicators hold 1) a *context of communication* providing separate safe "universes" of message passing; 2) *groups*, an ordered collection of processes using the corresponding index as a process rank; 3) *virtual topologies* creating a special mapping between the ranks in a group and a topology; and 4) *attributes*, local information added to a communication context for later reference. Communicators allow for the partitioning of the communication space (operations executed on a given communicator do not interfere with operations executed on another communicator), while a group holds an ordered set of process identifiers (by extension, we speak of processes belonging to a communicator).

As such, there are multiple reasons why MPI programmers take advantage of the benefits of communicators: library composition, topology association, neighborhood definition, as well as dynamic process management. Communicators are key to the composition of libraries that use MPI routines, thanks to their context property: because a communication operation posted on a communicator cannot interfere with any other communication operation posted on a different communicator, portable libraries create their own set of communicators that they use exclusively (and often avoid exposing to other levels of the software stack) for their internal communications.

Many applications need a different rank notation mechanism than the default linear group rank, one related to the logical communication pattern of the algorithm. Generally speaking, processes are usually either viewed as a multidimensional grid of processes, or as a potentially weighted graph. The topology notion provided by MPI does not reflect the underlying network topology – which is, and will remain, unknown to the application – but rather affects the communication pattern imposed by the application. The existence of the topologies allows, as an example, for a mapping between the application defined communication topology and the underlying network topology with the goal of improving the expected communication performance.

Another side effect of the topologies is the neighborhood communication, which is another reason why the creation of communicators became more mainstream with the introduction of the neighborhood collective. The topology will define a communication pattern between neighbor processes, and the collective algorithms will perform the collective communication pattern along the edges of the described topology, leading to a sparse collective communication pattern. By removing processes from an existing communicator, the programmer can restrict the scope of collective operations, and simplify the computation of peers in point to point communications.

A further reason to create communicators involves the dynamic process management: at initialization of the MPI program, the standard defines few, predefined, communicators that must be provided by the MPI implementation. `MPI_COMM_WORLD` is one of these pre-defined communicators, maybe the most important as it holds all the processes that were launched together with the current process by the `mpirun` command. When using the MPI-2 dynamics operations, the

programmer can launch additional processes, or connect different parallel applications. These operations create new inter-communicators that can be used to enable communications between the different applications.

Recently, several fault tolerance solutions have been proposed and investigated by the Fault-Tolerance Working Group in the context of the MPI Forum [3]. These approaches try to break the traditional checkpoint/restart barrier, where applications have to take checkpoints at regular intervals and where process fault translate automatically to the application being completely removed from the execution platforms and rescheduled from the last checkpoint at a later date. Instead, the proposed solution will help the application developers by providing some level of support, where the application can remain in place despite process faults, and, either the developer or 3rd party tools will spawn the lost processes and recreate the original execution context. Whatever the solution, when it comes to process-level fault tolerance, recreating communicators upon process failure has become a critical recurrent operation. Looking more specifically at the User-Level Failure Mitigation (ULFM) proposal, if a rebuild of the communicator is required, the application developer has to first shrink a communicator with faulty processes to get a sane communicator then to merge newly spawned processes into a inter-communicator, and finally to merge the inter-communicator into an intra-communicator. As a result, multiple temporary communicators will be created, highlighting the necessity of an efficient, and space-constrained communicator creation support.

These new uses of communicators creation places the concept in a new light. Indeed, in the traditional usage of communicators, communicator creation usually seldom occurred, mostly during the initialization of the application or of the different libraries used by the application, or during important reconfigurations (*e.g.*, when connecting different applications together, or launching new processes, or drastically changing the communication pattern). Moreover, when used in a fault-tolerant application, communicator creation is an operation that must be done almost every time a failure hits the application and a reconstruction of all the objects that were hit by such failure is necessary. Since many communicators may hold the same failed process, many of them must be rebuilt.

Building a communicator is not an inexpensive operation, and depending on the interface used, it may be necessary to communicate in order to gather the group of processes that belong to it; once that group is decided, processes belonging to the communicator must agree on a common name for it. That name is the Communicator Identifier (CID).

We present the challenge of creating a uniquely defined communicator identifier, and depict the existing algorithms used by two widely available MPI implementations, Open MPI and MVAPICH2. Building upon these algorithms, we introduce a new algorithm that uses a highest integer heuristic to select the next available communicator identifier, and evaluate different storage methods allowing for fast communicator retrieval. Compared with the existing algorithms, we show that our approach presents the advantage of succeeding in a smaller number of steps to allocate the unique communicator identifier, while preserving good density on the storage.

This paper is organized as follows. In Section 2 we present the current functions proposed by the MPI standard to create communicators. In Section 3 we present the challenge of creating communicators that are uniquely identifiable by all the participating processes, and present some of the existing algorithms to create them, as well as introduce the algorithms we propose. In Section 4 we present an empirical evaluation of the overhead and performance impact of all the algorithms described in the Section 3. Section 5 describe prior research into this topic and then finally in Section 6 we summarize our findings and conclude.

## 2. CREATING COMMUNICATORS

Communicators in MPI exist in two variants: 1) intra-communicator featuring a single group of processes over which the communication routines identify a source and destination process with the same rank in that communicator, and 2) inter-communicator that holds two disjoint groups of processes, the local and the remote groups, over which the communication operations uses the rank of the target process within the target group. Communicator creation routines can create either intra-communicators or inter-communicators.

MPI 3.0 defines a few routines that enable the creation of communicators. Most of those inherited from previous versions were targeted to creating a communicator from a parent communicator by duplicating a communicator (`MPI_COMM_DUP`), or by discarding some the of participants, or by grouping them in several resulting communicators based on information provided either globally (`MPI_COMM_CREATE`) or distributively (`MPI_COMM_SPLIT`). One of the properties of these functions is that they are collective over the group of processes of the parent communicator, and as a result their cost might be significant when large groups are the source, and small groups are the target of the operations. To cover this case, a new API (`MPI_COMM_CREATE_GROUP`) has been added in MPI 3.0, allowing the communicator creation operation to be collective only over the participants of the resulting communicator.

The MPI-2 dynamic process management chapter added the capacity to MPI applications to extend over the static group of processes imposed by the original `MPI_COMM_WORLD` and instead build an elastic universe where processes can be added or removed dynamically during the application execution. These management capabilities integrate with the rest of the MPI concepts, and pass through the creation and manipulation of communicators. These routines allow for spawning a new group of processes with their own `MPI_COMM_WORLD` (`MPI_COMM_SPAWN` and `MPI_COMM_SPAWN_MULTIPLE`), joining two disjoint MPI universes using a sever/client approach (`MPI_COMM_ACCEPT` and `MPI_COMM_CONNECT`), or using a BSD socket (`MPI_COMM_JOIN`). Unlike the communicator creation functions presented previously, all of these APIs create inter-communicators.

In addition to the communicator creation functions already available in the MPI standard, in the context of the ULFM proposal, a new function has been proposed. The scope of this function is to provide a simple mechanism for creating

communicators based on parent communicators with failed processes. Once a process failure occurs, the communicators where the failed process has been part of might become unsuitable for safe communications. When a communicator is in this state, it cannot be used for collective communications, and might become an impediment for the application's successful completion – as without the possibility to execute collective operations no communicator creation functions will succeed. To cover such cases, ULFM proposes a new concept that would allow the creation of a communicator by removing all processes discovered failed from a parent communicator. This collective function, known as `MPIX_COMM_SHRINK`, has a similar behavior to the `MPI_COMM_SPLIT`, and creates a communicator in which the groups have excluded all failed processes. As explained in Section 1, this new addition – together with the expected usage of dynamic process management in the case of fault handling – motivates us to reconsider the implementation of the communicator creation, since – in this context –this operation becomes a critical building block for resilient concepts. Moreover, as the cost of the recovery is now tied to the performance of the communicator creation, a high efficiency communicator creation is necessary to tolerate volatile environments.

## 2.1 Unique Communicator Identifier

Communicator creation is most often implemented as a collective operation because all participating processes must agree on a unique *Communicator IDentifier* (CID). This CID is of extreme importance for a large number of MPI functions, not only from the user perspective but also from the perspective of the implementors of the MPI library. As an example, it will be used in performance critical operations such as message matching. The CID uniquely identifies a communicator for the group of processes that belong to that communicator. Because allocating a new CID is an operation that may involve only a subset of the processes belonging to the application, it is possible that two subsets of processes, $A$ and $B$, of empty intersection ($A \bigvee B = \emptyset$) are involved, in parallel, in the allocation of two new CIDs. Multiple cases are legitimate: $CID_A = CID_B$, where both of them allocate the same CID that becomes unavailable (until the communicator is freed) over the union of $A$ and $B$; $CID_A \neq CID_B$, where the two sets allocate different CIDs, and $CID_A$ cannot be re-used in any communicator with a group that holds any process of $A$, and respectfully, $CID_B$ cannot be re-used in any communicator with a group that holds any process of $B$. If, after these operations, a subset $C$ of processes, such that $C \bigvee A \neq \emptyset$ and $C \bigvee B \neq \emptyset$ tries to allocate a new $CID_C$, the allocation algorithm must guarantee that $CID_C \neq CID_A$, and $CID_C \neq CID_B$. Thus, the CID allocation has a collective meaning over all involved processes.

It is of particular interest that in most MPI implementations the CIDs are typed as integer values, and used as indexes in sparse arrays to associate a given CID with its internal representation of a communicator. Finding such "free" CID among all possible values is the key part of communicators creation. The other part is a local operation that associates a given CID to the local object that represents the communicator for a process, in order to allow it to communicate with other processes belonging to the same communicator. Using a communication identifier allocation, we designate

**Algorithm 1:** Open MPI Vanilla Algorithm

```
start ← lowestAvailableCID()
done ← false
Atomically inProg ← inProg ⋃ {CID(parent)} while
 !done do
    Wait Until min inProg = CID(parent)
    candidate ← start
    while ¬CAS(comms[candidate], nil, parent) do
     candidate++
    MPI_ALLREDUCE( &candidate, &contextID,
                   1, MPI_INT,
                   MPI_MAX, parent)
    done ← (contextID = candidate) ∨
           CAS(comms[contextID], nil, parent)
    MPI_ALLREDUCE( &done, &done,
                   1, MPI_BOOL,
                   MPI_AND, parent)
    if ¬done then
        comms[candidate] ← nil
        if candidate ≠ contextID then
         CAS(comms[contextID], parent, nil)
        start ← contextID + 1
Atomically inProg ← inProg \ {CID(parent)}
```

an algorithm that solves both of these problems, and in this work we propose, evaluate, and compare different communicator identifier allocation algorithms over different criteria: the speed at which the allocation of new CIDs succeeds; the space the algorithm uses to store the associative array between CIDs and local representations of communicators; and the impact of communicator lookup operations over communications.

# 3. COMMUNICATOR ALLOCATION

In this section, we present the different algorithms for the CID allocation that we will evaluate in the rest of the paper.

## 3.1 Open MPI Vanilla Algorithm

The default algorithm of Open MPI is represented in Figure 1. It consists of a loop of trial and confirmations using a simple `MPI_ALLREDUCE` operation. All processes start the CID allocation algorithm by finding the smallest locally unused CID that is proposed as a candidate. To allow for a multithreaded execution, CID candidates are reserved using Compare and Swap (CAS) atomic operations. Moreover, to guarantee progress in that case, and remove potential livelocks, only the thread that entered with the parent communicator having the smallest CID is allowed to iterate over the main loop until it decides upon a new CID. Then a *MAX* operation is computed between all the proposers, and the returned value is compared to the proposed value. If the returned value is free or has been reserved for this CID allocation, the process then participates to a second `MPI_ALLREDUCE` to confirm the success with `true`, otherwise it participates with `false`, and a logical *AND* is computed between the proposed values. If every process in the `parent` communicator participated to the confirmation `MPI_ALLREDUCE` with `true`, then that CID is used, otherwise all reserved CIDs are released, and the algorithm starts another round

**Algorithm 2:** MPICH Algorithm

```
contextID ← 0
lowestCID ← MAXINT
iOwnMask ← false
maskInUse ← false
while contextID = 0 do
    lock()
    if
      CID(parent) < lowestCID∨
      (CID(parent) = lowestCID ∧ myTag < lowestTag)
     then
        lowestCID ← CID(parent)
        lowestTag ← myTag
     maskInUse ←  ¬maskInUse∧
                  CID(parent) = lowestCID∧
                  myTag = lowestTag
    iOwnMask ← maskInUse
    if maskInUse then
        localMask ← copy(mask)
    unlock()
     MPI_ALLREDUCE( &localMask, &localMask,
                    NBCID/8, MPI_BYTE,
                    MPI_BAND, parent)
    if iOwnMask then
        lock()
        if localMask ≠ 0 then
            contextID ← i s.t.  localMask[i] = 1∧
                                ∀j < i, localMask[j] = 0
            mask[contextID] ← 0
            if  lowestCID = CID(parent)∧   then
                lowestTag = myTag
                lowestCID ← MAXINT
        maskInUse ← false
        unlock()
```

with the next locally available CID that is higher than the last failed candidate.

This algorithm is costly in terms of the number of `MPI_ALLREDUCE` operations: in the best case, it will complete in two sequential `MPI_ALLREDUCE` calls of one int and one boolean (respectfully), and these operations cannot be pipelined. It always produces the smallest CID available among all the ranks, in an effort to reduce the memory requirements of the pointers array that Open MPI uses to store and lookup communicators.

Communicator objects associated with the CIDs are stored in a resizable pointer array. Finding the communicator associated with a specific CID requires a single dereference from the array base address. The number of possible communicators is bounded by the maximum size of the pointer array.

## 3.2 MPICH Algorithm

As described in [6], MPICH uses a bounded range for its internal communicator identifiers. The communicator allocation algorithm uses this constraint to exchange information about all local communicator identifiers in a single `MPI_ALLREDUCE` operation. Figure 2 shows the algorithm used by MPICH for multithreaded environments. Instead of reserving specific CID candidates for a given CID allocation, a token is passed between the threads that enter simultaneously in a CID allocation routine. If a thread possesses the token, it proposes all locally available CIDs simultaneously in an `MPI_ALLREDUCE`, using a bit array of locally available CIDs, thus guaranteeing that all participants of the same `MPI_ALLREDUCE` will find a globally available CID if all threads get the same token. If the thread does not possess the token, it participates to the `MPI_ALLREDUCE` (in order to avoid deadlocks), but prevents any CID from being selected by providing 0. Threads get the token following a global ordering defined by the identifier of the parent CID that is common to all callers.

In the best case, if a single thread at a time enters the CID allocation algorithm on each process, all processes obtain the token immediately, and the decision is taken in a single `MPI_ALLREDUCE` operation, with the size of the operation being the number of bits necessary to represent the whole CID space (this space is bounded to a few thousands possible CIDs). Only in the case where there are contentions in a multithreaded run, the algorithm may do multiple `MPI_ALLREDUCE` operations.

Communicator objects associated with the CIDs are stored in a fixed size pointer array, and finding the communicator associated with a specific CID requires a single dereference from the array base address. The number of possible communicators is directly derived from the size of the array, but it has a direct impact on the performance of the CID allocation, as larger arrays will impose costlier `MPI_ALLREDUCE`.

## 3.3 Highest-CID Algorithm

Highest-CID algorithm is the variant proposed in this work. It is a variant on the Open MPI Vanilla Algorithm: instead of focusing the search on the smallest CID available, processes enter the CID allocation algorithm with the highest used CID plus one (see Figure 3). In a single-threaded environment, this guarantees that after the first `MPI_ALLREDUCE` operation using the MAX operand, all processes get a CID that is unused by all of them. In such an environment, the decision can thus be taken in a single `MPI_ALLREDUCE` round. In the case of a multithreaded environment, the confirmation round of the Open MPI Vanilla Algorithm must be kept, as scheduling exists under which the maximal of the unused CIDs proposed by each process might have been claimed by another thread on at least one process during the reduce operation.

One issue with this variant is that it does not guarantee that the smallest CID is used: the size of the communicators array may grow faster with this algorithm than with the previous two algorithms. Thus, we consider sparse data structures to store the associative array between the CID and the communicator object. We considered two sparse storage data structures: a red-black tree, and a simple multidimension dynamic array storage.

### 3.3.1 Red-Black Tree Communicators Storage

In the Red-Black Tree variant, we replace the dynamic associative array that stores the communicators with a red-black

**Algorithm 3:** Highest-CID Algorithm

```
candidate ← highestUsedCID() + 1
done ← false
if MPIThreadMultiple then
  Atomically inProg ← inProg ⋃ {CID(parent)}
while !done do
  if MPIThreadMultiple then
    Wait Until min inProg = CID(parent)
  while ¬CAS(comms[candidate], nil, parent) do
    candidate ← highestUsedCID() + 1
  MPI_ALLREDUCE( &candidate, &contextID,
                 1, MPI_INT,
                 MPI_MAX, parent)
  if ¬MPIThreadMultiple then
    assert(comms[contextID] = nil)
    if contextID ≠ candidate then
      comms[candidate] ← nil
    done ← true
  else
    done ← (contextID = candidate) ∨
           CAS(comms[contextID], nil, parent)
    MPI_ALLREDUCE( &done, &done,
                   1, MPI_BOOL,
                   MPI_AND, parent)
    if ¬done then
      comms[candidate] ← nil
      if candidate ≠ contextID then
        CAS(comms[contextID], parent, nil)
if MPIThreadMultiple then
  Atomically inProg ← inProg \ {CID(parent)}
```

tree [12]. The tree uses a simple read-write-lock mechanism at the scope of the tree to handle atomic insertion and deletion operations, but allows multiple lookup operations to run simultaneously on the tree. Natural order of integers is used to order the keys, that are communicator identifiers.

During insertion and deletion of elements in the tree, rotation operations may be triggered, following the traditional conditions of a red-black tree to keep it reasonably balanced (the distance between the root and the closest leaf is at least half the distance from the root to the farthest leaf).

### 3.3.2 Multidimension Dynamic Array Storage

In the multidimensional array storage, the key (a 4-byte int, as the CID must be an int) is split in four separate bytes $(k_3, k_2, k_1, k_0)$, (from high significant bit to low significant bit), that are used as an index in a 4-level multidimension redimensionable array allocated dynamically (see Figure 1). In order to constrain the amount of memory used to store this multidimensional array, a reference count is maintained on the array at each level, and the array can be freed once its reference count reaches zero.

# 4. EVALUATION

This section describes the metrics that we will use to assess the costs and overheads of the different algorithms and storage methods, and present the performance evaluation.

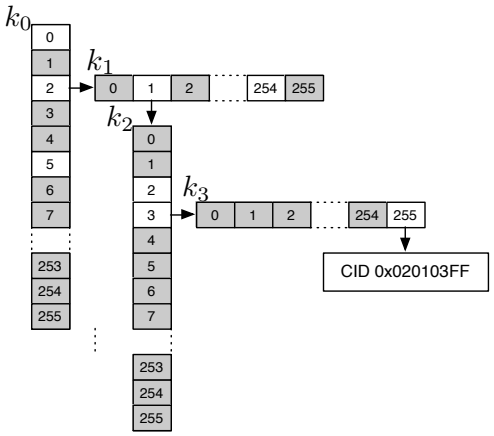


**Figure 1: Multidimension Dynamic Array Example**

## 4.1 Metrics

We are interested in three metrics to validate the different overheads of the proposed algorithms. First, the cost of the CID allocation, then the impact of the storage methods on the retrieval of the local communicator structure based on the CID, and finally the sparsity of the storage methods.

*CID allocation stress benchmark.* We designed a synthetic benchmark to stress the allocation algorithms. The main routine invokes CID allocation through the `MPI_COMM_SPLIT` operation, which is one of the most complex communicator creation procedures, and reflects typical communicator creations in response to a failure in ULFM. For a given number of iterations, a communicator among the existing one is selected according to a *selection* criterion. All processes belonging to that selected communicator participate together in an `MPI_COMM_SPLIT` operation, where the order of ranks is preserved, and the participation is chosen randomly using a *participation* criterion. The benchmark keeps individual times of the split operation for each operation and for each process that participates in one. Some communication time is then measured, both on a default communicator, and on the newly created communicator to evaluate the impact of the data structure used to store the communicators associative array.

In both cases, the selection criterion requires that the selected communicator be at least as large as the target communicator size. We consider two participation criteria: in LargeComm, a random number of ranks (0 to 8) are eliminated from participating; these ranks are selected using uniform random; in SmallComm, the created communicator has a size between 8 and 16.

*Latency.* Another interesting metric describes the impact of the communication retrieval cost on the MPI communications. Here, we focus the study on latency of point-to-point communications, as the expected overhead will be mainly due to communicator lookup in sparse storage data structures. We measure, independently, the latency between two processes on the same node (ranks 0 and 1 of `MPI_COMM_WORLD`) and two processes on different nodes (ranks 0 and

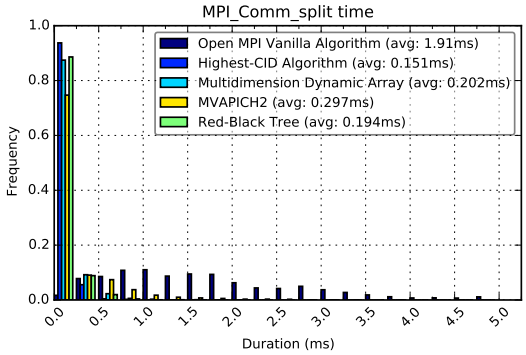


**Figure 2: Repartition of durations for `MPI_COMM_SPLIT` (SmallComm)**

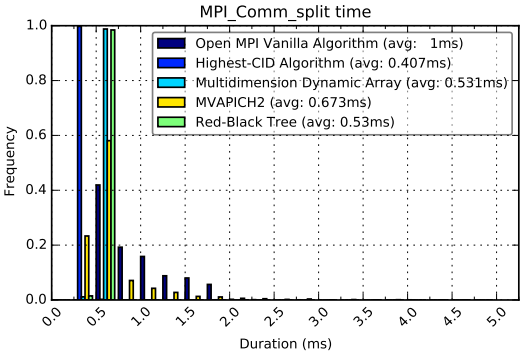


**Figure 3: Repartition of durations for `MPI_COMM_SPLIT` (LargeComm)**

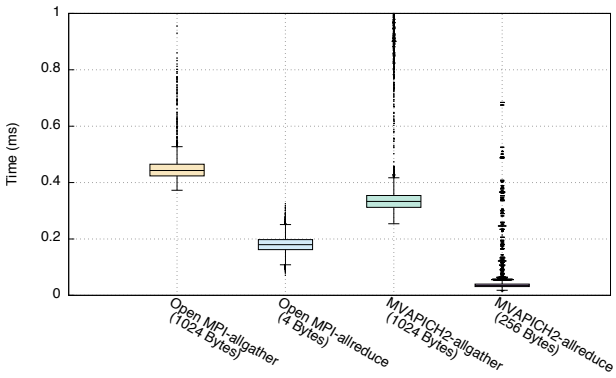


**Figure 4: Duration of specific collective operations for Open MPI and MVAPICH2 on the experimental platform**

$np - 1$ on `MPI_COMM_WORLD`) over the default communicator, and the latency between ranks 0 and $size - 1$ on the newly created communicator (where $size$ is the size of the newly created communicator). That latency averages a thousands ping-pong exchanges between the processes described above.

*Sparsity.* In addition to latency, we also measure the sparsity of the allocated CID in both scenarios. We define the sparsity as the number of empty allocated cells in the data structure divided by the total number of cells that structure can store at a given time. After each communicator identifier allocation, each rank reports the local sparsity, and we consider the average sparsity for every process in `MPI_COMM_WORLD` independently.

## 4.2 Performance Evaluation

All performance evaluations were conducted on a 16-node, 8 core / node, cluster. All nodes are equipped with two 2.27GHz quad-core Intel E5520 CPUs with a 20GB/s Infiniband interconnect. We used the release `r32795` of the Open MPI trunk for the Open MPI Vanilla implementation, and the different proposed algorithms were implemented inside a fork of this version. To take advantage of the Infiniband cards and compare with the MPICH algorithm, we used MVAPICH version `2.0rc1` that implements the CID allocation algorithm described in Section 3.2 (with a limit of 2,048 communicators). When using Open MPI based implementations, the benchmark allocated 10,000 communicators, while we restricted the number of communicators to allocate to the maximum allowed for MVAPICH2. All runs used all 128 cores available, with one MPI process per core, and processes were mapped onto the cores by putting consecutive ranks on a same node until the node was full. All runs of the benchmark start with the same pseudo-random seed, to guarantee that all experiments create the same communicators in the same order. We report the average times, and the time distribution, for each allocation in the following figures.

Figure 4 presents the duration and variability of the `MPI_ALLGATHER` and `MPI_ALLREDUCE` operations over 128 processes at different message sizes for the two implementations we consider. These operations at the given size are the basic building blocks used in the `MPI_COMM_SPLIT` operation, and we will use them to explain some of the performance measurement. The figure shows a box around the first and third quartiles of the measurements, with a horizontal line at the median value. Outliers outside the first and third quartiles are plotted, showing a larger variability of the measurement for MVAPICH2 on this platform.

Figures 2 and 3 present the duration of the `MPI_COMM_SPLIT` operation, in the SmallComm and the LargeComm cases. The figures represent the duration distributions: the x-axis shows intervals of measured durations, and the y-axis shows the frequency of a duration falling in this interval. Figures also report the average duration over the whole set of runs.

Both figures highlight an issue with the Open MPI Vanilla Algorithm: the duration of the `MPI_COMM_SPLIT` operation is distributed over a large interval, more clearly in the case of SmallComm, but still evident in the case of LargeComm. This is due to the conservative algorithm of Open MPI: when the set of available CIDs is sparse and fragmented

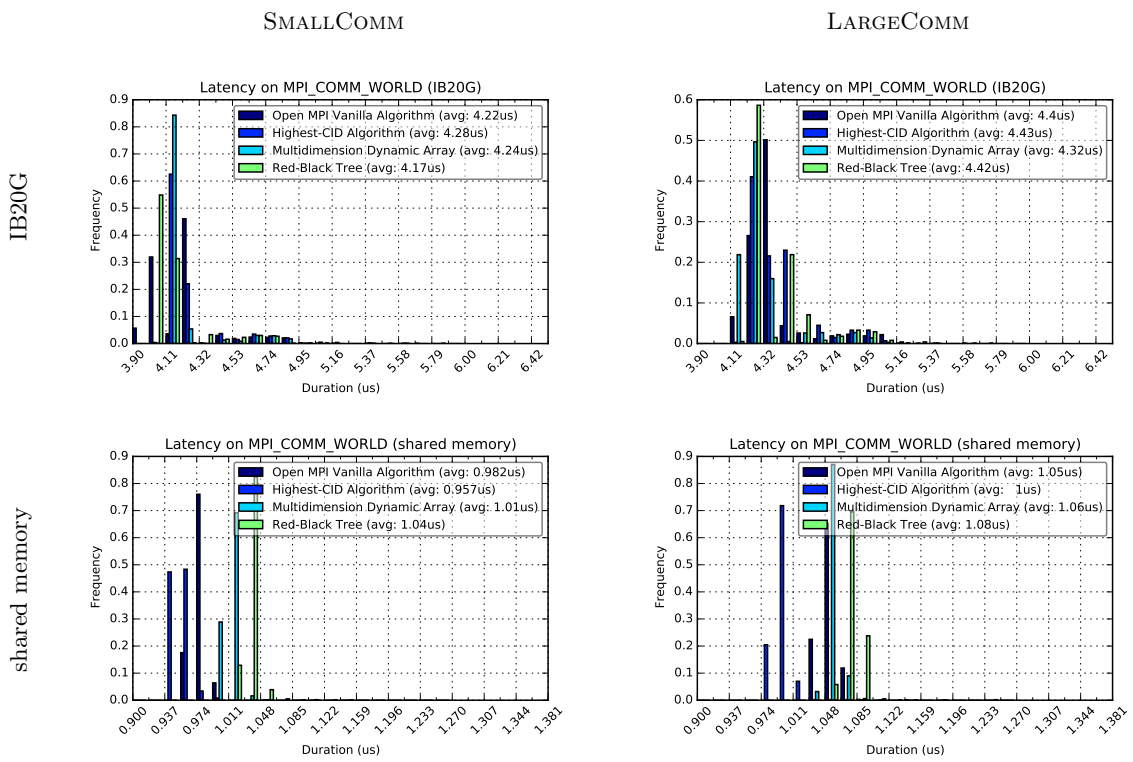


**Figure 5: Latency on `MPI_COMM_WORLD`**

between the processes that participate in the operation, the algorithm starts with a low candidate, and needs multiple phases to converge to a globally available CID. These successive `MPI_ALLREDUCE` operations cannot be pipelined and take significant time to complete. With a higher fragmentation (SmallComm), durations are distributed over a larger interval. With a more symmetrical situation (LargeComm), durations are distributed over a smaller interval.

A similar phenomenon is observable with MVAPICH2, to a lesser extent: most of the durations (70% to 80%) fall between 0.298ms and 0.533ms for LargeComm. This small variation in the distribution, however, is due to a different reason: the MPICH algorithm, which is the one used in the MVAPICH2 implementation does a single `MPI_ALLGATHER` of 2 integers per rank at the beginning of the split (as does all Open MPI implementations), then a single `MPI_ALLREDUCE` of 64 integers (to represent the 2,048 bits that each represent a possible communicator Identifier). As illustrated in Figure 4, MVAPICH2 `MPI_ALLGATHER` operation at 1,024 bytes (256 integers) exhibits a variability that explains this distribution of durations for the `MPI_COMM_SPLIT` operation.

The three proposed algorithms replace the conservative loop of `MPI_ALLREDUCE` calls in theOpen MPI Vanilla implementation with a single `MPI_ALLREDUCE` of a single integer, to decide on the highest available CID. Compared to the previous implementations, they all feature faster performance: compared to Open MPI Vanilla Implementation, the benefit of removing the `MPI_ALLREDUCE` loop is obvious; compared to the MVAPICH2 implementation, the reduction in size for the `MPI_ALLREDUCE` call is still significant. Durations are distributed over a small interval, with more than 90% of the durations in the first interval of Figure 2, and in the first two intervals of Figure 3.

In both cases, the Highest-CID Algorithm runs faster, as insertion and lookup are a single operation, once the associative array storage is dimensioned to the right size. Average performance of the Multidimension Dynamic Array and the Red-Black Tree are similar.

## 4.3 Latency Overheads

For this section, we focus on the overheads introduced by the proposed algorithms. As we will discuss internal management of data structures, we will compare the four Open MPI-based implementations in a homogeneous environment and under strictly identical conditions: all runs produce the same number of communicators, in the same order, and involving the same processes.

In the experiment represented in the first row of Figure 5, the two processes that communicate are always the rank 0 and the rank 127 of `MPI_COMM_WORLD`. Based on the process map, these processes always reside on different nodes and therefore use the Infiniband network to exchange messages.

One observes that the latency is not significantly impacted in both cases: all four algorithms present a similar duration distribution, with an average corresponding to the mean value

given by a traditional latency measurement like NetPIPE.

Figure 5's second row does the same measurement, but between ranks 0 and 1 of the `MPI_COMM_WORLD` communicator. Processes are collocated on the same node – by the processes map –, in which case all communications between them happen on shared memory. It is interesting to include these results as the shared memory latency is extremely sensitive to all external factors, and will be quickly impacted by any overhead related to the CID usage.

In both cases (LargeComm and SmallComm), the different implementations provide significantly different latency distributions: Highest-CID obtains the best performance, then Open MPI Vanilla, the Multidimension Dynamic Array implementation, and then the Red-Black Tree implementation. The Highest-CID Algorithm, and the Open MPI Vanilla Algorithm use a single memory reference to find the communicator associated with a message upon reception. The two other algorithms use a series of lookup (4 in the case of the Multidimension Dynamic Array, up to $2\log_2(n)$, where $n$ is the number of local communicators, for the Red-Black Tree), and these lookups (and the corresponding memory loads that they imply) explain this behavior.

## 4.4 Memory Overheads

Figures 6, and 7 present the average density of the associative array used to store the communicators objects in three cases: for the Open MPI Vanilla Algorithm, the Highest-CID Algorithm and the Multidimension Dynamic Array Algorithm. It is unnecessary to evaluate the Red-Black Tree algorithm, because it maintains its structure density at all times, since single elements are added or removed from the tree as communicators are allocated or freed. In contrast, the three variants under investigation use a sparse structure to build the association, so they introduce a memory overhead that we evaluate in this experiment.

The Open MPI Vanilla Algorithm and Highest-CID Algorithm both use a linear array that is extended when a new CID is outside the bounds of the array. These arrays are not shrunk in size when CIDs are freed. The Multidimension Dynamica Array algorithm allocates new arrays by chunks of 256 elements when a CID that falls in a new chunk is allocated. They are also freed when chunks become empty.

The density represented in these figures is the number of elements that are occupied, divided by the number of elements allocated. A density of 1 means that no extra memory is used (as is the case with the Red-Black Tree approach). In both cases (SmallComm and LargeComm), the Open MPI Vanilla Algorithm allocates the smallest available CID, thus minimizing the size of its array. This is well represented in Figure 6, where, for most ranks, the density of the Open MPI Vanilla Algorithm is the highest. SmallComm creates small communicators, thus favors the cases when the same CID can be used in two non-overlapping groups, reducing the size of its linear array compared to the Highest-CID Algorithm, and thus increasing the density. In the case of the LargeComm, by contrast, communicators are large and the probability of having two communicators with the same CID on non-overlapping groups is negligible. As a result, the Highest-CID Algorithm and the Open MPI Vanilla Algorithm allocate the same CIDs and present exactly the same density.

The Multidimension Dynamic Array storage strategy of freeing empty chunks allows for obtaining a slightly higher density in this case: when communicators are freed, the associative array releases the empty chunks and decreases the memory overheads. In the case of SmallComm, however, this strategy is not fruitful as the average density remains rather low. For a large number of small communicators, only the Red-Black Tree allows a high memory utilization.

# 5. RELATED WORK

The impact of dynamic process management on communicators creation is evaluated in [8]. The context ID allocation we address in this work, as well as its improvement, will benefit all MPI routines that create communicators, and thus also benefits the dynamic process management routines.

In [11], the authors focus on the group-related memory scalability issues in communicators: different data structures are proposed to internally represent the groups connected to different communicators, and especially techniques to reduce the redundancy of information of such groups within a manycore shared memory machine. The same issue of group representation and storage is addressed in [4], and [2] demonstrate how this issue is critical for scalability. Their contributions can be combined with ours to provide a better usage of memory as well as accelerate the allocation algorithm itself.

In [7], the authors note that if the user provides enough information, the existing communicator creation functions within the current MPI Standard allow for creating the communicator by communicating only between the processes of the target group, avoiding the traditional collective meaning of the MPI communicator creation functions over the group of the parent process. They proposed an API that provides the target group, allows such an implementation, and they evaluate the performance of this approach. Since then, the proposed API was integrated in the MPI Standard. In this work, the authors do not address the challenge of improving the efficiency of Context Identifier Allocation.

Inside the MPI Forum, and in the literature, ideas on enhancing the communicator concept are discussed. In [5], the authors propose generalization of the communicator interface to enable a finer grain of parallelism in MPI. In [10], the authors consider the specific case of handling failures and its impact on communicators management. The approach presented would allow for dynamically changing the groups of communicators to remove the failed processes. [3] argues that providing an interface based on new communicator creation provides a better abstraction to the user, by clearly separating communications that happened before and after a failure, if the user needs to enter a global recovery and fix the existing communicators. This is partly the reason why communicator creation must become more efficient.

[1] looked at optimizing many collectives during communicator creation for the BlueGene platform. Depending on the shape of the communicator, collectives, including the ones involved in communicator creation routines, are optimized

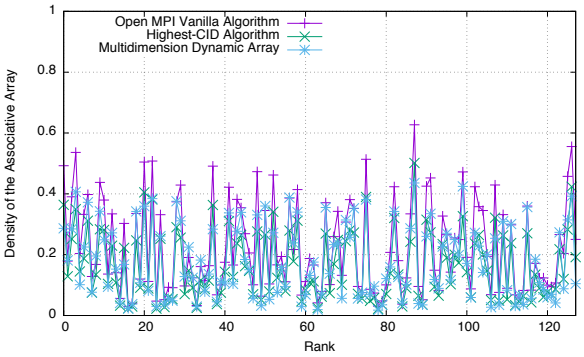


**Figure 6: Associative Array Density (SmallComm)**

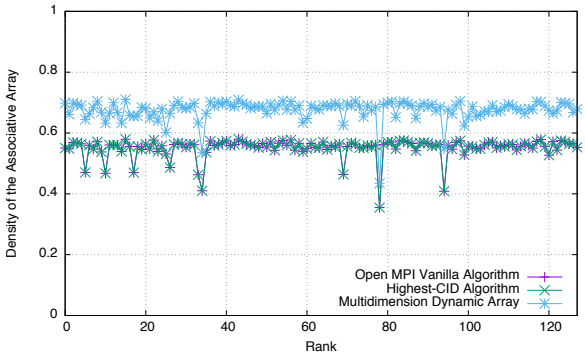


**Figure 7: Associative Array Density (LargeComm)**

to take advantage of the underlying Torus network topology. However, in the general case, the implementation resorts to the MPICH algorithm, and communicator identifier allocation is not specifically targeted in this work.

In [9], the authors present the Open MPI Implementation and its CID allocation algorithm. In [6], the authors present the multithreaded CID allocation in MPICH3 (see Section 3). We compared the algorithms we propose with both implementations. The benefit over the Open MPI Algorithm lies in the drastic reduction of the number of collective calls to allocate a new CID. While the MPICH3 algorithm is efficient, it relies on a fixed bound in the number of communicators, and we show in this paper that better performance in the communicator allocation can be achieved at the price of using sparse data structures to store the associative array of the communicator objects.

## 6. CONCLUSIONS

Communicator creation is an essential part of an MPI application that is taken out of the sole initialization phase by the use of dynamic process management and fault tolerance. In this work, we considered three sources of cost linked to communicator creation and management: the time to assign a unique identifier (the Communicator IDentifier) for the new communicator; the time overhead to find a communicator from its CID in an associative array; and the memory overhead of storing this associative array.

We proposed three new variants in Open MPI, and compared the features and performance of these three variants with the algorithm implemented in MVAPICH2 (Section 3.2). By releasing the constraints of density in the search for a CID, we were able to significantly improve the performance of CID allocation in Open MPI. The proposed approach requires a single `MPI_ALLREDUCE` of a single integer, allowing for taking advantage of the hardware accelerated implementation of such collective. Our implementation achieved the same level of performance as MVAPICH2, but without enforcing an extremely low limit on the number of potential CIDs.

However, as the set of CIDs will become less dense, we considered a few options to store these CIDs without sacrificing a significant part of the memory, or increasing the overhead for the lookup of communicators from their CIDs. As often is the case, a trade-off arises between computation and memory, and the most efficient solutions – in terms of storage – are also the ones that introduce the highest timing overheads.

These overheads were measured at approximately 10%, and can be considered as acceptable under certain circumstances: the user may be inclined to chose one approach or another depending on if memory consumption, latency, or frequent communicator creation is the critical metric of performance. Thanks to the modular nature of Open MPI, all approaches can co-exist in the code base, under the Modular Architecture Component, providing control to the user to choose, at runtime, the fittest approach for the application.

We plan to pursue this work by integrating other performance improvements into the communicator creation routines, like the reduction of redundant information shared by other processes in the same node.

## 7. ACKNOWLEDGMENTS

This work was supported in part by the NSF through the Award #1339763 "SI2-SSE: Collaborative Research: ADAPT: Next Generation Message Passing Interface (MPI) Library - Open MPI", and by the CREST project of the Japan Science and Technology Agency (JST).

## 8. REFERENCES

[1] G. Almási, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng. Optimization of MPI collective communication on BlueGene/L systems. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 253–262, New York, NY, USA, 2005. ACM.

[2] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, and J. L. Traff. MPI on millions of cores. *Parallel Processing Letters*, 21(01):45–60, 2011.

[3] W. Bland, A. Bouteiller, T. Hérault, G. Bosilca, and

J. Dongarra. Post-failure recovery of MPI communication capability: Design and rationale. *IJHPCA*, 27(3):244–254, 2013.

[4] M. Chaarawi and E. Gabriel. Evaluating sparse data storage techniques for MPI groups and communicators. In M. Bubak, G. van Albada, J. Dongarra, and P. M. Sloot, editors, *Computational Science - ICCS 2008*, volume 5101 of *Lecture Notes in Computer Science*, pages 297–306. Springer Berlin Heidelberg, 2008.

[5] E. Demaine, I. Foster, C. Kesselman, and M. Snir. Generalized Communicators in the Message Passing Interface. *Parallel and Distributed Systems, IEEE Transactions on*, 12(6):610–616, Jun 2001.

[6] J. Dinan, D. Goodell, W. Gropp, R. Thakur, and P. Balaji. Efficient multithreaded context ID allocation in MPI. In *Recent Advances in the Message Passing Interface - 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings*, pages 57–66, 2012.

[7] J. Dinan, S. Krishnamoorthy, P. Balaji, J. Hammond, M. Krishnan, V. Tipparaju, and A. Vishnu. Noncollective communicator creation in MPI. In Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 6960 of *Lecture Notes in Computer Science*, pages 282–291. Springer Berlin Heidelberg, 2011.

[8] E. Gabriel, G. Fagg, and J. Dongarra. Evaluating the performance of MPI-2 dynamic communicators and one-sided communication. In J. Dongarra, D. Laforenza, and S. Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2840 of *Lecture Notes in Computer Science*, pages 88–97. Springer Berlin Heidelberg, 2003.

[9] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In D. Kranzlmuller, P. Kacsuk, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*, pages 97–104. Springer Berlin Heidelberg, 2004.

[10] R. Graham and R. Keller. Dynamic communicators in MPI. In M. Ropo, J. Westerholm, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5759 of *Lecture Notes in Computer Science*, pages 116–123. Springer Berlin Heidelberg, 2009.

[11] H. Kamal, S. M. Mirtaheri, and A. Wagner. Scalability of communicators and groups in MPI. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 264–275, New York, NY, USA, 2010. ACM.

[12] H. Park and K. Park. Parallel algorithms for red-black trees. *Theor. Comput. Sci.*, 262(1-2):415–435, July 2001.