Efficient Multithreaded Context ID Allocation in MPI*

James Dinan,[1] David Goodell,[1] William Gropp,[2]
Rajeev Thakur,[1] Pavan Balaji[1]

[1] Argonne National Laboratory. {dinan,goodell,thakur,balaji}@mcs.anl.gov
[2] University of Illinois at Urbana-Champaign. wgropp@illinois.edu

Abstract. An important aspect of support for multithreaded MPI executions is the management of communication context identifiers (IDs), which are used to associate MPI communication operations with a communicator. New communicator creation functionality in MPI 3.0 adds complexity to this core resource management problem. We present an efficient algorithm for multithreaded context ID allocation that builds on an existing production algorithm developed to support MPI 2.2. Through this work, we have discovered a subtle concurrency bug in the existing algorithm that can result in deadlock. We correct this bug and develop methods to overcome the performance impact of deadlock prevention. We evaluate the performance of the new algorithm and prove that it is free from deadlock.

1 Introduction

Hybrid parallel programming that combines MPI with a shared-memory programming model, such as threads or OpenMP, has become a popular paradigm for constructing scalable and efficient high-performance applications. In this model, MPI is used for internode coordination and data movement, while hardware-supported shared memory is leveraged to achieve efficient intranode execution. The adoption of such hybrid programming techniques has been driven by sustained increases in the number of cores and hardware threads provided per processor. This trend indicates not only that MPI must interoperate well with a variety of shared-memory programming models but also that MPI must efficiently manage increasing levels of concurrency within the MPI library [6].

The MPI 2.2 standard [7] defined the interaction of MPI with threads, and significant effort was invested to extend MPI implementations to support this new hybrid execution model [5,8,9]. An important component of this effort was the development of multithreaded context identifier (ID) allocation algorithms [5]. MPI uses context IDs internally to match communication operations with communicators; a new context ID must be generated each time MPI constructs a new communicator.

* This work was supported in part by the U.S. Department of Energy under contract DE-AC02-06CH11357 and the Advanced Scientific Computing Research program, Office of Science, U.S. Department of Energy award DE-FG02-08ER25835.

The MPI 3.0 specification that is nearing completion contains new functionality that will require modification to how an MPI implementation allocates context IDs. Among these changes is a new, noncollective communicator creation routine that supports multiple concurrent invocations that are differentiated by using a tag argument [3]. Traditional communicator creation routines are collective over all processes in a parent communicator; in contrast, this new routine is collective over only the group of processes that will be members of the new communicator. The functionality of this routine is expected to address several key application needs. For example, when a node failure occurs, processes must create a new communicator to re-establish collective communication; however, traditional communicator creation would require participation from failed processes. This routine also enables the use of load balancing techniques that asynchronously reassign idle processes to heavily loaded execution teams [2]. Moreover, this routine can significantly reduce the cost of communicator creation by including only processes that will be members of the new communicator.

We have extended the MPICH2 [1] multithreaded context ID allocation routine to support this new functionality. Through this work, we discovered a bug in the production algorithm that can result in deadlock. We correct this bug and discuss techniques to eliminate the performance impact of deadlock prevention. We prove that the new algorithm meets all constraints and is free from deadlock. We compare the performance of native noncollective communicator creation with the user-level algorithm [3] and demonstrate that the native implementation provides a significant speedup by eliminating $O(\log n)$ communicator creation operations. Furthermore, we show that noncollective communicator creation can provide a several-fold speedup over collective communicator creation when the output group is smaller than the parent communicator.

This paper is organized as follows. In Section 2 we discuss the interaction of MPI with threads and the new, noncollective communicator creation routine that will be added in MPI-3. Section 3 presents the enhanced multithreaded context ID allocation algorithm, which prevents the deadlock condition that was possible in the existing algorithm. In Section 4 we prove correctness and deadlock freedom for this new routine. An optimization to eliminate deadlock prevention overheads is presented in Section 5. We evaluate the performance of our implementation in Section 6 and summarize our conclusions in Section 8.

2 Multithreading and MPI

The MPI 2.0 standard defined MPI's interaction with threads; several levels of multithreading are supported depending on the needs of the application. In this work, we consider the case where MPI is initialized to support MPI_THREAD_MULTIPLE because other, more restrictive threading levels don't subject the MPI implementation to a level of concurrency that necessitates these techniques.

Of particular importance to this work is the interaction of MPI collectives with threads. The MPI standard states that a program may perform only one

collective operation per communicator at a time. If the application uses threads, the programmer must ensure that threads do not perform multiple collectives concurrently on the same communicator. However, multiple collectives can be issued concurrently on different communicators. In the MPI-2 standard, communicator construction is a collective operation that is performed on a parent communicator. Thus, when threads are in use, multiple communicator construction operations can be issued concurrently if the parent communicators are different.

A new routine, MPI_Comm_create_group, which was proposed by the authors, is anticipated in MPI 3.0. The input to this routine is a parent communicator, a group of processes (represented by an MPI_Group object) that will be members of the new communicator, and a tag argument. All processes in the input group must call this routine with the same arguments. A new communicator containing the processes in this group is returned as output; if a process is not a member of the group, MPI_COMM_NULL is returned. In contrast with other collective operations, this routine can be invoked concurrently by multiple threads on the same parent communicator. In such a case, each call must use a distinct tag argument, which is used to distinguish operations. Communication generated by this routine is defined not to interfere with other point-to-point communication on the parent communicator, even if the same tag (or MPI_ANY_TAG) is already in use.

3 Multithreaded Context ID Allocation Algorithm

Communicators are internally identified by an integer context ID in most MPI runtime systems. The context ID value uniquely identifies a given communicator on all processes that are members of the communicator's group. This value is included as a part of the message envelope and is used to ensure that communication operations match only within the same communicator. Allocation of the context ID is at the core of all communicator creation operations. To ensure efficient message matching, all known MPI implementations use context ids that are unique and uniform across all involved processes.

In MPI-2, a parent communicator that contains all processes in the output communicator's group is used to perform any communication needed to select the context ID—typically a collective allreduce operation. Multiple communicator creation operations can be issued concurrently by different threads on different parent communicators; however, the user must ensure that operations are ordered such that only one collective operation is performed on a given parent communicator at any time. In contrast with these semantics, the new MPI_Comm_create_group routine permits threads to issue multiple such operations concurrently on the same parent communicator, and individual operations are distinguished through an additional tag argument.

We have extended the existing multithreaded MPICH2 context ID allocation algorithm [5] to include support for MPI_Comm_create_group; we present the modified algorithm in Listing 1.1. In this algorithm, the state of all context IDs is tracked through a vector of Boolean entries, called mask. The context ID mask

```
/* Input: my_comm, my_group, my_tag.  Output: integer context ID          */
/* Shared variables ( shared by threads at a each process )               */
mask                  /* Bit array, indicates if each context ID is free  */
mask_in_use   = 0     /* Flag, indicates if mask is in use                */
lowest_ctx_id = MAXINT /* Indicates which thread has the highest priority */
lowest_tag            /* Breaks lowest_ctx_id priority ties               */

/* Private variables ( not shared across threads )                        */
local_mask            /* Thread private copy of the mask                  */
i_own_the_mask = 0    /* Flag indicating if this thread holds the mask    */
context_id     = 0    /* Output context ID                                */

MPIR_Barrier_group(my_comm, my_group, my_tag) /* new barrier, prevents deadlock */
while ( context_id == 0 ) {
  Mutex_lock ()
  if ( my_comm->context_id < lowest_ctx_id
       || ( my_comm->context_id == lowest_ctx_id && my_tag < lowest_tag ) ) {
    lowest_ctx_id  = my_comm -> context_id
    lowest_tag     = my_tag
  }
  if ( !mask_in_use
       && my_comm->context_id == lowest_ctx_id && my_tag == lowest_tag ) {
    local_mask     = mask
    mask_in_use = 1, i_own_the_mask = 1
  }
  else {
    local_mask  = 0, i_own_the_mask = 0
  }
  Mutex_unlock ()
  MPIR_Allreduce_group ( local_mask, MPI_BAND, my_comm, my_group, my_tag )
  if ( i_own_the_mask ) {
    Mutex_lock ()
    if ( local_mask != 0 ) {
      context_id       = location of first set bit in local_mask
      mask[context_id] = 0
      if ( lowest_ctx_id == my_comm->context_id && lowest_tag == my_tag ) {
        lowest_ctx_id = MAXINT
      }
    }
    mask_in_use = 0
    Mutex_unlock ()
  }
}
```

Listing 1.1. Multithreaded context ID allocation algorithm.

vector is an array of bits, where the nth bit identifies whether the context ID value n is unused. To allocate a context ID, processes perform a bitwise AND allreduce on the full vector and select the first context ID corresponding to the first nonzero bit.

Group-collective allreduce and barrier routines were created that include only the processes specified by a group argument. Multiple group-collective operations can occur concurrently in different threads, and operations are distinguished by using the `tag` argument provided by the user. One bit in the tag space was reserved to indicate messages using the given tag correspond to group-collective operation traffic. Thus, we ensure that messages generated during context ID allocation do not conflict with application-generated point-to-point operations.

Concurrent attempts to allocate context IDs by multiple threads in the same process are managed by allowing only one thread to access the context ID mask at any given time. If a thread is not able to gain access to the mask, it must

still participate in the allreduce to prevent another thread in its communicator creation operation from blocking indefinitely while holding the mask. Threads that are unable to access the context ID mask pass a vector of zeroes to the allreduce, indicating that no context IDs are currently available and effectively aborting the attempt to allocate the context ID. Thus, multiple attempts may be needed for a successful allreduce to occur; threads continue to retry until a context ID is successfully allocated. In order to prevent livelock where threads cause each other to mutually abort indefinitely, a simple prioritization scheme is used where the threads whose parent communicator has the lowest context ID are given priority for access to the context ID mask.

Two components of the existing algorithm were modified to support the new MPI_Comm_create_group routine. A group-collective version of allreduce was substituted for the communicator-collective implementation. In addition, the prioritization scheme was modified to prioritize threads based on the $\langle context\ id, tag \rangle$ pair.

Freeing a given context ID, n, requires simply acquiring the mutex and marking $mask[n] = 1$ without waiting for the mask to be available. Threads update only one bit in the mask at a time while holding the mutex. If a context ID, n, is freed during a concurrent allocation attempt, the ongoing allocation attempt will continue to use the initial value $n = 0$ of the given context ID. Context ID n will be observed as available during the next allocation attempt.

3.1 Deadlock Issue and Prevention Mechanism

The existing version of this multithreaded context ID algorithm has been deployed in production for several years in MPICH2, and it was verified by a student collaborator using model checking [5]. However, while extending the algorithm to support noncollective communicator creation, we discovered a subtle bug that can lead to deadlock in the multithreaded case The existence of this bug for several years indicates that the algorithm may not have been used extensively in the field in multithreaded situations, and that the model used for verification did not capture the case that leads to the deadlock.

The existing algorithm did not contain the barrier in Listing 1.1, line 13. In the absence of the barrier, a thread is permitted to reserve the context ID mask and perform the collective allreduce even though the other threads may not have made matching calls to the communicator creation routine, introducing the possibility of a hold-and-wait scenario. If another thread in the same process attempts to perform a second communicator creation operation, a circular wait can occur, resulting in deadlock. One such scenario is illustrated in Figure 1, where there are two processes and two threads per process; each thread executes the following code.

```
if (thread_id == mpi_rank) { MPI_Comm_dup(MPI_COMM_SELF, &self_dup); }
MPI_Comm_dup(thread_comm, &thread_comm_dup);
```

Here, thread_comm is a communicator of the threads at all processes with the same thread ID. In this scenario, calls to duplicate thread_comm block in the allreduce

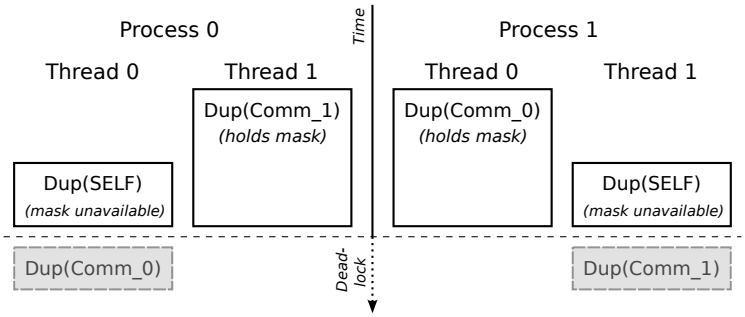


Fig. 1. Deadlock scenario with two processes and two threads per process. Blocked threads hold the mask, preventing other threads from making matching collective calls.

while holding the mask, preventing the calls to duplicate MPI_COMM_SELF from successfully allocating a context ID.

To avoid this deadlock scenario, we break the hold-and-wait condition by ensuring that all threads have arrived before allowing them to reserve the mask. This is accomplished by performing a barrier before entering the allocation loop. Once threads have completed the barrier, they can perform the allreduce and reserve the mask without risk of deadlock. The addition of the barrier, can have a negative impact on performance; in Section 5 we present an optimization that avoids this barrier in many cases while still avoiding the deadlock.

4 Proof of Correctness

We demonstrate that the multithreaded context ID allocation algorithm guarantees progress and ensures that allocation invariants are not violated under a failure-free assumption. MPICH2 requires that the following conditions are satisfied for a context ID allocation to be correct.

Property 1. For a given operation, all processes select the same context ID and this ID is allocated at most once at every process.

Uniform context IDs are guaranteed through the allreduce, which ensures that the output mask is the same at all processes. In addition, the locking discipline and bitwise AND reduction operation ensure that if the given context ID is unavailable (the corresponding bit is zero) at any process, it will be observed as unavailable in the output mask at all processes.

4.1 Proof of Progress

To prove global progress, we must first prove the following liveness property, which did not hold in the original version of the algorithm:

Property 2. No thread can block indefinitely.

The initial barrier added to the algorithm ensures that all processes with the same parent communicator, group, and tag arguments are present before they can reserve the mask or be prioritized for access to the mask. If the mask has been reserved by another thread, threads supply a zero mask to indicate that the mask is locally unavailable. These mechanisms ensure that all necessary threads participate in the ensuing allreduce operation.

Property 3. Threads with the globally highest priority will eventually succeed.

Access to the mask is prioritized according to the $\langle context\,id, tag\rangle$ pair. This prioritization is critical to ensure that threads do not enter a livelock situation where allocation attempts repeatedly fail because all threads in a given group are unable to obtain the mask at the same time. As threads iterate in the allocation loop, they update the $\langle context\ id, tag\rangle$ pair when their value is less than the current minimum. Threads are permitted to reserve the mask only when they have the highest priority, and they must release the mask after an allocation attempt if they no longer have the highest priority. A total $\langle context\,id, tag\rangle$ ordering can be imposed across all threads. The prioritization scheme ensures that threads with the globally highest priority will eventually be able to acquire the mask at all processes involved in the given operation. Assuming that a common, free context ID exists, the globally highest priority operation will eventually succeed.

Property 4. The highest priority thread at a process will eventually succeed.

Once the globally highest priority operation succeeds, the priority variables are reset, and the next $\langle context\ id, tag\rangle$ pair in the total ordering has the highest priority. A consequence of a prioritization scheme based on such a total ordering of operations is that a low-priority operation can be starved by repeated arrivals of higher-priority operations. Since realistic MPI programs perform a finite number of communicator creation operations, this starvation is bounded in practice. If we assume such finite MPI programs, we observe that as the globally highest-priority operations complete, a locally highest-priority operation will eventually become the globally highest priority and also complete.

Property 5. Every allocation attempt will eventually succeed.

When a locally highest-priority operation completes, the next highest-priority operation becomes the locally highest priority. Thus, assuming finite MPI programs, every operation eventually gains the locally highest priority and completes.

5 Eliminating the Overhead of Deadlock Avoidance

As discussed in Section 3, a synchronization step before entry to the context ID allocation loop is necessary to avoid deadlock. In Listing 1.1 a barrier is used as a simple solution; however, any operation that synchronizes all threads is sufficient to prevent them from entering the context ID allocation loop until all threads

```
/* Shared variables ( shared by threads at a each process )                */
eager_split             /* Reserves mask[0..eager_split-1] for eager alloc.*/
eager_mask_in_use = 0   /* Flag, indicates if eager mask is already in use */

/* Private variables ( not shared across threads )                         */
i_own_eager_mask  = 0   /* Flag, indicates if this thread has eager mask   */

Mutex_lock ( eager_lock )
if ( ! eager_mask_in_use ) {
  eager_mask_in_use = 1, i_own_eager_mask  = 1
  local_mask        = mask[0..eager_split-1]
} else {
  local_mask        = 0
}
Mutex_unlock ( eager_lock )
MPIR_Allreduce_group ( local_mask , MPI_BAND , my_comm, my_group, my_tag )
if ( i_own_eager_mask ) {
  Mutex_lock ( eager_lock )
  eager_mask_in_use = 0, i_own_eager_mask  = 0
  context_id        = location of first set bit in local_mask
  mask[context_id]  = 0
  Mutex_unlock ( eager_lock )
}
```

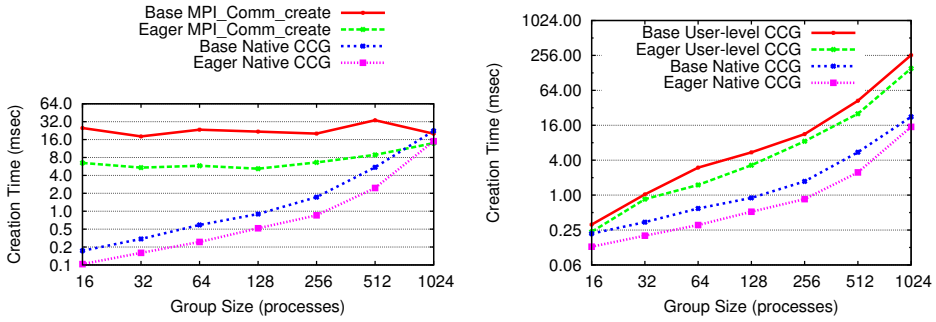
Listing 1.2. Multithreaded context ID allocation algorithm.

have made matching calls. We use this observation to attempt allocation of a context ID during the synchronization step itself. In most MPI programs, this method results in successful context ID allocation in a single step and eliminates the additional overhead incurred by the new synchronization step.

This *eager* mode of context ID allocation is achieved by splitting the context ID mask into *eager* and *base* segments. During the synchronization step, an allreduce on the eager context ID space is performed. If eager allocation fails, the allreduce effectively acts as a barrier and threads proceed to the base allocation algorithm, which utilizes the base segment of the mask. Many variations of this optimization are possible, and an important property is the method used to divide the context ID space between eager and base protocols. Dynamic approaches where each process selects the first n available context IDs are possible. However, fragmentation in the context ID mask due to repeated communicator creation and destruction can cause individual masks to diverge and render this approach ineffective. Static allocation approaches are not as susceptible to this issue, but they limit the number of context IDs available to each protocol.

We define a static allocation strategy that reserves the first n context IDs for eager allocation and utilizes the remaining $max\_id - n$ IDs for the base protocol, where n is a configurable parameter. This strategy maximizes the likelihood of successful eager allocation and, as a tradeoff, in the worst case reduces the size of the context ID space to $max\_id - n$. The algorithm is shown in Listing 1.2, and this code is substituted in place of the barrier in Listing 1.1, line 13. In addition, line 23 in Listing 1.1 must be modified as follows.

```
local_mask[ 0..eager_split-1 ] = 0
local_mask[eager_split..MAXID] = mask
```



(a) Comparison with MPI_Comm_create (b) Comparison with User-level CCG

Fig. 2. Comparison of native MPI_Comm_create_group (CCG) performance with two other communication creation routines. The impact of eager versus baseline allocation is shown for each algorithm. The input communicator always contains 1024 processes.

6 Experimental Evaluation

We compare the cost of several communicator-creation operations on a QDR InfiniBand cluster. Each node is configured with two 2.6 GHz, quad-core Intel Nehalem processors, and 36 GB memory. In Figure 2, we use MPI_COMM_WORLD as the parent communicator and vary the size of the output communicator. We compare the execution time of MPI_Comm_create with MPI_Comm_create_group (CCG) in Figure 2a. Two implementations of MPI_Comm_create_group are evaluated in Figure 2b: a user-level implementation that performs recursive intercommunicator merging [3] and a direct implementation that uses the new context ID allocation algorithm. For each comparison, we show the impact of the eager allocation optimization.

From these results, we see that MPI_Comm_create_group provides a significant performance advantage over MPI_Comm_create, whose cost is always proportional to the size of the parent communicator. In addition, we see that there is an $O(\log p)$ performance advantage of the direct $O(\log p)$ MPI_Comm_create_group algorithm over the $O(\log^2 p)$ user-level algorithm. Moreover, the eager-allocation protocol accomplishes allocation in one, rather than two, allreduce operations, yielding a factor of two or more improvement in communicator-creation cost.

7 Related Work

Open MPI [4] utilizes a similar approach to context ID allocation; however, rather than considering the full context ID mask when performing allocation, one context ID is evaluated at a time. In this algorithm, each thread starts with its lowest available context ID and a MAX allreduce is performed. A second AND allreduce is performed to determine if the operation succeeded at all threads and, if not, the process is repeated with the next highest context ID. Assuming

a fixed-size context ID space, this algorithm is also susceptible to the deadlock scenario presented in Section 3.1 when the number of threads attempting allocation approaches the number of available context IDs at any process. In comparison with the algorithm presented in this work, the algorithm currently used by Open MPI always performs two allreduce operations (on a single integer, rather than the full mask) and can require multiple iterations if the context ID space becomes fragmented. This algorithm can also be extended to support MPI_Comm_create_group through the same priority and tag space approaches presented in Section 3.

8 Conclusions

We have presented an efficient, multithreaded context ID allocation routine that includes support for new functionality in MPI 3.0. This work builds on the existing MPICH2 algorithm that was found to contain a subtle deadlock bug. We corrected this bug and proposed an eager allocation protocol that eliminates the performance impact of deadlock avoidance. We prove correctness of the new algorithm and evaluate its performance relative to existing approaches. Results indicate that the MPI_Comm_create_group routine built on top of the multithreaded context ID allocation algorithm significantly reduces the cost of communicator creation when the output group is smaller than the parent communicator's group.

References

1. MPICH2 Project Website, `http://www.mcs.anl.gov/research/projects/mpich2/`
2. Arafat, M.H., Dinan, J., Krishnamoorthy, S., Windus, T., Sadayappan, P.: Load balancing of dynamical nucleation theory Monte Carlo simulations through resource sharing barriers. In: Proc. 26th Intl. Par. and Distrib. Processing Symp. (May 2012)
3. Dinan, J., Krishnamoorthy, S., Balaji, P., Hammond, J.R., Krishnan, M., Tipparaju, V., Vishnu, A.: Noncollective communicator creation in MPI. In: Recent Adv. in the Message Passing Interface - 18th European MPI Users' Group Mtg. (2011)
4. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting. pp. 97–104. Budapest, Hungary (September 2004)
5. Gropp, W., Thakur, R.: Thread-safety in an MPI implementation: Requirements and analysis. Parallel Computing 33(9), 595–604 (2007)
6. Kamal, H., Mirtaheri, S.M., Wagner, A.: Scalability of communicators and groups in MPI. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing. pp. 264–275. ACM, New York (2010)
7. MPI Forum: MPI: A Message-Passing Interface Standard. Version 2.2 (September 4, 2009)
8. Protopopov, B.V., Skjellum, A.: A multithreaded message passing interface (MPI) architecture: Performance and program issues. Journal of Parallel and Distributed Computing 61(4), 449–466 (April 2001)
9. Tang, H., Yang, T.: Optimizing threaded MPI execution on SMP clusters. In: Proc. 15th ACM International Conference on Supercomputing. pp. 381–392 (June 2001)