Context Id Allocation with MPI Endpoints

Dr Daniel Holmes
EPCC
The University of Edinburgh
Edinburgh, UK
dholmes@epcc.ed.ac.uk

Abstract—The MPI Forum is considering the introduction of MPI endpoints into a future version of the MPI Standard to improve interoperability between MPI and threads, which is suggested as a necessary incremental step towards a practicable exascale programming model. This paper shows two viable approaches to providing support for MPI endpoints in the current algorithm used by the MPICH library for context ID allocation. Initial performance results from a full prototype implementation of one of these approaches in the McMPI library demonstrate that no additional communication overhead is introduced to McMPI due to endpoints.

Keywords—exascale programming; MPI; threads; context ID allocation

I. INTRODUCTION

It seems likely that exascale hardware will involve CPUs with many more cores than currently available and it is generally accepted that programming models will have to evolve to make efficient use of such architectures. The MPI Standard [1], the de facto standard expression of the message-passing programming model, is a process-centric programming system. It is not obvious that a pure-MPI application code can efficiently scale to take advantage of the next generation of large supercomputers, i.e. exascale machines [2]. Other programming systems that may be combined with MPI can be described as thread-centric, like OpenMP, or thread-agnostic, like UPC and most other expressions of the PGAS programming model. The EESP Roadmap [3] suggests that “applications may contain ten billion threads”. Therefore, for MPI to remain relevant and useful for high-performance computing, good interoperability between MPI and threads must be a priority.

For efficient interoperability with threads, a programming model must support “unambiguous addressability of remote targets, separability [sic] of communication paths, and full direct reachability” [4]. Currently, there are two approaches to MPI programming when multiple threads will communicate using point-to-point messages – tags and communicators. The tag parameter can be used to provide unambiguous addressability for each thread individually: part of the tag can be used to specify the thread that sent the message and another part can be used to specify the thread that should receive the message. However, using a wildcard value for the tag parameter requires extreme care, collectives can still only be used by one thread at a time per process and the internal message queue structures cannot be partitioned because the MPI library is not aware that messages with particular tag values will always be received by particular threads. The alternative is to use a different communicator for each thread to provide separability of communication paths used by each thread in the same process. However, in this model only one thread from each MPI process can safely use each communicator. This either limits the direct reachability of threads or requires a prohibitively large number of communicators.

The endpoints proposal [5, 6] permits the creation of a communicator with multiple ranks for each MPI process in a parent communicator. The new communicator creation function is collective over all the MPI processes in the parent communicator and it returns an array of handles for the new communicator. The length of the array, which is also the number of endpoints created for the calling MPI process, is specified by an argument to the function. Each handle in the array corresponds to one of the ranks in the new communicator. The ranks represented by the handles in the array are contiguous and the starting rank is defined by the rank of the calling MPI process in the parent communicator. The number of endpoints can be different for each calling MPI process. The new communicator will be the same type as the parent communicator, which can be either an intra-communicator or an inter-communicator. Each endpoint created by the proposed function must ensure the same semantics as already defined elsewhere in the MPI Standard for an MPI process, i.e. section 3.5 of the MPI-3.0 document [1]. No correspondence is enforced by MPI between the handles for an endpoints communicator and the threads of an operating system process. The number of endpoints may or may not be the same as the number of threads. Any thread can make use of any communicator handle at any time during the execution that is permitted by the thread support level chosen when MPI was initialised.

II. MPICH ALGORITHM FOR CONTEXT ID ALLOCATION

The current MPICH source code includes four versions of context id allocation algorithm. There is an outstanding issue with the two multi-threaded versions, which requires that the older single-threaded versions must still be maintained. In this paper the single-threaded versions are considered to be legacy code and the focus is exclusively on the multi-threaded versions. Previous work [7] explains the blocking algorithm in detail and sets out the necessary changes to add support for the
MPI_COMM_CREATE_GROUP function. Essentially, bit vectors that represent the local availability of context identifiers from all involved MPI processes are combined to determine global availability. Various scenarios cause the global availability bit vector to indicate that no context identifiers are available. At least one MPI process may legitimately have exhausted the supply of context identifiers. Alternatively, at least one MPI process may have been unable to determine the local availability because another operation (on another thread) has higher priority. When this happens the algorithm is repeated until it is successful. The blocking algorithm for intra-communicators can be summarised by the following:

```c
while (!context_id) {
    // may be set to all zeros, if not highest priority
    local_bit_vector = set_local_mask();
    // may result in all zeros
    global_bit_vector = all_reduce(local_bit_vector);
    // may result in a zero for context id
    context_id = find_and_allocate(global_bit_vector);
}
```

The set_local_mask function in the above pseudo-code attempts to set a local variable to the current value of the availability mask from shared state. It can only succeed if it acquires a critical section mutual exclusion lock, is currently the highest priority operation and the mask in not already in use by another operation. If not successful, then the local mask variable (called local_bit_vector above) is set to zero. The all_reduce function combines all the local bit vectors using the binary “and” operator and returns the resulting global bit vector to all involved processes. The find_and_allocate function determines the position of the least significant bit that is set to one, sets the bit in the corresponding position in the shared state availability mask and returns the integer value of that position. This integer position is the newly allocated context id. If all bits are set to zero then zero (an invalid value for the new context id) is returned and the algorithm repeats.

For inter-communicators, the local and remote groups each allocate a context id separately. There is an extra step at the end of the algorithm whereby each context id is broadcast to all members of the other group and both context ids are stored in the communicator handle structure. Thus, for the context id allocation algorithm, the inter-communicator case resolves to be identical to the intra-communicator case plus some subsequent communication using the parent communicator.

The non-blocking versions follow the same control flow as the blocking versions but perform each step asynchronously using a scheduling mechanism and use a non-blocking reduction operation. In addition, a non-blocking barrier is added to the schedule between each step.

In the absence of endpoints, it can be correctly assumed that each invocation of the context id allocation function in a particular process must have been caused by a different communicator creation operation. This means that each invocation must return a different context id. However, with the introduction of endpoints, some invocations of the context id allocation function must get the same resulting context id as some other invocations. Specifically, for communicator creation operations that are collective over multiple endpoints in a single operating system process, multiple endpoints must initiate the communicator creation operation and they must all obtain the same resultant context id.

For each communicator creation operation, the first endpoint that initiates the current context id allocation algorithm will correctly obtain control of the availability bit-vector. However, when subsequent endpoints initiate the context id allocation algorithm they will discover that the availability bit-vector is already in use. They will still participate in the reduction operation but will supply a local availability bit-vector with all bits set to all zero. The reduction will produce a global availability bit-vector with all bits set to all zero and no context id will be allocated. Control of the availability bit-vector will be given up and then acquired by exactly one of the endpoints (not necessarily the same one as last time). This will cause live-lock; all endpoints are active but with no hope of making meaningful progress.

### III. Fixing the MPICH Context ID Algorithm

Constraining the reduction operation so that only one endpoint per OS process participates in the communication would remove the cause of live-lock. The group used for the reduction operation would need to be constructed; it must contain exactly one endpoint per OS process. For MPI_COMM_CREATE_GROUP, this sub-group would need to be created each time; for all other communicator creation routines this sub-group could be constructed once at the time of creating the parent communicator. This approach requires communicating the value of the new context id from the endpoint that participates in the reduction operation to the other local endpoints that did not. This could be achieved via a local broadcast, if all the local endpoints can independently determine which one will be the root. The simplest approach here would be for the endpoint with the lowest rank in the parent communicator to be the only one that attempts to take control of the availability bit-vector and participate in the reduction operation with other processes. All other local endpoints would proceed directly to the local broadcast operation. The blocking version of the algorithm would use a blocking broadcast whereas the non-blocking version would schedule a non-blocking broadcast. This is similar to the inter-communicator scenario where each “leader” MPI process broadcasts the remotely allocated context id to all MPI processes in their local group.

An alternative approach is to enable all local endpoints to be active in the reduction communication with a valid copy of the availability bit-vector. The concurrency requirement stipulated in the endpoints proposal for all collective operations means that the MPI library can rely on the concurrent participation of all necessary endpoints. The changes introduced into the MPICH algorithm to support the MPI_COMM_CREATE_GROUP routine added an initial barrier that ensures all necessary MPI processes are concurrently active before any of them attempts to take control of the availability
bit-vector. After this barrier completes, all of the endpoints are able to concurrently take control of the bit-vector. This can be achieved by replacing the Boolean flag that indicates if the bit-vector is owned with an integer reference count that indicates the number of owners. If the current operation has highest priority and the bit-vector is not currently owned (i.e. the reference count is zero) then it can take ownership and make a local copy of the availability bit-vector from shared state. This behaviour is exactly the same as the current algorithm:

```c
if(owners==0) {
  if(context_id==lowestContextId && tag==lowestTag) {
    i_am_owner=1; ownerTag=tag;
    owners++; ownerContextId=context_id;
    local_bit_vector=shared_bit_vector;
  } else {
    i_am_owner=0;
    local_bit_vector=all_zeros_vector;
  }
}
```

If the bit-vector is already owned, the current algorithm does not gain ownership and sets its local bit-vector to zero. The first priority to take ownership will have recorded information (context id and tag) sufficient to uniquely identify the communicator creation operation. If the context id and tag for the current operation match that of the existing owner, then this is another endpoint from the same operation and it can be permitted to take concurrent ownership:

```c
if (owners > 0) {
  if (context_id==ownerContextId && tag==ownerTag) {
    i_am_owner=1; owners++; ownerTag=tag;
    owners++; ownerContextId=context_id;
    local_bit_vector=shared_bit_vector;
  } else {
    i_am_owner=0;
    local_bit_vector=all_zeros_vector;
  }
}
```

Having set the local bit-vector, each endpoint then participates in a reduction operation. All local endpoints for the highest priority communicator creation operation supply the same valid, non-zero bit-vector as each other to this reduction operation. This allows the reduction operation to produce a non-zero bit-vector and a valid context id can be allocated. If a context id is successfully allocated, the joint ownership of the availability vector must be given up to allow future attempts to take ownership by as yet incomplete communicator creation operations. If a context id is not successfully allocated, the joint ownership of the availability vector must still be given up in case a higher priority operation is waiting to take ownership. Each endpoint must decrease the reference count to release its ownership and, before continuing, must wait for the reference count to reach zero, e.g. by performing a barrier operation. This barrier protects against the race-condition caused by an endpoint releasing ownership and then immediately attempting to re-take ownership while other endpoints still have ownership, which would cause the ownership to be retained even if a higher priority operation is waiting. The following shows how the joint ownership can be safely released: The barrier does not need to be a full barrier; it could be a local barrier only, i.e. a barrier that is collective only over a group containing all local endpoints but no endpoints from other OS processes

```c
if(i_am_owner) {
  if(global_bit_vector != 0) {
    new_context_id=
    find_and_allocate(global_bit_vector);
    if(context_id==lowestContextId && tag==lowestTag) {
      lowestContextId = MAXINT;
    }
  }
  i_am_owner=0; owners--;
  if(owners==0) {
    if(context_id==ownerContextId && tag==ownerTag) {
      ownerContextId = MAXINT;
    }
  }
  barrier();
}
```

IV. PROTOTYPE IMPLEMENTATION IN McMPI

A fully-functional prototype implementation of the algorithm with all threads active was implemented in the McMPI library [8]. The scalability of the context id allocation algorithm has not been changed by the addition of support for multiple MPI endpoints – it is dominated by the scalability of the constituent communication operations, i.e. global reduction and barrier, which are known to be implementable in a scalable manner.

Preliminary results show that no additional overhead is introduced to point-to-point communication via a communicator that has multiple local endpoints relative to a communicator that does not. This was expected for the McMPI implementation because McMPI already supported multiple MPI processes being hosted by a single OS process prior to the changes necessary for complying with the proposed specification of MPI endpoints. No code changes to the internal messaging protocols or network communications were needed so no performance changes were expected.

V. CONCLUSION

The current algorithm used by the MPICH library for context id allocation will not work correctly if the parent communicator has multiple local MPI endpoints. However, the algorithm can be enhanced to provide support for MPI endpoints without affecting the scalability of the communicator creation functions and without affecting the communication performance of the MPI library. A fully-functional prototype of one viable approach has been implemented in the McMPI library.

Further work on the MPICH implementation of context id allocation is required to prove the correctness of the algorithm and to solve or mitigate outstanding known issues. Work on the OpenMPI algorithm for context id allocation is on-going.

ACKNOWLEDGMENT

This work was funded by the European Commission through the EPIGRAM project (grant agreement no. 610598.
REFERENCES


