MPI3: The Behavior of MPI Processes Sharing Memory

Hybrid Programming Working Group

December 12, 2011

0.0.1 Rationale

In most MPI implementations, each MPI process runs in a separate address space. However, the MPI standard does not require this. It says [3, §2.7]:

> An MPI program consists of autonomous processes, executing their own code, in a MIMD style. The codes executed by each process need not be identical. The processes communicate via calls to MPI communication primitives. Typically, each process executes in its own address space, although shared-memory implementations of MPI are possible.

In fact, there are MPI implementations where an MPI process is mapped onto an OS thread [1, 4], or even a task that can be dynamically scheduled by a run-time on different threads, for load balancing [2].

As the number of cores per node increases, there is renewed interest in such MPI implementations: It is often desirable to have more than one MPI process per node. For example, message matching time tends to increase linearly with the number of pending requests; the problem is inherent to the matching semantics of MPI. Multiple MPI processes will usually support a higher message rate. The problem becomes more acute as the number of concurrent threads per node increases. Similarly, a large node may have multiple network interfaces; using them for one MPI process will increase MPI overheads and may result in superfluous communication in a NUMA system. On the other hand, it is often desirable to use shared memory for intra-node communication, as this reduces communication overheads and avoids the replication of shared data. In order to achieve both goals, it is convenient to run multiple MPI processes in the same address space.

This hybrid model (multiple MPI processes within the same address spaces) also facilitates the efficient support of hybrid programming models such as MPI+OpenMP or MPI+PGAS: One may desire one OpenMP program with multiple MPI processes, or one UPC program where each UPC thread is an MPI process, but multiple UPC threads run in the same address space.

This section clarifies the semantics of multiple MPI processes running within the same address space. In addition, it provide a mechanism for an MPI program to figure out which MPI processes are running within the same address space.

0.0.2 Definitions

We use the following terminology

An MPI endpoint consists of a set of resources that enable MPI calls. Such an endpoint is identified in MPI by a rank in a communicator.

An MPI process consists of an MPI endpoint and a set of threads that can perform MPI calls using that endpoint. We say that the threads are *attached* to the endpoint they use in MPI calls.

All references to a "process" in other parts of the MPI standard should be understood as referring to an "MPI process".

> *Discussion.* There is no necessary connection between MPI processes and any other notion of process that a system may support; thus, an "MPI process" could actually be associated with a OS thread and an OS process could contain multiple MPI processes.

Similarly, no assumptions are made about threads, except that (i) a thread that executes a blocking MPI call is "descheduled", i.e., cannot prevent the progress of other threads; and (ii) when the blocking MPI call is complete the blocked thread is eventually "rescheduled", i.e., resumes running the code following the call. (*End of discussion.*)

0.0.3 Initialization

The total number of available endpoints and their locations are determined by the MPI job startup command, e.g., mpiexec; see Section 0.0.4.

After MPI is initialized, at least one thread is attached to each endpoint. A spawned thread is attached to the same endpoint as its parent.

0.0.4 Clarifications to MPI Standard

Unless said otherwise, whenever "process" is mentioned in the MPI standard, it should be understood to mean "MPI process", i.e., an endpoint and all the threads attached to that endpoint. The rules listed below follow from this interpretation.

A thread must be attached to an endpoint (i.e. must be part of an MPI process) in order to make MPI calls other than MPI_INIT, MPI_INIT_THREAD, MPI_GET_VERSION and MPI_INITIALIZED. A nonblocking call started on an endpoint must be completed on the same endpoint.

The rules and restrictions specified by the MPI standard [3, §12.4] for threads continue to apply. In particular, a blocking MPI call will block the thread that executes the call, but will not affect other threads. The blocked thread will continue execution when the call completes. Since each thread can be attached to only one endpoint, deadlock situations do not arise. Two distinct threads should not block on the same request.

Progress

The MPI standard specifies situations where progress on an MPI call at an agent might depend on the execution of matching MPI calls at other agents. Thus, a blocking send operation might not return until a matching receive is executed; a blocking call to a collective operation call might not return until the call is invoked by all other processes in the communicator. On the other hand, a non-blocking send or non-blocking collective will return irrespective of activities at other MPI processes.

These rules are extended to the situation where multiple MPI processes run in the same address space: a blocking send on an endpoint might not return until a matching receive has occurred at the destination endpoint; and a blocking call to a collective operation might not return until the operation is invoked at all endpoints of the communicator (including endpoints in the same address space). On the other hand, a non-blocking send or a non-blocking call to a collective operation will return, irrespective of the activities of threads attached to other endpoints (including threads in the same address space).

Initialization and Finalization

The initialization functions MPI_INIT or MPI_INIT_THREAD must be invoked by all threads that will use MPI, or will have descendants that will use MPI. The function MPI_FINALIZE must be invoked once on each endpoint.

Rationale. The association of threads to endpoint may not be determined before the intialization call. (*End of rationale.*)

Discussion. This is a slight change to previous MPI standard, as MPI may be initialized multiple times in the same MPI process. (*End of discussion.*)

Thread Support

If the level of thread support is MPI_THREAD_FUNNELED then only one thread can execute MPI calls on each endpoint; different threads within the same address space can execute concurrently MPI calls on different endpoints. If the level of thread support is MPI_THREAD_SERIALIZED then no two threads can make concurrent MPI calls on the same endpoint.

Memory Allocation

Memory allocated by MPI_ALLOC_MEM [3, §6.2] can be used only for communication with the endpoint the calling thread is attached to.

Rationale. Endpoints may be supported by distinct adapters, each requiring different memory areas for efficient communication. (*End of rationale.*)

Error Handling

Error handlers are attached to endpoints. A communicator may have different error handlers attached to the different endpoints of that communicator within the same address space.

The same rule applies to error handlers attached to windows or files.

Process Manager Interface

The functionsMPI_COMM_SPAWN and MPI_COMM_SPAWN_MULTIPLE can be used to spawn new endpoints. The argument argument maxprocs specifies the number of new endpoints to be spawned. A thread is attached to each newly spawned endpoint.

A new key multiplicity is reserved for the info argument of MPI_COMM_SPAWN and MPI_COMM_SPAWN_MULTIPLE. The value is a number specifying how many endpoints are created within each address space. The value of multiplicity must divide the value of maxprocs. If a value is not specified for mltiplicity then it is assumed to be equal to one.

Implementation are not required to recognize the multiplicity key. An implementations that recognizes it will return an error of class MPI_ERR_SPAWN if it cannot generate the required number of endpoints per address space.

Portable Process Startup

If the implementation provides an mpiexec function then the argument n specifies the total number of endpoints, i.e., the size of the group of MPI_COMM_WORLD. The argument multiplicity specifies the number of endpoints per address space.

Windows

An invocation to MPI_WIN_CREATE(base, size, disp_unit, info, comm, win) may return a different window for different endpoints in the same address space. Windows associated with different endpoints in the same address space may overlap. However, the outcome of a code where conflicting accesses occur to a location that appears in two windows is undefined.

I/O

The invocation to MPI_FILE_OPEN returns a distinct file handle at each endpoint. Note that the function is collective and must be invoked at all endpoints of the communicator with the same file name and access mode arguments.

An invocation to MPI_FILE_SET_VIEW can set a different view of the file for each file handle argument (passing different disp, filetype or info arguments) – hence a different view at each endpoint within the same address space.

Data access calls that use individual file pointers (such as MPI_FILE_READ) maintain a distinct file pointer for each file handle; hence different endpoints within the same address space are associated with distinct individual file pointers.

0.0.5 Identifying Endpoints within Same Adress Space

A new predefined type is added: MPI_COMM_TYPE_ADDRESS_SPACE. A call to MPI_COMM_SPLIT_TYPE(comm, split_type, key, info, newcomm), with split_type equal to MPI_COMM_TYPE_ADDRESS_SPACE will create a new communicator for each subgroup of endpoints that are in the same address space.

Bibliography

[1] E.D. Demaine. A threads-only MPI implementation for the development of parallel programs. In *Proceedings of the 11th International Symposium on High Performance Computing Systems*, pages 153–163. Citeseer, 1997. 0.0.1

[2] C. Huang, O. Lawlor, and L.V. Kale. Adaptive mpi. *Lecture notes in computer science*, pages 306–322, 2003. 0.0.1

[3] MPI Forum. MPI: A Message-Passing Interface Standard V2.2, 2009. 0.0.1, 0.0.4, 0.0.4

[4] H. Tang, K. Shen, and T. Yang. Program transformation and runtime support for threaded MPI execution on shared-memory machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(4):673–700, 2000. 0.0.1