Problem

How to involve multiple threads in a single collective (or any single communication).

Terms Used

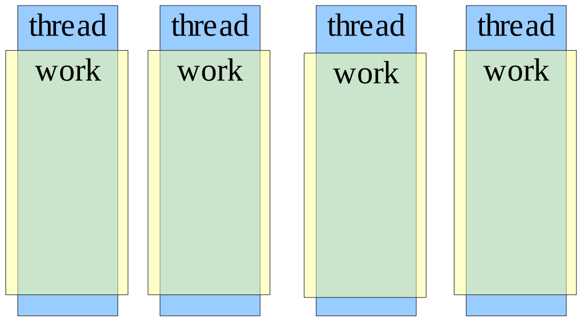
Message layer: MPI + implementation.

Program: Codes that are linked with the message layer to form an executable job.

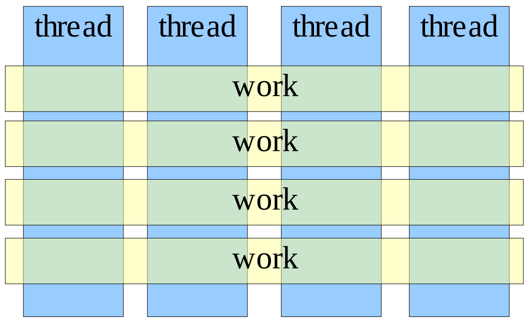
Processor resources: cores, hardware threads, etc. The hardware component of a "thread".

Vertical parallelism: one communication ("work") is performed using one thread. Parallelism is achieved by performing several different communications at the same time, each using different threads.

Horizontal parallelism: one communication is spread out across several threads, each thread performing some part or role of the larger communication.



Vertical Parallelism



Horizontal Parallelism

Motivations

For communications hardware that supports such things as DMA, parallelism through processor resources is not beneficial since setting up and posting a DMA descriptor (for example) can not be parallelized. However, some communications require more software support such as packing/unpacking non-contiguous data or performing combine/multicast operations on local (intranode) data buffers. In addition, some hardware may require more software support, such as copying packets out of a receive FIFO (or even hardware that does not support DMA). Also, use of standard network stacks may introduce more software overhead, for example TCP/IP Sockets. When a communication involves large amounts of data, it may also benefit from striping (for example, to spread out the data movement over multiple memory channels).

All of these software overheads present opportunities for horizontal parallelism. But exposing these nuances to the program for it to parallelize require the program to understand a great deal about not only the hardware platform but also the mode in which the program was execute (e.g. Virtual Node Mode). This makes it desirable for the message layer to decide how and when to use additional processor resources to parallelize communications.

Processor resources are limited, however. Oversubscribing the processor resources in HPC is generally not done because of the overhead (and other side-effects) of preemption and context switching. In addition, current HPC programming paradigms do not expose methods for programmers to communicate their (potentially competing) intentions about use of threads (or sharing/lending of threads) with the message layer, making it difficult (if not inappropriate or even impossible) for the message layer to usurp processor resources for communications. A better approach would be to provide more interaction between the program and the message layer with respect to which processor resources may be used during what times. This is the direction taken in the current draft proposal "MPI3: Hybrid Programming", whereby "endpoints" represent message layer resources or objects (potential points of parallelism) which the program then associates with threads (creating an "agent") and provides those threads to the message layer when the agent makes MPI calls. This proposal works well for vertical parallelism.

As an example, consider an Allreduce with multiple local contributions, probably the most compelling case. To further illustrate, consider a platform where no collective network support exists (or the collective network cannot be used for local contributions). The allreduce must combine the local data, then inject that data onto the network, receive the data and combine (depending on the algorithm, repeating as necessary), then copy the results to all local contributors. There are three obvious stages for horizontal parallelism here.

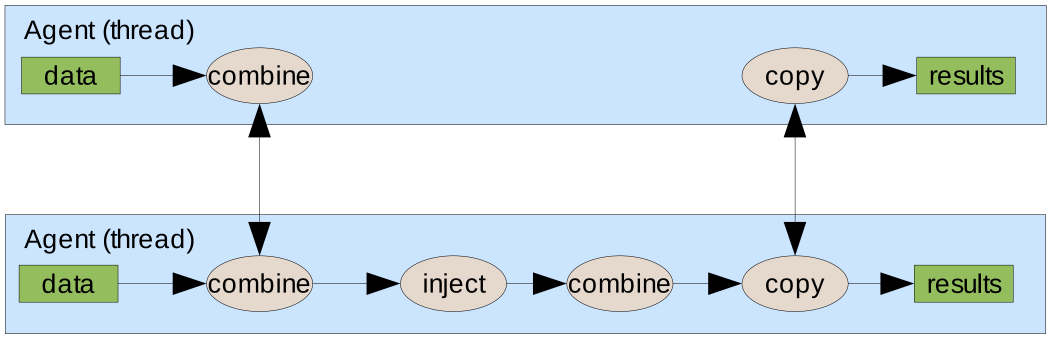


Figure 1

In figure 1, each (single-thread) agent makes the call to MPI_Allreduce. One thread takes responsibility for handling the (global) network while both share work for the local combine and broadcast (copy). In the case of a large amount of data, both threads are overloaded and will be dividing their time between local combine, local copy, and for one global inject/receive/combine. Since (most of) these operations are CPU-intensive, they would benefit from additional threads.

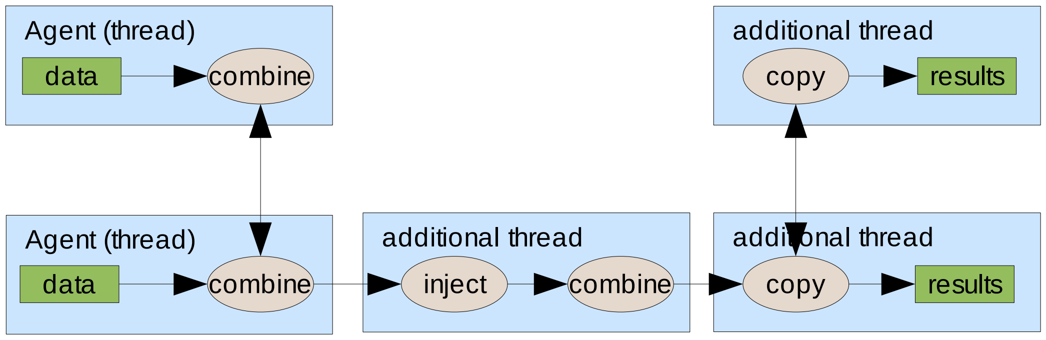


Figure 2

In figure 2, additional threads are used to relieve the agents of some of the work. In this case, the agents which called MPI_Allreduce work on the local combine operation while other threads manage the network and local broadcast stages.

Background

Some examples for horizontal parallelism are:

1) a software-driven collective network, where injection and reception must be performed by separate threads in order to realize maximum throughput.

2) A collective on a hierarchical set of networks where threads perform/manage separate network stages and data (intermediate results, etc) is pipelined between the stages.

3) Another way to parallelize a collective is to stripe the data over multiple threads. In this case, each thread performs the same collective operation but uses an isolated chunk of data. Note, this sort of parallelism could be performed by either the program (using vertical parallelism from the message layer's perspective) or by the message layer. The reason it might make sense for the message layer to perform it is that the message layer would have more information about the resources available in the hardware and thus be better able to determine when, and how much, parallelism makes sense.

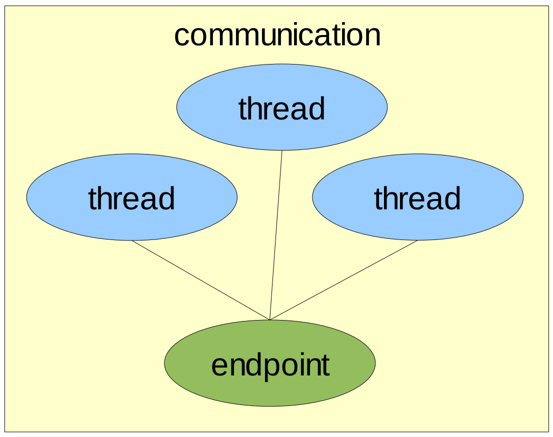
In these examples, each thread is performing one role of a larger, single, collective. The program would view the operation as a single collective, but multiple threads are required to get optimal performance. In order for the program to lend threads to the collective, more than one thread would need to participate in some way. Additionally, the programming paradigm must either support methods by which a program can determine how many threads should participate or must otherwise naturally result in adequate participation to ensure all work is completed.

In addition, the algorithm(s) used by a collective internally are not exposed to the programmer. But, the specifics of the algorithm(s) may have direct bearing on how and what parallelism is possible (or optimal). The processor resources that a program "lends" to the message layer need to be general-purpose, i.e. the program should not make any assumptions about how (or even if) the message layer uses them. All of this impacts how (the threads of) a program waits for completion of communications.

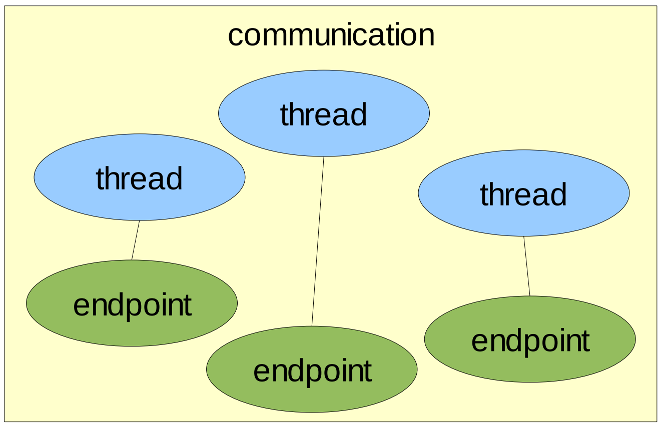
Also note that the same types of parallelism discussed for collectives here could also apply to point-to-point communications. It would seem reasonable to extend any proposed solutions to point-to-point as well as collectives.

Discussion

For horizontal parallelism, there needs to be a formalization of how the program can associate multiple threads with a single communication (collective). Since the proposal for MPI3 Endpoints associates one or more threads with an endpoint, and endpoints seem to be the unit of parallelism, the issue seems to be how to associate multiple endpoints with a single communication. Additionally, there should be some formal method by which the program can decide whether to use multiple endpoints and how many, and then ensure the message layer understands that decision. Here are some possible methods:



Multiple threads per endpoint



Multiple endpoints per communication

I) Multiple threads per endpoint. More than one thread attaches to an endpoint in order to participate in a communication.

II) Multiple endpoints per communication. One thread in each agent makes the communication call, they all work on the same communication and provide parallelism.

Both methods have some difficulty differentiating between multiple endpoints participating in a single communication and multiple endpoints performing similar (but distinct) communications at the same time. Here are some ways to handle this:

> A) The program explicitly involves endpoints in a communication. For example, multiple agents could all make "the same" collective call providing some parameter to indicate that they are participating together on the same set of data rather than each providing their own data. If the collective were

non-blocking, each agent would have a different request on which to wait. This probably requires changes to all MPI communications APIs in order to incorporate the extra parameters for grouping related calls.

B) The message layer simply uses whatever endpoints it wants. The program must ensure that all endpoints make progress while waiting for communication(s) to complete. This also requires some degree of coordination between the agents since each may be responsible for communications that it did not start. This becomes difficult since MPI waits on requests and these additional participating agents would not know about the extra requests that they are implicitly responsible for. Blocking communications are also difficult.

C) The program will arrange for "idle" threads (those that have completed current computation) to make a special call (e.g. "attach helper") or otherwise "join" the group of threads which will work on the communication. This in affect lends the thread/endpoint to the message layer. The message layer can then use only endpoints (threads) which are part of the group.

There are several potential solution discussions in MPI Forum tickets:

MPI Forum Ticket #208	A variation of (II) with (C)
MPI Forum Ticket #209	(A), either (I) or (II)
MPI Forum Ticket #210	A variation of (I) with (C)
MPI Forum Ticket #211	A variation of (II) with (C)
MPI Forum Ticket #212	(B)