JEFF SQYURES

More Joys of Asynchronous Communication

Halloween is coming up soon. That means scary masks and sugar candy: a combination sure to elicit an explosive response in even the most demure children. Hmm. Interesting concept: add a little something to a pre-existing entity and get higher output performance...

The Story So Far

Last month we started discussing non-blocking communication (get it?). We covered the basic non-blocking (or *immediate*) send and receive functions — all of which start a communication — and touched on their various flavors. We also discussed the `TEST` and `WAIT` functions, and how they are used to *complete* communications.

Recall that previous articles have only covered *standard* communication (sometimes called "blocking" communication, even though the functions may not always block!): functions that will not return until MPI guarantees that the buffer can be [re]-used. Using non-blocking communications effectively allows the separation of communication initiation and completion, and allows for the possibility of communication and computation overlap.

This month, we'll talk more about non-blocking methods and benefits, and fuel the fire with some more examples about how and why they can be useful to your MPI application. And remember, latency is like a good speech; the shorter, the better.

Persistent Sends and Receives

Another form of non-blocking communication is MPI's *persistent* messages. Persistent communication offers a slight optimization to applications that repeatedly send or receive a buffer with the same message signature. In such cases, the use of persistent communication can reduce overall latency.

The rationale is to pass all the arguments (buffer, count, datatype, tag, source/destination, and communicator) and perform the setup required for the communication only *once*. Then, in each iteration of the application, simply say "go" on the previously set-up operation and let the communication commence. For example:

```
1 MPI_Status status;
2 MPI_Request req;
3 MPI_Send_init(buf, count,
                dtype, dest,
                tag, comm, &req);
4 while (looping) {
5  MPI_Start(&req);
6  do_work();
7  MPI_Wait(&req, &status);
8 }
9 MPI_Cancel(&req);
```

The `MPI_SEND_INIT` function creates a request and sets up the communication. Its signature is identical to `MPI_ISEND` (all the normal sending parameters and the address of an `MPI_Request` to fill). The `MPI_START` function actually starts the communication operation. The send is a non-blocking operation and therefore must be finished with a `TEST` or `WAIT` operation. During the next iteration, there is no need to invoke `MPI_SEND_INIT` again — we simply `START` and `WAIT` the request. After the loop has completed, it is proper to `MPI_CANCEL` a persistent request. This command tells MPI that the application will not use that request again — it is safe to destroy and free all associated resources.

`MPI_SEND_INIT` is a standard mode persistent send; `MPI_SSEND_INIT`, `MPI_BSEND_INIT`, and `MPI_RSEND_INIT` are the synchronous, buffered, and ready mode persistent functions, respectively. `MPI_RECV_INIT` is the persistent receive. They all function similarly to `MPI_SEND_INIT`: use the INIT function to create the request, use the `START` function to initiate the communication, and finally use some flavor of `TEST` or `WAIT` to complete it. Also note that just like the `TEST` and `WAIT` functions, `START` has a variant that can operate on an array of requests: `MPI_STARTALL`.

Why Bother With Non-Blocking?

Invoking special functions and creating additional logic for splitting the initiation and completion of communications can be quite a hassle. Why bother?

For example, a communication co-processor, separate from the main CPU, can process message-passing events independently of the operating system and user's application. A coprocessor allows even single-threaded MPI implementations to perform some communication asynchronously, while the application is executing outside the MPI library. The network itself takes responsibility for some portion of MPI semantics. Additionally, standard mode functions allow only one communication to occur at a time. Non-blocking functions allow the application to initiate multiple communication operations, enabling the MPI implementation to progress them simultaneously. Consider the following code example:

```
1 while (looping) {
2  if (i_have_a_left_neighbor)
```

```
3   MPI_Recv(inbuf, count,
             dtype, left,
             tag, comm,
              &status);
4  if (i_have_a_right_neighbor)
5   MPI_Send(outbuf, count,
             dtype, right,
             tag, comm);
6  do_other_work();
7 }
```

Assume at that least one process does not have a left neighbor, and consider how this code will run in parallel: every process will receive from its left and then send to its right. But notice that the above code uses standard mode sends. As a direct result, this algorithm is actually serialized — it will execute in a domino-like fashion, causing each process to block while waiting for its left neighbor.

Using non-blocking communication allows the MPI to progress both communications simultaneously:

```
1 while (looping) {
2  count = 0;
3  if (i_have_a_left_neighbor)
4    MPI_Irecv(inbuf, count,
               dtype, left,
               tag, comm,
               &req[count++]);
5  if (i_have_a_right_neighbor)
6    MPI_Isend(outbuf, count,
               dtype, right,
               tag, comm,
               &req[count++]);
7  MPI_Waitall(count, req,
   &statuses);
8  do_other_work();
9 }
```

The `MPI_WAITALL` on line 7 allows *both* communications to progress simultaneously. Specifically, the send can proceed before the receive completes. This code will therefore operate in a truly parallel fashion and will avoid the domino effect. Note, however, that this particular code example has a subtle implication: the `WAITALL` will block until both communications are complete. Indeed, the astute reader will recognize that a clever use of `MPI_SENDRECV` could be used for the same result. Specifically, blocking on line 7 means that there still may be some "dead" time while waiting for network communication to complete — time that could have been used for other work. This situation may be unavoidable in some applications, but others may have some work that can be performed while waiting for the communications to complete. For example:

```
1 while (looping) {
2  count = 0;
3  if (i_have_a_left_neighbor)
4    MPI_Irecv(inbuf, count,
               dtype, left,
               tag, comm,
               &req[count++]);
5  if (i_have_a_right_neighbor)
6    MPI_Isend(outbuf, count,
               dtype, right,
               tag, comm,
               &req[count++]);
7  do_some_work();
8  MPI_Waitall(count, req,
   &statuses);
9  do_rest_of_work();
10 }
```

Note the addition of `do_some_work()` and `do_rest_of_work()` on lines 7 and 9, respectively. `do_some_work()` represents work that can be done before the communication completes. Hence, the application can even utilize the "dead" time while message passing is occurring in the background — an overlap of communication and computation. This method works best on networks and/or MPI implementations that allow for at least some degree of asynchronous progress, but can even benefit single-threaded, synchronous MPI implementations. Once the communication completes, `do_rest_of_work()` executes, and one assumes it is performing work that was dependent upon the received messages.

Note that since the same buffers and communication parameters are used in every iteration, a further optimization could use the persistent mode. This improvement allows the MPI to setup the communications once, and simply say "go" every iteration:

```
1 int count = 0;
```

Pre-Posting Receives

Just because an operation is non-blocking does not mean that it is somehow automatically more efficient than if it were blocking. Indeed, many of the same best practices that apply to blocking communication also apply to non-blocking communication. One such best practice judiciously pre-posting non-blocking receives. This method potentially helps an MPI implementation reduce the use of temporary buffers.

For example, if a message is received in an MPI process that is unexpected — meaning that the application did not [yet] post a corresponding receive — the MPI implementation may have to allocate a temporary buffer to receive it. If a matching receive is ever posted, the MPI implementation copies the message from the temporary buffer into the destination buffer.

However, if a non-blocking receive is posted before the message is received, once the message arrives, it is expected and can be received directly into the target buffer. No temporary buffer needs to be allocated and no extra memory copy is necessary. Hence, ensuring to pre-posting receives can increase the efficiency of an MPI application.

```
2 if (i_have_a_left_neighbor)
3   MPI_Recv_init(inbuf, count,
                  dtype, left,
                  tag, comm,
                  &req[count++]);
4 if (i_have_a_right_neighbor)
5   MPI_Send_init(outbuf, count,
                  dtype, right,
                  tag, comm,
                  &req[count++]);
6 while (looping) {
7   MPI_Startall(count, req);
8   do_some_work();
9   MPI_Waitall(count, req,
                &statuses);
10  do_rest_of_work();
11 }
```

All production-quality MPI implementations can handle simultaneous progress of multiple requests, even those that do not allow true asynchronous progress. Hence, even if polling (via `TEST` operations) is required, non-blocking communication programming models can still represent a large performance gain as compared to standard/blocking mode communication.

MPI-2: Multiple Types of Requests

MPI-2 defines two new types of operations that can be started and completed using `MPI_Request` handles: parallel I/O and user-mode "generalized" requests. Although those operations are the topics for future columns, suffice it to say that both of them follow the same general model as non-blocking point-to-point communication: actions are started with calls to MPI functions that generate requests and are completed with calls to `TEST` or `WAIT` operations.

A subtle implication is that the array-based `TEST` and `WAIT` functions can accept multiple `MPI_Request` handles regardless of the type of pending operation that they represent. Hence, it is possible to create an array of requests that encompasses both point to point and I/O communications, and have `MPI_WAITALL` wait for the completion of all of them.

Where to Go From Here?

Non-blocking communications, when used properly, can provide a tremendous performance boost to parallel applications. In addition to allowing the MPI to perform at least some form of asynchronous progress (particularly when used with communication co-processor-based networks), it allows the MPI to progress multiple communication operations simultaneously.

Got any MPI questions you want answered? Wondering why one MPI does *this* and another does *that*? Send them to *jsquyres@lam-mpi.org*.

Jeff Squyres is a post-doctoral research associate at Indiana University and is the lead developer for the LAM implementation of MPI. He can be reached at jsquyres@lam-mpi.org

ROMIO: A Popular MPI-2 I/O Implementation

ROMIO is a popular implementation of many of the MPI-2 I/O function calls from Argonne National Laboratory (e.g., `MPI_FILE_OPEN`, `MPI_FILE_READ`, etc.). ROMIO's implementation is layered on top of MPI-1 point-to-point communication; it is specifically designed as an add-on to existing MPI implementations (such as LAM/MPI, LA-MPI, FT-MPI, and MPICH, to name a few). This layering creates problems because ROMIO cannot re-define the underlying type `MPI_Request` since it has already been defined by the underlying MPI implementation. Moreover, the back-end of `MPI_Request` is different in every MPI implementation; ROMIO can't extend it in a portable way.

ROMIO's solution was to create a new type: `MPIO_Request`. All `MPI_FILE*` functions that are supposed to take an `MPI_Request` as a parameter instead take an `MPIO_Request`. This situation means that ROMIO technically does not conform to the MPI-2 standard, but this detail is usually overlooked for the sake of portability and functionality.

There is a notable side effect, however. Since `MPI_TEST` and `MPI_WAIT` (and their variants) take `MPI_Request` arguments, they cannot accept ROMIO `MPIO_Requests`. Hence, ROMIO implements its own `MPIO_TEST` and `MPIO_WAIT` functions. As such, MPI implementations that use ROMIO generally do not support invoking the various `TEST` and `WAIT` functions with arrays of point-to-point and I/O requests

Resources

ROMIO: A High-Performance, Portable MPI-IO Implementation: www.mcs.anl.gov/romio

MPI Forum (including the MPI-1 and MPI-2 specification documents): www.mpi-forum.org

MPI — The Complete Reference: Volume 1, The MPI Core by Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra.

MPI — The Complete Reference: Volume 2, The MPI Extensions by William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir.

NCSA MPI tutorial webct.ncsa.uiuc.edu:8900/public/MPI