JEFF SQYURES

The Joys of Asynchronous Communication

Did you ever notice that Fred Flintstone was the only one who powered his car? You could see his feet running below the car, pushing it along, but no one else ever helped. This is clearly a single-inertia, multiple-dependents (SIMD) situation — it's almost like Fred was providing a taxi service for all the other freeloaders. Indeed, getting some of the others to help push would have resulted in greater efficiency — a multiple-inertia, multiple-dependents (MIMD) scenario. I call this the Flint's Taxi Economy principle. (Say it out loud.)

The Story So Far

So far, we've talked about a lot of the basics of MPI. Parallel startup and short, MPI terminology, some point-to-point and collective communication, datatypes, communicators... what on earth could be next? Is there really more?

Why yes, Virginia, yes there is. What we've talked about so far can be used to write a wide variety of MPI programs. But most of the previous discussion has been about blocking calls that, for example, may or may not execute in parallel to the main computation. Specifically, we've discussed *standard* mode sends and receives (`MPI_SEND` and `MPI_RECV`). Most communication networks function at least an order of magnitude slower than local computations. Hence, if an MPI process has to wait for non-local communication, critical CPU cycles are lost while the operating system blocks the process, waits for the communication, and then resumes the process. Parallel applications therefore typically run at their peak efficiency when there can be a true overlap of communication and computation.

The MPI standard allows for this overlap with several forms of non-blocking communication.

Immediate (Non-Blocking) Sends

The basic non-blocking communication function is `MPI_ISEND`. It gets its name from being an immediate, local operation. That is, calling `MPI_ISEND` will return to the caller immediately, but the communication operation may be ongoing. Specifically, the caller may not re-use the buffer until the operation started with `MPI_ISEND` has completed (completion semantics are discussed below). In all other ways, `MPI_ISEND` is just like the standard-mode `MPI_SEND`: think of `MPI_ISEND` as the starting of a communication, and think of `MPI_SEND` as both the starting and completion of a communication (remember that MPI defines completion in terms of buffers — the operation is complete when the buffer is available for re-use, but doesn't specify anything about whether the communication has actually occurred or not).

The C binding for `MPI_ISEND` is:

```
int MPI_Isend(void *buf,
              int count,
              MPI_Datatype dtype,
              int dest,
              int tag,
              MPI_Comm comm,
              MPI_Request *req);
```

You'll notice that `MPI_ISEND` takes the same parameters as `MPI_SEND` except for the last one: a pointer to an `MPI_Request`. The MPI implementation will fill the request with a handle that can be used to test or wait for completion of the send. More on that later.

"Immediate" also applies to MPI's other modes of sending: synchronous, buffered, and ready. Synchronous sends will not complete until a matching receive has been posted at the destination process. Hence, when a synchronous send completes, not only is the message buffer available for re-use, the application is also guaranteed that at least some communication has occurred (although it does not guarantee that the entire message has been received yet).

Buffered sends mean that the MPI implementation may copy the message to an internal buffer before sending it. See the sidebar entitled "Buffered Sends Are Evil" for more information on buffered mode sends. Ready sends are an optimization when it can be guaranteed that a matching receive has already been posted at the destination (specifically, it is erroneous to initiate a ready send if a matching receive has not already been posted). In such cases, the MPI implementation may be able to take shortcuts in sending protocols to reduce latency and/or increase bandwidth. Not all MPI implementation actually implement optimizations; in the worse case, a ready send is identical to a standard send.

Aside from their names, all three modes have identical bindings to `MPI_ISEND`: `MPI_ISSEND`, `MPI_IBSEND`, and `MPI_IRSEND` for immediate synchronous, buffered, and ready send, respectively.

Immediate (Non-Blocking) Receives

MPI also has an immediate receive. Just like the immediate send, the immediate receive starts a receive (or "posts" a receive). Its C binding is identical to `MPI_RECV` except for the last argument:

```
int MPI_Irecv( void *buf,
           int count,
           MPI_Datatype dtype,
           int src,
           int tag,
           MPI_Comm comm,
           MPI_Request *req);
```

Just like the immediate send functions, the final argument is a pointer to an `MPI_Request`. Upon return from `MPI_IRECV`, the request can be used for testing or waiting for the receive to complete.

Completion of Non-Blocking Communications

Once a non-blocking communication has been started (regardless of whether it is a send or a receive), the application is given an `MPI_Request` check for completion of the operation. Two operations are available to check for completion of a request: `MPI_TEST` and `MPI_WAIT`:

```
int MPI_Test( MPI_Request *req,
          int *flag,
          MPI_Status *status);
```

Who Matches What?

With all the different flavors of sending and receiving, how can you tell which send will be matched with which receive? Thankfully, it's uncomplicated. When a process receives a message, it looks for a matching receive *only* based on the *signature* of the message: the message's tag, communicator, and source and destination arguments. The sending mode (standard, synchronous, buffered, or read) and type (blocking, non-blocking, or persistent) are not used for matching incoming messages to pending receives.

Buffered Sends Are Evil

There are three buffered send functions: `MPI_BSEND`, `MPI_IBSEND`, or `MPI_BSEND_INIT`, corresponding to standard, immediate, and persistent modes, respectively. Unlike the other kinds of sends, these functions are *local* operations. If the send cannot complete immediately, the MPI implementation must copy the message to an internal buffer and complete it later. Regardless, the message buffer must be returned to the caller and be available for use.

This may sound like a great convenience — the application is free from having to perform buffer management. But it almost guarantees a performance loss; at least some MPI implementations *always* do a buffer copy before attempting to send the message. This adds latency to the message's travel time.

Whenever possible, avoid buffered sends.

```
int MPI_Wait( MPI_Request *req,
          MPI_Status *status);
```

Both take pointers to an `MPI_Request` and an `MPI_Status`. The request is checked to see if the communication has completed. If it has, the status is filled in with details about the completed operation. `MPI_TEST` will return immediately, regardless of whether the operation has completed (the flag argument is set to `1` if it completed); it can be used for periodic polling. `MPI_WAIT` will block until the operation has completed.

The `TEST` and `WAIT` functions shown above check for the completion of a single request; other variations exist for checking arrays of requests (e.g., checking for completion of one in the array, one or more in the array, or all in the array): `MPI_*ANY`, `MPI_*SOME`, `MPI_*ALL`, respectively, where `*` is either `TEST` or `WAIT`.

Some Examples

Let's tie together all this information into a simple example:

```
1  MPI_Request req[2];
2  MPI_Status stat[2];
3  int rank, size, left, right;
4  MPI_Comm_rank(comm, &rank);
5  left = (rank + size - 1) % size;
6  right = (rank + 1) % size;
7  MPI_Irecv( &buffer[1],
               1, MPI_INT,
               right, tag,
               comm, &req[1]);
8  MPI_Isend( &buffer[0], 1,
               MPI_INT,
               left,
               tag,
               comm,
               &req[0]);
9  do_other_work();
10 MPI_Waitall(2, req, stat);
```

In this example, each process sends a single integer to the process on its right and receives a single integer from the process on its left (wrapping around in a torus-like fashion; it's left as an exercise for the reason to figure out the clever `left` and `right` calculations). Note that both the send and receive are *started* in lines 7 and 8, but are not *completed* until line 10, allowing the application to do other work on line 9.

The MPI standard states that once a buffer has been given to a non-blocking communication function, the application is not allowed to use it until the operation has completed (i.e., until after a successful `TEST` or `WAIT` function). Note specifically that the example above sends and receives to *different* buffers; since both communications are potential-

ly ongoing simultaneously, it important to give them different working areas to avoid race conditions.

It is also important to note that an MPI implementation may or may not provide asynchronous progress on message passing operations. Specifically, single-threaded MPI implementations may only be able to make progress pending messages while inside the MPI library. However, this may not be as bad as it sounds. Some types of networks provide communication co-processors that progress message passing regardless of what the application program is doing (e.g., InfiniBand, Quadrics, Myrinet, and some forms of TCP that have offload engines on the NIC). Although such networks can progress individual messages, periodic entry into the MPI library may still be necessary to complete the MPI implementation's communication protocols.

For example, consider that there is always "other work" to do in an application, and the total time required for this "other work" is always going to be more than what is required for communications. Lines 8-9 in the previous listing only allow one pass through the "other work" followed by waiting for all MPI communication to finish. This may be inefficient because all communication may be suspended during `do_other_work()` and only resumed during `MPI_WAITALL`. It may be more efficient to use a logic structure similar to:

```
1 while (have_more_work) {
2  do_some_other_work();
3  if (num_reqs_left  0) {
4   MPI_Testany(total_num_reqs,
                reqs, &index,
                &flag, stats);
5   if (flag == 1)
6    --num_reqs_left;
7  }
8 }
9 if (num_reqs_left  0)
10  MPI_Waitall(total_num_reqs,
                reqs, stats);
```

The rationale with this logic is to break up the "other work" into smaller pieces and keep polling MPI for progress on all the outstanding requests. Specifically, line 2 invokes a *small* amount of work and then uses `MPI_TESTANY` to poll MPI and see if any pending requests have completed. This process repeats, giving both the application and the MPI implementation a chance to make progress.

There are actually a lot of factors involved here; every application will be different. For example, if `do_some_work()` relies heavily on data locality, polling through `MPI_TESTANY` may effectively thrash the L1 and L2 cache. You may need to adjust the granularity of `do_some_other_work()`, or use one of the other flavors of the `TEST` operation to achieve better performance.

Resources

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

MPI — The Complete Reference: Volume 1, The MPI Core (2nd ed) (The MIT Press) by Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra.

MPI — The Complete Reference: Volume 2, The MPI Extensions (The MIT Press) 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

The moral of the story is to check and see if your MPI implementation provides true asynchronous progress or not. If it does not, then some form of periodic poll through a `TEST` operation may be required to achieve optimal performance. If asynchronous progress is supported, then additional polling may not be required. But unfortunately, there's no silver bullet: if you're looking for true communication and computation overlap in your application, you may need to tune its behavior with each different MPI implementation. Experimentation is key — your mileage may vary.

If you have a multi-threaded MPI implementation that does not support asynchronous progress, it may be more efficient to have a second thread block in `MPI_WAITALL` and let the primary thread do its computational work. L1 and L2 caching effects (among other things) will still affect the overall performance, but potentially in a different way. Threaded MPI implementations are a sufficiently complex topic that they will be discussed in a future column.

Where to Go From Here?

Non-blocking communications, when used properly, can provide a tremendous performance boost to parallel applications by allowing the MPI to perform at least some form of asynchronous progress (particularly when used with communication co-processor-based networks). Next month, we'll continue with a more in-depth look at non-blocking communications, including persistent mode sending and more examples of typical non-blocking communication programming models.

Jeff Squyres is a post-doctoral research associate at Indiana University and is the lead developer for the LAM implementation of MPI. E-mail him at jsquyres@lam-mpi.org.