JEFF SQUYRES

Is Your Application *spawnworthy*?

March — a month of celebration. It's the time when everyone around the world unites in joy and praise. I'm speaking of Pi Day, of course — on 3/14. All HPC users should contribute to the Pi revelry by computing and reciting as much of Pi as possible. What better way to do this than to optimize approximate computations of Pi in parallel?

The Story So Far

Last month we outlined the three models of dynamic processes in MPI: spawning new processes using `MPI_COMM_SPAWN` and `MPI_COMM_SPAWN_MULTIPLE`, client/server connections between existing MPI processes using `MPI_COMM_ACCEPT` and `MPI_COMM_CONNECT` (and supporting functions `MPI_OPEN_PORT`, `MPI_PUBLISH_NAME`, `MPI_LOOKUP_NAME`, and `MPI_CLOSE_PORT`), and using independently established sockets between existing MPI processes using `MPI_COMM_JOIN`.

All of these models are synchronous, meaning that they block until the action is completed. With some strong caveats about scheduled environments (discussed last month), the `SPAWN` functions will likely be completed more or less immediately (i.e., they will probably take as much time as an MPI implementation's job startup mechanism, such as `mpirun`). Hence, it will usually block for a short while, but complete in finite time. `JOIN`, while fundamentally asynchronous in nature, is likely to be used mainly in synchronous situations. Specifically, since a TCP socket must be established prior to invoking `JOIN`, the asynchronous aspects of connecting two previously existing processes are satisfied elsewhere, and `JOIN` will likely be invoked right after the socket has been established. So `JOIN` is also likely to be used in finite/time-bounded situations.

`ACCEPT` and `CONNECT`, however, are different. They are fundamentally asynchronous both in nature and use. The "server" process blocks in `ACCEPT` until a "client" process calls a corresponding `CONNECT`. Since the client process is likely to be independent of the server, it is effectively random as to when the client will invoke `CONNECT`. This situation can leave the server blocking indefinitely, and is unsuitable for most single-threaded applications/MPI implementations.

Threads to the Rescue

`ACCEPT` works best when it can be left blocking in an independent thread. This thread can simply loop over `MPI_COMM_ACCEPT`, accepting client connections and then dispatching them to other parts of the server upon demand. This method is actually quite similar to how many client/server applications are implemented. The server process can continue other meaningful work and be interrupted with client requests only as necessary.

A side effect of this approach (and the MPI design) is that the `ACCEPT` cannot be interrupted or killed cleanly. In order to shut down the server process, a dummy connection must be made to the server's pending `ACCEPT` (probably originating from within the server process itself) that issues a command telling the accepting thread to break out of its `ACCEPT` loop and die. This trick is necessary because it is illegal for an `ACCEPT` to be pending when another thread in the server invokes `MPI_FINALIZE`.

Note that not all MPI implementations support `ACCEPT`/`CONNECT` (or MPI-2 dynamic processes in general) and multi-threaded MPI applications. The MPI implementation that I work on, Open MPI, does, and is the basis for the examples provided in this column.

Disconnecting

Once communication between dynamic processes is no longer required, the function `MPI_COMM_DISCONNECT` can be invoked to formally break communication channels between the processes (see the "MPI Connected" sidebar). Connected processes impact each other in several ways; independent processes are unaffected by each other's run-time behavior (in terms of MPI semantics).

Hence, processes that are spawned are connected to their parents. Processes that establish communication via `CONNECT` and `ACCEPT` or `JOIN` are also connected.

To disconnect from another job, all groups referring to processes in that job must be freed. Groups spanning the two jobs may exist in communicators, file handles, or one-sided window handles (the later two are not discussed in this month's column). Hence, it may be necessary to free multiple handles (communicators, files, windows) before processes become independent of each other.

Note that communicators must be released via `MPI_COMM_DISCONNECT` instead of `MPI_COMM_FREE`. There is a subtle but important difference: MPI says that `MPI_COMM_FREE` only *marks* the communicator for deallocation and is guaranteed to return immediately; any pending communication is allowed to continue (and potentially complete) in the background. `MPI_COMM_DISCONNECT` will not return until all pending communication on the communicator has completed. Hence, when `DISCONNECT` returns, the communicator has truly been destroyed.

Concrete Example

Last month, I mentioned that many of the early uses of MPI-2 dynamic processes were rather mundane and usually unnecessary (e.g., launch a singleton `./a.out` that launches all of its peers). Now that threads can be mixed with MPI function calls, particularly with respect to dynamic process functionality, more interesting (and useful) options are available.

In short, MPI has previously been used mainly for *parallel* computing. With proper use of MPI-2 dynamic process concepts, MPI can be used for *distributed* computing.

For example, the canonical manager/worker parallel model is as follows: a manager starts a set of workers, doles out work to each of them, and waits for results to be returned. The send-work-and-wait-for-answers pattern is repeated until no work remains and all the results have been collected. The master then tells all workers to quit and everything shuts down.

However, consider reversing the orientation of model: the manager starts up and *waits* for workers to connect and ask for work. That is, workers start — and possibly shut down — independently of the manager. This concept is not new; it is exactly what massively distributed projects such as distributed.net and SETI@home (and others) have been doing for years. Although this has been possible in some MPI implementations for some time, only recently have some implementations started to make scalable, massively distributed computing a reality.

Consider a large corporation that has thousands of desktop computers. When the employees go home at night, the machines are typically powered off (or are otherwise idle). What if, instead, those machines could be harnessed for large-scale distributed computations? This goal has actually been the aspiration of many a CIO for years.

Corralling all the machines simultaneously to start a single parallel job is an enormous task (and logistically improbable, to say the least). But if a user-level MPI process on each machine started itself — independently of its peers — when the employee went home for the evening, the model becomes much more feasible. This MPI process can contact a server and join a larger computation and run all night. When the employee returns in the morning, the MPI process can disconnect from the computation (independently from its peers) and go back to sleep.

The model is also interesting when you consider the heterogeneous aspects of it: employee workstations may be one of many different flavors of POSIX, or Windows. A portable implementation of MPI can span all of these platforms, using the full power of C, C++, or Fortran (whatever the science/engineering team designing the application prefers) to implement the application on multiple platforms. MPI takes care of most of the heterogeneous aspects of data communication — potentially allowing the programmers to concentrate on the application (not the differences between platforms).

The server will need to exhibit some fault-tolerant characteristics. For example, it must be smart enough to know when to re-assign work to other resources because a worker suddenly became unavailable. However, these are now fairly well-understood issues (particularly in manager-worker models) and can be implemented in a reasonable fashion.

Granted, this model only works for certain types of applications. But it is still a powerful — and simple — concept that can is largely unexploited with modern MPI implemen-

See MPI, page 46

MPI "Connected"

MPI formally defines the communication status between two processes — they are either "connected" or "disconnected" (MPI-2 section 5.5.4):

Two processes are *connected* if there is a communication path (direct or indirect) between them. More precisely:

1. Two processes are connected if:
   (a) they belong to the same communicator (inter- or intra-, including `MPI_COMM_WORLD`) *or*
   (b) they have previously belonged to a communicator that was freed with `MPI_COMM_FREE` instead of `MPI_COMM_DISCONNECT` *or*
   (c) they both belong to the group of the same window or filehandle.
2. If A is connected to B and B to C, then A is connected to C.

Two processes are *disconnected* (also *independent*) if they are not connected.

As such, the state of being "connected" is transitive. This situation has implications for `MPI_COMM_ABORT` (used to abort MPI processes), run-time MPI exception handling, and `MPI_FINALIZE` (used to shut down an MPI process). `MPI_COMM_ABORT` and `MPI_ERRORS_ABORT` are allowed (but not required) to abort all connected processes. `MPI_FINALIZE` is collective across all connected processes. Hence, in order to ensure that processes do not unintentionally block in `MPI_FINALIZE`, it is a good idea for dynamic processes to `DISCONNECT` when communication between them is no longer required.

tations, mainly (I think) because people are unaware that MPI can be used this way.

Where to Go From Here?

It should be noted that there are research projects and commercial products that are specifically designed to utilize idle workstations. Condor, from the University of Wisconsin at Madison, is an excellent project that whose software works well, but is mainly targeted at serial applications (although recent efforts are concentrating on integrating Condor into grid computations). Several vendors have products that function similarly to distributed.net and SETI@home clients (a small daemon that detects when the workstation is idle and requests work from a server). This situation is also quite similar to what some people mean by the term "grid computing." However, none of these current efforts use MPI for their communication framework.

I want to be absolutely clear here: I am *not* saying that MPI is the answer to everyone's distributed computing problems. I am simply saying that the familiar paradigm of MPI can also be used for distributed computing. While the concepts for it may be relatively young in MPI implementations, the definitions in the standard make it possible, and support in MPI implementations is growing all the time (e.g., in Open MPI). I encourage readers to explore this concept and demand more from your MPI implementers.

Jeff Squyres can be reached at jsquyres@open-mpi.org.

Resources

Pi Day FAQ	mathforum.org/t2t/faq/faq.pi.html
Condor project	www.cs.wisc.edu/condor
Open MPI	www.open-mpi.org
MPI Forum	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. ISBN 0-262-69215-5.
- 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. ISBN 0-262-57123-4.

The real jump is with Bioinformatics