The Spawn of MPI

There once was a man from Nantucket
Whose PVM code kicked the bucket.
He ranted and raved,
"Oh, what can I save?"
Then he re-wrote his application with MPI and used `MPI_COMM_SPAWN` *and his life became fundamentally better.*

The Story So Far

One of the big additions in MPI-2 is the concept of dynamic processes. However, early uses of it were rather mundane and, truth be told, unnecessary. Indeed, dynamic processes were added to MPI, at least in part, as a political necessity since one of the more important parallel run-time systems prior to MPI included the ability to spawn new processes at run-time. Let's take a trip back in history to examine one of MPI's predecessors, the Parallel Virtual Machine (PVM)...

PVM

PVM was a project out of the Oak Ridge National Laboratory (ORNL) in the early '90s and was one of the first truly portable parallel run-time systems. PVM allowed scientists and engineers to develop parallel codes on their workstations and then run them on "big iron" production machines. PVM became enormously popular and enjoyed a large, fervent user base.

Although PVM had the capability to launch multiple processes simultaneously and bind them together into a job, most users tended to prefer a different model for launching their parallel applications. They would simply launch a single, serial process (e.g., *./a.out*) and use PVM's spawning capabilities to launch the rest of the processes required for the parallel application. Indeed, this became the *de facto* method of launching parallel applications in PVM.

MPI's design was strongly influenced by PVM (among others). However, for a variety of technical reasons, the ability to spawn new processes was left out of the initial MPI specification (MPI-1). Although the MPI-1 standard does not specify how to start a parallel job, most implementations launch a set of processes together using an implementation-dependent mechanism, frequently a command named *mpirun*. This set of processes comprises the `MPI_COMM_WORLD` communicator. The size and composition of `MPI_COMM_WORLD` is fixed upon initiation: no processes can be added to or removed from `MPI_COMM_WORLD`.

The PVM community scoffed at this aspect of MPI-1 — why should a parallel application be limited in the number of processes that it could have?

Even though the vast majority of PVM applications only used spawning capabilities to launch their initial job, and even though MPI implementations could support parallel applications as large as PVM (if not larger), this misconception on the part of many PVM users slowed the initial adoption of MPI. Ironically, the startup mechanism in MPI is *simpler* than PVM's launch-one-process-that-launches-all-the-rest model. Specifically, the typical PVM model requires that the user write the spawning code. MPI implementations' built-in `mpirun` commands (or equivalent) handled most of the same functionality.

The fact remains that MPI was built upon the shoulders of PVM and used many of its good ideas

These facts were lost in the Great MPI/PVM Religious Debates of the early- and mid-'90s.

Admittedly, I'm presenting the MPI view of most of the arguments. But the fact remains that MPI was built upon the shoulders of PVM and used many of its good ideas (indeed, the PVM developers were on the MPI Forum). Spawning simply was (initially) not one of them. Three different dynamic process models were later added in the MPI-2 standard.

Spawning New Processes

The first model is, unsurprisingly, spawning new processes. Keep in mind, however, that MPI-2 was intended to be *extensions* to MPI-1 — not *changes*. So if the static model of a fixed `MPI_COMM_WORLD` remains, what does spawning new processes mean in MPI?

In short, it means launching another `MPI_COMM_WORLD`. Spawning is a collective action, meaning the processes in a communicator must unanimously decide to launch a new set of processes. That is, they all invoke the function `MPI_COMM_SPAWN` (or `MPI_COMM_SPAWN_MULTIPLE`) and instruct MPI to launch a new MPI job that has its own `MPI_COMM_WORLD`.

The code snippet in *Listing One* launches four copies of an executable named "`child`" collectively across the processes in the spawning job's `MPI_COMM_WORLD`. This action creates a *new* MPI job with its own `MPI_COMM_WORLD`, containing four processes. That is, at the end of MPI_COMM_SPAWN, there will be *two* MPI_COMM_WORLD instances — one per job. Each will have ranks 0 through (number of processes — 1).

Communication is established between the two jobs through an *intercommunicator* — the children argument to MPI_COMM_SPAWN, above. An intercommunicator is similar to an intracommunicator (i.e., a "normal" communicator, such as MPI_COMM_WORLD) except that it contains two "groups." In this case, one group is the spawning process; the other group is the spawned process. When using intercommunicators, the peer argument of all communication calls is always expressed in terms of the *other* group. Hence, line 6 in *Listing One* is sending to rank 0 of the *children's* group.

The newly spawned application can call MPI_COMM_GET_PARENT to obtain this communicator (see *Listing Two*). Again, since communication with intercommunicators is expressed in terms of the other group, the peer argument given to MPI_RECV on line 9 in Listing 2 is rank 0 of the *parent's* group. Hence, it is receiving the message sent from the MPI_SEND on line 6 in *Listing One*.

Some applications are flexible in that they may be run directly (e.g., via mpirun) or they may be spawned. If an application was spawned, a valid communicator will be returned from MPI_COMM_GET_PARENT. If it was not, MPI_COMM_NULL will be returned (i.e., there is no parent because it was not spawned).

The MPI_COMM_SPAWN_MULTIPLE function behaves the same as MPI_COMM_SPAWN, except that it allows launching an array of different executables and command line arguments in a single MPI_COMM_WORLD — a multiple process, multiple data (MPMD) style of launching.

Connect/Accept

The SPAWN functions are used for creating *new* MPI jobs. But what about *existing* (potentially unrelated) MPI jobs that want to establish communication between each other?

Taking inspiration from the TCP socket connect/accept model, the

LISTING ONE

Sample spawn

```
1. int rank, err[4];
2. MPI_Comm children;
3. MPI_Comm_rank(MPI_COMM_WORLD, &rank);
4. MPI_Comm_spawn("child", NULL, 4, MPI_INFO_NULL,
                  0, MPI_COMM_WORLD,
                  &children, err);
5. if (0 == rank)
6.   MPI_Send(&rank, 1, MPI_INT, 0, 0, children);
```

LISTING TWO

Sample spawned child

```
1. #include "mpi.h"
2. int main(int argc, &argv) {
3.   int rank, msg;
4.   MPI_Comm parent;
5.   MPI_Init(&argc, &argv);
6.   MPI_Comm_get_parent(&parent);
7.   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8.   if (0 == rank)
9.     MPI_Recv(&msg, 1, MPI_INT, 0, 0,
                parent, MPI_STATUS_IGNORE);
10.  /* ... */
```

MPI-2

In 1994, the MPI Forum re-convened to add on to the MPI-1 standard. Several large topics were proposed for inclusion: parallel I/O, new language bindings, one-sided operations, and dynamic processes (to include spawning).

Although strong technical cases were not initially presented as to *why* dynamic processes needed to be included in the MPI-2 standard, it was seen as a political necessity to address the PVM community's concerns. In typical MPI fashion, the MPI-2 standard includes not only spawning, but a total of *three* different models for dynamic process management (three is better than one, right?).

Initial implementations of the MPI-2 dynamic process control models started appearing around 1997. The first uses of it were pretty much a direct port of the PVM start-one-process-that-starts-all-the-rest model. These mainly comprised PVM users porting their applications to MPI in order to take advantage of low-latency networks or utilize vendor-tuned MPI implementations. It was only within the last few years that more interesting uses have become possible through mature, thread-safe implementations of the MPI dynamic process models.

MPI functions MPI_COMM_CONNECT and MPI_COMM_ACCEPT can be used to emulate client-server functionality. Specifically, a "server" process can invoke MPI_COMM_ACCEPT and wait for a "client" process to invoke MPI_COMM_CONNECT to connect to it. This sequence allows two independent MPI jobs to establish communication with each other. Similar to the SPAWN functions, the output of CONNECT and ACCEPT is an intercommunicator.

In the TCP model, IP addresses and port numbers are used to specify the destination of a connect attempt. In MPI, such distinctions are meaningless — some MPI implementations do not even support TCP. Analogous to an (IP address, TCP port) tuple, MPI uses the somewhat confusingly named concept of "ports" as connection endpoints (which have nothing to do with TCP ports).

A server process opens a port with a call to MPI_OPEN_PORT. This port is passed to MPI_COMM_ACCEPT to create a connection endpoint. MPI_OPEN_PORT will also return the name of the port in a dynamic, implementation-dependent string that can be used by the client in its call to MPI_COMM_CONNECT. However, this is a chicken-and-egg problem — how could the client know the server's port name unless they already have some established form of communication?

MPI provides a port name lookup service: the server publishes its port name under a well-known string (e.g., "server") with a call to MPI_PUBLISH_NAME. The client invokes MPI_LOOKUP_NAME with the well-known string ("server") and obtains the server's port name, which is then used to call MPI_COMM_CONNECT.

This publish/lookup system is analogous to how DNS is used to translate human-readable names to IP addresses. For example, when a user types *www.yahoo.com* into a browser, the browser performs a DNS query to resolve this to an IP address that can be used to connect to the server. Change *www.yahoo.com* to "server," and "IP address" to "[MPI] port name," and the above example is probably much clearer.

The Problem with Schedulers

Dynamic processes present many problems for MPI implementers, the most notable of which is what to do in a scheduled environment. Most MPI users have become accustomed to reserving enough nodes/CPUs for their initial parallel job. For example, consider a scheduled cluster where a user receives an allocation of four CPUs and launches a four-process MPI job (i.e., a MPI_COMM_WORLD size of four). If this MPI application invokes MPI_COMM_SPAWN to launch eight more processes, where should these processes be invoked?

- Oversubscribe the nodes: launch the eight new processes the current allocation. This method is possible (and easy), but most HPC applications will not want this because it will likely lead to performance degradation, since multiple processes will be timesharing each CPU.

- Launch on new nodes: this can only occur by obtaining new nodes/CPUs from the scheduler. This action will most likely mean putting the resource request at the end of the scheduler's queue, and may involve a lengthy wait (minutes, hours, or even days). The result is a blocked MPI_COMM_SPAWN for the entire time, potentially wasting a lot of time in the current allocation.

Hence, this is still very much an open question for MPI implementers. Indeed, some vendor MPI implementations have not implemented the MPI-2 dynamic functionality only because they are typically used in production scheduled environments where the focus is to keep the computational resource full — there will never be free resources to SPAWN new jobs on without waiting in the scheduler's queue.

Resources

• MPI Forum	www.mpi-forum.org
• NCSA MPI tutorial	webct.ncsa.uiuc.edu:8900/public/MPI

- 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.

Join

The third and final dynamic process model in MPI-2 is MPI_COMM_

JOIN. Two processes invoke MPI_COMM_JOIN, each with one end of a common TCP socket (MPI does not specify how this socket was created — it is the application's responsibility). MPI can use the socket for startup negotiation in order to establish its own communication channel(s). Upon return from MPI_COMM_JOIN, the TCP socket is drained (but still open) and an intercommunicator containing the two processes (each in their own group) is returned.

Although this model may seem unnatural, having the application establish the initial communication channel is valuable in that it allows the use of an external connection mechanism (i.e., the socket). This feature effectively provides an "escape" from the MPI run-time environment and allows a potentially much wider range of connectivity than is natively supported by the MPI implementation — anywhere that the application can connect a socket.

Keep in mind, however, that MPI-2 was intended to be *extensions* to MPI-1 — not *changes*

Keep in mind that there is no guarantee that the MPI implementation will be able to establish an intercommunicator with the process on the remote end of the socket.

For example, some MPI implementations are geared toward operating system bypass networks; if there is no common OS-bypass network between the two processes, the join may fail. Other problematic scenarios may include intermediate firewalls or other limited connectivity between peer processes.

Where to Go From Here?

We've covered the background and the basics of MPI-2 dynamic processes. Next month, we'll provide some meaningful examples of why and how they can be useful in HPC applications, especially when paired with multi-threaded scenarios.

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

Jeff Squyres is a post-doctoral research associate at Indiana University and is the one of the lead technical architects of the Open MPI project. He can be contacted at jsquyres@open-mpi.org.

Supergeek not required.

You have a compute cluster and want to program it to run a scientific parallel algorithm – but you don't know MPI!

Stop right there. Don't even think about learning MPI. If you don't want to spend the next few years transforming yourself into a parallel computing expert, use CxC® and get better results faster – even in a day. Define a virtual computer to fit your problem, instead of trying to fit your problem to the cluster.

CxC offers simple parallel programming for scientific experts who don't want to be experts in parallel computing, but who want to use the power of parallelism in a smarter way.

- Fast prototyping of parallel algorithms (simple C derivative)
- Develop and test on your Laptop (many million VMs)
- Run and scale on your cluster (mapping SW-2-HW)

Download a trial copy at:
www.engineeredintelligence.com/download/
or request your free CD with examples, manuals, and benchmarks by mail, email, fax or phone:

Engineered Intelligence Corp.
1360 Willow Road #201
Menlo Park, CA, 94031

phone: (650) 587 7100
fax: (650) 587 7101
email: info@engineeredintelligence.com