MPI Mechanic

January 2004

Provided by ClusterWorld for
Jeff Squyres
cw.squyres.com

ClusterWorld™

REDEFINING HIGH PERFORMANCE COMPUTING

www.clusterworld.com

Processes, Processors, and MPI, Oh My!

The Story So Far

Last month, we covered the basics and fundamentals: what MPI is, some a simple MPI example program, and how to compile and run the program. This month, let's dive into a common terminology misconception: processes vs. processors - they're not necessarily related!

In this context, a processor typically refers to a CPU. Typical cluster configurations utilize uniprocessors or small Symmetric Multi Processor (SMP) nodes (e.g., 2-4 CPUs each). Hence, "processor" has a physical - and finite - meaning.

Last month, I said that MPI is described mostly in terms of "MPI processes," where the exact definition of "MPI process" is up to the implementation (it is usually a process or a thread). An MPI application is composed of one or more MPI processes. It is up to the MPI implementation to map MPI processes onto processors.

Threads and (MPI) Processes

Most MPI implementations define an MPI process to be a Windows or POSIX process. Hence, each MPI process has its own global variables, environment, and does not need to be thread-safe. Some MPI implementations, however, do define MPI processes as threads. The Adaptive MPI (AMPI) project from the University of Illinois, for example, uses this model.

Other notable items about MPI, threads, and processes:

- The MPI standard does not define interactions of MPI processes with non-MPI processes. Specifically, what happens when an MPI process invokes `fork(2)` is implementation-dependent.
- Although the MPI-2 document does define the behavior of threads in an MPI process, an MPI implementation may or may not support concurrency in multi-threaded MPI applications.

Mapping MPI Processes to Processors

An implementation may allow you to run M processes on N processors, where M may be less than, equal to, or greater than N. Although maximum performance is typically achieved when each process has its own processor (i.e., when M <= N), there are cases where over-subscribing processors is useful as well (i.e., where M > N).

Table 1 gives a brief description of each possible scenario.

When there the number of processes is less than or equal to the number of processors, the application will run at its peak performance. Since the total system is either underutilized (there are unused processors) or fully utilized (all processors are being used), the application is not hindered by context switching, cache misses, or virtual memory thrashing caused by other local processes.

The "underutilized" model may also be somewhat misleading. It is not uncommon for an application to use MPI to launch one process per node (and therefore have processors on a node that are not initially used) and spawn computation threads to utilize the additional processors. As such, shared memory/threaded programming techniques are used for on-node coordination and data transfer; MPI is used for off-node message passing. Combined MPI and OpenMP applications use this model, for example.

Over-subscribing processors, where more processors are launched than there are physical processors, is typically only used for development and testing, or when access to large parallel resources (such as a production cluster) are limited, expensive, or otherwise constrained. Hence, even though the overall application is almost guaranteed to run with some level of performance degradation, this scenario can be useful to isolate problems, identify performance bottlenecks, or cause artificial race conditions. It can be quite difficult to debug a 4,096 process parallel application; scaling down and running 32 processes (perhaps even on a handful of development workstations, depending on the nature of the application) can make the difference between an impossible-to-locate-and-replicate Heisenbug and an easily-identifiable-and-fixable typo in the code.

It is common to develop and debug parallel applications with a small number of processes (e.g., 2, 4, or 8) on a single workstation. As the application becomes more fully developed and stable, larger testing runs can be conducted on actual clusters to check for scalability and performance bottlenecks.

The Art of Over-subscribing

Most MPI implementations will allow running arbitrary numbers of

TABLE ONE
Possible Mappings of MPI Processes to Processors

SCENARIO	DESCRIPTION
Less processes than processors	Resources are potentially underutilized, unless additional threads are spawned on unused processors
One process per processor	Resources are fully utilized, potentially running more than one MPI process per node
More processes than processors	Resources are oversubscribed, likely degrading overall performance

MPI processes, regardless of the number of available processors. This is somewhat of a black art - if you run too many processes, the processors will thrash, continually trying to give each process its fair share of run time. If you run too few, you may not be able to run meaningful data through your application, or may not trigger error conditions that occur with larger numbers of processes.

For example, running 8 computational and memory-intensive MPI processes on a single uniprocessor workstation will likely result in the machine slowing to a crawl while the cache, virtual paging system, and process schedulers are thrashed beyond reasonable bounds. Conversely, running only 2 lightly-computational, tightly-synchronized processes on a uniprocessor workstation may be acceptable in terms of performance, but may fail to show errors that only occur when running with an odd number of processes.

Different MPI Implementations

As mentioned in last month's column, there are many different implementations of MPI available. This month, we'll go step-by-step in using four different MPI implementations. All examples will use the sample "Hello world" MPI program from last month (see Figure One).

The four implementations that we'll focus on are all open source and freely available:

- FT-MPI v1.0 from the University of Tennessee
- LA-MPI v1.3.6 from Los Alamos National Laboratory
- LAM/MPI v7.0.2 from Indiana University
- MPICH v1.2.5.2 from Argonne National Labs

Each implementation has its particular strengths and weaknesses (to be discussed in future columns). Here, we'll focus simply on compiling under each implementation and then running in a few different scenarios. It should be noted that we'll only cover common scenarios in each implementation; consult the extensive documentation and manual pages available with each implementation for more details.

Compiling

Both LAM/MPI and MPICH all offer a `mpicc` "wrapper" for compiling and linking C MPI programs (and corresponding `mpif77` and `mpiCC` for Fortran and C++ programs), making compilation and linking easy (provided your environment variables are set correctly):

```
$ mpicc hello.c -o hello
```

FT-MPI has wrapper compilers, but it is named `ftmpicc` (and `ftmpif77`). It behaves identically to LAM/MPI's `mpicc`.

LA-MPI does not provide wrapper compilers; note the following when compiling MPI applications with LA-MPI:

- A C++ compiler must be used for linking LA-MPI applications
- Depending on where it was installed, you may need to provide the relevant `-I`, `-L` flags.
- You may need to provide additional linker flags (e.g., `-pthread`)
- You need to provide `-lmpi` to the linker

This sounds scary; it's not. Most of the time, these details are hidden in a Makefile and are therefore unnoticed by the user.

In this example, LA-MPI was installed on a Linux machine with the GNU compilers in `/usr/lampi`:

```
$ g++ hello.c -I/usr/lampi/
    include -L/usr/lampi/lib
  -pthread -lmpi -o
    hello.la-mpi
```

Running 4 MPI Processes on the Localhost

Now that we have a `hello` MPI program compiled, how do we run it in parallel? All three implementations come with an `mpirun` program designed to launch MPI applications. In this section, we'll simply launch 4 MPI processes on the localhost (a common debugging/development scenario).

FT-MPI requires starting up a run-time environment (RTE) before launching MPI applications. There are multiple ways to do this; one way is to use the FT-MPI console:

```
$ console
con> add localhost
con> spawn -np 4 -mpi
hello.ft-mpi
```

To have LA-MPI's `mpirun` launch locally, it is easiest to set the `LAMPI_LOCAL` environment variable to 1 and use the `-np` switch to request the number of processes to run:

```
$ export LAMPI_LOCAL=1
$ mpirun -np 4 ./hello.
  la-mpi
```

LAM/MPI requires its RTE to be started with the command `lamboot` before using `mpirun`. To start the RTE on just the localhost, invoke `lamboot` with no arguments. `mpirun` can then be used with the same `-np` switch to indicate how many MPI processes to launch:

```
$ lamboot

$ mpirun -np 4 hello.
  lam-mpi
```

Similar to LAM/MPI, MPICH has a daemon-based RTE, but most MPICH installations still default to the ubiquitous `rsh`/`ssh`-based startup mechanisms. In this configuration, MPICH's `mpirun` will always use a hostfile to specify which hosts to run on. If one is not supplied on the command line, a default file (created when MPICH was installed) will be used. For this example, create a text file named `my_hostfile` with a single line "localhost" in it. Then use the `-machinefile` switch to specify your hostfile, along with the `-np` switch:

```
$ cat my_mpich_hostfile
localhost
$ mpirun -machinefile
my_mpich_hostfile -np 4
hello.mpich
```

FIGURE ONE

Hello World MPI Program

```
#include <stdio.h>
#include <mpi.h>
 int main(int argc, char **argv) {
   int rank, size;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   printf("Hello, world. I am %d of %d.\n",
     rank, size);
   MPI_Finalize();
   return 0;
 }
```

Running 4 MPI Processes on 2 Dual Processor SMPs

FT-MPI will run across as many hosts are running in its RTE. If you add multiple hosts, it will launch on both, placing adjacent ranks in `MPI_COMM_WORLD` on the same node:

```
con> add node1.example.com
con> add node2.example.com
con> spawn -np 4 -mpi
hello.ft-mpi
```

The `haltall` console command shuts down the FT-MPI RTE.

LA-MPI allows the specification of process counts and hosts on the `mpirun` command line. For example:

```
$ mpirun -N 2 -H node1.
  example.com,node2.example
  .com -n 2,2 hello.la-mpi
```

The `-N` switch says to use 2 hosts, `-H` provides a comma-separated list of hosts, and `-n` specifies how many processes to start on each host.

LAM/MPI allows flexible specification of process placement via both the hostfile given to `lamboot` and the `mpirun` command line.

```
$ cat my_lam_hostfile
node1.example.com cpu=2
node2.example.com cpu=2
$ lamboot my_lam_hostfile
$ mpirun C hello.lam-mpi
```

When you are finished with LAM's RTE, shut it down with the `lamhalt` command.

Each host is listed once in `my_lam_hostfile` with a second tag indicating how many CPUs it has (i.e., how many processes LAM should start on that machine). Instead of `-np`, use `C` on the `mpirun` command line, telling LAM to start on "all available CPUs." LAM will automatically place adjacent ranks of `MPI_COMM_WORLD` be in the same node. This can be ideal, for example, in batch environments where the number of target processes may be variable.

MPICH also requires a hostfile:

```
$ cat my_mpich_hostfile
node1.example.com
node2.example.com
$ mpirun -machinefile
my_mpich_hostfile -np 4
hello.mpich
```

MPICH will run on each host in the machinefile in round-robin fash-

ion for the number of processes specified by the `-np` parameter (hosts can be listed more than once to force adjacent ranks in `MPI_COMM_WORLD` to be on the same node).

Where To Go From Here?

So MPI is MPI is MPI, but not all MPI implementations are created equal. Every MPI implementation is slightly different in minor ways, to even include compiling and running applications. Despair not - even though the differences are typically annoying, they're nothing that users can't figure out with a few minutes perusal of a man page.

If you ran the `hello.c` program from the last column, you may have noticed that the output order was not as expected. There is no guarantee of output order based on rank (unless specifically programed as part of th operation). As we have seen, process placement can vary from run to run depending upon how your MPI is configured and thus effect output order as well. parallel input and output will be addressed in a future column.

Jeff Squyres is the lead developer for the LAM implementation of MPI. Email him at jsquyres@lam-mpi.org

Resources

AMPI Project:
- charm.cs.uiuc.edu/research/ampi

Distributed Debugging Tool:
- www.streamline-computing.com

FT-MPI:
- icl.cs.utk.edu/ft-mpi

LA-MPI:
- www.acl.lanl.gov

LAM/MPI:
- www.lam-mpi.org

MPICH:
- www.mcs.anl.gov/mpi/mpich

MPI Forum (MPI-1 and MPI-2 specifications 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. 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.

Totalview:
- www.etnus.com/Products/TotalView