The Case for an MPI ABI

Greg Lindahl
Distinguished Engineer
PathScale, Inc.

Outline

- What is MPI?
- Why do we need an ABI?
- Winners and losers in the ABI game
- What should the ABI include?
- Next steps

What is MPI?

- MPI is an applications programming interface (API) for message-passing
- It has conquered the marketplace, leading to better portability and performance
- Being an API instead of an ABI allows maximum flexibility
  - This has been proven to be good and bad
  - Good: innovative interconnects can change what needs to get changed to run fast

MPI has won. What's next?

- MPI is now widely used. But it has some issues:
  - Needs to be integrated with queue systems; no generic way to do this
  - Needs recompilation for new MPI implementations... not so easy for commercial apps, or easy distribution of open source apps
  - Startup syntax is not standard; this makes automated cross-implementation testing harder (and thus rare)
  - Lack of standardization of “environmental issues” leads to lack of portability

Queue system integration

- All MPI implementations have different startup mechanisms
- Some have more than 1!
- Needs integration with queue systems
  - Communicate the nodelist, create the actual processes, and make sure they're cleaned up
- Most solutions only work for 1 queue system and 1 MPI implementation. Ewww.

Startup Syntax

- “mpirun” is not “mpirun”
- Ever wonder why MPI applications don't come with a “make check” target?

Recompilation Considered Harmful

- Commercial vendors, for good reasons, insist on testing every binary very carefully. Any small change can bring out a compiler bug.
- Open source projects are now frequently distributed as binary packages, but that's awkward for MPI apps.
- MPI applications are spreading to disciplines where the average consumer does not compile anything: bioinformatics, music, graphics, weather forecasting...

Environmental Issues

- MPI doesn't specify quite a few things
- This is great at the start, but now apps has subtle unportabilities built into them
- Getenv()... putenv()... cwd... buffering of stdout/stderr...
- Is this a problem in practice? I'm not sure.

The Commercial Squeeze

- As more users don't compile, the MPI market is pushed in a winner-take-all direction: One MPI implementation will eventually rule them all
  - Current commercial contenders for the one ring: Intel, HP, Scyld, Scali, MPIPro(tm)
- MPICH may yet win. But if it doesn't:
  - Reduced innovation because a commercial MPI winner dictates who gets to play
  - Reduced innovation due to the structure of the winner's component strategy (ADI2/3, DAPL, etc.)

The Solution

- An Applications *Binary* Interface for MPI
- Compile anywhere. A shared library is linked at run-time.
- It was premature to do this 3 years ago.
- Now is the right time.
- An ABI might include more than libc's ABI
  - Could cover queue system integration and startup
  - More details later

Winners and Losers

- MPI end-users
- MPI implementation researchers
- Interconnect implementers
- Commercial software vendors
- Open source software projects
- Existing MPI implementations

Winner: MPI end-users

- Any MPI app works on your system
- Your app works on your collaborator's system
- Less compiling required
  - “We don't modify MM5, why are we recompiling it?”
- Less work to integrate queue system with MPI implementations... often left to end-users to do
- Standardized scripts to run MPI apps

Winner: MPI implementation researchers

- Today: you can implement the API and test with anything you can get source for
  - Want to test on AMD's cluster? Oh no, you have to match its build environment and integrate with its queue system!
- With an ABI, you can easily test with any app in any environment (maybe even commercial apps!)
- (Requirement: The ABI must not prevent implementation of cool features.)

Winner: Interconnect Implementers

- Today: You can only reach the market which recompiles. Oh, and queue systems are a problem.
- With an ABI, you can reach almost all of the market.
- Avoid winner-take-all squeeze
- (Requirement: the ABI must not prevent implementation of cool features.)

Winner: Commercial Software Vendors

- Today: Uh, which single MPI am I going to support? Um... Um...
- With an ABI, all MPIs look the same
- (Not solved: Testing. Commercial software vendors will mostly only certify one MPI implementation... but at least some will let you run with anything as long as you attempt to replicate bugs on the supported implementation.)
- Note that the ABI makes automated testing easier.

Winner: Open Source Software Projects

- Today: Can't ship binary packages that work on diverse systems.
- Tomorrow: MPI is just like everything else... install your application package, and it works with whatever batch system and MPI
- Also, testing on multiple MPI implementations is much easier.

Mixed: Existing MPI implementations

- Some changes required to support ABI
  - Who pays to update “free” implementations?
- No more staying up nights worrying about winner-take-all process
  - “MPICH vs LAM” always was a silly argument
- Increase in MPI portability a big plus
- Increased automated testing a big plus

Creating an ABI

- Standardize existing practice as much as possible
  - Minimizes cost, mistakes, and work to write ABI
- Include what is required, leave out what is not required
  - A balancing act
- Don't crush too many egos along the way
  - Process partly social, partly technical
  - Without broad acceptance, the ABI will fail

Elements of an ABI

- Fully specify what's in <mpi.h> and “mpif.h”
  - This is similar to libc... and is the easiest part.
  - We might stop here.
- Fully specify startup and queue system interaction in a flexible way
  - “mpistart” -- “mpirun” with standard args
  - Generic interface with queue systems
- Specify environmental items so that most apps are portable

Issues: Header Files

- For C, MPI doesn't specify types or values of constants.
- Oops: Most MPIs have extensions
  - how widely used by actual apps?
- Variably-sized items have to be nailed down
  - MPI_Status must have fixed length
- These issues are relatively straightforward

Issues: Shared Libraries

- Once we standardize <mpi.h>, we need a sane shared-library setup to avoid relinking
- I think this means we need 1 base library instead of the usual chaos
- Implementers can dlopen() additional libs as needed
- Perhaps we can have a clever upward-compatible scheme for MPI 1 and MPI 2 libs?

Issues: mpirun

- Start over: “mpistart” with standardized args
- Require features such as supporting pipes of stdin, stdout, error
- --batch and –batch-wait to assist scripting in the presense of batch queues... noop if no queue sys
- I have a strawman suggestion for this:

Issues: startup and queue systems

- Queue sys handles creation of MPI processes
- Standard command-line for MPI process, with enough info to connect to the “job”
- MPI implementation provides “job” process; all MPI processes contact to coordinate.
  - “job” process started by mpistart
- Queue sys ensures all MPI processes exit at end.
- I think this supports all existing queue systems and MPI implementations

Issue: Environment

- Getenv / putenv: most implenentations don't propagate env vars to ranks 1+; few support using putenv to other processes.
- Cwd depends on batch system and sysadmin
  - 2 choices: least common denominator, or query fns?
- Stack size. Usually painful for users to fix.
- stdin/stdout/stderr... oh, don't forget buffering...
- MPI-I/O.
- Actions before/after mpi_init/finalize

Next steps

- Assemble list of stake-holders
- Do they find the case for an ABI compelling?
  - If not, oh-oh
- Run ABI process like MPI committee
  - A proven model in this community
- At minimum, MPICH and OpenMPI as reference implementations
  - So both groups need to find the ABI compelling...
- Timeframe: One year? (I dream...)