EE542 - Reading Assignment – 06

A high-performance, portable implementation of the MPI message passing interface standard

Presenter: Boyang Xiao

USC id: 3326-7302-74

Email: boyangxi@usc.edu

Index

Introductions and background

- **MPI (Message Passing Interface)** is a is a specification for a standard library for message passing that was defined by the MPI Forum.
- **MPICH** is a unique implementation of MPI among existing implementations in its design goal of combining portability with high performance.
- **Precursor systems:** MPICH came into being quickly because it could build on stable code from existing systems. These systems prefigured in various ways the portability, performance, and some of the other features of MPICH.

MPICH’s challenge: Portability and performance

- Portability of MPICH
    - The MPI standard itself addresses the message-passing model of parallel computation, where processes with separate address spaces communicate with one another by sending and receiving messages.
- Ways MPICH uses to achieve portability:
    - Exploiting high-performance switches
    - Exploiting shared-memory architectures
    - Exploiting networks of workstations

MPICH’s challenge: Portability and performance

- Performance of MPICH
  - The MPI specification was designed to allow high performance in the sense that semantic restrictions on optimization were avoided wherever user convenience would not be severely impacted.
  - Furthermore, a number of features were added to enable users to take advantage of optimizations that some systems offered, without affecting portability to other systems that did not have such optimizations available.
- Performance of MPICH compared with native vendor systems
  - The MPI performance is quite good, compared to Intel’s NX message-passing, and can probably be improved with the second-generation ADI, planned for a later release of MPICH.

MPICH’s architecture

- Designing principles:
  - Maximize the amount of code that can be shared without compromising performance.
  - Provide a structure whereby MPICH could be ported to a new platform quickly, and then gradually tuned for that platform by replacing parts of the shared code by platform-specific code
- Abstract device interface (ADI): the **central mechanism** for achieving the goals of portability and performance
  - ADI is a set of function definitions in terms of which the user-callable standard MPI functions may be expressed.
  - ADI provides the message-passing protocols that distinguish MPICH from other implementations of MPI.
  - the AD1 layer contains the code for packetizing messages and attaching header information, managing multiple buffering policies, matching posted receives with incoming messages or queuing them if necessary, and handling heterogeneous communications.

MPICH’s architecture

- The channel interface
    - The channel interface consists of:
        - 5 required functions.
        - 3 routines send and receive envelope (or control) information: *MPID_SendControl*, *MPID_RecvAnyControl*, and *MPID_ControlMsgAvai1*
        - 2 routines send and receive data: *MPID_SendChannel* and *MPID_RecvFromChannel*.
    - The channel interface implements three different data exchange mechanisms:
        - **Eager:** In the eager protocol, data is sent to the destination immediately. If the destination is not expecting the data, the receiver must allocate some space to store the data locally.
        - **Rendezvous:** In the rendezvous protocol, data is sent to the destination only when requested. When a receive is posted that matches the message, the destination sends the source a request for the data. In addition, it provides a way for the sender to return the data.
        - **Get:** In the get protocol, data is read directly by the receiver. This choice requires a method to directly transfer data from one process’s memory to another.

Part of subsystems of MPICH

- Groups

  The basis of an MPICH process group is an ordered list of process identifiers, stored as an integer array. A process's rank in a group refers to its index in this list. Stored in the list is an address in a format the underlying device can use and understand.

- Communicators
  - The MPI standard describes two types of communicators, intracommunicators and intercommunicators, which consist of two basic components, namely process groups and communication contexts. MPICH intracommunicators and intercommunicators use this same structure.
  - In order to provide safe point-to-point communications within a collective operation, an additional "collective" context is allocated for each communicator. This collective context is used during communicator construction to create a "hidden" communicator (*comm_coll*) that cannot be accessed directly by the user.

Part of subsystems of MPICH

- Collective operations
  - As noted in the preceding section, MPICH collective operations are implemented on top of MPICH point-to-point operations
  - MPICH collective operations retrieve the hidden communicator from the communicator passed in the argument list and then use standard MPI point-to-point calls with this hidden communicator.
  - Each MPI collective operation checks the validity of the input arguments, then forwards the function arguments to the dereferenced function for the particular communicator.
- Attributes
  - Attribute caching on communicators is implemented by using a height-balanced tree (HBT or AVL tree). Caching an attribute on a communicator is simply an insertion into the HBT, retrieving an attribute is simply searching the tree and returning the cached attribute.

Part of subsystems of MPICH

- Topologies
  - Support for topologies is layered on the communicator attribute mechanism.
  - For communicators with associated topology information, the communicator's cache contains a structure describing the topology.
  - The MPI topology functions access the cached topology information as needed, then use this information to perform the requested operation.
- The profiling interface
  - The MPI Forum wished to promote the development of tools for understanding program behavior, but considered it premature to standardize any specific tool interface.
  - Users can interpose their own "profiling wrappers" for MPI functions by linking with their own wrappers, the standard version of the MPI library, and the profiling version of the MPI library in the proper order.
  - MPICH also supplies a number of prebuilt profiling libraries

Part of subsystems of MPICH

- The Fortran interface
  - The determination of the name is handled by our configure program, which compiles a test program with the user’s selected Fortran compiler and extracts the external name from the generated object file. This allows us to handle different Fortran compilers and options on the same platform.
- Job startup
  - The extreme diversity of the environments in which MPICH runs and the diversity of job-starting mechanisms in those environments suggested to us that we should encapsulate the knowledge of how to run a job on various machines in a single command. We named it ***mpirun***.
- Building MPICH
  - An important component of MPICH’s portability is the ability to build it in the same way in many different environments.

Part of subsystems of MPICH

- Building MPICH (Cont'd)
    - The configure script determines aspects of the environment
    - Perform tests of the environment to ensure that all components required for the correct compilation and execution of MPICH programs are present
    - Construct the appropriate Makefiles in many directories, so that the make command will build MPICH.
    - After being built and tested, MPICH can be installed in a publicly available location.
- Documentation
    - MPICH installation guide: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.457.8961&rep=rep1&type=pdf
    - MPICH user's guide: https://digital.library.unt.edu/ark:/67531/metadc682504/m2/1/high_res_d/378910.pdf

Tools toward a portable parallel programming environment

- The MPE extension library
  - MPE (Multi-Processing Environment) is a loosely structured library of routines designed to be “handy” for the parallel programmer in an MPI environment.
  - MPE routines categories:
    - **Parallel X graphics:** Routines are provided to set up the display (probably the hardest part) and draw text, rectangles, circles, lines, etc.
    - **Logging:** The MPE library provides simple calls to produce time-stamped event trace files.
    - **Sequential sections:** The MPE library provides functions to ensure that this type of execution occurs, when a section of code that is executed on a set of processes must be executed by only one process at a time
    - **Error handling:** The MPI specification provides a mechanism whereby a user can control how the implementation responds to run-time errors

Tools toward a portable parallel programming environment

- Command-line arguments and standard I/O
  - MPICH ensures that on each process, the command-line arguments returned from *MPI_Init* are the same on all processes, thus relieving the user of the necessity of broadcasting the command-line arguments to the rest of the processes from whichever process actually was passed them as arguments to main.
  - In MPICH, all processes have access to *stdin* , *stdout, and stderr*, and on networks these I/O streams are routed back to the process with rank 0 in MPI_COMM_WORL
- Support for petionnance analysis and debugging
  - The MPI profiling interface allows the convenient construction of portable tools that rely on intercepting calls to the MPI library:
    - Profiling libraries
    - Upshot
    - Support for adding new profiling libraries

Tools toward a portable parallel programming environment

- Network management tools
    - The Scalable Unix Tools (SUT) are a useful part of the MPICH programming environment on workstation clusters. Basically, SUT implements parallel versions of common Unix commands such as Is, ps, cp, or rm.
    - Graphical displays also show the load on each workstation and can help one choose the sub-collection of machines to run an MPICH job on.

Software management techniques and tools

- MPICH was written by a small, distributed team sharing the workloads, who have worked to distribute new releases in an orderly fashion, track and respond to bug reports, and maintain contact with a growing body of users.
- Methods to cooperate and collaborate between different distributed teams:
  - **Configuring for different systems**
  - **Source code management**: They use RCS via the Emacs VC interface
  - **Testing**: MPICH contains a test suite that attempts to test the implementation of each MPI routine.
  - **Tracking and responding to problem reports**
  - **Preparing a new release**: To help test a new release of MPICH, teams use several programs and scripts that build and test the release on a new platform.

Q&A

- **Questions:** What is/are MPICH's most features that distinguish it from other implementations of MPI and what mechanisms does MPICH use to achieve this/these features.

- Answer:

  The most distinguished features of MPICH are its portability and its performance. To achieve these two features, MPICH uses two key mechanism which are:

  1. Abstract device interface (ADI) which is a set of function definitions in terms of which the user-callable standard MPI functions may be expressed
  2. The channel interface: which take cares of the underlying message passing mechanisms and provide several API for users to use.

Thanks for watching!

Presenter: Boyang Xiao

USC id: 3326-7302-74

Email: boyangxi@usc.edu