Design of DMPI on DAWNING-3000

Wei Huang, Jie Ma, Zhe Wang, Dan Meng, Ninghui Sun

Institute of Computing Technology, Chinese Academy of Sciences
P.O. Box 2704, Beijing 100080, P.R.China
{hw , majie, wz, md, snh}@ncic.ac.cn

Abstract. This paper introduces the design and implementation of DMPI, a high performance MPI running on a cluster of SMPs (CLUMPS) called DAWNING-3000 super-server. DMPI provides high performance communication layer with remote memory access (RMA) functions and a flexible and sufficient extension of ADI (DADI-E). The bandwidth of DMPI is almost reached the peak bandwidth of the lower level communication protocol. With the expanded DADI-E, it is easily to provide a high performance PVM library with the efficient lower communication layer of MPI. Important features of DMPI and DADI-E are presented in the paper, including special data path, RMA mechanism, dynamic process management and special support for PVM. The performance of DMPI over Myrinet is also given.

Keywords: DMPI, BCL, DAWNING-3000, DPVM, RMA, Dynamic process management, DADI-E.

Introduction

MPICH, developed by Argonne National Laboratory, U.S.A, is an important implement of MPI standard. It can easily be ported to other MPI implementation over different low-level communication protocols. The MPICH implementation of the MPI standard is built on a lower level communications layer called ADI (abstract device interface). The purpose of this interface is to make it easy to port the MPICH implementation by separating into a separate module that handles just the communication between two processes. The ADI provides basic, point-to-point message-passing services for the MPICH implementation of MPI. The MPICH code handles the rest of the MPI standard, including the management of datatypes, and communicators, and the implementation of collective operations with point-to-point operations.

DMPI is a high-performance MPI designed and implemented on DAWNING-3000. DAWNING-3000 is a CLUMPS consisting of 70 nodes. Each node is a 4-way 375MHz Power3 SMP, running IBM AIX4.3.3. All nodes are connected with Fast Ethernet, Myrinet and Dnet. Dnet is high performance interconnect developed by us. DMPI is based on Resource Management System (RMS) and Basic Communication

Library (BCL), which are developed on our own system. A few important features of DMPI are described as follows.

- Special data path. DMPI provides a special data path to improve the communication bandwidth. Since the lower level communication layer can provide high bandwidth and low latency, the normal data path used in MPICH is not efficient enough. DMPI provides an efficient data path to improve the communication performance.
- RMA mechanism. RMA is a new module in MPI-2 standard. It is a complement to the point-to-point message-passing programming model. This new model facilitates the coding of some application with dynamically changing data access patterns where the data distribution is fixed or slowly changing.
- Dynamic process management. Dynamic process management is another new module provided in MPI-2 standard. In this new model, new processes can be created and management after an MPI application has been started.
- Flexible and sufficient DADI-E. MPI and PVM are two widely used communication libraries. Each super-server and super computer needs to port both of the libraries. DADI-E extents the ADI-2 to support DPVM, so that the DPVM can benfit from the efficient communication data path using in DADI-E.

Section 2 gives an overview of the communication software used on our system, including BCL, DPVM and DMPI. In the following sections, DMPI is discussed in detail, including an optimization that can obviously increase the bandwidth of long message in the point-to-point communication, and the implementation of RMA operation and dynamic process management. Then the evaluation of DMPI on DAWNING-3000 platform is shown in Section 3. Finally, we present our conclusions and discuss ongoing research in Section 5.

Overview

BCL is a low-level communication software used on DAWNING-3000. The current version of BCL is BCL-3. It is implemented on Myrinet and Dnet. The upper levels can make full use of the hardware performance via BCL-3. BCL-3 supports point to point message passing. All other collective message passing should be implemented in the higher level software.

RMS is the resource management software used on DAWNING-3000, all of its management work is done by two daemon applications. They are RMD and CSD. RMD is a resource management daemon and CSD is a communication service daemon. RMD manages all the system's resource including nodes, ports, pools, and parallel tasks. RMD takes charge in adding, finishing and killing parallel tasks. It also takes part in creating, modifying and deleting pool. CSD charge for adding tasks. it also provide information for each process running for the parallel task. When a process's information is changed, CSD then broadcast the changes to other process.

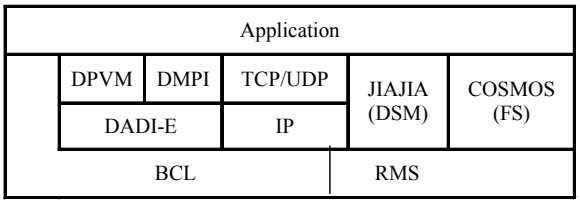

Fig. 1. Protocol stack of DAWNING-3000 software

Fig. 1 shows the protocol stack from BCL-3 applications' point of view. Applications can directly access the BCL-3 level or use other functionality levels. BCL-3 is the lowest level software in DAWNING-3000's communication software. MPI is implemented directly on BCL-3. PVM is implemented on ADI-2 (Abstract Device Interface) of MPI. Notice that this may be changed according to the different implementations that are done. In order to keep high performance, we can also port PVM directly on top of BCL-3. TCP/IP is designed to port onto our BCL-3 in the next step. A prototype will be completed early next year. The other two components in the stack are JIAJIA and COSMOS. The former is an DSM software, and the latter is a distributed file system of DAWNING-3000.

Our DMPI is ported from MPICH-1.2.1.10, while DPVM is ported from PVM-3.3.1.1. They are both based on BCL and RMS. Origin PVM used TCP/IP as its communication protocol, for TCP/IP is a general communication protocol and can adapt to a good many of environments both of homogeneous and of heterogeneous. But it is too complex to provide high performance when it is used in some special system such as our DAWNING-3000 super-servers. DMPI and DPVM are implemented to use BCL communication protocol to improve the performance. To benefit from the high performance ADI layer, ADI is extended to DADI-E to support DPVM. DADI-E remains all the functionality of ADI, and adds some new features as follows.

- Add communication calls provided by BCL, which provide the upper communication software MPI and PVM of low-level communication APIs. So that the upper level software can make full use of the hardware performance.
- Add resource management calls provided by RMS. It can be easy for PVM to manage process dynamic resource. And it also is propitious to extend MPI-1 standard to MPI-2 for the functionality of dynamic process management.
- Provide common APIs both for MPI and PVM. The DADI-E provides basic, point-to-point message-passing services and resource management services for DMPI and DPVM. The users can use MPI or PVM easily, while not knowing how it is really implemented.

RMA and dynamic process management have been implemented in DMPI. All the implementation lies in the DADI-E layer. It will be discussed in detail in the next section.

Design Issues

Performance Optimization

When tested the bandwidth of BCL and DMPI using the ping-pong mode under the same environment (it will be described later, but the bandwidth of PCI bus is 33MHz). We can see that the peak bandwidth of DMPI is about 180MB/s, which is only 75% to the peak bandwidth of BCL, about 240MB/s.

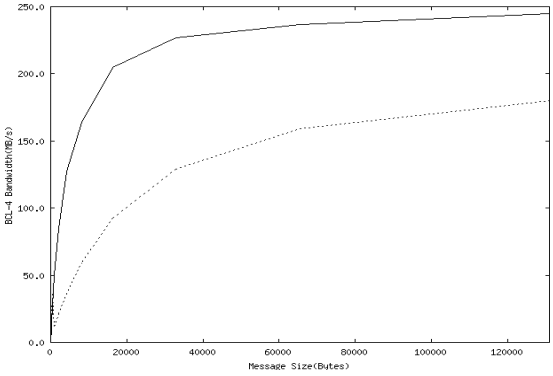

Fig. 2. Bandwidth of BCL and DMPI

As we known, DMPI lies on the BCL, and BCL provides APIs for DMPI. When a process communicates with another one, it calls MPI_Send or MPI_Isend, but at last it will call the BCL_Send to do the true sending. Then some time will be wasted in the course from MPI_Send to BCL_Send. This is why our DMPI's bandwidth can only 180MB/s, contrast to BCL's 240MB/s. When analysising the procedure flow of DMPI, we will find that there is an added handshake process when communicate with DMPI protocol against doing communication directly with BCL protocol. In a point-to-point communication, whether sender or receiver is provided only one source buffer address for sending or receiving, but they don't know where to send (from he view of the sender) and from where to receive (from the view of the receiver). The sender and receiver need a handshake to exchange the buffer address.

On the other side, the maximum data can be sent by BCL_Send is limited to128KB, While the size of data message is of no limit when using MPI_Send. If a large data message's size is larger than 128KB, then we should cut the data message into data fragments. Each data fragment is size of 128KB, except that the last data fragment's size maybe is less than 128KB. Then we will send the data fragments by to by. A whole sending/receiving procedure can be described as such:

- The sender sends a control message to the receiver, and tells the receiver that he is ready to send a data message.
- The receiver has received the control message, then knows that the sender will send to him a data message, so he posts his receive buffer to the net card. Then he

sends back an ack control message packed with the receive buffer address to the sender.

- The sender received the ack control message and got the target address, then he calls the BCL_Send to send the data message to the target buffer.
- The sender does polling the sent complete event and the receiver does polling the received complete event.
- When the receiver have polled the received complete event (it means that the data have been received), if all the data fragments have been received at this time, the procedure will be end. Otherwise, goto 2.

To improve the bandwidth, the central aim is to reduce the total time used. Because the most work done by the MPI_Send call lies in the check_device function, whose procedure has been explained above. We will introduce one way of pipeline-sending to reduce the total time check_device spent when send with long data message.

If we send a n*128KB large data message, there will be n times handshake between sender and receiver, every data fragment is sent after the preceding one is really received. We assume one handshake will spend x (us) times, and all the other function will spend total y (us) times. So when we send such a large data message, it will spend total (y + n*x) (us) times. Because when a data fragment is sending, both the sender and receive are idling to wait the sending complete. If we can make use of this idling time to complete their handshake operation, there will need only (y + x) (us) times to complete the total procedure, (n -1) (us) times less than before. This way is a little like the pipeline in some factory, so we name it pipeline-sending.

Then I will explain our implement to this arithmetic. We changed the procedure in step 3: if the send received the ack control message, before sending the large data message, we first produce a control message include some information of the send request, and send it to the receiver. The receiver got this control message, it will return the ack control message to the sender, but not to wait the receiving complete event. When the sender receive the ack control message, he can begin the next sending, not to wait util the preceding data fragment being truly sent complete. Thereby, the handshake operation is processed parallel with the data sending except the first handshake which is necessary.

With this optimization, we improve the bandwidth of large data message as expected. The large data message's bandwidth of DMPI improved to 230MB/s, about 95% of BCL's bandwidth.

In the future's work, we will extend this arithmetic to optimize the collective operations, such as MPI_Bcast. The current implement of MPI_Bcast uses a fairly basic recursive subdivision algorithm. The root sends data to the process size/2 away; the receiver becomes a root for a subtree and applies the same process. When handling the long message, we can do pipelining-sending the messages in data fragments so that the message can be pipelined through the network without waiting for the subtree roots to receive the entire message before forwarding it to the other processors.

Implementation of RMA

RMA (Remote Memory Access) is one of the important extended components of MPI-2 standard. RMA extends the communication mechanisms of MPI by allowing one process to specify all communication parameters, both for the sending side and for the receiving side. It is a complement to the point-to-point message-passing programming model, which needs the both sides (source side and target side) to cooperate for completing a communication.

In the communication model of RMA, each process can compute what data it needs to access or update at other processes. However, processes may not know which data in their own memory need to be accessed or updated by remote processes, and may not even know the identity of these processes. Thus, the transfer parameters are all available only on one side. This mode of communication facilitates the coding of some applications with dynamically changing data access patterns where the data distribution is fixed or slowly changing. One another advantage of RMA is that it can save the handshake time in a communication needed by point-to-point message-passing programming model.

Our BCL have been extended to provide RMA operations. So the DMPI's RMA operation implementation is directly based on the RMA operation of BCL. It can be partitioned into two parts: Window Management and RMA Communication.

In the module of RMA, all the communication operation is done through the object: "Window", a "window" represents a memory buffer that is allocated only for RMA operations. Window Management consists of such operations as Window Creation, Window Freeing, Setting Window attribute, and so on.

MPI_Win_create is a collective call executed by all processes in the group of comm. It returns a window object that can be used by these processes to perform RMA operations. The return of this function is a win object which record a win id and some other information include base address, win size and so on. The win id is not the true id of the win in the memory, but a index to the table: MPIR_Win_table. The true win id is return from BCL_Win_create, and is recorded in the table. From the index, we can entry the table and get the true win id. So when an window is create, the user can get a win id that presents the window. But the window's attributes can be changed during the processing In some case, even the true win id may be changed, and points to another memory buffer. But from the user's point, he operates still on a same window object. When a process has created a new window, it should synchronization its information to other processes in the same group. This operation also can be delay to do when the synchronization calls (e.g. MPI_Fence) are called. The Window's attributes can be modified during operating, including changing its base address, size, authority, and datatype.

RMA communication operation includes three types of communication calls: MPI_Put, MPI_Get, MPI_Accumulate and some different synchronization calls. MPI_Put transfers data from the caller memory (origin) to the target memory; MPI_Get transfers data from the target memory to the caller memory; and MPI_Accumulate updates locations in the target memory. These three calls are all non-blocking, and should cooperate with the synchronization calls, such as MPI_Fence and MPI_Start. The execution of a put operation is similar to the execution of a send by the origin process and a matching receive by the target

process. While the execution of a get is similar to the execution of a receive by he origin process and a matching send by the target process.

Because RMA operations are all done by one side, who has got both the source data address and the target address before he is ready to begin communication, so there will be no need to have some handshake for exchange data buffer address between sender and receive as before. Consequently, the implement of RMA communication functions can a little easier than that of point-to-point communication. But there still need some structure, which we call RMA_FIFO and RMA_HANDLE, to record the information of send request and receive request. BCL provide a common poll call both for point-to-point message passing and RMA to poll the send/receive complete events. When a complete event is polled, it will continue to do next work by the event type.

There should be a synchronization operation whether before or after a communication operation. The course within two consecutive synchronization is called access epoch. MPI provides three synchronization mechanisms:

- The MPI_WIN_FENCE collective synchronization call supports a simple synchronization pattern that is often used in parallel computations: all the process in the same communicator should synchronize.
- The four functions of MPI_WIN_START, MPI_WIN_COMPLETE, MPI_WIN_POST and MPI_WIN_WAIT can be used to restrict synchronization to the minimum: only pairs of communicating processes synchronize.
- Finally, shared and exclusive locks are provided by the two functions MPI_WIN_LOCK and MPI_WIN_UNLOCK: the target process don't synchronize.

The synchronization operation of DMPI is similar to the barrier operation of the collective calls, but it does more works. During synchronization, every process's window information for communication is broadcast to other process involved in the communication. For the communication calls of RMA are all non-blocking, there should be a call do the polling to decide when to end the communication. In the point-to-point message-passing model, this call is check_device. In RMA, after the communication calls is done, we call the check_device function in the synchronization operation.

Dynamic Process Create and Management

In the MPI-1 standard, an MPI-1 application is static; that is, no processes can be added to or deleted from an application after it has been started. MPI users have asked that the MPI-1 model be extended to allow process creation and management after an MPI application has been started. This functionality is realized in the MPI-2 standard in which a new module for dynamic process management is added.

We realized this functionality of Dynamic process management of DMPI in the DADI-E layer. The central works are done by RMS, which allocates and manages the dynamic process information. A balance loading can be provided with mpirun command when beginning to spawn new processes. This module can be divided into three parts: spawning new processes, establishing communication between two different applications, and resource control.

MPI users have two different functions to create new processes in runtime. The MPI_Comm_spawn function allows the creation of one or more process that will run the same executable with the same arguments. If the users wish to run different executables or to pass different arguments to the several processes they should use the MPI_Comm_multiple_spawn function. The spawning operation is collective over a certain intra-communicator. Only the process in the intra-communicator will be include in the inter-communicator that connect them to the new processes. A root process, indicated in the function's arguments, is responsible for actually creating the new processes. When one MPI's spawn function is executed, the new processes start their own MPI environment. Then the new environment and old environment are connected through the inter-communicator and users can exchange information between all processes.

The MPI-2 provides functions to establish communication in a client/server model. The server process opens a MPI port and waits the client processes to connect to it. The MPI port is an implementation dependent entry point that allows the two type processes to exchange information about the two environments to create the inter-communicator between the processes. To the user, it is opaque. The user can only know the port name, with which the client process can set up a connection with the server process, and establish a communication.

Resource control is not included in the MPI-2 standard because it was not able to design a portable interface that would be appropriate for the broad spectrum of existing and potential resource and process controllers. Resource control can encompass a wide range of abilities, including adding and deleting nodes from a virtual parallel machine, reserving and scheduling resources, managing compute partitions of an MPP, and returning information about available resources. Resource control of DMPI is totally implemented by RMS.

In DMPI, the resource control for dynamic processes is something like that of PVM. There are two main tables to keep the process's information. They are GPT and CPT. The GPT records the global information of process, through which both the communication within processes and the management of processes can become easy. The information includes the global process id, the physic node id, and the port number of processes. Every communicator has a CPT that keeps the information of processes in the communicator. The information kept in CPT mainly includes the global rank, local rank in the communicator, and the index to the GPT. CPT is static table. The CPT is created as soon as a communicator is created. However, the GPT is a dynamic table. Once a process is created or updated, it is needed to change the GPT to keep the information up to date.

The works for resource control are mainly done with these two tables. These works include table creation, initialization, updating and so on. It is not a bad idea to let a daemon to do these works just like the implementation of Lam MPI.

Performance and Analysis

The tests are done on our DAWNING-3000L, which consists of 70 Linux workstations. Each node is a 2-way 1GHz PIII(Coppermine) SMP, which is running

Turbo Linux 2.2.18-2smp. Two Myrinet M3S-PCI64B NICs are used on each node, the nodes are interconnected by M3S-SW16-8S switches.

The first figure shows the bandwidth of DMPI after optimized, compared with that of DMPI before optimized. Before optimization, the peak bandwidth is about 280MB/s when the data size is 128KB, which is only about 67% contrast to the bandwidth of BCL, 419MB/s. But the bandwidth of DMPI after optimized can reach 389MB/s, about 93% to that of BCL. The improvement is so obviously.

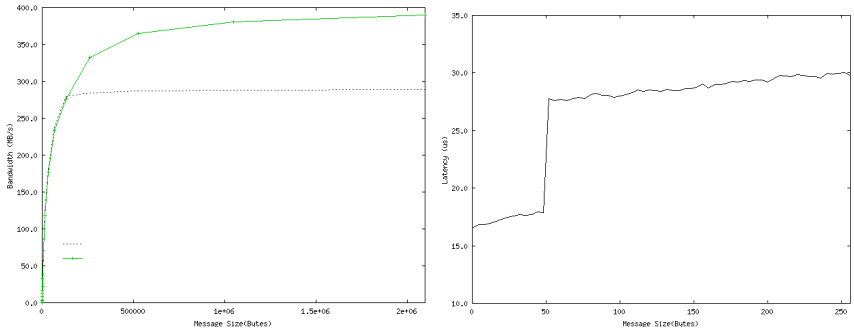

Fig. 3. Bandwidth and latency of DMPI

The second figure describes the latency of DMPI. It is about 16.3 us, while the BCL's latency is about 12us. Because our optimization aims only to improve the bandwidth of large data message, the latency keeps unchanged after the procedure is optimized.

Conclusion and Future Work

DMPI is ported from MPICH. Based on BCL, DMPI provides an efficient message-passing interface for a cluster for commodity SMPs. DMPI achieves 16.3 us of latency and 389MB/s of bandwidth for inter-node message passing.

With the characteristics of our system, we extended the ADI-2 to DADI-E that adds two functionality of dynamic process management and remote memory access. Furthermore, DADI-E provides common APIs to both DMPI and DPVM. We also use an arithmetic of pipeline-sending to have the procedure of point-to-point communication optimized. After the optimization, DMPI now can have a very high performance, very closely to that of BCL.

At the future work, we will continue to optimize the performance of bandwidth when sending data message which size is less than 128KB, and the performance of collective operation. Another emphasis may lies to upgrade current version to our next system that aims to be based on open grid architecture.

Reference

1. Kai Hwang, Zhiwei Xu, Scaleable Parallel Computing: Technology, Architecture, Programming, WCB/McGraw-Hill, 1998
2. Message Passing Interface (MPI) Forum Home Page, http://www.mpi-forum.org
3. Message Passing Interface Forum: MPI-2: Extension to the Message-Passing Interface. (June 1997), available at http://www.mpi-forum.org.
4. W. Gropp, E. Lusk, N. Doss, and A. Skjellum: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. Parallel Computing Vol.22, No.6, (September 1996).
5. W. Gropp and E. Lusk: MPICH Abstract Device Interface Version 3.3 Reference Manual Draft, Mar. 20, 2002.
6. MPICH Home Page, http://www-unix.mcs.anl.gov/mpi/mpich.
7. Xingfu Wu and Ninghui Sun: DPVM: PVM for Dawning Cluster Systems. Proc. of 4th International Conference/Exhibition on High Performance Computing in Asia-Pacific Region, 2000.
8. Ma Jie, He Jin, Meng Dan and Li Guo Jie: BCL-3: A High Performance Basic Communication Protocol for Commodity Superserver DAWNING-3000. Journal of Computer Science and Technology, 2001, 16(6): 522-530.
9. Meng Dan, Ma Jie, et al: Semi-User-Level Communication Architecture. In Proc. of IPDPS 2002, 2002.
10. H. Pedroso and J. Gabriel Silva: MPI-2 Process Create & Management Implementation for NT Clusters. Proc. of 7th European PVM/MPI User's Group Meeting, pp. 183-238, Balatonfured, Hungary (September 2000).
11. G. Burns, R. Daoud, J. Vaigl: LAM: An open cluster environment for MPI. Proc. of Supercomputing Symposium'94, pp. 379-386, Toronto, Canada (1994).
12. Nanette J.Boden, etc.: Myrinet: A Gigabit-per-Second Local Area Network. IEEE Micro, Feb, 1995.
13. Myricom. The GM Message Passing System, http://www.myri.com/GM/doc/, 2000