Final Report

Design Study for DPCL Port for MPICH on Linux And Performance Evaluation of SMG2000 Benchmark on Myrinet Cluster

Team Members

Kunal Shah
Anubhav Dhoot

Table of Contents

Introduction 4
DPCL System Architecture 4
PoeAppl Class Design 5
Design Details 7
  PoeApplD 7
  PoeAppl 7
Limitations of this design for use with MPICH 9
Proposed Changes In Mpich 10
Proposed *MpichAppl* Class Design 10
  MpichApplD 10
  MpichAppl 11
Implementation and Feasibilty tests 11
Conclusions 12
References 13

Acknowledgements

We are thankful to Dave Wootton (Project lead for DPCL Linux port) for providing us with insights on the design details of the DPCL and Rusty Lusk (one of the designers of MPICH) for providing us with some of the details of MPICH implementation. We also thank Dr. Mueller for helping us out with some problems that we faced.

DPCL SUPPORT FOR MPICH ON LINUX

Introduction

DPCL is a C++ class library whose application programming interface (API) enables a program to dynamically insert instrumentation code patches, or "probes", into an executing program. The program that uses DPCL calls to insert probes is called the "analysis tool", while the program that accepts and runs the probes is called the "target application". The ability of DPCL to dynamically insert probes relieves the need to recompile the code, allows switching from one tool to other without restarting the application and a slew of other advantages that can be found in [3] and [4].

DPCL System Architecture

The following figure taken from DPCL user guide documentation explains the way DPCL analysis tool instruments a parallel application. A DPCL daemon, which is created per user on each different node is responsible for communication co-ordination between the DPCL analysis tool and the target application process. When an analysis tool tries to connect to a target application, a DPCL superdaemon (dpclSD) is created, if it is not already running. Then, the dpclSD creates a DPCL Daemon (dpclD) if there is no DPCL daemon already running for this user. dpclSD is also responsible for user authentication on remote host. dpclD then performs much of the work requested via DPCL function calls by the analysis tool. It also relays data collected by the instrumentation probes within the target application back to the analysis tool.

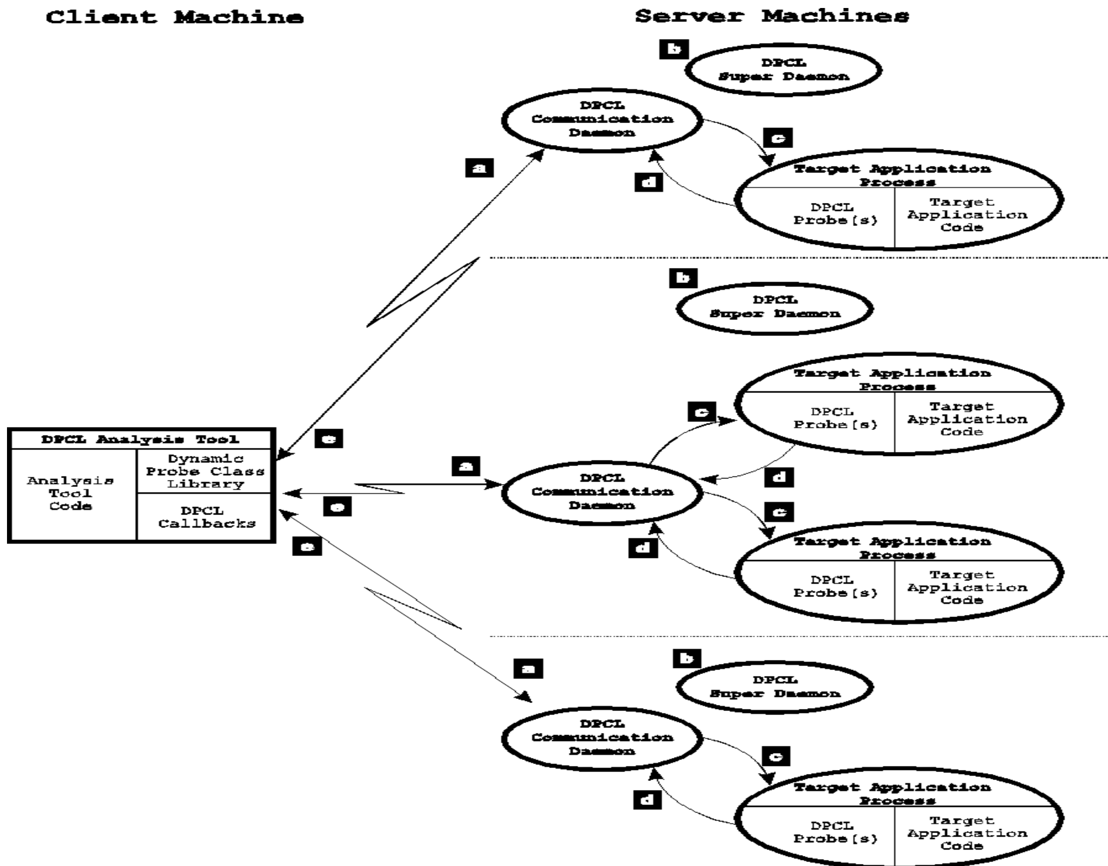

Figure 1. DPCL System Architecture

The DPCL port on Linux doesn't support instrumentation of MPI applications. The class which handles MPI application on IBM AIX is the *PoeAppl* class.Hence to provide support for MPICH on Linux, we need to replicate much of the functionality provide by PoeAppl class, taking into consideration the differences between POE and MPICH and other runtime environment differences between Linux and AIX. Our design takes into account these differences as well as the current limitations of MPICH and proposes some changes not only in the *MpichAppl* class, but also in MPICH.

First, we describe the way *PoeAppl* class has been designed and handles parallel processes. Then, we point out the limitations of this design for MPICH and the changes needed in MPICH. Finally, we propose a design of the *MpichAppl* class so that the analysis tool can use it very much the same way it uses PoeAppl for instrumenting parallel processes running under POE.

PoeAppl Class Design

This is a C++ class derived from *Application* class, which is responsible for providing convenience functions for connecting to or starting a job in the Parallel Operating Environment (POE). It makes the instrumentation of a parallel application transparent to the analysis tool by hiding the differences in the methods required for instrumenting serial and parallel jobs. In POE, a parallel application (called POE job) is executed from a home node, and POE will allocate host machines on which the various processes of the POE job will run. *PoeAppl* class makes use of the information available from the run time environment for automatically finding out the various host machines on which the POE processes are running and their process ids. Once it has this information available, it will create Process objects for each of the POE processes and will encapsulate all of them. It then functions the same way as an *Application* class object works. The *Application* class is a grouping of related *Process* class objects. By grouping a number of *Process* objects under an *Application* object, the analysis tool is able to manipulate a set of related UNIX processes (represented by the *Process* objects) as a single unit. More information about it can be found in [1] and [2].

The following figure explains the various possible states a Process object can have and the functions which lead to transition from one state to another.

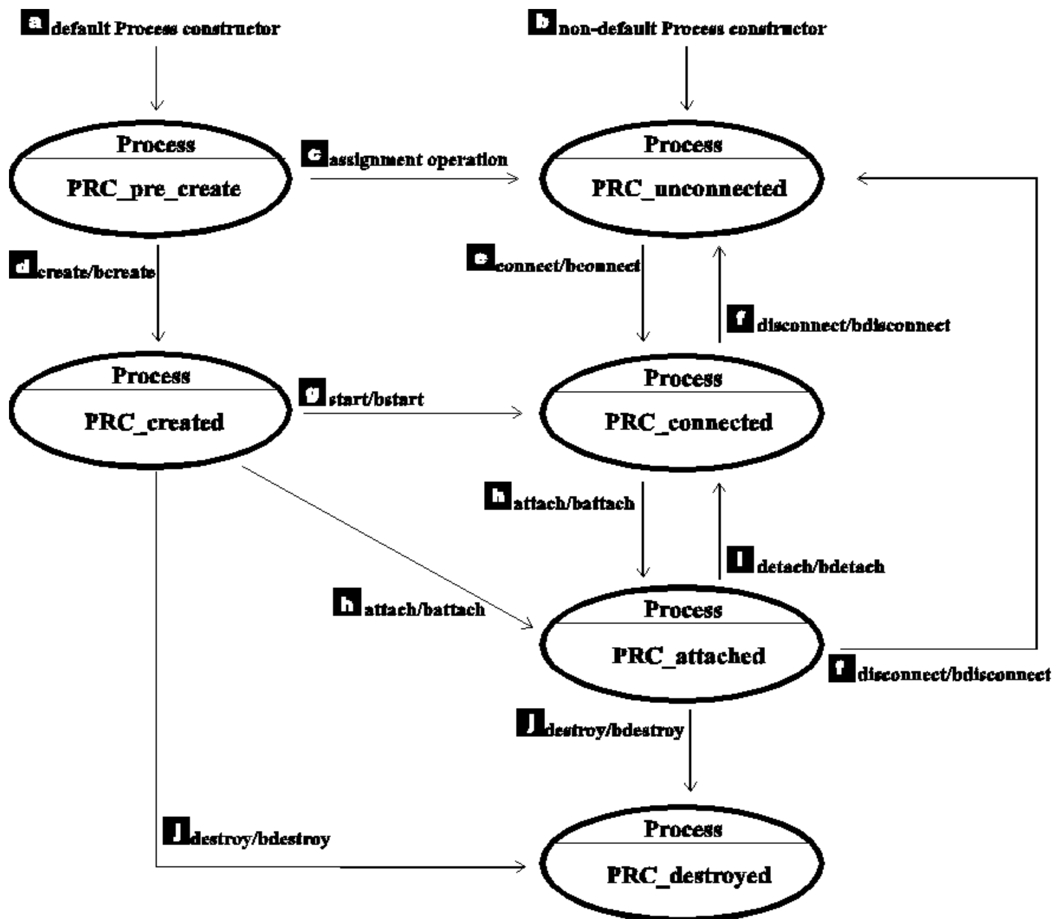

Figure 2. Process State Diagram

Since *PoeAppl* is derived from Application, it inherits all the methods contained in Application, and overrides some of them. The methods of interest for the purpose of providing MPICH support are

- *create()*

  It is responsible for creation of POE application in suspended state. At the completion of this operation, the *PoeApp* object will contain Process objects that represent the various processes of the POE application. The analysis tool can insert probes into this application and then start its execution.

- *bcreate()*

  blocking create (synchronous).

- *init_procs()*

  It initializes an empty *PoeAppl* object to contain Process objects representing a particular POE target application's processes. It is used by the analysis tool to connect to an already

running POE application. i.e. it first calls init_procs() followed by connect()/bconnect() on the *PoeAppl* object.

- *binit_procs()*

  blocking init_procs (synchronous).

Design Details

PoeApplD

This is the Poe application daemon which is responsible for reading the Poe configuration file (task list file). This file contains information about:

- Machines on which the poe processes are running
- Pids of the poe processes
- Task numbers of the poe processes

and some other information such as task session id and the name of the executable for the poe process. This task list file is created for each POE job and is found as /tmp/.ppe.<poe_pid>.attach.config

PoeAppl

- *init_procs()*

  The flow diagram for the init_procs method is shown below. It first creates a Processs object for the poe process using the hostname and its pid information. It then communicates with the *PoeApplD*. Before it actually initiates *PoeApplD*, init_proc registers a callback with it that will be invoked once the daemon is done reading the Poe task list file.

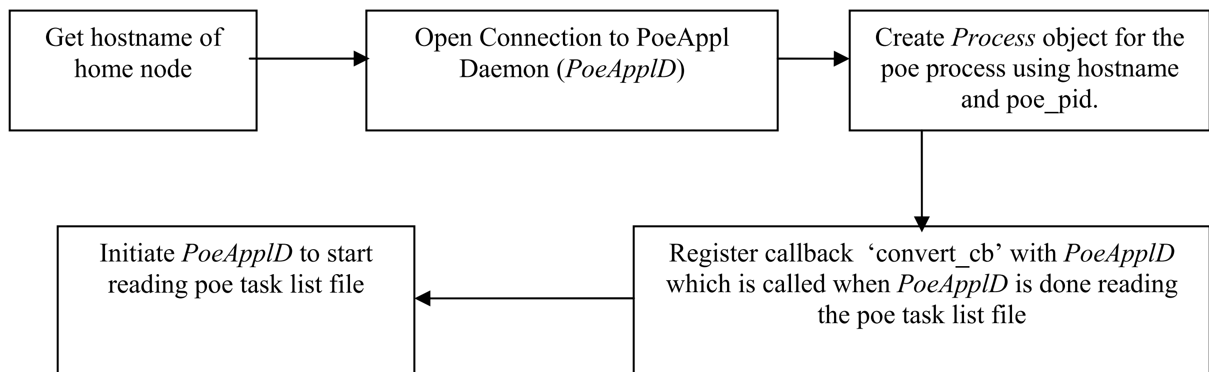

Figure 3.Flow diagram for init_procs(homenode,poe_pid)

- *convert_cb()*

  This is the callback that is invoked once the poe task file has been read. It is responsible for creating the Process objects for each Poe process. These Process objects are added to the *PoeAppl* object, which is then ready to be used by the analysis tool.

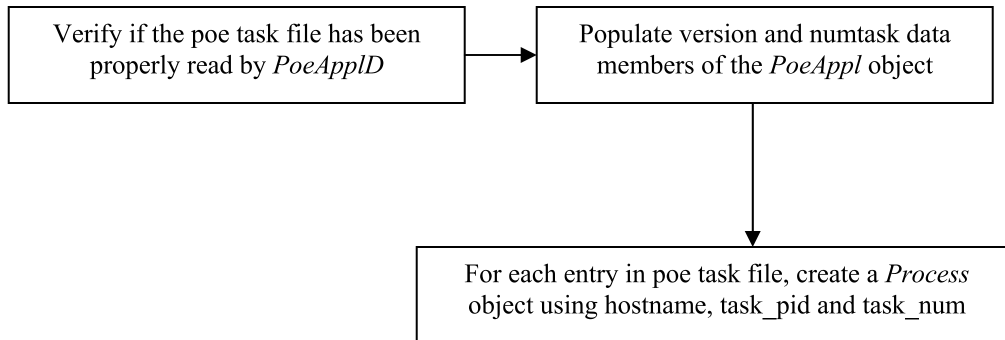

Figure 4. Flow Diagram for convert_cb

- *create()*

  This creates a poe application. This includes creating poe itself on the host specified and having poe start the individual application processes in the stopped state. dpcl will then attach to them. This involves the following important steps.

  1. adding the _MP_DBG_STOPTASK env variable to poe so that the application processes start under ptrace control and they stop at the first executable instruction
  2. creating the poe home node process
  3. Issuing init_procs to get locations of application processes and add them to the PoeAppl's list
  4. Connecting to the poe app processes

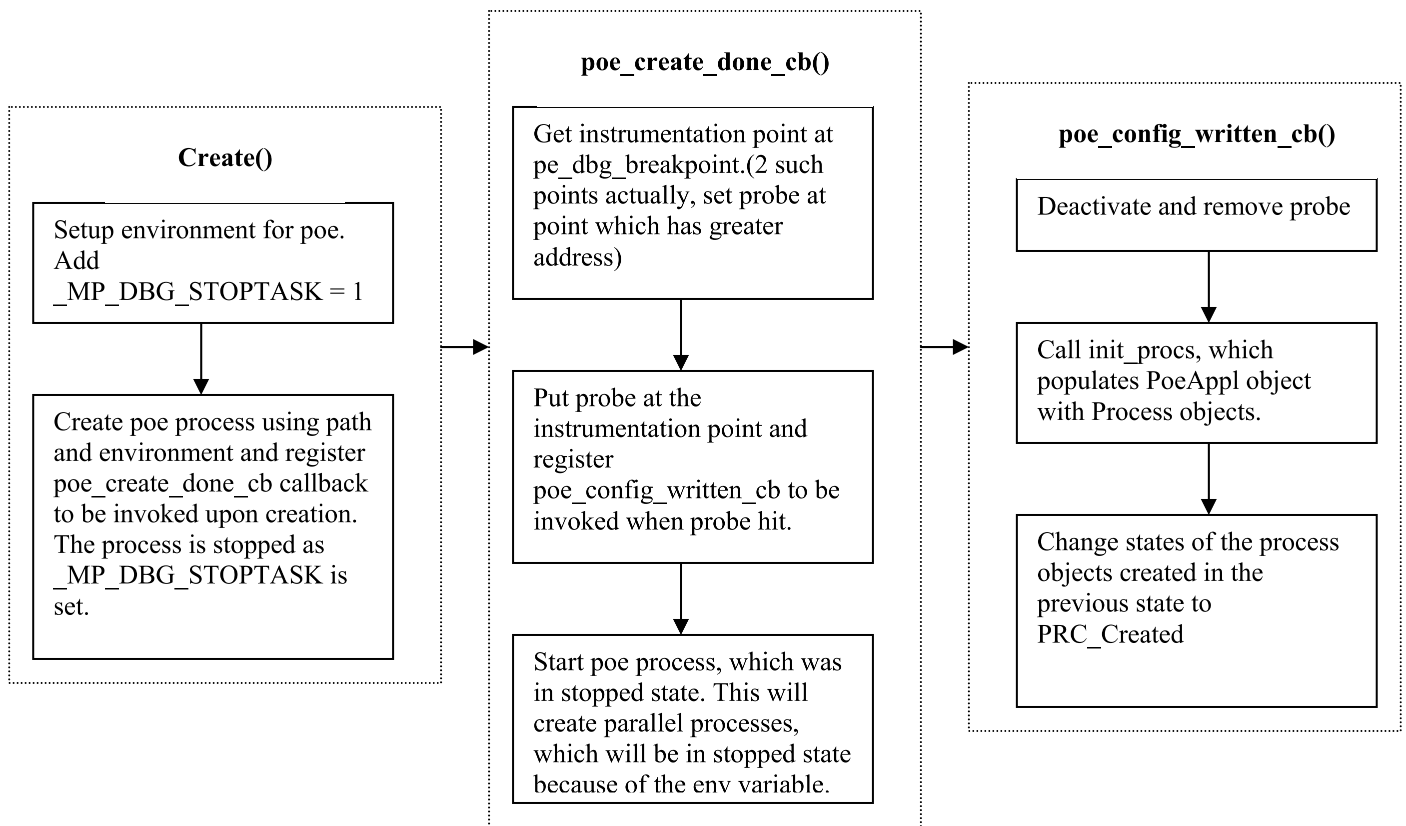

Figure 5. Flow diagram for create(host, path, argv, envp. ……)

- *binit_procs()*

  This internally calls init_procs and waits until a response is received.

- *bcreate()*

  This internally calls create and waits until a response is received.

Limitations of this design for use with MPICH

- As of the current implementation of MPICH, there is no support for querying the process manager to get the list of nodes on which the mpi processes are running and their process ids.

  In an upcoming version of MPD process manger for MPICH2, there will be a way to inquire this of a running program. The way it would work is that that the job would be given a name (alias), or it would return an id, when the job is run with mpiexec. Then the process manager can be queried about this job, specifying it by either id or alias, and it would return the ranks, hosts, and pids for the job.

- It is not possible to start MPI Processes in a stopped state. Hence, it is not possible to support create/bcreate() methods.

Proposed Changes In Mpich

As mentioned earlier, in the current MPICH implementation, no task list file is written. However, for the ch_p4 device, there is a structure called 'proctable' that is maintained which keeps a track of the pids of the remotely started processes along with the hostnames. The loop which fills in this proctable can be modified so that it writes all the records to the MPICH task list file, containing task index and the hostname and pid of each task. This file should be ended with a special end of file delimiter once the loop terminates. ***We were able to get this working by modifying the file /home/kdshah/mpich-1.2.5/src/env/initutil.c .*** A cleaner solution to this problem is to have mpirun accept a –mpirun <pid> option. The mpirun script will set an evironment variable in the master task's environment pool with the value of pid of the mpirun script. MPIR_Init function which is called by MPI_Init will be modified to handle the –mpirunpid option. The pid specified with this opiton will be used to create the task list file named /tmp/mpich_appl_<pid>. mpirun script will be modified to remove the /tmp/mpich_appl_<pid> file after the application terminates. This file will always be written and the environment variable will always be set since the invoker of mpirun may not anticipate DPCL connection to the running application.

The current implementation of mpich does not provide any method to create a process and suspend it at entry to main. To get around this, mpirun script will be modified so that it accepts –dpcl as an option. When mpirun is invoked with –dpcl flag, it will set an environment variable which will indicate that all application tasks are to be suspended in MPI_Init processing in order for DPCL to be able to attach to those tasks. The way this will work is by modifying the MPIR_Init function which is called by MPI_Init to check this environment variable in all tasks. If this environment variable is set, a loop will be entered which periodically queries a global flag MPIR_debug_gate to determine if the loop should be exited. This variable will be set by DPCL when it successfully attaches to the process and runs a one-shot probe intended to set the flag. However, with this scheme, if DPCL detaches from the application before the variable is set, then it may not be possible to ensure that all application tasks are terminated. The other implication is that only those instrumentations which execute after MPI_Init will work correctly. Those installed at points prior to MPI_Init will never be executed.

Proposed *MpichAppl* Class Design

MpichAppl is our proposed class and it is based on PoeAppl Class. This class will be used by the client exactly the same was as it uses *PoeAppl* class for instrumenting a parallel process in POE. In order for *MpichAppl* to provide the same functionality as *PoeAppl*, a mechanism is needed for obtaining the hosts and pids of processes. Below, we describe the way this can be achieved and also some other changes that need to be made to the *PoeAppl* class.

MpichApplD

This is the MpichAppl daemon, and this class will be based on *PoeApplD* class. The only function in this class is read_POEconfig, which will be renamed to read_MPICHConfig. This function will need to poll for the existence of the config (task list) file, as it is not possible to use probes (see figure 5) in mpirun (as it is a script and there is no support for instrumenting shell scripts) to indicate that the file has been completely written. The other issue that needs to be dealt with is determining whether the file has been completely written or not. To overcome this, the

read_MPICHConfig will repeatedly read the file until it encounters a special end of file delimiter. Once the file is completely read, the data will be returned to the caller.

MpichAppl

This class will be based on the *PoeAppl* class.

- *init_procs()*

  The process of connecting to an existing MPICH application will be essentially the same as connecting to an existing POE application (see figure 3 and figure 4).

- *convert_cb()*

  convert_cb() callback function needs to properly parse the MPICH task list file created by MPICH. So that the existing parsing code for POE can be reused, the format of the task list file generated by MPICH should be the same as that generated by POE.

- *create()*

  The process of creating a new MPICH application will follow the same basic model as for creating a new POE applicaton. However, some changes need to be done due to difference between MPICH and POE.

  The code which sets up the _MP_DBG_STOPTASK will be removed. A *Process* object for representing the mpirun script will be created, and mpirun script will be invoked with the **-dpcl** flag (this will cause MPICH to suspend all tasks in MPI_Init) and other arguments. This is similar to the PoeAppl::create function creating the POE home task. However, since mpirun may be a shell script, DPCL cannot insert a probe to detect when the MPICH task list file has been created. This is taken care of by the *MPICHApplD::read_MPICHConfig* class which polls for the existence of the MPICH task list file and returns only when that file has been completely read or a reasonable time out occurs.

  Mpich_create_done_cb will be same as poe_create_done_cb, except that it does not install probes for detecting the creation of the task list file. It will poll for the existence of the task list file, very much like the *MpichApplD* and does not exit until the file has been completely written. When this function exits, DPCL will have connected to all the application tasks.

  The function start() will be overwritten. This will load and invoke a one-shot probe on each task in the application. This one shot will set the environment variable MPIR_debug_gate, which will allow task execution to continue past the point of MPI_Init.

Implementation and Feasibilty tests

- Installed MPICH on os29 and os30.
- Installed rsh server required by MPICH on os29 and os30.
- Used dpcl to profile simple serial test program and two parallel processes running on same and separate nodes.
- Wrote an analysis tool to instrument a simple MPI test program.
- Modified a MPICH source file to dump the task lists to STDOUT.

We installed the MPICH ch_p4 version on the linux machine. Initially the compilation gave an error. After removing the initial path which was set for the Cluster, we did make again which succeeded this time. Then when we ran a few test programs, but it did not work because rsh was not installed. Hence we then installed and enables rsh server. After this the sample programs which used communication started working.

We were able to use DPCL for simple serial programs. DPCL requires the hostname and pid of the executing processes so that it can *connect* to it. As explained in our report, for MPI support, it needs the same for all master and worker processes. As that is not provided by MPICH by default, DPCL can't connect to it. However, since the MpichAppl class would be inherited from the Application class, we wrote an analysis tool for instrumenting two processes on a single node. The analysis tool was successful in instrumenting the two processes. Then, we tried running these processes on two separate nodes. However, due to a bug in the current DPCL implementation in the way a socket connection is established between the analysis tool and the dpcl daemon, the analysis tool did not succeed in attaching to the process. We identified this bug and modified one of the source files (/root/kunalanu/dpcl/src/lib/src/Daisd.C). After this, we were successful in instrumenting processes running on separate nodes with a single analysis tool.

Applying the same concept, we tried to instrument an MPI application. Note that we explicitly specified the process ids of the MPI processes started. But we encountered that communication failed while connecting to the MPI processes which was due to the DPCL daemon aborting. Upon investigating the reasons for that, we found that the target application was not being compiled with the –gdwarf-2 flag and some compile and link flags were required to link the target with DPCL support routines. After these flags were specified, we could successfully instrument the MPI application.

Next, our goal was to get the process ids and hostnames from MPICH so that we need not specify them explicitly. We did find the structure, as suggested by one of the designers of MPICH, where the process ids for each of the master and worker MPI processes are stored, and we were able to dump it to the screen. This is done in the MPIR_Init function in the file /home/kdshah/mpich-1.2.5/src/env/initutil.c.Now we could get the pids and hostnames which can be used by DPCL to connect to the MPI programs so that it can be instrumented.

Conclusions

From the above two tests viz.

1) Changing the MPICH source file to write the task list to a file
2) Instrumenting an MPI application by explicitly specifying the process ids and hostnames

We can conclude that it is possible to create an *MpichAppl* class which makes use of the task file and uses the process ids and host names to build Process objects which represent the parallel processes. We haven't still conducted a test for suspending MPI processes in MPI_Init(), which is required by the MpichAppl::create() function. However, this should be trivial as explained in "Proposed changes to MPICH" section.

References

[1]
http://oss.software.ibm.com/developerworks/opensource/dpcl/doc/dpclpg/dpclpg08.html#HDRAPPLICC
[2]
http://oss.software.ibm.com/developerworks/opensource/dpcl/doc/dpclref/dpclref07.html#HDRCH3
[3] http://oss.software.ibm.com/developerworks/opensource/dpcl/
[4] http://www.ptools.org/projects/dpcl/