Parallel Debugger

Luthra Milind Vikas 150363 Subhdeep Saha 150732

I. Problem Statement

While there are a plethora of debuggers for serial code - every IDE has one, and of course there is `gdb` - it is quite hard to debug parallel code, especially at the student level, and we are forced to use inelegant mechanisms like inserting `printf` functions in our code and re-running them. Our project focuses on making an open-source parallel debugger well-suited to educational purposes.

II. Previous Work

There are several existing debugging methods that can be used for debugging MPI based programs.

A. Commercial Alternatives

There are two commercially developed programs that facilitate parallel debugging, Arm DDT (Allinea DDT) [1] and TotalView for HPC [2]. Both of these tools are feature rich, working for not just MPI but also OpenMP, CUDA etc, and having support for basic Linux systems to supercomputers. These provide pretty GUI tools for visualization as well.

However, these tools are very expensive. Allinea DDT costs around $600 for one licence which is further limited to a certain number of processes, and TotalView does not even list a cost outright, only the company can be contacted for evaluation of the software. Both these are targeted towards enterprise/large-scale users, and not those who are learning parallel computing. Further, both these tools are closed source, which limits access to their source code, and we cannot modify it as per our needs.

B. Using xterm and gdb

Another way to debug is to run all your processes inside `gdb` and then open all of them inside different `xterm` instances [3], and then debug them each individually. This strategy does not scale to more than a handful of processes, because it is very tedious to keep changing terminals while trying to debug. In certain cases, this will not even work for processes distributed amongst nodes.

Clearly, this is too limited to be of any use once any serious debugging is needed.

III. Overall Project Idea

Our idea is based on the idea given above in [3], through multiple `gdb` instances. However, instead of forcing the user to manually deal with each of the processes, we use a centralized server that connects to and orchestrates each process. The server does additional bookkeeping as well, to track some common errors that users are often prone to making.

The project is written almost entirely in Go, and communication between processes and the server is through TCP sockets. An overview of the project in Fig 1 describes the various roles that will be discussed in detail in Section V

IV. Features

There are four main features of the project, whose implementation is discussed in detail later.

- **Client-Server Orchestration**: Our code instruments the target process, connects the client/server together, and makes possible controlling remotely the gdb instances. Commands can be sent to all or specific rank(s).
- **TUI**: We provide a terminal-based user interface, suitable for use on the CSE cluster. We group outputs by the ranks.
- **Collective Tracking**: Our code does some book-keeping to track collective calls, and find out which calls are pending at which ranks. This is useful for detecting common errors where the call to some collective is missing.

V. Implementation Details

A. gdb/mi Interface

`gdb/mi` is a machine-readable interface to the debugger. As opposed to the usual back and forth interaction that one has with the CLI, this can be thought of as two distinct streams: one in which we can write the input commands to be run, and the other an output stream, which runs asynchronously.

For example, calling `continue` in the CLI interface will cause the program to run (blocking any other commands) till a breakpoint is hit or the program ends. In MI interface, calling the equivalent of `continue` will return immediately, and tell us that the program is running. A stop notification is generated distinct to the `continue` call.

There is a Go library [4] which allows us to interact with the MI interface. We used that with a slight modification [5], to allow for commands in the usual CLI format as well as the MI format that the library permits.

The file of interest is `gdb.go`. Here, we will highlight a few things. First, how we have implemented a synchronous (blocking) method to send commands - it involves two functions, `SynchronizedSend` and `handleNotifications`. The first function is the exposed API for the `GDBInstance` object, while the second is an event handler that is automatically called by the `gdb` library when an asynchronous notification is received.

Listing 1. Partial pseudocode for Synchronized Send

```
1 func handleNotification(notification) {
```

```
    switch notification["class"] {
      case "running":
        bkptHitNotification.wait()
      case "stopped":
        bkptHitNotification.notify()
      }
    }

    func SynchronizedSend(message) {
      result = internalSend(message)
      handleNotification(result)
    }
```

In explanation, consider the following sequence:

- We run a command like `continue`.
- `internalSend` returns a result - of the same type as a notification, with the class as running. We call `handleNotifications` and the thread waits on `bkptHitNotification`.
- When `continue` eventually runs and the program stops again, we get a notification again, with the class stopped. Then, the `bkptHitNotification` is notified, and the whole command is thus synchronized.

The second thing to be highlighted is the presence of hooks. An example of a hook can be found in `pd-client/main.go:51`, where the console output is sent to the server so that it can be echoed for the user.

Listing 2. Custom hooks for notifications

```
...
gdbInstance.AddNotificationHook(
func(notification) {
  if notification["type"]=="console" {
    // On getting a console notification,
    // relay it to the server.
    sendToServer(payload)
  }
  return true
})
...
```

B. Start-up Instrumentation

Starting up involves the connection of the server with all the clients, and the server needs to know the rank of each of the process running inside each of the clients as well.

As soon as the `pd-client` starts up, it connects to a `pd-server` (the host/port are command line arguments) and starts up an instance of `gdb`. Then, the target binary is loaded into this debugger instance. While loading the binary, we make use of `LD_PRELOAD`. We have a shared library distributed with the client, which provides an implementation for `MPI_Init`. This custom implementation calls `PMPI_Init`, and then writes the rank of the process and the size of the WORLD communicator to a file.

Listing 3. Start-up instrumentation

```
    int MPI_Init(int *argc, char ***argv) {
      ...
      return_code = PMPI_Init(argc, argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      write_to_file(filename, rank, size)
      return return_code;
    }
```

We insert a breakpoint at the end of this function using `gdb`, and run the program. We access the file written by the custom implementation inside Go code to extract rank/size information and send to the server. When all of the processes have completed these steps, then the start-up is complete.

Listing 4. gdb.go: InitGdb

```
func InitGdb(debugTarget string) {
  ...
  SynchronizedSend("file", targetBinaryFilename)
  // Set up gdb environment containing LD_PRELOAD
  // and random filename FILENAME.
  SynchronizedSend("break PMPI_Init")
  SynchronizedSend("run")
  SynchronizedSend("finish")
  SynchronizedSend("finish")
  SynchronizedSend("clear PMPI_Init")
  ...
  // Send to server data read from FILENAME.
}
```

This is also the reason why our debugger starts with a `continue` at `MPI_Init`, rather than a `run` before the `main` function.

C. Collective Call Tracking

We also support tracking collective calls made on `MPI_COMM_WORLD`. This is a two-fold process. The first step is the use of `PMPI` functions. We write custom collective functions, in which we fetch the rank of the calling process before invoking the actual function.

We have created a few macros so new collectives can be added easily to tracking. One generates an empty function on which we can break on, and the other fetches the rank and stores it so we can read it from `gdb`. We don't do anything with this rank inside the custom function, and we don't use the empty function at all, so we use the `volatile` keyword to make sure that the compiler does not optimize them away.

Listing 5. Collective call instrumentation

```
_GENERATE_INTERNAL_METHOD(MPI_Barrier);
_GENERATE_EXT_METHOD(MPI_Barrier,(MPI_Comm comm), (comm));
```

If tracking of collective calls is enabled, we set breakpoints on all the collectives to track. Once we hit the breakpoint, we step through, and use the `volatile` rank to do the necessary book-keeping in terms of what rank called what collective. This way, at any point, we have a list of all the ranks currently pending on some collective on `MPI_COMM_WORLD`.

Listing 6. Pseudocode for processing collectives inside Go

```
func processBkpt(collective) {
  if !tracking(collective) {
    return
  }
  SynchronizedSend("finish")
  result = SynchronizedSend("-stack-list-variables 1")
```

```
7     communicator, rank = process(result)
8
9     if communicator != MPI_COMM_WORLD {
10      return
11    }
12    sendToServer(collective, communicator, rank)
13    SynchronizedSend("continue")
14  }
```

One major caveat of using custom MPI functions is that we can't set breakpoints on them anymore. So, it's not possible to break on `MPI_Send`, rather you need to break on `PMPI_Send`.

D. TUI

Listing 7. Struct for the complete interface

```
1     type TUI struct {
2         root          *tui.Box
3         ui            tui.UI
4         clients       map[int]*tui.Box
5         Input         *tui.Entry
6         clientParent  *tui.Box
7         conn          map[int]*net.Conn
8         numOfClients  int
9         history       map[int][]string
10        cmdHistory    []string
11        histPtr       int
12
13  }
```

We have used a Go library [**?**] to implement the TUI for our debugger. Here `root` is the parent container which can hold the window for every client which is connected to the server. The interface is implemented in a decoupled fashion where all the functions which are required to update the screen are mentioned in a separate file `pd-server/tui`. Therefore we can extend the tui by just adding functions to the library.

VI. Future Work

There is potentially a large amount of future work that can be done in this project.

- **GUI Interface/Visualization**: Instead of a TUI, the project is very amenable to having a GUI front-end without too many code changes (except that of the GUI itself, of course).
- **Communicator Tracking**: Tracking what process belongs to what communicator is slightly more challenging. We tried doing it, but had some problems. The basic idea is that each method which leads to communicator creation (`MPI_Comm_dup`, `MPI_Comm_split` etc) can be instrumented, and the processes with the same `newcomm` are in the same communicator. The issue here is that the same `newcomm` might have different representations across processes, and thus, is impossible to compare.
- **Collective Calls Across Communicators**: If the above is implemented, the code for tracking collective calls can be modified to track calls not just for `MPI_COMM_WORLD`, but for any communicator.

VII. Appendix

This section details the division of work amongst us.

- Making Interface to GDB: Both
- Start-up instrumentation, setting up communication: Both
- TUI: Subhdeep
- Collective Tracking: Milind

References

[1] "Arm DDT/Allinea DDT," https://www.arm.com/products/development-tools/server-and-hpc/forge.

[2] "TotalView for HPC - RogueWave Software," https://www.roguewave.com/products-services/totalview.

[3] "OpenMPI FAQ: Debugging Applications in Parallel," https://www.open-mpi.org/faq/?category=debugging#serial-debuggers.

[4] "cyrus-and/gdb," 1https://github.com/cyrus-and/gdb.

[5] "milindl/gdb," https://github.com/milindl/gdb.

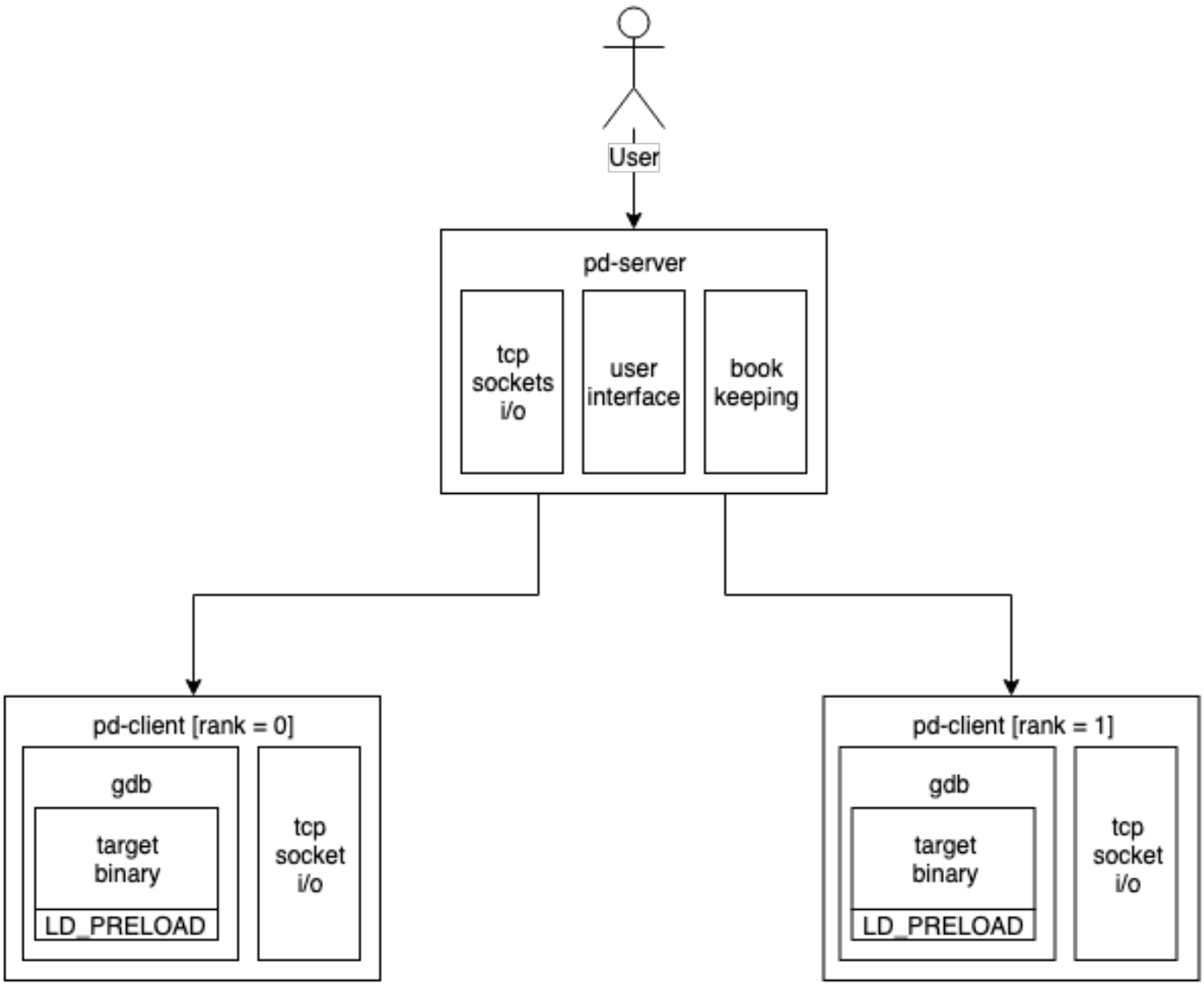

Fig. 1. High level overview of how the parallel debugger works