CS633 Project: Parallel Debugger

Milind Luthra (150363) Subhdeep Saha (150732)

26 April 2019

Table of Contents

Table of Contents

Motivation

```
===================================================================================
=   BAD TERMINATION OF ONE OF YOUR APPLICATION PROCESSES
=   PID 21523 RUNNING AT hostname
=   EXIT CODE: 6
=   CLEANING UP REMAINING PROCESSES
=   YOU CAN IGNORE THE BELOW CLEANUP MESSAGES
===================================================================================
YOUR APPLICATION TERMINATED WITH THE EXIT STRING: Aborted (signal 6)
This typically refers to a problem with your application.
Please see the FAQ page for debugging suggestions
```

Table of Contents

Allinea DDT and Totalview

- Debuggers already in use to debug large parallel applications.

Allinea DDT and Totalview

- Debuggers already in use to debug large parallel applications.
- Both have a rich feature set and GUIs.

Allinea DDT and Totalview

- Debuggers already in use to debug large parallel applications.
- Both have a rich feature set and GUIs.
- However, both are proprietary, commercial software.

Allinea DDT and Totalview

- Debuggers already in use to debug large parallel applications.
- Both have a rich feature set and GUIs.
- However, both are proprietary, commercial software.

Allinea DDT and Totalview

- Debuggers already in use to debug large parallel applications.
- Both have a rich feature set and GUIs.
- However, both are proprietary, commercial software.
- Restrictive licenses (locked to one node, or four processes etc) and high cost (a few hundred dollars).

Allinea DDT and Totalview

- Debuggers already in use to debug large parallel applications.
- Both have a rich feature set and GUIs.
- However, both are proprietary, commercial software.
- Restrictive licenses (locked to one node, or four processes etc) and high cost (a few hundred dollars).
- Can't be extended any further.

Using XTerm and GDB

- Possible Idea: For n processes, launching n XTerm instances with gdb.

Using XTerm and GDB

- Possible Idea: For n processes, launching n XTerm instances with gdb.
- Each terminal can be used to debug the individual processes.

Using XTerm and GDB

- Possible Idea: For n processes, launching n XTerm instances with gdb.
- Each terminal can be used to debug the individual processes.
- `mpiexec -n 4 xterm -e gdb ./test`

Problems with XTerm + GDB

```
mpiexec -n 30 xterm -e gdb ./test
```

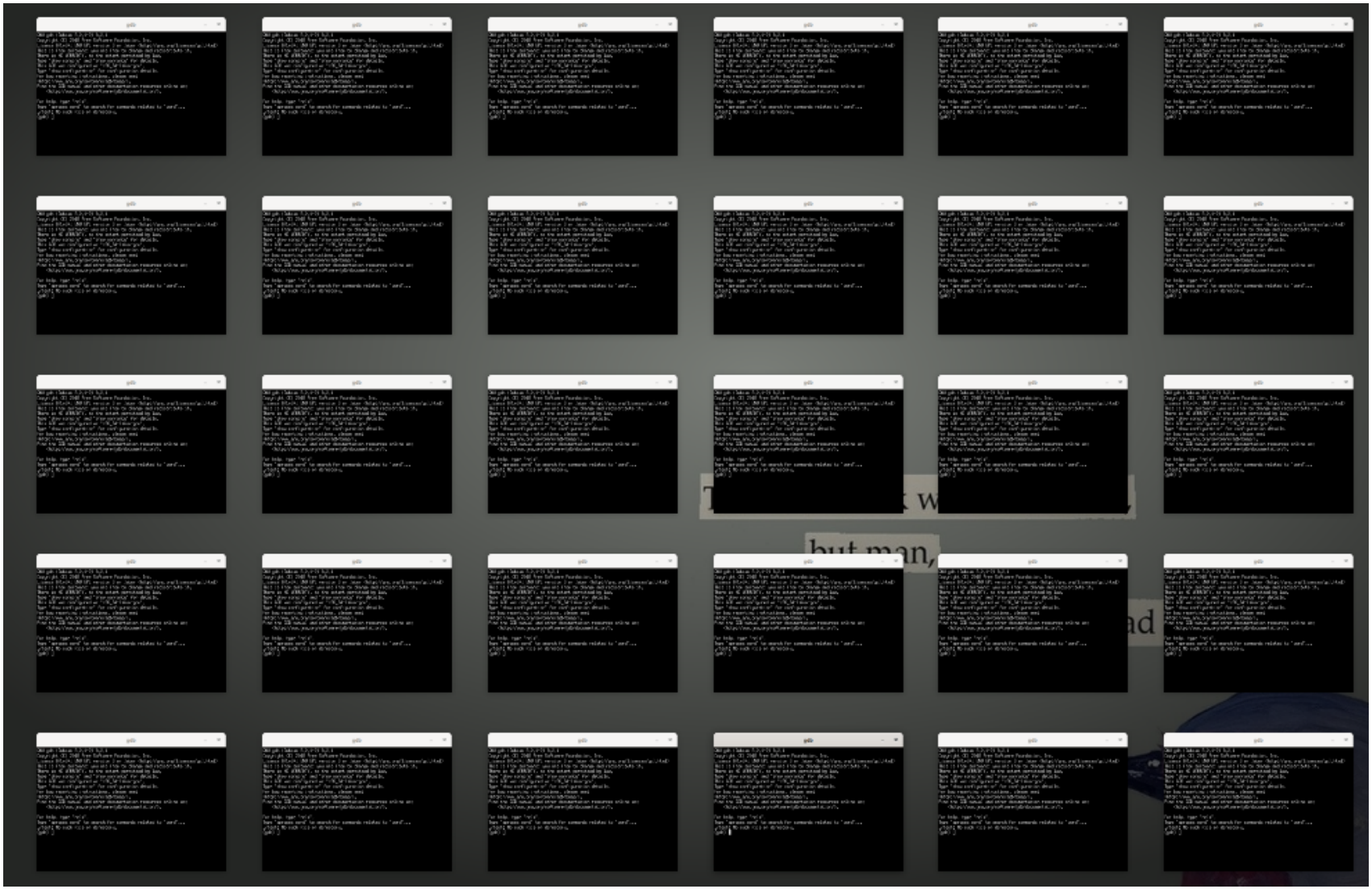

Table of Contents

Basic Idea

- The basic idea is to launch a number of clients on nodes using `mpiexec`.

Basic Idea

- The basic idea is to launch a number of clients on nodes using `mpiexec`.
- Each client instance will run a `gdb` instance with the program to be debugged.

Basic Idea

- The basic idea is to launch a number of clients on nodes using `mpiexec`.
- Each client instance will run a `gdb` instance with the program to be debugged.
- All the instances of `gdb` are controlled using a single, centralized interface.

Table of Contents

Timeline

- Load and then instrument target binary in all nodes. DONE

Timeline

- Load and then instrument target binary in all nodes. DONE
- Communicate with `gdb/mi`. DONE

Timeline

- Load and then instrument target binary in all nodes. DONE
- Communicate with `gdb/mi`. DONE
- Set up client and server communication (extract rank/size from instrumented binary). DONE

Timeline

- Load and then instrument target binary in all nodes. DONE
- Communicate with `gdb/mi`. DONE
- Set up client and server communication (extract rank/size from instrumented binary). DONE
- Allow commands from server to all clients, and echo client console on server. DONE

Timeline

- Allow sending commands from server to only specific clients DONE

Timeline

- Allow sending commands from server to only specific clients DONE
- Make a nice TUI. DONE

Timeline

- Allow sending commands from server to only specific clients DONE
- Make a nice TUI. DONE
- Track communicator data. NOTDONE (we tried this, problems explained later.)

Timeline

- Allow sending commands from server to only specific clients DONE
- Make a nice TUI. DONE
- Track communicator data. NOTDONE (we tried this, problems explained later.)
- Track collective calls. DONE (for `MPI_COMM_WORLD`)

Timeline

- Allow sending commands from server to only specific clients DONE
- Make a nice TUI. DONE
- Track communicator data. NOTDONE (we tried this, problems explained later.)
- Track collective calls. DONE (for `MPI_COMM_WORLD`)
- Track asynchronous call buffers. NOTDONE

Table of Contents

Architecture Overview

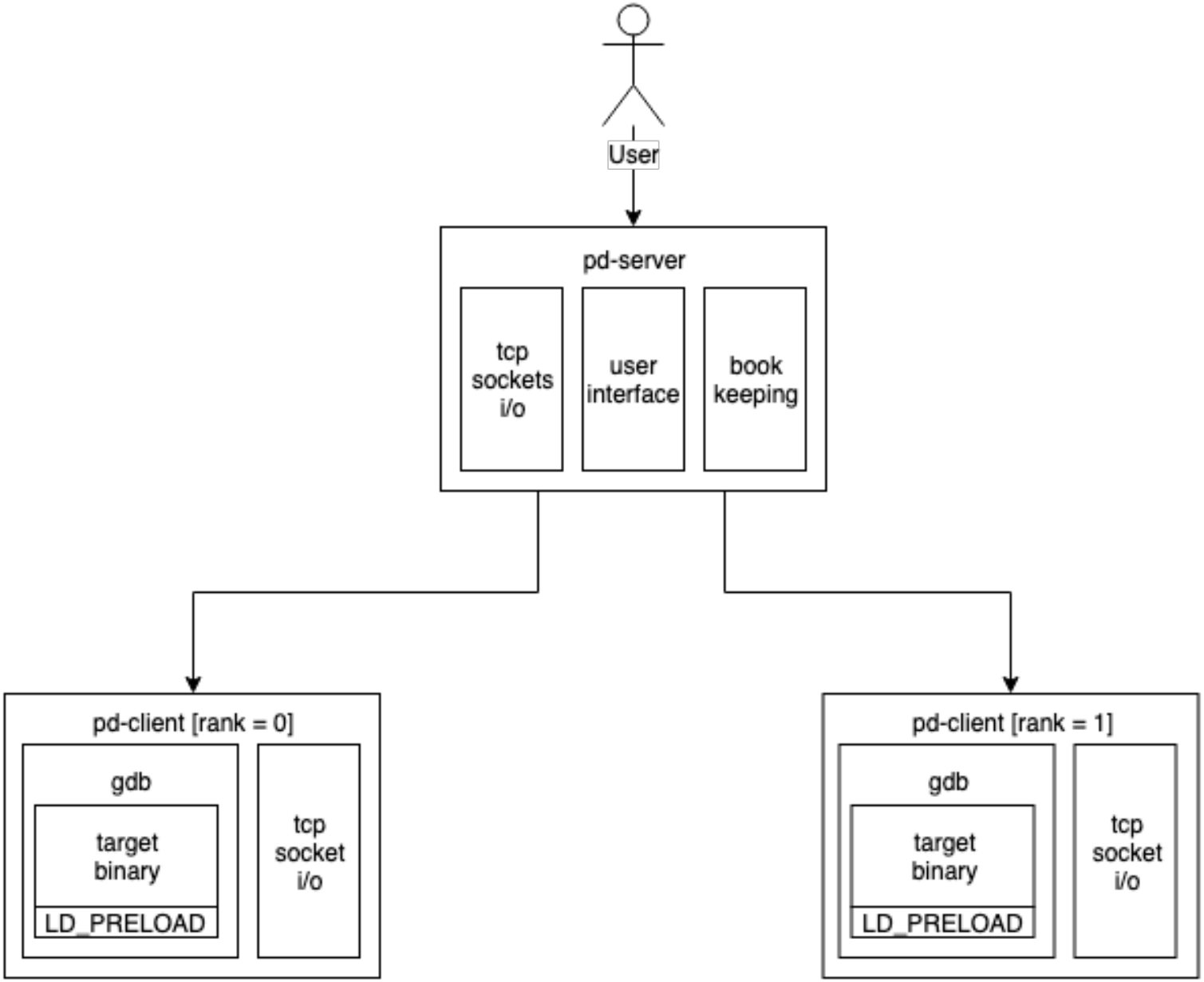

Table of Contents

Starting Up

- When the `pd-client` starts up, it starts an instance of `gdb`.

Starting Up

- When the `pd-client` starts up, it starts an instance of `gdb`.
- The `gdb` instance loads the target binary, adding to `LD_PRELOAD` our custom library.

Starting Up

- When the `pd-client` starts up, it starts an instance of `gdb`.
- The `gdb` instance loads the target binary, adding to `LD_PRELOAD` our custom library.
- We write our own version of `MPI_Init` inside our custom library.

Starting Up

- When the `pd-client` starts up, it starts an instance of `gdb`.
- The `gdb` instance loads the target binary, adding to `LD_PRELOAD` our custom library.
- We write our own version of `MPI_Init` inside our custom library.
- This consists of a call to `PMPI_Init`, followed by writing the rank and the world size of `MPI_COMM_WORLD` to disk.

Starting Up

- The gdb instance sets a breakpoint on this custom function.

Starting Up

- The gdb instance sets a breakpoint on this custom function.
- Then, we run the program till this breakpoint is hit, and move to the end of the function.

Starting Up

- The gdb instance sets a breakpoint on this custom function.
- Then, we run the program till this breakpoint is hit, and move to the end of the function.
- We extract the necessary information from the file, and send it to the server.

Starting Up

- The gdb instance sets a breakpoint on this custom function.
- Then, we run the program till this breakpoint is hit, and move to the end of the function.
- We extract the necessary information from the file, and send it to the server.
- Finally, the server sends acknowledgement when all clients are connected.

Starting Up

- The gdb instance sets a breakpoint on this custom function.
- Then, we run the program till this breakpoint is hit, and move to the end of the function.
- We extract the necessary information from the file, and send it to the server.
- Finally, the server sends acknowledgement when all clients are connected.
- When `MPI_Init` ends, all the clients are connected to the server.

Table of Contents

C Code

- For all the collectives we ever want to track, we add them to the library we are loading at runtime using `LD_PRELOAD`.

C Code

- For all the collectives we ever want to track, we add them to the library we are loading at runtime using `LD_PRELOAD`.
- A "marker" function is inside each of the collectives we ever intend to instrument.

C Code

- For all the collectives we ever want to track, we add them to the library we are loading at runtime using `LD_PRELOAD`.
- A "marker" function is inside each of the collectives we ever intend to instrument.
- Then, the MPI collective function sets up some variables to extract rank and communicator info, and calls the corresponing PMPI function.

C Code

- For all the collectives we ever want to track, we add them to the library we are loading at runtime using `LD_PRELOAD`.
- A "marker" function is inside each of the collectives we ever intend to instrument.
- Then, the MPI collective function sets up some variables to extract rank and communicator info, and calls the corresponing PMPI function.
- We use `volatile` to ensure that these functions/variables are not optimized out.

C Code

- For all the collectives we ever want to track, we add them to the library we are loading at runtime using `LD_PRELOAD`.
- A "marker" function is inside each of the collectives we ever intend to instrument.
- Then, the MPI collective function sets up some variables to extract rank and communicator info, and calls the corresponing PMPI function.
- We use `volatile` to ensure that these functions/variables are not optimized out.
- Easy to add new MPI collectives - just two lines of code.

Go Code

- We can toggle tracking a particular collective with `pdb_trackcoll <collective_name>`.

Go Code

- We can toggle tracking a particular collective with `pdb_trackcoll <collective_name>`.
- Once we begin tracking a collective, the Go code for the client (the one which controls gdb) sets a breakpoint on the "marker" function.

Go Code

- We can toggle tracking a particular collective with `pdb_trackcoll <collective_name>`.
- Once we begin tracking a collective, the Go code for the client (the one which controls gdb) sets a breakpoint on the "marker" function.
- On hitting that breakpoint, we extract rank and communicator, and then resume the running of the collective.

Go Code

- We can toggle tracking a particular collective with `pdb_trackcoll <collective_name>`.
- Once we begin tracking a collective, the Go code for the client (the one which controls gdb) sets a breakpoint on the "marker" function.
- On hitting that breakpoint, we extract rank and communicator, and then resume the running of the collective.
- This is completely transparent to the user.

Go Code

- We can toggle tracking a particular collective with `pdb_trackcoll <collective_name>`.
- Once we begin tracking a collective, the Go code for the client (the one which controls gdb) sets a breakpoint on the "marker" function.
- On hitting that breakpoint, we extract rank and communicator, and then resume the running of the collective.
- This is completely transparent to the user.
- At any point the user can view pending collectives using `pdb_listcoll`.

Table of Contents

Idea

- Each `MPI_Comm` is actually an integer.

Idea

- Each `MPI_Comm` is actually an integer.
- Whatever function can create a new communicator, write a wrapper for it in our custom library.

Idea

- Each `MPI_Comm` is actually an integer.
- Whatever function can create a new communicator, write a wrapper for it in our custom library.
- Break on that wrapper and extract communicator ID and rank of process transparently.

Idea

- Each `MPI_Comm` is actually an integer.
- Whatever function can create a new communicator, write a wrapper for it in our custom library.
- Break on that wrapper and extract communicator ID and rank of process transparently.
- Hence, we know which process belongs to the newly created communicator.

Issues

- `MPI_Comm` integer representation is **not the same across processes**.

Issues

- `MPI_Comm` integer representation is **not the same across processes**.
- Makes it hard to identify communicators uniquely and see what ranks (with respect to `MPI_COMM_WORLD`) are included.

Issues

- `MPI_Comm` integer representation is **not the same across processes**.
- Makes it hard to identify communicators uniquely and see what ranks (with respect to `MPI_COMM_WORLD`) are included.
- Alternative: trying to understand the context using rules set by the MPICH library.

Issues

- `MPI_Comm` integer representation is **not the same across processes**.
- Makes it hard to identify communicators uniquely and see what ranks (with respect to `MPI_COMM_WORLD`) are included.
- Alternative: trying to understand the context using rules set by the MPICH library.
- Alternative: using `MPI_Bcast` to send random integer from zero rank of new communicator and all the processes which send the same integer to the server can be grouped together.

Table of Contents

Table of Contents

References

- GDB/MI2 Documentation
- Go library for talking to GDB/MI - Our modified version of the library.
- Allinea DDT, TotalView
- OpenMPI FAQ for Parallel Debugging
- Common MPI Programming Errors
- MPICH Wiki: Communicators and Context IDs