# CS633 Project: Parallel Debugger

Milind Luthra (150363)  Subhdeep Saha (150732)

15 March 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

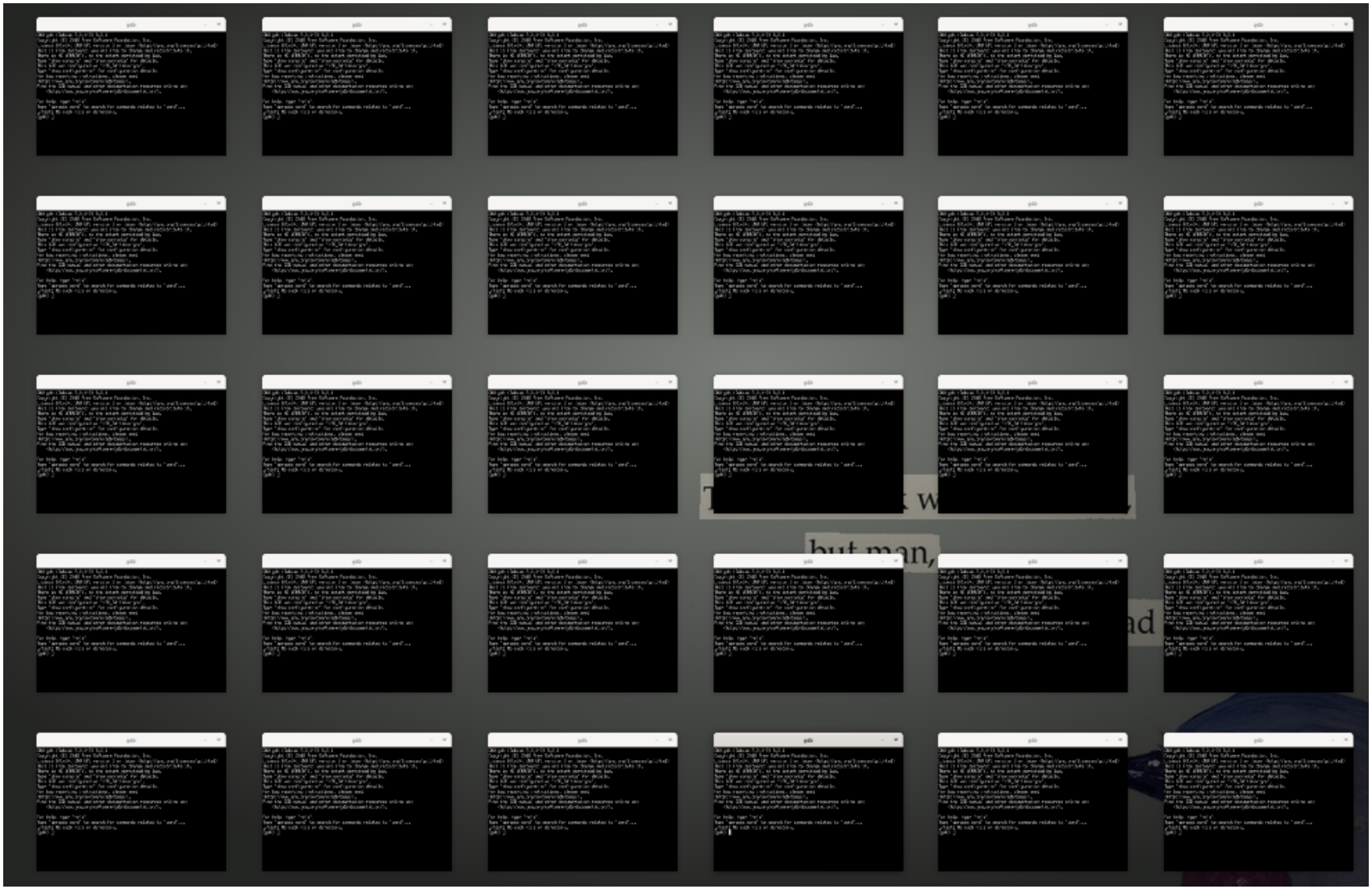
- Possible Idea: For n processes, launching n XTerm instances with gdb.
- Each terminal can be used to debug the individual processes.

# 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`.

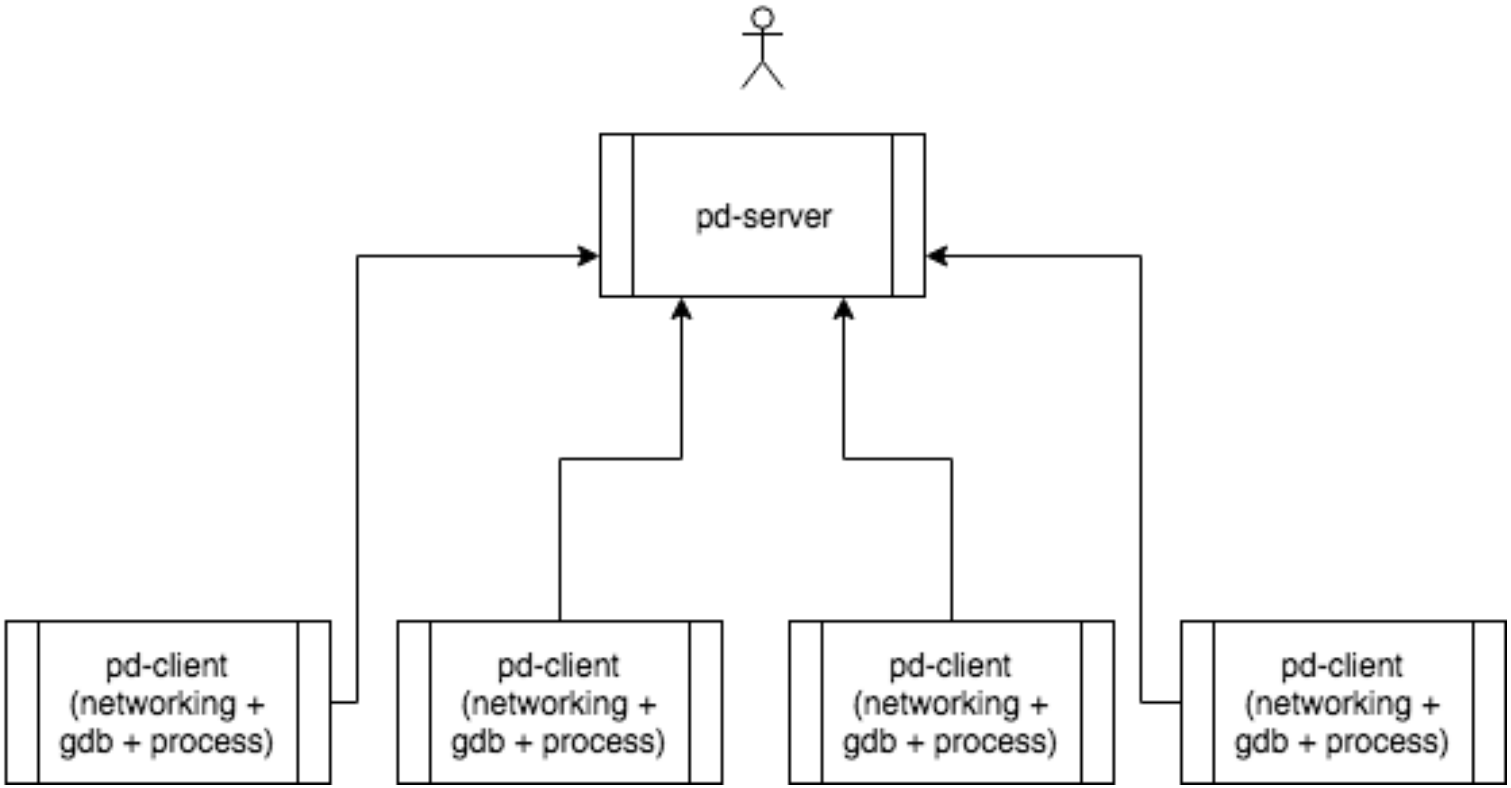
# Basic Idea

- 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.

# Basic Idea


pd-server
pd-client (networking + gdb + process)
pd-client (networking + gdb + process)
pd-client (networking + gdb + process)
pd-client (networking + gdb + process)

# Table of Contents

# Table of Contents

# pd-client

- Handles the networking of an individual client to the server.

# pd-client

- Handles the networking of an individual client to the server.
  - Sends request to the server using **TCP sockets**.

# pd-client

- Handles the networking of an individual client to the server.
  - Sends request to the server using **TCP sockets**.
- Loads the target binary `./test` and instrument it using LD preload.

# pd-client

- Handles the networking of an individual client to the server.
  - Sends request to the server using **TCP sockets**.
- Loads the target binary `./test` and instrument it using LD preload.
- Acts as a frontend to the GDB interpreter `mi2`.

# pd-client

- Handles the networking of an individual client to the server.
  - Sends request to the server using **TCP sockets**.
- Loads the target binary `./test` and instrument it using LD preload.
- Acts as a frontend to the GDB interpreter `mi2`.
  - Conveys the instructions from the server to individual clients.

# Machine Interface (mi)

- Currently GDB supports multiple command interpreters, and some of them are used to make user interface for the GDB

# Machine Interface (mi)

- Currently GDB supports multiple command interpreters, and some of them are used to make user interface for the GDB
- **mi2** is a line based machine oriented text interface to GDB

# Machine Interface (mi)

- Currently GDB supports multiple command interpreters, and some of them are used to make user interface for the GDB
- **mi2** is a line based machine oriented text interface to GDB
- The default interpreter is the **Console Interpreter**. For a different interpreter user can use
- `gdb .test --interpreter=(console|mi|mi2)`

# mi2 vs console

**mi2**

- `-file-exec-and-symbols ./test`

**console**

- `file ./test`

# mi2 vs console

**mi2**

- `-file-exec-and-symbols ./test`
- `-break-insert main`

**console**

- `file ./test`
- `break main`

# mi2 vs console

**mi2**

- `-file-exec-and-symbols ./test`
- `-break-insert main`
- `-exec-run`

**console**

- `file ./test`
- `break main`
- `run`

# mi2 vs console

**mi2**

- `-file-exec-and-symbols ./test`
- `-break-insert main`
- `-exec-run`
- `-exec-continue`

**console**

- `file ./test`
- `break main`
- `run`
- `continue`

# Table of Contents

# pd-server

- Ensures that all the clients are connected to it.

# pd-server

- Ensures that all the clients are connected to it.
- Maintains a in-memory `map[rank int]conn *net.Conn` to store the clients that are already connected.

# Table of Contents

# Table of Contents

# General Debugging Features

- The main feature is to expose all the power of gdb in a user-friendly way (through a TUI).

# General Debugging Features

- The main feature is to expose all the power of gdb in a user-friendly way (through a TUI).
- Example: Set a breakpoint on `MPI_Allgather` in *each* process.

# General Debugging Features

- The main feature is to expose all the power of gdb in a user-friendly way (through a TUI).
- Example: Set a breakpoint on `MPI_Allgather` in *each* process.
- `(pdb) break MPI_Allgather`

# General Debugging Features

- The main feature is to expose all the power of gdb in a user-friendly way (through a TUI).
- Example: Set a breakpoint on `MPI_Allgather` in *each* process.
- `(pdb) break MPI_Allgather`
- Example: Selectively print `argc` in processes with rank 4, 5, 6 (with respect to MPI_COMM_WORLD).

# General Debugging Features

- The main feature is to expose all the power of gdb in a user-friendly way (through a TUI).
- Example: Set a breakpoint on `MPI_Allgather` in *each* process.
- `(pdb) break MPI_Allgather`
- Example: Selectively print `argc` in processes with rank 4, 5, 6 (with respect to MPI_COMM_WORLD).
- `(pdb) pdb_print argc [r=4,5,6]`

# General Debugging Features

- Example: List all communicators and associated processes' rank.

# General Debugging Features

- Example: List all communicators and associated processes' rank.
- `(pdb) pdb_listcomm`

# General Debugging Features

- Example: List all communicators and associated processes' rank.
- `(pdb) pdb_listcomm`
- Example: Selectively set breakpoint in processes in communicator 4.

# General Debugging Features

- Example: List all communicators and associated processes' rank.
- `(pdb) pdb_listcomm`
- Example: Selectively set breakpoint in processes in communicator 4.
- `(pdb) pdb_break MPI_Allgather [c=4]`

# Table of Contents

# Missing Collective Calls

- Missing a collective call in any one process of a communicator is a source of errors.

# Missing Collective Calls

- Missing a collective call in any one process of a communicator is a source of errors.
- We are maintaining a communicator to processes mapping.

# Missing Collective Calls

- Missing a collective call in any one process of a communicator is a source of errors.
- We are maintaining a communicator to processes mapping.
- We can use this to maintain a list of collectives pending on a process.

# Missing Collective Calls

- Missing a collective call in any one process of a communicator is a source of errors.
- We are maintaining a communicator to processes mapping.
- We can use this to maintain a list of collectives pending on a process.
- Example: Listing all pending collectives.

# Missing Collective Calls

- Missing a collective call in any one process of a communicator is a source of errors.
- We are maintaining a communicator to processes mapping.
- We can use this to maintain a list of collectives pending on a process.
- Example: Listing all pending collectives.
- `(pdb) pdb_listcoll`

# Modifying Buffer During Asynchronous Calls

- A buffer passed to `MPI_Isend` cannot be written to until we're sure that the it has been copied.

# Modifying Buffer During Asynchronous Calls

- A buffer passed to `MPI_Isend` cannot be written to until we're sure that the it has been copied.
- Usually done using `MPI_Wait` or `MPI_Test`.

# Modifying Buffer During Asynchronous Calls

- A buffer passed to `MPI_Isend` cannot be written to until we're sure that the it has been copied.
- Usually done using `MPI_Wait` or `MPI_Test`.
- We can track asynchronous calls using breakpoints.

# Modifying Buffer During Asynchronous Calls

- A buffer passed to `MPI_Isend` cannot be written to until we're sure that the it has been copied.
- Usually done using `MPI_Wait` or `MPI_Test`.
- We can track asynchronous calls using breakpoints.
- We can track buffer writes using watchpoints.

# Modifying Buffer During Asynchronous Calls

- A buffer passed to `MPI_Isend` cannot be written to until we're sure that the it has been copied.
- Usually done using `MPI_Wait` or `MPI_Test`.
- We can track asynchronous calls using breakpoints.
- We can track buffer writes using watchpoints.
- Hence, we can give feedback for any invalid write.

# 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 TODO

# Timeline

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

# Timeline

- Allow sending commands from server to only specific clients TODO
- Make a nice TUI. TODO
- Track communicator data and collective calls. IFPOSSIBLE

# Timeline

- Allow sending commands from server to only specific clients TODO
- Make a nice TUI. TODO
- Track communicator data and collective calls. IFPOSSIBLE
- Track asynchronous call buffers. IFPOSSIBLE
- Make a nice GUI. IFANYONEVOLUNTEERS

# 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