GDBase v1.0

GDBase Parallel Offline Debugger User Guide
Document Revision 1.0
September 24, 2012

Carlos Rosales Fernández, `carlos@tacc.utexas.edu`
Doug James, `djames@tacc.utexas.edu`

High Performance Computing Group
Texas Advanced Computing Center
The University of Texas at Austin
Austin, Texas (USA)

Contents

1 Overview

GDBase is an open source offline parallel debugger designed for scalable execution and flexible analysis. The debugger works by collecting user-specified data during the execution of a job and storing it in a database, which is available for offline inspection.

GDBase decouples execution and analysis, and provides the means of data mining the collected data for faults in the code.

There are many advantages to this debugging model, and we describe them briefly in the following.

1.1 Scalability

GDBase eliminates the need for a centralized debugging interface or control. This reduces the network overhead required for the debugging effort, and opens the possibility of debugging large scale applications requiring 10,000s to 100,000s of tasks.

1.2 Flexibility

By decoupling execution and analysis GDBase provides the user with a high level of flexibility in the way they approach the debugging effort. The traditional debugging technique of trying to focus on a very specific section of the code and collecting as little data as possible in order to isolate the problem is available to the user. But a new option exists now, which is to collect a larger amount of data in a single application execution and then data mine the information offline. This is a valuable option for users working in systems where the number of computing hours is limited, such as XSEDE, since a single application run at scale may provide all the information needed to debug the code.

1.3 Productivity

Offline debugging is critical for debugging large scale applications. These codes are typically run through batch systems, where the user submits a job to a queue and then has to wait for hours, often days for large core counts, for the necessary resources to be available and the execution to start. Having to be ready to pay attention to the debugger at an indeterminate point of time over the next few days is likely to discourage people from running a large scale interactive debugging session. Unfortunately, however, many code errors only show in the code when running at scale. Offline debugging provides the user with the ability to collect the data in batch mode and do the debugging analysis at a later time, increasing productivity and reducing frustration.

A more detailed description of the GDBase architecture is provided in Appendix A.

2 Installation

GDBase is distributed as a series of C files as well as tcl and python scripts. Download the latest instance from the git repository https://github.com/TACC/GDBase.git

2.1 Requirements

The following packages are required for GDBase to install and run correctly:

Package	**Version**	**Website**
gdb	6.8	sourceware.org/gdb
sqlite3	3.5.9	www.sqlite.org
tcl	8.4.19	www.tcl.tk
postgresql	8.3.3	www.postgresql.org
python	2.5	www.python.org
PyGreSQL	3.8	www.druid.net/pygresql

Other versions of of these packages may work as well, but have not been tested.

2.2 Installation Process

The installation of GDBase itself is simple. Run the configuration script and specify the install location and the tcl version you will be using:

```
./configure --prefix=</FullPathToInstallDir> --with-tcl=<version>
```

This will create the appropriate Makefile, so that you only need to run:

```
make
make install
```

This will create the directory specified by `prefix`, compile the source code, and move the necessary files to the install directory. You may want to add the bin subdirectory to your `PATH` for ease of use:

(bash users) `export PATH=$PATH:/FullPathToInstallDir/bin`
(csh users) `setenv PATH $PATH:/FullPathToInstallDir/bin`

2.3 PostgreSQL (Postgres) Configuration

To make the most of GDBase it is necessary to have superuser access to a postgres database. Notice that said database does not need to be running in the same system where the data is collected. A typical operation mode is to install postgres in a local workstation and then use the steps below to configure it for use with GDBase.

First of all, you will need the gdbase schema and database configuration files provided with gdbase. These can be found in the `db` subdirectory. The database configuration file is a text file that we will call `db.config`, with a line that reads like:

```
dbname=opd host=localhost port=5432 user=postgres password=mypassword
```

You can change the information in this file to suit your own needs.

After you have located the `schema.sql` and `db.config` files, start postgres with administrative rights:

```
sudo -u postgres psql
```

Create a user named opd that has administrative rights (notice the postgres prompt):

```
postgres=# createuser --superuser opd
```

Change the password for the opd user and set it so that it matches the password entry in the database configuration file `db.config`:

```
postgres=# \password opd
```

Now exit postgres with `Ctrl+D` and import the schema:

```
psql < schema.sql
```

After this you should be able to use the central database to analyze your results.

3 Quick Start

Compile your application with debugging information by using the `-g` compiler flag. Then use the `opd` executable to run your code:

```
mpiexec opd -e myApp arg1 arg2
```

This will run your application and collect all segmentation fault errors into local databases. No other debugging information is recorded by default.

Upon launch, `opd` will execute `myApp` and its arguments if any. You may use `mpirun` or any other MPI job launcher provided you can access the following information through environment variables:

Job ID The job ID provided by the batch system. The default is `PBS_JOBID`, but it can be overridden by using the `--jobid` command line switch with `opd`.

MPI Task Rank The defaults are `MPIRUN_RANK` when using MVAPICH and `OMPI_MCA_ns_nds_vpid` when using OpenMPI 1.2.x or earlier. The `--rank_var` switch will override this.

Number of Processes in the MPI job The default is `MPIRUN_NPROCS` when using MVAPICH and `OMPI_MCA_ns_nds_num_procs` when using OpenMPI 1.2.x or earlier. Use the `--nprocs_var` switch to override this.

The output will be a series of sqlite3 files, which you can then merge into a single central database:

```
dbmerge myApp
```

Now that all the debugging information has been collected you can use the gdbase analysis tool to identify the problem:

```
gdbase -c ./db.config -j JOBID -a segreport
```

where `JOBID` is the identifier used by your scheduling system. The output for a successfully terminated run would look like:

```
PBS JOBID: JOBID
DatabaseID: 266
Statistics:
        start:  2012-04-17 21:19:16
        end:    2012-04-17 21:19:17
        elapsed:    00:00:01
        ncpus:      2
        Messages:   53
segreport
No Segfault
```

While a report for a segfault would appear as:

```
PBS JOBID: JOBID
DatabaseID: 267
Statistics:
        start:  2012-04-18 01:30:27
        end:    2012-04-18 01:30:28
        elapsed:    00:00:01
        ncpus:      2
        Messages:   51
segreport
Job crashed on Rank: 0  Thread: 1
At:
main    in  seg.c:21

With stack:
Stack: Level    Function        File    : Line
    0  main     in seg.c:21
```

Notice that even by using the default collection and analysis agent we can pinpoint a segfault and find the exact file, function and line where it occurs.

In some cases you may want to use the above commands individually or create your own wrapper script that includes them all, but is often convenient to do the merge and the analysis separately so they can be run on a workstation or laptop.

4 Running GDBase

A typical debugging session with GDBase consists of four major steps:

1. Configure the collection agents
2. Execute the offline parallel debugging session (`opd`)
3. Merge data into central postgres database
4. Use agents to filter and analyze the results

4.1 Collection Agents

By default `opd` will only collect information about any segmentation faults that occur during execution, but there are several ways to extend and personalized the data that is collected during the program execution.

4.1.1 Setting Breakpoints and Watchpoints

There are two ways of setting breakpoints in your application. The simplest way is to create a debugging specification file. You may tell GDBase the location of your specification file using the `-d` or `--debug-spec` command line switch.

```
    mpiexec opd -d specfile -e myApp
```

The specification file is a text file that contains keywords specifying the location of the breakpoints and watchpoints. Use `@bp` to set a breakpoint:

```
@bp functionName
```

Notice that the location specification works exactly as in the gdb command line. One can use a file name followed by a colon and a line number, or a function name.

When breakpoints are hit, a stack trace will be automatically logged in the database. Immediately following a breakpoint you may declare any number of variables you wish to log. Each variable should be in its own line. For example, to set a breakpoint at the entrance of a function and then inspect variables `var1` and `var2` one would use:

```
@bp functionName
    var1
    var2
```

The declaration of watchpoints is very similar, with the exception that watchpoint declarations take two arguments: the location of the watchpoint, and the name of the variable to watch:

```
@watch myApp.c:10 var3
```

This would set a watchpoint for variable `var3` at location `myApp.c:10`

Here is what a debugging spec file using these examples would look like:

```
@bp functionName
    var1
    var2
@watch myapp.c:10 var3
@bp myapp.c:231
    var4
```

It is critical that no blank lines appear int eh specification file, as they would be treated by GDBase as a variable with no name associated with the last processed breakpoint, and cause gdb to throw an error during run time.

4.1.2 Customized Collection Agents

If the debugging specification file doesn't give you enough control, you may write your own debugging script to perform any set of operations the TCL language provides including conditionals.

```
mpiexec opd -s scriptfile -e myApp
```

Within the script, the first method executed is `user_setup`. Every debugging script you write must contain this method. This is where you want to set your initial breakpoints and the methods that are called when a breakpoint is hit. The example script below illustrates setting a single a breakpoint and performing a few operations when the breakpoint is reached.

```
# TCL Comments begin with a # symbol

        proc user_setup {} {
                gdb_setBreakpoint "main" "get_info"
                set output [gdb_lastOutput]
                db_logMessage "user.break" $output
        }

        proc get_info {} {
                gdb_getStackFrames
                set output [gdb_lastOutput]
                db_logMessage "myinfo" $output

                gdb_listLocals
                set output [gdb_lastOutput]
                db_logMessage "myinfo" $output

                gdb_evalExpr var_name
                set output [gdb_lastOutput]
                db_logMessage "myinfo" $output

                gdb_continue
        }
```

More examples can be found in the gdbase/share directory, including examples for deadlock detection and allgatherv argument mismatch identification. This scripts can be as simple or as complicated as the user needs, and Tcl together with the mi interface to gdb provide enough flexibility to encapsulate a fully fledged debugging session into a relatively simple agent script.

A list of currently supported methods for use in a debugging script files is provided in Appendix B.

4.2 Central Database

Once the application has competed its run a series of sqlite3 files, one per MPI task, will be generated. These files will be named:

```
gdblog-myApp.TaskID
```

One can do a full merge by simply typing:

```
dbmerge myApp
```

This will merge the information contained in all sqlite3 files into a single postgresql database called opd.

It is possible to do a partial merge by creating a subdirectory and copying there only a few files, as long as they are consecutive and start with TaskID=0. This can be useful to simply look at the data collected in the database, especially when designing new collection agents.

4.3 Analysis Agents

Analysis agents are written in Python, and use the pygresql interface to postgres. A gdbase Python module is provided which contains many useful definitions. The first executable line in the agent script should be:

```
from gdbase import *
```

After this one can include the agent definition and initiate the connection to the database easily:

```
def agent( jobid=None, database=None ):
    db = GDBase()
    db.connect(database)
```

After the connection to the database has been established one can query the database for an specific jobid entry:

```
J = db.getJob(jobid)
```

And get all messages associated to such jobid:

```
M = J.getMessages()
```

The messages can then be parsed with several provided methods, described in Appendix C.

An example of a simple analysis script that prints the stack trace when it finds a segmentation fault is given below.

```
#!/usr/bin/python

from gdbase import *

def agent(jobid=None,database=None):
        # Connect to the postgres database
        db = GDBase()
        db.connect()

        # Select entries from a given jobid
        J = db.getJob(jobid)

        # Select messages from this jobid
        M = J.getMessages()

        # Restrict selection to entries with
        # Key value equal to opd.SEGFAULT
        M.setKey(opd.SEGFAULT)

        # Print the stack if segmentation fault
        # instances are found
        if M.getCount() > 0:
                result = M.getNext()
                stack = parsegdb(result[value])
                printstack(stack)
        else:
                print No segmentation fault detected

if __name__ == __main__:
        agent(None)
```

Acknowledgments

This software build upon the original implementation by Karl Lindekugel IV and Anthony DiGirolamo. We would like to acknowledge their original work and useful discussions with Anthony DiGirolamo. This work was sponsored by the National Science Foundation under grant 1019055. For a more complete history and additional references see the technical report that is part of this distribution.

A Offline debugger architecture

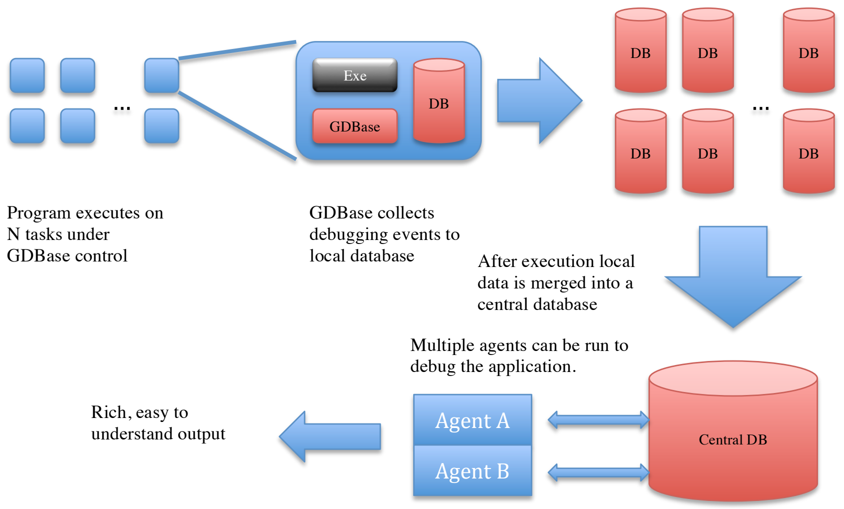
The offline debugger architecture has the following components:

Event collection: Task-local debugging event collection of pre-determined information

Event Management: Central aggregation of event data

Event Analysis: Framework for automatic analysis of collected data

The flow of data in GDBase is described in Figure 1.



...
Exe
GDBase
DB
DB
DB
DB
...
DB
DB
DB
Program executes on N tasks under GDBase control
GDBase collects debugging events to local database
After execution local data is merged into a central database
Multiple agents can be run to debug the application.
Rich, easy to understand output
Agent A
Agent B
Central DB

Figure 1: Data flow in GDBase architecture. The three major steps involved are local data collection; merge to a central database, and analysis

In its current implementation the local data is saved in a sqlite3 file, and later merged to a central postgreSQL database. The data collection and analysis can be done using the provided agents or creating custom ones.

B Collection Data Methods.

The following list describes all currently supported methods for use in debugging script files. Most of these replicate a function of gdb.

`db_logMessage messagekey messagevalue`
Log a message to the database.

`gdb_lastOutput`
Retrieve the last GDB output.

`gdb_setBreakpoint location tclmethod`
Set a breakpoint.

`gdb_setWatchpoint mode variable`
Set a watchpoint.

`gdb_getStackFrames`
Get the local stack frames.

`gdb_evalExpr expression`
Use GDB's expression evalutation.

`gdb_listLocals`
List all local variables.

`gdb_continue`
Resume execution.

`gdb_stepNext`
Step one line of code.

`gdb_stepFinish`
Resumes the execution of the inferior program until the current function is exited.

`opd_getRank`
Get the current rank or MPI task number.

`opd_getSize`
Get the number of tasks in the current job.

C Analysis Agent Methods

`printstack`
Prints the stack trace in a easy to read form.

`parsegdb`
Removes the leading part of a string entry in the database and provides a parsed output with individual elements in a dictionary.

Class GDBase `connect`
Connects to the central database

`getJob(jobid)`
Get database entries for jobid

Class Job `getStartTime`
Get timestamp of earliest database entry

`getEndTime`
Get timestamp of last database entry

`getElapsedTime`
Get time elapsed between first and last database entries

`getJobSize`
Get number of MPI tasks associated with this job

`getMessages`
Create new Messages object will all entries for the current job

Class Messages `getNext`
Get the next entry in the current Message object. This is typically a full database row.

`setKey(myKey)`
Select all messages with a Key entry matching the pattern myKey*. In SQL terms this is equivalent to a SELECT * FROM MESSAGES WHERE KEY LIKE myKey

`setKeyExact(myKey)`
Select all messages with a Key entry matching myKey exactly. In SQL terms this

is equivalent to a SELECT * FROM MESSAGES WHERE KEY = myKey

`setRank(myRank)`
Select all messages with a rank entry matching myRank exactly.

`setValue(myValue)`
Select all messages with a Value entry matching the pattern *myValue*. In SQL terms this is equivalent to a SELECT * FROM MESSAGES WHERE VALUE LIKE