A Scalable Framework for Offline Parallel Debugging

Karl Lindekugel, Anthony DiGirolamo, and Dan Stanzione
{klindeku, anthony.d, dstanzi}@asu.edu

Fulton High Performance Computing,
Arizona State University

October 7, 2008
A Scalable Framework for Offline Parallel Debugging

{klindeku, anthony.d, dstanzi} @asu.edu

High Performance Computing systems continue to grow in size and complexity

- The recent advent of multi- and many-core chips has only accelerated this trend
- Large scale applications require 10,000-100,000s of threads to achieve maximum performance

Debugging technology has remained fairly constant

- Most effort is still focused on interactive debugging schemes
- It is not clear that large scale interactive debugging is compatible with the way large systems are operated

In this talk we will look at how debugging parallel applications in an offline manner solves these issues.
Large Scale Debugging

- The volume of data generated by debugging and monitoring tools requires an efficient infrastructure for collection and organization.
- Visual representation of data may be untenable for applications and systems executing at very large scales.
Correctness and Performance Debugging

- Applications can be faulty without producing an error. Performance can be degraded by:
  - Calculation mistakes
  - Large amount of I/O
  - Poor communication patterns

- Inefficiency wastes expensive computational cycles

- Finding errors of this kind can be very difficult

- **Future debugging systems must combine application performance and correctness data, including data across multiple runs, to find application efficiency errors.**
Reliability

- As systems grow, hardware failures occur more often.
- As applications utilize more cores, hardware failures may be a part of many jobs.
- It will not be immediately clear if errors are software or hardware related.
- **Future debugging systems must be able to monitor system performance across jobs in order to detect hardware related errors.**
Production Environments

- Most sites use batch systems in order to maintain high utilization
- Interactive debugging complicates batch operation
- Sites often limit the scale at which interactive debuggers can run
- **Future debugging systems must be able to operate properly inside of existing batch queuing systems so they may run at the largest scales.**
Challenges

To summarize, future debugging systems must be able to:

- Operate within batch queue systems
- Detect hardware related errors
- Combine performance and correctness debugging information across multiple runs
- Provide different methods of presenting information
- Scale to next generation systems
In response to these challenges we have built GDBase, a framework for offline parallel debugging.

**GDBase Provides**

- scalable offline debugging
- the functionality of GDB
- operation within batch queuing systems

**GDBase Functionality**

- GDBase gathers runtime information from a GDB instance
- Collects this information to a distributed event database
- Provides a mechanism for analysis of this data
Workflow

1. Specify debugger behavior
   - Multiple interfaces are available to users (or agents) for controlling debugger behavior

2. Run your application under debugger control
   - Debugging messages are collected local to each task or to shared storage

3. Collect debugging messages
   - After execution, events from debugging tasks are moved to a central location for analysis

4. Use analysis agents on collected information
   - Agents provide a simple way for users to detect common problems
Design

Tasks

Node 1  Node 2  Node 3  ...  Node n

Data Collection and Management

Analysis and Mining Tools

Challenges

GDBase

Implementation

Interface
Agents
Segmentation Fault
Deadlock
AllgatherV
Agent Development

Results

Future Work

Conclusions
A Scalable Framework for Offline Parallel Debugging

{klindoku, anthony.d, dstanzi}@asu.edu

Challenges
GDBase
Implementation
Interface
Agents
Segmentation Fault
Deadlock
AllgatherV
Agent Development
Results
Future Work
Conclusions

Runtime

MPI Application is launched under GDB control
Events are logged to a local disk or shared storage
Behavior may be controlled via interfaces
Data Collection

Events from multiple jobs collected and stored in a relational database

Analysis tools can compare data between runs
Offline Analysis

- Each analysis agent is designed to search for a specific type of error
- A few example agents are provided
Interface

Debugging Specification

- Multiple interfaces are available: simple and advanced
- Debugging specification files allow setting of breakpoints, watchpoints and variable logging
- Aids users in the transition from interactive to offline

```plaintext
@bp functionName
variable1
variable2

@bp myapp.c:231
variable3

@watch myapp.c:10 variable4
```
Interface

Advanced Scripting

- If you need more control, use a debugging script
- Debugging scripts are written in TCL
- Provides fine grained control over GDB
- Intended for agent development

```tcl
proc user_setup {} {
    gdb_setBreakpoint "main" "myMethod"
    db_logMessage "user.break" [gdb_lastOutput]
}
```

- Messages are stored in the database as a key-value pair using the `db_logMessage` command.
Interface

Advanced Scripting

```bash
proc myMethod {} {
    gdb_getStackFrames
    db_logMessage "stack" [gdb_lastOutput]

    gdb_listLocals
    db_logMessage "locals" [gdb_lastOutput]

    set \$result [gdb_evalExpr "var % 2"]
    if { \$result == "1" } {
        db_logMessage "var" "even"
    }

    gdb_continue
}
```
Agents

Agents mine the collected event data to find and locate faults or problems in code. It’s a simple way for users to detect common problems in their parallel applications. Agents can produce reports text or graphical.

Sample Agents Constructed

▶ Segmentation Fault
▶ Deadlock
▶ Allgatherv
Segmentation Fault Agent

Segmentation Fault Agent detects a Segmentation Fault in the event database and produces a report of:

- Task Affected
- Code Location
- Current Stack
- Local Variables

Job Execution

```
baseexec ./myprogram
```
### Segmentation Fault Agent

#### Output

<table>
<thead>
<tr>
<th>PBS JOBID: 198325.moab.local</th>
</tr>
</thead>
<tbody>
<tr>
<td>DatabaseID: 206</td>
</tr>
<tr>
<td>elapsed: 00:00:51</td>
</tr>
<tr>
<td>ncpus: 64</td>
</tr>
<tr>
<td>Messages: 964</td>
</tr>
</tbody>
</table>

Detector Results:
Job crashed on rank: 16
At: main in fdtd.c:385
With stack:
0 main in fdtd.c:385
With locals:
int * p = (int *) 0x4
int i = 132
int t = 4250209
int n = 68
Deadlock Agent

The Deadlock Agent identifies tasks involved in a communication pattern that cannot continue. The tasks are then organized according to number of dependencies. A report is produced containing:

- Outstanding communications
- Location in code for each task
- Stack for each Task

This Agent was motivated by user problems in an asynchronously communicating code. A race condition existed causing the program to randomly deadlock during execution.
Deadlock Agent Usage

Job Execution

```
baseexec --agent deadlock -t 600 ./myprogram
```

- Deadlock catches each Send and Recv from an application and logs their parameters, start, and end to the database.
- The -t option tells the program to time out if it has not received an event from the program in x seconds.
Deadlock Agent Usage

Analysis

gdbase --agent deadlock --jobid 1234.moab

Example Output

Incomplete Communication
Send : 12 ---> 13
Send : 11 ---> 12
Send : 10 ---> 11
Send : 9 ---> 10
Send : 8 ---> 9

Final Stack for Rank 13
Stack:
Level Function File:Line
11 main in deadtest.c:37
10 PMPI_Barrier in pbarrier.c:52
AllgatherV Agent

An AllgatherV collects variable amounts of data from each task and puts the resulting array on every task.

AllgatherV takes data of varied length from each task and distributes it to all tasks.

AllgatherV can also alter the order of data based on task.

The AllgatherV Agent analyzes each AllgatherV call across tasks to:

- Identify improper item counts
- Identify improper offset values

The Agent then produces a report with the affected Task, Location in Code, etc.
**AllgatherV Agent Usage**

**Job Execution**

```
baseexec --agent allgatherv ./myapp myargs
```

**Analysis**

```
gdbase --agent allgatherv --jobid 1234.moab
```

**Example output**

```
Error at element 4 in Recvcount array on task 7 was 64 but should be 63
Stack:
  Depth  Function          Location
     0  main          in mpi-nbody-allg.c:270
     1  PMP1_Allgatherv in pallgatherv.c:39
```
Agent Development

- Python API for accessing database (anything that can query a SQL database will work)
- Provides an abstraction for reading messages
- Provides helper methods for parsing gdb output

```python
from gdbase import *

## connect to database
db = GDBase()
db.connect()

## obtains job id to read from environment
J = db.getJob()

## Look for messages starting with 'opd.SEGFAULT'
M = J.getMessages()
M.setKey('opd.SEGFAULT')
```
Results

GDBase

- Can launch with mpiexec or mpirun
- Tested with the Torque batch queue system
- Tested with OpenMPI and MVAPICH

Impact of GDBase on Application Run Time with -g

<table>
<thead>
<tr>
<th>Execution Type</th>
<th>Time (s)</th>
<th>Percent Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>No GDBase</td>
<td>39.00</td>
<td>-</td>
</tr>
<tr>
<td>Segfault Only</td>
<td>40.04</td>
<td>2.67%</td>
</tr>
<tr>
<td>Ten breakpoints</td>
<td>40.65</td>
<td>4.23%</td>
</tr>
<tr>
<td>40 breakpoints</td>
<td>45.50</td>
<td>16.66%</td>
</tr>
</tbody>
</table>
User Experience

GDBase was used to find several bugs in a user Finite Difference Time Domain (FDTD) code using the Segmentation Fault Agent and its generated report. Three bugs were found:

- Swapped loop indexes on nested loops
- Incorrect ghost row boundaries on another loop
- Incorrect initialization on another loop

Each of these bugs caused the application to crash at sizes about 1024 tasks.
### Test Environment

#### Clusters

- **Saguaro** — ASU, 240 Dual socket, Quad Core Linux computers with Infiniband interconnect. Used for runs up to 1024 tasks.
- **Ranger** — TACC, 3,963 Quad socket, Quad Core Linux computers with Infiniband interconnect. Used for runs up to 4096 tasks.

#### Debuggers

- **Gnu Debugger (GDB)** — Freely available, launches a debugger instance for each task.
- **Intel Debugger (IDB)** — Commercial Product, uses a tree to manage communication with each task.
Segmentation Fault

Segfault predictably after 89 iterations

Overhead (Runtime Multiplier)

Number of Tasks

GNU Debugger
Intel Parallel Debugger
GDBase

Challenges
GDBase
Implementation
Interface
Agents
Segmentation Fault
Deadlock
AllgatherV
Agent Development

Results
Future Work
Conclusions
Break & Collect

Breakpoint set on each iteration, 10 iterations

Overhead (Runtime Multiplier)

GNU Debugger
Intel Parallel Debugger
GDBase

Number of Tasks

9
8
7
6
5
4
3
2
1
0
8
16
32
64
128
256
512
1024
2048
4096
Future Work

Continued development of the GDBase will focus on:

- Test scalability at 50,000 tasks
- Comprehensive user interface (web or graphical)
- Enhancing the facilities provided through the interfaces
- Developing additional analysis agents, including ones that compare data between runs
- Adding support for other tools besides GDB, such as performance and profiling tools (Tau, DPCL)
Conclusions

- Offline debugging shows a lot of promise
- More scalable than interactive debugging
- The framework may provide a viable alternative for debugging at the petascale level
Thank you!