A Generic Data Logger for MPI Programs and Its Use in I/O Characterization

Robert A. Ballance ∗ Sandia National Laboratories Albuquerque, NM, USA raballa@sandia.gov
Jonathan Cook † New Mexico State University Las Cruces, NM, USA joncook@nmsu.edu

ABSTRACT

Monitoring distributed programs on high performance supercomputers is a challenging task, yet it is essential for the proper administration of the machines, and for users to understand what their program is doing on production runs. To this end, we created a flexible monitoring capability for a major class of scientific applications, programs using MPI, that efficiently gathers information from the distributed program and collects it at a central point. This data can then be used to both understand application-centric issues and system-centric issues; and for improvement, administration, and maintenance of both the complex applications producing important scientific results and the complex systems that execute them.

1. INTRODUCTION

High performance computing (HPC) platforms are very complex systems of heterogeneous components that attempt to provide reliable high-capacity computational resources for advanced scientific computing needs. Computational capacity continues to grow rapidly, recently reaching the petaflop level [10], and the interconnect and I/O systems that are built to keep up with all of this computational capability are equally complex.

Large scientific applications running on these high performance computing platforms are often hard to monitor, due to a variety of factors. One, the platforms themselves are complex systems of heterogeneous nodes, and the platforms differ widely between themselves. Two, the varied build environments require that tools must be cross-platform, and allowing significant environment differences. And three, beyond highly-intrusive debugging and analysis tools that are used during development, application deployers typically do not want anything interfering with their application and potentially affecting its performance (and success).

Yet, both application users and system administrators could benefit from at least some level of monitoring on production runs of the applications. Users may be able to better tell if their application is making progress or if it is stuck in some livelock-type failure mode, and administrators could better understand the various performance characteristics of the system on a per-node level and could use this information to achieve higher RAS characteristics of the system (reliability, availability, and serviceability). Unfortunately the state of the art on these systems is virtually no feedback from application execution to system management.

Current efforts are deploying external monitoring of applications [6] (e.g., monitoring output file changes, exit codes, and the like); these will be a great improvement, but they still have several walls that block full potential. Monitoring files for changes can incur a serious delay in observing a problem due to the filesystem’s buffering of the file; the change may not appear on disk soon, or ever. There is no agreement on what exit codes from applications mean, so that a generic system cannot necessarily decide what a particular application exit code means. Finally, the process deployer task will often hide the reason for an individual node’s reason for failure as it aborts all the processes in the application. Ultimately, some level of internal monitoring is needed to truly achieve a reasonable level of system-and-application cooperation, feedback, and management.

Most of the scientific applications use the MPI message passing framework [5] for achieving their parallel computation. To this end, we built a lightweight generic monitor/logger for MPI programs, called EZM, that takes advantage of the way programs use MPI to provide very low-cost monitoring support that would be useable on production application runs. EZM itself delegates actual monitoring to individual clients, thus allowing unique monitors to be separately deployed and managed. We demonstrate the usefulness of EZM by applying it to the problem of monitoring I/O activity in MPI programs. Administrators can often look at the I/O that a program is doing and have some idea of its internal state; e.g., if a program reads from a checkpoint file at startup, then it is probably restarting due to a previous failure, and too many restarts can indicate problems at the hardware or system level.

Section 2 presents background for EZM. Section 3 presents the design and implementation of EZM. Section 4 presents the example I/O monitor, and Section 6 concludes with ideas.
for future work and insights from the current EZM.

2. BACKGROUND

MPI, or Message Passing Interface [5], is the de facto standard for building scientific applications that run on distributed-memory machines. Virtually all of the applications that make use of large computing platforms use MPI. In MPI, programmers explicitly decide when and where to pass messages among the cooperating processes, each of which has its own local data that it is computing over. MPI also supports process synchronization (barriers) and automatic data reduction operations (such as sum). Since MPI is focused on supporting numerical applications that typically process arrays, it has special functions which perform collective data transfer patterns and simultaneous data reduction and process synchronization.

MPI programs usually have a main “Rank0” process that is the first initial process of the program; it is generally responsible for managing the rest of the processes, assigning work if the problem decomposition is not implied or automatic, and gathering results.

There has been quite a bit of work in supporting the debugging and analysis of MPI programs, such as the Paradyn project [3], the Tau monitoring framework [9], and others; we are not claiming novelty in terms of implementing some monitoring capability that has never been done before. However, these frameworks typically are heavyweight in their instrumentation, meant for the debugging and tuning phase of software development, not for the monitoring of production runs. Often these packages are available on development machines but not even installed on machines intended for production runs, and users are typically loathe to add such complexity to a long production run, where system fragility is already an issue.

Large computing machines with many thousands of nodes have very special I/O requirements, and several parallel file systems have been devised to support their I/O needs. However, these file systems are often a source of instability, and applications need to be careful how they use them. For example, it is often logically easier to have an application write a single checkpoint file from all of its processes, but in practice some platforms have difficulty in handling this; thus for reliability a developer may choose to write out one file per process when running with 32 to 120 thousand processes (over as many nodes) this is a non-trivial management problem!

Monitoring and analysis tools must deal with the same I/O issues; logging cannot, e.g., simply assume that it is allowed to write one log per process.

3. METHODS

We are thus motivated to create a monitoring capability that could actually be used in production runs of scientific applications. To accomplish this, the five basic requirements for our framework are:

- a generic mechanism for logging monitor data at a single node is needed;
- the ability to create custom monitors is needed;
- a collector mechanism for transmitting node data back to a single place is needed;
- the monitoring and transmitting must be low overhead; and
- instrumentation must be non-intrusive to the development process.

Our EZM library implements the generic logging and collecting capabilities needed to efficiently monitor MPI programs. Figure 1 shows a high level view of the EZM architecture. EZM interposes itself between a few key MPI calls that the application makes, and performs its work at the points where these calls happen. Additionally, EZM has its own interface that client instrumentation libraries can use to record their own monitoring data; EZM manages the data and handles the collecting of the data back at a single node.

MPI programs naturally reach points at which the computations at individual processes must be synchronized. Since the wait time for processes that reach this point first is unproductive, this is a natural point at which to initiate a transfer of local monitoring data back to the Rank0 node. The points are indicated by calls to MPI functions, namely the various flavors of synchronization and collective communication operations.

3.1 Interposition on MPI Calls

HPC platforms vary in their O/S-level capabilities; many run a flavor of Unix or Linux, but a few run very lightweight O/S kernels on the compute nodes of the platform, with few “normal” system services. EZM is currently developed and tested on platforms that run Linux on all nodes, but it is designed to be portable to more restricted systems.

EZM must intercept a few key MPI calls. The calls that it currently intercepts are:

MPI_Init  MPI_Init_thread  MPI_Finalize
MPI_Allgather  MPI_Allgatherv
MPI_Allreduce  MPI_Alltoall
MPI_Alltoallv  MPI_Alltoallw
MPI_Barrier  MPI_Gather
MPI_Gatherv  MPI_Reduce
MPI_Scatter  MPI_Scatterv
MPI_Reduce_scatter

The first three are for initialization and finalization, respectively. The rest are those MPI functions that entail some synchronization and/or are operations that would happen at key steps in the computation and not extremely frequently.
EZM can take either of two approaches to interposing these MPI calls. One is to use the basic and often-used “wrapping” capability of preloading a shared library with the same symbols as a dependent library, in this case the MPI libraries. This is entirely transparent to the application, and is done at run time so can be performed on any dynamically linked application binary (which is the case for our current test platforms). This is the default configuration of EZM; it simply declares its own versions of the above MPI functions, and then uses dynamic symbol resolution to find the real MPI functions, and calls them appropriately.

Some systems with lightweight kernels on the compute nodes do not support dynamic linking, however. In these cases, MPI itself offers a “profiling” interface, where an interposer can be linked to application MPI calls and then in turn call “PMPL” functions which are the real MPI functions; with the differing names, static linking can be used without conflict. EZM can be configured as a statically linked library which uses this approach to interposition. In this case, the application needs to link in the EZM library during the build process.

Two final issues in interposition are that of programming language and differential library support. In scientific computing, many applications are still written and supported in Fortran, although C and C++ are common as well. These languages typically have different calling conventions, and MPI implementations do not try to configure various compilation parameters to force, e.g., Fortran to call the C MPI routines. Rather, two sets of interface functions are maintained.

EZM must intercept both sets, and one application may have both Fortran and C code, meaning that EZM must correctly handle both in one application. Furthermore, various implementations of MPI (e.g., OpenMPI and MPICH-2 are our main test platforms) define the interface differently, with different internal symbol names. EZM employs a wrapper generation script that automatically generates slightly different interposition wrappers for the different platforms and configurations. In this way, EZM can easily be rebuilt for different settings.

### 3.2 Client Library Support

As described earlier, EZM does not itself monitor the application; it delegates this responsibility to client tools, each built as a library. The client is a specific monitoring tool that watches some part of the application and reports data that it wants to export to EZM, which in turn handles the exporting of the data to the Rank0 process. To do this efficiently, EZM has the following interface:

- `ezmRegisterLoggerCallback (notifyCallback, nameCallback, class);` this function allows the client library to register two callback functions, and declare its class ID;
- `ezmAddEventToLog (eventID, time, data);` this function allows a client library to report a data event to EZM; EZM locally records the event for later sending.

Every client library has an integer class associated with it; this is currently a number with at least the lowest 12 bits being zero. The lowest 12 bits are used to specify the unique data event type within the client’s class. EZM records the class of each client library and uses this to interact with it.

Each client implements two functions that EZM will call back to. The first is a notify function that EZM calls to notify the client that the local data event log is about to be sent back to the Rank0 node. This gives the client an opportunity to announce any new data it has to EZM. For example, a client could intercept memory allocation calls and accumulate how much memory the application is using, but only announce it to EZM when EZM is about to export the data, and then only if the amount has changed since the last data export, or even whether a specified time interval has elapsed or not; this enables clients to be as efficient as possible with aggregated monitoring data. If a client monitor is not aggregating data, then it would simply add each event to the EZM log as it was detected. EZM clients use `ezmAddEventToLog()` to announce data events to EZM. Every data event has a type ID, a timestamp, and some amount of opaque data.

The second callback that a client monitor implements is an event type ID to name string map. EZM uses this at the Rank0 node to attach meaningful strings to each event in the output log. This callback is only used at the Rank0 node; this allows EZM to only send integer data type IDs between processes, but still create human-readable output at Rank0, if desired (some monitors may produce large amounts of data and prefer to leave event types encoded).

### 3.3 Aggregating Data in One Place

At each intercepted MPI synchronization call, EZM reports all local data events back to the Rank0 node, and then removes the local data so that the area can be used to log more events. The initial implementation of EZM does this in a flat communication model, but there is nothing in the architecture of EZM that requires this, so scalability improvements will be possible. Still, even with a flat communication model, the expectation is that since nodes do not arrive at the synchronization point all at once, much of the log communication can be accomplished within time that the nodes would have to be waiting anyways. Another assumption is that log data is small relative to application data being sent, so in practice even the current flat communication of EZM may be usable to reasonable scales, though we of course need to investigate this.

At all non-Rank0 nodes, EZM’s job is simply to send the entire data log to the Rank0 node; once this is done it can go ahead and invoke the actual MPI call the application requested. MPI efficiently supports large message sizes, so log sending is done as a single MPI message. A special first event is maintained in the data log which is timestamped with the time that this data log is sent. At the Rank0 node, however, all other node messages must be received and logged (currently just written to a file).

Our initial desired method of implementation was to receive these log messages in a separate thread; this would provide the most robust model of communicating “out of band” data within the application execution space. However, thread support in MPI implementations is still experimental, and most production platforms do not provide it. MPI’s communication model is that of always reliable inter-process communication. Particularly, for a synchronization call or a collective communication operation, this operation will only complete successfully if all processes successfully complete it. If one process fails, the whole MPI operation (and program) will fail.
Table 1: EZM Performance.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Events logged</th>
<th>Log size</th>
<th>App Time (sec)</th>
<th>Unix Time (sec)</th>
<th>Overhead % (Unix)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 proc, no monitor</td>
<td>0</td>
<td>0</td>
<td>2167.6</td>
<td>2171.7</td>
<td>0.00%</td>
</tr>
<tr>
<td>16 proc, I/O monitor</td>
<td>15,000</td>
<td>974KB</td>
<td>2170.2</td>
<td>2177.1</td>
<td>0.25%</td>
</tr>
<tr>
<td>16 proc, Mem monitor</td>
<td>242,000</td>
<td>18MB</td>
<td>2168.4</td>
<td>2173.3</td>
<td>0.07%</td>
</tr>
<tr>
<td>128 proc, no monitor</td>
<td>0</td>
<td>0</td>
<td>887.2</td>
<td>891.4</td>
<td>0.00%</td>
</tr>
<tr>
<td>128 proc, I/O monitor</td>
<td>9,566</td>
<td>618KB</td>
<td>887.1</td>
<td>905.6</td>
<td>1.60%</td>
</tr>
<tr>
<td>128 proc, Mem monitor</td>
<td>793,088</td>
<td>58.5MB</td>
<td>890.2</td>
<td>899.8</td>
<td>0.94%</td>
</tr>
<tr>
<td>1024 proc, I/O monitor</td>
<td>100,308</td>
<td>6.72MB</td>
<td>1609.0</td>
<td>1811.2</td>
<td>10.8%</td>
</tr>
</tbody>
</table>

Given this, we can make some assumptions in EZM that help perform in-thread reception of the data logs at the Rank0 node. Namely, we can assume that all processes will indeed successfully transmit their local data log. This means that EZM, at the Rank0 node, can simply count and wait for \( N - 1 \) messages, indicating a data log from every other node. Only after receiving all \( N - 1 \) messages does EZM at Rank0 go ahead and invoke the actual MPI call that the application requested.

This assumption is no stronger than that made by the application and MPI itself, and has proven robust in our tests on real scientific applications. The only downside to this is that all nodes must send a message even if they have no new data logged by any client; they at least must send an empty log message. In the MPI synchronization call all of the processes will be sending messages anyways, and so we feel that this requirement is not prohibitive, and it does not appear to incur a large penalty in reality.

The last note to be made regarding data transmission is that all nodes also send their data log one final time when the MPI_Finalize call is intercepted. Any last data recorded by client monitors since the last synchronization call is sent at this point.

### 3.4 EZM Summary

EZM thus provides a clean, lightweight platform on which to create application monitors that do not have to worry about aggregating the data from a distributed application. The individual monitors can be crafted to whatever level of intrusiveness is needed; obviously a highly intrusive monitor will be more effort to create and less desirable for production runs, but EZM does not constrain them.

EZM must still be shown to not add undue performance penalties to the application. Our expectation was that since an application is “pausing” at a synchronization anyways, the small overhead of having EZM transmit its data logs should be somewhat masked. Our initial results seem to support this.

Table 1 shows results from running EZM with two different client libraries on a real scientific application actively being used (but not named for sanitization purposes). The configurations are small test runs, one with 16 processes and roughly 36 wall-clock minutes of execution and the other with 128 processes and roughly 15 wall-clock minutes of execution. The I/O monitor is that described in Section 4 below, and the memory monitor is one that does not intercept any application functions but instead checks `/proc/[pid]/stat` under Linux to check the current memory usage at each node and report it through EZM.

The timings are individual runs and not averaged over some collection of runs, thus while the the numbers display very low overhead from EZM and the monitors, particular conclusions about the specific numbers should be made with care. Indeed we saw one run where the 128-process memory logger was 37% slower than the no-logging run, but we were unable to reproduce the result; this was during a period of heavy cluster use with many queued-up jobs, in a week when there were some problems with the Lustre filesystem. The reported numbers are from runs that were completed within a single hour on a less-busy machine; we do not yet have enough data for valid statistical aggregation.

The I/O monitor overhead is probably larger than the memory monitor because it intercepted application I/O calls, which can happen much more often than the EZM notify callbacks, the only thing that the memory monitor activates on. The memory monitor only generates a few data events for each time that EZM notifies it of a log send, so although it generates a much larger data log, the amount of data per EZM message is not very much more than the empty EZM message that the I/O monitor must often send (when no I/O happened); thus its greater log output does not seem to be an important factor when compared with the I/O monitor.

There is a slight uptick in the overheads for the larger run. We certainly must perform a larger scalability analysis, and we already have ideas for improving the scalability of EZM.

### 4. EXAMPLE: IO CHARACTERIZATION

As an example client library of EZM, we built an I/O monitor that watches the I/O that a program does and classifies that I/O based on user-defined class definitions. I/O performance is a major concern in data-intensive scientific applications, and certain types of I/O can indicate system problems that need managed. However, it is not usually useful for an tool to simply capture all I/O in a single aggregate cost, because applications have different types of parameterizable I/O that is done; writing out intermediate data results (often in the form of generated image or plot visualizations), progress logging, and checkpointing are three main categories.

Our I/O logging library intercepts the basic I/O functions (open, close, read, write, etc.), and measures the amount of time spent in each routine. Upon each open call, the library uses the name of the file being opened, and the mode that it is being opened in, and classifies the type of I/O being performed. The IOMLogger library also works on the preloaded-wrapper principle, in this case wrapping all low-level I/O calls. Each time the EZM library notifies it that data is about to be sent to Rank0, the I/O logger reports total accumulated time for each I/O class that has changed since the
last reporting time. Since during computation phases the program is doing little or no I/O, this optimizes the amount of data sent back to the Rank0 process. Accumulated times are from the program beginning and are not reset after each report.

The results shown in this section are from running the I/O monitor (and EZM) on a small test dataset of a large scientific application actively used within several national laboratories. The configuration used was 16 processes with a run time of about 23 minutes (wall clock time).

The application-specific I/O categorization is done using a “hints” files that the user creates (derived from a Lustre hints feature for file striping optimization [1]). The example hints file for this application used is:

```
out *rsapp.* checkpointing
out *brapp* backup_checkpoint
in *rsapp.* restarting
out *happ history_trace
out *oapp output_log
out *.jpg images
out *.jpeg images
out *ezmLog* ezm_logging
in *spl* spl_input
out *plapp.* plotting
out *spapp.* spl_plots
```

Each line is of the form `mode file-pattern category-name;` mode is input (in), output (out), or both (inout). The filename pattern is one that matches typical glob-style wildcards. The category name is a non-spaced string that identifies the category associated with this specification; a repeated category name on a later specification line will aggregate to the same category (e.g., the “images” category in the example above).

Figure 2 shows the I/O time for checkpointing in this application, per node, for both a “normal” checkpoint operation and an alternative “backup” checkpoint operation (sometimes checkpoints do fail, or get corrupted). For this small run, checkpointing is the main I/O cost (compare the y axis scale on this figure to the others), but is very small, approaching 1 second on the highest nodes out of a total run of about 1400 seconds (on large production runs, checkpointing can approach 30% of total wall clock time). This figure also shows that per-node checkpointing cost is very non-uniform, with nodes 1 and 9 the highest for both forms of checkpointing, and nodes 0, 6, and 8 next highest in cost. These results mirror the sizes of the checkpoint files written by each process, with 1 and 9 writing about 90MB files for this small test run; 0, 6, and 8 writing about 50-60MB files; and other nodes writing smaller files. This can point to a non-uniform allocation of work among the nodes, and a potential source of application improvement.

Figure 3 shows the I/O cost for generating two forms of plotting files, a native form (PL) and one for an external plotting package (SP). The PL-labeled lines show two plots being written for each node, while the SP plots only generate a single datapoint, all clustered at the beginning. The I/O costs for the plot files mirrors the non-uniformness seen in the checkpointing costs; nodes 1 and 9 are highest, with nodes 0, 6, and 8 also higher than the rest of the nodes.

Figure 4 shows all other I/O costs captured from this application run. The first thing to note is that all of these categories are only produced by the Rank0 node; no other node generates any data in these categories. Thus the key does not refer to node number. The second observation is that the category “Unknown” is much higher than all the
that EZM intercepts. Thus EZM can only ship data back to the application calling one of the MPI synchronization functions. EZM therefore sends data back to the application sending requests to it. Its own independent data logging timer, but depends on the intervals of data points. This is because EZM does not have regular growth of the EZM logging cost, but with irregular monitoring framework. The most obvious is the periodically monitoring data, but MRNet assumes a full-featured O/S, and need for hierarchical tool communication is not enough to ensure scalability. We note that their focus is on debugging, and not on monitoring production runs, where monitors will necessarily be much less intrusive.

One possible criticism of EZM is that all nodes send data directly back to the Rank0 node. As programs scale up, this flat model obviously breaks down. Systems such as MRNet [7] have been built that allow tools to create a tree-based overlay network for efficient communication of monitoring data, but MRNet assumes a full-featured O/S, and uses daemon processes and TCP/IP to achieve its capability; these are not always available on lightweight compute nodes, and require more complicated node setup and initialization. Nevertheless, we have been in communication with MRNet developers about using it under EZM, where available.

5. DISCUSSION AND RELATED WORK

As mentioned briefly in the background, there are plenty of efforts that have created and deployed useful monitoring tools for distributed applications, several which have long, well-established histories. Paradyn and its related subprojects continue to make an impact in this area [3] as does Tau [9]. The more recent effort of OpenSpeedShop (OSS) [8] is leveraging pieces of both Tau and Paradyn to try to create a cross-platform consistent monitoring toolset.

The main distinction between these large-scale efforts and the goals of EZM are when and how they expect to be deployed. Large toolsets have typically been designed to help in the development, debugging, and profiling/optimization phases of software construction. We are interested having tools that are acceptable for deployment on production runs, to increase the information available for the administration and maintenance of production HPC systems.

Even recent work by members of the Paradyn community [4] recognize the need to bypass heavyweight toolsets and deploy a lightweight technique if monitoring is to be done in production environments. In their work, which found system problems in a distributed cluster environment, their per-process monitors did not attempt to coalesce function-level traces, but simply wrote out one trace per process.

Other work in the Paradyn community has investigated the issues when trying to scale debugging up to hundreds of thousands of CPU cores [2], where even just addressing the need for hierarchical tool communication is not enough to ensure scalability. We note that their focus is on debugging, and not on monitoring production runs, where monitors will necessarily be much less intrusive.

5. DISCUSSION AND RELATED WORK

As mentioned briefly in the background, there are plenty of efforts that have created and deployed useful monitoring tools for distributed applications, several which have long, well-established histories. Paradyn and its related subprojects continue to make an impact in this area [3] as does Tau [9]. The more recent effort of OpenSpeedShop (OSS) [8] is leveraging pieces of both Tau and Paradyn to try to create a cross-platform consistent monitoring toolset.

The main distinction between these large-scale efforts and the goals of EZM are when and how they expect to be deployed. Large toolsets have typically been designed to help in the development, debugging, and profiling/optimization phases of software construction. We are interested having tools that are acceptable for deployment on production runs, to increase the information available for the administration and maintenance of production HPC systems.

Even recent work by members of the Paradyn community [4] recognize the need to bypass heavyweight toolsets and deploy a lightweight technique if monitoring is to be done in production environments. In their work, which found system problems in a distributed cluster environment, their per-process monitors did not attempt to coalesce function-level traces, but simply wrote out one trace per process.

Other work in the Paradyn community has investigated the issues when trying to scale debugging up to hundreds of thousands of CPU cores [2], where even just addressing the need for hierarchical tool communication is not enough to ensure scalability. We note that their focus is on debugging, and not on monitoring production runs, where monitors will necessarily be much less intrusive.

One possible criticism of EZM is that all nodes send data directly back to the Rank0 node. As programs scale up, this flat model obviously breaks down. Systems such as MRNet [7] have been built that allow tools to create a tree-based overlay network for efficient communication of monitoring data, but MRNet assumes a full-featured O/S, and uses daemon processes and TCP/IP to achieve its capability; these are not always available on lightweight compute-node kernels, and require more complicated node setup and initialization. Nevertheless, we have been in communication with MRNet developers about using it under EZM, where available.

6. CONCLUSION

EZM is a small lightweight monitoring/logging library for distributed MPI programs that is designed with the goal that it be useable in production environments and not just in the debug and performance evaluation and optimization
phases of software development. We have shown that it is usable on real scientific applications with very little overhead.

Some issues still remain to be addressed in the development of EZM. One is the issue of communicators that an application might create and configure to use a sub-group of processes. Rather than just using calls on the default “world” communicator, EZM could potentially piggyback on application communicators and use the natural hierarchy of these groups for more efficient data transfer. This would require the nodes that perform as Rank0 for the sub-groups to cache log data and then transfer it to the whole-application Rank0 node, but it could offer an efficient hierarchical model of monitoring data communication that would not require a separate logical network of processes, as MRNet does. Additionally, MPI libraries typically have relatively efficient implementations of the data-collecting “reduce” operation, and EZM may be able to take advantage of this to better scale to larger numbers of processes.

Currently EZM just writes data out to a textual log file, but options should be created for logging more efficient binary data, for logging into a database, or for better client library customization of textual data logging forms. Client library could be assigned a dynamically generated class ID rather than requiring them to be configured with a static but still unique ID.

Finally, the EZM client library API is also meant to support application-specific monitoring, where if a developer wants, they can insert EZM event announcements to record application-specific data. This can be used to record the end of major compute phases, iterations, and other points in the application; such information, when coupled with generic monitoring information, could provide a valuable collection of application and system data that would be useful for the ongoing administration, maintenance, and improvement of both.

7. REFERENCES