POSITION PAPER – pFLogger: The parallel Fortran logging framework for HPC applications

Thomas L. Clune
NASA Goddard Space Flight Center
Greenbelt, Maryland
tom.clune@nasa.gov

Carlos A. Cruz
Science Systems Applications Inc.
Lanham, Maryland
carlos.a.cruz@nasa.gov

ABSTRACT

In the context of high performance computing (HPC), software investments in support of text-based diagnostics, which monitor a running application, are typically limited compared to those for other types of IO. Examples of such diagnostics include reiteration of configuration parameters, progress indicators, simple metrics (e.g., mass conservation, convergence of solvers, etc.), and timers. To some degree, this difference in priority is justifiable as other forms of output are the primary products of a scientific model and, due to their large data volume, much more likely to be a significant performance concern. In contrast, text-based diagnostic content is generally not shared beyond the individual or group running an application and is most often used to troubleshoot when something goes wrong.

We suggest that a more systematic approach enabled by a logging facility (or 'logger') similar to those routinely used by many communities would provide significant value to complex scientific applications. In the context of high-performance computing, an appropriate logger would provide specialized support for distributed and shared-memory parallelism and have low performance overhead. In this paper, we present our prototype implementation of pFlogger – a parallel Fortran-based logging framework, and assess its suitability for use in a complex scientific application.

CCS CONCEPTS

• **Software and its engineering** → *Software creation and management*; *Software development techniques*; *Object oriented development*;

KEYWORDS

MPI, Fortran, application diagnostics

ACM Reference format:
Thomas L. Clune and Carlos A. Cruz. 2017. POSITION PAPER – pFLogger: The parallel Fortran logging framework for HPC applications. In *Proceedings of 2017 International Workshop on Software Engineering for High Performance Computing in Computational and Data-Enabled Science and Engineering, Denver, CO USA, November 2017 (SE-CoDeSE),* 4 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

SE-CoDeSE, November 2017, Denver, CO USA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The HPC community has invested considerable effort into establishing high-level libraries and frameworks ([3–5, 9]) that orchestrate fast, parallel access to the file system using standardized file formats. The existence of these packages obviates the need for individual applications to develop their own high-performance I/O layer, and also enables the development of shared analysis and visualization tools. But there is one category of I/O that is typically not handled in the manner described above. Namely, the text-based diagnostics that monitor the execution progress that are an important element of nearly all HPC applications. Examples of such diagnostics include

- reiteration of input configuration parameters
- progress indicators such as time step or phase of computation
- high-level scientific diagnostics such as min/max of important quantities, conservation quantities, convergence measures, etc.
- timers and other measures of system performance

These diagnostics are generally not used for data analysis, nor are they typically shared with the community. Rather their value lies mostly in the person or group that is managing the execution of the application, especially in the case of troubleshooting problems.

Our motivation for addressing text-based diagnostics stems from common issues that we have witnessed in multiple large scientific models. First among these is the tendency of diagnostics to evolve into a chaotic collection of disparate messages that range in severity from no-longer needed debugging aids, to vital warnings and errors. At best such clutter increases the difficulty of obtaining information from the diagnostics, and at worst important errors can go unnoticed. The second issue is that of inadequate or ad-hoc support for parallel diagnostics. Here we may want a report from a single process, or to have all processes produce output but with some sort of label indicating the process that produced it, or we may want each process to write to a separate file. A systematic approach should provide a clear and simple mechanism for controlling which of these approaches is to be used for a given diagnostic as well as a flexible means to specify the process labeling and naming of files. The final problem relates to the ability to activate/deactivate diagnostics at run-time. This is particularly valuable for debugging. Custom debugging diagnostics can be extremely useful, but can also generate absurdly large amounts of data and become performance bottlenecks. Thus, a systematic mechanism to throttle certain categories of diagnostics would be quite desirable.

In most applications, diagnostic messages are generated via direct print/write statements). The great benefit of this approach is the versatility. These statements accept an arbitrary list of data

items of any intrinsic type – a feature that is quite difficult to emulate with user-defined procedures, at least in Fortran. Further, print statements include a format specifier that can either leave the details up to the compiler (default formatting) or allow the user fine-grained control of output precision and alignment. The downside of direct print statements is the inability to modify behavior at run time. Deactivation/activation and format changes all require recompilation or additional logic in the surrounding code. E.g., if an application intends to only emit a given message on the root process, then the print statement must be contained within a conditional. For simple messages (e.g., just a string), applications often implement a write_parallel procedure that encapsulates this common case, but supports only a restricted set of method signatures.

We also recognize that some libraries and frameworks provide their own internal logging facilities.[2] While such capabilities are potentially quite useful for their intended purpose, they are generally not suitable for application-specific diagnostics.

2 PFLOGGER – A PARALLEL FORTRAN LOGGING FRAMEWORK

While considering possible paths to creating a flexible logging capability in our development environment, we became aware of the Python logging[7] package.[1] It was immediately apparent that aside from the lack of support for parallelism, the Python logger included all of the features we had planned as well as many other attractive capabilities we had not anticipated. With the advent of object-oriented features in Fortran 2003[6], adaptation of such packages is generally relatively straightforward, and we were confident that only a minor amount of additional development would be required to provide the planned support for parallelism.

2.1 Description of the Python logging utility

The Python Logger class provides a handful of methods which can be used to instrument source code to generate log messages. These interfaces have identical signatures, differing only in their name which indicates the desired *severity* of the requested message: debug(), info (), warning(), error (), and critical (). The arguments to these methods are (1) a format string (i.e., text with edit descriptors), (2) corresponding data arguments, and (3) an optional key-value dictionary. Each Logger object is associated with a severity threshold. If the Logger threshold is higher than the severity of a given request, then no action is taken.[2] Suppressed messages have extremely low overhead. Alternatively, if the request severity equals or exceeds the Logger threshold, then the framework generates a LogRecord object and performs further processing. Each LogRecord object contains the following elements:

- severity level
- name of logger object
- a format string
- list of data arguments
- key-value dictionary

Here is a simple example which demonstrates instrumentation:

```
foo_logger = logging.getLogger('foo')
foo_logger.debug("i=%d; j=%d", i, j)
```

Each Logger object is associated with one or more Handler objects, which receive the generated LogRecord objects. The various Handler classes represent generalizations of output devices, with the FileHandler subclass corresponding to our usual notions about routing output to a file. Other subclasses can be defined to update web pages, send an email notification, send text to a phone number, etc. Each handler object has its own severity threshold, distinct from that of the Logger. A common pattern is to have a FileHandler object that ignores debug and info messages; thereby enhancing the visibility of warnings and errors.

Each handler object is in turn associated with a Formatter object. Formatters provide a mechanism for embedding additional information in messages as they are processed. A familiar example of this is that of desktop system logs in which a time stamp is prepended to each entry. Formatters can also embed many other types of information such as the severity level of the message, the name of the requesting Logger object, and so on.

The Logger and Handler classes are both themselves subclasses of the abstract Filterer class which provides a further mechanism for embellishing or suppressing messages. Common uses include the ability of a handler to be configured to only emit messages from a specific logger or setting a *maximum* severity threshold to route debugging messages.

The Python logger is usually configured with a YAML input file which provides a simple and intuitive mechanism for specifying names and configuration parameters for various filters, formatters, handlers, and loggers. With a handful of edits an end-user can efficiently toggle various behaviors of the logging system without any recompilation of the application. Because our Fortran analog uses a nearly identical mechanism, we will defer providing an example until the next section.

2.2 Parallel features of pFLogger

As mentioned above, our Fortran logger is mostly a straightforward translation of the Python design, with the notable exception of extensions that support parallelism. An alternative approach could have been to provide Fortran wrappers to directly access the python implementation. We believe that there are several merits to that approach, but there would also be significant multi-language inconveniences when introducing new subclasses. We prefer the pure Fortran approach when possible. In this section we describe these extensions as well as a few Fortran-specific details and implementation difficulties.

The first major extension relates to the ability to have multiple processes emit messages to the same handler. For this we have introduced an abstract Lock class with two concrete subclasses: MpiLock and FileSystemLock. The MpiLock class uses MPI one-sided semantics to implement a passive lock which can only be acquired by one process at a time. The alternative FileSystem lock uses atomic file system operations to prevent more than one process from opening a file at the same time. This could be used for instance to allow independent applications to share a log file safely. At this time, our FileSystemLock implementation is not particularly robust across different types of file systems. Nonetheless we provide it for

[1] We only subsequently became aware that the Python logging package was itself based upon Java's logging class.

[2] Logger objects can have associated "ancestor" loggers which also processes each request.

users that may find it useful in their environments. We also note that when using a lock, handlers must open/close files for each message which does introduce additional overhead.

When multiple processes emit messages to the same file, it is useful to label each message by the rank of its process. To facilitate this, pFLogger includes an MpiFormatter subclass of Formatter which includes keywords: mpi_rank and mpi_size that capture these values for the MPI communicator used. For instance, if the MpiFormatter is specified with the format string:

```
'PE=%(mpi_rank)i4.4~: $(message)a'
```

output messages would appear as

```
PE=0172: ...
PE=0001: ...
PE=1854: ...
```

Sometimes instead of aggregating messages from multiple processes into a single file, developers wish to have a separate log file for each process. For this scenario, pFLogger provides an MpiHandler factory method which creates a FileHandler object that opens a different file on each process. The method accepts a file name template and replaces any occurrences of mpi_rank with the corresponding MPI value.

Our final MPI-specific extension is to support the situation in which a message should only be emitted on a single process (usually the root, or rank 0). The MpiFilter subclass is configured with a process rank and only emits messages on the process whose rank matches. This class could easily be extended to work with a set of ranks by specifying a rank stride.

2.3 Other pFLogger features

We have significantly altered the syntax for format strings in pFLogger as compared with those of the Python logger. We had two primary motivations for this decision. First, our targeted audience, Fortran programmers, are far more likely to be familiar with Fortran edit descriptors than their C analogs. Further, by using a syntax similar to native Fortran edit descriptors, pFLogger can leverage the compiler to perform the lowest level of string formatting.[3]

pFLogger provides a StreamHandler subclass that are specified with Fortran I/O units instead of file names. This allows pflogger output to be intermixed with other conventional print statements in an application, and also provides a convenient mechanism to route messages to OUTPUT_UNIT and ERROR_UNIT (Fortran 2003 predefined values for standard output and error units).

One final noteworthy feature is support for embedding *simulation* time in a Formatter. While conventional time stamps are useful for monitoring progress of a run, simulation time is more appropriate for correlating messages with the state of a simulation. Use this feature is a bit involved and requires the end user to implement a function which returns the simulation time in the form of a small dictionary of user-defined units.

Our Fortran analog of the Python logger would have been far more difficult to implement without two important advances. First, the Fortran 2003 standard[6] introduced object-oriented features which enable a direct translation of the Python class hierarchy. And second, we had previously developed FTLC, a Fortran template library for containers, which provides analogs of C++ Standard Template Library Vector and Map templates.[1]

One of the more challenging aspects of pFLogger's implementation was allowing for a (nearly) arbitrary list of data arguments in the primary interface methods. Unlike C and Python, Fortran lacks language support for variable argument lists. To avoid a combinatorial explosion in the number of supported interfaces we opted instead to support a long (but finite) list of optional arguments of unlimited polymorphic type. The dynamic type of unlimited polymorphic entities can be of any Fortran type (intrinsic or user-defined).[4] The present arguments are aggregated into a vector container for subsequent processing.

2.4 Configuration examples

Here we provide an example that demonstrates a simple pFLogger configuration file and a snippet of instrumented Fortran code. The configuration file below consists of a single logger, 'main', which has a single handler, 'console', that routes messages to standard output:

```
handlers:
   console:
      class: streamhandler
      unit: OUTPUT_UNIT
      level: WARNING
loggers:
   main:
      parallel: .false.
      handlers: [console]
      level: INFO
```

With this configuration, only messages with a severity of warning or above will appear in the output. Below is an instrumented Fortran program that works with the above configuration:

```
use pflogger
class (Logger), pointer :: lgr
real :: x(4)
! Get a pointer to the logger named 'main'
lgr => logging%get_logger('main')
! Format a string and an integer
call lgr%info('%a␣:␣%i2␣', month, imon)
! Format 3 comma separated integers.
call lgr%info('DIMS:␣%i4~,%i4~,%i4', im, jm, km)
! 4 comma separated reals in an array
call lgr%warning('x=%4(f12.2,1x)',WrapArray([x]))
end program
```

3 DISCUSSION

To assess pFLogger we have partially instrumented modelE[8], a large climate model developed at the NASA Goddard Institute for Space Studies. As this was an internal development project, the metrics of success were informal and broadly amount to whether other model developers found sufficient value to migrate remaining

[3] One unfortunate drawback of this choice is that the field widths Fortran edit descriptors *follow* the type indicator which can lead to some ambiguity in where a descriptor ends. To resolve this we require descriptors to be followed by a space character or a tilde character when a trailing space is not desired.

[4] To work with non-scalar arguments, users unfortunately must wrap array arguments using the provided WrapArray() function, but this inconvenience will disappear once compilers start supporting the assumed rank features promised in Fortran 2015.

diagnostics to the new approach. During development, we had received very positive feedback from several developers, but it was unclear as to how the features would be perceived in actual use. Some specific metrics we have considered are:

- Does the package behave as expected in terms of the desired output?
- Is the run time overhead within tolerable limits?
- Are the interfaces and configuration sufficiently understandable by physical scientists?

Unfortunately, very early in the deployment we encountered a significant issue that we had not anticipated – *compilation* overhead. While pFLogger itself compiled quickly, the compilation of modelE stalled after simply inserting a single 'USE pflogger' into a low-level modelE module. We found similar issues with a second compiler. Investigation revealed that the ".mod" files produced by the compilers were effectively doubling in size at each level in the use hierarchy. The problem is *not* intrinsic to Fortran nor our design, as yet another compiler clearly maintained modest file sizes for ".mod" files. Unfortunately modelE itself did not support that compiler which prevented its use for that aspect of our evaluation. We were able to implement some workarounds that enabled a complete build the instrumented model with both of the original compilers, but still with significant increase in build time. Table 1 shows the average times for compiling modelE with and without pFLogger and using the Fortran compilers from Intel and GNU. Problem reports have been filed with vendors and as the times for the beta release of Intel 18 indicate, this issue should improve in the future. Unfortunately, the existing overhead was sufficient for rejection of pFLogger by modelE developers, and we therefore cannot yet evaluate other measures of user acceptance.

	GCC 7.1	Intel 17.0	Intel 18.0 beta
Baseline	44	81	80
pFLogger	95	210	119

Table 1: Typical compilation time in seconds for 3 different compilers.

To evaluate run-time performance we have measured the partially instrumented version of modelE and crafted several synthetic use cases that stress different. Ideally pFLogger performance would be nearly identical to hard-coded print statements, but some performance overhead is unavoidable due to the greater run-time flexibility. We anticipate that modest performance impacts will be acceptable to most users in trade for the improved capabilities. Production runs can leverage the tool to eliminate or minimize low-severity log messages, and debugging runs can tolerate a larger performance impact. For modelE, with only 1% of the output instrumented, there was no measurable impact to performance. We will measure again when a larger fraction of the model is instrumented, but do not expect any significant overhead.

Our first three synthetic use cases compare the relative performance of equivalent pFLogger and native logging.

(i) A text message with basic pFLogger processing.
(ii) A message that formats a few scalar variables
(iii) A parallel log message, with each process writing to a separate file.

For these cases, we report the time ratio of pFLogger to hard-coded implementations. The remaining two use cases do not have simple native analogs:

(iv) A below-threshold message.
(v) A parallel log message written to a shared file.

Case (iv) evaluates the effectiveness of suppressing a message at runtime. This capability is of limited value unless it at least measurably faster than a normal print statement. Case (v) measures synchronization overhead by comparing against a linear extrapolation from one process. Table 2 summarizes the results.

These results show that the initial performance is likely acceptable for many applications/environments but that further optimization is desirable. We note that there is actually considerable variability (as much as 5x) among these vendors for their baseline performance. Based upon separate microbenchmarks, we surmise that these are due to differences in the provided I/O libraries as well as in the management of small dynamically allocated strings. Variations in the reported performance ratios for pFLogger are to a large extent a reflection of this baseline variability.

Use case	Intel 17.0[a]	GCC 7.1[b]	NAG 6.1
(i)	5.5x	10x	15x
(ii)	8x	16x	5x
(iii)	1.1x	7x	-
(iv)	0.004x	0.03x	0.15x
(v)	5x	8x	-

[a] Intel MPI 5.1 [b] OpenMPI 2.1

Table 2: Measured times and performance ratios for use cases (i)-(v) for 3 different compilers.

We believe that tools such as pFLogger will be of significant benefit to HPC application developers and end users in the future by allowing a more coherent and flexible approach to application diagnostics. To this end, we have started NASA's internal process for releasing pFLogger as open source and hope to make it available to the community in the coming year.

REFERENCES

[1] T. Clune and D. Feldman. The Fortran template library for containers. in preparation.
[2] C. Hill, C. DeLuca, V. Balaji, M. Suarez, and A. d. Silva. The architecture of the Earth System Modeling Framework. *Computing in Science and Eng.*, 6(1):18–28, Jan. 2004.
[3] C. Jin, S. Klasky, S. Hodson, W. Yu, J. Lofstead, H. Abbasi, K. Schwan, W. G. Tech), and R. O. S. N. Laboratories. Adaptive IO System (ADIOS).
[4] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 39–, New York, NY, USA, 2003. ACM.
[5] Y. Meurdesoif. Xios fortran reference guide. http://forge.ipsl.jussieu.fr/ioserver/raw-attachment/wiki/WikiStart/XIOS_reference_guide.pdf, 2016.
[6] J. Reid. The new features of Fortran 2003. *SIGPLAN Fortran Forum*, 26(1):10–33, Apr. 2007.
[7] V. Sajip. logging – logging facility for Python. "https://docs.python.org/3/library/logging.html". Accessed: 2017-13-08.
[8] G. Schmidt et al. Configuration and assessment of the GISS ModelE2 contributions to the CMIP5 archive. *J. Adv. Model. Earth Syst.*, 6(1):141–184, Jan. 2014.
[9] The HDF Group. Hierarchical data format version 5, 2000-2010.