2015 IEEE 23rd International Conference on Program Comprehension

Limpio — LIghtweight MPI instrumentatiOn

Milan Pavlovic[*†], Milan Radulovic[*†], Alex Ramirez[‡] and Petar Radojković[*]
[*]Barcelona Supercomputing Center, Barcelona, Spain
[†]Universitat Politècnica de Catalunya, Barcelona, Spain
[‡]NVIDIA Research, Santa Clara, CA
http://www.bsc.es/computer-sciences/computer-architecture/memory-systems/limpio

***Abstract*—Characterization of high-performance computing applications often has to be done without access to the source code. Computer architects, therefore, have a narrowed choice of instrumentation tools. Moreover, potentially large amount of collected data can prohibit creating a full timestamped event trace and analyzing it post-mortem. This paper describes Limpio — a LIghtweight MPI instrumentatiOn framework, that allows dynamic instrumentation of user-selected MPI calls, and customization of data gathering, analysis and visualization.**

I. Introduction

Computer architecture research often involves measuring various hardware-related statistics on a given platform during workload executions. The ability to easily setup experiments, and process collected results with little overhead, becomes very important when the researcher needs to repeat the experiments many (hundreds of) times in different execution environments. Thus, the choice of profiling tool, suitable for the problem under study, can be a significant research decision.

Profiling tools typically target two categories of users. First, software engineers, that are interested in application-related statistics, such as time consumption breakdown for each function in the code. They can use this information as an indicator on where to prioritize their code optimization efforts. Second, computer architects, that look for statistics about hardware performance counters, for having a better insight to the system from an architectural perspective. Their goal is to better understand *how well the hardware executes the workload*, instead of *how well the application is written*.

Both types of profiling tools collect their target statistics by introducing instrumentation routines in key points in the application. That typically requires manual or automatic modification of the code, and recompiling, or at least relinking application with the instrumentation library. While this is a trivial step in application development, hardware engineers are usually provided only with the application executable, without access to its source code, or details about the compilation process. Profiling can then only be done with tools that directly instrument the binary, or use wrapper interposition libraries.

Modern high-performance computing (HPC) clusters execute parallel applications typically written around an inter-process communication library, OpenMP or MPI. Therefore, communication function calls are the most obvious candidates for the instrumentation points. However, even on a 1000+ core machine, many applications can execute for hours or days, calling communication functions countless times. This raises two main concerns. First, instrumentation routines need to be lightweight, to avoid introducing overhead with respect to native execution, and that way distort time-related measurements, or have side-effects to cache or memory usage. Second, a timestamped trace of all communication events can become larger than what the processing tools can handle, or even larger than the available storage space on a shared cluster[1]. In such cases user might want to explicitly select the instrumentation points or choose to trace only a short execution segment.

Tools for post-mortem analysis, with their complexity, can impose a long learning curve to the beginners. Many times users find it easier to express the requirements of the analysis with short snippets of code, than to spend significant effort learning the features and coping with versatility of such tools.

In this paper we present Limpio — LIghtweight MPI instrumentatiOn framework. Limpio enables the following design goals:

- Users can profile the application without recompiling. Instrumentation is done by dynamic linking.
- Limpio core is lightweight — it introduces no overhead to the native execution. Users themselves are responsible for the overhead of the instrumentation routines.
- Limpio is customizable — users can write instrumentation routines highly specific to the analysis requirements.
- Limpio is extensible — users can invoke external tools from within instrumented calls.

II. Background

In order to parallelize numerical computation, scientific HPC applications divide and distribute input data over a large number of processes. Then, through a series of iterations and inter-process communication steps, they combine intermediate calculations into a final result. Therefore, scientific HPC applications naturally follow repetitive patterns, so characterizing their behavior in a few iterations of the main loop is equivalent to characterizing their entire execution [1]. Similarly, most of the processes execute the same algorithm on different data, so the behavior of a few processes can well represent the behavior of the entire application.

Most production MPI applications are by default dynamically linked against MPI library on a given system[2]. Although

[1]Some of the applications we analyze produce traces of over 500 GB

[2]In order to use static linking, user needs to explicitly specify an appropriate flag when compiling the application, which is rarely done.

preinstalled shared libraries are typically found and loaded by the operating system, user can also create and preload his own library instead. By creating a wrapper for each standard library function, user creates a mechanism to inject his own instrumentation routines before and after the library call, without recompiling or relinking the original application, and without disrupting native application behavior and functionalities.

III. Architecture

Figure 1 presents a sequence diagram of an instrumented MPI call in the Limpio framework, and shows the interaction between the application, user-defined instrumentation routines, and standard MPI library. Limpio exposes two levels of interfaces — one to the application, and other to the user functions. From the application perspective, Limpio provides a set of wrapper functions for intercepting all MPI calls. In wrapper functions, Limpio first invokes a user-defined function that instruments the start of the MPI call. Then, the execution is passed to the corresponding function from the MPI library. Finally, after the MPI call completes, Limpio invokes a user function for instrumenting the end of the MPI call (see Figure 1).

Limpio, by itself, performs no application profiling. Instead, it allows users to create their own analysis tools by writing instrumentation routines. Limpio hooks the instrumentation routines with relevant MPI wrapper functions, and creates a shared library object, preloaded before the execution. The instrumentation routines, when invoked by the application MPI calls, measure statistics relevant to the user. Gathered data can be stored as a timestamped event trace for post-mortem analysis, or processed online (while the application runs) for producing a summary of statistics after the application completes.

```
#include <stdio.h>
#include "limpio.h"

static void start(mpi_function_id_t mpi_function_id) {
   printf("Start %s\n", get_mpi_fn_name(mpi_function_id));
}
static void end(mpi_function_id_t mpi_function_id) {
   printf("End %s\n", get_mpi_fn_name(mpi_function_id));
}
__attribute__ ((constructor))
static void mpilog_init(void) {
   set_start_hook(&start);
   set_end_hook(&end);
}
```

Listing 1. Example Limpio tool that logs start and end of MPI calls

Listing 1 shows an example of a Limpio tool that logs start and end of MPI calls. Function `mpilog_init()`, invoked when the library is initialized, uses functions from Limpio framework to define MPI instrumentation routines — `start()` for instrumenting MPI call entry, and `end()` for MPI call exit. These routines print a message on the standard output, when invoked by the MPI call from the application. Similarly to `mpilog_init()`, there may exist `mpilog_fini()`, invoked after the application is completed, for outputting any statistics accumulated during the execution.

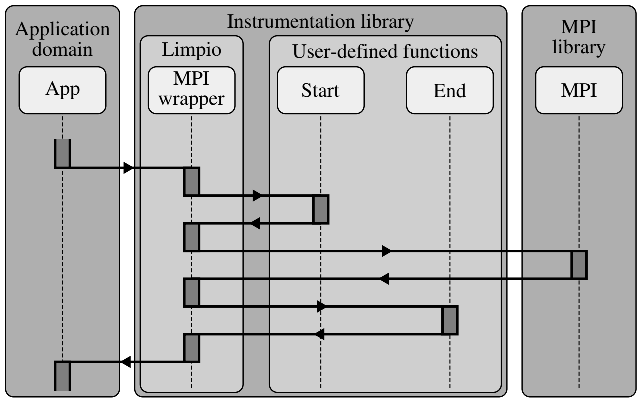

Fig. 1. Limpio instrumentation — sequence diagram

IV. Case studies

Limpio was developed for characterizing large-scale HPC applications. In Computer Architectures group at Barcelona Supercomputing Center, Limpio is regularly used to analyze production HPC applications running on MareNostrum [2] supercomputer. MareNostrum is one of six Tier-0 HPC systems in PRACE [3], containing 3,056 compute nodes, each with two Intel Sandy Bridge-EP E5-2670 sockets with eight cores operating at 2.6 GHz.

Access to MareNostrum allowed us to run experiments using up to several thousand cores. Because of a large number of application processes and long execution time of production HPC applications, it was essential to prepare the instrumentation tools for processing massive amounts of data. Limpio provided sufficient room for customization, and we used it to create several tools, each targeting different aspect of application characterization. In the following sections, we describe some of the Limpio tools, and illustrate their usage for profiling of ALYA application [4].

A. *MPI profiler*

MPI profiler tool produces basic statistics about MPI calls — their per-process and total number, and the accumulative time they consume relative to the total application execution. MPI profiler calculates these statistics using online processing — on each instrumented call, it increments the counter of the corresponding MPI function, and accumulates the duration of the call. After the execution, the tool generates a summary, as presented in Table I, without producing any traces or intermediate results during the application run.

Table I gives a breakdown of MPI calls (columns) over application processes (rows). Each entry in the table shows how many times one MPI function is invoked by a particular process, as well as the aggregate duration of these calls relative to the total application execution time (shown in parentheses). Last column summarizes MPI communication for each process by showing total number of MPI calls, and their relative duration. Last row gives an overview of each MPI function usage combined over all the processes.

TABLE I
MPI PROFILE, ALYA APPLICATION, 256 PROCESSES

	MPI_Send	MPI_Recv	MPI_Allreduce	MPI_Barrier	MPI_Bcast	MPI_Allgatherv	MPI_Gatherv	MPI_Sendrecv	Total
Process #1	7650(0.4%)	2314(0.1%)	74757(90.0%)	63(0.0%)	28(0.0%)	2(0.2%)	1(0.0%)	0(0.0%)	84819(92.9%)
Process #2	9(0.2%)	30(6.2%)	74757(30.3%)	63(0.0%)	28(0.0%)	2(0.1%)	1(0.0%)	140635(12.8%)	215529(51.4%)
Process #3	9(0.2%)	30(6.2%)	74757(41.2%)	63(0.0%)	28(0.0%)	2(0.1%)	1(0.0%)	84381(3.3%)	159275(52.9%)
Process #254	9(0.3%)	30(6.1%)	74757(40.0%)	63(0.0%)	28(0.1%)	2(0.1%)	1(0.0%)	56254(1.6%)	131148(49.7%)
Process #255	9(0.3%)	30(6.1%)	74757(40.1%)	63(0.0%)	28(0.1%)	2(0.1%)	1(0.0%)	56254(2.0%)	131148(50.2%)
Process #256	9(0.3%)	30(6.1%)	74757(38.9%)	63(0.0%)	28(0.1%)	2(0.1%)	1(0.0%)	112508(2.6%)	187402(49.6%)
Total	9967(0.3%)	9967(6.1%)	19137792(31.4%)	16128(0.0%)	7168(0.1%)	512(0.1%)	256(0.0%)	39489832(7.6%)	58672646(47.2%)

Profile of MPI calls is our first step in HPC application analysis. It gives an overview of the MPI calls that the application uses, and verifies that most of the processes have similar behavior from MPI communication perspective. Table I shows that ALYA has a master-worker architecture, where first process, substantially different from the rest, has a role of master, while other 255 processes serve as workers, and have similar MPI communication profiles.

The same analysis could well be performed with some of the existing tools, that can generate full timestamped MPI event traces, and, in a post-mortem analysis, summarize event count and duration. However, with that approach, HPC applications would generate very large MPI traces, which would make post-mortem analysis prohibitively slow, and could easily surpass the available disk quota on a shared HPC system. For this reason we chose online analysis to produce this kind of statistics.

B. Computation to communication ratio

Another important metric for analysis of HPC applications is computation to communication ratio. When inter-process communication dominates over effective computation time, especially on high number of processes, gains in parallel performance can diminish and be prevailed by the execution cost. Limpio MPI profiler measures relative MPI communication time — the time spent in all MPI library function calls, relative to the total execution time. Increase in relative MPI communication time, with the increase in the number of processes, indicates limited application scalability.

Figure 2 shows relative MPI communication time for ALYA application, for various number of processes. The figure separates MPI communication time of the master process, and the average of the worker processes. The results show how MPI communication starts being dominant for high number of processes, which decreases the relative time spent in computational segments. Therefore, the increase in parallelism is not paid off by the appropriate performance improvement (for a 256× increase in number of processes — from 4 to 1,024, we measure only 86× speedup).

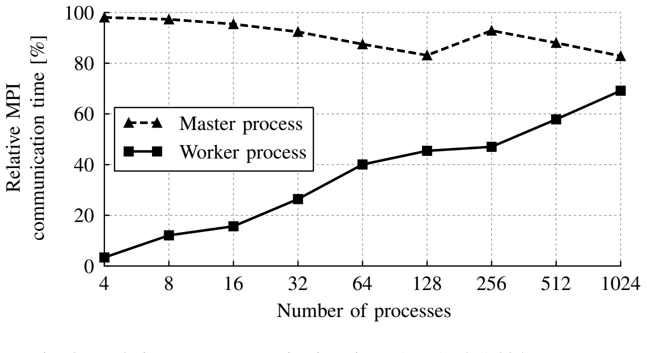

Fig. 2. Relative MPI communication time, ALYA, 4–1,024 processes

This experiment required analysis of nine executions of the application (from 4 to 1,024 processes). By performing online analysis with Limpio MPI profiler, we avoided time-consuming process of collecting a large timestamped event trace and its post-processing. Moreover, optimized instrumentation routines introduced negligible overhead in the instrumented execution, which is important when measuring any time-related statistics.

C. Tracing and visualization

Profilers typically summarize their target statistics in a concise table, sacrificing data about their evolution and variability throughout the execution. Observing the changes in measured metrics over time gives insight in application-specific patterns, iterative behavior, correlation between measured metrics, or transient bottlenecks in the system. For that, we need a trace of timestamped events, triggered by an MPI call, where the current state of user-defined statistics is measured and saved.

In order to avoid producing large trace files, and to make their analysis or visualization feasible, Limpio provides a mechanism to selectively disable instrumentation in specific MPI function calls, by setting the environment variable `MPIINSTR_EXCLUDE`. Once the users obtain the MPI profile, they can disable instrumentation in the most frequent MPI calls. That way the granularity of the instrumentation points is coarsened, and the trace size is reduced. In our example, ALYA produced a profile (Table I) where `MPI_Allreduce` and `MPI_Sendrecv` constitute the vast majority of all MPI calls. Disabling instrumentation of these two calls (Listing 2) reduces the trace to the size that can be handled by the tools for visualization or post-mortem analysis.

```
...
export MPIINSTR_EXCLUDE=MPI_Allreduce,MPI_Sendrecv
...
```

Listing 2. Excluding instrumentation of specific MPI functions

Traditional tools for trace visualization have to be compatible with the trace format generated by instrumentation tools. In

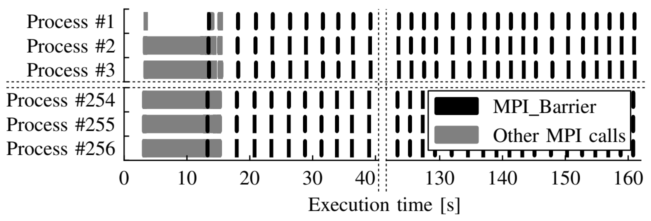

Fig. 3. MPI call visualization, ALYA application, 256 processes

Limpio, users can define the trace format in the instrumentation routines, and then visualize the trace using any external general-purpose visualization tool or library.

Figure 3 presents an example of a visualization of an MPI communication trace. It is obtained by running ALYA, instrumented and traced with Limpio (with most frequent MPI calls excluded), and rendered using Matplotlib. The figure confirms application repetitive behavior, with `MPI_Barrier` as the call that precedes and follows each iteration.

D. External instrumentation tools

The ability to recognize boundaries of application iterations, allows us to use Limpio for invoking external tools, and using them in a precisely defined segment of the application execution. For example, with the information extracted from a trace of timestamped events (Section IV-C), Limpio tool can be configured to detect iterations of the main loop, that are good representative of the overall application behavior. In that segment, it can invoke tools for instrumenting application at the instruction-level granularity, such as Pin [5] or Valgrind [6].

An example code for that functionality is given in Listing 3. In function `start()`, it is determined if the execution has reached target iteration (nth call of a target MPI function), and in that case, external instrumentation tool is attached to a running process. Similarly, in function `end()`, when the target iteration completes, Limpio sends a signal to the external tool, and detaches it from the process.

```
#include <signal.h>
...
static void start(mpi_function_id_t mpi_function_id) {
   if (is_instrumentation_segment_starting()) {
      sprintf(attach_instumentation_command, "%s -pid %ld",
         path_to_instrumentation_binary, (long)getpid());
      system(attach_command);
   }
}
static void end(mpi_function_id_t mpi_function_id) {
   if (is_instrumentation_segment_ending())
      kill(getpid(), SIGUSR1);
}
...
```

Listing 3. Attaching and detaching an external instrumentation tool

Combining Limpio with external instrumentation tools has multiple uses. First, instruction-level instrumentation usually introduces large performance overhead, and it is reasonable to apply it only on a short execution segment. Second, if external instrumentation is used for generating instruction or memory traces, their size can be controlled by modifying the length of the tracing segment.

V. Related work and tools

Tool mpiP [7] produces MPI function profiles similar to Limpio MPI profiler. It does not require recompiling, but it needs linking to the mpiP library. For profiling a segment of the execution, user needs to change application source code.

Score-P [8] provides an infrastructure for profiling, tracing, and online analysis of HPC applications. Instead of direct instrumentation, it can use sampling, but users need to recompile the application with Score-P instrumentation command.

Extrae [9] is a tool for tracing application performance data. Extrae can instrument applications based on a wide range of programming models (MPI, OpenMP, CUDA, etc.). With various interposition mechanisms for injecting probes into the target application, it provides many instrumentation options for HPC application analysis. Extrae uses clustering to detect iterative patterns and to choose regions of interest to present to the analyst [10]. However, it does not allow user to instrument a subset of MPI calls, nor to perform online analysis. Extrae is not designed for triggering calls to external profiling tools.

VI. Summary

This paper presents Limpio, a framework for profiling of MPI applications. Limpio overrides standard MPI functions, and executes instrumentation routines before and after the selected MPI calls. Users themselves can write and customize the instrumentation routines to fit the requirements of the analysis. Limpio can invoke external application profiling tools, and can switch between various tools in a single execution. It can also generate application traces of timestamped events that can be visualized by general-purpose visualization tools or libraries. Limpio is regularly used in Barcelona Supercomputing Center for instrumentation of large-scale HPC applications.

Acknowledgment

This work has been supported by the Spanish Ministry of Science and Innovation under grant TIN2012-34557, Generalitat de Catalunya under grant 2014-SGR-1261 and 2014-SGR-1051, and Severo Ochoa recognition SEV-2011-0067.

References

[1] R. Preissl *et al.*, “Detecting patterns in MPI communication traces,” in *International Conference on Parallel Processing (ICPP)*. IEEE, 2008.
[2] Barcelona Supercomputing Center, “MareNostrum III System Architecture,” http://www.bsc.es/marenostrum-support-services/mn3, 2013.
[3] Partnership for Advanced Computing in Europe (PRACE), “Prace research infrastructure,” http://www.prace-ri.eu.
[4] *Unified European Applications Benchmark Suite*, Partnership for Advanced Computing in Europe (PRACE), Jul. 2013.
[5] C.-K. Luk *et al.*, “Pin: Building customized program analysis tools with dynamic instrumentation,” *SIGPLAN Not.*, vol. 40, no. 6, Jun. 2005.
[6] N. Nethercote *et al.*, “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation,” *SIGPLAN Not.*, vol. 42, no. 6, Jun. 2007.
[7] J. Vetter *et al.*, “mpiP: Lightweight, Scalable MPI Profiling,” 2005.
[8] A. Knüpfer *et al.*, “Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir,” in *Tools for HPC 2011*. Springer Berlin Heidelberg, 2012, pp. 79–91.
[9] H. Gelabert *et al.*, “Extrae User Guide Manual for version 2.2.0,” *Barcelona Supercomputing Center (BSC)*, 2011.
[10] G. Llort *et al.*, “On-line detection of large-scale parallel application’s structure,” in *International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–10.