Lightweight Performance Data Collectors 2.0 with Eiger Support

Benjamin Allan - baallan@sandia.gov

Contents

1 Overview

The lightweight performance data collectors with Eiger support is intended to be a tailorable tool, not a finished product, as profiling needs vary widely. A single code markup scheme is provided which, based on compilation flags, can send performance data from parallel applications to CSV files, to an Eiger mysql database, or (in a non-database environment) to flat files for later merging and loading on a host with mysql available.

1.1 Introduction

Light-weight Performance Data Collectors (lwperf) is a tiny collection of simple, portable macros and an underlying tiny set of tailorable C++ classes aimed at making it easy to gather high-level compute cost numbers and the driving algorithm parameters from individual cluster nodes running real applications. The author's intended use of these numbers is to construct models that support interpolation-based estimates of compute costs at other parameter values.

The collectors support three log formats:

- CSV (tabular) data for modeling with spreadsheets, matlab, and other common tools.
- (optional) Eiger[**?**] database logging, which requires MySQL libraries and a server.
- (optional) FakeEiger text logging, which avoids the MySQL requirements by generating portable data files that can be read into an Eiger database later in a MySQL-enabled environment.

A key constraint is that the code markup scheme this performance tool uses must balance having minimal affects on the appearance, performance, or compilation of the code while at the same time not requiring a proprietary library or compiler or analysis tool or interposed virtual machine. Most prior work sacrifices at least one major aspect of this constraint.

This tool is not intended to collect data for code segments containing inter-node communication code, particularly MPI code. Rather, it is intended for characterizing node-level serial, locally multi-threaded (OpenMP nests in an MPI/OMP hybrid), or locally accelerated code sections. It is usually the performance of the node or local (co)processor group in aggregate rather than individual thread performance which is of interest, as it is the group which provides the total workload to shared local resources such as memory. Single-node but multi-rank instances of MPI-only codes may also be usefully profiled with this tool. There is no technical reason lwperf cannot be used to profile MPI calls, but there are better tools for profiling MPI widely available.

Using simple shell scripts, lwperf has been extended to support C and modern Fortran (any compiler supporting the BIND(C) feature).

1.2 Background

The driving need for this tool is portability: numerous superior (and heavy-weight) but proprietary performance analysis tools exist. As with mini-applications, we need mini-analysis tools. Free MPI-oriented profiling tools already exist, but they usually stop at profiling communications and do not collect the parameters that control local workloads. Source-level insight is required to identify and capture the influential parameters.

2 Implementation

The collector is aware that node-locally there may be a number of OMP threads or MPI ranks relevant to performance modeling and expects the user to pass in that information as part of the initialization or input parameter data at measurement sites. In the case of multiple MPI ranks on the node, each log file name is suffixed by the MPI communicator rank and size passed in with an init call. No interprocess communications (except those of connecting to the Eiger database or the output filesystem) are introduced by using the collector.

The logger backends (eigerformatter, csvformatter) support eiger, fakeeiger, or CSV. In the eiger cases, the default is to generate CSV files also. This default can be changed by passing -D_EIGER_NOCSV to the c++ compiler.

The backends can also generate text to the screen (in csv format) if the screen option is turned on by passing -D_LWPERF_SCREEN to the c++ compiler.

2.1 Dependencies

Eiger The collectors depend on libmpieiger or libmpifakeeiger if -D_USE_EIGER is applied. Eiger is not needed if only CSV collection is used.

MPI From MPI, the collectors use only MPI_Wtime. This may be easily replaced with boost or any other available and acceptable high resolution clock difference. The MPI compiler wrappers are used because of this dependency.

C++ STL classes and variadic macro support.

Fortran Basic macro substitution and the bind(c) feature, which is available in gfortran and most other fortrans as of 2011. Formally, bind(c) is part of the Fortran 2003 specification. Variadic macro support is not required for the Fortran binding.

C Variadic macro support

3 C++ Application Programming Interface

The C++ interface and usage is presented here. Alternate versions for C and Fortran are provided in later sections.

3.1 Setup

For C++, the profiled code must include a single header, lwperf.h, which pulls in definitions for the data collection macros based on the compiler flags: -D_USE_EIGER and -D_USE_CSV. If neither flag is provided, a set of null macros make all the collector-related code disappear from the build. After lwperf.h is included in the driver and all files where collection is needed are listed in the updateLoggersCXX.sh script, the driver has a simple init/configure/run/-finalize use sequence.

From application driver cxxapp.cpp:

```
PERFDECL(PERF::init();)
// collect communicator rank and size : profiling a single-node job using
PERFDECL(PERF::mpiArgs(me,csize);)
PERFDECL(PERF::stringOptions("x5550","gcc","cxxapp.cpp","cxxapp","tesla.",".log");)
```

Here all the initialization calls are wrapped in PERFDECL, a macro that deletes the calls if the profiler is not enabled by one of the USE flags. Unlike many tools, performance collection sites can be disabled on a per-file basis by eliminating from the compiler arguments the -D_USE_(CSV|EIGER) flag. On the driver and all files where collection is enabled, the same flag (EIGER or CSV) must be used; mixing is not allowed.

3.2 Profile sites

At any site (usually a loop nest) where data collection is wanted, the collection point is defined at the beginning of the site with a unique name and additional log names for parameters characterizing the work load (usually integer). The values of the parameters do not need to be available at the beginning of the site; only names are needed. The name given the site must be unique across the entire application. The parameter names are specific to the site and may be reused elsewhere.

From a molecular dynamics example, with a variable work load where nlocal is known at the beginning but nneigh is dependent on double-precision input data arrays:

```
PERFDECL(int nneigh=0;); // counter used only when profiling
// site Tenergy, log parameter names nlocal, nneigh
PERFLOG(Tenergy, ID(nlocal),ID(nneigh));
for (int i = 0; i < nlocal; i++) { // outer loop
  ... // fixed work calculating, in part, data_dependent_condition
  if ( data_dependent_condition) {
    for (int j=0; j< numneigh; j++) {
      PERFDECL(nneigh+=numneigh;); // assignment needed only for profiling
      ... // conditional work
    }
  }
```

```
}
PERFSTOP(Tenergy, ID(atom.nlocal),ID(nneigh));
```

At the end of the site, PERFSTOP computes the elapsed performance measures (at minimum, wallclock) and these values are logged along with the values passed to the slots defined in the matching PERFLOG statement. In this case the value of nlocal comes from data named atom.nlocal. The data-dependent nneigh parameter is declared and updated only when lwperf is enabled (through use of PERFDECL).

The metric macros must be used on each parameter being recorded at the PERFLOG/PERFSTOP sites, or incorrect code will be generated. The generator scripts need these to detect the argument types. In the event of a macro or function name conflict, these macros can be renamed if all the affected lwperf headers and generator scripts are correspondingly updated.

There are several different parameter macros, as listed in the following table.

Name	Variable Type	Metric Type
IR	int	RESULT
DR	double	RESULT
ID	int	DETERMINISTIC
DD	double	DETERMINISTIC
IN	int	NONDETERMINISTIC
DN	double	NONDETERMINISTIC

3.3 Performance counters

The performance measures captured are defined in csvformatter.h:DEFAULT_PERFCTRS, with supporting definitions and computations required for each in the private data of csvformatter and the start/stop member functions. These are easily extended by the demanding user. At present, support for hardware performance counters is not included as it would add considerable dependency and portability problems. The enterprising user may tailor lwperf to collect hardware counters.

4 Building

Being lightweight by design, this tool is intended to be used by incorporation into the investigated application and its build processes rather than as an independent library. In order to keep performance impact to minimum without introducing a proprietary compiler, some simple portions of the support classes must be generated based on a scan of the sites defined in the application code. This scan is performed and the generated code is updated by the simple utility *updateLoggersCXX.sh*. This utility may be tailored to the user's application source code, in most cases by adjusting the definition of the *cfiles* variable at the top of the script.

The logger can be kept consistent with the user code with a simple makefile rule such as:

```
GENSRC=CSVInitFuncs.h InitSwitchElements.h EigerInitFuncs.h LocationElements.h
$(GENSRC):  $(SRC) updateLoggersCXX.sh
        ./updateLoggersCXX.sh
```

or by invoking *updateLoggersCXX.sh* at the beginning of the build process.

5 Running

The application is run as normally, but with output going to screen, disk, or eiger database. A simple example script, *runfake*, is provided to illustrate a parameter study and basic handling of the output files. The timing data obtainable should be reasonably accurate for measured non-overlapping compute sections, but the overall application performance may be strongly influenced by the extra I/O. The screen output option is for debugging purposes. Enabling screen output in MPI applications with many measurements has been observed to consume large amounts of stack data space; all-rank screen output is non-parallel and should generally be avoided.

6 C application support

6.1 C api

The application code changes are similar to those for C++, inserting macros where needed. The header to be included is *lwperf_ c.h*. For example:

```
#include "stdio.h"
#include "stdlib.h"
static
int n = 1;

void sub1(int x) {
  if (x < n) {
    x = x + 1;
    printf("x = %d\n", x);
    sub1(x);
  }
}

#include "lwperf_c.h"

int main(){
  int x = 0, nx=0;
  double dp = 3.14;

  PERF_INIT;
  PERF_MPI(0,1);
```

```
  PERF_OPTS("x5550","gcc","capp.c","capp","tesla.",".log") ;
  char buf[26];

  printf("Enter number of repeats\n");
  fgets(buf,sizeof(buf)-2,stdin);
  char *endPtr;
  n = (int)strtol(buf, &endPtr, 10);
  if (endPtr == &(buf[0])) {
    printf("given string, \"%s\" has no initial number\n", buf);
    exit(1);
  }
  printf("n = %d\n", n);

  PERFLOG(sub1,ID(n)) ;
  PERFLOG(sub2,ID(nx),DD(dp));
  sub1(x);
  PERFSTOP(sub2,ID(nx),DD(dp));
  PERFSTOP(sub1,ID(n));
  PERF_FINAL;
  return 0;
}
```

The PERFDECL macro is available for C use as it is in C++.

6.2 C build

As with C++, some simple portions of the C binding must be generated based on a scan of the sites defined in the application code. This scan is performed and the generated code is updated by the simple utility *updateLoggersC.sh* and supporting scripts *gencsave.body.sh*, and *./gencsave.sh*. These scripts may be tailored to the user's application source code, in most cases by adjusting the definition of the *cfiles* variable at the top of *updateLoggersC.sh*.

The logger can be kept in sync with the user code with a simple makefile rule such as:

```
cperf._stop.h: $(CSRC) updateLoggersC.sh
./updateLoggersC.sh
```

Compiling and linking mixed language code is often tricky. Here is an example that may be inspirational; further explanation is beyond the scope of this note. Several additional examples are included in the Make.c* build scripts.

```
mpicc.openmpi -g capp.c -D_USE_EIGER \
cperf.o eperf.o eigerformatter.o csvformatter.o diffrusage.o \
-lmpi_cxx -lstdc++ \
/home/baallan/sst/gatech/eiger-svn/api/gcc/libeigerInterface.a \
/usr/lib/libmysqlpp.a -lmysqlclient \
-o ../capp_u11c
```

7 FORTRAN application support

The fortran binding is, as lwperf in general, functional, preliminary, and intended for tailoring by the user. A preprocessor must be included in the compile step, either by passing a compiler option to force it or by using a suffix the compiler recognizes as needing preprocessing. The preprocessor does not need to be a separate C preprocessor; the builtin gfortran preprocessor is known to work.

7.1 FORTRAN api

The prefix lwperf_ is used for all lwperf symbols in fortran, to avoid conflicts with user code. The header to be included is *lwperf_f.h*. The application code changes are similar to those for C/C++, inserting macros where needed, e.g.:

```
   module test1
     integer :: n
     contains
     recursive subroutine sub1(x)
     integer,intent(inout):: x
       if (x < n) then
         x = x + 1
         print *, 'x = ', x
         call sub1(x)
       end if
     end subroutine sub1
   end module test1

#include "lwperf_f.h"

  program main
    use test1
    PERF_USE

    integer :: x = 0
    integer :: nx = 0
    integer :: ierr,rank = 0, sz
    PERFDECL(real(kind=lwperf_double) :: dp =0 ) !! 8-byte C double if using lwperf

    call MPI_INIT(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD,rank,ierr)
    call MPI_Comm_size(MPI_COMM_WORLD,sz,ierr)
    PERF_INIT
    PERF_MPI(rank,sz)
    PERF_OPTS("x5550","gfortran","app.F90","fortapp","tesla.",".log")
```

```
    if (rank == 0) then
      print *, 'Enter number of repeats'
      read (*,*) n
    end if
    call MPI_Bcast(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr);

    PERFLOG1(sub_1,ID(n))
    PERFLOG2(sub_2,ID(nx),DD(dp))
    call sub1(x)
    PERFSTOP2(sub_2,ID(nx),DD(dp))
    PERFSTOP1(sub_1,ID(n))
    PERF_FINAL
    call MPI_FINALIZE()
  end program main
```

Note that Fortran preprocessors do not support variable-length macros a la C99. Thus, we must use macro names that include the number of measurements. Also, the PERFDECL macro in Fortran will not accomodate statements containing commas.

7.2 FORTRAN build

As with C++, some simple portions of the Fortran binding must be generated based on a scan of the sites defined in the application code. This scan is performed and the generated code is updated by the simple utility *updateLoggersF90.sh* and supporting scripts *./gencsave.body.sh*, *./gencsave.sh*, and *./genfsave.sh*. These scripts may be tailored to the user's application source code, in most cases by adjusting the definition of the *ffiles* variable at the top of *updateLoggersF90.sh*.

The logger can be kept in sync with the user code with a simple makefile rule such as:

```
flocations.h: $(FSRC) updateLoggersF90.sh
        ./updateLoggersF90.sh
```

Compiling and linking mixed language code is often tricky. Here is an example that may be inspirational; further explanation is beyond the scope of this note. Several additional examples are included in the Make.f90* build scripts.

```
mpif90.openmpi  fperf.o app.F90 -D_USE_EIGER \
cperf.o eperf.o eigerformatter.o csvformatter.o diffrusage.o \
/home/baallan/sst/gatech/eiger-svn/api/gcc/libeigerInterface.a \
/usr/lib/libmysqlpp.a -lmysqlclient \
-lmpi_cxx -lstdc++ \
-o ../app_u11f90
```

A full FORTRAN MPI application (gtc) has been instrumented with lwperf, where the lwperf support code is treated as a library (generated at application

build time). This approach has the least impact on the fortran build process and directory structure; contact Ben Allan or Gilbert Hendry for access to this example.

8 Mixed-language application support

There is nothing in principle that prevents using lwperf in an application with mixed C/C++/Fortran sources. In practice, the code generating scripts assume a single list of files in the corresponding language can be searched with grep for all the profiling sites. Converting updateLoggersF90.sh to updateLoggersMixed.sh is an exercise left to the reader.

9 Instrumentation Reuse

In benchmarking, code is sometimes manually (or mechanically) translated between C++ and Fortran for various reasons. The macro design of lwperf is intended to support cut-and-paste reuse of site markup across language boundaries. The Fortran instrumentation tolerates but does not require trailing semicolons on macro uses, so PERFLOG/PERFSTOP uses from C++ can be pasted into equivalent Fortran sites. The only modifications needed being to insert or remove the argument count suffix to obtain the equivalent macro name.

10 Performance of the instrumentation

The overhead of performance measurements with lwperf is dominated by the costs of writing output to files or database connections and of calls on the microsecond real-time clock. The C and Fortran bindings, being wrappers, involve a few extra function invocations compared to the C++, but this is negligible compared to the I/O at present.

When eiger database profiling is active, the database presents a bottleneck to recording results from parallel execution. When CSV or flat file (fakeeiger) output is used, there is no contention for individual files; however, an unshared file is created for each site in each MPI process. With a very large number of sites and processes on a single host, the maximum number of file descriptors available from the operating system may be exhausted. Profiling sites which are never executed do not generate files at all; you pay for what you use.

10.1 Minutiae

The aims of the lwperf implementation tactics are:

- to make lookup of the logging objects (eiger pointer or file pointer containers) inexpensive. This is achieved by converting the site names into an enumeration or integer index. This enumeration must is constructed by

analyzing the entire application source code. The names are prefixed by the code generators so that the names of code entities (functions, classes, enum members) may be used as site names without conflict in nearly all cases.

- to avoid reconstructing or reopening complex objects with operating system presence, such as any file or any eiger object with a commit() member.
- to avoid inter-language string conversion.

In the C++ binding, a local variable is generated that enables a single object lookup to handle each site. In the C and Fortran bindings, portability and syntactic considerations force us to use two object lookups per site execution.

The only place that inter-language string processing occurs is in the PERF_OPTS macro of the Fortran binding.

11 Analysis

Lwperf makes collecting large amounts of performance data easy. Analysis of the data is beyond the scope of this document. Studying the impact of application parameters and compiler options on the runtime performance of specific code segments (loop nests) requires careful construction of a set of benchmark runs spanning some relevant parameter space. The generated CSV files can be easily aggregated and then imported or translated for use in any preferred tool (a spreadsheet, gnuplot, etc).

Early use of the tool has provided one general insight. When examining the wall-clock and processor-clock time reported for a profiling site, the two measurements may differ widely due to process interruptions. The most common interruptions will be reported in the other rusage fields recorded, though these fields normally report 0 for number-crunching loops. Consult the system documentation of getrusage for explanation of the reported fields.

The default data collected includes the wall-clock time. This enables production of timelines which capture what code section was active when, not just how much time was spent in each call.

12 Future work

There are several tasks of minor housekeeping or performance value:

- Put all the collector related items into a separate c++ namespace. At present, Perf and EigerPerf classnames and lwperf prefix are used.
- Modify the update scripts to accept an argument list instead of computing the list of files based on suffix with ls.

- Add dumping to memory buffer (ostringstream) and flush to disk on some schedule rather than i/o at each time a PERFSTOP is reached. In principle, a reasonable OS is already doing this for us, but in practice the buffer size is likely to be smaller than we might like.
- Modify non-eiger stringOptions semantics to put all files in a subdirectory, as noted in aperf.cpp.
- Revisit using a map instead of an array for the log index in Perf and EigerPerf classes.
- Revisit adding a flag to eliminate MPI-dependence and add boost dependency for timing. Not terribly interesting for HPC purposes, where MPI is a given.

13 Postscript

A rose by any other name would still be thorny. Abandon all hope ye who seek GUIs.