Internal Use MPI Wrappers for Linux

Bob Walkup (walkup@us.ibm.com), March 2, 2017

Quick Start.

There are many available MPI implementations and compiler choices for Linux systems, and so it is often necessary to build MPI wrapper libraries specifically for your particular combination. This can normally be done by setting the appropriate path to "mpicc", where mpicc uses gcc under the covers. Alternatively, you can compile with gcc and specify the path to the MPI include files. There are two main methods to use the wrapper libraries: either (1) link with a wrapper library such as libmpitrace.a or libmpitrace.so when you build your executable, or (2) use a shared-library library at run time. This second approach is the most convenient method and is recommended. Examples of the shared-library method are:

(A) In your run script, set LD_PRELOAD to point to libmpitrace.so :
```
export LD_PRELOAD=/path/to/libmpitrace.so
mpirun -np 128 your.exe
unset LD_PRELOAD
```

(B) If your "mpirun" command supports it, add an option to mpirun as shown in this example:
```
mpirun -trace /path/to/libmpitrace.so -np 128 your.exe  (Intel MPI)
```

An exception has been noticed for IBM's pempi implementation on Linux/Power. With that software, the "poe" command is used instead of mpirun, and with wrapper libraries, "poe" can be invoked recursively unless the LD_PRELOAD follows the "poe" command. For example :

(C) for IBM pempi, put in your run script : poe myscript ; where "myscript" has :
```
export LD_PRELOAD=/path/to/libmpitrace.so
your.exe
unset LD_PRELOAD
```

There are several ways that using LD_PRELOAD can run into difficulties. Dynamically loading the wrapper libraries is convenient, but in some cases you may need to be inventive or fall back to directly linking one of the wrapper libraries when you build the executable. If you directly link, you can choose either a static or dynamic wrapper library to link with. On Linux systems that use shared libraries for MPI, I suggest building and linking with libmpitrace.so. If your system uses statically linked executables, you will need to link with the static library, libmpitrace.a

Run the code and look at the text outputs that are produced, using the text editor of your choice.

It is recommended that you test the wrappers first, to make sure the call counts are correct; some examples are included for this purpose. MPI implementations use different methods to handle the profiling entry-points for Fortran MPI routines. As a result, if you use a library that contains wrappers for both Fortran and C entry points (libmpitrace.a), you may get double-counting of the calls from Fortran. The libmpitrace_c.a library has only C entry points, and the libmpitrace_f.a library has only Fortran entry points. Normally, one of these libraries will be appropriate for applications that call MPI from a mix of Fortran, C, and/or C++ routines … but which library is appropriate varies with the MPI

distribution, so testing is essential. At the present time (2016-2017), the most common MPI implementations are based on openmpi-2.0 or higher, IBM's Spectrum MPI, mvapich2, Intel MPI, and other mpich-based implementations. For all of those, the combined library, libmpitrace.so, is appropriate for applications that call MPI from any mix of Fortran, C, and C++.

An example of the MPI timing summary for the Sequoia benchmark SPHOT is shown below.

```
Data for MPI rank 0 of 1024
Times and statistics from MPI_Init() to MPI_Finalize().
-----------------------------------------------------------------
MPI Routine                   #calls     avg. bytes      time(sec)
-----------------------------------------------------------------
MPI_Comm_size                      3            0.0          0.000
MPI_Comm_rank                      5            0.0          0.000
MPI_Send                        1023          192.0          0.005
MPI_Irecv                       5115          206.4          0.003
MPI_Waitall                        5            0.0          1.668
MPI_Bcast                          3        62125.3          0.001
MPI_Barrier                        3            0.0          0.000
MPI_Reduce                         4         3925.0          0.001
MPI_Allreduce                      5           15.2          0.000
-----------------------------------------------------------------
MPI task 0 of 1024 had the maximum communication time.
total communication time = 1.677 seconds.
total elapsed time       = 38.647 seconds.
...
```

In this particular case the total amount of time spent in MPI is a small fraction of the elapsed time, and the parallel efficiency is very high … so one should focus on computational performance.

Interpretation of MPI timing data requires some care. The most common issue is sorting out the effects of process synchronization and actual message transmission. To do this, it helps to have access to basic information, such as ping-pong times as a function of message size for processes that are either on the same node, or on different nodes. Similarly, basic information on the performance of collective MPI routines is useful. For example, if an application spent 100 seconds making 4000 calls to MPI_Allreduce using 8-byte messages, you can be sure that synchronization time dominates, because the network time required for one 8-byte MPI_Allreduce is roughly in the ~10 microsecond range, depending on the communicator, reduction operator, and so forth … so ~4000 calls should take less than ~1 sec. It is very common that the majority of the time spent in MPI is related to process synchronization instead of the latency or bandwidth characteristics of the network. A feature has been added to the wrappers, to help identify synchronization time: you can set env variable COLLECTIVE_BARRIER=yes. That will automatically add a barrier before every collective call, and separately report the synchronization time. This variable can also be set more selectively, as described later. However, many applications accumulate a lot of synchronization time in routines such as MPI_Wait, MPI_Waitall, etc., and there is no simple mechanism to separate the synchronization time for those routines. As a result, you will normally need to be aware that there may be hidden synchronization included in the time reported for MPI functions.

All features are included in both the shared-library and static-library versions. The shared-library version should work for basic MPI profiling and tracing if the MPI implementation is using a shared library for MPI routines. The shared-library approach is the simplest one to use because you typically need to add a line or two to your run script, without requiring a special linking step.

To use one of the static libraries, you must link the appropriate library, such as libmpitrace.a, with your application. A sample makefile might look like this:

```
CC = mpicc
TRACE = /path/to/libmpitrace.a
CFLAGS = -g -O3
LDFLAGS = -g
OBJS = your list of .o files ...

your.exe : $(OBJS)
      $(CC) $(LDFLAGS)  -o  your.exe  $(OBJS)  $(TRACE)

%.o : %.c
      $(CC)  -c  $(CFLAGS)  $<
```

If you directly use one of the compiler commands such as “icc” or “xlc_r” for the linking step, then you must explicitly include the system's MPI libraries; and the MPI wrapper library must come after the user's object files, and just before the system's MPI libraries. If you use some form of convenience script for the linking step, such as mpicc or mpif90, the system's MPI libraries are automatically included, and so it would normally be sufficient to add $(TRACE) at the end of the list of items to link. Note that link order is important on Linux systems. The MPI wrapper library must come after all of the user's .o and .a files that contain references to MPI routines, and it must come before the actual MPI library that is being used; otherwise the MPI entry points might not get wrapped, and there will be no profiler output.

If you use the static library method, it is possible to check the executable to make sure the instrumented MPI entry points were built in. To do that, use “nm your.exe | grep MPI_Init” to locate the instruction-address of the program text section (T) for MPI_Init(). Then feed that instruction-address into the addr2line utility : addr2line -e your.exe hex_address. The output from addr2line should point to the source-code for the instrumented version of MPI_Init() in the wrapper library.

Introduction.

The MPI specification provides profiling entry points for MPI routines. This enables end users to replace the normal MPI entry-points with their own wrappers, which can be useful for performance analysis and sometimes for debugging purposes. For example, you can use a wrapper for MPI_Recv() that has this outline:

```
int MPI_Recv(void * rbuf, int count, MPI_Datatype type, int src, int tag,
                MPI_Comm comm, MPI_Status status)
{
    int rc;
    time1 = get_timer();
    rc = PMPI_Recv(rbuf, count, type, src, tag, comm, status);
    time2 = get_timer();
   log_the_event(...);
   return rc;
}
```

With the approach sketched above, it is simple to keep track of the total amount of time spent in MPI routines, and there are many such tools that can do this. It is preferable if the overhead required to keep track of timing information can be kept to a minimum. This usually requires a trade-off because it often takes extra time to obtain more detailed information. With the MPI wrappers for Linux, options are set via environment variables that are picked up in the wrapper for MPI_Init() or MPI_Init_thread(), and timing data is written when the application calls MPI_Finalize(). It is possible to control the region of code that is sampled … more on that later … but it is crucial for the application to eventually call MPI_Finalize(), so that you can get the timing data that was collected.

The MPI entry points can be used to make other kinds of tools such as hardware counters and traditional Unix-based profilers more accessible and/or control-able in a parallel environment. For example, when profiling parallel applications with "vprof", it is convenient to use the MPI entry points to limit profiler output to a few selected MPI ranks … one does not normally want to get profiler output from each of say 200000 MPI ranks. The MPI profiling entry points make it convenient to collect a considerable amount of performance data from large-scale parallel applications, and to control output to a few selected MPI ranks.

Features.

The MPI wrappers for Linux have a number of optional features that can be controlled by setting environment variables. The default behavior is to simply collect a summary indicating the total amount of time spent in MPI routines, and cumulative data such as the total number of calls for each MPI function and a crude histogram of message-sizes (for routines where that applies). This data is intended to give you a rough picture of the breakdown of time in the application: computation vs. communication. However, it is important to keep in mind that the timing data includes all of the time spent in MPI routines, not just the time required to move data over the network. Time in spent in MPI is frequently "wait" time, where one process has finished it's computational phase, and is waiting in an MPI routine to receive data from another process. It is expected that the default options for the MPI wrappers would be sufficient for most uses, but one can obtain additional data at the expense of some extra overhead. For example, an application might spend a large amount of time in MPI _Wait() … but there may be many such calls … so how do you find out which calls are performance-sensitive? You can set an environment variable, PROFILE_BY_CALL_SITE=yes. That will make the MPI wrappers obtain the call-stack for every MPI function call. Then, when the program calls MPI_Finalize(), you get a breakdown of time per call-site in the code. That clearly adds overhead, but it can be very useful for identifying performance-critical sections of the code. The call-stack consists of instruction-addresses, and so you will need to use the -g option when you compile and link your code, in order to properly associate source-files and line-numbers with the instruction addresses.

An example of a call-site section in the mpi_profile text files is shown below:

```
-------------------------------------------------------------
Profile by call site, traceback level = 0
-------------------------------------------------------------
Use addr2line to map the address to source file and line number.
Ensure -g is used for compilation and linking.
-------------------------------------------------------------

communication time = 66.358 sec, call site address = 0x010f625c
   MPI Routine                  #calls        time(sec)
   MPI_Waitall                 2203906          66.358

communication time = 25.804 sec, call site address = 0x010f6974
   MPI Routine                  #calls        time(sec)
   MPI_Allreduce                226151          25.804

communication time = 18.498 sec, call site address = 0x010f619c
   MPI Routine                  #calls        time(sec)
   MPI_Wait                    1180478          18.498
  ...
```

The “communication time” is just the total elapsed time spent in the MPI routine(s) that are associated with the given call-site address. To locate the source-file and line number for one of these routines, you can use the addr2line utility … check the man page for addr2line for details. An example would be :

addr2line -e your.exe 0x010f625c

The -g option is needed to allow translation from instruction address to source-file and line number. Instead of translating addresses one at a time, you can translate them all in one shot:

grep “site address” mpi_profile.#.rank | awk '{print $10}' | addr2line -e your.exe

If you want to tag the elapsed time with a call-site higher up the call chain, then you have to set TRACEBACK_LEVEL to an appropriate value when you run the application. This may be necessary when the application has its own messaging layer, where MPI calls are limited to that layer instead of appearing directly in the application code.

For benchmarking purposes, it is sometimes necessary to make performance projections. It is useful to separate the total elapsed time into categories related to computation, communication, and I/O. One common problem is that much of the time spent in MPI routines is related to waiting for other tasks to finish their work, as opposed to the actual time that it takes to transmit messages. This synchronization time tends to pile up in blocking calls, including collective-communication routines. If you want to sort out how much time is spent waiting for process synchronization, vs. the actual communication component for a specific collective routine, you can add a barrier before the call to the collective routine, and the synchronization time will shift to the barrier. You can add a barrier before every MPI collective routine by setting the env variable COLLECTIVE_BARRIER=yes. This is only a partial solution to the general problem because process synchronization is also a factor for point-to-point communication such as MPI_Wait and MPI_Waitall, where there is no simple method to separate the wait time caused by process synchronization. Setting COLLECTIVE_BARRIER=yes adds a loop with MPI_Iprobe in front of MPI_Recv in order to measure the time spent waiting for the message to be sent. This can change the behavior of MPI_Recv because it amounts to actively polling for an incoming message. The COLLECTIVE_BARRIER env variable can also be used more selectively: you can set it to a comma-separated list of routines that you specifically want to place a barrier in front

of … for example COLLECTIVE_BARRIER=MPI_Bcast,MPI_Allreduce will add a barrier before those two routines, but not before other collective calls.

Sometimes it is beneficial to get a time-resolved picture of the communication and computational phases. This “event-tracing” capability is also supported, however one has to be careful to limit the total volume of trace data, to keep it manageable. Event tracing is described in a later section.

It is often very valuable to collect standard Unix-based profiling data, using either the -pg option or an equivalent method. The MPI wrappers for Linux have features to support that … see the section on application profiling by interrupt-based program sampling.

A feature has been added to track the communication pattern for point to point MPI routines. For a variety of reasons, the only operations that are tracked are point to point routines with a specific destination rank. You can enable this feature by setting TRACE_SEND_PATTERN=yes. When the wrapper for MPI_Finalize() is called, you should get "pattern" files that contain the number of bytes and the number of messages sent to each destination rank, and the processor-name of each destination rank. This feature can help determine the relative importance of communication via shared-memory vs. off-node, among other things. However, it is important to remember that collective communication is not included in the “send pattern” ... and there are other caveats. For example, persistent communication requests made via MPI_Start() or MPI_Startall() have request arguments instead of a given destination rank, hence those routines will not be included in the pattern files.

Controlling the region of code that is profiled.

With the MPI wrappers, it is always convenient to collect timing data from MPI_Init() to MPI_Finalize() … but sometimes one needs to focus on the MPI communication in a specific section of code. To do that it is necessary to instrument the code with calls to start and stop the collection of MPI summary data:

C example:

```
      MPI_Pcontrol(1)
    // summary_start();
      do_work();
    // summary_stop();
      MPI_Pcontrol(0);
```

Fortran example:

```
      call mpi_pcontrol(1)
     ! call summary_start()
      call do_work()
     ! call summary_stop()
      call mpi_pcontrol(0)
```

You can directly add calls to summary_start() and summary_stop(), but then you would have to link the executable with libmpitrace.a or a libmpitrace.so or equivalent. The use of MPI_Pcontrol() makes it possible to control the profiled region when using a shared-library version like libmpitrace.so, without explicitly linking with a wrapper library. Those two approaches do the same thing. The first time that summary_start() is called either directly or via MPI_Pcontrol(), it zeroes out any data collected up to that point in the code, A call to summary_stop() temporarily stops the collection of MPI timing data.

The start/stop calls can be inside a loop … one will get the aggregate timing data for the code block(s) that are bracketed by the start/stop calls. A more flexible method to associate MPI time by code-block is described later. Please see the table in the Appendix for a list of MPI_Pcontrol() options.

Sometimes one might want to collect MPI data over some time window, without necessarily instrumenting the code. This time window profiling approach is supported. You have to specify a starting time and a stopping time, for example setting these variables

```
export PROFILE_BEGIN_TIME=100
export PROFILE_END_TIME=120
```

will start profiling ~100 seconds into the job (relative to the time that MPI_Init was called), and stop data collection at ~120 seconds into the job. In order for this to work with good accuracy, it is best if the application makes fairly frequent calls to MPI on each rank. If you choose to do profiling over a specific time window, you will also need to control the ranks that will generate output files, by setting the environment variables SAVE_LIST or SAVE_ALL_TASKS, as described below.

Controlling Output.

The MPI wrappers produce plain text output that contains cumulative performance data. The default is to save specific data from MPI rank 0, and the ranks that had the minimum, median, and maximum times in MPI. That way one can get a pretty good idea about most applications, without generating a large number of output files. The MPI data is in files with names:

```
mpi_profile.#.rank
```

where # is a unique number for each job, such as a job number. The file for MPI rank 0 is special … it contains a summary of data from all other MPI ranks. If you really want to save a separate output file for every MPI rank, you can set an env variable:

```
export SAVE_ALL_TASKS=yes.
```

You can also save data from a specific list of MPI ranks by setting a different env variable like this:

```
export SAVE_LIST=0,2,4,6,8,10
```

which in this example will result in output from MPI ranks (in the MPI_COMM_WORLD communicator) of 0, 2, 4, 6, 8, 10. These output methods apply to other types of output, including hardware-counter output, and interrupt-based profiler outputs. All outputs are written in the wrapper for MPI_Finalize(), and so it is crucial for the application to call MPI_Finalize(). An exception to that rule applies when profiling for a specified time-window. In that case, outputs are generated at the end of the specified time-window, and the application will continue to run.

The MPI rank that spent the least amount of time in MPI is of particular interest because that rank has often done the most work, and other MPI ranks must wait for that one before they can continue. There tends to be an inverse correlation between time spent in MPI vs. time spent doing computation, and so the rank with the minimum time in MPI is a good candidate for interrupt-based program sampling.

Normally output will be written in the working directory for the application. Sometimes, applications use temporary working directories that are deleted upon job completion. In a case like that you probably want to send the profiling output to some other directory. You can do that by setting an env variable:

```
export TRACE_DIR=/path/to/your/profile/files
```

and then the mpi_profile.#.rank files should be written in the TRACE_DIR directory upon completion of MPI_Finalize().

Obtaining the memory footprint.

Recent Linux distributions provide memory utilization via the getrusage() routine. The current MPI wrappers for Linux use getrusage() to obtain the maximum "resident set size" for the application. This should be a good indicator of memory utilization by the application.

Profiling by communicator size.

For collective-communication routines and for other purposes, it is useful to get information about the communicators used for MPI calls. For example, many applications have Cartesian communicators and do collective operations on sub-communicators, not on MPI_COMM_WORLD. An alltoall operation on a row or column communicator will behave differently from an alltoall operation on MPI_COMM_WORLD. You can get some information specific to communicators by setting env variable PROFILE_COMMUNICATORS=yes. That turns on an option to separate messaging time according to communicator size. Note, however, that many important MPI routines such as MPI_Wait(), MPI_Waitall(), etc., have no communicator argument … and so the time spent in those routines can't easily be sorted into communicator buckets. Also, the book-keeping is done based on communicator size, so different communicators with the same size will be lumped into the same bucket.

Collecting data in separate communication contexts.

In some cases you might want to collect MPI data separately for several different code sections within the same job. There is a "ctx" (for context) directory with library versions that support this feature. The collection of MPI data per named code-block is not in the default wrappers because it adds significant complexity and requires additional memory … but it can be very useful in some cases. The MPI communication contexts should be disjoint, as indicated below :

```
C/C++ :  Context_start("phase1");
         do_phase1();
         Context_stop("phase1");
         …
         Context_start("phase2");
         do_phase2();
         Context_stop("phase2");
         …
```

```
Fortran :  call context_start('phase1')
           call do_phase1()
           call context_stop('phase1')
           …
           call context_start('phase2')
           call do_phase2()
           call context_stop('phase2')
```

You must instrument your code as illustrated above, using code-block labels that match in the start/stop calls. Linking with a library such as libmpitrace.so in the "ctx" directory is required. The output will show MPI timing data separately for each labeled context. In addition, there is a default context, which is inclusive and normally starts in MPI_Init() and stops in MPI_Finalize(), but can be controlled via the usual methods (summary_start/summary_stop). That way you can get a picture of the aggregate MPI data in addition to a separate timing section for each named code-block.

Interrupt-based Program Sampling.

Interrupt-based program sampling is popular for a good reason: it can provide very useful insight into the computational aspects of an application. The most common method is to use the -pg option, preferably along with -g, and analyze output using gprof. You have choices: you can compile your code with the options -g -pg and link with -g -pg, and you will get call-graph data along with function-level profiling data. That imposes a significant amount of overhead per function-call, because when you add -pg as a **compiler** option, the compiler inserts a call to a routine that tracks the call-stack and the number of calls, etc., for every compiled function. It is often preferable to add -g as a compiler option (not -pg), and then specify both -g and -pg when you **link**. That way you get all of the function-level (and statement-level) profiling data, without the overhead associated with collecting call-graph information. This second method uses only interrupts at 100 times-per-second to check the position of the program counter, or instruction address. The basic profile data is then a histogram showing how many interrupts occurred for each instruction address in the program text section of the executable file. Sampling at 100 interrupts per second is quite coarse given that instructions are zipping through the cores at rates of ~10^9 instructions per second … so keep that in mind when interpreting profile data. Profiling with -pg samples only the range of instruction addresses in the program text section of the executable … the time spent in shared libraries will not be included. Instead of the -pg method, one can use hardware counters to trigger the interrupts, and that approach is far more powerful, as described later.

You can control the region of code that is profiled with -pg using the moncontrol() routine, if your Linux distribution supports it. In C, it would look like this:

```
int main(int argc, char * arg[])
{
  moncontrol(0); // turn off profiling
  initialization_code();
 …
  moncontrol(1);  // turn on profiling
  do_work();
  moncontrol(0);  // turn off profiling
 ...
}
```

For Fortran applications you need to let the compiler know that the moncontrol() routine uses an argument passed by value, not by reference. The method to do that varies with compiler; an alternative would be to wrap the C moncontrol() routine and use your wrapper for it in Fortran.

Most Linux distributions do not provide a "mondisable()" routine to turn off profiler output, so you will ordinarily get gmon.out files from all MPI ranks. The MPI runtime might add the MPI rank to the name of the gmon.out file, or it might use "gmon.out" as the name of the profiler output-file for all of the ranks. In the latter case, it would be best to save profiler output in a separate directory for each MPI rank that you really want to keep. You can enable that feature by setting PG_PROFILE=yes. When that is set, the same output filter applies for MPI profile output and gmon.out files; you will get a separate directory, pgdir.rank, for each rank that was selected, while all the other ranks will write "gmon.out" (which you should delete).

An alternative profiling method that uses regular timer interrupts is provided by the profil() routine, which is user-callable and is included in GNU libc.a. The profil() routine uses the same interrupt mechanism as -pg, but since it is user callable, you can control it more finely. With profil(), there is no call-graph capability. The basic data that is collected is a histogram with the number of interrupts recorded for each instruction address in the program text section of the executable file. By using the same underlying mechanism as "addr2line", one can identify the routine, and the source file and line-number for each instruction address. The MPI wrappers for Linux have built-in support for profiling with the profil() routine. If you want to start profiling in MPI_Init() and stop profiling in MPI_Finalize() using the profil() routine, it is not necessary to add calls to your code. You can simply set an env variable:

```
export VPROF_PROFILE=yes
```

and you should get vmon.out files written during the wrapper for MPI_Finalize(). Setting that env variable effectively calls vprof_start() in the wrapper for MPI_Init() and vprof_stop() in the wrapper for MPI_Finalize(). Those routines basically map to the profil() routine.

Instead of setting VPROF_PROFILE=yes, you can add calls to start/stop profiling for a specific code block. The format for controlling that with the MPI wrappers is:

C example:

```
vprof_start();   // or MPI_Pcontrol(11);  // start profiling
do_work();
vprof_stop();   // or MPI_Pcontrol(10);  // stop profiling
```

Fortran example:

```
call vprof_start()  ! or call mpi_pcontrol(11)  ! start profiling
call do_work()
call vprof_stop()   ! or call mpi_pcontrol(10)  ! stop profiling
```

If you add explicit calls to vprof_start()/vprof_stop() calls to your code, you will need to link with libmpitrace.a or libmpitrace.so or equivalent. You can get the same functionality at run time using the shared library libmpitrace.so and the MPI_Pcontrol() mechanism.

The wrapper for MPI_Finalize() will write the profile data that has been collected in vmon.out format.

You use the bfdprof utility to analyze that output. Typical use would be:

bfdprof your.exe vmon.out.n > profile.n.txt

The output, profile.n.txt, is a plain text file that has the function-level profile, equivalent to the flat profile from gprof, and a listing by source file, followed by annotated text showing the number of "hits" for each source line. The annotated source-code sections make this tool considerably more useful than gprof. The "vmon.out" file-format follows the conventions of an earlier tool, cprof/vprof from Sandia National Labs. The vmon.out files are very compact because they include only those addresses that got profile "hits". In contrast, gmon.out files produced by -pg contain an entry for every instruction-address in the program text section, whether that address has "hits" or not. With a simple addition to "gprof" included in GNU binutils, one can convert gmon.out files to vmon.out format … so you can benefit from the statement-level annotation capability of bfdprof, using histogram data contained in a gmon.out file.

A simple example of a function-level profile using bfdprof is shown below:

```
#########################
Function-level profile:
#########################
    tics   function-name
-------------------------
    3532  swforce_
     120  world_
     103  neighbor_
      91  md
```

The profil() routine takes 100 samples/sec, so you can directly relate the number of "tics" to time in seconds. In the example above, the "swforce" routine took about 35.3 seconds. After the function-level profile, there is a section listing tics (profile hits) by source file. For this small example, it looks like this :

```
#########################
Source-file profile:
#########################
    tics   source-file
-------------------------
    3532  /home/user/codes/md/serial/swforce.f
     223  /home/user/codes/md/serial/neighbor.f
      91  /home/user/codes/md/serial/md.f
```

After that, each source file that the tool can find will be annotated with profile hits (tics) associated with the source code. An example is shown below:

```
#########################
Annotated source for file: /home/user/codes/md/serial/swforce.f
#########################
  tics | source
...
       |    !-------------------------------------------------------
       |    ! Do the pair-force and save values for the neighbors of i
       |    !-------------------------------------------------------
       |         npairs = 0
    42 |         do nb = 1, nbnumber(i)
       |
    49 |            j = nbtable(nb, i)
       |
    51 |            xji = pos(1,j) - xi
    29 |            yji = pos(2,j) - yi
    39 |            zji = pos(3,j) - zi
       |
   139 |            rjisq = xji*xji + yji*yji + zji*zji
       |
   137 |            if (rjisq .lt. aswsq) then
       |
   282 |               rji = dsqrt(rjisq)
```

One should keep in mind that the optimizer mixes instructions over a range of source statements, and that interrupt-based program sampling is quite coarse grained. So one should use judgment when interpreting profiling data at the source line level.

One limitation of the profil() profiling method is that it needs the starting and ending addresses for program text in the executable. In this version of the MPI wrappers, these are obtained using methods from the “bfd” (binary file descriptor) library. You need the GNU binutils development files (in particular bfd.h and libbfd.a) to build the “vprof” tools. Since calls to “bfd” routines are built in, you need to link your application with -lbfd, if you are linking with static wrapper libraries. If your system doesn't have the GNU binutils development files, you can either build them yourself, or ask an administrator to install them. To build them from source, follow this sequence of steps, which takes just a few minutes:

wget ftp://ftp.gnu.org/pub/gnu/binutils/binutils-2.27.tar.gz (or latest version)
tar zxf binutils-2.27.tar.gz
cd binutils-2.27
./configure –prefix=$HOME/gnu (or directory of your choice)
make
edit libiberty/Makefile ; set target_header_dir = ${prefix}/include
make install
copy bfd/config.h to your install/include directory

Because of the extra complexity associated with building vprof support, you can turn it off if you just want MPI profiling capability. To do that, there is a VPROF C-preprocessor setting that you can disable/enable in the makefile for the MPI wrappers.

One can alternatively use hardware-counters to generate interrupts for program-sampling. This is the most flexible method because you can select both the hardware-counter event and the sampling rate. In addition, it is possible to add support for profiling shared-libraries by making use of Linux-specific routines such as dladdr() in the interrupt handler. This is the approach used in libhpmprof.so and variants. The default with libhpmprof.so is to generate interrupts on the master thread every N cycles in the run queue, where the sampling rate amounts to ~100 samples per cpu-second. When you preload this library, sampling begins in MPI_Init() and stops in MPI_Finalize(). With the HPM program-sampling method, you can control the sampled region like this :

```
C/C++ :   HPM_Prof_start();   // or MPI_Pcontrol(21);
           do_work;
          HPM_Prof_stop();   // or MPI_Pcontrol(20);

Fortran :  call hpm_prof_start()   ! or call mpi_pcontrol(21)
           call do_work()
           call hpm_prof_stop()   ! or call mpi_pcontrol(20)
```

Link the application with libhpmprof.so or pre-load that library, and run the code. You should get output files “hpm_histogram.jobid.rank”, which can be analyzed with the “bfdprof” utility :

```
bfdprof  your.exe   hpm_histogram.jobid.rank > profile.jobid.rank
```

If you want to specify a different counter event and/or a different sampling rate, you can set these env variables:

```
HPM_PROFILE_EVENT=”your_event”
HPM_PROFILE_THRESHOLD=value
```

An interrupt will be generated each time the hardware counter increments by the threshold value. Sampling rates of order 100 samples per second are normally adequate. PAPI uses a thread-specific counter context, so interrupts will be generated by the thread or threads that called the PAPI initialization routines. The current default in libhpmprof.so is that only the master thread for each MPI rank sets up hardware counters for program-sampling. This can be extended as needed. With the standard interrupt handler, you get statement-level data for the program text section of your executable file. If you want to get statement-level data for code residing in a shared library, just set:

```
PROFILE_SHAREDLIB=/path/to/your/sharedlib.so
```

and when you analyze the data with bfdprof, you must specify the shared-library instead of the executable file as the first argument to bfdprof.

Event-tracing.

The idea for event tracing is to obtain insight into the time-dependent nature of messaging between various MPI processes. It is often possible to visually spot problems such as load imbalance, or coding issues that can result in effective serialization, or less than ideal parallelization in the code. The most common display is an x-y plot with time as the x-axis and MPI rank as the y-axis, using colored rectangles to represent each MPI event. There have been many similar tools in use for MPI applications for many years, such as jumpshot and vampir. A key problem has been that it is really

easy to generate an unwieldy amount of data. The approach taken here is to be as selective as possible about event tracing, in order to keep the data volume manageable, and then use a lightweight viewer that can relate each MPI event back to source code. To activate event-tracing in your code, the preferred mechanism is to insert calls:

C example:

```
trace_start();  // or MPI_Pcontrol(101); // start event-tracing
do_work();
trace_stop();  // or MPI_Pcontrol(100);   // stop tracing
```

Fortran example:

```
call trace_start()  ! or call mpi_pcontrol(101)  ! start tracing
call do_work()
call trace_stop()  ! or call mpi_pcontrol(100)  ! stop tracing
```

If you have an application that makes regular time-steps or iterations of some kind, it is usually sufficient to trace a few iterations or time-steps, because the pattern should repeat. Also, in most MPI applications, many of the MPI ranks are ostensibly doing the same kind of thing … so it is possible to get some insight into the time-dependent behavior by looking at a subset of MPI processes.

Instead of instrumenting your code with trace_start()/trace_stop() calls, you can tell the MPI wrappers to start event tracing in the wrapper for MPI_Init(), by setting an environment variable:

```
export TRACE_ALL_EVENTS=yes
```

When event tracing is enabled, each call to an MPI function takes 48 bytes to record it, and a small buffer is reserved in memory to hold the event records on each MPI rank. The default buffer size is enough to hold records for 50000 MPI calls, which takes 2.4*10^6 bytes of memory. Once the trace buffer is full, additional event records are discarded … so you can get a maximum of 50000 events saved in the trace buffer on each rank. If that is not sufficient, you can set the buffer size at run-time with an environment variable:

```
export TRACE_BUFFER_SIZE=4800000  (for example)
```

where the value is in bytes … the example above would be sufficient for 10^5 MPI calls per rank. It is best if the total volume of trace data can be kept to less than a few hundred MB, otherwise trace visualization will be unmanageable, so keep that limitation in mind when setting the buffer size. If the trace-buffer overflows, you will get a warning message when the application calls MPI_Finalize(). The trace-buffer limitation applies both to selective tracing using trace_start()/trace_stop() and to tracing that starts in MPI_Init() via the TRACE_ALL_EVENTS environment variable.

Sometimes you may need to trace for a reasonable time window, and it may not be feasible to instrument the code with calls to trace_start()/trace_stop(). You can specify a rough time-window that will be used for trace-data collection. For example, if you want to start tracing 100 seconds after MPI_Init() and stop tracing 120 seconds into the job, you can set env variables like this:

```
export TRACE_BEGIN_TIME=100
export TRACE_END_TIME=120
```

The begin/end times should be set to an integer number of seconds. This feature should work provided each MPI rank makes fairly frequent calls to MPI routines.

Some applications make millions of calls to routines such as MPI_Iprobe(), which may be called in a loop, waiting for a message to come in. Such frequently-called routines can quickly overflow any reasonable-sized trace buffer, so it may be necessary to disable event-tracing for such MPI routines. You can do this by setting an environment variable:

export TRACE_DISABLE_LIST=MPI_Iprobe,MPI_Comm_rank (for example).

Use a comma-separated list of MPI routines to exclude them from the event tracing … those routines will still be included in the overall timing summary files, mpi_profile.#.rank.

Since the graphical display is an x-y plot with MPI rank as the y-axis, it is most convenient to display data for a band of MPI ranks with a given min and max rank. Experience has been that it is often sufficient to examine ranks 0-127, so this is set as the default range. You can set an environment variable TRACE_MAX_RANK=N, which sets the largest rank to N, that is you get trace data for ranks 0 through N saved in the aggregate trace file, written when the application calls MPI_Finalize(). By default, event tracing starts with rank 0, but you can optionally specify TRACE_MIN_RANK. If you really want to save events from all MPI ranks, you could set TRACE_ALL_TASKS=yes, but that would be asking for trouble (too much data) for very large-scale parallel jobs.

The output from event-tracing is a binary file “events.trc”, which contains the concatenated records (48 bytes each) for all of the events saved, ordered by rank in MPI_COMM_WORLD. There is a simple trace visualization tool, traceview, for display of this data. The traceview utility is written using OpenGL, and it is intended to be used locally on your laptop or workstation, because graphics-intensive applications work best that way. It is also possible to use a version built for a Linux front-end, and use X-windows to display the data. That would require an X-server on your local display that supports OpenGL extensions … but it is recommended to use a local copy of traceview. This utility uses “glut” and “glui” software layers which are broadly available. The traceview utility has been built on Windows, cygwin, Linux-x86, Linux-on-power, AIX, and Apple OS X.

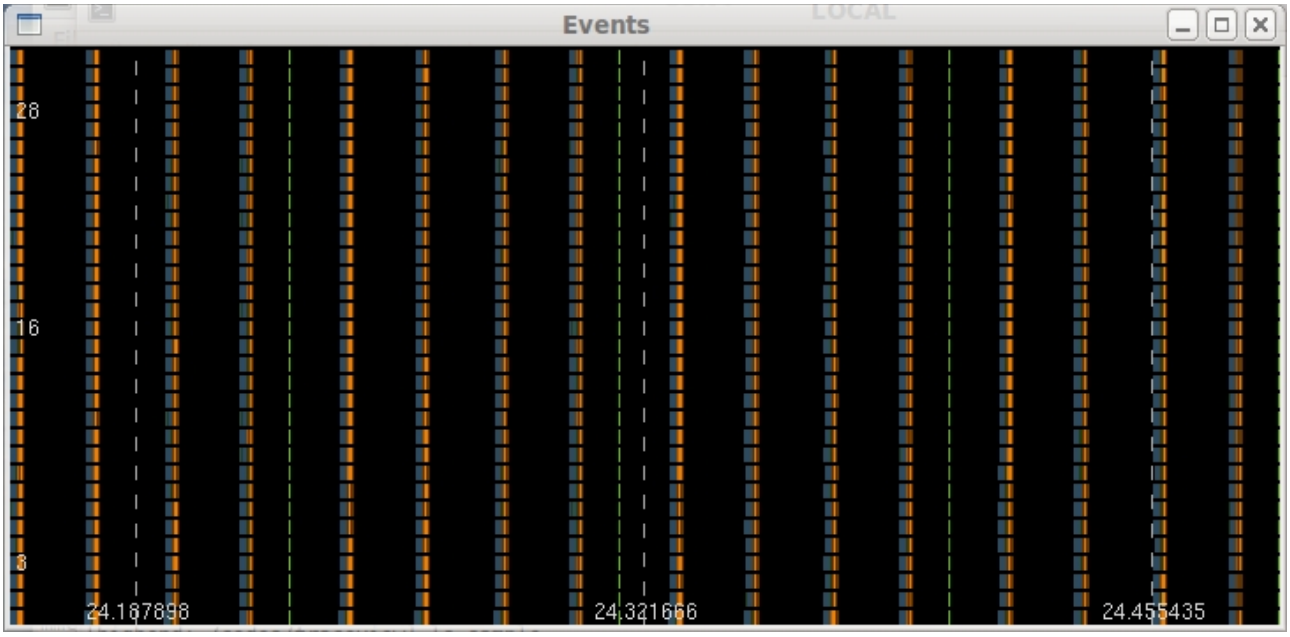
Typical use of traceview would be: traceview events.trc. There is a help button which describes most of the things that you need to know, including the key assignments for controlling the viewing region. It is highly recommended to learn and use the hot-keys to navigate around in the trace data. Some example screen shots are included here.

In the display above, the x-axis is elapsed time in the job, and the y-axis is MPI rank from 0-31, and data is from a 2048-way parallel job using the MILC NSF benchmark. MILC uses a conjugate-gradient solver, and the code was instrumented with calls to trace_start()/trace_stop() in order to capture a few iterations. The colored bars correspond to MPI routines : orange is MPI_Wait, green is MPI_Allreduce. This data was recorded on BlueGene/P during an early stage of system-software development. The relatively long times that are sometimes spent in MPI_Wait() were due to errors in the messaging layers for BG/P … the messaging software was not properly handling multiple outstanding non-blocking calls to MPI_Isend()/MPI_Irecv(). In this example, the code was attempting to overlap computation and communication by using a sequence of steps like this:

call MPI_Isend/MPI_Irecv for the first set of messages
do some work
call MPI_Isend/MPI_Irecv for a second set of messages
call MPI_Wait for the first set of messages
do more work
call MPI_Wait for the second set of messages

Unfortunately, some of the MPI_Wait() calls for the first set of messages did not return until some of the MPI_Wait() calls for the second set of messages completed. The result was poor parallel performance … some MPI ranks were stuck waiting for others to finish computational steps before they could proceed. In this case the defect was reported and got fixed, but it was possible to re-structure the code to avoid the problem:

do some work
call MPI_Isend/MPI_Irecv/MPI_Wait for the first set of messages
do more work
call MPI_Isend/MPI_Irecv/MPI_Wait for the second set of messages



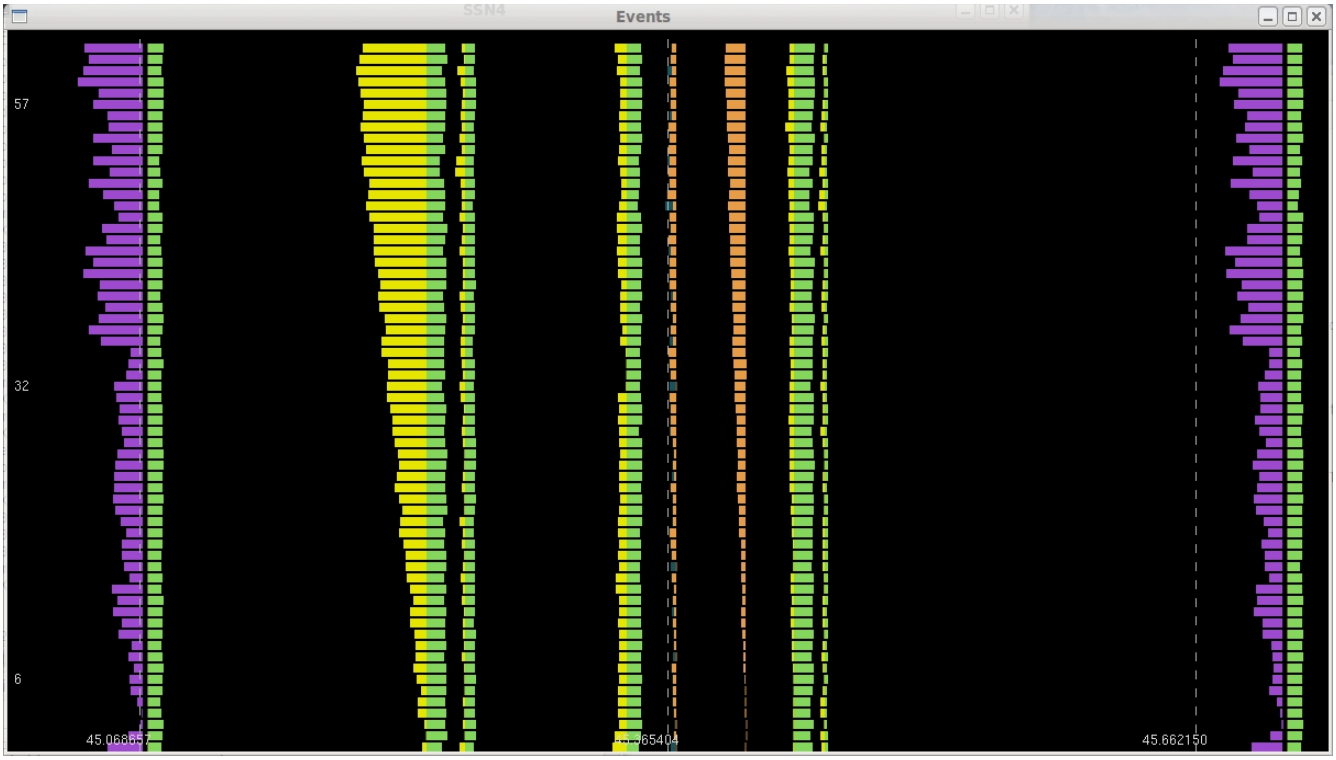
This second approach made no attempt to overlap computation and communication, but it avoided the problem with additional wait time. A display for the modified code is shown above. Black corresponds to computation, and one can see that now all of the MPI ranks are in sync, and the MPI events take a very small fraction of the elapsed time. The image above is a good example of a well-behaved parallel application … all of the MPI ranks are busy working concurrently, the load is beautifully balanced … there are no ranks unduly waiting on others to finish some stage of the work, and the fraction of time spent in MPI is small.

A key feature of the trace viewer is the ability to map MPI events back to the source-code location. Each trace record includes the instruction-address for the MPI routine and the grandparent, going up the call-stack. When you click (left-mouse button) on an MPI event, the details for that event will be displayed, and then you can use the addr2line utility to translate from instruction address to source-file and line-number. This requires -g as one of your options for compilation and linking. Example output from clicking on an MPI event is shown here:

```
task id = 62, event = MPI_Barrier
  tbeg = 45.193914, tend = 45.229736, duration = 35.823 msec
  parent address = 0x0115648c
  grandparent address = 0x01139150
```

In this example, the MPI routine was MPI_Barrier, and you can find the source-file and line-number from the instruction addresses, using the addr2line utility. The event records include destination-ranks for flavors of MPI_Send, source ranks for flavors of MPI_Recv, and message-sizes where that information is available. The key information, however, is the instruction-address.

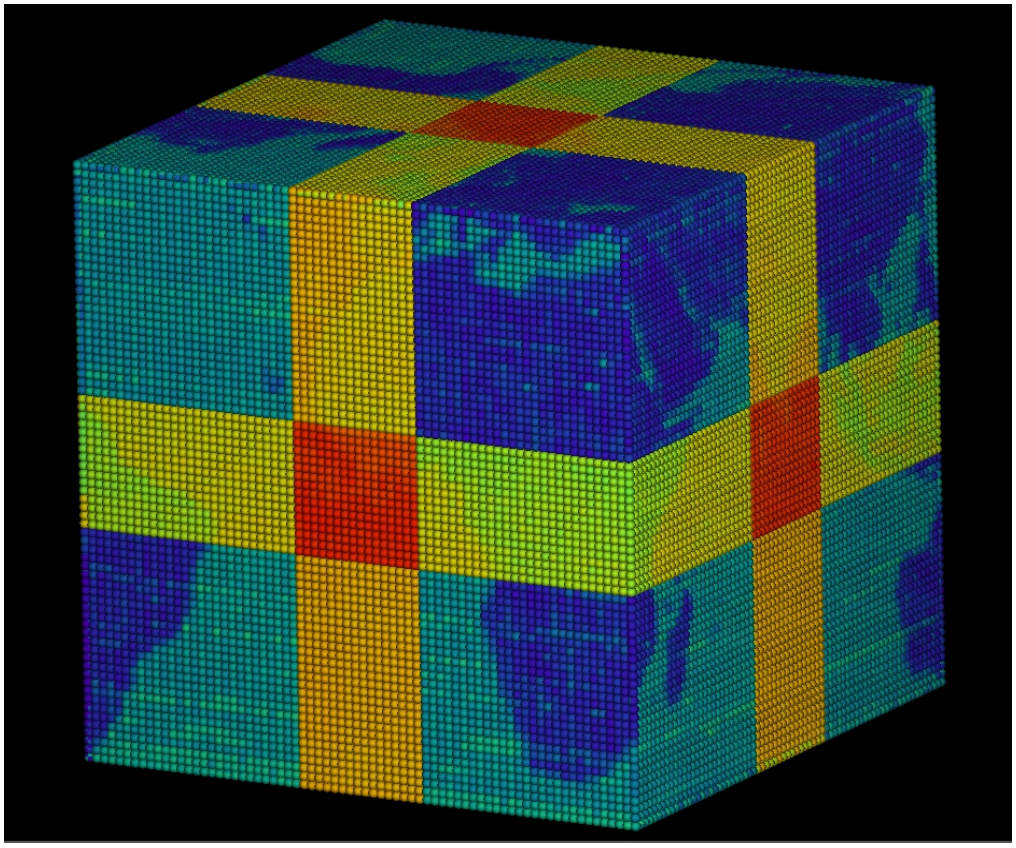
An example of an application with inherent load imbalance is shown below:



In this example the time spent in MPI_Barrier (the yellow rectangles) increases roughly linearly with MPI rank, and a similar pattern occurs for MPI_Bcast (light orange rectangles). In this application (the GFS code from the US National Weather Service), the load imbalance arises naturally from the parallel decomposition strategy. At this scale, load imbalance, not network latency or bandwidth, is what limits the parallel efficiency.

Sometimes applications have their own messaging layers … for example MPI_Isend() might always be called from an application routine “my_send()”. In cases like that you may need to look at the grandparent address, or even deeper into the call-stack for deeply layered cases. You can save instruction address data starting at any point in the call stack, by setting the environment variable TRACEBACK_LEVEL to the appropriate value … but you have to set that when you record the trace data because each event record has space for just two entries (called parent and grandparent) for instruction addresses.

In some cases it would be better to display performance metrics in a format that reflects the physical problem for the simulation. One example is shown below, from a cubed-sphere model of the earth's atmosphere:



The data in this figure is the total amount of time spent in MPI routines, for each of 31104 MPI ranks, from a BlueGene/P job. Dark blue corresponds to the smallest time spent in MPI, and red corresponds to the largest time spent in MPI. The data looks like a gift-wrapped planet earth, and the timing variations are all due to load imbalance. The code uses a 2D decomposition for each of the six faces of the cube. In this case there were 72x72 MPI ranks for each face, and a total of 2000 grid points in each of the two dimensions. The yellow stripe occurs because the number of grid points (2000) is not evenly divisible by 72 … the MPI ranks in the yellow stripe have one less grid point. As a result, they finish their computation sooner, and wait in MPI longer. The MPI ranks in the red square have one fewer grid point in each of two dimensions, so they do the least amount of work, and wait in MPI the longest. Finally, MPI ranks that are positioned on a land mass have some extra work to do, relative to ranks positioned over the ocean … so ranks over the ocean must wait for the ones over land to finish their extra work … ranks over land have the most work, and do the smallest amount of waiting in MPI.

It would be possible to use similar kinds of displays for time-dependent data, such as event tracing, but this kind of approach is clearly dependent on the nature of the simulation, and so there is not much in the way of general tools that map performance data back to the physical simulation domain.

Appendix A

Example of env variable setting.	What it does .
COLLECTIVE_BARRIER=yes	Set to add barriers before every collective call and report synchronization time; default = no. Can also be more specific, using a comma-separated list of MPI collective routines.
PROFILE_BY_CALL_SITE=yes	Set to assign time spent in MPI on a per-call-site basis. Used to identify the source file location for expensive MPI calls.
PROFILE_COMMUNICATORS=yes	Set if you want to see MPI timing data sorted by communicator size. Useful for understanding collective-communication.
SAVE_ALL_TASKS=yes	Set if you want a summary file from every MPI rank; default = no.
SAVE_LIST=2,4,6,8	Set to a specific list of MPI ranks that will produce output files; default is to automatically select output from certain ranks.
SUMMARIZE_ALL_TASKS=yes	Set if you want a one-line summary from each rank printed in the output for MPI rank 0; default = no.
VPROF_PROFILE=yes	Set to enable program-sampling via timer interrupts starting in MPI_Init; default = no.
TRACE_ALL_EVENTS=yes	Enables event tracing starting in MPI_Init; default = no.
TRACE_ALL_TASKS=yes	Set to collect event records from every MPI rank. This may result in very large trace files; default is to save data for ranks 0-255.
TRACE_BUFFER_SIZE =#bytes	Sets the buffer size used to hold trace records, default = 2400000 bytes.
TRACE_DISABLE_LIST=MPI_Iprobe	Set to a list of MPI functions that you want to exclude from event tracing for any reason; default none.
TRACEBACK_ERRORS=yes	Set to enable an error-handler that provides the call-stack when an MPI function fails; default = no.
TRACEBACK_LEVEL=nsteps	Set to save the instruction address that is nsteps up the call stack. Useful if the MPI routine is called from wrappers; default = 0;
TRACE_MAX_RANK=number	Set to collect event records for all ranks =< “number”; default 127.
TRACE_SEND_PATTERN=yes	Set to obtain information about point-to-point message traffic; default = no.

PROFILE_BEGIN_TIME=100 Starts time-window profiling 100 seconds after job start.
PROFILE_END TIME=120 Stops time-window profiling 120 seconds after job start.

TRACE_BEGIN_TIME=100 Starts MPI event tracing 100 seconds after job start.
TRACE_END_TIME=120 Stops MPI event tracing 120 seconds after job start.

PROFILE_SHAREDLIB=/path/to/your/sharedlib.so Used by the hpmprof library to specify a shared library for program sampling.

MPI_Pcontrol(argument)	function
1	summary_start()
0	summary_stop()
11	vprof_start()
10	vprof_stop()
21	hpm_prof_start()
20	hpm_prof_stop()
101	trace_start()
100	trace_stop()