MPIProf – A Tool for MPI Function and I/O Profiling

Version 2.15

Henry Jin

NASA Advanced Supercomputing (NAS) Division, NASA Ames Research Center

<hjin@nas.nasa.gov>

Contents

List of Figures

List of Tables

1. Overview

MPIProf is a tool that reports profile information of MPI functions called by an application. The statistics is gathered in a counting mode via the PMPI profiling interface. The tool also reports I/O statistics, memory and power used by processes on each node. There are two ways to obtain the profiling information:

- via the `mpiprof` profiling tool (see Section 2),
- via the `mprof` API routines (see Section 4).

The first method does not require change or recompilation of an application; the second method involves instrumentation of an application. A set of environment variables can be used to control the type of profiling (see Section 3). This document also includes an explanation of sample profiling results for an application in Section 5 and a study of accuracy and overhead of the tool in Section 6. A list of instrumented MPI and I/O functions is given in Appendix A. The installation of MPIProf is discussed in Appendix B. MPIProf has been designed to work transparently with different versions of MPI and library implementations (see Appendix C). A brief version history of development is given in Appendix D.

2. The `mpiprof` Profiling Tool

Use of the `mpiprof` profiling tool is the most straightforward way to perform MPI profiling of an application. The tool loads the proper profiling environment, including libraries, and writes results at the end of a run. There is no need to recompile the application. The tool accepts options to control profiling itself and its output. The basic usage of `mpiprof` is as follows:

```
% mpiexec -np <n> mpiprof [-options] [-h|-help] a.out [args]
```

or as a standalone tool for serial (non-MPI) programs:

```
% mpiprof [-options] [-h|-help] a.out [args]
```

For serial programs, the tool reports only I/O statistics, memory and power usage. Besides [`-h`] and [`-help`] for getting a quick help, all standard `mpiprof` options (other than `mbind` options) have corresponding environment variables to support the intended functionality as described in Section 3. The difference is that command options override those defined by environment variables. Available `mpiprof` options are listed in Table 1 and `mbind` options for process binding are listed in Table 2.

2.1 Description of the mpiprof Options

The following describes some details of the `mpiprof` options listed in Table 1. These options are intended for overriding the default behavior.

The [`-lib`] option specifies a runtime (MPI or serial) profiling library to override with the default setup or when the default auto-detection of the runtime environment fails. The option overrides the `MPROF_LIB` environment variable. "`ser`" is a valid option for serial programs. A "`+`" or "`-`" sign in front of <*mproflib*> indicates prepending or appending (default) the shared profiling library to the `LD_PRELOAD` environment variable. This may be needed when working with other tools to enforce a particular order. Use [`-lib=+|-`] to enforce the order for the auto-detected library.

If [`-o`] or `MPROF_OUT` is not specified, the profiling results are written to "`mpiprof_stats.out`" or "`<a.out>_mpiprof_stats.out`" if the executable name is known. If <*outfile*>="`-`", the results go to <stdout>. If <*outfile*> starts with a "`+`" sign, the results will be appended to the end of the file specified after the "+" sign in <*outfile*>. If <*outfile*> starts with a "`%`" sign, the output filename will be suffixed with a number (*nnnn*) to avoid overwriting an existing file. The [`-os`] option is equivalent to specifying <*outfile*>="`-`" for [`-o`].

Table 1: Summary of the `mpiprof` options

Option	**Description**
`-lib [+|-]<`*`mproflib`*`>`	selects a runtime profiling library <*mproflib*>
`-o <`*`outfile`*`> | -os`	writes profiling results to <*outfile*> or <stdout>
`-[c|p]blk[=<`*`n`*`>]`	estimates blocking time for collective and/or point-to-point communication calls
`-msgm`	collects rank-based message size and count maps
`-cpath[=<`*`depth`*`>]`	collects call-path information
`-mpmt`	enables stats collection from multiple threads
`-mio`	reports stats from I/O calls made inside MPI I/O
`-mem | -pwr`	reports memory and power usage only
`-sum | -nsum`	reports summary without per-rank/per-node stats
`-ios`	reports I/O statistics, including MPI I/O, and memory usage only
`-ios_nompi`	reports I/O statistics without MPI functions
`-gn`	disables the report of GPU memory and power usage
`-pflag <`*`value`*`>`	adds <*value*> to `MPROF_PFLAG`
`-mfunc <`*`func:m`*`>`	specifies a function to be monitored
`-csig[=<`*`signo`*`>]`	writes output stats when a signal is caught
`-psec <`*`ival:gival`*`>`	sets power sampling interval in seconds
`-byte`	prints message size in bytes
`-smpc`	reports shared memory usage without correction
`-expr=<`*`exps`*`>`	performs cpu+comm scaling experiments (experimental)
`-debug[=<`*`rank`*`>]`	starts processes in a wait stage for a debugger
`-xio`	disables the loading of I/O profiling library
`-v[<`*`n`*`>]`	sets verbose flag

The [`-cblk`], [`-pblk`] or [`-blk`] option can be used to estimate the blocking time in collective, point-to-point, or both types of calls due to load imbalance. This measurement is not enabled by default. Additional value `<`*`n`*`>` controls the level of details. Default or `<`*`n`*`>=2` includes per-function blocking time. `<`*`n`*`>=3` disables blocking time measurement for collective I/O functions. The [`-cblk`] option can be used in conjunction with [`-mfunc`] to select specific collective functions for blocking time measurement. See `MPROF_MFUNC` in Section 3 for details.

By default, rank-based message size and count maps are not collected. Use [`-msgm`] to enable this feature. The map information will be printed on a rank-by-rank basis. To print size and count maps in a rank-rank matrix form, use option [`-msgmx`].

The [`-byte`] option forces message-size printing in bytes. The default behavior is to scale for proper unit.

The [`-pflag`] option can be used for finer control of profiling. The option adds a value to `MPROF_FLAG`. The accepted values and their meanings are described for `MPROF_PFLAG` in Section 3.

MPIProf writes profiling results at the end of a run or when the `mprof_end` routine is called. To report stats of a function at each call instance during a run for monitoring purpose, one can use option [-mfunc] or environment variable MPROF_MFUNC. The option takes up to four functions in the short-name form as listed in Appendix A. An additional value <:*m*> controls the number of instances to be monitored. Default for *m* is 1000 if no signal is specified or 0 if a signal is specified. By default, this feature is not enabled.

The profiling stats can be written out to individual files for each rank if a signal type is specified by option [-csig] or MPROF_CSIG. The stats are written when a signal <*signo*> is caught or at end of a run if no signal is caught. Up to two signals may be specified as a list in <*signo*>. If a <*signo*> value is not specified, it defaults to "2" (interrupt). A signal can arise from program execution or external tools, such as kill, sigpg, or sigpg_job.

When both [-mfunc] and [-csig] are specified, the function monitoring will be toggled by the signal if <*signo*>=1 or 2.

The [-ios] option is useful for focusing on I/O characteristics of applications from both Posix I/O and MPI I/O. Posix I/O functions called both from and outside MPI I/O functions will be reported. The [-ios_nompi] option is for suppressing any MPI functions in the report. Use the "mio" value for [-pflag] to fine control the report of Posix I/O called from MPI functions. See Section 5.6 for further details.

With the option [-cpath], inclusive time along the call path is gathered for each rank and the final results are reported as average from all ranks. To print per-rank call-path results, use the option [-cpathx]. An optional value <*depth*> can be specified to limit the stack unwinding depth. The default depth is 30. Collecting call-path information is only possible if it is enabled at the build of the MPIProf package (see Appendix B). It is recommended that the application be compiled with the [-g] option.

By default profiling of Posix I/O functions is enabled, but disabled in the call-path collection. In order to include I/O functions in the call-path report, use the option [-cpath -io] or set the environment variable MPROF_PFLAG=cpath+io.

By default, only stats from the main thread are collected in a multi-threaded environment. With the [-mpmt] option, stats from multiple threads are collected and reported as aggregated values for counts and message size, and averaged values for timing. There is a larger overhead associated with this mode. The exception is the handling of call-path data in general and I/O data in a multi-threaded environment that does not support MPI_THREAD_MULTIPLE, for which cases only relevant stats from the main thread are collected and reported.

MPIProf reports memory and power usage of the application together with profiling results. Either [-mem] or [-pwr] option is to instruct MPIProf to report memory and power usage only without profiling results. In addition, the [-pwr] option disables the report of per-rank memory information. Support for host power usage report is only possible if the underlying OS provides access to the /sys/class/powercap interface. Report of GPU memory and power usage is enabled by default for a system that contains NVIDIA GPUs. To disable power report, include the "-pwr" value for [-pflag] (see the description of MPROF_PFLAG in Section 3). The GPU report can be separately disabled by option [-gn]. Use option [-psec] or environment variable MPROF_PSEC to reset power sampling interval from its default value (*ival*=300 seconds or 5 minutes for host and *gival*=5 seconds for GPU). The power usage information is described in Section 5.2.

When reporting memory usage, MPIProf attempts to correct memory over-counting due to shared memory pages accessed by processes on the same node. Detailed description of this correction is in Section 5.1. Use option [-smpc] to disable the correction.

The [-sum] or [-nsum] option (or value sum or nsum for the [-pflag] option) reports summary information without per-rank stats. In addition, the [-nsum] option disables the report of per-node stats.

The [-debug] option will put processes in a wait stage at the startup so that a debugger (such as gdb) can attach from a separate window. By default, the process to be attached is rank 0. Use the optional value to select a different rank. From the debugger, set variable `GO`=1 or 2 to continue execution with or without profiling support. See Section 7.3 for a usage example.

The [`-xio`] option disables the loading of I/O profiling library (`libmprof_io`). This option may be useful for cases where the loading of I/O profiling library might interfere with underlying system calls. It effectively disables any I/O profiling, similar to the "`-io`" value specified for `MPROF_PFLAG`. The difference is that the latter case does not affect the loading of I/O profiling library.

CPU and communication scaling experiments may be performed via [`-expr`]. <*exps*> is a comma-separated list of "`cpu=`*factor*", "`comm=`*factor*", and "`io=`*factor*" pairs, where *factor* is a scaling factor. For the `cpu` type, *factor* greater than one indicates a faster speed; for either `comm` or `io`, *factor* less than one indicates a faster network or I/O speed. Special case of <*exps*>="`ideal`" is equivalent to "`cpu=1,comm=0,io=0`".

The verbose flag can be set via the [`-v`] option and is intended for debugging the tool or confirming process binding. This option may take an integer value larger than 1 to print more verbose information. This option also saves the process binding information in the stats output when binding is enabled (see Section 2.2).

2.2 Description of the mbind Options for Process Binding

By default, process binding is not applied. Any of the `mbind` options as listed in Table 2 triggers the action of process binding. Use option [`-v`] to confirm binding and to save the process binding information in the stats output.

Table 2: Summary of the `mpiprof` options for process binding (`mbind` options)

Option	**Description**
`-c<`*`opt`*`>`	specifies *cpulist* in list form or mnemonic form
`-x[c|s]<`*`list`*`>`	defines a set of cpus (cores | socket) to be excluded
`-s<`*`seq`*`>`	restricts cpu sequence to <*seq*>=`core|smt|<`*`nseqs`*`>`
`-t<`*`n`*`[:`*`s`*`]>`	sets the number of threads per process with an optional cpu stride
`-b<`*`n`*`>`	selects a thread binding mode (def=any)
`-n<`*`n`*`[:`*`s`*`]>`	sets the number of processes per node (def=auto)
`-r<`*`list`*`>`	defines a process rank list for a node
`-nb`	prints binding masks without applying binding
`-nb-thr`	binds MPI processes but does not spread OpenMP threads
`-nb-any`	turns off MPI process binding if any
`-nt`	does not set `OMP_NUM_THREADS` from option [`-t`]
`-mbind`	performs process binding without profiling

`-c<`*`list`*`>` ; specifies *cpulist* for process ranks

<*`list`*> is a comma-separated group of numbers with possible range (*`n1-n2`*) and stride (`:`*`s`*) specifiers, such as 0-4,6,7 or 0-7:2. A negative number counts a CPU from the end of the list in a node, such as -1 for the last CPU. The number of CPUs listed should match with the number of process ranks within a node. CPU numbers are relative within a *cpuset* if present.

`-c[p|s|c]<`*`s`*`>` ; specifies *cpulist* in a compact, spread, or cyclic form

This is an alternative way to define *cpulist* in symbolic form. The approach utilizes the auto-detection of the processor hierarchical structure and has the following forms:

`-cp`	block assignment of CPUs in a compact form
`-cs`	block assignment of CPUs in a spread form (default)
`-cc`	scatter of CPUs in different sockets (cyclic form)

The option <*s*> indicates the stride for blocking (default <*s*>=1).
<*s*> = 0 to put CPU0 to the end of the list.
<*s*> < 0 to have a list in reversed order.

`-c[node|socket|numa|cache|core]<s>` ; specifies *cpulist* in a package
The *cpulist* is defined from CPUs in a package with a given granularity of node, socket, numa, cache (last level), or core. The option <*s*> indicates the package sequence at the requested granularity; for MPI programs this is an offset for each rank. If option [`-t`] is not present, the number of OpenMP threads will be set to the number of cores (or smts if option [`-ssmt`] is defined) in each package.

`-x[c|s]<list>` ; defines a set of cpus (core | socket) to be excluded
The option defines a list of cpus to be excluded from the CPU list as defined by the `-c` option. <*list*> has a similar meaning as that in option `-c<list>`. A couple of variants:

`-xc<list>`	all CPUs within a core associated with a CPU
`-xs<list>`	all CPUs within a socket associated with a CPU

`-s<seq>` ; sets CPU sequence `<seq>=core|smt|<nseqs>`
The option defines CPU sequence type for `-c[p|s|c]` or `-c<package>`. A value of "`core`" indicates processor cores; a value of "`smt`" indicates SMTs if available; a number specifies the number of SMT sequences per core, 0 (or "`score`") indicating all SMTs in a core sequence. The default is the core sequence.

`-t<n[:s]>` ; sets number of threads per process
<*n*> defines the number of threads per process for an OpenMP or hybrid MPI+OpenMP program. The default value is given by the `OMP_NUM_THREADS` environment variable.

The [`:s`] option specifies the CPU stride in thread binding. For a pure OpenMP program, [`:0`] (i.e., `s=0`) can be used to indicate that the `-c<list>` option will be taken as the CPU list for threads.

If the `-c<list>` option is not present, the number of process ranks may be determined as (<*node_cpus*> + <*n*> - 1)/<*n*>), where <*node_cpus*> is the number of CPUs in a node. For this case, the `-t<n>` option needs to be present.

`-b<n>` ; selects thread binding mode
This option selects a binding mode for threads. The supported values for <*n*> are:

1 or `gomp`	– for GNU OMP library (`GOMP_CPU_AFFINITY`)
2 or `pgi`	– for PGI compiler (`MP_BIND`)
3 or `psc`	– for Pathscale compiler (`PSC_OMP_CPU_AFFINITY`)
4 or `itc`	– for Intel compiler (`KMP_AFFINITY`)
6 or `xls`	– for IBM XL compiler (`XLSMPOPTS`)
7 or `sun`	– for SUN/Oracle compiler (`SUNW_MP_PROCBIND`)

The default (any) is to use any one of the modes compiled in. Any other value indicates that threads will be bound to the subset of CPUs defined for each process rank.

`-n<n[:s[:np]]>` ; sets number of processes per node

The number of processes per node is automatically determined by default (to the number of processes identified by MPI library if possible or the number of cores on a node). If the number of processes per node is less than the actual core counts on a node (such as when running hybrid codes) and the value cannot be automatically determined from the MPI library, use of this option is recommended. `<`*`n`*`>` is a value between 1 and the number of CPUs in a node.

The [:*`s`*] suboption specifies the spread of processes assigned to nodes. This is useful if the assignment of ranks to a node is not contiguous, such as from cyclic distribution, for which case `[`*`n`*`]` is the process chunk size for each node distribution and `[`*`s`*`]` is the number of nodes. Additional suboption [:*`np`*] indicates the total number of processes.

`-r<`*`list`*`>` ; defines process rank list for a node

`<`*`list`*`>` is a comma-separated group of numbers. The option is intended for more accurate pinning when ranks assigned to a node is not contiguous. It would be more effective when combined with an external tool, such as `mpiexec`, to set the rank list for each node. This option also sets the number of processes per node.

`-nb` ; no process-CPU binding

This option can be used to print the current CPU information without performing additional binding.

`-nb-thr` ; binds MPI processes but does not spread OpenMP threads

This option performs binding for MPI processes as usual, but OpenMP threads will be bound to the same CPU as for MPI process.

`-nb-any` ; no process-CPU binding for a library

This option turns off the process binding by a library, including MPI and OpenMP, if any.

`-nt` ; no setting of `OMP_NUM_THREADS` from the `[-t]` option

With this option, `[-t<`*`nc`*`>]` defines the number of CPUs used for threads in each process. The actual number of OpenMP threads is taken from the `OMP_NUM_THREADS` environment variable.

`-mbind` ; performs process binding without profiling

This option disables the profiling environment and enters the binding-only mode. It is similar to applying the stand-alone tool `mbind.x` directly, but with more accurate binding capability for irregular cases. The option is useful for cases where process binding is required for a rank distribution to nodes that is not in a regular block fashion, such as cyclic or round robin.

The only non-binding `mpiprof` options that are effective in the binding-only mode are `[-lib]`, `[-debug]` and `[-v]`.

3. Environment Variables

This section describes the environment variables that can be used to control profiling itself and output. The environment variables can be overridden or modified by `mpiprof` options as described in Section 2.

`MPROF_PFLAG=`*<value>*

controls the type and details of profiling information. Possible values are listed in Table 3 and Table 4.

The default setting from the `mpiprof` tool is "`on`". The "`disable`" value is only useful for controlling user-instrumented codes. The default behavior for message-size and data-size printing is to scale to a proper unit of [K,M,G,T,P,E,Z]bytes. Use "`byte`" to force a uniform unit.

Note: 1 KB = 10^3 bytes, 1 MB = 10^6 bytes, 1 GB = 10^9 bytes, 1 TB = 10^{12} bytes, 1 PB = 10^{15} bytes, 1 EB = 10^{18} bytes, 1 ZB = 10^{21} bytes.

The same set of symbols [K,M,G,T,P,E,Z] may also be used to scale power usage values, for which case the unit is in joule or watt(=joule/sec). It is useful to know that
1 joule = 1 watt-second, 1 kilowatt-hour = 3.6 megajoules.

The additional value `<n>` for option `cblk`, `pblk`, or `blk` controls the level of details in reporting blocking time. The default or `<n>=2` reports per-function blocking time; `<n>=0` or `1` reports the total blocking time. The `cblk` option can be used in conjunction with `MPROF_MFUNC` to select specific collective functions for blocking time measurement.

Table 3: Supported values for `MPROF_PFLAG`

<value>	**Description**
`disable`	disables profiling environment
`off | false`	switches off profiling
`on | true`	switches on profiling
`cblk[=<n>]`	estimates collective blocking time
`pblk[=<n>]`	estimates point-to-point blocking time
`blk[=<n>]`	is equivalent to "`cblk+pblk`"
`msgm | msgmx`	collects message size and count maps
`cpath | cpathx`	collects call-path information
`mpmt`	enables stats collection from multiple threads
`mem | pwr`	reports memory and power usage only
`sum | nsum`	reports summary only
`ios`	reports I/O statistics and memory usage only
`byte`	prints message size in bytes
`smpc`	reports uncorrected shared memory pages

Some of the `MPI_PFLAG` values can be specified together with "`+`" (enabling) or "`-`" (disabling) to indicate multiple functions. For example, "`blk+msgm`" means to enable blocking time and message maps. By default profiling of Posix I/O functions is enabled but disabled by "`cpath`". To include I/O functions in the call-path collection, use "`cpath+io`".

Value "`sum`" or "`nsum`" reports summary information with per-rank stats. In addition, "`nsum`" disables the report of per-node stats.

Additional values in Table 4 can be used for finer control of profiling from the default setting.

Table 4: Additional values for `MPROF_PFLAG`

<value>	**Description**
`-mpio`	disables MPI-I/O functions
`-mpic`	disables MPI calls
`-io`	disables Posix I/O
`+mio`	reports Posix I/O called from MPI functions
`-pwr`	disables power usage report
`+ack`	enables send-recv hand-shake acknowledgement
`+pbtag`	uses corresponding message tags in `pblk` measurement

Note: Use of `cpath` will disable `io`. "`mem`" is equivalent to "`-mpic-io`". "`ios`" disables MPI calls (`-mpic`) and enables MPI I/O and Posix I/O (`+mpio+io`). Use "`ios+mio`" to include Posix I/O called from MPI functions other than MPI I/O functions. "`ack`" and "`pbtag`" are for internal debugging purpose.

`MPROF_OUT=`<*filename*>
specifies the output filename for writing stats.

If <*filename*>="`-`", the results will be written to <stdout>. If <*filename*> starts with a "`+`" sign, the results will be appended to the end of the file. If <*filename*> starts with a "`%`" sign, the output filename will be suffixed with a number (*nnnn*) to avoid overwriting an existing file. The default filename is a combination of the executable name (if known) and "`mpiprof_stats.out`".

`MPROF_LIB=`<*mproflib*>
specifies the profiling library (for MPI) to use. <*mproflib*> can be one of those listed in Table 5. More accurate *mproflib* names supported can be found with command "`mpiprof -h`".

This variable is useful if the default auto-detection of the runtime environment fails or if the *mproflib* name is different from the default.

Table 5: Supported `mpiprof` libraries

<mproflib>	**Description**
`sgimpt | hpempt`	SGI/HPE MPT library
`mpich`	MPICH library
`openmpi`	Open-MPI library
`hpcx`	HPCX MPI library compatible with Open-MPI
`intelmpi`	Intel-MPI library
`mvapich`	MVAPICH library
`ser`	Serial library

`MPROF_EXEC=`<*executable*>
specifies an executable for symbol information. This is useful for call-path collection. Without the setting,

symbol information is dynamically built. By default, `mpiprof` uses the program name supplied to the command line.

`MPROF_CPATH_DEPTH=`<*depth*>
sets the stack unwinding depth for call-path collection. A value of <*depth*>=0 means no limit on the depth. The default depth is 30.

`MPROF_MFUNC=`<*func*:*m*>
specifies a function to print out stats at each call instance from each rank for monitoring purpose. The function name specified is a short-handed name without the `MPI_` prefix (see Appendix A). Up to four functions may be specified in a comma-separated list. The optional value `<:`*m*`>` controls the number of instances to be monitored. Default for *m* is 1000 if no signal is specified or 0 if a signal is specified. A value of 0 means no limit.

In conjunction with [`cblk=`<*n*>], this option can be used to select specific collective functions for blocking time measurement. Assume

<*n*>	– value supplied to `cblk`
#blk-print	– 0/1 don't/do print per-function blocking time
mfunc-print	– 0/1 don't/do monitor function calls
blk-mfunc	– 0/1 ignore/check *mfunc*-list for blocking time measurement

Effect of control from different values of <*n*> is as follows:

<*n*>	0	1	2	3	4	5	6	7
#blk-print	0	0	1	1	0	0	1	1
mfunc-print	0	1	0	1	0	1	0	1
blk-mfunc	1	1	1	1	0	0	0	0

If the *mfunc*-list is not specified, a value of `cblk`=1 or 3 (or *mfunc-print*=1) disables blocking time measurement for collective I/O functions.

`MPROF_CSIG=`<*signo*>
catches a signal of type <*signo*> and write stats to individual files for each rank. <*signo*> is a signal number or name supported on a system; for example, <*signo*>=2 or `INT` is an interrupt signal. Up to two signals may be specified in a comma-separated list. When multiple signals are specified, any of the specified signals can trigger the writing of stats to output files. If <*signo*> is not specified, it defaults to "2". If <*signo*>=0, no signal is checked.

`MPROF_OUT` defines a common prefix for individual filenames.

Two special cases: <*signo*>=1 or `HUP` will clear stats after they are written to the output; <signo>=2 or `INT` will not clear stats after they are written. If `MPROF_MFUNC` is also specified, <signo>=1 or 2 will toggle on and off of the function monitoring.

`MPROF_PSEC=`<*ival:gival*>
defines power sampling interval (host:gpu) in seconds. The default value for <*ival*> is 300 seconds (or 5 minutes) and for <*gival*> is 5 seconds.

`MPROF_EXPR=`<*exps*>
performs CPU and communication scaling experiments. <*exps*> is a comma-separated list of "`cpu=`*factor*", "`comm=`*factor*", and "`io=`*factor*" pairs, where *factor* is a scaling factor; or "`ideal`" for "`cpu=1,comm=0,io=0`".

`MPROF_DEBUG=`<*rank*>
puts processes in a wait stage at the startup and allow a debugger to attach. The default process to be

attached is rank `0`. From the debugger, set variable `GO=`1 or 2 to continue execution with or without profiling support.

`MPROF_CTAG=`<*ctag*>

defines a tag to be used for calls to either `mprof_init2` or `mprof_pcheck` when accessing via the MPI_Pcontrol interface. Without the environment variable, <*ctag*> defaults to "" (empty string).

4. User-Defined Profiling Interface (`mprof` APIs)

The functions listed in the following subsection can be called inside an application to perform user-defined profiling of MPI functions in selected regions of an application. These functions are not thread-safe. Calling sequence and examples of usage are given in the subsequent subsections.

4.1 The mprof API Routines

- Header file
  The header file contains definitions of constants and prototype functions.

  C interface -
  ```
  #include "mprof_lib.h"
  ```

  Fortran interface -
  ```
  include "mprof_flib.h"
  ```
  or
  ```
  use mprof_flib
  ```

- Initialization of the profiling environment (collective)

  C interface -
  ```
  int mprof_init(int pflag);
  ```
  or
  ```
  int mprof_init2(int pflag, char *ctag);
  ```
  returns the number of implemented functions

  Fortran interface -
  ```
  subroutine mprof_init(pflag)
  ```
  or
  ```
  subroutine mprof_init2(pflag, ctag)
  integer pflag
  character(len=*) ctag
  ```

`pflag`	= `MPF_OFF`	initial profiling mode is off
	= `MPF_ON`	initial profiling mode is on, and previous stats are cleared if any
	= `MPF_CONT`	initial profiling mode is on, and previous stats are not cleared
`ctag`	–	a string tag to label the environment

- Switching on profiling

  C interface -
  ```
  void mprof_start(void);
  ```

  Fortran interface -
  ```
  subroutine mprof_start()
  ```

- Switching off profiling

  C interface -
  ```
  void mprof_stop(void);
  ```

  Fortran interface -
  ```
  subroutine mprof_stop()
  ```

- Check-point and reset of the profiling environment

  C interface -
  ```
  int mprof_pcheck(int pflag, char *ctag);
  ```
  returns the number of saved data sets

  Fortran interface -
  ```
  subroutine mprof_pcheck(pflag, ctag)
  integer pflag
  character(len=*) ctag
  ```

`pflag`	– with the same meaning as for `mprof_init`
`ctag`	– a string tag to label the new environment

- Finalization of the profiling environment (collective)
  The function finalizes the profiling environment, writes stats to output, and deletes saved data sets if any.

  C interface -
  ```
  void mprof_end(void);
  ```

  Fortran interface -
  ```
  subroutine mprof_end()
  ```

4.2 API Calling Sequence

Routines `mprof_init`, `mprof_init2` and `mprof_end` are collective routines and must be called by all ranks. Either `mprof_start` or `mprof_stop` routine may be called multiple times by each rank to start and pause data collection. Either `mprof_init` or `mprof_init2` routine can be used to initialize the profiling environment without or with a tag. When `mprof_pcheck` is called by a rank, the current profiling data for the rank is saved to memory and the environment is re-initialized according to the supplied `pflag`. This function may be called multiple times by each rank. A call to `mprof_end` stores the profiling data, including any saved data sets from `mprof_pcheck`, to disk files and closes the profiling environment. For multiple data sets, `mprof_end` aggregates them from different ranks according to tags, and also writes the accumulated results of these data sets. The `mprof_init` or `mprof_init2` routine may be called again to re-initialize the profiling environment. Without calling `mprof_init` or `mprof_init2` first, any calls to other `mprof` APIs made after `mprof_end` are treated effectively as no-ops.

The basic functionality of the `mprof` API routines is summarized in Table 6.

Table 6: Basic functionality of the `mprof` API routines

Function	**Init-prof-data**	**Save-to-memory**	**Store-to-file**	**Switch-on**	**Switch-off**	**Collective call**
`mprof_init`	Yes	No	No	Yes	Yes	Yes
`mprof_init2`	Yes	No	No	Yes	Yes	Yes
`mprof_pcheck`	Yes	Yes	No	Yes	Yes	No
`mprof_end`	No	No	Yes	No	Yes	Yes
`mprof_start`	No	No	No	Yes	No	No
`mprof_stop`	No	No	No	No	Yes	No

With an instrumented MPI library, the `mprof_init` routine is automatically called at `MPI_Init` (or `MPI_Init_thread`) and the `mprof_end` routine is automatically called at `MPI_Finalize`. So in principle the user is not required to make calls to these two functions. However, because of the dynamic nature of loading the instrumented libraries, it is not guaranteed that the instrumented version of `MPI_Init` (or `MPI_Init_thread`) will be accessed at runtime if the `mpiprof` tool is not used. To get consistent results, it is preferable to explicitly call `mprof_init` or `mprof_init2` in the user-instrumented code.

4.3 Compiling and Executing the Instrumented Code

It is preferable to use the `module` command to load the proper `mpiprof` module for both compilation and runtime execution. The module will set up a few environment variables, including `MPROF_DIR` for the tool installation directory.

```
    % module load mpiprof-module
```

If module is not used, you would need to add "`$MPROF_DIR/lib`" to `LD_LIBRARY_PATH` and "`$MPROF_DIR/bin`" to `PATH` in your runtime environment.

At compilation time, include the following flag:

```
    -I${MPROF_DIR}/include
```

At link time, link with an appropriate `mpiprof` library:

```
    -L${MPROF_DIR}/lib -lmpiprof_<mproflib> -lmprof_io [-lmprof_flib]
```

`<mproflib>` is one of the supported profiling libraries (`sgimpt`, `mpich`, …). The [`-lmprof_flib`] option is needed if the Fortran module interface (`use mprof_flib`) is used. The [`-lmprof_io`] option ensures the linking with the I/O profiling library.

At runtime, you can choose to run the code as usual with or without the use of the `mpiprof` tool. When `mpiprof` is used, it loads the appropriate `mpiprof` library, including `libmprof_io`, and controls profiling via command options.

```
    % mpiexec -np <n> mpiprof [-options] a_inst.out [args]
```

Without the use of `mpiprof`, environment variables described in Section 3 can be used to control the profiling environment. For this case, if the [`-lmprof_io`] option is not used at link time, the I/O profiling will be effectively disabled. The instrumented code will be run as usual:

```
    % mpiexec -np <n> a_inst.out [args]
```

4.4 Accessing mprof API Routines with the MPI_Pcontrol Interface

The `mprof` API routines can be accessed via the MPI_Pcontrol interface for MPI codes. The value of the "`level`" argument passing into MPI_Pcontrol defines the associated `mprof` routine. The supported values for the "`level`" argument and the associated `mprof` routines are summarized in Table 7.

The calling sequence for MPI_Pcontrol should follow the same sequence as defined for the `mprof` routines in Section 4.2. In particular, the property of collective call needs to be observed. The instrumented code with MPI_Pcontrol is compiled as a regular MPI code, i.e., there is no need to link with an `mpiprof` library. However, at runtime the instrumented code needs to run with the use of the `mpiprof` tool as described in Section 4.3.

Table 7: Supported level values for MPI_Pcontrol and the associated `mprof` routines

MPI_Pcontrol call	**`mprof` routine**	**Collective call**
`MPI_Pcontrol(0)`	`mprof_stop()`	No
`MPI_Pcontrol(1)`	`mprof_start()`	No
`MPI_Pcontrol(2)`	`mprof_init2(MPF_ON,ctag)`	Yes
`MPI_Pcontrol(3)`	`mprof_pcheck(MPF_ON,ctag)`	No
`MPI_Pcontrol(4)`	`mprof_init2(MPF_CONT,ctag)`	Yes
`MPI_Pcontrol(5)`	`mprof_pcheck(MPF_CONT,ctag)`	No
`MPI_Pcontrol(-1)`	`mprof_end()`	Yes
"`ctag`" is defined by environment variable `MPROF_CTAG` as described in Section 3.		

4.5 Usage Examples of the Profiling Routines

Example 1: Basic usage of the `mprof` Fortran APIs.

```
% cat mycode.f90

use mprof_flib

...code for initialization and setup
call mprof_init(MPF_ON)     ! initial profiling=on, all ranks
...computation with profiling
call mprof_stop()           ! stop profiling
...code without profiling
call mprof_start()          ! start profiling again
...2nd computation for profiling
call mprof_end()            ! done with profiling and write results, all ranks
...more codes

% module load mpiprof-module
% module load mpi-gcc/mpich-3.3
% mpif90 -I${MPROF_DIR}/include mycode.f90 -o mycode.x \
         -L${MPROF_DIR}/lib -lmpiprof_mpich -lmprof_io
% env MPROF_PFLAG=cblk mpiexec -np 4 ./mycode.x
```

An alternative way of executing the instrumented code (the last line above) is to use `mpiprof` to control profiling:

```
% mpiexec -np 4 mpiprof –cblk ./mycode.x
```

In either case, the profiling data is only reported for the code segments that are instrumented by the user.

Example 2: Illustration of check-point capability (C version).

```
% cat mycode2.c

#include “mprof_lib.h”

...code for setup
mprof_init2(MPF_ON, “init”);    // initial profiling=on, all ranks
```

```
...code for initialization phase
mprof_pcheck(MPF_ON, “iter”);   // check-point at iteration loop
...code iteration phase
mprof_pcheck(MPF_ON, “finish”); // check-point at finish
...finalizing phase
mprof_end();                    // done with profiling and write results, all ranks
...other codes

% module load mpiprof-module
% module load mpi-gcc/mpich-3.1
% mpicc -I${MPROF_DIR}/include mycode2.c -o mycode2.x \
        -L${MPROF_DIR}/lib -lmpiprof_mpich -lmprof_io
% mpiexec –np 4 mpiprof -cblk ./mycode2.x
```

The example produces three sets of profiling data in the output for the corresponding code segments: the first set with a tag “`init`”, the second “`iter`”, and the third “`finish`”. The output also includes an aggregated set of the data tagged as “`total`” at the end.

Note: In versions earlier than 2.8, to define a tag for the first data set, one would need to replace “`mprof_init2`” with a call to “`mprof_init`” followed by a call to function “`mprof_pcheck`”. The [`-lmprof_io`] option was introduced since version 2.9.

Example 3: Basic usage of the `mprof` APIs via MPI_Pcontrol.

```
% cat mycode3.f90

include ‘mpif.h’

...code for initialization and setup
call mpi_pcontrol(2)       ! mprof_init(MPF_ON), all ranks
...computation with profiling
call mpi_pcontrol(0)       ! mprof_stop()
...code without profiling
call mpi_pcontrol(1)       ! mprof_start()
...2nd computation for profiling
call mpi_pcontrol(-1)      ! mprof_end(), all ranks
...more codes

% module load mpiprof-module
% module load mpi-gcc/mpich-3.1
% mpif90 mycode3.f90 -o mycode3.x
% mpiexec -np 4 mpiprof –cblk ./mycode3.x
```

This code is similar to Example 1, but using MPI_Pcontrol. The compilation process is simplified since no linking to an `mpiprof` library is needed. At runtime, the use of the `mpiprof` tool is required. As above examples, the profiling data is only reported for the code segments that are instrumented by the user.

5. Sample Profiling Results Explained

We use the Overflow NTR benchmark test case to explain MPIProf profiling results. For collecting profiling data, we ran the test case with 128 processes on 8 Sandy Bridge nodes of Pleiades in three different setups:

```
   module load comp-intel/2015.3.187 mpi-sgi/mpt.2.15r20
   module load mpiprof-module
   mpiexec –np 128 mpiprof –o=ov_mpf_def.out ./overflowmpi
   mpiexec –np 128 mpiprof –cblk –o=ov_mpf_cblk.out ./overflowmpi
   mpiexec –np 128 mpiprof –cblk –cpath –o=ov_mpf_cpath.out ./overflowmpi
```

The first setup uses the default set of parameters for profiling; the second setup adds the measurement of blocking time for MPI collectives; the third setup reports call-path information in addition. The outputs were written to three separate files at the end of each run.

5.1 Run Summary

An output file may contain multiple data sets separated by marker "`====<<Stats..>>`". Each section of the output file (`ov_mpf_def.out`) is marked with "`==>`". The first section of the output file contains the `mpiprof` version number, run date, a few basic host information, and a list of run conditions indicated by environment variables, including the MPI library used. The "Summary of this run" section contains a few statistics about the run as well as overall timing, volume and rate information in groups. For data sets produced from `mprof_pcheck` (check-point), only results from the active ranks that are involved in the call for a given data tag (`ctag`, see Section 4) are reported. At the very end of the section, a set of high-level remarks about profiling data may be included.

```
==> Summary of this run:
Number of nodes           = 8
Number of MPI ranks       = 128
Number of inst'd functions  = 18

Execution time
   Total wall clock         = 1014.83 secs
   Average wall clock       = 1014.71 secs
   Average computation      = 849.274 secs (83.70%)
   MPIProf overhead         = 0.11080 secs ( 0.01%)

   Average communication    = 165.183 secs (16.28%)
      collective            = 97.8256 secs ( 9.64% or 59.22%Comm)
      point-to-point        = 46.6131 secs ( 4.59% or 28.22%Comm)
      test-wait             = 20.7440 secs ( 2.04% or 12.56%Comm)
   . . . . . .
```

Most of the timings are averaged over all active ranks. Total wall clock time is the maximum wall clock time from all active ranks. Percentage of time is relative to the average wall clock time. Rates are first calculated from data size and timing information for each rank and then aggregated for all active ranks. Meaning of a few key entries in this section is summarized in Table 8.

The total (or average) wall clock time does not include time spent in `mprof_end`, which is reported separately at the end, and in `MPI_Finalize`. Gross rate is a sum of rates from all active ranks. The rate value is likely over-estimated in many cases because, for instance, the effect of overlapping communication and computation via nonblocking communication calls is not taken into consideration. Effective I/O time is estimated by dividing the total I/O data size with the aggregated I/O rate from all active ranks. It is useful for a case where I/O occurs only on a small number of ranks, such as rank 0. For this case, average I/O time may not give a sensible idea about how much time the application spends in I/O. This is demonstrated by the profiling data for Overflow where rank 0 performs most of the I/O:

```
   Average I/O (%, L, H)     = 0.13942 secs ( 0.01%, 0.00000, 17.8451)
      write+close            = 0.10208 secs ( 0.01%, 0.00000, 13.0656)
      read+open              = 0.03731 secs ( 0.00%, 0.00000, 4.77506)
   Effective I/O (%, iF)     = 17.8121 secs ( 1.76%, 126.76)
      effective write        = 13.0371 secs ( 1.28%, 126.72)
      effective read         = 4.77506 secs ( 0.47%, 127.00)
```

where '`%`' is percent of the total time, '`L`' is the minimum I/O time on a rank, and '`H`' is the maximum I/O time on a rank. The "`iF`" value in the effective I/O time stands for imbalance factor, which is calculated as

(*T*(effective) – *T*(average)) / *T*(average).

It serves as an indicator of how I/O is balanced across ranks. For a perfectly balanced case, `iF` should be 0. For the Overflow run a large value of 126, which is close to the number of ranks, suggests a large I/O imbalance.

Table 8: A few report entries and their meanings

Entry	**Symbol**	**Description**

`Total wall clock time`
`Average wall clock time`
	T(wallclock)	Time spent from `MPI_Init` (inclusive) up to `MPI_Finalize`, or from `mprof_init` to `mprof_end`, or in between `mprof_pcheck`, `mprof_init` and `mprof_end`

`Average computation time`	*T*(comp)	= *T*(wallclock) – *T*(comm) – *T*(i/o) – *T*(overhead)
`MPIProf overhead time`	*T*(overhead)	Average time used by MPIProf for gathering data, including `mprof_init` and `mprof_pcheck` but excluding `mprof_end`
`Average communication time`	*T*(comm)	Average time spent in MPI calls, excluding MPI-IO
`Average I/O time`	*T*(i/o)	Average time spent in MPI-IO and Posix I/O
`Effective I/O time`	*T*(eff_i/o)	Time estimated from I/O rates for each rank
`Communication rate`	*r*(comm)	= Message size / *t*(comm) for each rank
`I/O rate`	*r*(i/o)	= Data size / *t*(i/o) for each rank

The summary-run section may include a few high-level remarks about profiling data that are deduced from timing, message volume, and I/O volume to characterize a run. Remarks may contain:
›› basic characteristics – compute intensive, communication intensive, and/or I/O intensive
›› load balance or imbalance (spread) information
›› any particular performance patterns observed.

For the Overflow run, sample remarks look like:

```
   compute intensive (849.274 secs, %comp=83.70 > 80)
   compute time: imbalance clustered in 16 ranks in range 0-15
      average value in cluster 683.496, median 869.226 (spread 21.37%)
   i/o: from a single rank (0), time 17.8451
      rank 0, volume 18.011G
```

The summary section also reports the memory usage (total and per-rank) of the application as well as node memory information (five parts: available, used by the application, shared memory page, used by system+other applications, and free). The memory information is from High Water Mark (HWM) at the time when `mprof_end` is called. The reported process memory (`AppUsed`) may contain a correction for memory over-counting due to shared memory pages. The assumption is that each process on a given node accesses a similar set of shared memory pages. The amount of shared memory, M_S (`SmpEsti`), accessed by the application on a node is estimated by the average of the shared memory pages reported for each process (`SmpUsed`). The process memory from HWM is then corrected by subtracting out the amount of

$(p - 1)/p * M_S$

where p is the number of processes on a node. The correction is also reflected in the node memory report. When the correction is disabled via the `[-smpc]` option, the reported shared memory page for a node is simply a sum of those reported for processes on the node. The memory summary section may also include minimum, maximum, average, and standard deviation of process and node memories if the number of processes or nodes is more than one.

5.2 Power and Energy Usage

If supported by the underlying operating system, the summary section may also report the energy and power usage (total and per-node) of the application measured from the first call of `mprof_init` to each call of `mprof_end`. The unit for energy usage is in *joule* and for power rate is in *joule/sec* or *watt*. A sample power report from an 8-node run looks like the following:

```
==> Power usage summary, rate in watts
Label   EnerTot PwrRate EnerAve EnerMin (node)  EnerMax (node)
.pack0  823.59K 811.650 102.95K 99.799K (0)     107.13K (7)
:core0  689.19K 679.198 86.148K 83.318K (2)     90.060K (7)
.pack1  806.31K 794.625 100.79K 97.716K (3)     103.22K (4)
:core1  672.71K 662.956 84.088K 81.214K (3)     86.555K (4)
----------
.pack+  1629.9K 1606.27 203.74K 198.60K (0)     209.55K (7)
:core+  1361.9K 1342.15 170.24K 165.89K (0)     175.88K (7)

Sampling interval = 300 secs, 4 power states

Total energy = 1629.9KJ, power = 1606.27W

==> Node power rate information
Node    Rank    Time    .pack0  :core0  .pack1  :core1  .pack+  :core+
0       0       1014.83 98.3404 82.1059 97.3526 81.3635 195.693 163.469
1       16      1014.67 102.336 85.3011 99.7342 82.6113 202.070 167.912
2       32      1014.71 98.6626 82.1100 98.9018 82.3881 197.564 164.498
3       48      1014.67 101.711 85.2744 96.3033 80.0393 198.014 165.314
4       64      1014.67 98.8100 82.2098 101.730 85.3040 200.540 167.514
5       80      1014.72 102.394 85.9658 99.7662 83.2503 202.161 169.216
6       96      1014.71 103.817 87.4731 99.8913 83.4161 203.708 170.889
7       112     1014.67 105.579 88.7578 100.946 84.5830 206.525 173.341
```

"[.:]*label*" is a power state (*.package*) or a substate (*:core* or *:dram*). An asterisk (`*`) or plus (`+`) in label indicates summation across sockets at a given level. `EnerTot` is the total energy usage aggregated over all nodes for each power state, and power rate in watts is calculated from energy (joule) / time (sec). `EnerAve` is the energy usage averaged over nodes. `EnerMin` and `EnerMax` are the minimal and maximal energies, respectively, measured for a node. `Rank` indicates the corresponding MPI process that reads power information on each node.

The sample output above indicates that the total energy is 1629.9 KJ and the total power is 1606.27 watts.

On system with NVIDIA GPUs, MPIProf also reports GPU memory and power usage for the duration between `mprof_init` and `mprof_end`. The fields include the following, one line for each GPU:

```
Node    GPU     MemTot  MemUsed Energy  PwrRate
0       0       33.291G 731.90M 2530.22 74.7343
0       1       33.291G 731.90M 2643.45 78.0787
Total   2       66.582G 1463.8M 5173.67 152.813
```

Aggregated values are included at the end of the section.

5.3 Function Summary

The function summary section includes a list of instrumented functions used in the application and breakdown timing and message size for each function. Data of functions that are collective, point-to-point, one-sided, test-wait, MPI I/O, Posix I/O, or miscellaneous in nature are cumulated. Each function is labeled with a letter to indicate its type:

- `(c)` – collective communication
- `(p)` – point-to-point communication
- `(o)` – one-sided communication
- `(w)` – test-wait for nonblocking communication
- `(C)` – collective MPI I/O
- `(N)` – non-collective MPI I/O
- `(I)` – posix I/O
- `(<)` – posix I/O from collective MPI I/O
- `(:)` – posix I/O from non-collective MPI I/O
- `(m)` – miscellaneous.

Below is an example of the per-function summary section. Functions are sorted by time. The reported timing is an average over all active ranks, while the number of functions calls and message/data sizes are accumulated results.

```
==> Per function summary
Func     Time    %Total   %CommIO NCalls   MsgSend MsgRecv Written Read
gatherv 78.5834 7.74     47.53   384128   127.37M 127.37M 0       0
send    42.5279 4.19     25.72   11.770M 1.4573T 0       0       0
waitall 20.744  2.04     12.55   192279   0       0       0       0
bcast   10.8975 1.07     6.59    683904   39.970G 39.970G 0       0
barrier 7.70817 0.76     4.66    192128   0       0       0       0
recv    4.05408 0.40     2.45    2091989 0       25.945G 0       0
allredu 0.58059 0.06     0.35    19200    131.06M 131.06M 0       0
write   0.10194 0.01     0.06    18621    0       0       16.116G 0
allgath 0.05391 0.01     0.03    128      23.929M 23.929M 0       0
read    0.03676 0.00     0.02    772      0       0       0       1.8946G
irecv   0.02763 0.00     0.02    9683540 0       1.4358T 0       0
isend   0.00345 0.00     0.00    5901     4.4371G 0       0       0
gather  0.00172 0.00     0.00    128      101.60K 101.60K 0       0
open    0.00055 0.00     0.00    97       0       0       0       1
allgatv 0.00031 0.00     0.00    128      186.94K 186.94K 0       0
init    0.00031 0.00     0.00    128      0       0       0       0
close   0.00013 0.00     0.00    92       0       0       1       0
lseek   0.00003 0.00     0.00    1960     0       0       0       0
-------
collv   97.8256 9.64     59.17   1279744 40.252G 40.252G 0       0
p2p     46.6131 4.59     28.20   23.551M 1.4617T 1.4617T 0       0
twait   20.744  2.04     12.55   192279   0       0       0       0
px-io   0.13942 0.01     0.08    21542    0       0       16.116G 1.8946G
misc    0.00031 0.00     0.00    128      0       0       0       0
-------
Comm+IO 165.322 16.29    100.00  25.045M 1.5020T 1.5020T 16.116G 1.8946G
Ovhead  0.1108  0.01
TotComp 849.274 83.70
Total   1014.71 100.00
```

For this particular case, the section indicates that the application spent 83.70% time in computation and 16.29%

time in communication and I/O, of which a majority time was in `MPI_Gatherv` (47.53%) followed by `MPI_Send` (25.72%) as shown in Figure 1. The figure also includes results from a different setup where the blocking time in MPI collectives was measured. Details of this setup are described in Section 5.5.

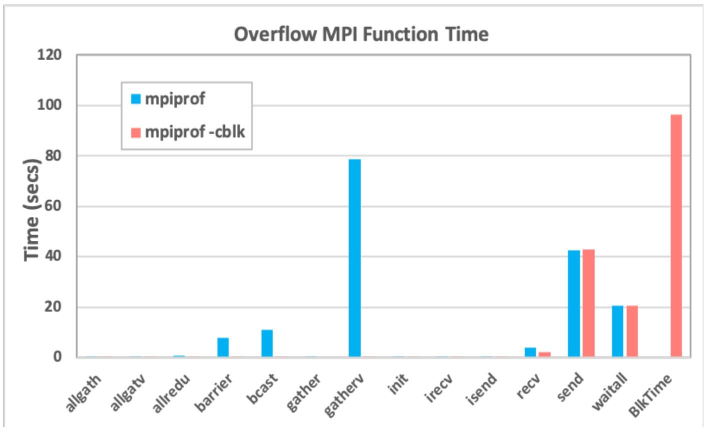


Figure 1: MPI functions time as reported by `mpiprof` in two different setups.

Message size and I/O size histograms give characteristics of message size and I/O size distribution in the application. The `CSndCnt` and `CRcvCnt` columns include message size from collective calls; whereas the `SendCnt` and `RecvCnt` columns cover message size from rest of the communication calls. The histogram results for Overflow are shown in Figure 2. As one can see from the figure, the message size spreads out in a wide range from 4 bytes to 1MB, while I/O is mainly from write of 64-byte and 4-MB data.

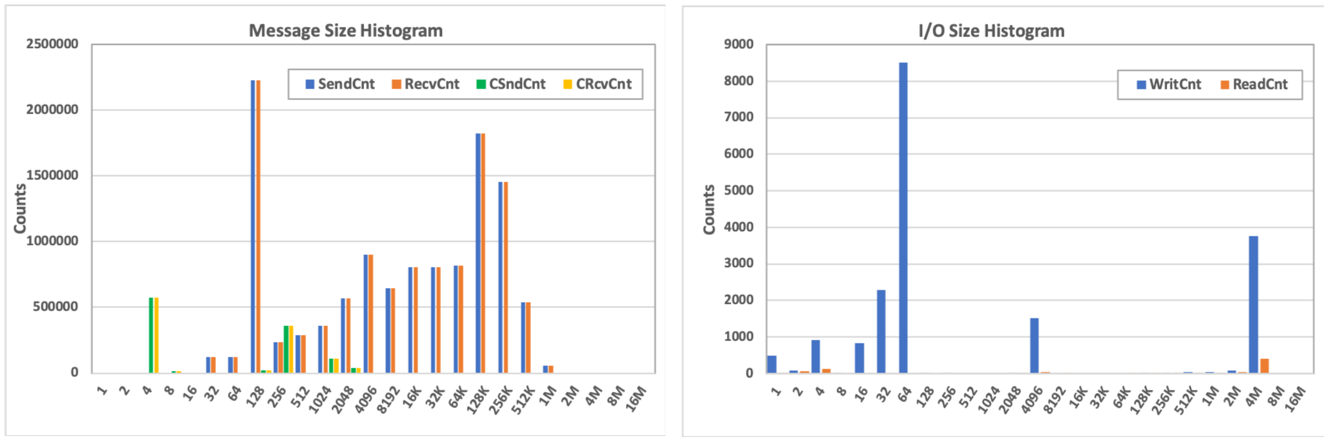


Figure 2: Message size and I/O size histograms.

5.4 Function Profile Details for Each Rank

For each instrumented function, `mpiprof` collects the following accumulated statistics: time spent in the call, the number of calls, message size (send, receive, or collective), and data size (read, write). The results are reported for each rank and grouped for each statistic type. At the end of each type block, averages are reported for timing and totals are reported for other types.

A snapshot of the timing blocks for the Overflow run is given below.

```
==> Summary - Timing in seconds
Rank     TotTime TotComp TotComm Tot-IO  collv   p2p     twait   px-io   misc    Ovhead
0        1014.83 707.832 288.913 17.8451 238.971 17.6006 32.3408 17.8451 0.0002  0.24328
1        1014.83 683.191 331.419 0       264.912 29.7278 36.7784 0       0.00018 0.21804
2        1014.83 671.095 343.517 0       274.697 50.6704 18.1497 0       0.00016 0.21558
```

```
3       1014.83 668.267 346.337 0       271.533 42.766  32.0379 0       0.00015 0.22305
. . . . . .
--------
Average 1014.71 849.274 165.183 0.13942 97.8256 46.6131 20.744  0.13942 0.00031 0.1108
Stddev  0.04992 67.0469 66.7758 1.57729 67.4232 11.745  11.1739 1.57729 0.00007 0.04423
%Total  100.00  83.70   16.28   0.01    9.64    4.59    2.04    0.01    0.00    0.01

==> Timing of functions in seconds
Rank    Comm+IO init    allgath allgatv allredu barrier bcast   gather  gatherv irecv   isend ...
0       306.758 0.0002  0.07796 0.00041 0.25595 14.1784 0.59195 0.00278 223.864 0.02398 0.02159
1       331.419 0.00018 0.07339 0.00029 0.30472 15.2336 5.42283 0.00001 243.878 0.01519 0.00541
2       343.517 0.00016 0.0784  0.0003  1.89483 14.6314 4.77305 0.00001 253.319 0.02009 0.00651
3       346.337 0.00015 0.06881 0.0003  1.37812 10.1448 4.32272 0.00001 255.619 0.02624 0.00438
. . . . . .
--------
Average 165.322 0.00031 0.05391 0.00031 0.58059 7.70817 10.8975 0.00172 78.5834 0.02763 0.00345
Stddev  67.0542 0.00007 0.01446 0.00006 0.41902 2.46189 5.6718  0.00072 65.6345 0.00849 0.00294
%CommIO 100.00  0.00    0.03    0.00    0.35    4.66    6.59    0.00    47.53   0.02    0.00
```

The first block contains a summary of timing for function types as well as tool overhead measured on each process. The second block contains breakdown timing for each function. This information can be easily fed into a graph representation as illustrated in Figure 3. The graph shows a sizable imbalance in the first 16 ranks from `MPI_Gatherv`, which is in line with the summary remarks in the Summary section.

5.5 Blocking Time in MPI Calls

Time spent in a message-passing call consists of two parts – time in waiting for the post of a message from a remote rank and the actual time in transmitting the message. A large waiting (or *blocking*) time is usually an indication of load imbalance. Without knowing the implementation details of an MPI library, there is no direct way of measuring the two parts separately. MPIProf estimates the blocking time in collective MPI calls by inserting a barrier in front of each collective call and measuring the time spent in the barrier. This capability is enabled by the option [`-cblk`] to `mpiprof`. A limited support for estimating blocking time in point-to-point MPI calls is provided by the option [`-pblk`].

With blocking time measurement, additional fields are included in the reported results.

```
Execution time
   . . . . . .
   Average communication    = 163.153 secs (16.13%)
   Average blocking         = 96.2980 secs ( 9.52% or 59.02%Comm)
   Effective communication  = 66.8546 secs ( 6.61%)
      collective            = 0.91136 secs ( 0.09% or  0.56%Comm)
      point-to-point        = 45.2177 secs ( 4.47% or 27.71%Comm)
      test-wait             = 20.7254 secs ( 2.05% or 12.70%Comm)
   . . . . . .
Communication and I/O rate
   Gross communication      = 10.4065 Gbytes/sec
      collective            = 414.078 Mbytes/sec
      point-to-point        = 22.1664 Gbytes/sec
   Communication per rank   = 81.3011 Mbytes/sec
   Effective communication  = 22.8912 Gbytes/sec
      collective            = 44.1672 Gbytes/sec
      point-to-point        = 22.1664 Gbytes/sec
   Effective comm per rank  = 178.838 Mbytes/sec
```

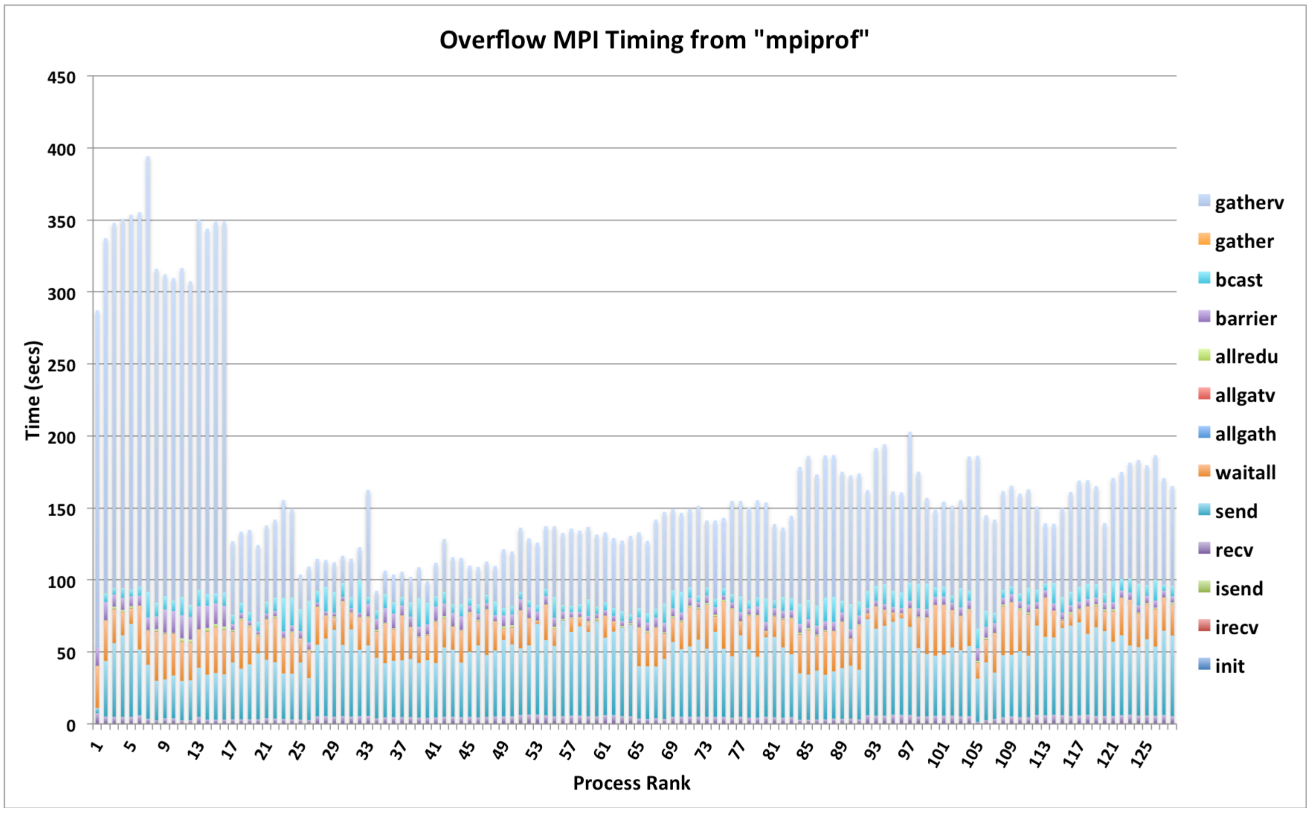


Figure 3: Timing of MPI functions for each rank.

Effective communication time excludes blocking time in communication calls. Effective communication rate is calculated from the effective communication time. This information gives a more accurate report of the capability of the underlying network. Effective rates calculated for each communication type (such as collective or point-to-point) are also included in this section. In the per-function summary section, the "`BlkTime`" column shows the average blocking time across all processes for each function. In the breakdown time sections, the reported time for the corresponding communication functions also excludes blocking time. A '`#`' sign in front of a name indicates blocking time measured for the corresponding function or communication type, as shown in the following.

```
==> Timing of functions in seconds
Rank    Comm+IO init    allgath #allgat allgatv #allgav allredu #allred gatherv #gathev barrier ...
0       308.556 0.00017 0.00039 0.07777 0.00022 0.0002  0.01807 0.3057  0.70432 226.283 0.07111
1       329.488 0.00016 0.0005  0.07292 0.00022 0.0001  0.01813 0.30047 0.29052 244.971 0.07094
2       342.691 0.0002  0.0005  0.07802 0.00022 0.0001  0.01794 1.01658 0.28913 255.818 0.07095
3       343.618 0.00016 0.0005  0.06843 0.00022 0.0001  0.01795 1.01684 0.28539 255.088 0.07078
. . . . . .
```

With blocking time measurement via the [`-cblk`] option, the MPI function time for Overflow is shown in Figure 4. Comparing to Figure 3, it is obvious that the sizable imbalance in the first 16 ranks is mostly due to blocking in collective calls (`#gathev` in particular). The actual time spent in collective MPI calls, including `MPI_Gatherv`, is relatively small, as shown in the bar graph in Figure 1.

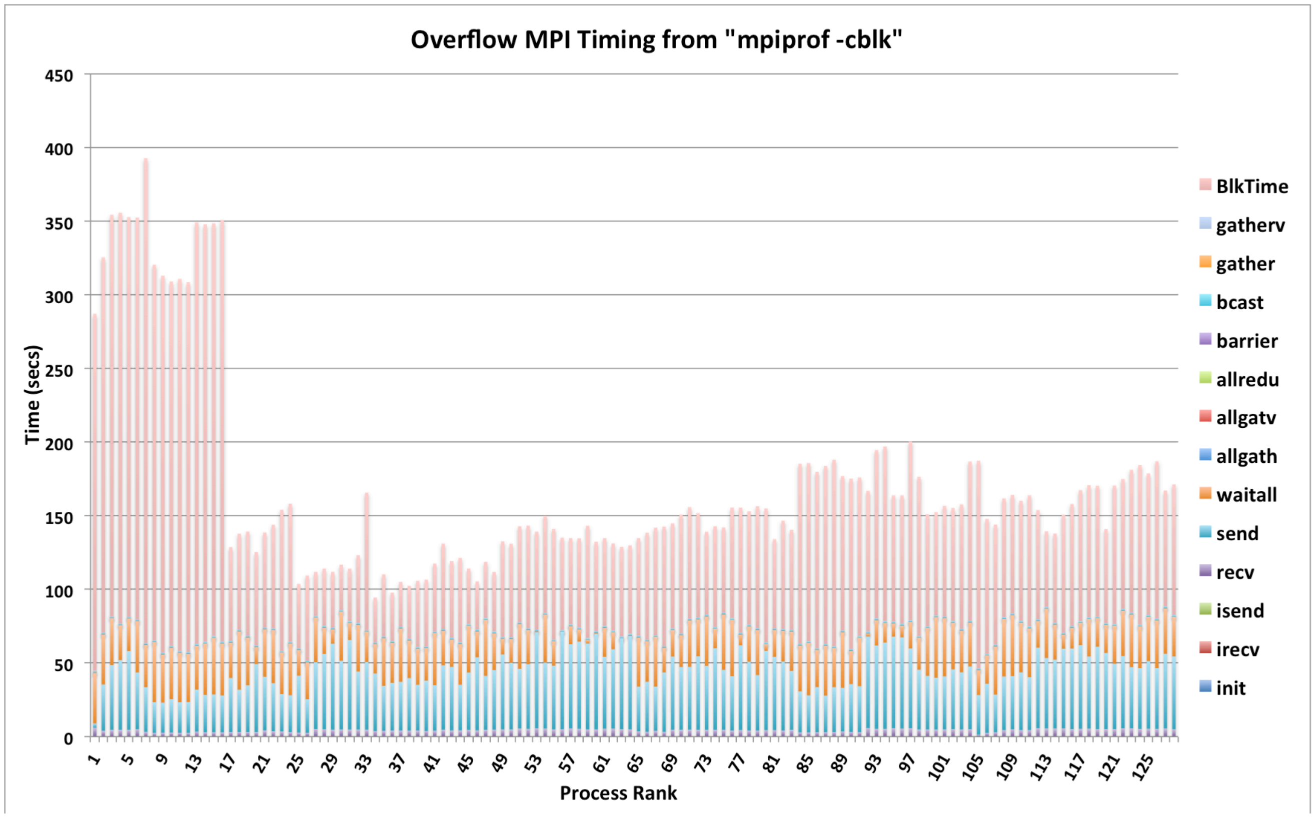


Figure 4: Timing of MPI functions after excluding blocking time

5.6 I/O Statistics from the `ios` Report

The [`-ios`] option for `mpiprof` (or the `ios` value to `MPROF_PFLAG`) can be used to profile only Posix I/O and MPI I/O calls. With this option, the statistics of low-level Posix I/O calls made by MPI I/O functions are also included and marked with the '`<`' sign in the front of each called function name in the report. Timing and data sizes of the called low-level I/O functions are not included in the aggregated statistics. Table 9 summarizes how Posix I/O data are reported under different profiling controls. In the table, symbol "`px-io`" stands for Posix I/O, whereas symbols "`c-mpio`" and "`nc-mpio`" stand for collective and non-collective MPI I/O. For profiling control, the "`ios`" option implies "`+io+mpio-mpic`". The [`-ios_nompi`] option to `mpiprof` is equivalent to a value of "`ios-mpio+mpic`" specified to `MPROF_PFLAG`. The "`+mpic`" value (or `+mio`) in this case indicates that Posix I/O calls made by MPI functions other than MPI I/O functions will be reported.

Without the [`-ios`] option, `mpiprof` reports only Posix I/O calls that are not made by MPI functions. For this case, specifying the "`+mio`" value to `MPROF_PFLAG` will include the report of Posix I/O calls made by MPI I/O functions (labeled as "`<px-io`").

Table 9: Posix I/O data in the `ios` report

Posix I/O calls	**Not in MPI**		**In MPI I/O**		**In Other MPI**	
Profiling control	`+io`	`-io`	`+mpio`	`-mpio`	`+mpic` or `+mio`	`-mpic`
MPI (I/O) reporting	N/A	N/A	`c-mpio` `nc-mpio`	None	None	None
Posix I/O reporting	`px-io`		`<px-io`	`px-io`	`px-io`	
Total I/O reporting	`px-io`		`c-mpio` `nc-mpio`	`px-io`	`px-io`	

The `ios` report is useful for understanding some of the MPI implementation details in dealing with parallel I/O. As an example, an application with collective MPI I/O was run on a Lustre filesystem, using 120 ranks across 5 nodes with 24 ranks per node. The underlying MPI library was the SGI MPT with Lustre filesystem support. I/O stats from the [`-ios`] option show that

- with a Lustre stripe count of 1, only rank 0 does the writes;
- with a Lustre stripe count of 12, only 4 ranks (0, 24, 48, 72) do the writes.

The results reflect the optimization made by SGI MPT for collective I/O buffering under different stripe counts.

5.7 Call-path Information

To get a better sense of where the communication time is spent in an application, one can invoke the call-path measurement of MPI calls via the [`-cpath`] option to `mpiprof`. Inclusive time along the call path is gathered for each rank and the final results are reported as average from all ranks. To include call-path information for individual ranks, use the [`-cpathx`] option to `mpiprof`. A few statistics about the call graph (depth, number of nodes and edges) is included in the beginning of the report section. The depth is limited to 30 by default, which can be changed by setting the `MPROF_CPATH_DEPTH` variable.

The [`-cpath`] option disables Posix I/O profiling. To enable I/O profiling in the call-path measurement, one can use the [`-cpath -io`] option to `mpiprof` or set the environment variable `MPROF_PFLAG=cpath+io`.

Below is a sample output from the Overflow run with the [`-cblk -cpath`] option. A '`#`' sign following a name indicates the critical path in the call graph; that is, a path that spends the most time in the call graph. Dotted lines (`......`) mark leaf nodes, while a name without dotted lines indicates a parent node. A leaf node is typically an MPI function, but in some cases leaf nodes have names like `*mprof_get_cbtime`, which indicate functions used for performing the blocking time measurement for the corresponding MPI functions. The standard deviation (`stddev`) for an averaged time is calculated across all ranks. Function call count, indicated by the number following (`stddev`), is only reported for the maximum value among all ranks for the leaf nodes. The (`Rank`) field indicates the associated rank for the reported count.

Along the critical path in routine "`overfl`," a majority of the time was spent in blocking for function "`mpi_gatherv`."

```
==> Call path inclusive time for MPI calls (average+stddev)
Call path stats: depth = 7, nodes = 62, edges = 131, name_size = 12

ProcedureName_______________________________________________ Time     Stddev   Count (Rank)
MAIN__#:                                                     164.3534 66.39164
  init_all_:                                                 0.000159 0.000012
    mpi_init_:.............................................. 0.000159 0.000012 1 (0)
  overgl_:                                                   0.374469 0.134416
    dist_global_:                                            0.208910 0.125505
      mpi_bcast_:                                            0.208910 0.077744 3 (0)
        __mpifprof_MOD_mprof_get_cbtime:.................... 0.199483 0.143102 3 (0)
    dist_omiglb_:                                            0.032375 0.018584
      mpi_bcast_:                                            0.032375 0.012048 2 (0)
        __mpifprof_MOD_mprof_get_cbtime:.................... 0.030914 0.022177 2 (0)
    mpi_bcast_:                                              0.000076 0.000028 1 (0)
      __mpifprof_MOD_mprof_get_cbtime:...................... 0.000073 0.000052 1 (0)
    . . . . . .
    finit_:                                                  0.037544 0.022339
      dist_floinp_:                                          0.037544 0.022339
        mpi_bcast_:                                          0.037544 0.013972 2 (0)
          __mpifprof_MOD_mprof_get_cbtime:.................. 0.035850 0.025718 2 (0)
```

```
oversz_:                                              0.327860 0.080913
  initdi_:                                            0.159493 0.040509
    dist_split_:                                      0.057968 0.023808
      mpi_bcast_:                                     0.057968 0.021572 4 (0)
        __mpifprof_MOD_mprof_get_cbtime:............. 0.055352 0.039707 4 (0)
  . . . . . .
overst_:                                              3.843016 0.467809
  xinit_:                                             1.726394 0.330580
    read_gridin_:                                     1.726394 0.369672
      mpi_send_:..................................... 0.007902 0.001964 1464 (0)
      mpi_recv_:..................................... 1.718492 0.625556 12 (16)
  xintout_read_:                                      2.036445 0.461129
  . . . . . .
overfl_#:                                             158.7676 66.33126
  igend_#:                                            75.44221 65.64528
    dist_monitr_#:                                    75.06397 65.64527
      mpi_gatherv_#:                                  75.06397 65.59037 3000 (0)
        __mpifprof_MOD_mprof_get_cbtime#:............ 74.72019 53.60181 3000 (0)
    mpi_bcast_:                                       0.378247 0.140761 1500 (0)
      __mpifprof_MOD_mprof_get_cbtime:............... 0.361178 0.259097 1500 (0)
  cbcxch_:                                            39.30533 13.20705
    cbc_recv_:                                        0.026994 0.010707
      mpi_irecv_:.................................... 0.026994 0.010683 205500 (30)
    cbc_send_:                                        20.45587 7.677727
      mpi_send_:..................................... 20.45587 5.083873 174000 (19)
    cbc_dist_:                                        18.82247 10.74610
      mpi_waitall_:.................................. 18.82247 10.72265 1500 (0)
  inend_:                                             41.40045 8.044489
    mpi_bcast_:                                       14.30447 5.323283 3000 (0)
      __mpifprof_MOD_mprof_get_cbtime:............... 13.65895 9.798483 3000 (0)
    mpi_barrier_:                                     0.205062 0.110057 1500 (0)
      __mpifprof_MOD_mprof_get_cbtime:............... 0.187074 0.134201 1500 (0)
    fomoin_:                                          3.189077 1.556822
      mpi_send_:..................................... 0.009689 0.002408 117000 (31)
      mpi_recv_:..................................... 0.870846 0.317000 148650 (92)
      dist_fmint_:                                    2.208768 1.215117
        mpi_allreduce_:                               2.208768 1.225398 150 (0)
          __mpifprof_MOD_mprof_get_cbtime:........... 2.152196 1.543914 150 (0)
    . . . . . .
overdo_:                                              0.961642 0.621435
  mpi_barrier_:                                       0.961642 0.516117 1 (0)
    __mpifprof_MOD_mprof_get_cbtime:................. 0.877287 0.629337 1 (0)
timer_summ_:                                          0.078622 0.030362
  mpi_gather_:                                        0.002355 0.010100 1 (0)
    __mpifprof_MOD_mprof_get_cbtime:................. 0.000069 0.000049 1 (0)
  grdwghts_:                                          0.076267 0.028632
    mpi_gatherv_:                                     0.076267 0.066642 1 (0)
      __mpifprof_MOD_mprof_get_cbtime:............... 0.075918 0.054461 1 (0)
```

6. Accuracy and Overhead Study

We used the NAS Parallel Benchmarks (NPBs) for conducting the accuracy and overhead study. Version 3.3.1 of NPBs contains internal timers for communication calls. These timers are compared with `mpiprof` results for accuracy study. For collecting profiling data, we ran the Class C problem size of NPBs with 64 processes on Sandy Bridge nodes of Pleiades and the Class D problem size with 4096 processes on Haswell nodes of Pleiades with two different MPI libraries (SGI MPT and Intel MPI) in three setups:

- without `mpiprof`
- with `mpiprof` in default setting
- with `mpiprof -cpath` option.

The reported benchmark time, benchmark communication time, and measured communication time by the tool are summarized in Table 10.

Table 10: Comparison of benchmark and communication times from three sets of runs

	no `mpiprof`		with `mpiprof`			with `mpiprof -cpath`		
Benchmark	Bmk-Time	Bmk-Comm	Bmk-Time	Bmk-Comm	mpiprof-Comm	Bmk-Time	Bmk-Comm	mpiprof-Comm
bt.C.64	18.9623	1.8871	18.8956	1.9270	1.9089	19.0324	1.9993	1.9696
cg.C.64	4.8216	1.5901	4.8298	1.6629	1.6765	4.9218	1.7533	1.6244
ft.C.64	6.3641	2.4831	6.4016	2.5216	2.6989	6.3393	2.4657	2.6104
is.C.64	0.5781	0.3517	0.5966	0.3704	0.4254	0.5920	0.3653	0.4882
lu.C.64	16.2089	2.8413	16.3658	2.9973	2.4488	17.5130	4.2008	2.7898
mg.C.64	1.4375	0.1477	1.4569	0.1656	0.1851	1.4793	0.1908	0.2518
sp.C.64	18.9689	2.6778	18.4860	2.4800	2.4303	18.6505	2.6977	2.6114

The difference between reported benchmark times without and with `mpiprof` is an indication of overhead induced by the tool measurement. As we see from the table, the difference is less than 3% with `mpiprof` in default setting, and slightly larger for the `-cpath` option. For accuracy of the measurement, we see a general agreement between the benchmark-reported and tool-measured communication times, except for the LU benchmark for which the tool reports a smaller value (2.8 secs versus 4.2 secs). After inspecting the LU code, we noticed that the benchmark-reported communication time includes data copies between communication buffers and solution data, which made the benchmark-reported time larger.

In general the overhead of MPIProf is small because the tool uses a counting approach. However, collecting call-path information (from the `-cpath` option) could induce more overhead. The reported overhead time includes time spent in `mprof_init` as well as extra time in gathering data for each function. Time spent in `mprof_end`, which writes results to output file, is reported separately at the end. For the 4096-process runs of the Class D NPBs on 171 Pleiades-Haswell nodes the average values of measured overhead times from six benchmarks (without IS) are:

For SGI MPT, overhead = 0.144 secs, `mprof_end` time = 0.172 secs;
For Intel MPI, overhead = 3.225 secs, `mprof_end` time = 0.145 secs.

The slightly larger overhead for Intel MPI is mainly from `mprof_init` in the initialization process.

Extra memory induced by MPIProf itself is relatively small; however, there is a linear dependency on the number of processes, as estimated by:

For rank=0: $(656+40*M)*N+5984$
For rank>0: $40*(M+N)+5488$

where N is the number of processes and M is the number of instrumented functions in the application, typically

10-15. When call-path information is collected, additional memory is required and the amount of memory increase depends on the size of the call graph.

For a case of N=4096, M=12, we have
    mem(rank=0) = 4.659 MB
    mem(rank>0) = 0.170 MB
For a case of N=10K, M=15, we have
    mem(rank=0) = 12.566 MB
    mem(rank>0) = 0.406 MB

7. Miscellaneous Capability and Tools

Besides profiling, `mpiprof` may be used for monitoring or debugging a running application. Three options are provided for this purpose:

[`-csig`]	– to write profiling information in the middle of a run;
[`-mfunc`]	– to print timing stats for selected functions during a run;
[`-debug`]	– to allow debugger attachment at the startup.

In addition, a utility tool, `rd_stats`, may be used for post-processing profiling results.

7.1 Processing Signals

With the [`-csig`] option, when a specified signal (such as via `kill`, `sigpg`, or `sigpg_job`) is caught, profiling information at the time will be written to output files from each rank, separately. This allows one to monitor the statistics of running jobs. A typical use of this option is as follows. First, launch the job with the desired options, such as:

```
% module load mpiprof-module
% mpiexec –np 32 mpiprof –csig=1,2 bin_mpt_avx/lu.D.32
```

The option [`-csig=1,2`] indicates to catch either signal=1 or 2. The stats will be written to the output files when either of the signals is caught. The difference is that signal=1 will reset internal buffers after the stats are written out. Assume a job is running under PBS with a job_id=299326. We can use the tool "`sigpg_job`" to send a signal to the job from a separate window (the default signal to catch is 2):

```
% sigpg_job 299326 lu.D.32
host=r301i0n0 sigpg ppid|cmd=lu.D.32 sig=2
host=r301i0n1 sigpg ppid|cmd=lu.D.32 sig=2
signal 2 sent to 16 processes with ppid|cmd=lu.D.32
signal 2 sent to 16 processes with ppid|cmd=lu.D.32
```

The first argument to `sigpg_job` is the job id, followed by the name of the executable. The messages indicate that signal=2 has been sent to each of the 16 processes on two nodes. In the job running window, messages are printed to indicate the signal has been received and the statistics are written to output files:

```
MPIPROF v2.x, built 01/05/17.  MPROF_PFLAG = on, CSIG = 1+2, MPROF_LIB = sgimpt
... ...
rank 0: signal 2 received
rank 1: signal 2 received
... ...
MPIPROF stats written to <lu.D_32_mpiprof_stats.out_0> ...
MPIPROF stats written to <lu.D_32_mpiprof_stats.out_1> ...
... ...
MPIPROF v2.x, built 01/05/17.  MPROF_PFLAG = cont, CSIG = 1+2, MPROF_LIB = sgimpt
```

"`MPROF_PFLAG=on`" indicates that profiling is on and internal buffers are reset; "`MPROF_PFLAG=cont`" indicates that internal buffers are not reset after statistics are written to output files. We can send a different signal(=1) to the job via:

```
% sigpg_job 299326 lu.D.32 1
```

In the job window, we will see the signal is received and data are written to outputs again. Each rank writes to its own file with multiple profiling data blocks.

7.2 Monitoring Functions

Use of [-mfunc] to monitor selected functions is straightforward as:

```
    % mpiexec -np 32 mpiprof -mfunc=wait,send bin_mpt_avx/lu.D.32
```

The command specifies two functions (MPI_Wait and MPI_Send) to be monitored. When the selected functions are called, a few statistics about timing and count will be printed for each rank. Sample printouts from the above command look like:

```
    [r301i0n1,r=24] send: count=1000, dtime=0.0000, acc-time=0.0380
    [r301i0n0,r=7] send: count=999, dtime=0.0000, acc-time=0.0421
    [r301i0n0,r=7] send: count=1000, dtime=0.0000, acc-time=0.0421
    [r301i0n1,r=31] send: count=999, dtime=0.0000, acc-time=0.0218
    [r301i0n1,r=31] send: count=1000, dtime=0.0000, acc-time=0.0218
    [r301i0n1,r=31] wait: count=9, dtime=0.0122, acc-time=0.1750
    [r301i0n1,r=30] wait: count=13, dtime=0.0027, acc-time=0.0538
    [r301i0n1,r=29] wait: count=13, dtime=0.0004, acc-time=0.0520
    [r301i0n1,r=28] wait: count=13, dtime=0.0003, acc-time=0.0606
    [r301i0n0,r=6] wait: count=15, dtime=0.0029, acc-time=0.0241
    [r301i0n1,r=22] wait: count=20, dtime=0.0000, acc-time=0.0212
```

The "count" field indicates the call instance for the selected function, "dtime" is the time spent in this call instance, and "acc-time" is the accumulated time for the selected function. One possible use of the information is to track any anomaly in message passing. For example, if dtime for one rank is much larger than for other ranks, it may indicate a slow or bad connection to the rank. Combining with the [-csig] option, one can use a signal to toggle the information printing in the monitoring process.

7.3 Debugger Attachment

MPIProf provides a hook for a debugger (such as gdb) to attach to the running process at the startup. This capability is enabled by the [-debug] option from mpiprof or the MPROF_DEBUG environment variable. To use the feature, start the job as follows:

```
    % mpiexec -np 32 mpiprof -debug bin_mpt_avx/lu.D.32
    MPIPROF v2.x, built 03/21/17.  MPROF_PFLAG = on, MPROF_LIB = sgimpt
    >>> rank 0 (r301i0n0, pid 5112) waiting for debugger to set:
        GO=1 with profiling or 2 without profiling ...
```

The job is suspended and by default rank 0 is waiting for debugger attachment, as indicated by the message. One can use [-debug=<*rank*>] to select a different rank for attachment. From a separate window, start the debugger on the host (r301i0n0) and attach to the process (pid 5112) as prompted:

```
    % gdb -p 5112
    (gdb) set var GO=1
    (gdb) cont
```

The "GO" variable is a special global variable that needs to be set within the debugger and informs MPIProf to continue execution from the point of suspension. A value of "1" indicates that profiling results will be gathered and reported; whereas a value of "2" indicates no profiling. This allows one to inspect program states, set break points, and catch abnormal run conditions (such as segmentation faults).

7.4 Post-processing Profiling Results

7.4.1 rd_stats

The utility tool `rd_stats` can be used to post-process the collected data files for:

- repacking data in unit of bytes,
- extracting a desired data set or data sections in a data set,
- producing a node message map (from rank message map),
- transposing row and column, and
- producing a summary of a data set.

This is useful for importing data into a third-part tool, such as Excel, for making graphs.

The tool automatically converts the memory usage values from a mix of binary+decimal units (KiB+KB) used in earlier versions (2.6-) of MPIProf to decimal units (KB).

Syntax of `rd_stats.x` is as follows:

```
    % rd_stats.x [-options] [-h] statfile [<mstr>]
```

where `<mstr>` is a string to match with section headers for selection of the desired data sections. The output from the tool goes to `<stdout>`. Use [`-l`] to give a brief summary of available data sets in a file, [`-g <n>`] to select a specific data group, and [`-h`] to find out other available options. As a typical use the following example selects the timing data sections in the first data set from file "`my_stats.out`" and redirects to another file:

```
    % rd_stats.x my_stats.out timing > timing_stats.out
```

Here is a sample summary output from the [`-l`] option:

```
    % rd_stats.x -l ov_mpf_def.out
    MPIPROF=v2.11 CDATE="02/27/20 08:35:44"
    HOST_INFO="r301i0n8 -- Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz"
    JOBID=8179105.pbspl1
    MPROF_EXEC=./overflowmpi
    GROUP=1 NNODES=8 NPROCS=128 NFUNCS=18
    SKEYS=6 TWALL=1014.83 OVERH=0.1108 TCOMM=165.183 TEIO=17.8121 TCOMP=849.274 \
        TIOM=17.8451
```

Besides the basic run information, the summary includes a set of timing keys (in seconds) since version 2.8:

`TWALL`	– total wallclock time
`OVERH`	– mpiprof overhead
`TCOMM`	– communication time
`TCOMP`	– compute time
`TIOM`	– maximum I/O time
`TEIO`	– effective I/O time

7.4.2 remarks

This is a stand-alone tool that takes an `mpiprof` stats file as input and produces high-level remarks about basic characteristics and load balance of timing, message and i/o volume, memory and power usage. MPIProf version 2.7 (and later) includes remarks in the stats file. The stand-alone tool is useful for stats files that were generated by earlier versions. A typical use of the tool is as follows:

```
    % remarks.x ov_mpf_def.out
```

A sample output from the command looks like:

```
Stats file: ov_mpf_def.out
   HOST_INFO="r301i0n8 -- Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz"
   JOBID=8179105.pbspl1
   MPROF_EXEC=./overflowmpi
   GROUP=1 NNODES=8 NPROCS=128 NFUNCS=18

   Wallclock time = 1014.83 secs

Remarks
   compute intensive (849.274 secs, %comp=83.69 > 80)
   compute time: imbalance clustered in 16 ranks in range 0-15
      average value in cluster 683.496, median 869.226 (spread 21.37%)
   message volume: imbalance occurred from 13 ranks
      value range 13.078G-80.569G, median 23.073G (spread 56.89%)
   i/o: from a single rank (0), time 17.8451
      rank 0, volume 18.011G
   memory usage: a single outlier in an otherwise balanced set:
      node 0, app-used 5.5586G, median 6.2706G (spread 11.35%)
      node 0, sys-used 6.4363G, median 1.8710G (spread 244.00%)
```

7.4.3 `plt_stats` and `cg_graph`

The utility tool `plt_stats` makes graphs for pre-selected data from an `mpiprof` stats file. The tool requires the pre-installation of the `ngrf` plotting package. The syntax of the tool is as follows:

```
% plt_stats [-options] [-h] statfile
```

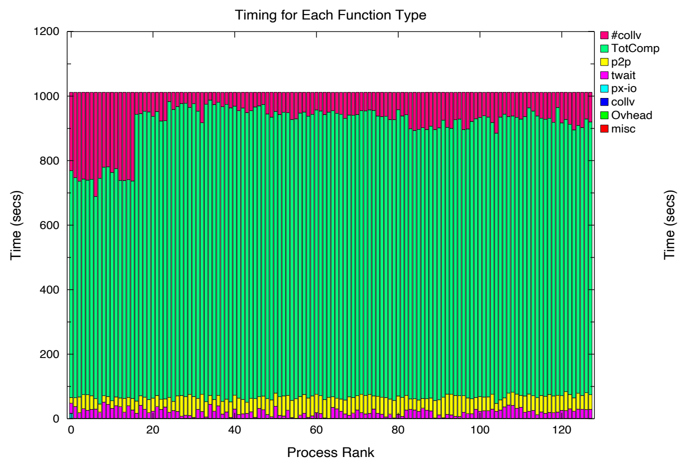
The default is to make graphs in `pdf` format for timing, message size histogram, memory and power usage. A few useful options are:

`-o <nm>`	; give a different name for the output
`-d`	; display graphs without creating pdf file
`-mem`	; plot memory and power usage only
`-prof`	; plot profiling data (timing and size histogram) only
`-g <n>`	; select a specific data group for a multi-group stats file

A typical use of the tool is as follows:

```
% plt_stats -o ov ov_mpf_cblk.out
```

This command generates an output <`ov_pstats.pdf`> for the supplied `mpiprof` stats file. The following figures (Figure 5, Figure 6 and Figure 7) illustrate a few sample graphs generated by `plt_stats` for an Overflow run with 128 ranks on 8 nodes.



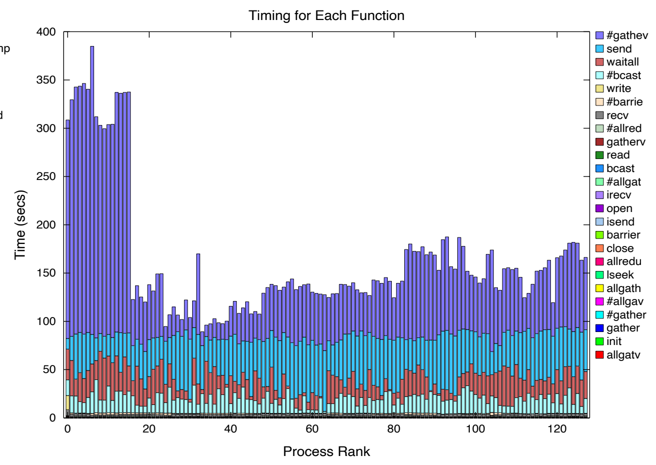
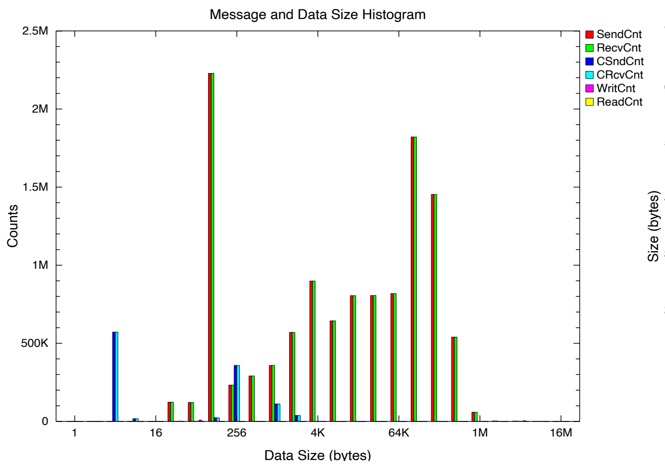


Figure 5: Sample timing graphs generated by `plt_stats`



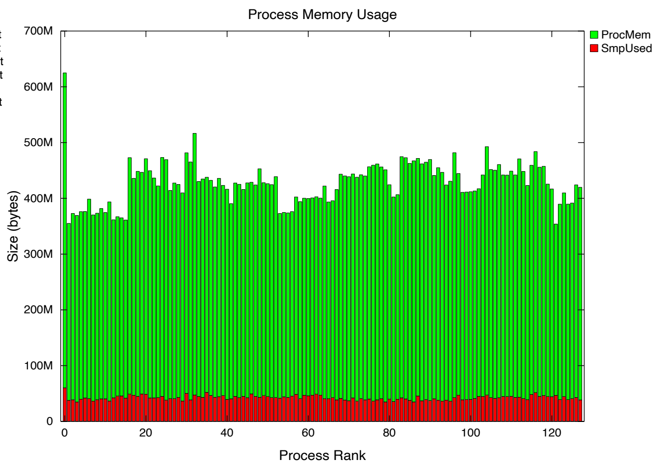
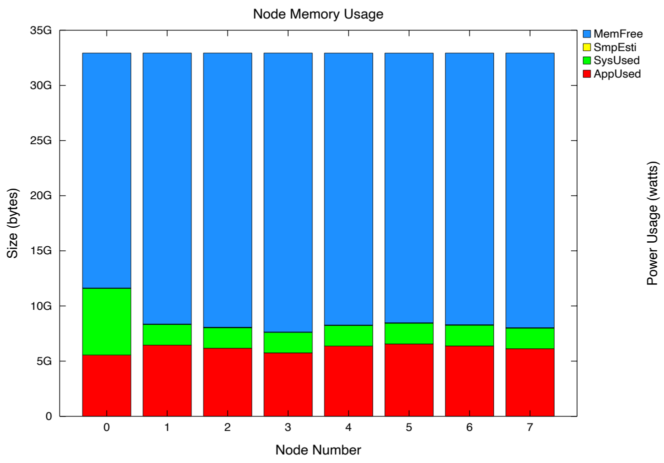


Figure 6: Sample data graphs generated by `plt_stats`



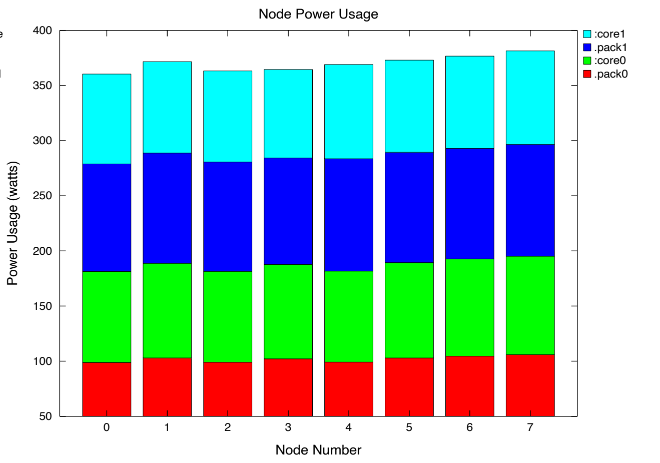


Figure 7: Sample memory and power usage graphs generated by `plt_stats`

The utility tool `cg_graph` makes graphs for call-path data in an `mpiprof` stats file generated with the `mpiprof` option [`-cpath`] or [`-cpathx`]. The tool uses the `graphviz` package. The syntax of the tool is as follows:

```
% cg_graph.x [-options] [-h] statfile
```

The default is to produce graphs in `pdf` format. Use option [`-d`] to display a graph.

7.5 Limitations and Workarounds

Following is a list of limitations in the current implementation of MPIProf and possible workarounds.

- In a multi-threaded MPI environment with `MPI_THREAD_MULTIPLE` support, by default only MPI stats from the main thread are reported. The [`-mpmt`] option to collect aggregated stats from multiple threads does not apply to callpath data and is only implemented for pthreads.
- In a multi-threaded MPI environment without `MPI_THREAD_MULTIPLE` support, only I/O stats from the main thread are reported.
- The user-defined profiling routines (mprof APIs) can only be called by a single thread.
- When using with the pagecache-management tool, the `mpiprof` command needs to be specified after the `pcachem` command; otherwise, I/O stats will not be collected.
- In order to get profiling stats for dynamic processes created via `MPI_Spawn` or `MPI_Spawn_multiple`, one needs to include `mpiprof` and possible options explicitly as part of the commands supplied to these functions.

Appendix A. Instrumented MPI and I/O Functions

The following subsections list the instrumented MPI and I/O functions and their short names used in outputs. The column "VerNo" without a value indicates MPI 2 functions. Support for a specific MPI version can be selected by setting the `MPIVERN` variable at the compile time (see Appendix B).

A.1 Instrumented MPI Functions

VerNo	ShortName	MPI Function
	allgath	MPI_Allgather
	allgatv	MPI_Allgatherv
	allredu	MPI_Allreduce
	alltoal	MPI_Alltoall
	alltoav	MPI_Alltoallv
	alltoaw	MPI_Alltoallw
	barrier	MPI_Barrier
	bcast	MPI_Bcast
	cancel	MPI_Cancel
	comm_cr	MPI_Comm_create
3.0	comm_cg	MPI_Comm_create_group
	comm_sp	MPI_Comm_split
3.0	comm_st	MPI_Comm_split_type
	cm_spwm	MPI_Comm_spawn
	cm_spmp	MPI_Comm_spawn_multiple
	exscan	MPI_Exscan
	gather	MPI_Gather
	gatherv	MPI_Gatherv
	init	MPI_Init
	inco_cr	MPI_Intercomm_create
	inco_me	MPI_Intercomm_merge
	iprobe	MPI_Iprobe
	irecv	MPI_Irecv
	isend	MPI_Isend
	ibsend	MPI_Ibsend
	irsend	MPI_Irsend
	issend	MPI_Issend
	probe	MPI_Probe
	recv	MPI_Recv
	reduce	MPI_Reduce
	reducsc	MPI_Reduce_scatter
	reducsb	MPI_Reduce_scatter_block
	scan	MPI_Scan
	scatter	MPI_Scatter
	scattev	MPI_Scatterv
	send	MPI_Send
	bsend	MPI_Bsend
	rsend	MPI_Rsend
	ssend	MPI_Ssend
	sendrev	MPI_Sendrecv
	sendrer	MPI_Sendrecv_replace
	test	MPI_Test
	testall	MPI_Testall
	testany	MPI_Testany
	testsom	MPI_Testsome
	wait	MPI_Wait
	waitall	MPI_Waitall
	waitany	MPI_Waitany

VerNo	ShortName	MPI Function
	waitsom	MPI_Waitsome
3.0	improbe	MPI_Improbe
3.0	imrecv	MPI_Imrecv
3.0	mprobe	MPI_Mprobe
3.0	mrecv	MPI_Mrecv
	send_in	MPI_Send_init
	bsend_i	MPI_Bsend_init
	ssend_i	MPI_Ssend_init
	rsend_i	MPI_Rsend_init
	recv_in	MPI_Recv_init
	start	MPI_Start
	startal	MPI_Startall
	request	MPI_Request_free
	accumul	MPI_Accumulate
	get	MPI_Get
	get_acc	MPI_Get_accumulate
	put	MPI_Put
	win_com	MPI_Win_complete
	win_cre	MPI_Win_create
	win_fen	MPI_Win_fence
	win_loc	MPI_Win_lock
	win_pos	MPI_Win_post
	win_sta	MPI_Win_start
	win_tes	MPI_Win_test
	win_unl	MPI_Win_unlock
	win_wai	MPI_Win_wait
	win_fre	MPI_Win_free
3.0	compare	MPI_Compare_and_swap
3.0	fetch_a	MPI_Fetch_and_op
3.0	raccumu	MPI_Raccumulate
3.0	rget	MPI_Rget
3.0	rget_ac	MPI_Rget_accumulate
3.0	rput	MPI_Rput
3.0	win_all	MPI_Win_allocate
3.0	win_als	MPI_Win_allocate_shared
3.0	win_crd	MPI_Win_create_dynamic
3.0	win_att	MPI_Win_attach
3.0	win_det	MPI_Win_detach
3.0	win_flu	MPI_Win_flush
3.0	win_fal	MPI_Win_flush_all
3.0	win_fll	MPI_Win_flush_local
3.0	win_fla	MPI_Win_flush_local_all
3.0	win_loa	MPI_Win_lock_all
3.0	win_una	MPI_Win_unlock_all
3.0	win_syn	MPI_Win_sync
	mopen	MPI_File_open
	mread	MPI_File_read
	mreadc	MPI_File_read_all
	mreadcb	MPI_File_read_all_begin
	mreadce	MPI_File_read_all_end
	mwrite	MPI_File_write
	mwritec	MPI_File_write_all
	mwritcb	MPI_File_write_all_begin

VerNo	ShortName	MPI Function
	mwritce	MPI_File_write_all_end
	eread	MPI_File_read_at
	ereadc	MPI_File_read_at_all
	ereadcb	MPI_File_read_at_all_begin
	ereadce	MPI_File_read_at_all_end
	ewrite	MPI_File_write_at
	ewritec	MPI_File_write_at_all
	ewritcb	MPI_File_write_at_all_begin
	ewritce	MPI_File_write_at_all_end
	sread	MPI_File_read_shared
	sreadc	MPI_File_read_ordered
	sreadcb	MPI_File_read_ordered_begin
	sreadce	MPI_File_read_ordered_end
	swrite	MPI_File_write_shared
	swritec	MPI_File_write_ordered
	swritcb	MPI_File_write_ordered_begin
	swritce	MPI_File_write_ordered_end
	iread	MPI_File_iread
	ieread	MPI_File_iread_at
	isread	MPI_File_iread_shared
3.1	ireadc	MPI_File_iread_all
3.1	iereadc	MPI_File_iread_at_all
	iwrite	MPI_File_iwrite
	iewrite	MPI_File_iwrite_at
	iswrite	MPI_File_iwrite_shared
3.1	iwritec	MPI_File_iwrite_all
3.1	iewritc	MPI_File_iwrite_at_all
	msync	MPI_File_sync
	mclose	MPI_File_close
3.0	iallgat	MPI_Iallgather
3.0	iallgav	MPI_Iallgatherv
3.0	iallred	MPI_Iallreduce
3.0	ialltoa	MPI_Ialltoall
3.0	ialltov	MPI_Ialltoallv
3.0	ibarrie	MPI_Ibarrier
3.0	ibcast	MPI_Ibcast
3.0	iexscan	MPI_Iexscan
3.0	igather	MPI_Igather
3.0	igathev	MPI_Igatherv
3.0	ireduce	MPI_Ireduce
3.0	iredusc	MPI_Ireduce_scatter
3.0	iredusb	MPI_Ireduce_scatter_block
3.0	iscan	MPI_Iscan
3.0	iscatte	MPI_Iscatter
3.0	iscattv	MPI_Iscatterv
3.0	nballga	MPI_Neighbor_allgather
3.0	nballgv	MPI_Neighbor_allgatherv
3.0	nballal	MPI_Neighbor_alltoall
3.0	nballav	MPI_Neighbor_alltoallv
3.0	nballaw	MPI_Neighbor_alltoallw
3.0	inallga	MPI_Ineighbor_allgather
3.0	inallgv	MPI_Ineighbor_allgatherv
3.0	inallal	MPI_Ineighbor_alltoall
3.0	inallav	MPI_Ineighbor_alltoallv

VerNo	ShortName	MPI Function
3.0	inallaw	MPI_Ineighbor_alltoallw

A.2 Instrumented Posix I/O Functions

ShortName	I/O Function
open	open | fopen | creat
close	close | fclose
read	read | fread | readv | pread | preadv
write	write | fwrite | writev | pwrite | pwritev
fsync	sync | fsync | fdatasync
lseek	lseek | fseek | rewind
tmpfile	tmpfile | mkstemp | mkostemp
open	open64 | creat64
read	pread64 | preadv64
write	pwrite64 | pwritev64
lseek	lseek64
tmpfile	tmpfile64 | mkstemp64 | mkostemp64

Appendix B. Installation of MPIProf

The following are a few steps of building and installing MPIProf.

B.1 Pre-requisites

In order to build MPIProf, the following are prerequisites:
- – C and Fortran compilers (such as GCC, Intel compiler)
- – GNU make
- – MPI library (such as `MPICH`, `Open-MPI`)
- – the `ulib` package (v1.7+, supplied separately)

The following packages are optional:
- – the `libunwind` package (for collecting call-path information)
- – the `CUDA` development toolkit (for reporting GPU information)
- – the `pstat` tool (used by `sigpg` and `sigpg_job`)
- – the `ngrf` plotting package (used by `plt_stats`)
- – the `graphviz` package (used by `cp_graph`)

B.2 Build and Install

The basic steps for building MPIProf:

- – Unpack the `mpiprof` tar file in the same directory where the `ulib` package was unpacked and built
- – Load proper compiler and MPI modules if needed
- – In directory `mpiprof/makefiles`, take one of the files as a template and modify the parameters in the file as necessary, such as compilers and installation directory:

```
CC = mpicc              # C compiler for MPI codes
CFLAGS = -O2            # C flags
FC = mpif90             # Fortran compiler for MPI codes
FFLAGS = -O2            # Fortran flags
LDFLAGS =               # additional ld flags if any
MPILIB = mpich          # name of the MPI library <mpilib>
UFC = gfortran          # Fortran compiler used for creating shared libraries

INSTDIR = $(HOME)/Tools/mpiprof           # installation directory
UNWIND_DIR = $(HOME)/Tools/libunwind      # location of libunwind

BACKTRACE = 1           # 1 (def) for libunwind, 0 for no libunwind
NOFOR = 0               # 0 (def) for Fortran binding, 1 for Fortran via C binding
GETTID = 0              # 0 (def) for no gettid check, 1 for gettid check
MPIVERN = 3.1           # 3.1 (def), other possible values: 3.0, 2.0
TIMEOFDAY = 0           # 0 (def) for clock_gettime, 1 for gettimeofday
GINFO = 1               # 1 (def) for GPU info, 0 for no GPU info
```

- – In the top directory of `mpiprof`, type

```
% make mpich            # create Makefiles, assuming "Makefile.mpich"
                        #   was the template used in the previous step
% make all              # build the package
```

Installation of files:

– After building, use the following
```
% make install
```
– Files installed include `mpiprof` library, tools, documents, and module

– By default, the `sigpg` tool (a script) is also installed in `$INSTDIR/bin`. This tool uses "`pstat.x`". Make sure the "`bindir`" variable points to the correct directory where "`pstat.x`" is located.

– The `plt_stats` tool (a script) contains a variable "`pgmdir`" that points to a directory where the plotting tool "`xplot`" (from the `ngrf` package) is located. By default, `pgmdir` is set to `$INSTDIR/../ngrf/bin`.

To build and install for multiple MPI libraries, check the "`comp-lib*`" scripts in the `misc` directory. For a detailed example, refer to "`build-pld.txt`" in the `doc` directory.

To configure for multiple versions of an MPI implementation, create `$INSTDIR/lib/mprof_lib.conf` from the supplied template in the `misc` directory. An example of `mprof_lib.conf` contains

```
#mprof_lib, VCmd,          VerStr,    VerNo,  mprof_lib2
mvapich,    mpichversion,  Version:,  2.1+,   mvapich-2.1
openmpi,    ompi_info,     Open MPI:, 1.8+,   openmpi-1.8
```

The first column is the name of a supported MPI library, the second column is the tool used to obtain the version information of an MPI library, the third column is a unique string that marks the version number, the fourth column is the intended version number with an optional "+" (beyond) or "-" (up to) sign, and the last column is the corresponding `mpiprof` library name to be used for the intended version. For any name of supported MPI libraries not appeared in the configure file, the default version will be used.

B.3 Test and Use

To use `mpiprof`:

– Load proper compiler and MPI modules

– Load the `mpiprof` module
```
% module use -a $INSTDIR
% module load mpiprof-module
```
– To build the supplied test cases, in the top directory of `mpiprof`
```
% make tests
```
– To run the test cases, in the `tests` directory
```
% ./runit
```

Appendix C. MPIProf and MPI Library Compatibility

The following table contains compatibility information between different versions of MPIProf and MPI libraries in reference to the build process of MPIProf as shown in Appendix B. A "+" sign following an MPI library version indicates the upper version support of the MPI library for the same MPI Version by the corresponding `mproflib`. For example, `mpich-3.2` has been tested to work with MPICH-3.2 and MPICH-3.3, both of which implements MPI 3.1.

Table 11: Compatibility summary between different versions of MPIProf and MPI libraries

Name (`mproflib`)	**MPI Library**	**MPI Version** (`MPIVERN`)	**C/Fortran Interface** (`NOFOR`)	**Other**
`mpich`	MPICH-3.0	3.0	Both (0)	
`mpich`	MPICH-3.1	3.0	Both (0)	
`mpich-3.2`	MPICH-3.2+	3.1	C only (1)	
`openmpi`	OpenMPI-1.8	2.0	C only (1)	
`openmpi-1.10`	OpenMPI-1.10	3.0	C only (1)	
`openmpi-2.1`	OpenMPI-2.1	3.1	Both (0)	
`openmpi-3.0`	OpenMPI-3.0+	3.1	Both (0)	
`hpcx-2.4`	HPCX-2.4+	3.1	Both (0)	
`sgimpt`	SGIMPT-2.12	3.0	Both (0)	
`sgimpt-2.14`	SGIMPT-2.14	3.1	Both (0)	
`hpempt`	HPEMPT-2.16	3.1	Both (0)	
`hpempt-2.20`	HPEMPT-2.20	3.1	Both (0)	
`hpempt-2.21`	HPEMPT-2.21+	3.1	Both (0)	
`intelmpi`	IntelMPI-5.0	3.0	Both (0)	
`intelmpi`	IntelMPI-5.1	3.0	Both (0)	
`intelmpi-2017`	IntelMPI-2017+	3.1	C only (1)	
`mvapich`	MVAPICH2-2.0	3.0	Both (0)	`BACKTRACE=0`
`mvapich-2.1`	MVAPICH2-2.1	3.0	C only (1)	`BACKTRACE=0`
`mvapich-2.2`	MVAPICH2-2.2	3.1	C only (1)	`BACKTRACE=0`
`ser`	n/a	n/a	n/a	

Appendix D. Version History

Below is a brief revision history of different MPIProf versions. See doc/Changes.log for detailed changes.

Version 2.15

Added the [-cnuma] option to support process-cpu binding at the numa node level and the [-nsum] option to report summary without per-node stats. Slightly reorganized the memory summary report. Fixed potential bad return values in implementing a number of instrumented MPI functions (MPI_Iprobe, MPI_Improbe, MPI_Test*).

Version 2.14

Support the report of GPU memory and power usage information. The GPU usage report can be disabled by mpiprof option [-gn]. Included function call counts in the call-path report and added a new utility tool "cp_graph" to graph call-path data. If an output file starts with a "%" sign, the output filename will be suffixed with a number (*nnnn*) to avoid overwriting.

Version 2.13

Added two more MPI routines in the profiling list (MPI_Comm_create_group, MPI_Comm_split_type). Support the access of mprof routines via the MPI_Pcontrol interface. Added new environment variable MPROF_CTAG to define ctag for use with the interface. Updated interface support files for mprof Fortran APIs so that the inclusion of "mprof_flib.h" in source codes is now optional.

Version 2.12

Added function MPI_Cancel and check cancelled requests for async calls. Report the compiled MPROF_LIB value for the selected mpiprof library. Include process-binding information in the stats output when the mpiprof option [-v] is used in conjunction with any of the mbind options. Fix issues in message handle matching for processing MPI_Mprobe/MPI_Mrecv in C and in tracking twait time for P2P functions. Added new built option TIMEOFDAY(=0/1) to select different timers.

Version 2.11

Report separate histogram stats for collective message size as CSndCnt/CRcvCnt. Added a new function type TWAIT and reclassified the function type of test-wait routines (MPI_Test*, MPI_Wait*) from P2P to TWAIT to reflect the fact that these functions also support nonblocking communication besides point-to-point type. The output includes a "twait" column in the summary section of the stats. The rd_stats tool now returns proper exit codes for use with plt_stats and has better support for help information.

Version 2.10

Added more MPI routines in the profiling list (MPI_Comm_spawn, MPI_Comm_spawn_multiple, MPI_Win_allocate, MPI_Win_allocate_shared, MPI_Win_attach, MPI_Win_dettach). Properly count messages for MPI calls that involve MPI_PROC_NULL and MPI_IN_PLACE as arguments. Fix in calculation of standard deviation in both stats report and remarks. Support for MPI 3.1 is now the default.

Version 2.9

Added support for data collection in a multi-threaded environment. Added the option [-mpmt] in mpiprof and the value "mpmt" for the environment variable MPROF_PFLAG to enable the multi-threaded mode. Separation of i/o functions from MPI functions in the profiling library to allow more modular handling of function instrumentation. Added option [-xio] in mpiprof to disable loading of and flag "-lmprof_io" for mprof APIs to link with i/o profiling library. Added (the missing) routine MPI_Get_accumulate in profiling. Report the largest time-consuming communication and i/o functions in remarks. The timing of open/close calls is included as part of the "read/write" stats in the summary. The tool rd_stats can now produce a node message map with option [-nmap].

Version 2.8

Added a new mprof API mprof_init2 for initialization of profiling environment with a tag. Included neighborhood collective communication routines and their non-blocking versions (MPI_Neighbor_allgather*, MPI_Neighbor_alltoall*, MPI_Ineighbor_allgather*, MPI_Ineighbor_alltoall*) in profiling. Improvement in the instrumented function generator for better code maintenance.

Version 2.7

Included high-level remarks about profiling data in the summary report and reorganized the summary. A stand-alone tool, remarks, was added to produce high-level remarks about profiling data in a stats file. Improvement in process binding for irregular rank distributions. Option [-mbind] added to perform process binding without profiling. Use a more consistent unit for memory values. Updated the graph tool plt_stats to generate a single set of plots for profiling data and memory usage.

Version 2.6

Integrated with mbind.x for process binding. The binding action is triggered by specifying any of the supported mbind options in the mpiprof command line. Attempt to correct memory over-counting due to shared memory pages among processes. Rearranged the report of system memory usage.

Version 2.5

Support the report of per-application power usage through the kernel /sys/class/powercap interface. Use "+-pwr" pflag value to enable or disable this functionality. Provided a way to reset power sampling interval. Included a standalone utility "pwr.x" to monitor power usage.
Added a few more I/O functions in profiling: readv, writev, preadv, pwritev, lseek, fseek, rewind.

Version 2.4

Support for debugger attachment at the startup via the [-debug] option from mpiprof. Utility rd_stats was improved for reading and writing multiple data sets. Included four more MPI functions (MPI_Comm_create, MPI_Comm_split, MPI_Intercomm_create, MPI_Intercomm_merge) in profiling.

Version 2.3

Added a new API "mprof_pcheck()" for user-defined profiling to save the profiling data in memory and re-initialize the profiling environment as a check-point capability. "mprof_end()" writes out the saved data sets as well as the aggregated results. Utility "rd_stats" was updated to process multiple data sets.

Version 2.2

Separate the report of stats for Posix I/O calls made from collective and non-collective MPI I/O. Report maximum wall clock time as well as average wall clock time. A utility "rd_stats" was added to reformat output data for selected entries. Improved performance and reduced overhead in dealing with a large number of message requests for non-blocking calls.

Version 2.1

Use the -mfunc option in conjunction with -cblk to select collective functions for blocking time measurement. The -cblk option now applies to collective file I/O functions as well. Count file close and file sync as part of write time in calculating I/O rates.

Version 2.0

Included more MPI functions from MPI 3.0 (matched probe/recv, request-based one-sided comm.) and 3.1 (nonblocking collective I/O). Implemented selectable support for different MPI releases. Fixed a major issue with measuring blocking time for point-to-point calls. Report per-function blocking time. Added the option "sum" to report summary information without per-rank details. Function types are marked with more sensible letters over the previously numerical values.

Version 1.9

The option "`ios`" to report I/O stats now includes MPI I/O as well. Use the new option [`-ios_nompi`] for `mpiprof` to exclude MPI functions in the `ios` report. Separate MPI I/O count histogram from Posix I/O count histogram. Support auto-detecting multi-versions of MPI libraries.

Version 1.8

Added new option "`ios`" to report I/O stats only. Improvement in auto-detecting MPI libraries. Trace additional I/O functions for temp file creation (`mkstemp`, `mkostemp`, `tmpfile`, etc.).

Version 1.7

Support for serial codes via the "`ser`" library type, useful for I/O profiling and memory usage report. Added more missing I/O functions (`open64`, `creat64`, `pread`, `pwrite`, `pread64`, `pwrite64`, `fdatasync`) in the profiling list. Sort functions by time in the per-function summary section for better presentation.

Version 1.6

Added three more functions (`MPI_Sendrecv_replace`, `fopen`, `fclose`) in the profiling list. Use `MPI_Is_thread_main()` to resolve multi-thread issues. Reduce the `mpiprof` overhead by making send-recv hand-shake acknowledgement optional (via the "`ack`" `pflag` value). Report time for p2p calls in the message map printout. Catch `popen` issue similar to `system`. Fix an issue for setting the default `pflag` when encountering a bad value. Included [`--io`] option to disable Posix I/O profiling.

Version 1.5

Report MPIProf setup time (mainly from `mprof_init`) and a few call path tree stats. The stack unwinding depth can now be dynamically changed via `MPROF_CPATH_DEPTH`. Mark critical path along call path. Fix an issue with recursive call path. Fix potential output buffer overflow. Workaround added for avoiding race in multi-threaded environment. The `mpiprof` [`-lib`] option accepts names other than the built-in list. Estimate and report MPIProf overhead. Report effective I/O time based on I/O rate.

Version 1.4

Support for non-blocking collectives from MPI 3. Match messages in `MPI_Wait/Test` with requests from original functions. Separate non-blocking MPI I/O time from `MPI_Wait/Test` functions for better rate estimation. Report time spent in `mprof_end`. Clean up make and install process.

Version 1.3

The default now includes profiling several Posix I/O functions. Added a check to avoid double counting of each function. Report node memory usage. New option [`-mem`] to report memory usage only.

Version 1.2

Reorganized the output content to include per-rank summary. Report process memory usage (HWM) in the summary. Separate report of I/O from communication, and added `io`=factor experiment.

Version 1.1

Support for catching a signal and writing results to individual output files for each rank (via `-csig` or `MPROF_CSIG`). New scripts "`sigpg`" and "`sigpg_job`" to send a signal to a process group. Added a new `pflag` value `MPF_CONT` to `mprof_init()` so that the previous values will not be cleared when this function is called.

Version 1.0

Support the monitor of a specific function at each call instance with option [`-mfunc`] in `mpiprof` and environment variable `MPROF_MFUNC`. Update `mpiprof` to use dynamic help content for a list of <*`mpilib`*>. Support multi-segment instrumentation in user-defined profiling.