The Intel Application Performance Snapshot (APS) tool provides a quick overview of your application's performance on processor and memory usage, message passing interface (MPI), and I/O, as well as load imbalance between threads or processes. In addition to the performance metrics listed below, APS also provides suggestions for performance enhancement opportunities and additional Intel profiling tools you can use to get more in-depth analysis.

Note: Although the Intel APS tool can be used for any of the Aitken and Electra Intel Xeon processor types, it does not fully support the Sandy Bridge and Haswell processor types. APS cannot be used for the Aitken Rome and Milan processors.

Before You Begin

Note the following information:

- The Intel C/C++ or Fortran Compiler is not required, but is recommended. However, analysis of OpenMP imbalance is only available for applications that use the Intel OpenMP runtime library.
- There is no support for OpenMPI.
- Analysis of MPI imbalance is only available when you are using the Intel MPI Library version 2017 and later. (With the exception of the MPI imbalance metric, APS MPI analysis works for applications that use the NAS-recommended HPE MPT library.)

TIP: The NAS-developed MPIProf tool provides similar MPI analysis with MPI imbalance information included.

Running an APS Analysis

On NAS Intel x86_64 systems, APS is available via the vtune/2021.9 module. See aps --help for available APS options.

To run an analysis for serial and OpenMP applications:

```
module load vtune/2021.9
aps <options> ./a.out
```

To run an analysis for MPI applications:

```
module load vtune/2021.9

module load mpi-hpe

setenv MPI_SHEPHERD true
setenv MPI_USING_VTUNE true
mpiexec -np xx aps <options> ./a.out
```

TIP: The process/thread pinning tools mbind.x and omplace can be used with aps. Either of the following two methods may work:

- mpiexec -np xx aps mbind (or omplace) <mbind or omplace *options*> a.out
- mpiexec -np xx mbind (or omplace) <mbind or omplace *options*> aps <*options*> a.out

At the end of the run, APS provides a directory that contains the data collected during the analysis. The default name of the directory is aps_result_*yyyymmdd*.

WARNING: The directory will be overwritten if you run another APS analysis in the same directory on the same day.

To send the results to another directory, you can include --result-dir=*dir_name* or -r=*dir_name* option in the aps command line (where *dir_name* represents the directory name of your choice).

Viewing the Report

APS provides a text summary and an HTML file named aps_report_*yyyymmdd_hhmmss*.html (where *yyyymmdd_hhmmss* is the date and time when the HTML file is created). The HTML file contains the same information as the text summary, but also offers the following features when you open the file with a web browser:

- A description of each metric is shown when you hover your mouse over the metric name.
- Possible performance issues of your run are highlighted in red.
- Suggestions with links to Intel tools are provided to help with performance enhancement.

Note: NAS security policy prohibits using web browsers on NAS high-end computing systems. To view the HTML file, transfer it to your own workstation.

For Serial and OpenMP Applications

For serial or OpenMP applications, no action is required to generate the reports. At the end of the run, APS automatically generates the text summary (appended to your application's stdout) and the HTML file.

For MPI Applications

For MPI applications, you must generate the text summary (shown on your screen) and HTML file after the analysis completes, as follows:

aps --report=*dir_name*

where *dir_name* is the name of the directory created at the end of the analysis run.

If the summary report shows that your application is MPI-bound, run the following command to get more details about message passing, such as message sizes, data transfers between ranks or nodes, and time in collective operations:

aps-report <*options*> *dir_name*

See aps-report --help for available options.

You can also rerun the analysis after setting additional environmental variables. For example:

setenv MPS_STAT_LEVEL *n*
or
setenv APS_STAT_LEVEL *n*

where *n* is 2, 3 or 4 to get more detailed information on your application's MPI performance.

Note: If MPS_STAT_LEVEL (or APS_STAT_LEVEL) is set to 3 or 4, you must use mpi-hpe/mpt.2.18r160 or a later version when you run the analysis to avoid a segmentation fault (SEGV).

Quick Metrics Reference

Intel APS collects the metrics described in the following list.

Note: These descriptions are adapted from the Application Performance Snapshot User's Guide for Linux OS, where you can find additional details about each metric.

Elapsed Time
: Execution time of specified application in seconds.

SP GFLOPS
: Number of single precision giga-floating point operations (gigaflops) calculated per second. SP GFLOPS metrics are only available for 3rd Generation Intel Core processors, 5th Generation Intel processors, and 6th Generation Intel processors.

DP GFLOPS
: Number of double precision giga-floating point operations calculated per second. DP GFLOPS metrics are only available for 3rd Generation Intel Core processors, 5th Generation Intel processors, and 6th Generation Intel processors.

Cycles per Instruction Retired (CPI) Rate
: The amount of time each executed instruction took measured by cycles. A CPI of 1 is considered acceptable for high performance computing (HPC) applications, but different application domains will have varied expected values. The CPI value tends to be greater when there is long-latency memory, floating-point, or SIMD operations, non-retired instructions due to branch mis-predictions, or instruction starvation at the front end.

CPU Utilization
: Helps evaluate the parallel efficiency of your application. Estimates the utilization of all the logical CPU cores in the system by your application. 100% utilization means that your application keeps all the logical CPU cores busy for the entire time that it runs. Note that the metric does not distinguish between useful application work and the time that is spent in parallel runtimes.

MPI Time
: Time spent inside the MPI library. Values more than 15% might need additional exploration on MPI communication efficiency. This might be caused by high wait times inside the library, active communications, non-optimal settings of the MPI library. See MPI Imbalance metric to see if the application has load balancing problem.

MPI Imbalance
: Mean unproductive wait time per process spent in the MPI library calls when a process is waiting for data.

Serial Time
: Time spent by the application outside any OpenMP region in the master thread during collection. This directly impacts application Collection Time and scaling. High values might signal a performance problem to be solved via code parallelization or algorithm tuning.

OpenMP Imbalance
: Indicates the percentage of elapsed time that your application wastes at OpenMP synchronization barriers because of load imbalance.

Memory Stalls
: Indicates how memory subsystem issues affect the performance. Measures a fraction of slots where pipeline could be stalled due to demand load or store instructions. This metric's value can indicate that a significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. See the second level metrics to define if the application is cache- or DRAM-bound and the NUMA efficiency.

Cache Stalls
: indicates how often the machine was stalled on L1, L2, and L3 cache. While cache hits are serviced much more quickly than hits in DRAM, they can still incur a significant performance penalty. This metric also includes coherence penalties for shared data.

DRAM Stalls
: Indicates how often the CPU was stalled on the main memory (DRAM) because of demand loads or stores.

DRAM Bandwidth
: The metrics in this section indicate the extent of high DRAM bandwidth utilization by the system during elapsed time. They include:

  - **Average Bandwidth:** Average memory bandwidth used by the system during elapsed time.
  - **Peak:** Maximum memory bandwidth used by the system during elapsed time.

- **Bound:** The portion of elapsed time during which the utilization of memory bandwidth was above a 70% threshold value of the theoretical maximum memory bandwidth for the platform.

Some applications can execute in phases that use memory bandwidth in a non-uniform manner. For example, an application that has an initialization phase may use more memory bandwidth initially. Use these metrics to identify how the application uses memory through the duration of execution.

NUMA: % of Remote Accesses

In non-uniform memory architecture (NUMA) machines, memory requests missing last level cache may be serviced either by local or remote DRAM. Memory requests to remote DRAM incur much greater latencies than those to local DRAM. It is recommended to keep as much frequently accessed data local as possible. This metric indicates the percentage of remote accesses, the lower the better.

Vectorization

The percentage of packed (vectorized) floating point operations. The higher the value, the bigger the vectorized portion of the code. This metric does not account for the actual vector length used for executing vector instructions. As a result, if the code is fully vectorized, but uses a legacy instruction set that only utilizes a half of the vector length, the Vectorization metric is still equal to 100%.

Instruction Mix

This section contains the breakdown of micro-operations by single precision (SP FLOPs) and double precision (DP FLOPs) floating point and non-floating point (non-FP) operations. SP and DP FLOPs contain next level metrics that enable you to estimate the fractions of packed and scalar operations. Packed operations can be analyzed by the vector length (128, 256, 512-bit) used in the application.

- **FP Arith/Mem Rd Instr. Ratio:** This metric represents the ratio between arithmetic floating point instructions and memory read instructions. A value less than 0.5 might indicate unaligned data access for vector operations, which can negatively impact the performance of vector instruction execution.
- **FP Arith/Mem Wr Instr. Ratio:** This metric represents the ratio between arithmetic floating point instructions and memory write instructions. A value less than 0.5 might indicate unaligned data access for vector operations, which can negatively impact the performance of vector instruction execution. The metric value might indicate unaligned access to data for vector operations.

Additional Resources

- Getting Started with Application Performance Snapshot - Linux OS
- Application Performance Snapshot User's Guide for LinuxOS

You can use Intel Advisor to identify issues and help you with threading and vectorization optimization on the specific Intel processors that your Fortran, C, or C++ applications run on. The software, which is available as a module on HECC systems, provides recommended workflows for these two optimization areas.

Intel Advisor offers both a command-line interface (CLI) and a graphical user interface (GUI). We recommend that you use the CLI (advixe-cl) in a PBS batch job to collect data, and then use the GUI (advixe-gui) to analyze the result, as it provides a more comprehensive view than the CLI. However, please note that learning how to navigate the GUI will take some effort and time. For more information, see the Intel documentation, Intel Advisor GUI.

Analysis Types

Each Intel Advisor workflow involves a combination of some of the analysis types listed below. For detailed information about these workflows, see the Intel documentation, Getting Started with Intel Advisor, or access the CLI help page by running advixe-cl -h workflow.

Survey analysis
: Helps to identify loop hotspots and provides recommendations for how to fix vectorization issues.

Trip counts and flops analysis
: Counts the number of times loops are executed and provides data about floating-point operations (flops), memory traffic, and AVX-512 mask usage. This analysis requires more instrumentation and will perturb your application more. Thus, it takes longer time than the survey analysis.

Dependencies analysis
: Checks for real data dependencies in loops the compiler did not vectorize because of assumed dependencies.

Memory access patterns analysis
: Checks for memory access issues such as non-contiguous, or non-unit stride access.

Suitability analysis
: Predicts the maximum speed-up of your application, based on the inserted annotations and a variety of what-if modeling parameters you can experiment with. Use this information to choose the best candidates for parallelization with threads.

Roofline analysis
: Runs two analyses one by one: (1) the survey analysis, and (2) the trip counts and flops analysis. A roofline chart is then generated which plots an application's achieved floating-point performance and arithmetic intensity (AI) against the machine's maximum achievable performance.

  Note: For more information about running this analysis, see Roofline Analysis with Intel Advisor.

Compiling Your Code

At the minimum, you should include the following options when you compile your code:

```
-g -O2 (or higher)
-g -O2 (or higher) -qopenmp  (for OpenMP applications)
```

Depending on the processors your application runs on, you may consider adding an option that instructs the compiler to generate a binary that targets certain instruction sets. The options are listed in the following table.

Processor Type	Option
Cascade Lake, Skylake	-xCORE-AVX512
Broadwell	-xCORE-AVX2
Multiple auto-dispatch code paths	-axCORE-AVX512,CORE-AVX2,AVX -xSSE4.2

Collecting Data

Load the module and collect data:

```
module use -a /nasa/modulefiles/testing
module load advisor/
advixe-cl -collect=<string> [-action-option] [-global-option] [--] <target> [<target options>]
```

where:

- <string> is one of the following analysis types: survey, tripcounts, dependencies, map, or suitability
  Note: For non-MPI applications, <string> can also be roofline. For MPI applications, roofline analysis can only be done by performing survey and tripcounts in two separate steps.
- <target> is your executable.

For example:

```
advixe-cl -collect survey -project-dir my_result -- ./a.out
```

For information about action-option and global-option, run advixe-cl -h collect.

Analyzing the Result

The data collected are stored in a directory tree (for example, my_result), which you specify using an action-option called -project-dir.

Depending on the analysis type used in data collection, you might find the subdirectories hsxxx, trcxxx, dpxxx, and mpxxx in the my_result/e000 directory (for serial or pure OpenMP applications) or in the my_result/rank.n directory (for the nth-rank of an MPI application).

To view the result on a PFE, use either the Intel Advisor GUI (advixe-gui) or the CLI (advixe-cl).

Use the Intel Advisor GUI

Load the module and run advixe-gui:

```
module use -a /nasa/modulefiles/testing
module load advisor/
advixe-gui my_result
```

After the GUI starts, click **Show My Result**. In the main window, you can choose one of three panels: **Summary** (default view), **Survey & Roofline**, or **Refinement Reports**. All three are described below.

Summary
: Might show gigaflops count, AI, number of threads, loop metrics, vectorization gain/efficiency, top time-consuming loops, and platform information.

Survey & Roofline
: Might show the roofline chart, source code, assembly code, detailed info on flops, AI, efficiency, reason for no vectorization of the top-time-consuming loops.

Refinement Reports
: Might show memory access patterns report (1-stride, 0-stride, constant-stide, irregular stride) and dependencies report.

Use the Intel Advisor CLI

Load the module and run advixe-cl:

```
module use -a /nasa/modulefiles/testing
module load advisor/
advixe-cl -report=<string> [-action-option] [-global-option] [--] <target> [<target options>]
```

For information on <string>, action-option, and global-option, run: advixe-cl -h report

Note: If the job runs too quickly, there is not enough data collected to generate a report.

Additional Resources

- Getting Started with Intel Advisor
- Intel Advisor CLI
- Intel Advisor GUI
- Intel Advisor presentation by Intel engineer Jackson Marusarz
- Transforming Serial Code to Parallel Using Threading and Vectorization Part 1 and Part 2.

A roofline analysis helps you determine whether your application has achieved the best possible performance, limited by the machine capabilities. If not, you can explore the possibility of algorithm changes to improve performance.

Running the Analysis

The Intel Advisor tool is available as a module on HECC resources. To do a roofline analysis, load the module and run two analyses: survey and tripcounts, as follows:

```
module use /nasa/modulefiles/testing
module load mpi-hpe/mpt.2.18r160
module load advisor/2018
mpiexec -np x advixe-cl -collect survey -project-dir my_result -- ./a.out
mpiexec -np x advixe-cl -collect tripcounts -flop -project-dir my_result -- ./a.out
```

Note: Be sure to specify the same directory (*-project-dir*) for the survey and tripcounts analyses. The tripcounts run will take longer time than the survey run.

If these two runs are successful, you can start the Intel Advisor GUI and choose one of the ranks to analyze.

Reviewing the Results

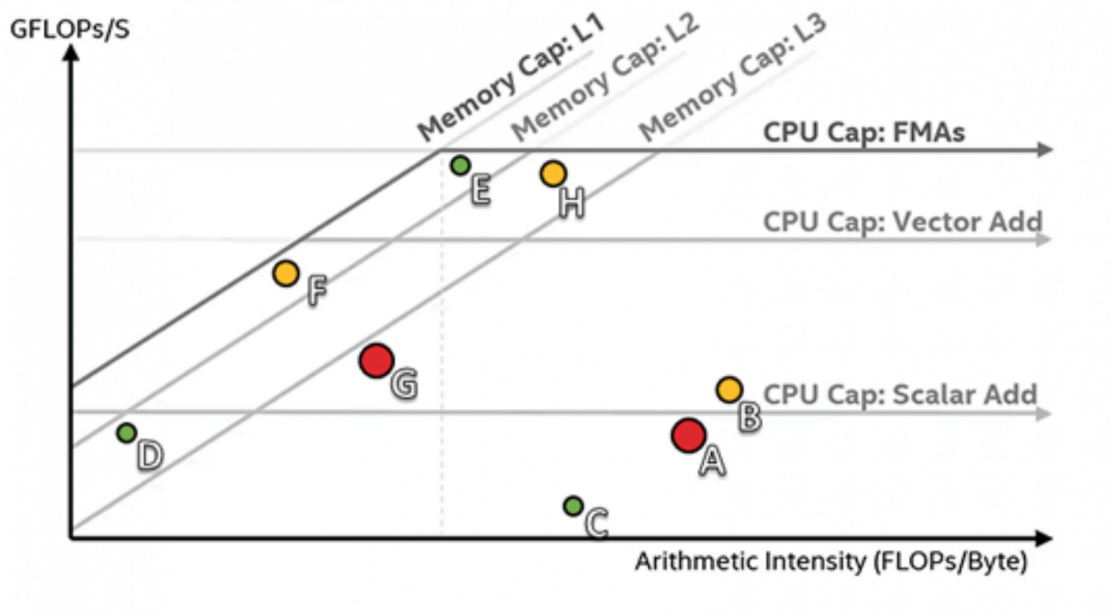
To start the GUI:

```
advixe-gui my_result
```

To see a roofline chart for the rank chosen, do one of the following:

- In the navigation panel, click the black icon (⚑) under **Run Roofline**.
- Click the **Survey & Roofline** panel in the main window, then click the vertical bar labeled **ROOFLINE**.

In the sample roofline chart shown below, the X axis is arithmetic intensity (measured in flops/byte) and the Y axis is the performance in Gflops/second, both in logarithmic scale:



Note: This sample chart is reproduced from the Intel documentation, Getting Started with Intel Advisor.

Before it collects data on your application, Intel Advisor runs benchmarks to measure the hardware limitations of your machine. It plots these on the chart as lines, called *roofs*. The horizontal lines represent the number of floating point computations of a given type your hardware can perform in a given span of time. The diagonal lines are representative of how many bytes of data a given memory subsystem can deliver per second.

In the chart, each dot is a loop or function in your application. The dot's position indicates the performance of the loop or function, which is affected by its optimization and its arithmetic intensity. The dot's size and color indicate how much of the total application time the loop or function takes. In the sample chart shown above, loops A and G (large red dots), and to a lesser extent loop B (yellow dot far below the roofs), are the best candidates for optimization. Loops C, D, and E (small green dots) and H (yellow dot) are poor candidates because they do not have much room to improve.

Important: Intel Advisor uses a cache-aware roofline model. In the classic roofline model, a kernel's arithmetic intensity would change with problem size or cache usage optimization, because the byte count was based on DRAM traffic only. This is not the case in the cache-aware roofline model, where arithmetic intensity is a fixed value tied to the algorithm itself; it only changes when the

algorithm is altered, either by the programmer or occasionally by the compiler.

Additional Resources

- Getting Started with Intel Advisor
- Intel Advisor Roofline
- Getting Started with Intel Advisor Roofline Feature
- Intel video: Introduction to Intel Advisor Roofline Feature
- Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architecture, 2009 (PDF)
- Cache-aware Roofline Model: Upgrading the Loft, 2013 (PDF)

MPIProf is a lightweight, profile-based application performance analysis tool that works with many MPI implementations, including HPE MPT. The tool gathers statistics in a counting mode via PMPI, the MPI standard profiling interface.

This tool is available for either the NAS x86_64 or aarch64 systems.

MPIProf can report profiling information about:

- Collective and/or point-to-point MPI functions called by an application (time spent, number of calls, and message size)
- MPI and POSIX I/O statistics
- Memory used by processes on each node
- Memory and power usage by applications
- Call path information of MPI calls (requires the -g compiler option and the -cpath mpiprof option)

In addition, mpiprof incorporates the mbind process binding capabilities through its various options. For more information about the options, read mpiprof -h after loading the scicon/mpiprof module.

There are two ways you can use MPIProf:

- The mpiprof profiling tool on the command line:

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

- MPIProf API routines

mpiprof Usage Example

Because the command-line method does not require changing or recompiling your application, we recommend this method for general use. To run mpiprof, load the modulefiles for your compiler, MPI library, and the mpiprof tool, then run the command line. For example:

```
module load comp-intel mpi-hpe/mpt
module load scicon/mpiprof
mpiexec -n 128 mpiprof -o mpiprof.out a.out
```

Note: If the -o option is not included, the profiling results are written to the mpiprof_stats.out file, or the <a.out>_mpiprof_stats.out file if the executable name is known.

For more details about using this tool, see the MPIProf user guide on a NAS system:

/u/scicon/tools/opt/mpiprof/default/doc/mpiprof_userguide.pdf

MPIProf was developed by NAS staff member Henry Jin.

MPInside, an MPI application performance analysis and diagnostic tool from HPE, is a lightweight program that can be used with thousands of ranks. The tool is profile-based via PMPI, the MPI standard profiling interface. An MPInside analysis can provide you with communication statistics (collective and point-to-point) as well as information that can help you diagnose communication issues such as:

- Slow MPI communications caused by non-synchronous send/receive pairs
- Load imbalance (compute time or message bandwidth) among ranks
- Reasons a code runs well with one MPI library or one platform, but not another

In addition to MPI communication profiling, MPInside provides the following optional features:

- MPI performance modeling
- I/O measurement

MPInside supports the full MPI 2.2 standard and the commonly used MPI libraries: HPE MPT, Intel MPI, and OpenMPI.

Note: Some features may be limited to HPE MPT.

Basic Usage

Most features of MPInside do not require instrumentation, recompiling, or relinking of your application. However, using the `-g` compiler flag is recommended.

The basic MPInside measurement produces a flat profile. No information is provided about the source lines that make the MPI calls.

Environment Setup

To set up your environment to use MPInside, load the MPInside module and an MPI library module. Set the `MPINSIDE_LIB` environment variable to match the MPI library module. For example:

```
module load MPInside/3.6.2

module load mpi-hpe/mpt
setenv MPINSIDE_LIB MPT (or IMPI, or OPENMPI)
```

Note: `MPINSIDE_LIB` defaults to `MPT`.

Communication Profiling

To perform an analysis using MPInside, run the following commands (where *XX* = the number of MPI processes):

```
mpiexec -np XX MPInside a.out
```

Note: The `MPInside` command is case-sensitive.

By default, MPInside measures the elapsed times of the MPI functions and collects message statistics. After the MPI application completes successfully, the measurement and the statistics data are reported in a text file called mpinside_stats. This output file contains five sections:

- The elapsed time of each function in seconds (broken down into columns for computation time and for various MPI function times, with one rank per row)
- Total megabytes (MBytes) sent by each rank
- Number of "send" calls by each rank
- Total MBytes received by each rank
- Number of "recv" calls by each rank

You can import the mpinside_stats file to an Excel spreadsheet and plot the data for visual analysis (for example, analysis of possible load imbalance or communication characteristics). You can also compute derived metrics, such as average message size, from the mpinside_stats file.

Additional Features

MPInside uses environment variables to set all of its options. These optional variables are documented in the **mpinside** man page.

Recommended Features

We recommend the following settings:

`MPINSIDE_LITE=1`
: Use when synchronization frequency is very high

`MPINSIDE_CUT_OFF=0.0`
: Do not cut off reporting of any MPI functions

MPINSIDE_OUTPUT_PREFIX={run i.d.}
Change default report file name
MPINSIDE_PRINT_ALL_COLUMNS=1
Enable reporting of all columns (even if the values are zero) to allow easier comparisons between runs
MPINSIDE_PRINT_SIZ_IN_K=1
Print in KB units instead of MB

Advanced Features

The advanced features described below may also be useful.

MPINSIDE_SHOW_READ_WRITE

Measures the I/O time and includes several columns in the mpinside_stats output file indicating the time, number of Mbytes, and number of calls associated with the libc I/O functions. Note that the MPI function times and I/O times are subtracted from the total elapsed run times, and the result is reported as "Comput" time.

MPINSIDE_SIZE_DISTRI [T+]nb_bars[:*first-last*]

Prints a histogram of the request sizes distribution at the end of mpinside_stats for each rank specified in [:*first-last*]. If T+ is specified, size distribution time histograms are also printed. For example, setting MPINSIDE_SIZE_DISTRI to T+12:0-47 will print two histograms, as follows, for each rank (from 0 to 47) at the end of mpinside_stats:

1. Size distribution for sizes 0, 32, 64, 128, 256, 512, ..., up to 65536
2. Size distribution time histogram

MPINSIDE_CALLSTACK_DEPTH

Unwinds the stack up to the depth specified, and creates an mpinside_clstk.xxx file for each rank. The mpinside_clstk.xxx files can be post-processed with the utility MPInside_post -l to get the routine names and source line numbers. Use the-h option of MPInside_post to get more information.

MPINSIDE_EVAL_COLLECTIVE_WAIT

Checks whether a slow collective operation (such as MPI_Bcast or MPI_Allreduce) is caused by some ranks reaching the rendezvous point much later than others.

Setting this variable will put an MPI_Barrier and time it before any MPI collective operation. This assumes that the time of a collective operation is the sum of the time for all processors to synchronize plus the time of the actual operation (i.e, the physical transfer of data). In the mpinside_stats output file, the "b_*xxx*" column will give the MPI_Barrier time of the corresponding *xxx* MPI collective function, and the "*xxx*" column gives the remaining time.

MPINSIDE_EVAL_SLT

Checks whether a slow point-to-point communication is caused by late senders.

Setting this variable will measure the time the Sends arrived late compared to Recv or Wait arrivals. In the mpinside_stats output file, the column "w_*xxx*" (for example, w_wait or w_recv) represents the send-late-time (SLT) for the corresponding *xxx* function.

Note: The MPINSIDE_EVAL_COLLECTIVE_WAIT and MPINSIDE_EVAL_SLT variables are useful for identifying some problems that are intrinsic to an application, i.e., issues that cannot be remedied even with an ideal system (zero latency and infinite bandwidth).

References

- **mpinside** man page
- MPInside Reference Manual on the PFEs: /nasa/sgi/MPInside/3.6.2/doc/mpinside_3.5_ref_manual.pdf
- HPE Online Documentation: MPInside Reference Guide
- Publication: MPInside: a Performance Analysis and Diagnostic Tool for MPI Applications

The Intel VTune Profiler is an analysis and tuning tool that provides predefined analysis configurations to address performance questions.

This article provides basic information on several Intel VTune Profiler analysis types that examine various aspects of performance and identify potential benefits for your application from available hardware resources.

VTune Profiler Analysis Types

The following list provides a brief description of each analysis type that can be used with NAS resources.

performance-snapshot
: Get a quick snapshot of your application performance and identify next steps for deeper analysis. This analysis type became available starting with VTune Profiler Version 2020 Update 2. The Intel Xeon Sandy Bridge processors are not supported for this analysis.

hotspots
: Investigate call paths and find where your code is spending the most time; identify opportunities to tune your algorithms. This analysis type is in the Vtune Profiler's Algorithm analysis group.

threading
: Discover how well your application is using parallelism to take advantage of all available CPU cores. Best for visualizing thread parallelism on available cores, locating causes of low concurrency, and identifying serial bottlenecks in your code. This analysis type is in the Parallelism analysis group.

memory-consumption
: Analyze memory consumption by your application, its distinct memory objects, and their allocation stacks. This analysis type is in the Algorithm analysis group.

uarch-exploration
: Analyze CPU microarchitecture bottlenecks affecting the performance of your application. This analysis type is in the Microarchitecture analysis group.

memory-access
: Measure a set of metrics to identify issues related to memory access. Best for memory-bound applications to determine which level of the memory hierarchy is impacting your performance by reviewing CPU cache and main memory usage, including possible NUMA issues. This analysis type is in the Microarchitecture analysis group.

hpc-performance
: Analyze performance aspects of compute-intensive applications, including CPU and GPU utilization. Get information on OpenMP efficiency, memory access, and vectorization. This analysis type is in the Parallelism analysis group.

io
: Analyze utilization of IO subsystems, CPUs, and processor buses. This analysis type is in the Input and Output analysis group.

VTune Profiler Collection Types

There are two collection types:

- User-mode sampling and tracing collection.
- Hardware event-based sampling collection.

Each VTune analysis type uses one of the two collection types by default.

User-Mode Sampling and Tracing Collection

This collection method uses the operating system interrupts. No sampling drivers are needed. Default resolution is 10 millisecond (ms). The method works for the Intel Xeon nodes—PFEs and compute—and the AMD nodes.

Hardware Event-based Sampling (EBS) Collection

This collection method uses the counter overflow feature of the on-chip Performance Monitoring Unit (PMU). Default resolution is 1 ms. The is enabled for the Intel Xeon compute nodes only; it is not applicable to the PFEs (for security reasons) or the AMD processors (for hardware reasons).

There are two ways to facilitate EBS:

- Install and load the Intel sampling drivers
- Enable the Perf driverless collection, where the Linux Perf driver is used instead of the Intel sampling drivers

Each method is described below. Both require administrator privilege to configure them.

Install and Load the Intel Sampling Drivers

Different VTune Profiler versions may need different versions of the sampling drivers. A change in the kernel may result in the need to reinstall and reload the drivers. Here is one way to check whether the VTune sampling drivers have been loaded on the system for a particular version of VTune:

```
% <VTUNE_PROFILER_DIR>/sepdk/src insmod-sep -q
```

For example:

```
% /nasa/intel/Compiler/2021.4.0/vtune/2021.9.0/sepdk/src insmod-sep -q
```

Enable the Perf Driverless Collection

With this method, the Linux Perf driver is used instead of the Intel sampling drivers. The Intel VTune Profiler can use this driverless mode if the following requirements are satisfied:

- Access to core and uncore events. All hardware event-based collections in VTune Profiler use core PMU events. Some of them, such as the Memory Access and IO analysis types, also require access to uncore events that enable collecting metrics such as DRAM bandwidth, QPI/UPI bandwidth, PCI bandwidth, and others.
- Perf for Linux kernel 2.6.32 or later. PMU events are exposed by the Linux kernel through /sys/bus/event_source/devices/cpu and /sys/bus/event_source/devices/uncore_* directories.
- The value of /proc/sys/kernel/perf_event_paranid must be 0 or 1.

See Intel Processor Events Reference to learn more about core and uncore events.

Compared to the Intel sampling drivers method, there are limitations to using the Perf driverless collection mode. For example, in order to use Perf driverless collection mode with the uarch-exploration and memory-access analysis types, a system administrator must set the default limit of opened file descriptors (listed in the /etc/security/limits.conf file) to exceed 100 x *number_of_logic_CPU_cores*. More limitations are listed in the Intel documentation, Profiling Hardware Without Intel Sampling Drivers.

In all cases except for using the hotspots analysis type with stack collection, VTune Profiler uses the Intel sampling drivers if they are loaded. To make the VTune Profiler use the driverless Perf mode for sampling without stacks, create a custom analysis type and select the **Enable driverless collection** option in the GUI, or set the knob value in the command line to enable-driverless-collection=true as follows:

```
vtune -collect-with runsa -knob enable-driverless-collection=true \
-knob event-config=<event-list> <application>
```

For example,

```
-knob enable-driverless-collection=true -knob \
event-config=CPU_CLK_UNHALTED.CORE,CPU_CLK_UNHALTED.REF,INST_RETIRED.ANY
```

The enable-driverless-collection option is available starting with VTune 2019 Update 4.

Note: The command lines above is too long to be formatted as one line, so they are broken with a backslash (\).

Readiness of Running Various Analysis Types

To check whether VTune Profiler is ready for use on a system, you can use the vtune-self-checker.sh script, which can be found under <VTUNE_PROFILER_DIR>/bin64. This script runs a subset of the available analysis types against a matrix multiply application, and reports whether the analysis runs successfully. If EBS is used for the analysis, it also states whether the Intel sampling drivers or the Perf driverless mode is used.

Using the VTune Profiler Command Line

We recommend using the latest version of VTune on HECC systems, vtune/2021.9:

```
module load vtune/2021.9
```

To learn more about various analysis types, use:

```
vtune -h collect
```

or

vtune -h collect *name_of_analysis_type*

To run the VTune self checker (to confirm whether the correct drivers are installed and the system is set up properly):

```
vtune-self-checker.sh > output
```

When using the HPE MPT libraries with VTune, set the MPI_SHEPHERD and MPI_USING_VTUNE variables as follows.

Note: For some analyses, MPT may generate error message in your PBS output files stating that additional variables (such as MPI_UNBUFFERED_STDIO) need to be set.

```
(bash)
export MPI_SHEPHERD=true
export MPI_USING_VTUNE=true

(csh)
setenv MPI_SHEPHERD true
setenv MPI_USING_VTUNE true
```

For an example of running an analysis type and viewing the results, see Finding Hotspots in Your Code with the Intel VTune

Command-Line Interface.

Summary Table

The information provided above was tested on NAS systems using vtune/2021.9 and is summarized in the following table.

Analysis Type	Additional Knobs Used	Analysis Group	Sampling Method	If EBS, Which Mode?	Unsupported Processor Types
performance-snapshot		N/A	EBS	Intel Driver	Sandy Bridge, AMD processors
hotspots		Algorithm	User-Mode	N/A	None
hotspots	-knob sampling-mode=hw	Algorithm	EBS	Intel Driver	AMD processors
hotspots	-knob sampling-mode=hw -knob enable-stack-collection=true	Algorithm	EBS	Perf[1]	AMD processors
threading		Parallelism	User-Mode	N/A	None
threading	-knob sampling-and-waits=hw	Parallelism	EBS	Intel Driver	AMD processors
threading	-knob sampling-and-waits=hw -knob enable-stack-collection=true	Parallelism	EBS	Intel Driver	AMD processors
memory-consumption		Algorithm	User-Mode	N/A	None
uarch-exploration		Microarchitecture	EBS	Intel Driver	Haswell[2], AMD processors
memory-access		Microarchitecture	EBS	Intel Driver	AMD processors
hpc-performance		Parallelism	EBS	Intel Driver	Sandy Bridge[3], AMD processors
io		Input and Output	EBS	Intel Driver	AMD processors
-collect-with runsa	-knob event-config=<event-list>	Custom Analysis	EBS	Intel Driver	AMD processors
-collect-with runsa	-knob event-config=<event-list> -knob enable-driverless-collection=true	Custom Analysis	EBS	Perf	AMD processors

[1] Starting with VTune 2019 Update 4, hotspots with EBS and stacks uses the Perf driverless mode by default even when the Intel sampling drivers are available.

[2] No L2, L3, DRAM bound (in % of clockticks), memory bandwidth or memory latency available for Haswell when hyperthreading is turned on.

[3] Vectorization analysis is limited for Sandy Bridge. Only metrics based on binary static analysis such as vector instruction set will be available.

Additional References

- VTune Analysis Types
- Intel VTune Profiler User Guide
- Intel VTune Profiler Performance Analysis Cookbook

The Intel VTune Profiler (renamed from Amplifier starting with 2020.0 version) is an analysis and tuning tool that provides predefined analysis configurations to address various performance questions. Among them, the *hotspots* analysis type can help you to identify the most time-consuming parts of your code and provide call stack information down to the source lines.

The hotspots analysis type allows two data collection methods: (1) user-mode sampling and tracing collection, and (2) hardware event-based sampling collection. Both methods are supported on all current Aitken and Electra Intel processor types: Broadwell, Skylake, and Cascade Lake.

Note: For the AMD Rome and Milan nodes, the user-mode sampling and tracing collection is supported but the hardware event-based sampling collection is not.

The instructions below apply to using the user-mode sampling and tracing collection type, with a fixed sampling interval of 10 ms.

Setting Up to Run a Hotspots Analysis

Complete these steps to prepare for profiling your code:

1. Add the -g option to your usual set of compiler flags in order to generate a symbol table, which is used by VTune during analysis. Keep the same optimization level that you intend to run in production.
2. For MPI applications, build the code with the latest version of MPT library, as it will likely work better with Intel Vtune.
3. Start a PBS interactive session or submit a PBS batch job.
4. Load a VTune module in the interactive PBS session or PBS script, as follows:

```
module load vtune/2021.9
```

You can now run an analysis, as described in the next section.

Running a Hotspots Analysis

Run the vtune command line that is appropriate for your code, as listed below. Use the-collect (or -c) option to run the hotspots collection and the -result-dir (or -r) option to specify a directory.

Note: The vtune command replaces amplxe-cl, which was used prior to the 2020.0 version.

Running a Hotspots Analysis on Serial or OpenMP Code

To profile a serial or OpenMP application (for example):

```
vtune -collect hotspots -result-dir r000hs a.out
```

Data collected by VTune for the a.out application are stored in the r000hs directory.

Running a Hotspots Analysis on Python Code

To profile a Python application:

```
vtune -collect hotspots -result-dir r000hs /full/path/to/python3 python_script
```

Data collected by VTune while running the Python script will be stored in ther000hs directory.

Note: See Additional Resources below for more information about Python code analysis.

Running a Hotspots Analysis on MPI Code Using HPE MPT

To profile an MPI application using HPE MPT:

```
setenv MPI_SHEPHERD true
setenv MPI_USING_VTUNE true
mpiexec -np XX vtune -collect hotspots -result-dir r000hs a.out
```

Note: If your vtune run fails with MPT ERROR and a suggestion to set additional MPT environment variables, for example MPI_UNBUFFERED_STDIO, follow the suggestion and try again.

At the end of the collection, VTune generates a summary report that is written by default tostdout. The summary includes information such as the name of the compute host, operating system, CPU, elapsed time, CPU time, and average CPU utilization.

Data collected by VTune are stored and organized with one directory per host. Within each directory, data are further grouped into subdirectories by rank. As an example, for a 48-rank MPI job running on two nodes with 24 ranks per node, VTune generates two directories, r000hs.r587i0n0 and r000hs.r587i0n1, where each directory contains 24 subdirectories, (data.[*0-23*]).

To reduce the amount of data collected, consider profiling a subset of the MPI ranks, instead of all of them. For example, to profile only rank 0 of the 48-rank MPI job:

```
mpiexec -np 1 vtune -collect hotspots -result-dir r000hs  a.out : -np 47 a.out
```

Viewing Results with the VTune GUI

Once data are collected, view the results with the VTune graphical user interface (GUI), vtune-gui. Since the PFEs do not have sufficient memory to run vtune-gui, run it on a compute node as follows:

1. On a PFE, run echo $DISPLAY to find its setting (for example, pfe22:102.0).
2. On the same PFE window, start a VNC session.
3. On your desktop, use a VNC viewer (for example, TigerVNC) to connect to the PFE.
4. On the PFE (via the VNC viewer window), request a compute node either through an interactive PBS session (qsub -I -X) or a reservable front end (pbs_rfe).
5. On the compute node, start vtune-gui with name of the result directory, for example, r000hs.

```
compute_node% module load vtune/2021.9
compute_node% vtune-gui r000hs
```

Generating a Report

You can also use the vtune command with the -report option to generate a report. There are several ways to group data in a report, as follows:

- To report time grouped by functions (in descending order) and print the results to stdout, or to a specific output text file, such as output.txt:

  ```
  vtune -report hotspots -r r000hs
  ```

  or

  ```
  vtune -report hotspots -r r000hs -report-output output
  ```

- To report time grouped by source lines, in descending order:

  ```
  vtune -report hotspots -r r000hs -group-by source-line
  ```

- To report time grouped by module:

  ```
  vtune -report hotspots -r r000hs -group-by module
  ```

You can also generate a report showing the differences between two result directories (such as from two different ranks or two different runs), or select different types of data to display in a report, as follows:

- To report the differences between two result directories:

  ```
  vtune -report hotspots -r r000hs -r r001hs
  ```

- To display CPU time for call stacks:

  ```
  vtune -report callstacks -r r000hs
  ```

- To display a call tree and provide CPU time for each function:

  ```
  vtune -report top-down -r r000hs
  ```

Additional Resources

See the following Intel VTune articles and documentation:

- hotspots Command Line Analysis
- hotspots Analysis for CPU Usage Issues
- Python Code Analysis

Darshan is an open-source, lightweight high-performance computing I/O characterization toolkit developed at Argonne National Lab. Darshan instruments applications and captures calls to the POSIX, MPI-IO, STDIO, and HDF5 interfaces, and provides some limited information about PnetCDF. Statistics collected by Darshan include (but are not limited to) the number, size, and time spent on I/O calls, as well as the number and names of files, POSIX I/O access patterns (such as the access sizes and the numbers of total, consecutive, and sequential operations), timespan from first to last access, and estimated performance.

Darshan on NAS Systems

Two different Darshan builds are available. One was built on a PFE for MPI applications that use HPE MPT, and the other was built for non-MPI applications. Both are described below. Note that the two builds are installed in different locations.

Darshan for HPE MPI Applications

After loading an Intel compiler module and an HPE MPT module, this Darshan runtime was built with these options:

--prefix=/nasa/darshan/3.4.0_mpt
--with-log-path-by-env=DARSHAN_LOG_PATH
--enable-hdf5-mod
--with-hdf5=/nasa/hdf5/1.12.0_mpt
--with-jobid-env=PBS_JOBID CC=mpicc

Darshan for Non-MPI Applications

This Darshan runtime was built with these options:

--prefix=/nasa/darshan/3.4.0_non_mpi --with-log-path-by-env=DARSHAN_LOG_PATH
--with-jobid-env=PBS_JOBID --without-mpi CC=gcc

Instrumenting Your Application Using darshan-runtime

You can enable Darshan instrumentation either at compile/link time or at runtime. Runtime instrumentation is a quicker way to get started using Darshan, as it does not require modification to your existing executable; rather, it's done by setting the LD_PRELOAD environment variable for your PBS batch job as described below.

As the application runs, Darshan records statistics for each process. At the end of the run, Darshan collects, aggregates, compresses, and writes a log file to a location specified by the environment variable DARSHAN_LOG_PATH. The filename of the Darshan binary log will likely follow this format:

<USERNAME>_<BINARY_NAME>_<JOB_ID>_<DATE>_<UNIQUE_ID>_<TIMING>.darshan.

You can change the location and name of the Darshan log by setting the variable DARSHAN_LOGFILE=/path/*name*.darshan.

Using darshan-runtime for HPE MPI Applications

Use the following PBS script to run your HPE MPI application with Darshan.

Note: For MPI applications, the MPI_SHEPHERD environment variable must be set to true (highlighted in the script below) to avoid deadlock when preloading the Darshan shared library.

```
#!/bin/bash
#PBS -lselect=....

source /usr/local/lib/global.profile
module load mpi-hpe/mpt
#Note: Profiling HDF5 app with Darshan should NOT use hdf5 version 1.8.18_mpt
#Uncomment the following two lines if running HDF5 applications
#module use -a /nasa/modulefiles/testing
#module load hdf5/1.12.0_mpt

export DARSHAN_DIR=/nasa/darshan/3.4.0_mpt
export DARSHAN_LOG_PATH=/location_where_you_want_darshan_log_to_be_written_to
export MPI_SHEPHERD=true

mpiexec -genv LD_PRELOAD ${DARSHAN_DIR}/lib/libdarshan.so -np X mpi_executable
```

TIP: By default, Darshan aggregates statistics for files accessed by all ranks and collapses them into a single cumulative file record. If you want to retain the per-process information (which will result in a larger log file), set this variable:

export DARSHAN_DISABLE_SHARED_REDUCTION=1

Using darshan-runtime for Non-MPI Applications

Use the following PBS script to run your non-MPI application with Darshan.

Note: For non-MPI applications, the DARSHAN_ENABLE_NONMPI environment variable must be set to 1 (highlighted in the script, below).

```
#!/bin/bash
#PBS -lselect=....

source /usr/local/lib/global.profile
export DARSHAN_DIR=/nasa/darshan/3.4.0_non_mpi
export DARSHAN_LOG_PATH=/location_where_you_want_darshan_log_to_be_written_to
export DARSHAN_ENABLE_NONMPI=1
env LD_PRELOAD=${DARSHAN_DIR}/lib/libdarshan.so non-mpi_executable
```

Note: Darshan does not necessarily interfere with other profiling tools, such as the NAS-developed mpiprof tool. However, outputs generated by other profiling tools will likely also be counted in the Darshan binary log, which may confuse your analysis.

Darshan Extended Tracing

Darshan can perform eXtended Tracing (DXT) of the MPI-IO and POSIX I/O to provide details on each read or write operation issued by each rank, as well as which ranks are performing I/O and how long they are spending on I/O. A memory limit of 2 MebiByte (MiB) each for DXT_MPIIO and DXT_POSIX modules is set as default.

Note: DXT may result in longer runtime and higher memory overheads. It is also possible that the tracing log may be incomplete if a Darshan DXT module runs out of memory to store new record data.

To use DXT to trace all files, add one of the following lines in your PBS script:

```
export DARSHAN_MOD_ENABLE="DXT_POSIX,DXT_MPIIO"
or
export DXT_ENABLE_IO_TRACE=1
(This is a variable for earlier Darshan versions. It also works with 3.4.0.)
```

You can also select which items you want to include in or exclude from DXT (for example—files, ranks, or small I/O operations) by providing a trace configuration file or by setting certain DXT environment variables. For more information, read Darshan-runtime installation and usage documentation.

Analyzing Darshan Log Files with darshan-util Tools

The Darshan log file generated can be ported to and analyzed on systems wheredarshan-util is installed. You can perform the analysis on a PFE by loading either the HPE MPI version or the non-MPI version of the Darshan modulefiles, as shown below.

Note: Some of the darshan-util tools also require Perl, pdflatex, gnuplot, and epstopdf, which can be found in the pkgsrc/2023Q3 directory on NAS systems.

```
pfe% module use -a /nasa/modulefiles/testing

pfe% module load darshan/3.4.0_mpt
or
pfe% module load darshan/3.4.0_non_mpi

pfe% module load pkgsrc/2023Q3
```

After you load the Darshan and pkgsrc modulefiles, you can use one of the followingdarshan-util tools on a PFE.

darshan-job-summary.pl

The darshan-job-summary.pl tool processes the Darshan binary log (for example, *name*.darshan) and generates a multi-page PDF (*name*.darshan.pdf) containing graphs, tables, and performance estimates characterizing the I/O activity of the job:

```
pfe% darshan-job-summary.pl name.darshan
```

A sample PDF provided by the Darshan developers can be foundhere.

darshan-summary-per-file.sh

This script is similar to darshan-job-summary.pl except that it produces a separate PDF summary for every file accessed by the application:

```
pfe% darshan-summary-per-file.sh name.darshan output-dir
```

where *output-dir* is a directory to be created; it will contain the collection of PDFs (one PDF per file).

Note: Using this utility is not recommended if your application opens a large number of files.

darshan-parser

The darshan-parser tool extracts everything from a Darshan log and displays it in text format. It provides more information than darshan-job-summary.pl.

pfe% darshan-parser [options] *name*.darshan > *name*.txt

To show the available options, do: darshan-parser -h

To extract information for a specific file from a Darshan log containing statistics for many files, and produce a Darshan log for that file, complete these steps:

1. Show a list of filenames along with each file's record ID:
   darshan-parser --file-list *name*.darshan
2. Generate the log for the file you want:
   darshan-convert --file *a_record_id name*.darshan *a_file*.darshan.

darshan-dxt-parser

If you used DXT to generate a trace file (for example, *dxt_name*.darshan), use the darshan-dxt-parser tool to generate a text output:

pfe% darshan-dxt-parser [--show-incomplete] *dxt_name*.darshan > *dxt_name*.txt

The option --show-incomplete will display results even if the log is incomplete.

Known Issues

When you post-process the Darshan binary log for runs performed under a Lustre filesystem with progressive file layout, the following error will occur:

Error: failed to parse LUSTRE module record.

However, this error does not affect other I/O statistics (such as the POSIX or MPIIO stats) collected by Darshan for your application.

References

- Darshan-runtime installation and usage
- Darshan-util installation and usage

I/O characterization helps to identify opportunities for I/O related performance optimization for your application which, in turn, may also result in improved health of a shared filesystem used by many users.

The NAS in-house MPIProf tool is light-weight and easier to use compared to other I/O characterization tools. This article gets you started with I/O profiling of your application using MPIProf on NAS systems, and understanding some of the I/O statistics (such as overall, per-function, or per-rank) gathered in its text output.

After completing I/O profiling using MPIProf, you may choose to use DARSHAN to obtain finer-grain (per-file, per-call or time tracing) I/O information such as the name of the file(s), and when and where in the file the I/O occurs.

Basics on I/O Libraries

A user application performs I/O from and/or to the storage devices by calling some I/O libraries. These libraries can be divided into three layers:

- low-level layer

  The Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE to facilitate application portability. The POSIX I/O library with functions such as open, read, write, and close is at the lowest layer of all I/O libraries, and it interacts with the filesystem client and server software and hardware. All other I/O libraries above the POSIX I/O layer will eventually call the POSIX I/O functions.

- mid-level layer

  Examples of mid-level I/O libraries include those provided by the Fortran, C, C++ languages and the MPI I/O libraries.

  MPI I/O allows multiple processes to perform I/O on a file in either a collective or a non-collective way. The collective way requires all processes to participate when performing the I/O operations. The non-collective way lets each process do I/O independently.

  The names of MPI I/O functions start with MPI_File. Some examples are:

  - collective MPI I/O functions

    MPI_File_open,
    MPI_File_close,
    MPI_File_read_all, MPI_File_read_at_all, MPI_File_read_ordered
    MPI_File_write_all, MPI_File_write_at_all, MPI_File_write_ordered

  - non-collective MPI I/O functions

    MPI_File_read, MPI_File_read_at, MPI_File_read_shared
    MPI_File_write, MPI_File_write_at, MPI_File_write_shared

  Some implementations of collective MPI I/O provide performance improvement. This is done through reducing the number of times the filesystem is touched by transforming a large number of small, non-contiguous pieces of data in a file accessed by many MPI processes into fewer but larger, contiguous chunks of data handled by a smaller number of MPI processes.

- high-level layer

  There are many different high-level I/O libraries, for example, HDF5 and NETCDF. These libraries provide natural expressions of scientific data, which makes more sense to the scientists.

  The serial version of these high-level libraries calls the POSIX I/O functions directly while the parallel versions call the mid-level MPI I/O which in turn calls the POSIX I/O functions.

MPIProf does not instrument high-level I/O libraries such as HDF5 and NETCDF. It can selectively instrument the MPI communication, MPI I/O and/or POSIX I/O functions through its various options. After loading the mpiprof modulefile, you can read output from mpiprof -h for all available MPIProf command-line options. More detailed information is available through the output from mpiprof -help.

Usage

Load the appropriate compiler and MPI libraries used to build your application and the mpiprof modulefile before using MPIProf for I/O profiling, for example:

```
module load comp-intel
module load mpi-hpe/mpt
module load scicon/mpiprof
```

Choose one of the following three most commonly used methods for I/O profiling:

- Use the default:

  ```
  mpiexec -np X mpiprof a.out
  ```

  This will instrument MPI communication functions, MPI I/O functions and POSIX I/O functions called by the application. However, it does not instrument the underlying POSIX I/O functions called from the MPI I/O functions.

- Use the -mio option:

  mpiexec -np X mpiprof -mio a.out

  This will instrument MPI communication functions, MPI I/O functions, and POSIX I/O functions called by the application. Furthermore, it will keep track of the underlying POSIX I/O functions called from the MPI I/O functions.

- Use the -ios option:

  mpiexec -np X mpiprof -ios a.out

  The -ios option does not instrument MPI communication functions. It instruments the MPI I/O functions and POSIX I/O functions called by the application. Furthermore, it will keep track of the underlying POSIX I/O functions called from the MPI I/O functions.

The MPIProf Statistics Output File

MPIProf generates a text output file at the end of the run. When the -o option is not used to specify an output file name, the profiling results are written to "mpiprof_stats.out" or "<a.out>_mpiprof_stats.out" if the executable name is known. The output file includes various statistics in multiple sections separated by the ==> header. A few of the sections are briefly described here:

==> Summary of this run

This section reports the execution time, total message and I/O size, communication and I/O rate, and some remarks about this run. For the execution time, since MPIProf does not profile the computation of the application, the execution time for computation is deduced by substracting communication time and/or I/O time from the total wall clock time.

MPIProf marks an application as I/O intensive when the effective I/O time is more than 10%.

Note: The effective I/O time is estimated by dividing the total I/O data size with the aggregated I/O rate from all active ranks. This is a more meaningful representation than the average I/O time for cases where I/O occurs only on a small number of ranks, such as rank 0.

Note: For runs with the mpiprof -ios option, since MPI communcation is not profiled, no communiciation time is reported. The reported computation time includes both communication time and computation time which are not separable from the gathered statistics.

TIP: Tip: Use this section to get a high-level characterization of the application.

==> Instrumented function list

See Appendix A of the MPIProf User Guide (/u/scicon/tools/opt/mpiprof/default/doc/mpiprof_userguide.pdf) for a complete list of all functions available for instrumentation.

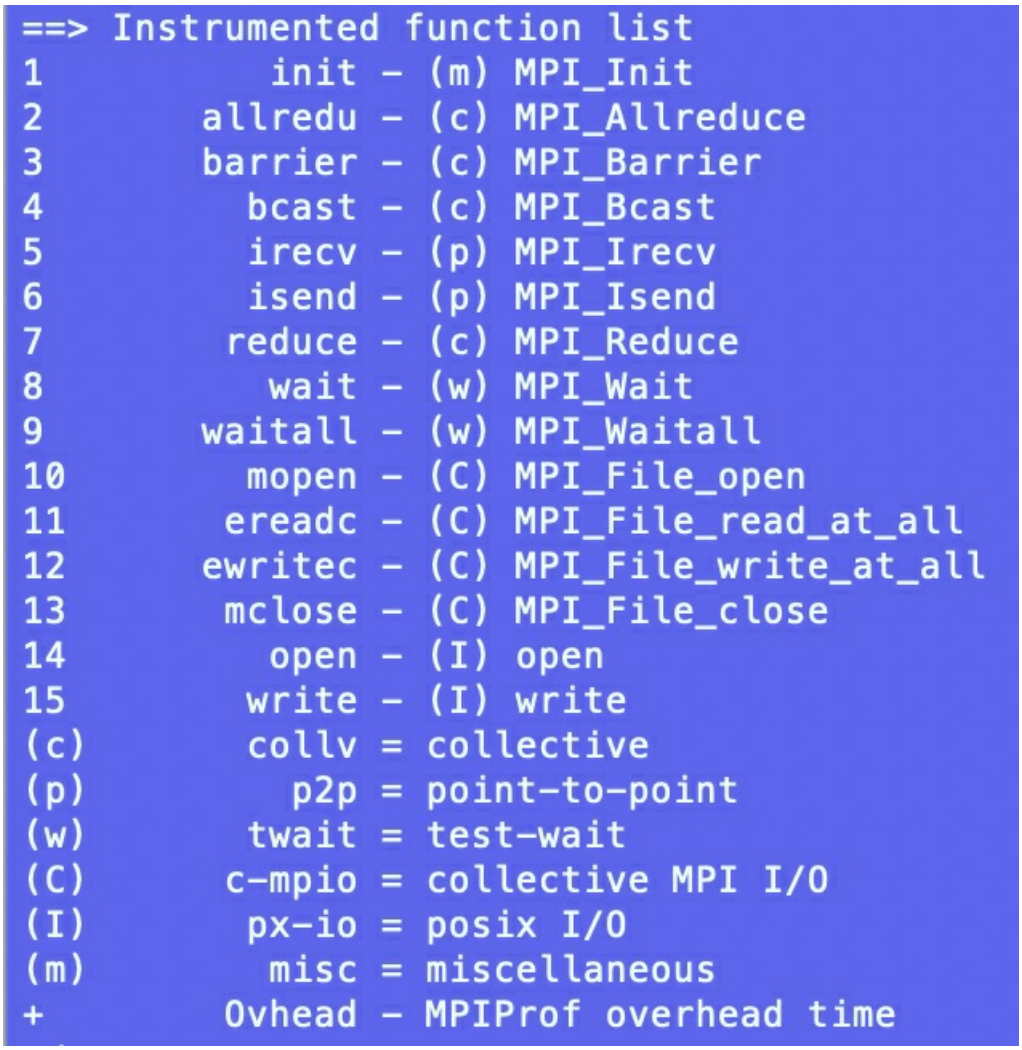
For illustration purposes, the lists below compare the instrumented function list using the above three methods for one of the NAS Parallel Benchmarks (NPB) BTIO benchmarks.

The instrumented functions are enumerated as shown in the first column of each list. The second column provides a short name of each instrumented function. For example, the short name for the function MPI_Init is init. Use of these short names allows a concise presentation of statistics in other sections of the output file. Each instrumented function is also denoted with one of { (c), (p), (w), (C), (N), (I), (<), (:), (m) } to indicate its nature, for example, (c), (p), and (w) for various MPI communication functions, (C) for collective MPI I/O functions, and (N) for non-collective MPI I/O functions. Be aware that MPIProf separates the statistics for (I) POSIX I/O functions not called by MPI I/O functions, (<) POSIX I/O from collective MPI I/O functions, and (:) POSIX I/O from non-collective MPI I/O functions.

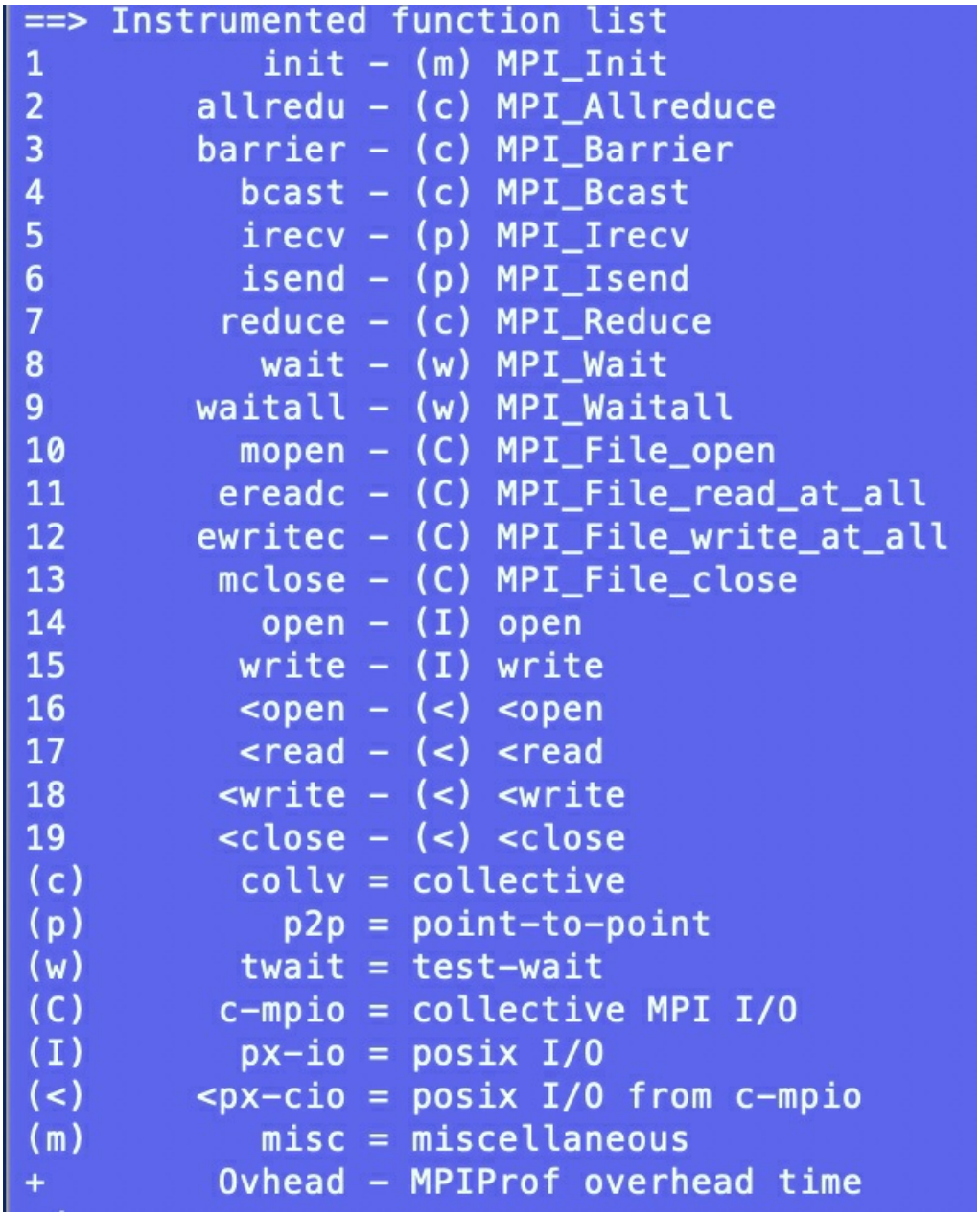
Note: The BTIO benchmark used for illustration does not call non-collective MPI I/O functions, so there are no functions denoted with (N) or (:) in the instrumented function list.

TIP: Use this section to find: what functions are instrumented in your run; the mapping of the short names and the original function names; and whether the function is an MPI communication, collective or non-collective MPI I/O function, or a POSIX I/O function.

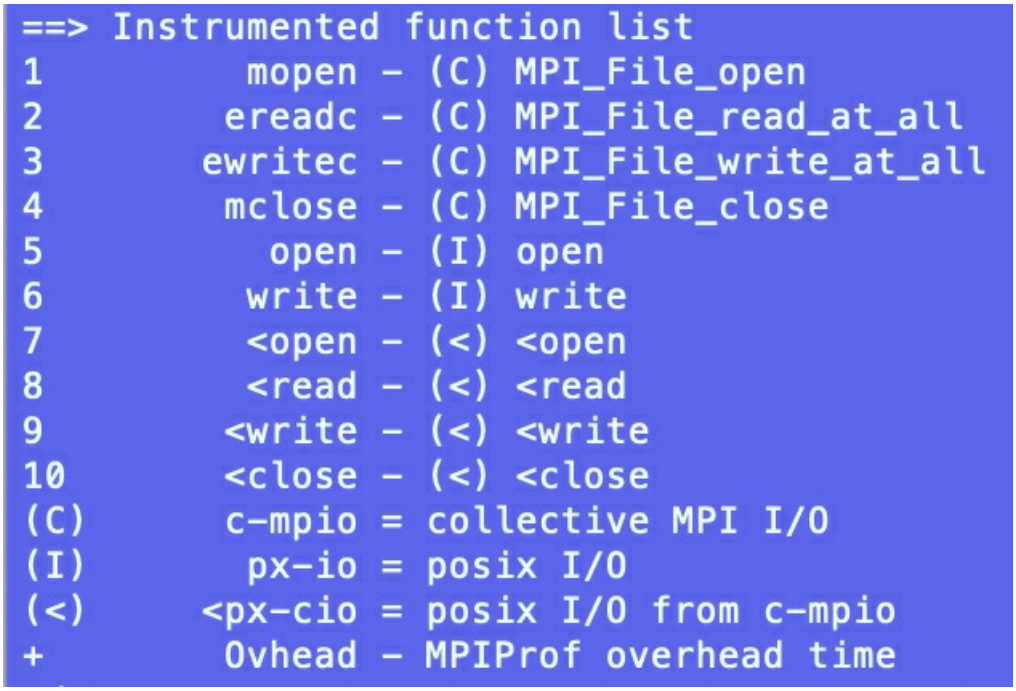
default instrumented function list



with -mio instrumented function list



with -ios instrumented function list



==> Per function summary

This section provides statistics (time spent and percentage time, number and sizes) of each instrumented function summing over all MPI ranks. For the runs with the -mio or -ios option, note that the time reported for <px-cio (that is, summing over all POSIX I/O from collective MPI I/O) and <px-nio (that is, summing over all POSIX I/O from non-collective MPI I/O) is already included in the time spent in the corresponding MPI I/O function groups (c-mpio and nc-mpio).

TIP: Use this section to identify which instrumented function(s) take most time in the application.

==> Message and data size histograms

For runs performed with -ios, the title of this section is "Data size historgrams" since MPI communication is not profiled.

This section provides the counts (summed over all MPI ranks) of send, receive, read and write in different size ranges. Note the unit of size is represented in decimal. For example, 16.384K is 16384 bytes.

TIP: High counts of large read and write sizes are more likely seen in applications with more effective I/O.

Multiple Sections Where Per-Rank Statistics are Reported

MPIProf reports per-rank statistics in multiple sections. For instance,

- ==> Summary - Timing in seconds
- ==> Summary - Function calls, node map, and memory bytes
- ==> Summary - Message sent or data written in bytes, rate in Mbytes/sec
- ==> Summary - Message received or data read in bytes, rate in Mbytes/sec

Some sections provide both per-rank and per-function statistics. For example,

- ==> Timing of functions in seconds
- ==> Counts of function calls
- ==> Message sent or data written in bytes
- ==> Message received or data read in bytes

For runs profiled with -ios, the 'Message sent', 'Message received' in the title are removed since no MPI communication is profiled.

In the ==> Summary - Timing in seconds section, total I/O time is broken down into time spent in c-mpio, nc-mpio, px-io, <px-cio, and <px-nio . As stated in the description of the ==> Per function summary above, time spent by each rank in <px-cio and <px-nio has already been included in the time spent in c-mpio and nc-mpio.

In the ==> Summary - Timing of functions in seconds section, the total I/O time is broken down into the specific MPI I/O function or POSIX I/O function being profiled. Note that the time reported for the underlying POSIX I/O calls from MPI I/O (such as <open, <read, <write, <close, <ncread, and <ncwrite) has already been included in the time reported for the MPI I/O functions (such as mopen, ereadc, ewritec, mclose, eread, and ewrite).

TIP: Examine the statistics reported for each rank in these sections to help identify which ranks perform the dominant POSIX I/O operations and whether the dominance is due to POSIX functions called or not called from MPI I/O functions.