Deliverable 2.2

Functional Specification of the Backend

Part 1: Pre-Selection of the Metric Collection Tools

Work Package 2

August 2017

ProfiT-HPC Consortium

Gefördert durch die Deutsche Forschungsgemeinschaft (DFG)
Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)
KO 3394/14-1, OL 241/3-1, RE 1389/9-1, VO 1262/1-1, YA 191/10-1

Abstract

This document will deliver a functional, yet abstract specification for the backend of the toolkit, which will be developed in the course of this project.

In this part one of the document, a preliminary choice of tools based on Deliverable 2.1 – "Concise Overview of Performance Metrics and Tools" is investigated. Based on these investigations and the experiences, the suitability of these tools to use within the context of the toolkit will be evaluated.

Gefördert durch die Deutsche Forschungsgemeinschaft (DFG)
KO 3394/14-1, OL 241/3-1, RE 1389/9-1, VO 1262/1-1, YA 191/10-1

Contents

1 Introduction and Rationale

This document aims to provide a functional specification of the backend of the developed toolkit. With this in mind, the main goal is to cover the answer to the questions of what performance metrics can be retrieved, what tools for which metrics can be used, how these metrics can be acquired from the output of the tools, and how much overhead is to be expected by using these tools. These information and the corresponding tool documentations have been gathered through the study of the available documentations, as well as performing several essential tests and benchmarks on different (HPC) systems.

Based on the evaluation in Deliverable 2.1 – "Concise Overview of Performance Metrics and Tools", the following tools have been chosen in order to perform further investigations.

- The proc filesystem,
- vmstat,
- GNU time,
- Sysstat utilities (including `sar`, `iostat`, and `pidstat`),
- Perf,
- LIKWID,
- Darshan,
- Open | SpeedShop.

After the survey done regarding Tier-2 and Tier-3 infrastructures (Deliverable 1.2 – "Results of a Survey concerning the Tier-2 and Tier-3 HPC-Infrastructure in Germany"), it has been turned out that many of these systems have installed and use the Intel MPI library. Therefore it will be investigated and evaluated here as well.

This document presents the details of the evaluation of these tools, and accordingly, a recommendation in regard to the choice of underlying tools for the toolkit is made. The evaluation of the tools underlies several criteria, formulated in the following questions:

1. Do we need root access?
2. Is a daemon necessary?
3. Is the tool usable with MPI / threads?
4. What metrics do I get?
5. What is the overhead?
6. What are the use-cases?
7. How can the tool be integrated?
   a) into the batch system
   b) for the user
8. Where is the tool available (architecture, OS, ...)?
9. Does the tool trace / sample?

10. How is the analysis data accessible?

These questions will be answered for the mentioned tool at the end of each tool-specific section.

Unless otherwise mentioned, for the benchmarks and tests the following system has been used:

- CPU: Workstation with Intel i5-6500 CPU (Skylake) at 3.20 GHz (1 socket, 4 cores).
- Memory (RAM): 8 GB.
- Software: GCC 4.8, Open MPI 1.10.

The benchmark test suite consists of the Berlin Quantum ChromoDynamics (BQCD) [1] and InterleavedOrRandom (IOR) [2] programs. We have run the two benchmarks with and without using the discussed tool, and compared the timing differences. For the use of toolkit, we set a limit of 3% added overhead caused by the tool as acceptable. The measured timings with different number of MPI tasks (if possible) are presented for each tool. The following color legend has been used for the percentage increase on the runtime for each experiment:

- red: increase in runtime more than 3%,
- orange: increase in runtime less than 3%,
- blue: decrease in runtime.

2 procfs

2.1 Tasks and Metrics

The `/proc` filesystem is a virtual filesystem that serves as an interface to read out or to manipulate process or system information. In Linux, every process possesses a directory under `/proc/pid/`, in which its information and characteristics can be retrieved on demand. Most of the files in this directory are empty (i.e. they have zero file size) and their information can only retrieved when the file is explicitly read from the kernel (for example using the `less` command). In the files `/proc/cpuinfo` and `/proc/meminfo` the static system information about CPU and memory are stored, respectively. The information stored in the procfs files is node-based.

The following files with their data and metrics are interesting for us:

- `/proc/pid/cgroups`
  Here, the specific information about the used cgroups of task / process ID is stored.
- `/proc/pid/stat`
  This file contains status information about the process PID, which is used for example by the command `ps`. The following parameters (which are a selection of all parameters) are stored in this file:
  - PID (process identifier) and PPID (parent process identifier) of the process,
  - number of minor and major page faults until now,
  - user and system time (measured in clock ticks),
  - number of threads spawned in this process (`num_threads`),
  - start time of the process after booting the system (`starttime`),
  - size of the virtual memory of the process in Bytes (`vsize`),
  - resident set size in pages (`rss`),
  - processor ID on which the process ID was located during the last time slice.
- `/proc/pid/statm`
  Returns information about the memory usage (in pages):
  - size of the whole process (`size`). This is the same as `vsize` in `/proc/pid/status`.
  - Resident set size (the same as `VmRss` in `/proc/pid/status`; in kbytes),
  - Amount of pages the code area as well as the data and stack area of the program uses (`text`, `data`).
- `/proc/pid/status`
  Provides much of the information of the files `/proc/pid/stat` and `/proc/pid/statm`. The advantage of the representation of the contents in this file is a better readability of the data. The following data (among others) can be retrieved:
  - name of the process or program name,
  - PID, PPID,
  - state of the process (i.e. running, sleeping, uninterruptible sleep, zombie, traced or stopped),
  - peak of the process virtual memory size (`VmPeak`; in kbytes),
  - size of the virtual memory of the process (`VmSize`; in kbytes),
  - high water mark, i.e. the peak RAM size the process used until now (`VmHWM`, in kbytes),

  - resident set size (`VmRSS`; in kbytes),
  - amount of memory of the data, stack and code area of the process (`VmData`, `VmStk`, `VmExe`; in kbytes),
  - total amount of used swap space (`VmSwap`; in kbytes),
  - number of threads of the process PID,
  - voluntary and involuntary context switches.

- `/proc/cpuinfo`
  Contains almost only static information about the CPU cores/hardware threads, such as,
  - CPU family, CPU model and model name, stepping, recent frequency of every core (dynamic feature), L3 cache size, number of cores, CPUID, instruction set architecture, cache alignment, address size, power management, apicid, number of CPU cores / hardware threads.

  This information can be used to summarize processor properties (for example in the expert mode).

- `/proc/loadavg`
  This file contains the averaged load information of the last 1, 5 and 15 minutes. The last number in the file output states the PID of the last created process.

- `/proc/meminfo`
  Contains static and dynamic information about the memory (RAM, virtual memory, swap):
  - total RAM memory of the system/node minus a few reserved bits and the amount of the kernel binary (`MemTotal`; in kbytes),
  - the whole free/unused RAM memory (`MemFree`; in kbytes),
  - active and inactive memory,
  - total and free amount of the swap space (`swap total`, `swap free`; in kbytes),
  - total and used amount of the vmalloc memory (`VmallocTotal`, `VmallocUsed`; in kbytes).

- `/proc/stat/`
  Creates a kernel / system report. For every core / hardware thread on the node we get information (line `cpu(n)`) which is also aggregated and summed (line `cpu`). Node wide information is also delivered:
  - number of clock ticks (`USER_Hz`, which is on most systems 1/100 of a second) in user, system, iowait and idle state. Furthermore, the parameters of virtual machines are listed in this file (time which is used for virtual machines; steal, guest),
  - number of context switches (`ctxt`),
  - number of pages which where paged in and paged out (`page`, not on all systems),
  - number of processes that are in the runnable state (`procs_running`),
  - number of processes that are blocked and are waiting for completing IO (`procs_-blocked`).

- `/proc/vmstat`
  Contains information about the virtual memory. The following metrics seem to be interesting for us:
  - amount of swapped in and swapped out data (over the whole uptime of the workstation/node; `pswpin`, `pswpout`),

- amount of paged in and paged out data (over the whole uptime of the workstation/node; `pgpgin`, `pgpgout`),
- amount of all and major page faults (over the whole uptime of the workstation/node; `pgfault`, `pgmajfault`),
- amount of successful and unsuccessful access to a NUMA node. A `NUMA_FOREIGN` follows a `NUMA_MISS` (see `numaset`, `numa_hit`, `numa_miss`, `numa_foreign`).

2.2 Benchmark Runs

In every of the two benchmark cases the appropriate program was run with and without reading procfs ten times and for every time series the arithmetic mean was calculated. In the BQCD case the runtime was measured by BQCD itself (in front of the slash in Table 1; 3rd and 4th columns) and by the shell command `time` (behind the slash in Table 1; 3rd and 4th columns). The variance was omitted, because of the small deviations between the runtimes of the test runs (with a maximum deviation of about five percent). In the last column of the table the relative deviations between the BQCD run without (in front of the slash) and with procfs (behind the slash) can be found. In the case of the IOR benchmark we only used the *single shared* file mode and we only measured the runtime with `time`, because IOR measures only in whole seconds, which is too coarse and in general the calculated deviations are therefore too large.

As well as in the BQCD case, the IOR test runs were made successively with one, two and four processes and these conditions are also valid for the benchmark tests in Chapters 3, 4, 5 and 7.

BQCD and IOR

BQCD was launched with the shell script `./job_proc.sh`, which has the following contents (in this example with four processes):

```
./job_proc_read.sh &
time -p mpirun -np 4 -bind-to core ./bqcd input.1nodes.1sockets.4cores
```

in which the Skript `job_proc_read.sh` contains the measuring procedure:

```
for i in $(seq 1 60)
do
    less /proc/cpuinfo >> output_proc.txt
    sleep 0.5
done
```

It can be seen, that BQCD was started with process binding on a per core basis. Furthermore, for simplicity, every 0.5 second the `cpuinfo` file was read out and the output was redirected into a file. After that the process was suspended for 0.5 seconds. This time interval was chosen in order to simulate a finer grained measurement than just 1 second. In the IOR case the method is analogous.

In Table 1 the arithmetic means of the different runs with no metric determination with procfs and with the metric determination with procfs for BQCD and IOR are given in which in the BQCD case the runtime values are measured by the program *and* the `time` shell command.

Program	MPI tasks	Without procfs	With procfs	Overhead
BQCD	1	29.05/29.19	29.28/29.42	0.79%/0.79%
	2	32.28/32.41	32.63/32.77	1.08%/1.13%
	4	54.50/54.64	55.21/55.26	1.30%/1.13%
IOR	1	7.43	7.31	-1.62%
	2	13.79	13.66	-0.94%
	4	26.44	26.30	-0.53%

Table 1: Benchmark results (in seconds) using procfs as performance metric collector for BQCD and IOR.

It can be seen from Table 1 that in all BQCD cases the relative "instrumented" benchmark times are not problematic and below the critical value of three percent. The values of IOR with the metric measurement with procfs are even better than the values of the "uninstrumented" IOR version. Interestingly, the run with fewer cores is faster than the run with the maximum number of cores on the workstation. This behavior is still under evaluation.

A problem in using procfs to measure the runtime of an application in the above fashion is, that waking up and getting into the run state takes longer than for example 0.5 seconds, hence the measurement will not take part exactly at that time stamp, because the process must after waking up get into the run state, which needs additional time. As a consequence a problem arises because of potentially unequaled measuring intervals.

2.3 Conclusion

- Do we need root access?
  No.
- Is a daemon necessary?
  No.
- Is the tool usable with threads / MPI?
  It can be used only node wise but with multithreading and multiprocessing.
- What metrics do I get?
  See Section 2.1.
- What is the overhead?
  No significant overhead (see Section 2.2).
- What are the use cases?
  Performance measurement and monitoring on the node level.
- How is the tool integrable into the batch system?
  It could be possible to read out the values of procfs in the prologue and / or epilogue script and calculate the difference of those values at the end of the job.
- How is the tool integrable for the user?
  In our test cases, the reading of the data was done using a background process.
- Where is the tool available (architecture, OS, ...)?
  procfs is part of the Linux kernel and only needs to be mounted.
- Does the tool trace/sample?
  Counting and sampling.

- How is the analysis data accessible?
  The output is on stdout/stderr (depends on the way the information is read out).

2.4 Remarks

The advantage of procfs in comparison to the tools from Sysstat (Section 5) is, that the data can be theoretically read out with an arbitrary rate and not just at integer timestamps (i.e., with integer interval lengths). Since most of the data of the Sysstat utilities can be found in procfs as well, for a fine granular measuring, using procfs seems to be more appropriate.

3 vmstat

3.1 Tasks and Metrics

`vmstat` creates a report about CPU, process, paging, swapping, block IO and disk activity. The report can be created every specified integer interval value and for a specified number of times ("infinity" is also possible).

The metrics which are provided by the tool (and which are important for us) are listed below.

- Processes:
  - number of processes in the states *runnable* or *waiting for getting CPU time* (parameter `r` in the output),
  - number of processes which are blocked in an uninterruptible sleep (waiting for IO) (parameter `b` in the output).
- Memory:
  - amount of used virtual memory (parameter `swpd`; in kbytes),
  - amount of free RAM memory (parameter `free`; in kbytes),
  - parameters buff, cache, inact, active (explanation see `sar -r`).
- Swap:
  - amount of memory which was read in / written out (in kbytes per second) from / to the disc (`swap in / swap out`).
- IO:
  - number of blocks which were read from / written to a block device per second (parameters `bi` and `bo`).
- CPU:
  - user time (including nice time; parameter `us`),
  - system time (parameter `sy`),
  - idle time (parameter `id`),
  - iowait (parameter `wa`),
  - steal (parameter `st`; see `sar -u`).
- Reporting selected disk properties and activities (option `-D`):
  - number of disks and partitions,
  - amount of the whole read operations,
  - merged reads and read sectors,
  - time for reading and writing (in milliseconds),
  - number of IO operations which are currently in action,
  - time spent in IO.
- Reporting selected system properties and activities (option `-s`):
  - total, free and used RAM memory (in kbytes),
  - active and inactive memory (in kbytes),

- buffered and cached memory (in kbytes),
- total, used and free amount of swap space (in kbytes),
- amount of interrupts since the system is up,
- number of context switches since the system is up,
- number of paging (in/out) since the system is up.

3.2 Installation and Requirements

In general `vmstat` is part of the Linux distribution and is "shipped" pre-installed. If the latter is not the case, it can be installed via the package manager.

3.3 Benchmark Runs

BQCD and IOR

In Table 2 the benchmark results without using `vmstat` (3rd column) and with using `vmstat` (4th column) are listed, while in 5th column the relative deviations are given.

Program	MPI tasks	Without vmstat	With vmstat	Overhead
BQCD	1	29.05 / 29.19	29.12 / 29.26	0.24% / 0.24%
	2	32.28 / 32.41	32.31 / 32.47	0.09% / 0.19%
	4	54.5 / 54.64	54.48 / 54.62	-0.04% / -0.04%
IOR	1	7.43	7.35	-1.08%
	2	13.79	13.79	0.0%
	4	26.44	26.33	-0.40%

Table 2: Benchmark results (in seconds) using `vmstat` as performance metric collector for BQCD and IOR.

Both benchmarks were run ten times (respectively for one, two and four tasks) and the arithmetic mean for each case was calculated. In Table 2 a negligible overhead running BQCD with `vmstat` in the case of using one and two processes could be observed, while in the case of four processes the arithmetic mean in the "instrumented" version is below the arithmetic mean in the "uninstrumented" one. In the case of IOR (*single shared file* scenario), the deviations are again negative, because the instrumented values are below the uninstrumented ones.

3.4 Conclusion

See Section 5.5, with the exception that this tool is part of most Linux distributions and was often mentioned in the online survey as installed.

3.5 Remarks

Similar to several other tools discussed in this document, there is no possibility to collect the data at non-integer intervalls.

4 GNU time

4.1 Tasks and Metrics

GNU time (located by default under `/usr/bin/time` in all Linux distributions, while `time` is a shell command with less functionality) is used to determine the runtime of a given program and other used resources. After calling `/usr/bin/time -v -p <executable> [arguments]` the user gets detailed information in the POSIX format.

The following list summarizes the metrics provided by GNU time:

- command being timed,
- user time, system time (in decimal seconds),
- percentage of the CPU this process used (for one core calculated as the sum of user and kernel time divided by the wall clock time),
- elapsed (wall clock) time (h:mm:ss or m:ss),
- maximum resident set size (kbytes),
- number of major (requiring IO) and minor (reclaiming a frame) page faults,
- number of voluntary and involuntary context switches,
- number of file system inputs and outputs while the lifetime of the process ,
- page size used by the operating system (in general 4096 bytes for Linux),
- exit status of the program (and not of GNU time itself).

The metrics *average shared memory size*, *average unshared data size*, *average unshared stack size*, *average resident set size*, *number of swaps*, *number of of sent and received socket messages*, as well as *amount of signals delivered* are not supported in Linux and are set to zero.

4.2 Benchmark Runs

BQCD

Both benchmarks were launched with `/usr/bin/time -v -p ./job_usrbintime.sh`. In the case of BQCD the job script `./job_usrbintime.sh` had the following contents (here an example with four processes):

```
mpirun -np 4 -bind-to core ./bqcd input.1nodes.1sockets.4cores
```

In Table 3 the arithmetic means of the different runs with no measurement with GNU time and with the measurement with GNU time are given for the case of BQCD. As it can be seen, the "instrumented" runtime has just a negligible or no overhead in every case and the worst overhead is below 0.6%, which is an acceptable value.

Program	MPI tasks	Without GNU time	With GNU time	Overhead
BQCD	1	29.05/29.19	29.12/29.25	0.24/0.20%
	2	32.28/32.41	32.27/32.43	-0.03/0.05%
	4	54.5/54.64	54.29/54.46	-0.39/-0.33%

Table 3: Benchmark results (in seconds) using GNU time as performance metric collector for BQCD.

IOR

The contents of the benchmark job script, from which IOR was called in the *single shared file mode*, was:

```
mpirun -np 4 ./src/ior -a POSIX -w -r -b 500m -t 128k -g -e -Z -i 3 \
       -o "test_ior.txt"
```

Program	MPI tasks	Without GNU time	With GNU time	Overhead
IOR	1	7.43	7.33	-1.35%
	2	13.79	13.76	-0.20%
	4	26.44	26.36	-0.29%

Table 4: Benchmark results (in seconds) using GNU time as performance metric collector for IOR.

4.3 Conclusion

- Do we need root access?
  No.
- Is a daemon necessary?
  No.
- Is the tool usable with threads/MPI?
  It is usable with threads and also with plain `mpirun` on a system. In combination with a batch job system problems are arising (wrong counts and wrong CPU utilization) which have to be evaluated.
- What metrics do I get?
  See Section 4.1.
- What is the overhead?
  In most cases there is just a low overhead, because the deviations between the instrumented and non instrumented program runs are all below the acceptable limit (see Section 4.2). The exception is the IOR benchmark with one core, where the deviation is above tolerance and which have to be evaluated.
- What are the use cases?
  GNU time is ideal to obtain a quick overview over the whole consumed time and selected consumed resources of a process/job.
- How is the tool integrable into the batch system?
  In SLURM it could be possible to put GNU time into the prologue and epilogue scripts. Since SLURM uses the same mechanism as GNU time for reading out the same accounting data with the `sacct` command (`wait3` and `getrusage` system calls), it is possible to use SLURM instead of GNU time.
- How is the tool integrable for the user?
  In the manual part it can be used via the command line with single and multithreaded applications and plain `mpirun`. It also works with batch scripts (c.f. child processes).
- Where is the tool available?
  The tool can be found in every standard Linux distribution since it is part of the Linux kernel.

- Does the tool trace/sample?
  Counting.
- How is the analysis data accessible?
  The output is written to stderr.

4.4 Remarks

In case of multithreading, the user and the system time of all threads are summed. As a consequence, in general one gets a greater count compared to only one thread (with the ratio being approximately the number of threads).

5 Sysstat

The Sysstat package [3] contains several utilities for monitoring system performance and usage activities. Its tools can be for example scheduled via cron jobs to collect and record the history of performance and activity data.

The package provides several tools, such as:

- `iostat` reports CPU statistics and input / output statistics for block devices and partitions,
- `mpstat` reports individual or combined processor related statistics,
- `pidstat` reports statistics of IO, CPU, memory, etc. for Linux tasks (processes),
- `sar` collects, reports and saves system activity information (see below a list of metrics collected),
- `sadc` is the system activity data collector, used as a backend for `sar`.

5.1 SAR

The System Activity Reporter (`sar`) records node-wise specific system parameters and creates an output after specified time intervals (for instance every second) for the specified number of times (or with no limit). Furthermore, the user can read "historical data" such as those collected via cron jobs and `sadc`. Here, we only consider the first case.

5.1.1 Metrics

The relevant metrics provided by `sar` are listed in the following.

- Averaged CPU utilization (averaged over all cores; using option `-u ALL`, to display all CPU fields)
  - user and system time (includes the time, spent running on virtual processors (call: `sar`); in decimal CPU seconds),
  - user and system time (excludes the time, spent running on virtual processors (call: `sar -u ALL`); in decimal CPU seconds),
  - `%iowait` and `%idle`: Percentage of runtime the process was in the idle state while (`iowait`) or while *not* waiting for the completion of an IO operation (`idle`).
- CPU utilization for all or selected cores (option `-P { cpu [,...] | ALL }`)
  With this option `sar` outputs among others the timestamp of the measurement, selected/specified core(s), `%user`, `%system`, `%iowait`, and `%idle` in a table to stdout. With `-P ALL`, `sar` creates a report for all cores / hardware threads and an averaged part for them. For example, with `-P 1,3`, `sar` creates a report only for cores numbered 1 and 3 and their respective averages.
- Memory utilization (option `-r`)
  - free (`kbmemfree`) and used RAM memory (`kbmemused`) (both in kbytes),
  - fraction of the RAM, which is used (`%memused`; in percent),
  - part of the memory that was used more recently and will be reclaimed only if necessary (`kbactive`; in kbytes),
  - part of the memory that was used less recently and can in general be requested for other tasks or processes (`kbinact`; in kbytes).

- Paging report (option `-B`)
  - amount of kbytes per second the system paged in/out from/to disk (`pgpgin/s`, `pgpgout/s`),
  - number of major page faults per second (`majflt/s`) and number of minor and major page faults per second of the system (`fault/s`).
- Swap space utilization report (option `-S`)
  - free and used swap space (`kbswpfree` and `kbswpused`; in kbytes),
  - used swap space in percent (`%swpused`).
- Load averages report (option `-q`)
  - average utilization of the system CPU(s) in the last 1, 5, or 15 minutes (`ldavg-1`, `ldavg-5`, `ldavg-15`, respectively).
- IO statistics report (option `-b`)
  - amount of data (in blocks of 512 bytes per second), which is loaded from / written to the devices (`bread/s` and `bwrtn/s`).
- Network utilization report (option `-n`)
  - sum of packets received and sent per second (`rxpck/s` and `txpck/s`),
  - amount of kbytes received and sent per second (`rxkB/s` and `txkB/s`).

5.1.2 Benchmark Runs

BQCD

Similar to the previous cases BQCD was run ten times and the arithmetic means were calculated in the case of one, two and four processes. BQCD was called with a measuring interval of 1 second and below an example call is listed:

```
/usr/bin/sar -u 1 60 > out_sar.txt&
time -p mpirun -np 4 -bind-to core ./bqcd input.1nodes.1sockets.4cores
```

The following values were obtained:

Program	MPI tasks	Without SAR	With SAR	Overhead
BQCD	1	29.05/29.19	29.14/29.26	0.31/0.24%
	2	32.28/32.41	32.41/32.57	0.40/0.48%
	4	54.50/54.64	54.57/54.72	0.13/0.14%

Table 5: Benchmark results (in seconds) using SAR as performance metric collector for BQCD.

In this examples no problematic overheads (positive deviations larger than 3%) running BQCD with `sar` were observed.

IOR

Program	MPI tasks	Without SAR	With SAR	Overhead
IOR	1	7.43	7.36	-0.94%
	2	13.79	13.81	0.15%
	4	26.44	26.36	-0.30%

Table 6: Benchmark results (in seconds) using SAR as performance metric collector for IOR.

In the case of IOR the benchmark runs were done in the *single shared* file mode. As in the previous case, no problematic deviations were observed.

5.2 pidstat

The `pidstat` tool creates a report for Linux processes and threads and a load report of recent CPU and IO utilization (disc and partitions). Without any option the user gets a list of processes (as in the case of top).

5.2.1 Metrics

The metrics of our interest are the following:

- Create a report for a process containing string (option `-C` *string*). Some important data in the report are:
  - timestamp (hh:mm:ss),
  - UID, PID,
  - `%usr` and `%sys` part of the process,
  - current core / hardware thread number the process is running on (`CPU`),
  - fraction of the CPU time the process is using (`%CPU`; in percent).
- Creating a CPU utilisation report (option `-u`)
  - see `pidstat -C`,
  - CPU, command: core / hardware thread number on which the *command* recently runs.
  - number of threads running in the process (additional flag `-v` is needed).
- Page fault and memory utilisation report (option `-r`)
  - minor and major page faults per second of the listed task / process (`minflt/s`, `maxflt/s`),
  - amount of virtual memory the task currently uses (`VSZ`; in kbytes),
  - resident set size of the task (`RSS`; in kbytes),
  - current memory utilization of the task in percent (`%MEM`).
- Generating an IO report (option `-d`)
  - amount of data loaded from / stored to the device caused by the task (`kB_rd/s`, `kB_wr/s`; in kbytes per second).

5.2.2 Benchmark Runs

Benchmarking `pidstat` led to similar results as in the `sar` case, with no significant overhead.

5.3 iostat

`iostat` creates reports about recent CPU and IO utilisation (disks and partitions). Its most important arguments are listed below.

Synopsis: `iostat [-cdtx]`

- `-c`: creates a CPU utilisation report,
- `-d`: creates an device / IO utilisation report,
- `-t`: additional output of the time stamp,
- `-x`: extended report statistics of `iostat`.

5.3.1 Metrics

The metrics collected by `iostat` are:

- user and system time, iowait and idle time percentages (CPU utilisation report),
- device to probe, transfers per second sent to the device (`tps`),
- number of finished read / write requests per device (`Blk_read/s`, `Blk_wrtn/s`; in blocks per second),
- total number of blocks read / written (`Blk_read`, `Blk_wrtn`).

5.4 Installation and Requirements

The Sysstat utilities can be found for example in the repositories of OpenSUSE, Debian and Ubuntu Linux distributions. To install `sar` manually, the classic "config; make; make install" chain has to be performed. Root access is needed to install `sar` manually or using the package manager.

5.5 Conclusion

- Do we need root access?
  Only for the installation of the tool.
- Is a daemon necessary?
  No.
- Is the tool usable with threads / MPI?
  Yes / yes, but only node-based.
- What metrics do I get?
  See Sections 5.1.1, 5.2.1, and 5.3.1.
- What is the overhead?
  No problematic overhead (see Section 5.1.2).
- What are the use cases?
  Performance measurement and monitoring at the node level.
- How is the tool integrable into the batch system?
  With for example SLRUM it could be possible to start / finish `sar` from a prologue / epilogue script. Another possibility is, that it can run node-wise and the metrics can be collected like in PerSyst.

- How is the tool integrable for the user?
  The test cases were done with the Sysstat tools as background processes.
- Where is the tool available (architecture, OS, ...)?
  The tool is available in standard Linux repositories.
- Does the tool trace/sample?
  Counting and sampling.
- How is the analysis data accessible?
  The output is written to stdout.

5.6 Remarks

With the `sar` monitoring tool it is not possible to report the values at non integer time stamps (intervall lengths). As the consequence the minimum time interval is one second. With this interval length it is not possible to get fine grained data. For example, `sar 2 3` creates a report three times every two seconds and `sar 2` creates a report every second second until the program is killed. Sameis true for the tools `iostat`, `mpstat`, `pidstat`, and `vmstat`.

6 Perf

6.1 Tasks

Perf is now part of the Linux kernel, thus should be available on all Linux clusters. If not, it can easily be installed over the command line with the package `linux-tools-generic`. Perf accesses hardware counters on each processor, but cannot be used across multiple nodes (i.e., with MPI). Perf can be used in several different manners, all listed in Listing 1. These tools need different permissions and will also show different detail depending on the granted permissions. The most interesting commands for us are:

- `perf stat`
  Generates a simple stdio output of the performance counter information (e.g., number of cycles per instruction, number of cache misses, branch misses or similar). The events to be counted and output can be chosen explicitly or a default set of counters can be shown. No root privileges are needed for a run with `perf stat`. A complete list of available user space events is given in Appendix A.

- `perf record`
  It is used for the collection of samples. With collecting samples, the data will be stored in a binary file (default `perf.data`, but can be changed with `-o <filename>`). To successfully trace applications, `perf record` has to be called as root. In order to get the data collected afterwards, the user needs to call, e.g., `perf report` for a general report of the overhead caused by single operations. Both `perf record` as well as `perf report` have a wide variety of options to be called. The events collected in `perf stat` can also be sampled in `perf record`. One can also get the time stamped records.

```
$ perf

 usage: perf [--version] [--help] COMMAND [ARGS]

 The most commonly used perf commands are:
  annotate        Read perf.data (created by perf record) and display
  annotated code
  archive         Create archive with object  files with build-ids
  found in perf.data file
  bench           General framework for benchmark suites
  buildid-cache   Manage <tt>build-id</tt> cache.
  buildid-list    List the buildids in a perf.data file
  diff            Read two perf.data files and display the differential profile
  inject          Filter to augment the events stream with additional information
  kmem            Tool to trace/measure kernel memory(slab) properties
  kvm             Tool to trace/measure kvm guest os
  list            List all symbolic event types
  lock            Analyze lock events
  probe           Define new dynamic tracepoints
  record          Run a command and record its profile into perf.data
  report          Read perf.data (created by perf record) and display the profile
  sched           Tool to trace/measure scheduler properties (latencies)
  script          Read perf.data (created by perf record) and display trace output
  stat            Run a command and gather performance counter statistics
  test            Runs sanity tests.
  timechart       Tool to visualize total system behavior during a workload
  top             System profiling tool.
```

Listing 1: Commands usable with perf.

One solution to the root privileges needed for an all-encompassing collection of data could be to modify the `/proc/sys/kernel/perf_event_paranoid` file (or similarly the sysctl setting with the same name), which sets the access rights for all users. When using the sampling mode with `perf record`, one has to be careful about the overheads, as the capture files can quickly become hundreds of MBs in size. It would however depends on the rate of the event tracing: the more frequent, the higher the overhead and larger `perf.data` file size.

Details about the events, including timestamps, the code path that has led to it, and other specific details can be collected. Also hardware counters can be recorded - with different settings, e.g., every x cycles, at every occurrence, or whenever the next xyz additions have been made.

However, this is not suitable for a timeseries database solution.

One important advantage of `perf` tracing (if available) instead of, e.g., `strace` is that perf buffers data in-kernel and thus reduces overhead as compared to `strace`. From Linux kernel 4.4 onwards, there are more advanced options of using `perf records` by using BPF programs.

`perf_events` has a built-in visualization tool called `timecharts`, as well as text-style visualization via its text user interface (TUI) and tree reports (see Ref. [4]).

6.2 Metrics

The types of events that can be measured by perf are:

- Hardware Events:
  These instrument low-level processor activity based on CPU performance counters. For example, CPU cycles, instructions retired, memory stall cycles, level 2 cache misses, etc. Some will be listed as Hardware Cache Events.
- Software Events:
  These are low level events based on kernel counters. For example, CPU migrations, minor faults, major faults, etc.
- Tracepoint Events:
  These are kernel-level events based on the `ftrace` framework. These tracepoints are placed in interesting and logical locations of the kernel, so that higher-level behavior can be easily traced. For example, system calls, TCP events, file system IO, disk IO, etc. These are grouped into libraries of tracepoints; e.g., "sock:" for socket events, "sched:" for CPU scheduler events.
- Dynamic Tracing:
  Software can be dynamically instrumented, creating events in any location. For kernel software, this uses the `kprobes` framework. For user-level software, `uprobes` is being used instead.
- Timed Profiling:
  Snapshots can be collected at an arbitrary frequency, using `perf record -F` *`Hz`*. This is commonly used for CPU usage profiling, and works by creating custom timed interrupt events.

A complete list of available user space events is given in Appendix A. This list may vary among different systems. Additional kernel space events are not presented here, as the collection of them would require either root access or further adaptions to the system.

6.3 Installation and Requirements

As perf comes pre-installed with most Linux distributions, there is usually no need to install it separately. Usage is straightforward with detailed man pages and further elaborate information in Ref. [5] and elsewhere.

6.4 Benchmark Runs

First experiments were conducted on the GWDG Scientific Compute Cluster. Timings were taken with GNU time, to keep the overhead calculations and comparable to those done for the other tools. The two tested settings were `perf stat` and `perf record`, both with no further options.

The output of the `perf stat` calls is directly delivered with the following text output.

```
$./ior -o test-filesystem -a POSIX -w -r -b 1024k -t 4k -F -g -e -Z -i 3

       161,379247      task-clock (msec)         #    0,490 CPUs utilized
              326      context-switches          #    0,002 M/sec
                2      cpu-migrations            #    0,012 K/sec
            2.751      page-faults               #    0,017 M/sec
      448.146.929      cycles                    #    2,777 GHz
  <not supported>      stalled-cycles-frontend
  <not supported>      stalled-cycles-backend
      765.162.013      instructions              #    1,71  insns per cycle
      147.940.746      branches                  #  916,727 M/sec
          365.621      branch-misses             #    0,25% of all branches
      0,329640875 seconds time elapsed
```

With this output, the user already gets a text-based overview on certain performance metrics. Very differently, `perf record` creates a database which can be read afterwards with `perf report`. This output delivered from `perf report` can either be viewed with a GUI or also be redirected to stdio with a simple flag. The output can be further refined through additional flags, but is rather overwhelming for an inexperienced HPC user. By redirecting the output of `perf report` to a temporary file, this file can be parsed and put into a better understandable format for the user.

The three versions were all run on Intel Sandy Bridge nodes and the median timing (real) from 10 runs is shown in Table 7, for both IOR and BQCD:

Program	MPI Tasks / Threads	Original	perf stat	Overhead	perf record	Overhead
BQCD (M)	2/8	14.4585	14.627	1.17%	18.583	28.53%
	3/8	19.289	19.2805	-0.04%	22.999	19.23%
	4/8	18.491	18.563	0.39%	22.7885	23.24%
	8/8	18.8405	18.7565	-0.45%	25.2315	33.92%
BQCD (A)	2/8	14.5998	14.7209	0.83%	18.8754	29.28%
	3/8	19.3832	19.1943	-0.98%	23.4066	20.75%
	4/8	18.545	18.5058	-0.22%	23.1927	25.06%
	8/8	19.011	18.811	-1.05%	25.136	32.22%
IOR (M)	2/1	5.2375	5.286	0.9%	5.241	0.06%
	3/1	4.5455	4.6475	2.2%	5.066	11.4%
	4/1	4.68	4.732	1.11%	5.163	10.3%
	8/1	4.9565	4.964	0.15%	5.547	11.91%
	10/1	5.3245	5.2355	-1.67%	5.89	10.62%
IOR (A)	2/1	5.3619	5.3458	-0.31%	4.977	-7.18%
	3/1	5.8935	4.7183	-19.95%	5.0509	-14.3%
	4/1	6.2892	4.7652	-24.23%	5.0943	-19%
	8/1	5.0335	4.9648	-1.36%	5.5153	9.57%
	10/1	5.4784	5.2717	-3.77%	6.1595	12.43%

Table 7: Benchmark results (in seconds) using perf as performance metric collector for BQCD and IOR. Median (M) and average (A) times from 10 runs.

In Table 7, we show both the median and the average times to emphasize the impact of single outliers on this platform.

6.5 Conclusion

1. Do we need root access?
   Not necessarily, but without the root permission certain counters can not be collected. However, this can be circumvented by setting the corresponding sysctl setting.

2. Is a daemon necessary?
   No.

3. Is the tool usable with MPI/threads?
   It is usable with MPI by starting it alongside each MPI process, i.e. `mpirun perf a.out`. Then, the aggregation of the collected metrics has to be done manually.

4. What metrics do I get?
   Various hardware counters, both from the user space and the kernel space, including but not limited to information on branch predictions, cache usage and misses, time, CPU information, context switches, cycles, memory stores and loads, and many more (see Appendix A for an example of user-space events).

5. What is the overhead?
   Highly dependent on the profiling style (`stat`, `record`, `trace`). The overhead can be decreased by reducing the sampling frequency or limiting the monitored events accordingly.

6. What are the use-cases?
   Single node jobs or system-wide performance profiling.

7. How is the tool integratable?
   Using command line with `perf <command> [options] <executable>`.

8. Where is the tool available (architecture, OS, ...)?
   Comes with the Linux kernel. The available hardware counters and their accessibility are vendor and version dependent, i.e., AMD has different counters than Intel and older Intel processors have generally less available counters than newer processors.

9. Does the tool trace/sample?
   Sampling is provided in all versions of perf. From kernel versions 4.4 and higher, perf also comes with a trace command. Even in the older version a call stack trace can be retrieved from `perf record` with according options set.

10. How is the analysis data accessible?
    `perf stat` delivers a stdio summary of the monitored events. `perf record` provides a data file that can be read with `perf report`. The latter can either be used with a GUI or the output can be redirected to stdio.

7 LIKWID

LIKWID [6, 7] consists of twelve tools, in which only the following tools are of interest to our use case:

- `likwid-perfctr`
  With this tool the user can configure and read the data of the hardware counters.
- `likwid-topology`
  Generates a report about the processor, cache and NUMA topology and data.
- `likwid-pin`
  Enables thread pinning/thread binding (Pthreads, OpenMP).
- `likwid-mpirun`
  Wrapper to start MPI and MPI/OpenMP applications (with explicit process binding).

We will present these tools in the following subsections.

7.1 likwid-perfctr

`likwid-perfctr` configures and reads out the hardware counters on x86 architectures. It contains the following four modes, which will be discussed subsequently:

- wrapper mode,
- stethoscope mode,
- timeline mode,
- marker API.

Wrapper mode

In this mode LIKWID gets the contents of the performance counters for the specified process without any need to change the code (in contrast for example to PAPI). With the command `likwid-perfctr -a` one gets a CPU dependent list of predefined performance metric groups (See appendix C for an Intel Core i5-6500 CPU). These predefined performance groups can be extended with self-defined performance groups. One can measure as much events as performance counters are built in.

The following two examples should explain the usage of `likwid-perfctr`.

- In this example a bubble sort program is measured on socket 0, core 0, and is pinned to these entities (flag `-C`). The performance group `L2CACHE` is used on these entities for bubble sort.

```
$ likwid-perfctr -C S0:0 -g L2CACHE ./bubblesort

CPU name:       Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz
CPU type:       Intel Skylake processor
CPU clock:      3.19 GHz

WARN: Linux kernel configured with paranoid level 1
WARN: Paranoid level 0 is required to measure Uncore counters
--------------------------------------------------------------
Die Sortierung mittels Bubblesort ist beendet!
```

```
-------------------------------------------------------------

Group 1: L2CACHE

+------------------------+---------+------------------+
|          Event         | Counter |       Core 0     |
+------------------------+---------+------------------+
| INSTR_RETIRED_ANY      |  FIXC0  |  185323159882    |
| CPU_CLK_UNHALTED_CORE  |  FIXC1  |   97560348322    |
| CPU_CLK_UNHALTED_REF   |  FIXC2  |   86936617208    |
| L2_TRANS_ALL_REQUESTS  |   PMC0  |    1241331181    |
| L2_RQSTS_MISS          |   PMC1  |     216486875    |
+------------------------+---------+------------------+

+----------------------+-----------+
|        Metric        |   Core 0  |
+----------------------+-----------+
|  Runtime (RDTSC) [s] |   27.6642 |
| Runtime unhalted [s] |   30.5640 |
|      Clock [MHz]     | 3582.0722 |
|          CPI         |    0.5264 |
|    L2 request rate   |    0.0067 |
|     L2 miss rate     |    0.0012 |
|     L2 miss ratio    |    0.1744 |
+----------------------+-----------+
```

Listing 2: Example output of `likwid-perfctr`.

In the first three output lines information about the processor is listed, while in the following two lines a warning is printed out followed by the output of the program. After that, the performance information follows. In every call of this tool the output is divided into a table with the raw event counts and another table with the derived metrics. If there are more than one cores, an additional statistic of the minimum, maximum and summed values of those cores is printed out. In the first table we have the number of instructions, which are finished and four other *Events*. In the metrics table, the runtime and the average clock rate plus the averaged CPI (Cycles Per Instruction) can be found. These are the metrics which are displayed for every group. `L2 miss rate` denotes the number of L2 misses based on all L2 accesses. The `L2 request rate` means the L2 requests per cycle.

- The second example shows how to pin the threads to the appropriate cores or hardware threads (option `-C`) and how to get the information of all `L2CACHE` metrics.

  ```
  likwid-perfctr -C 0-3 -g L2CACHE ./bubblesort
  ```

  The output of the first 5 lines is analogous to the above example, as well as the output of the program. After that, one can find two tables similar to above but with the difference, that the data is listed for all the cores listed in the command line. Furthermore, `likwid-perfctr` prints out the report of the cores/hardware threads (sum, min, max, avg) in another table below the two tables.

Timeline mode

In the timeline mode the user can explicitly choose the time resolution. For example, the user can switch the scan rate to five seconds, so that only every fifth second the data is collected from the hardware counters. The advantage is a lower system load because of smaller measure activities for the whole system and not just for one process. See the following example:

```
likwid-perfctr -C S0:0-3 -g BRANCH -t 5s
```

With `-t` one can specify the scan rate. The output channel is stderr.

Stethoscope mode

In this mode the user can access or monitor the resources (e.g. cores) for a given time without knowledge of the program. This is interesting for nodewide counting and sampling (among others).

Marker API

With the marker API of `likwid-perfctr` the user can measure specified regions of the code. Therefore one have to include special functions into the code like in the following example (this code stub is from the documentation of `likwid-perfctr`):

```
#include <likwid.h>

LIKWID_MARKER_INIT;
LIKWID_MARKER_THREADINIT;

LIKWID_MARKER_START("Compute");
// ...
// The code to measure
// ...
LIKWID_MARKER_STOP("Compute");
LIKWID_MARKER_CLOSE;
```

In this case only the marked region will be measured by `likwid-perfctr`.

7.2 likwid-topology

`likwid-topology` creates a textual report of the cores / hardware threads, caches, and the NUMA topology (in the latter two cases an ASCII graphic is generated). In its most general form (`likwid-toplogy -C -c -V 3 -g`), the user gets the following information:

- CPU name and type, stepping, current clock rate,
- topology of the CPUs (number of sockets, cores per socket, threads per core and the numbering of the hardware threads and their mapping to cores and sockets),
- cache topology: report about the cache levels with additional information for every level (amount of k/mbytes per cache / cache level, cache type, associativity of the cache, number of sets, cache line length, cache type (inclusive or non inclusive cache), shared by threads, cache group),
- NUMA topology: number of NUMA domains and the core assignment to a NUMA domain, free and total amount of RAM memory in MBytes (for a NUMA domain),
- graphical representation of the cache topology of every socket.

`likwid-topology` extracts the information from the `hwloc` library or the CPUID instruction.

7.3 likwid-pin

`likwid-pin` is used to pin threads to specified hardware threads / cores / sockets and NUMA regions without rewriting the code (system independent). It is possible to pin the threads explicitly to a predefined set of cores / hardware threads or automatically via a compact or scatter strategy. Furthermore, it is possible to specify regions of affinity. The user can specify nodes (N), sockets (S), NUMA regions (M) and cache groups (C). The number that follows that letter specifies for example the number of threads. For example `likwid-pin -c S0:4 ./stream_c.exe` specifies socket 0 with 4 threads.

This tool does not collect any metrics. `likwid-pin` explicitly supports Pthreads and the OpenMP implementations of Intel and GCC compilers.

7.4 likwid-mpirun

`likwid-mpirun` is a tool to start MPI jobs and to pin the processes and threads to nodes, sockets, cores or hardware threads, and it is based on `likwid-pin`. It also gets the values of (selected) metrics and has additionally the same functionality as `likwid-perfctr`. The two examples below should exemplify the usage:

- `likwid-mpirun -np 4 ./bqcd input.1nodes.1sockets.4cores`
  This command line starts the BQCD program with 4 processes and is just a MPI based job with no further pinning.
- `likwid-mpirun -np 4 -nperdomain S:2 ./bqcd input.1nodes.1sockets.4cores`
  In this example on a two socket machine on every socket two processes will be started and pinned with `likwid-pin` (implicit application). There are 4 domains, which can be specified: N (node), S (socket), C (last level cache), M (NUMA domain).

7.5 Benchmark Runs

We have performed benchmarking with `likwid-mpirun` to check, if measurements have a non negligible impact on the runtime.

BQCD

To benchmark BQCD in combination with LIKWID, we used the following command line (here an example with four processes).

```
time -p likwid-mpirun -nperdomain S:4 -g BRANCH ./bqcd input.1nodes.1sockets.4cores
```

The results are presented in Table 8.

Program	MPI Tasks	Without `likwid-mpirun`	With `likwid-mpirun`	Overhead
BQCD	1	29.05/29.19	29.09/30.35	0.14%/3.98%
	2	32.28/32.41	32.37/33.64	0.28%/3.81%
	4	54.50/54.64	54.60/55.86	0.18%/2.24%

Table 8: Benchmark results (in seconds) using `likwid-mpirun` as performance metric collector for BQCD.

As it can be seen, there is a non negligible overhead with running BQCD and `likwid-mpirun` in the case of one and two processes. In particular, using `likwid-mpirun` implies a non-negligible overhead for programs with a short runtime, because the wall-clock time values with "instrumentation" are significantly larger than the values measured by the application.

IOR

Program	MPI Tasks	Without `likwid-mpirun`	With `likwid-mpirun`	Overhead
IOR	1	7.43	8.45	13.73%
	2	13.79	14.9	8.05%
	4	26.44	27.29	3.22%

Table 9: Benchmark results (in seconds) using `likwid-mpirun` as performance metric collector for IOR.

In the case of the "instrumented" IOR results presented in Table 9, the deviations from the "uninstrumented" values are in two cases larger than ten percent, which is unsatisfactory for our use cases.

It was observed, that in all benchmark cases there was a difference of about one second between finishing the application and finishing `likwid-mpirun`. As the consequence, the relative deviation will decrease with increasing runtime.

7.6 Installation and Requirements

LIKWID can be installed with a "config; make; make install" routine. A detailed description can be found on the homepage of LIKWID. Since LIKWID have to use the MSR kernel module and the Uncore performance counters (in the PCI space), proper access permissions need to be set for LIKWID. One can also consider using the access daemon, `likwid-accessD`, which encapsulates this access and performs an address check for allowed registers, therefore removing the need for the root permission. However, since more recent kernels have tighter securities for accessing the MSR device, this will not be sufficient anymore, and the binary itself has to be registered. This is only possible on local file systems [7].

7.7 Conclusion

- Do we need root access?
  No, but the MSR kernel module has to be installed by root and access must be granted beforehands.
- Is a daemon necessary?
  Yes, if the access daemon is activated for security reasons.
- Is the tool usable with threads / MPI?
  Yes (see also `likwid-mpirun`).
- What metrics do I get?
  Run-time and hardware performance counter based metrics of the job (see appendix C); thread, cache and NUMA topology of the node.
- What is the overhead?
  Relatively high overhead (see Section 7.5).
- What are the use cases?
  Performance measurement with the use of hardware counters.

- How is the tool integrable into the batch system?
  In SLURM it could be possible to put `likwid-perfctr` into the prologue and/or epilogue scripts.
- How is the tool integrable for the user?
  See the examples above.
- Where is the tool available (architecture, OS, ...)?
  The tool can be built on any distribution with a recent GCC or CLANG compiler and the Linux kernel version 2.6 or newer without any changes.
- Does the tool trace/sample?
  Counting and tracing (the latter only with the marker API).
- How is the analysis data accessible?
  The output is done to stdout.

8 Darshan

8.1 Tasks and Metrics

Darshan [8, 9] is a scalable IO profiling tool, designed to capture an accurate picture of application's IO behaviour, including properties such as patterns of access within files, with minimum overhead. It can be used to investigate and tune the IO behavior of complex HPC applications.

The Darshan source tree is organized into two primary components:

1. *darshan-runtime*: Darshan runtime framework necessary for instrumenting MPI applications and generating IO characterization logs. It should be installed on the system where you intend to collect IO characterization information.

2. *darshan-util*: Darshan utilities for analyzing the contents of a given Darshan IO characterization log. The darshan-util package can be installed and used on any system regardless of where the logs were originally generated.

The *darshan-runtime* only instruments MPI applications (the application must at least call `MPI_Init` and `MPI_Finalize`). However, it captures both MPI-IO and POSIX file accesses.

8.2 Installation and Requirements

The *darshan-runtime* requires an MPI C compiler. For the installation of the *darshan-utils* the zlib development headers and library are needed. Optionally for generating PDF reports, libbz2 development headers and library, Perl, pdflatex, gnuplot 4.2 or later, and epstopdf are used.

8.3 Benchmark Runs

Run-time comparisons shown in Table 10 were performed on an eight node FDR InfiniBand system, each node equipped with two 6-core 2.4 GHz Intel Xeon Haswell processors and 64 GB RAM. BeeGFS was used as the parallel file system. IOR multi-stream benchmark was performed by the following command:

```
./ior -v -o <testdir> -a POSIX -w -r -b 1g -t 1m -F -g -e -Z -i 3
```

Program	MPI Tasks	OpenMP Threads	Without Darshan	With Darshan	Overhead
BQCD	96	1	61.58 s	61.67 s	0.1%
	48	1	60.09 s	60.05 s	-0.1%
	48	2	34.61 s	34.87 s	0.8%
	1	1	48.55 s	49.08 s	1.1%
	1	12	5.24 s	5.26 s	0.4%
	24	1	60.93 s	60.91 s	0.0%
IOR	96	1	(w) 5071.69 MiB/s	5027.17 MiB/s	-0.9%
			(r) 15638.61 MiB/s	16672.4 MiB/s	6.6%
	48	1	(w) 5419.85 MiB/s	5555.02 MiB/s	2.5%
			(r) 19885.65 MiB/s	20367.78 MiB/s	2.4%
	24	1	(w) 5934.18 MiB/s	6221.74 MiB/s	4.8%
			(r) 19775.40 MiB/s	19789.57 MiB/s	-0.1%

Table 10: Benchmark results (average of five runs) using Darshan as performance metric collector for BQCD and IOR.

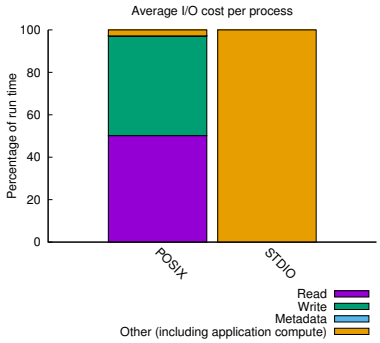
Only a negligibly small overhead in a range of a few percents can be observed in our benchmark runs.

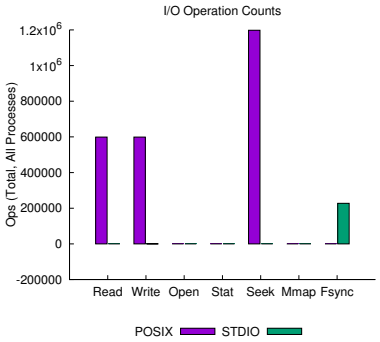
8.4 Example Output

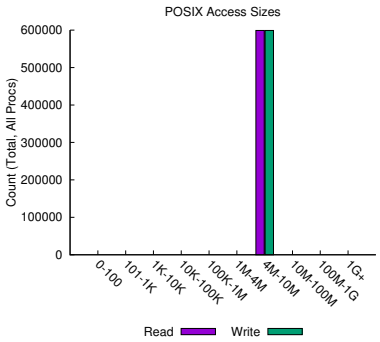
An example graphical summary of the IO activity for an IOR run generated by the *darshan-job-summary.pl* script from the *darshan-utils* as PDF document is shown on the next pages.

jobid: 30850	uid: 2126	nprocs: 5	runtime: 1141 seconds

I/O performance *estimate* (at the POSIX layer): transferred 2734 MiB at 5254.66 MiB/s







Most Common Access Sizes
(POSIX or MPI-IO)

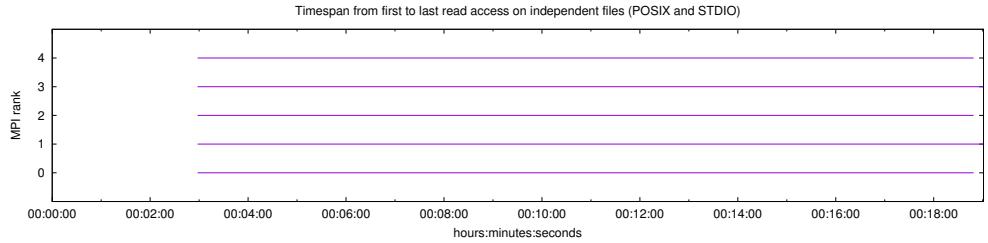
	access size	count
POSIX	5242880	1198080

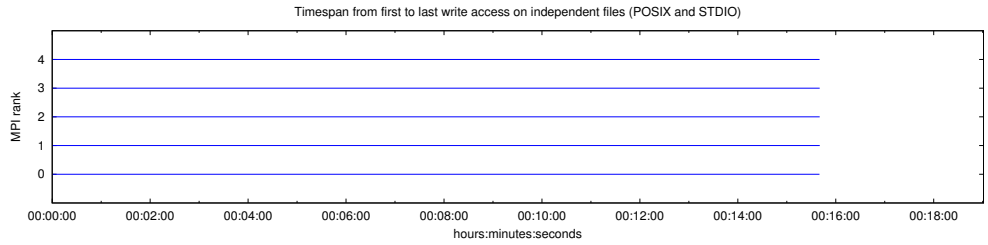
File Count Summary
(estimated by POSIX I/O access offsets)

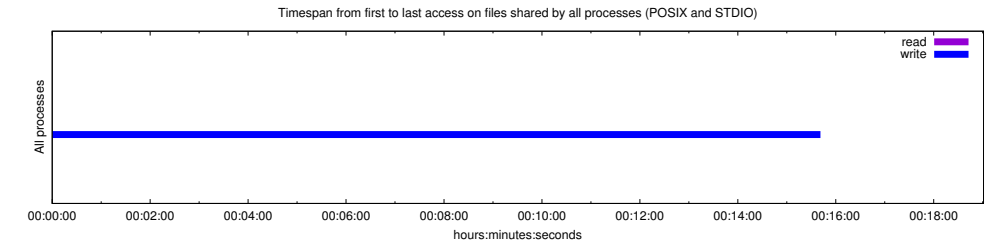
type	number of files	avg. size	max size
total opened	7	140G	195G
read-only files	1	195G	195G
write-only files	5	118G	195G
read/write files	1	195G	195G
created files	6	131G	195G

./ior -v -o /mnt/beegfs/bzadmgla/testdir_striped/testfile -a POSIX -w -r -b 195g -t 5m -F -g -e -Z -i 3

Figure 1: Example Darshan output, page 1/3.







Average I/O per process (POSIX and STDIO)

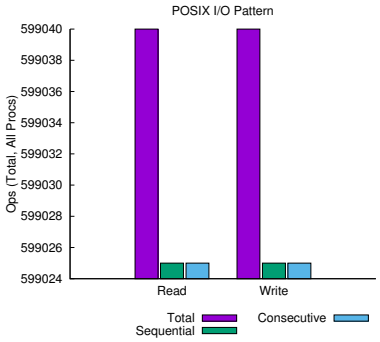
	Cumulative time spent in I/O functions (seconds)	Amount of I/O (MB)
Independent reads	572.5042094	599040
Independent writes	536.1680784	599040
Independent metadata	0.1305908	N/A
Shared reads	0	0
Shared writes	-7753.6137556	0.000521469116210937
Shared metadata	0	N/A

Data Transfer Per Filesystem (POSIX and STDIO)

File System	Write		Read	
	MiB	Ratio	MiB	Ratio
UNKNOWN	0.00261	0.00000	0.00000	0.00000
/mnt/beegfs	2995200.00000	1.00000	2995200.00000	1.00000

./ior -v -o /mnt/beegfs/bzadmgla/testdir_striped/testfile -a POSIX -w -r -b 195g -t 5m -F -g -e -Z -i 3

Figure 1 (cont.): Example Darshan output, page 2/3.



sequential: An I/O op issued at an offset greater than where the previous I/O op ended.
consecutive: An I/O op issued at the offset immediately following the end of the previous I/O op.

Variance in Shared Files (POSIX and STDIO)

File Suffix	Processes	Fastest			Slowest			σ	
		Rank	Time	Bytes	Rank	Time	Bytes	Time	Bytes
...<STDERR>	5	1	0.000000	0	0	0.000037	357	0	143
...<STDOUT>	5	0	-9121.970555	2.4K	2	-4560.962334	0	1.82e+03	951

./ior -v -o /mnt/beegfs/bzadmgla/testdir_striped/testfile -a POSIX -w -r -b 195g -t 5m -F -g -e -Z -i 3

Figure 1 (cont.): Example Darshan output, page 3/3.

8.5 Conclusion

1. Do we need root access?
   No.

2. Is a daemon necessary?
   No.

3. What are requirements?
   MPI C compiler, zlib development headers and library, optionally Perl, libbz2 development headers and library, pdflatex, gnuplot 4.2 or later, epstopdf.

4. Is the tool usable with MPI/threads?
   Yes, even necessary, as Darshan only instruments MPI applications.

5. What metrics do I get?
   IO access patterns at the POSIX and MPI-IO layers and IO cost per process in percentage of the run time of the application.

6. What is the overhead?
   Overhead is generally low.

7. What are the use-cases?
   IO access pattern and statistics of IO activity.

8. How is the tool integratable?
   Statically linked executables must be instrumented at compile time. For dynamically linked executables, Darshan relies on the `LD_PRELOAD` environment variable to insert instrumentation at run-time. The Darshan instrumentation is saved in a job specific log file, which can be read by the *darshan-parser* utility to obtain a complete, human-readable, text-format dump of all information contained in a log file. This step can be done automatically by the batch system in a job epilogue script.

9. Where is the tool available (architecture, OS, ...)?
   The tool has been tested on general Linux clusters, Cray architectures, and IBM Blue Gene systems.

10. Does the tool trace/sample?
    Darshan instruments applications to provide coarse-grained IO summary. Starting in version 3.1.3, Darshan also allows for full tracing of application IO workloads using the newly developed Darshan eXtended Tracing (DxT) instrumentation module.

11. How is the analysis data accessible?
    An application that has been instrumented with Darshan will produce a single log file each time it is executed. This log summarizes the IO access patterns used by the application. Darshan provides a collection of tools for parsing and summarizing log files.

9 Intel MPI

9.1 Tasks and Metrics

To collect data about communication information and communication patterns it is possible to use Intel MPI's *Statistics Gathering Mode*. The control of collecting information is done via setting appropriate environment variables and the data is collected between calls of the `MPI_Init` and `MPI_Finalize` functions.

A detailed listing of all relevant parameters can be found in Section 4.6 of the Intel MPI Library Developer Reference [10].

The report functionality can be switched on / off and controlled with the following environment variables:

- `I_MPI_STATS` switches report generation on and off,
- `I_MPI_STATS_SCOPE` selects the subsystem (and perhaps the communication functions), which should be reported,
- `I_MPI_STATS_BUCKET` defines for example the range of the message sizes, which should be reported, and the communicator size.

These environment variables are the most important variables to control the statistic functionality of Intel MPI. With the first one, the report functionality can be switched on or off. With the second variable, it is possible to select the general communication pattern which should be analyzed and with what a coarse analysis is possible. The following keywords describe the general communication patterns:

- `all`: collect data about all communication operations (default),
- `coll`: collect data of all collective operations,
- `p2p`: collect data of all point to point operations.

Additionally the user can specify concrete communication operations, which should be measured (for example `MPI_Bcast` to measure just the broadcast operations). Communication operations, which are not listed, are omitted.

With `I_MPI_STATS_BUCKET` it is possible to set the message and communicator sizes. With this property the user can for example set a range of message sizes, which should be evaluated. Messages, whose sizes are not in this range, are omitted.

There are two output formats: *native* and *IPM*. In the *native* format, which is available in plain text, the user gets information about the data transfer from every process to the other processes (in MBytes and amount of transfers). This is further divided in two tables of point to point and collective communications. In the first kind of communication, the user gets information of all point to point operations defined in the MPI standard (e.g., in the case of `I_MPI_STATS_SCOPE=all` the amount of transferred volume, the number of calls of the appropriate function and the min/avr/max/total transfer time in milliseconds). The user obtains similar information in the case of the collective communication mode. The *IPM* format is a more compact and a portable variant of the native data format. The default name of the output file is `stats.txt`, which can be changed by setting the `I_MPI_STATS_FILE` environment variable.

9.2 Installation and Requirements

Because of the multiple steps required to install the Intel MPI we refer to the Intel MPI Installation Guide [11]. For earlier versions of Intel MPI see the corresponding guides.

9.3 Benchmark Runs

In contrast to the other chapters of this deliverable, the STREAM benchmark was used to examine the runtime, when the Intel MPI *Statistics Gathering Mode* is applied. The STREAM benchmark was compiled with the Intel Compiler 15.0.3 and the Intel MPI library 5.0.3 was used in this benchmark case. The benchmark was repeated ten times with and without the reporting ability of Intel MPI in the case of 16, 32 and 64 tasks. The results are presented in Table 11. It can be

Program	MPI tasks	Without Intel statistics	With Intel statistics	Overhead
STREAM	16	34.41	34.41	0.0%
	32	17.49	17.53	0.23%
	64	8.93	9.0	0.78%

Table 11: Benchmark results (in seconds) using Intel MPI's statistics gathering mode as performance metric collector for STREAM.

seen, that there is only a negligible overhead in the case of the `STREAM` benchmark.

9.4 Conclusion

- Do we need root access?
  No.
- Is a daemon necessary?
  No.
- Is the tool usable with threads/MPI?
  Yes / yes.
- What metrics do I get?
  See Section 9.1.
- What is the overhead?
  No significant overhead (see Section 9.3).
- What are the use cases?
  Performance measurement of the network / MPI communication.
- How is the tool integrable into the batch system?
  Similar to the others tools, the results can be possibly gathered using prologue and/or epilogue scripts of the batch system.
- How is the tool integrable for the user?
  See Section 9.1.
- Where is the tool available (architecture, OS, ...)?
  It is available under Windows and Linux on x86 architectures and it is part of the Intel MPI library. It can be installed either as stand-alone or as part of the Intel Parallel Studio XE Cluster Edition, which is available in many of the surveyed institutions.
- Does the tool trace/sample?
  Counting.
- How is the analysis data accessible?
  The output is written to stdout.

10 Open | SpeedShop

10.1 Tasks

Open | SpeedShop (O | SS) [12] offers several different *experiments* with which profiling data can be collected. Each of these experiments is started in the same fashion:

```
oss<experiment> ‘‘<execution command>’’ [parameters]
```

Even though the program is run with a wrapper, it is suggested to compile the application with the `-g` option.

The experiments supplied by O | SS are:

- `pcsamp` periodic sampling of program counters,
- `usertime` periodic sampling of the call path,
- `hwc` counting of hardware events,
- `hwcsamp` hardware event sampling (max 6 events),
- `hwctime` hardware counters plus call path sampling,
- `io` accumulated wallclock measurement of IO including unique IO call paths,
- `iop` IO without individual call paths,
- `iot` IO plus additional information on bytes moved, file names, etc.,
- `mpi` timings and unique call paths for all MPI routines,
- `mpip` lightweight MPI,
- `mpit` tracing of MPI calls,
- `mem` tracking of memory leaks, memory allocation, call paths, etc.,
- `pthreads` timings and unique call paths for POSIX thread functions,
- `omptp` reports task idle, barrier and barrier wait times per OpenMP thread and attributes these to OpenMP parallel regions.

These experiments collect the profiling data in different databases (one per experiment). A GUI can be launched in order to visualize the data in the different databases. Some of the experiments also have a CLI, or directly output some information to the console after the successful completion of the program, and the “application’s symbol information is saved into the experiment output file so that the performance reports can be generated from the performance data file alone” (Ref. [12], p. 10). All information from the databases can also be exported as a CSV file, for which plotting instructions are supplied in the O | SS tutorials. Additionally, two further interfaces are supplied: the immediate command interface (batch), and a python scripting API.

Each experiment consists of *collectors* and *views*. *Collectors* define the different data sources, such as program counter samples, call stack samples, hardware counters, or traces of library routines. According to [12] p. 10, it is possible to implement multiple collectors per experiment, as some experiments already do so. On the other hand, *views* specify the aggregation and presentation of the data. The default aggregation is the sum across all nodes or threads, but the thread or node-specific data is still accessible. The data in the databases can be accessed and visualized either via the GUI or a CLI. Some experiments additionally print a default report to

Experiment	Sampling	Tracing	PAPI	CLI	GUI	stdio
pcsamp	✓				✓	✓
usertime	✓		✓	✓	✓	✓
hwc	✓		✓	✓	✓	
hwcsamp	✓		✓	✓	✓	
hwctime	✓		✓	✓	✓	✓
io		✓		✓	✓	
iop		✓		✓	✓	
iot		✓		✓	✓	
mpi		✓		✓	✓	
mpip		✓		✓	✓	
mpit		✓		✓	✓	
mem		✓		✓	✓	
pthreads		✓		✓	✓	
omptp						
cuda		✓		✓	✓	

Table 12: Accessibility of O | SS experiments.

stdio. Table 12 shows which experiment data is accessible in what way. It is possible to sample up to 6 hardware counter events at the same time, but `hwc` only samples one hardware counter.

The experiment `io` monitors several different IO operations, all listed in the specification, but does not record the arguments. `iot` additionally records the arguments of each call. Similarly, `iop` is a lightweight profiling solution that does not record timing of each individual call, but rather of all calls with the same name.

When using O | SS with OpenMP, thread wait time will only be detected if the program was built with the OMPT enhanced OpenMP runtime library [13]. Then, there are two different types of support available:

1. `omptp`: an OpenMP specific profiling experiment, or,
2. integration of the information gathered from the OpenMP runtime (through OMPT tool) into existing displays and experiments (Ref. [12], p. 90).

O | SS also offers the possibility of using and writing plugins. Newer versions of O | SS rely on the Component Based Tool Framework (CBTF), enabling a on the fly collection of the data in a database over a TCP/IP network from all nodes and threads, instead of writing local files and then aggregating these at the end of the application run.

10.2 Metrics

Depending on the experiment used, one can obtain information on:

- program counters,
- hardware counters,
- call path sampling,
- IO,
- MPI call tracing,

- memory tracking,
- Pthread call paths and timings,
- OpenMP idle, barrier and barrier wait times.

The availability of different hardware counters depends on the underlying hardware and the PAPI installation used. In Appendix B a list of usable PAPI events is given.

10.3 Installation and Requirements

The following experiences were made with O|SS versions 2.3.0 and 2.3.1 on the GWDG Scientific Compute Cluster with LSF Batch System.

O|SS is installed by compiling from source. No root access is needed for the installation. O|SS has many dependencies, most of which are installed by default during the installation process. During our installation attempts, some of these dependencies were not correctly identified and they needed to be installed explicitly. The whole installation process is divided into three parts; a Krell root installation, a CBTF installation, and the O|SS installation. The convenience script that installs all three instances at once cannot be recommended, as it does not stop the build if one of the steps fails. This in particular can be troublesome because the installation process takes usually several hours. The installation scripts are not logged, and that has to be taken care of additionally.

In the LSF batch system, O|SS cannot be used with the standard MPI run command `mpirun.lsf`. This is an issue known to the developers and it should be resolved in one of the next releases, as the LSF integration is still recent (at the time of writing).

O|SS relies on MRNet, which builds an underlying network for the aggregation of the measured data. MRNet is already known to have several issues, such as not finding a necessary script (`mrnet_commnode`). However, a workaround is addressed in the MRNet user guide. MRNet also needs a topology file for the creation of the subnetwork. In the best case, the topology file is automatically generated with the delivered `topology_generator`. Nevertheless in the case of LSF, the topology file could not be automatically generated. An (undocumented) option `-topology` lets the user hand over a self-designed topology file (the structure of which can be found in the MRNet guide). For LSF, a simple python script can be written to generate a (very basic) topology file.

One major drawback of O|SS is the (a priori) fact that only one instance of O|SS can be run per CWD. MRNet creates a file `attachBE_connections`, which has no further identifier. Thus, if multiple instances of O|SS experiments are to be run, they will overwrite each other and potentially use a wrong file, leading to an error or an undesired behaviour. This directly leads to another point: cleaning up is necessary after each run of O|SS to delete temporary files like the aforementioned or the topology file.

There seems to be a substantial problem exists when using O|SS across multiple nodes. In general, our tests showed several issues with files, shared objects or even the O|SS binary itself not being found. We were not be able to clarify so far where the problem lies, i.e., whether it is a batch system problem, a general problem, a configuration problem or any other.

Dependencies

O|SS relies on a series of other tools to collect the data, including Dyninst [14], MRNet [15], libmonitor [16] and PAPI [17]. A further list of packages is given in the O|SS installation guide, of which all except one (CMake) had to be downloaded individually.

O | SS requires the GNU compilers to build. Currently, it does not build with PGI, Intel, Cray, or any other compiler. In our tests, the package built with GNU supported performance analysis of applications compiled using Intel, PGI, Pathscale, and Cray compilers. Nonetheless, O | SS has to be offered in all combinations of compilers and MPI versions available, to reliably work in all experiments. To ensure this, further investigations needs to be done to see whether one needs a complete installation for each compiler-MPI combination, or if there is an easier way to establish the availability for all combinations.

10.4 Benchmark Runs

The experiments with O | SS were completed on only one node but with multiple MPI jobs, for the reasons mentioned above. For this reason, the number of threads per MPI process was always stable at one OpenMP thread per MPI process. The experiments were conducted on the GWDG Scientific Compute Cluster, but this time using nodes containing 20 Intel E5-2650 v3 CPUs.

All timings given in Tables 13 and 14 are the median *real* times of 10 runs from `/usr/bin/time` in seconds. We observed significant overheads regarding our benchmarks.

BQCD

Experiment	MPI Processes	OpenMP Threads	Without O | SS	With O | SS	Overhead
`ossio`	2	1	35.7565	50.017	40%
	3	1	36.155	50.583	40%
	4	1	36.7945	51.212	39%
	8	1	42.012	57.399	37%
	10	1	44.381	58.813	33%
	20	1	46.536	63.727	37%
`osspcsamp`	2	1		54.3965	52%
	3	1		55.7565	54%
	4	1		56.3705	53%
	8	1		60.898	45%
	10	1		60.525	36%
	20	1		67.8235	46%
`osshwcsamp`	2	1		54.6805	53%
	3	1		55.9625	55%
	4	1		56.786	54%
	8	1		61	45%
	10	1		62.156	40%
	20	1		68.18750	47%

Table 13: Benchmark results using O | SS as performance metric collector for BQCD: median times from 10 runs in seconds.

IOR

Experiment	MPI Processes	OpenMP Threads	Original	With O | SS	Overhead Factor
ossmpi	2	1	0.3195	11.5575	36.17
	3	1	0.3885	11.5935	29
	4	1	0.4755	11.6245	24.45
	8	1	0.924	12.435	13.46
	10	1	1.0925	12.717	11.64
	20	1	2.1335	15.0275	7.04
ossio	2	1		12.8685	40.27
	3	1		12.954	33.34
	4	1		13.2455	27.86
	8	1		14.621	15.82
	10	1		15.126	13.85
	20	1		19.009	8.91
osspcsamp	2	1		6.681	20.91
	3	1		7.369	18.97
	4	1		8.466	17.80
	8	1		12.81	13.86
	10	1		11.29	10.33
	20	1		15.5105	7.27
osshwcsamp	2	1		6.743	21.10
	3	1		7.231	18.61
	4	1		7.5705	15.92
	8	1		8.591	9.30
	10	1		11.519	10.54
	20	1		15.673	7.35

Table 14: Benchmark results using O | SS as performance metric collector for IOR: median times from 10 runs in seconds.

10.5 Conclusion

1. Do we need root access?
   No.
2. Is a daemon necessary?
   No.
3. Is the tool usable with MPI/threads?
   Yes, it even offers special wrappers for the usage with MPI.
4. What metrics do I get?
   Depending on the type of experiment used, one can get information on program counters, hardware counters, call path sampling, IO, MPI call tracing, memory tracking, pthread call paths and timings, OpenMP idle, and barrier and barrier wait times.
5. What is the overhead?
   Strongly dependent on type of the experiment used, but generally high.

6. What are the use-cases?
   Usable for profiling applications.

7. How is the tool integratable in the batch system and for the user?
   O|SS is used over the command line with any executable. In total there are four user interface options: batch, command line interface, graphical user interface, and Python scripting API.

8. Where is the tool available (architecture, OS, ...)?
   O|SS is usable on both single node and large scale Intel, AMD, ARM, Intel Xeon Phi, PowerPC, Power8, GPU processor based systems, as well as Cray and IBM Blue Gene platforms.

9. Does the tool trace/sample?
   Both, depending on the experiment.

10. How is the analysis data accessible?
    Traces are accessible in Open Trace Format (OTF). Other data is stored in a special format, which can be viewed with one of the delivered viewers or exported as CSV with visualization instructions. Some experiments also have an ASCII report printed directly to stdout.

10.6 Remarks

The shared library support in O|SS is limited and the normal manner of running experiments does not work. One must link the collectors into the static executable. Currently O|SS has static support on Cray and the Blue Gene P/Q platforms. For that, one must relink the application with the `osslink` command to add support for the collectors. However, when the execution of the statically linked executable with the O|SS collectors is linked, the workflow would be different. Since more flexible convenience scripts cannot be used, in order to change the arguments to the experiments one must set an environment variable (Ref. [12], p. 133).

11 Metrics Overview

In this section we briefly list the essential metrics for our use case, as well as the usage outline of aforementioned tools in order to obtain the corresponding metric(s).

Base information

- Requested and received number of nodes,
- requested walltime,
- requested memory and number of cores,

CPU: static data (`lscpu` and partly `/procfs/cpuinfo`)

- CPU model,
- number of sockets on a node, cores per socket, hardware threads per core,
- number of NUMA nodes and which cores are belonging to them,
- maximum (turbo boost frequency) and minimum CPU frequency,
- cache sizes (L1d/L1i, L2, L3),
- instruction set architecture (SSE, AVX, AVX2),
- cache alignment.

CPU: dynamic data (`lscpu` and `/proc/cpuinfo`)

- Core frequency

Memory: static data (`/proc/meminfo` or GNU time)

- `MemTotal`, `SwapTotal`, page size

Memory: dynamic data (`pidstat`, `sar`, `/proc/meminfo`, `/proc/pid/stat`)

- `MemFree` (`/proc/meminfo`, `sar -r`),
- `MemUsed` per node (`sar -r`),
- `SwapFree`, `SwapUsed`, `SwapUsed` in percent (`/proc/meminfo`, `sar -S`),
- `SwapUsed`, `SwapUsed` in percent (`sar -S`),
- the total number of swaps of the completed process (GNU time),
- active and inactive memory (`/proc/meminfo`, `sar -r`),
- buffered and cached memory (`/proc/meminfo`, `sar -r`),
- high water mark of RAM / maximum resident set size (VmHWM in `/proc/pid/status`, GNU time),
- current resident set size (`VmRss` in `/proc/pid/status`, `pidstat -r`),
- current virtual memory size (`VmSize` in `/proc/pid/status`, `pidstat -r`),
- peak virtual memory size (`VmPeak` in `/proc/pid/status`),
- `numa_hit`, `numa_miss`, `numa_foreign` (`/proc/vmstat`).

Processes, jobs (GNU time, `pidstat`, `sar`, or procfs)

- PID of the job (`pidstat -C`, `pgrep`),
- CPU number(s), on which the job runs (`pidstat -C`),
- number of threads in this process (`/proc/pid/status` or `pidstat -v`),
- wall clock time of the completed process/job (summary given by `GNU time`),
- user and system time (without the time running on a virtual processor; `sar -u`, GNU time, procfs),
- user and system time (including the time running on a virtual processor; `sar -u`, procfs),
- iowait and idle time (`sar -u`, procfs),
- voluntary and involuntary context switches (summary given by `GNU time`, `/proc/pid/status`, `pidstat -v`).

Paging

- Major and minor page faults (the sum over the process is given by GNU time, `/proc/pid/stat`, `pidstat -r`, `sar -B`),
- page in / page out (`sar -B`),

IO (disc and network)

- File system input / output (GNU time),
- `bread/s`, `bwrtn/s`: total amount of data, which is send to or received from disc (`sar -b`),
- `si`, `so`: amount of memory which was swapped from / to disk (per second; `vmstat`),
- `rxpck/s`, `txpck/s`: amount of packets, which are sent or received in one second (`sar -n`),
- `rxkB/s`, `txkB/s`: amount of kbytes, which are sent or received in one second (`sar -n`),
- data transfer from one process to another (specified) process (in MB; Intel MPI),
- data transfer actions from one process to another (specified) process (Intel MPI),
- data transfer and data transfer actions listed by MPI functions (Intel MPI),
- amount of time the data transfer needed (min/max/avrg/total; Intel MPI),
- size of every message (Intel MPI).

System

- Load average (if non-shared nodes; 1, 5, 15 minutes; `/proc/loadaverage`, `sar -q`, `uptime`).

Hardware counters

- See Appendices A, B (Perf and PAPI events) and C (Hardware Performance Groups and Counters of LIKWID).

12 Conclusion

In the first part of Deliverable 2.2, a selection of tools (based on the preceding evaluation in Deliverable 2.1 – "Concise Overview of Performance Metrics and Tools") as well their primary usage examples have been studied. This will help towards establishing a functional specification to be used for the implementation of the backend of the toolkit.

From our investigations and the assessment the following tools seem to be more suitable to be used for the toolkit:

- **GNU time** for CPU and wall-clock times and memory usage information,
- The **proc file system** for CPU time information and memory usage,
- **Perf**'s sub command `perf stat` for performance counter information for non-MPI cases,
- **LIKWID** for hardware performance counters for MPI and non-MPI applications,
- **Darshan** for detailed IO statistics and patterns for MPI applications.

It is worth mentioning that `sar`, `pidstat`, `mpstat`, `iostat`, and `vmstat` are appropriate tools for the backend as well. Nevertheless, they collect their information from procfs and gather metrics only in intervals of integer time stamps. Therefore, it might be more preferable to gather information directly from the procfs.

As also measured during our benchmarks, the sub command `perf record` has a very high overhead for the toolkit. Similarly, Open|SpeedShop shows a substantial overhead; additionally we had to address several software dependency issues during its installation.

Intel MPI (if present) provides a simple usage model for MPI performance analysis. Nonetheless, one drawback of Intel MPI is that the binaries need to be compiled by the Intel compilers, which might fail as for example in the BQCD benchmark program, and may not produce the optimal binaries for the users. This would severely limit the generality of the usage of the toolkit.

Another point of concern for our use cases is that the desired tool should have the possibility of integration with the batch system in conjunction with a system-wide performance data gathering daemon or framework. This will impose additional restrictions, since the tool should initialize, run, integrate the results, and terminate (or eventually fail) silently. GNU time and the proc filesystem due to their usage models are seemingly better choices regarding this matter. A further study regarding this theme will be presented in the second part of this deliverable.

List of Tables

A Perf User Space Events

The following list has been generated on a Dell Latitude E7470 laptop with an Intel Core i5-6200U CPU.

```
  branch-instructions OR branches                    [Hardware event]
  branch-misses                                      [Hardware event]
  bus-cycles                                         [Hardware event]
  cache-misses                                       [Hardware event]
  cache-references                                   [Hardware event]
  cpu-cycles OR cycles                               [Hardware event]
  instructions                                       [Hardware event]
  ref-cycles                                         [Hardware event]
  alignment-faults                                   [Software event]
  bpf-output                                         [Software event]
  context-switches OR cs                             [Software event]
  cpu-clock                                          [Software event]
  cpu-migrations OR migrations                       [Software event]
  dummy                                              [Software event]
  emulation-faults                                   [Software event]
  major-faults                                       [Software event]
  minor-faults                                       [Software event]
  page-faults OR faults                              [Software event]
  task-clock                                         [Software event]
  L1-dcache-load-misses                              [Hardware cache event]
  L1-dcache-loads                                    [Hardware cache event]
  L1-dcache-stores                                   [Hardware cache event]
  L1-icache-load-misses                              [Hardware cache event]
  LLC-load-misses                                    [Hardware cache event]
  LLC-loads                                          [Hardware cache event]
  LLC-store-misses                                   [Hardware cache event]
  LLC-stores                                         [Hardware cache event]
  branch-load-misses                                 [Hardware cache event]
  branch-loads                                       [Hardware cache event]
  dTLB-load-misses                                   [Hardware cache event]
  dTLB-loads                                         [Hardware cache event]
  dTLB-store-misses                                  [Hardware cache event]
  dTLB-stores                                        [Hardware cache event]
  iTLB-load-misses                                   [Hardware cache event]
  iTLB-loads                                         [Hardware cache event]
  node-load-misses                                   [Hardware cache event]
  node-loads                                         [Hardware cache event]
  node-store-misses                                  [Hardware cache event]
  node-stores                                        [Hardware cache event]
  branch-instructions OR cpu/branch-instructions/    [Kernel PMU event]
  branch-misses OR cpu/branch-misses/                [Kernel PMU event]
  bus-cycles OR cpu/bus-cycles/                      [Kernel PMU event]
  cache-misses OR cpu/cache-misses/                  [Kernel PMU event]
  cache-references OR cpu/cache-references/          [Kernel PMU event]
  cpu-cycles OR cpu/cpu-cycles/                      [Kernel PMU event]
  cstate_core/c3-residency/                          [Kernel PMU event]
  cstate_core/c6-residency/                          [Kernel PMU event]
  cstate_core/c7-residency/                          [Kernel PMU event]
  cstate_pkg/c2-residency/                           [Kernel PMU event]
  cstate_pkg/c3-residency/                           [Kernel PMU event]
  cstate_pkg/c6-residency/                           [Kernel PMU event]
  cstate_pkg/c7-residency/                           [Kernel PMU event]
  cycles-ct OR cpu/cycles-ct/                        [Kernel PMU event]
  cycles-t OR cpu/cycles-t/                          [Kernel PMU event]
  el-abort OR cpu/el-abort/                          [Kernel PMU event]
  el-capacity OR cpu/el-capacity/                    [Kernel PMU event]
  el-commit OR cpu/el-commit/                        [Kernel PMU event]
  el-conflict OR cpu/el-conflict/                    [Kernel PMU event]
  el-start OR cpu/el-start/                          [Kernel PMU event]
  instructions OR cpu/instructions/                  [Kernel PMU event]
  intel_bts//                                        [Kernel PMU event]
  intel_pt//                                         [Kernel PMU event]
  mem-loads OR cpu/mem-loads/                        [Kernel PMU event]
  mem-stores OR cpu/mem-stores/                      [Kernel PMU event]
  msr/aperf/                                         [Kernel PMU event]
  msr/mperf/                                         [Kernel PMU event]
  msr/pperf/                                         [Kernel PMU event]
  msr/smi/                                           [Kernel PMU event]
  msr/tsc/                                           [Kernel PMU event]
  tx-abort OR cpu/tx-abort/                          [Kernel PMU event]
  tx-capacity OR cpu/tx-capacity/                    [Kernel PMU event]
  tx-commit OR cpu/tx-commit/                        [Kernel PMU event]
  tx-conflict OR cpu/tx-conflict/                    [Kernel PMU event]
  tx-start OR cpu/tx-start/                          [Kernel PMU event]
  rNNN                                               [Raw hardware event descriptor]
  cpu/t1=v1[,t2=v2,t3 ...]/modifier                  [Raw hardware event descriptor]
  mem:<addr>[/len][:access]                          [Hardware breakpoint]
```

Listing 3: Perf user space events.

B PAPI Events

Below, PAPI events on an Intel Xeon E5-2650 v4 CPU are listed.

```
PAPI_L1_DCM  0x80000000  Yes   No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_L2_ICM  0x80000003  Yes   No   Level 2 instruction cache misses
PAPI_L2_TCM  0x80000007  Yes   No   Level 2 cache misses
PAPI_L3_TCM  0x80000008  Yes   No   Level 3 cache misses
PAPI_CA_SNP  0x80000009  Yes   No   Requests for a snoop
PAPI_CA_SHR  0x8000000a  Yes   No   Requests for exclusive access to shared cache line
PAPI_CA_CLN  0x8000000b  Yes   No   Requests for exclusive access to clean cache line
PAPI_CA_INV  0x8000000c  Yes   No   Requests for cache line invalidation
PAPI_CA_ITV  0x8000000d  Yes   No   Requests for cache line intervention
PAPI_L3_LDM  0x8000000e  Yes   No   Level 3 load misses
PAPI_TLB_IM  0x80000015  Yes   No   Instruction translation lookaside buffer misses
PAPI_L1_LDM  0x80000017  Yes   No   Level 1 load misses
PAPI_L1_STM  0x80000018  Yes   No   Level 1 store misses
PAPI_L2_LDM  0x80000019  Yes   No   Level 2 load misses
PAPI_L2_STM  0x8000001a  Yes   No   Level 2 store misses
PAPI_PRF_DM  0x8000001c  Yes   No   Data prefetch cache misses
PAPI_MEM_WCY 0x80000024  Yes   No   Cycles Stalled Waiting for memory writes
PAPI_STL_ICY 0x80000025  Yes   No   Cycles with no instruction issue
PAPI_STL_CCY 0x80000027  Yes   No   Cycles with no instructions completed
PAPI_FUL_CCY 0x80000028  Yes   No   Cycles with maximum instructions completed
PAPI_BR_CN   0x8000002b  Yes   No   Conditional branch instructions
PAPI_BR_NTK  0x8000002d  Yes   No   Conditional branch instructions not taken
PAPI_BR_MSP  0x8000002e  Yes   No   Conditional branch instructions mispredicted
PAPI_TOT_INS 0x80000032  Yes   No   Instructions completed
PAPI_LD_INS  0x80000035  Yes   No   Load instructions
PAPI_SR_INS  0x80000036  Yes   No   Store instructions
PAPI_BR_INS  0x80000037  Yes   No   Branch instructions
PAPI_RES_STL 0x80000039  Yes   No   Cycles stalled on any resource
PAPI_TOT_CYC 0x8000003b  Yes   No   Total cycles
PAPI_L2_DCA  0x80000041  Yes   No   Level 2 data cache accesses
PAPI_L2_DCR  0x80000044  Yes   No   Level 2 data cache reads
PAPI_L3_DCR  0x80000045  Yes   No   Level 3 data cache reads
PAPI_L2_DCW  0x80000047  Yes   No   Level 2 data cache writes
PAPI_L3_DCW  0x80000048  Yes   No   Level 3 data cache writes
PAPI_L2_ICH  0x8000004a  Yes   No   Level 2 instruction cache hits
PAPI_L2_ICA  0x8000004d  Yes   No   Level 2 instruction cache accesses
PAPI_L3_ICA  0x8000004e  Yes   No   Level 3 instruction cache accesses
PAPI_L2_ICR  0x80000050  Yes   No   Level 2 instruction cache reads
PAPI_L3_ICR  0x80000051  Yes   No   Level 3 instruction cache reads
PAPI_L3_TCA  0x8000005a  Yes   No   Level 3 total cache accesses
PAPI_L2_TCW  0x8000005f  Yes   No   Level 2 total cache writes
PAPI_L3_TCW  0x80000060  Yes   No   Level 3 total cache writes
PAPI_REF_CYC 0x8000006b  Yes   No   Reference clock cycles
PAPI_L1_DCM  0x80000000  Yes   No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes   Yes  Level 2 data cache misses
PAPI_L2_ICM  0x80000003  Yes   No   Level 2 instruction cache misses
PAPI_L1_TCM  0x80000006  Yes   Yes  Level 1 cache misses
PAPI_L2_TCM  0x80000007  Yes   No   Level 2 cache misses
PAPI_L3_TCM  0x80000008  Yes   No   Level 3 cache misses
PAPI_CA_SNP  0x80000009  Yes   No   Requests for a snoop
PAPI_CA_SHR  0x8000000a  Yes   No   Requests for exclusive access to shared cache line
PAPI_CA_CLN  0x8000000b  Yes   No   Requests for exclusive access to clean cache line
PAPI_CA_INV  0x8000000c  Yes   No   Requests for cache line invalidation
PAPI_CA_ITV  0x8000000d  Yes   No   Requests for cache line intervention
PAPI_L3_LDM  0x8000000e  Yes   No   Level 3 load misses
PAPI_TLB_DM  0x80000014  Yes   Yes  Data translation lookaside buffer misses
PAPI_TLB_IM  0x80000015  Yes   No   Instruction translation lookaside buffer misses
PAPI_L1_LDM  0x80000017  Yes   No   Level 1 load misses
PAPI_L1_STM  0x80000018  Yes   No   Level 1 store misses
PAPI_L2_LDM  0x80000019  Yes   No   Level 2 load misses
PAPI_L2_STM  0x8000001a  Yes   No   Level 2 store misses
PAPI_PRF_DM  0x8000001c  Yes   No   Data prefetch cache misses
PAPI_MEM_WCY 0x80000024  Yes   No   Cycles Stalled Waiting for memory writes
PAPI_STL_ICY 0x80000025  Yes   No   Cycles with no instruction issue
PAPI_FUL_ICY 0x80000026  Yes   Yes  Cycles with maximum instruction issue
PAPI_STL_CCY 0x80000027  Yes   No   Cycles with no instructions completed
PAPI_FUL_CCY 0x80000028  Yes   No   Cycles with maximum instructions completed
PAPI_BR_UCN  0x8000002a  Yes   Yes  Unconditional branch instructions
PAPI_BR_CN   0x8000002b  Yes   No   Conditional branch instructions
PAPI_BR_TKN  0x8000002c  Yes   Yes  Conditional branch instructions taken
PAPI_BR_NTK  0x8000002d  Yes   No   Conditional branch instructions not taken
PAPI_BR_MSP  0x8000002e  Yes   No   Conditional branch instructions mispredicted
PAPI_BR_PRC  0x8000002f  Yes   Yes  Conditional branch instructions correctly predicted
PAPI_TOT_INS 0x80000032  Yes   No   Instructions completed
PAPI_LD_INS  0x80000035  Yes   No   Load instructions
PAPI_SR_INS  0x80000036  Yes   No   Store instructions
PAPI_BR_INS  0x80000037  Yes   No   Branch instructions
PAPI_RES_STL 0x80000039  Yes   No   Cycles stalled on any resource
PAPI_TOT_CYC 0x8000003b  Yes   No   Total cycles
PAPI_LST_INS 0x8000003c  Yes   Yes  Load/store instructions completed
PAPI_L2_DCA  0x80000041  Yes   No   Level 2 data cache accesses
PAPI_L3_DCA  0x80000042  Yes   Yes  Level 3 data cache accesses
PAPI_L2_DCR  0x80000044  Yes   No   Level 2 data cache reads
PAPI_L3_DCR  0x80000045  Yes   No   Level 3 data cache reads
PAPI_L2_DCW  0x80000047  Yes   No   Level 2 data cache writes
PAPI_L3_DCW  0x80000048  Yes   No   Level 3 data cache writes
PAPI_L2_ICH  0x8000004a  Yes   No   Level 2 instruction cache hits
PAPI_L2_ICA  0x8000004d  Yes   No   Level 2 instruction cache accesses
PAPI_L3_ICA  0x8000004e  Yes   No   Level 3 instruction cache accesses
PAPI_L2_ICR  0x80000050  Yes   No   Level 2 instruction cache reads
PAPI_L3_ICR  0x80000051  Yes   No   Level 3 instruction cache reads
PAPI_L2_TCA  0x80000059  Yes   Yes  Level 2 total cache accesses
PAPI_L3_TCA  0x8000005a  Yes   No   Level 3 total cache accesses
PAPI_L2_TCR  0x8000005c  Yes   Yes  Level 2 total cache reads
```

```
PAPI_L3_TCR  0x8000005d  Yes   Yes  Level 3 total cache reads
PAPI_L2_TCW  0x8000005f  Yes   No   Level 2 total cache writes
PAPI_L3_TCW  0x80000060  Yes   No   Level 3 total cache writes
PAPI_SP_OPS  0x80000067  Yes   Yes  Floating point operations; optimized to count scaled single precision vector operations
PAPI_DP_OPS  0x80000068  Yes   Yes  Floating point operations; optimized to count scaled double precision vector operations
PAPI_VEC_SP  0x80000069  Yes   Yes  Single precision vector/SIMD instructions
PAPI_VEC_DP  0x8000006a  Yes   Yes  Double precision vector/SIMD instructions
PAPI_REF_CYC 0x8000006b  Yes   No   Reference clock cycles
```

Listing 4: PAPI events.

C Hardware Performance Groups and Counters of LIKWID

The following predefined hardware performance groups and metrics can be obtained by LIKWID.

- `BRANCH`: CPI, Branch rate, Branch misprediction rate, Branch misprediction ratio, Instructions per branch.
- `CLOCK`: Uncore Clock [MHz], CPI, Energy [J], Power [W], Energy DRAM [J], Power DRAM [W].
- `DATA`: Load to store ratio.
- `FALSE_SHARE`: Local LLC false sharing [MByte], Local LLC false sharing rate.
- `FLOPS_AVX`: Packed SP MFLOP/s, Packed DP MFLOP/s.
- `FLOPS_DP`: DP MFLOP/s, AVX DP MFLOP/s, Packed MUOPS/s, Scalar MUOPS/s.
- `FLOPS_SP`: SP MFLOP/s, AVX SP MFLOP/s, Packed MUOPS/s, Scalar MUOPS/s.
- `L2`: L2D load bandwidth [MBytes/s], L2D load data volume [GBytes], L2D evict bandwidth [MBytes/s], L2D evict data volume [GBytes], L2 bandwidth [MBytes/s], L2 data volume [GBytes].
- `L2CACHE`: L2 request rate, L2 miss rate, L2 miss ratio.
- `L3`: The same as for L2, but just for L3.
- `L3CACHE`: The same as for L2, but just for L3.
- `TLB_INSTR`: L1 ITLB misses, L1 ITLB miss rate, L1 ITLB miss duration [Cyc].
- `TLB_DATA`: L1 DTLB load misses, L1 DTLB load miss rate, L1 DTLB load miss duration [Cyc], L1 DTLB store misses, L1 DTLB store miss rate, L1 DTLB store miss duration [Cyc].

References

[1] "BQCD download : High Performance Computing : Universität Hamburg" [Online]. URL: https://www.rrz.uni-hamburg.de/services/hpc/bqcd (Retrieved: 01 August 2017).

[2] LLNL. "LLNL/ior: Parallel filesystem I/O benchmark" [Online]. URL: https://github.com/LLNL/ior (Retrieved: 01 August 2017).

[3] S. Godard. "SYSSTAT" [Online]. URL: http://sebastien.godard.pagesperso-orange.fr/ (Retrieved: 01 August 2017).

[4] Brendan Gregg. "Linux perf Examples" [Online]. URL: http://www.brendangregg.com/perf.html#Visualizations (Retrieved: 01 August 2017).

[5] Brendan Gregg. "Linux Performance" [Online]. URL: http://www.brendangregg.com/linuxperf.html (Retrieved: 01 August 2017).

[6] J. Treibig, G. Hager, and G. Wellein. "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments". In: *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*. San Diego CA, 2010.

[7] RRZE-HPC. "RRZE-HPC/likwid: Performance monitoring and benchmarking suite" [Online]. URL: https://github.com/RRZE-HPC/likwid (Retrieved: 01 August 2017).

[8] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. "Understanding and Improving Computational Science Storage Access Through Continuous Characterization". In: *Trans. Storage* 7.3 (October 2011), 8:1–8:26. ISSN: 1553-3077. DOI: 10.1145/2027066.2027068. URL: http://doi.acm.org/10.1145/2027066.2027068.

[9] "Darshan" [Online]. URL: https://www.mcs.anl.gov/research/projects/darshan/ (Retrieved: 01 August 2017).

[10] Intel Corporation. "Intel® MPI Library Developer Reference for Linux OS". Tech. rep. May 10, 2017. URL: https://software.intel.com/en-us/mpi-developer-reference-linux (Retrieved: 01 August 2017).

[11] Intel Corporation. "Intel® MPI Library for Linux OS Installation Guide". Tech. rep. May 10, 2017. URL: https://software.intel.com/sites/default/files/managed/7d/40/intelmpi-2017-update2-install-guide-linux.pdf (Retrieved: 01 August 2017).

[12] Contributions from Krell Institute, LANL, LLNL, SNL. "Open|SpeedShop Reference Guide". Version 2.3.0. November 12, 2016. URL: https://openspeedshop.org/wp-content/uploads/2016/11/OpenSpeedShop_Reference_Guide_v230_v2.pdf (Retrieved: 01 August 2017).

[13] Alexandre Eichenberger et al. "OMPT and OMPD: OpenMP Tools Application Programming Interfaces for Performance Analysis and Debugging". Tech. rep. April 24, 2013. URL: http://www.openmp.org/wp-content/uploads/ompt-tr.pdf (Retrieved: 01 August 2017).

[14] "Paradyn/Dyninst - Welcome | Putting the Performance in High Performance Computing" [Online]. URL: http://www.dyninst.org/ (Retrieved: 01 August 2017).

[15] "MRNet - Multicast Reduction Network" [Online]. URL: http://www.paradyn.org/mrnet/ (Retrieved: 01 August 2017).

[16] HPCToolkit. "HPCToolkit/libmonitor: HPCToolkit performance tools: libmonitor - a substrate for monitoring tools" [Online]. URL: https://github.com/HPCToolkit/libmonitor (Retrieved: 01 August 2017).

[17] "PAPI" [Online]. URL: http://icl.utk.edu/papi/ (Retrieved: 01 August 2017).