PfiT - The Developer's Guide

Documentation of the PfiT toolkit internals

August 19, 2020

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

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

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

Chapter 1

Introduction

High-Performance-Computing (HPC) has become a standard research tool in many scientific disciplines. Research without at least supporting HPC calculations is becoming increasingly rare in the natural and engineering sciences. On top of that, new disciplines are discovering HPC as an asset to their research, for example in the areas of bioinformatics and social sciences. This means that more and more scientists without a deep understanding of the architecture and the functioning of such systems start using HPC resources. This knowledge gap is further enlarged as the complexity of HPC resources increases and gains significant importance in the field of performance engineering.

Most scientists that are new to HPC run their applications on local Tier 3 systems and are satisfied if their research problem can be solved on an available system in an acceptable time frame. The missing knowledge with respect to performance measurements will often lead to a lock-in, because they are not able to scale their calculations to a Tier 2 or Tier 1 compute resource. At the same time, Tier 3 compute centers typically lack sufficient human resources to work with each user individually on application performance. In order to increase awareness for performance issues and enable users to assess possible gains from performance improving measures, systematic, unified, and easily understandable information on performance parameters should be provided across all scientific communities. This especially pertains to the performance parameters of HPC jobs and the importance of performance engineering since HPC cluster are very expensive resources. This applies to the procurement of the hard- and software as well as for the service of the system (especially the power supply), in which e.g. the overall energy costs amount is located in a five year life cycle in the order of several hundred thousand euros for a typical Tier 3 system. To reduce these costs, jobs should have for example lower run time to save energy or less waiting time until starting the job for better utilization of the cluster. These goals can e.g. be reached, if the user gets insight into the runtime behaviour of its program for example via different kinds of reports, including additional automatic evaluation, interpretation and presentation of the performance metric values.

Although the usage of many performance measurement tools is mostly straightforward, the serious disadvantage is the missing explanation of the results in a clear and simple manner. Often the interpretation of generated reports requires expert knowledge – this makes the profiling for normal cluster users nearly useless since they usually don't have

the background to interpret the results. By automatically assembling all data provided by available tools into a single centrally organized framework it will be much easier for the user (and for administrators, too) to identify potential performance bottlenecks or pathological cases. In addition, with the help of an all-in-one monitoring, profiling and reporting tool, the user acceptance for code optimizations might be drastically increased, especially when the code tuning shows a considerable performance boost. Additionally, the interface may also help to overcome the gap of understanding and communication between experts and end users by incorporating all data into a shared documentation system.

In order to achieve these goals a monitoring, profiling and reporting tool set, partly based on existing solutions, was implemented in the scope of this project. This tool set will automatically collect performance metrics and present them to researchers in easy to understand summaries or as a timeline (in appropriate reports). The tool set is completed by a documentation and best practices information, detailing, as applicable, measures regarding further investigation of the problem, recommended changes to the job submission, and promising performance engineering targets.

This document presents the developer's guide of the *PfiT* system. In this guide the implementation details of the products written within the project will be presented as well as for example collected metrics and their datatypes. This guide serves as the source for the future developing processes.

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

Chapter 2

Collectors

2.1 PfiTCollect

In this section the most important information about the sources and the collected metrics of the metric collector *PfiTCollect* is presented. The metrics will be collected by *PfiTCollect* reading the predefined sources every predefined time step (for example every 10 seconds). After collecting the metrics, they will be stored in the timeseries database (in our case *InfluxDB*) via the `curl` command. The collecting and sending procedure will take a fraction of a second and after that the collector will be sent to sleep and will be woken up after the end of the predefined time step. In every time step the metrics are collected and pushed to the *InfluxDB* database with the `curl` command.

2.1.1 PfiTCollect - Measurements

In this section the measurements as well as the tags and field keys of the measurements collected by *PfiTCollect* are listed, to give the interested user an overview over the metrics, which are collected by *PfiTCollect* and their sources. We will start with the list of the measurements and the sources of the metrics in this measurements (see table **??**).

After listing the measurements and their metric sources, the tags and field keys of every measurement with the corresponding data type will be listed below in the tables 2.3 to ??. In this table all measurements of the used community and project written *Telegraf* plugins are listed together with the metrics (tags and field keys) collected by the appropriate *Telegraf* plugin. It must be mentioned, that *PfiTcollect* does not collect all metrics, *Telegraf* collects and stores in the timeseries database *InfluxDB*. Only those metrics will be collected, which are needed for the different report types to save storage space. These metrics are marked in column five with an asterics.

Measurement	Source
cpu	/proc/cpuinfo
	/proc/stat
mem	/proc/meminfo
swap	/proc/swap
system	/proc/loadavg
	/proc/uptime
kernel	/proc/stat
gpu	nvidia-smi
beegfs_clients	beegfs-ctl
infiniband	perfquery
nfs	/proc/net/rpc/nfs
diskio	/proc/diskstats (scratch)
pfituprocstat	/proc/stat
processes	/proc/stat

Table 2.1: Table of the measurements *PfiTCollect* collects and the sources of the field keys in the measurements.

2.1.2 PfiTCollect - The source

And now to something completely different ... In this subsection some important parts of the *PfiTCollect* code will be described. This will be done by means of the `main` function. On the basis of this function the principal steps are described and if necessary a closer look to important functions will be done. It should be mentioned again, that exactly one *PfiTCollect* instance runs on every node of the cluster or on a selected subset of nodes of the cluster and the metrics will be collected every predefined time step. After collecting the metrics of one time step they will be pushed by every *PfiTCollect* instance into the timeseries database *InfluxDB*. The principal steps in `main` are as follows:

- Read in the configuration file of *PfiTCollect* and the architecture file,
- get the current/initial timestamp (in nanoseconds),
- iterate until an error occurs or *PfiTcollect* will be interrupted
  - read at the time stamp the different metrics from the sources on the node where *PfiTCollect* runs on, for example
    - jobid,
    - system metrics (CPU, Memory, Swap),
    - IO metrics (BeeGFS, NFS, scratch),
    - network metrics (Infiniband),
    - and if available metrics of the GPUs.
  - After collecting these metrics they will be pushed into the timeseries database (in our case *InfluxDB*).
  - get new timestamp (in nanoseconds)
  - Send *PfiTCollect* to sleep for a specified duration (e.g. 10 seconds).

Plugin	Tag	Metric	Datatype	*PfiTCollect metric/tag*
CPU	cpu		String	-
	host		String	*
	mode		String	-
	path		String	-
		usage_guest	float	*
		usage_guest_nice	float	*
		usage_idle	float	*
		usage_iowait	float	*
		usage_irq	float	*
		usage_nice	float	*
		usage_softirq	float	*
		usage_steal	float	*
		usage_system	float	*
		usage_user	float	*
Mem		active	integer	*
		available	integer	*
		available_percent	float	*
		buffered	integer	*
		cached	integer	*
		free	integer	*
		inactive	integer	*
		slab	integer	*
		total	integer	*
		used	integer	*
		used_percent	float	-
Swap	host		String	*
		free	integer	*
		in	integer	*
		out	integer	*
		total	integer	*
		used	integer	*
		used_percent	float	*
Kernel	host		String	*
		boot_time	integer	-
		context_switches	integer	*
		interrupts	integer	*
		processes_forked	integer	*

Table 2.2: Table (part 1) of the measurements *Telegraf* and *PfiTCollect* are collecting, their tags, field keys (with datatypes) and if *PfiTCollect* collects the metrics, *Telegraf* collects.

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

Plugin	Tag	Metric	Datatype	*PfiTCollect metric/tag*
System	host		String	*
		load1	float	*
		load15	float	*
		load5	float	*
		n_cpus	integer	*
		n_users	integer	*
		uptime	integer	*
		uptime_format	string	-
Processes	host		String	-
		blocked	integer	-
		dead	integer	-
		idle	integer	-
		paging	integer	-
		running	integer	-
		sleeping	integer	-
		stopped	integer	-
		total	integer	-
		total_threads	integer	-
		unknown	integer	-
		zombies	integer	-
Uprocstat	host		String	-
	process_name		String	-
	uid		String	-
		cpu_time	float	-
		cpu_time_guest	float	-
		cpu_time_guest_nice	float	-
		cpu_time_idle	float	-
		cpu_time_iowait	float	-
		cpu_time_irq	float	-
		cpu_time_nice	float	-
		cpu_time_soft_irq	float	-
		cpu_time_steal	float	-
		cpu_time_stolen	float	-
		cpu_time_system	float	-
		cpu_time_user	float	-
		cpu_usage	float	*
		invol_context_switches	integer	-
		memory_rss	integer	-
		memory_swap	integer	-
		memory_vms	integer	-
		num_threads	integer	-
		pid	integer	-
		vol_context_switches	integer	-

Table 2.3: Table (part 2) of the measurements *Telegraf* and *PfiTCollect* collects, their tags, field keys (with datatypes) and if *PfiTCollect* collects the metrics, *Telegraf* collects.

Plugin	Tag	Metric	Datatype	*PfiTCollect metric/tag*
BeeGFS Server	host			-
		bytes_read	float	-
		bytes_written	float	-
		data_busy	float	-
		data_queue	float	-
		data_reqs	float	-
		meta_busy	float	-
		meta_queue	float	-
		meta_reqs	float	-
Beegfs Client	host			*
		storage		-
		bytes_read	float	*
		bytes_written	float	*
		data_busy	float	-
		data_queue	float	-
		data_reqs	float	-
Disk	device			-
	fstype			-
		free	integer	-
		inodes_free	integer	-
		inodes_total	integer	-
		inodes_used	integer	-
		total	integer	-
		used	integer	-
		used_percent	float	-
DiskIO	host			-
	name			-
		io_time	integer	-
		iops_in_progress	integer	-
		read_bytes	integer	-
		read_time	integer	-
		reads	integer	-
		weighted_io_time	integer	-
		write_bytes	integer	-
		write_time	integer	-
		writes	integer	-

Table 2.4: Table (part 3) of the measurements *Telegraf* and *PfiTCollect* collects, their tags, field keys (with datatypes) and if *PfiTCollect* collects the metrics, *Telegraf* collects.

Plugin	Tag	Metric	Datatype	*PfiTCollect metric/tag*
Infiniband	host			*
		ExcessiveBufferOverrunErrors	float	-
		LinkDownedCounter	float	-
		LinkErrorRecoveryCounter	float	-
		LocalLinkIntegrityErrors	float	-
		PortMulticastRcvPkts	float	-
		PortMulticastXmitPkts	float	-
		PortRcvConstrErrors	float	-
		PortRcvData	float	*
		PortRcvErrors	float	-
		PortRcvPkts	float	*
		PortRcvRemotePhysErrors	float	-
		PortRcvSwitchRelayErrors	float	-
		PortUnicastRcvPkts	float	-
		PortUnicastXmitPkts	float	-
		PortXmitConstraintErrors	float	-
		PortXmitData	float	*
		PortXmitDiscards	float	-
		PortXmitPkts	float	*
		SymbolErrorCounter	float	-
		VL15Dropped	float	-

Table 2.5: Table (part 4) of the measurements *Telegraf* and *PfiTCollect* collects, their tags, field keys (with datatypes) and if *PfiTCollect* collects the metrics, *Telegraf* collects.

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

Now we will take a closer look at the main function (see listing 2.1), in which every step will be described shortly and if necessary important sections of the code will be explained in more detail seperately.

```
1  int main( int argc, char **argv )
2  {
3
4    char random_number_string[ 10000 ] = "";
5
6    int random_number = 0;
7    int return_value = 0;
8
9    //
10   // Get hostname with gethostname
11   hostname[ BUFFER_LENGTH_HOSTNAME ] = '\0';
12   gethostname( hostname, BUFFER_LENGTH_HOSTNAME + 1 );
13
14
15   //
16   // Create random number for file names (to identify them uniquely)
17   random_number = create_random_number( );
18   snprintf( random_number_string, 10000, "%d", random_number );
19
20
21   //
22   // Gets the data of the config file
23   get_config( argc, argv, random_number_string );
24
25
26   //
27   // Gets architecture for the node
28   get_architecture( random_number_string );
29
30
31   //
32   // Get the time in nanoseconds
33   time_in_nanoseconds = ( unsigned long long ) time( NULL ) *
34                         1000000000ULL;
35
36
37   //
38   // For every measurement step
39   flag = false;
40   while( true )
41     {
42
43       //
44       //  Get the Jobid
45       get_jobid( random_number_string );
46
47
48       //
49       // Collect metrics from sources
```

```
50        return_value = get_system_data( flag , random_number_string );
51        if( return_value != 0 )
52          fprintf( stderr , "Problem in get_system_data!\n" );
53
54        return_value = get_io_data( flag , random_number_string );
55        if( return_value != 0 )
56          fprintf( stderr , "Problem in get_io_data!\n" );
57
58        return_value = get_net_data( flag , random_number_string );
59        if( return_value != 0 )
60          fprintf( stderr , "Problem in get_net_data!\n" );
61
62        if( architecture_instance.nr_gpus > 0 )
63        {
64
65          return_value = get_gpu_data( random_number_string );
66          if( return_value != 0 )
67            fprintf( stderr , "Problem in get_net_data!\n" );
68
69        }
70
71        //
72        // Post data to database
73        time_in_nanoseconds = ( unsigned long long ) time( NULL ) *
74                              1000000000ULL;
75
76        return_value = push_data_to_db( random_number_string );
77        if( return_value != 0 )
78          fprintf( stderr , "Problem in push_data_to_db!\n" );
79
80        flag = flag || true;
81
82        time_in_microseconds = config_instance.intervall_length *
83                               1000000UL;
84        usleep( time_in_microseconds );
85
86        return_value = 0;
87
88      }
89
90    return EXIT_SUCCESS;
91
92  }
```

Listing 2.1: Listing of the *main* function

In line 12 the hostname of the node the *PfiTCollect* instance is running on will be read in. This is necessary to distinguish the metrics by node. The config file of *PfiTCollect* and the architecture file are read in line 23 and 28. In the configure file the

- paths to the collector tools *beegfs-ctl*, *nvidia-smi*, *perfquery* and the debug files are stored

- and the paths to the CA certificate file and the *netrc* file, where the credentials are stored to register to the database,
- as well as the (IP) address of the *InfluxDB* node,
- the interval length of the measurements,
- and the database name in *InfluxDB*, where the metrics should be stored.

The architecture file stores the architecture profile (#CPU, #mem, #GPUs) of the nodes of every partition. After getting the current system time in nanoseconds (line 33 f.) the core part of *PfiTCollect* will be entered. This is the infinite `while` loop beginning in line 40 and which ends at line 88. In every while loop the job ID of the job currently running on this node will be read from the batch system for this node. After that the metrics will be collected from line 50 until line 69. These metrics are listed in table 2.3 to ?? (see asterisks in column 5). The collecting process of the metrics will be continued until a severe error occurs (for example if a metric file like */proc/meminfo* cannot be opened) or the program run will be interrupted. In listing 2.2 a code snippet with the metric collection in action (collecting memory metrics in the function *get_system_data*) is presented. The source in this example is the *procfs* filesystem, while other metric sources/collector tools in other collector functions are `beegfs-ctl` (collecting *BeeGFS* metrics), `nvidia-smi` (Nvidia graphic cards metrics) and `perfquery` (Infiniband metrics). In line 16 ff. the */proc/meminfo* file will be opened and tested, if it exits. If it does not exist (for example the *procfs* filesystem was not mounted), an error will be thrown and the program will be interrupted, since these metrics are essential. If the file exists the memory metrics will be read in the while loop beginning at line 36. In an analoguous way the other metrics will be read. In every while iteration a line of */proc/meminfo* will be read and afterwards tested, if the metric string in the while loop is detected in this file. If this test was positive, the appropriate line will be split into the string and the metric value. The metric value string will then be converted into a number type (float or integer). In this function snippet all memory related metrics are collected from */proc/meminfo* (*mem_total*, *mem_free*, *mem_used*, ...). The converted metricvalues are stored in an instance of the system class (*system_instance.mem_total*, etc.). For example the *mem_total* metric is placed at the first line of */proc/meminfo* and in line 39 until 41 of the listing 2.2 this metric entry will be parsed, extracted and the value stored in `system_instance.mem_total` (see line 48 f.). The `while` loop in this function will end if no more lines can be read from the procfs file. The other metrics (CPU, swap, kernel, system) will be collected in the same manner in the same function and *BeeGFS*, graphics card and infiniband metrics will be collected and processed in *get_io_data*, *get_net_data* and *get_gpu_data*. All stored metric values will be pushed via the `curl` command in one step into the timeseries database *InfluxDB* and for debugging reasons these values of a time step will be stored into a file to be able to check if the right values were collected. Finally it is important to mention, that in */proc/meminfo* the values are given in kB, which usually means KiloBytes (kiB = 1000 bytes). This is due to legacy concerns, because nowadays kB in this Linux environment is actually KiB (= 1024 Bytes).

To summarize, the measurements will be collected in the following functions:

- get_system_data: mem, cpu, system, uprocstat (*procfs*),

- get_io_data: */proc/diskstats*, */proc/net/rpc/nfs*, *beegfs-ctl*
- get_net_data: *perfquery*,
- get_gpu_data: *nvidia-smi*,
- get_jobid: batchsystem.

```
1  int get_system_data ( bool flag_step, char *random_number_string )
2  {
3
4    char buffer[ BUFFER_LENGTH_GET_DATA ] = "";
5    char *string, *string1, *ptr;
6
7    const char *delimiter_colon = ":";
8    const char *delimiter_space = " ";
9
10   int return_value = 0;
11
12 #ifdef DEBUG
13   static int first_iteration = 1;
14 #endif
15
16   FILE *fp = NULL;
17
18 ....
19
20   //
21   // Open pseudo file /proc/meminfo
22   // Throw an error, if file does not exists
23   fp = NULL;
24   if( ( fp = fopen( "/proc/meminfo", "r" ) ) == NULL )
25     {
26
27       fprintf( stderr,
28          "PfiTCollect: Could not find /proc/meminfo! Aborting!" );
29
30       exit( EXIT_FAILURE );
31
32     }
33
34   //
35   // Read lines of above file ...
36   while( fgets( buffer, BUFFER_LENGTH_GET_DATA, fp ) != NULL )
37     {
38
39       string = strtok( buffer, delimiter_colon );
40       string = strtok( NULL, delimiter_colon );
41       string1 = strtok( string, delimiter_space );
42
43       // If string BUFFER contains the substring "MemTotal" ...
44       if( strstr( buffer, "MemTotal" ) )
45       {
```

```
46
47          // convert and store it (total RAM) in ...
48          system_instance.mem_total =
49              strtol( string1, &ptr, 10 ) * 1024;
50          continue;
51
52        }
53
54        // If string BUFFER contains the substring "MemFree" ...
55        if( strstr( buffer, "MemFree" ) )
56        {
57
58          // convert and store it (free RAM) in ...
59          system_instance.mem_free =
60              strtol( string1, &ptr, 10 ) * 1024;
61
62          continue;
63
64        }
65
66        // If string BUFFER contains the substring "Buffers" ...
67        if( strstr( buffer, "Buffers" ) )
68        {
69
70          // convert and store it (buffers) in ...
71          system_instance.mem_buffered =
72              strtol( string1, &ptr, 10 ) * 1024;
73
74          continue;
75
76        }
77
78        // If string BUFFER contains the substring "Cached"
79        if( strstr( buffer, "Cached" ) )
80        {
81
82          // convert and store it (cached memory) in ...
83          system_instance.mem_cached =
84              strtol( string1, &ptr, 10 ) * 1024;
85
86          continue;
87
88        }
89
90        // If string BUFFER contains the substring "SwapCached"
91        if( strstr( buffer, "SwapCached" ) )
92        {
93
94          // convert and store it (cached memory) in ...
95          system_instance.mem_swapcached =
96              strtol( string1, &ptr, 10 ) * 1024;
97
98          continue;
```

```
99
100       }
101
102 //
103 // ...
104
105       // If string BUFFER contains the substring "SwapTotal" ...
106       if( strstr( buffer, "SwapTotal" ) )
107       {
108
109         // convert and store it (swap_total) in ...
110         system_instance.swap_total =
111             strtol( string1, &ptr, 10 ) * 1024;
112
113         continue;
114
115       }
116
117       // If string BUFFER contains the substring "SwapFree" ...
118       if( strstr( buffer, "SwapFree" ) )
119       {
120
121         if( flag_step == false )
122         {
123
124           // convert and store it (swap_free) in ...
125           io_swapping_prev =
126             strtol( string1, &ptr, 10 ) * 1024;
127
128           system_instance.swap_free = system_instance.swap_total;
129           system_instance.swap_used = 0;
130
131         }else{
132
133           io_swapping_cur = strtol( string1, &ptr, 10 ) * 1024;
134
135           system_instance.swap_used =
136             io_swapping_cur - io_swapping_prev;
137
138           if( system_instance.swap_used < 0 )
139             system_instance.swap_used = 0;
140
141           system_instance.swap_free =
142             system_instance.swap_total - system_instance.swap_used;
143
144           if( system_instance.swap_total > 0 )
145             system_instance.swap_used_percent =
146               ( ( double ) system_instance.swap_used /
147                 ( double ) system_instance.swap_total ) * 100.0;
148           else
149             system_instance.swap_used_percent = 0.0;
150
151           io_swapping_prev = io_swapping_cur;
```

```
152
153           }
154
155           continue;
156
157         }
158
159
160         // If string BUFFER contains the substring "Slab" ...
161         if( strstr( buffer, "Slab" ) )
162         {
163
164           // convert and store it (Slab) in ...
165           system_instance.mem_slab =
166               strtol( string1, &ptr, 10 ) * 1024;
167
168           continue;
169
170         }
171
172
173         if( strstr( buffer, "KernelStack" ) )
174         {
175
176           // convert and store it (KernelStack) in ...
177           system_instance.kernelstack =
178               strtol( string1, &ptr, 10 ) * 1024;
179
180           continue;
181
182         }
183
184         if( strstr( buffer, "WritebackTmp" ) )
185         {
186
187           // convert and store it (WritebackTmp) in ...
188           system_instance.writebacktmp =
189               strtol( string1, &ptr, 10 ) * 1024;
190
191           continue;
192
193         }
194
195
196         if( strstr( buffer, "PageTables" ) )
197         {
198
199           // convert and store it (PageTables) in ...
200           system_instance.pagetables =
201               strtol( string1, &ptr, 10 ) * 1024;
202
203           if( system_instance.mem_total > 0 )
204           {
```

```
205
206             system_instance.mem_used =
207               system_instance.mem_total -
208               ( system_instance.mem_free+system_instance.mem_buffered +
209                 system_instance.mem_cached + system_instance.mem_slab +
210                 system_instance.mem_swapcached +
211                 system_instance.writebacktmp +
212                 system_instance.kernelstack +
213                 system_instance.pagetables );
214
215             system_instance.mem_used_percent =
216             ( ( double ) system_instance.mem_used /
217               ( double ) system_instance.mem_total ) * 100;
218
219           }else{
220
221             system_instance.mem_used = 0;
222             system_instance.mem_used_percent = 0.0;
223
224           }
225
226           continue;
227
228
229         }
230
231
232         // If string BUFFER contains the substring "MemAvailable" ...
233         if( strstr( buffer, "MemAvailable" ) )
234         {
235
236           // convert and store it (available memory) in ...
237           if( system_instance.mem_total > 0 )
238           {
239
240             system_instance.mem_available =
241               system_instance.mem_total - system_instance.mem_used;
242
243             system_instance.mem_available_percent =
244               ( ( double ) system_instance.mem_available /
245                 ( double ) system_instance.mem_total ) * 100;
246           } else {
247
248             exit( 1 );
249
250          }
251
252          continue;
253
254        }
255
256   }
257
```

```
258   fclose( fp );
259
260
261 #ifdef DEBUG
262   fp = NULL;
263   if( ( fp = fopen( buffer_tmp_file, "a" ) ) == NULL )
264     {
265
266       fprintf( stderr, "PfiTCollect (/proc/meminfo): Couldn't access
      metric value file! Aborting!\n" );
267
268       exit( EXIT_FAILURE );
269
270     }
271
272   fprintf( fp, "Function get_system_data (part /proc/meminfo).\n" );
273   fprintf( fp, "=================================================\n\n" );
274   fprintf( fp, "system_instance.mem_total = %ld\n",
275     system_instance.mem_total );
276   fprintf( fp, "system_instance.mem_free = %ld\n",
277     system_instance.mem_free );
278   fprintf( fp, "system_instance.mem_used = %ld\n",
279     system_instance.mem_used );
280  ...
281   fprintf( fp, "system_instance.mem_swap_total = %ld\n",
282     system_instance.swap_total );
283   fprintf( fp, "system_instance.mem_swap_free = %ld\n",
284     system_instance.swap_free );
285   fprintf( fp, "system_instance.mem_swap_used = %ld\n",
286     system_instance.swap_used );
287   fprintf( fp, "system_instance.mem_swap_used_percent = %lf\n\n",
288     system_instance.swap_used_percent );
289
290   fclose( fp );
291 #endif
292
293 ...
294
295 }
```

Listing 2.2: Listingsnippet of the *get_system_data* function

These collected values will be finally stored with the `curl` command in the timeseries database *InfluxDB* in the *push_data_to_db* function (see listing 2.3). We will present the `curl` flags used in the *PfiTCollect* case and will explain the structure of the data push which is based on the line protocol of *InfluxDB* (see for example https://docs.influxdata.com/influxdb/v1.7/write_protocols/line_protocol_tutorial/).

- -cacert *CA_file*: Allows server certification via CA certification file (see entry/value of string `full path certificate file` in config file),
- -netrc-file *netrc_file*: In this file the credentials and the machine, *InfluxDB* is running on, are stored to automatically login with `curl` to the *InfluxDB* database

(see entry/value of string `full path netrc file` in config file),

- -XPOST: Posts the metrics to the *InfluxDB* server adress given in the config file and the port 8086 (predefined port for InfluxDB) 'http://write means, that the data will be written to the specified database (in our case telegraf),
- –data-binary is the flag indicating, that the following data should be pushed into the DB,

Now the structure of the pushed data of `curl` will be explained. This strucure is in accordance with the line protocol of *InfluxDB* and the main parts will be sketched in the following enumeration. The following example snippet of the `curl` command should demonstrate that:

```
1 cpu,host=%s,jobid1=%lui,jobid2=%lui usage_user=%lf,usage_system=%
    lf, ...,cpu_usage=%lf %llu\nnext_measurement
```

- Measurement in which the following metrics should be stored (*cpu* in our example),
- comma separator,
- tags, in which the values are included dynamically into the formatter (`host=%s,jobid1=%lui,jobid2=%lui`),
- space separator,
- field keys, in which the values are included dynamically into the formatter (`usage_user=%lf,usage_system=%lf, ...,cpu_usage=%lf`)
- space separator,
- time step in seconds since the epoc,
- New line,
- next measurements, etc.

```
1 err = snprintf( buffer_cpu_mem_swap_system_procstat,
    buffer_length_push_procstat,\
2       "curl -s --cacert %s --netrc-file %s -XPOST '%s:8086/write?db=%
    s' --data-binary 'pfit-nodeinfo,host=%s cpu_model=\"%s\",main_mem
    =%lf,sockets=%lf,cores_per_socket=%lf,threads_per_core=%lf %llu\
    ncpu,host=%s,jobid1=%lui,jobid2=%lui usage_user=%lf,usage_system=%
    lf,usage_nice=%lf,usage_iowait=%lf,usage_idle=%lf,usage_irq=%lf,
    usage_softirq=%lf,usage_steal=%lf,usage_guest=%lf,usage_guest_nice
    =%lf,cpu_usage=%lf %llu\nkernel,host=%s,jobid1=%lui,jobid2=%lui
    boot_time=%ldi,context_switches=%lldi,interrupts=%lldi,
    processes_forked=%lldi %llu\n ..... processes,host=%s,jobid1=%lui,
    jobid2=%lui total=%ldi %s'",\
3       config_instance.certification_file, config_instance.netrc_file,
4       config_instance.ip4,
```

```
5         config_instance.database,
6         hostname,
7         architecture_instance.cpu_model,
8         ( double ) architecture_instance.mem,
9         ( double ) architecture_instance.nr_sockets,
10        ( double ) architecture_instance.nr_cores_per_socket,
11        ( double ) architecture_instance.nr_threads_per_core,
12        time_in_nanoseconds,
13        hostname,
14        uprocstat_instance.jobid,
15        uprocstat_instance.jobid,
16        system_instance.cpu_usage_user,
17        system_instance.cpu_usage_system,
18        system_instance.cpu_usage_nice,
19        system_instance.cpu_usage_iowait,
20        system_instance.cpu_usage_idle,
21        system_instance.cpu_usage_irq,
22        system_instance.cpu_usage_softirq,
23        system_instance.cpu_usage_steal,
24        system_instance.cpu_usage_guest,
25        system_instance.cpu_usage_guest_nice,
26        uprocstat_instance.cpu_usage,
27        time_in_nanoseconds,
28       ...
29        );
```

Listing 2.3: Listingsnippet of the curl function

The last step in one while loop is to get the timestamp and sending *PfiTCollect* to sleep for a predefined timeintervall (line 82 until 84). These steps will take place in every `while` loop.

2.2 Telegraf plugins

We created a modified procstat plugin, that evaluates the */proc/* $<$ *PID* $>$ */environ* file to identify the batch systems job id of the monitored process.

```
1  // Add batch job id's from /proc/<PID>/environ
2  // Environment variables containing the job id's in case of
3  // SLURM: SLURM_JOBID,
4  // Moab/Torque: MOAB_JOBID/PBS_JOBID,
5  // LSF: LSB_JOBID
6  pid := proc.PID()
7  pidenviron := "/proc/" + strconv.Itoa(int(pid)) + "/environ"
8
9  // ToDo: maybe better use Go functions instead of calling "strings"
10 stringsCmd := exec.Command("strings", pidenviron)
11 stringsOut, _ := stringsCmd.Output()
12
13 jobid_pat := regexp.MustCompile(".*JOBID=.*")
```

```
14 jobid_varexpr := jobid_pat.FindAllString(string(stringsOut), -1)
15
16 if len(jobid_varexpr) > 0 {
17         for i, _ := range jobid_varexpr {
18                 jobid_slice := strings.Split(jobid_varexpr[i], "=")
19                 if jobid_slice[0] == "MOAB_JOBID" {
20                         jobid1 := jobid_slice[1]
21                         proc.Tags()["jobid1"] = jobid1
22                 } else if jobid_slice[0] == "PBS_JOBID" {
23                         jobid2 := jobid_slice[1]
24                         proc.Tags()["jobid2"] = jobid2
25                 } else if jobid_slice[0] == "SLURM_JOBID" {
26                         jobid := jobid_slice[1]
27                         proc.Tags()["jobid1"] = jobid
28                         proc.Tags()["jobid2"] = jobid
29                 } else if jobid_slice[0] == "LSB_JOBID" {
30                         jobid := jobid_slice[1]
31                         proc.Tags()["jobid1"] = jobid
32                         proc.Tags()["jobid2"] = jobid
33                 }
34         }
35 }
36
37 acc.AddFields("pfit-uprocstat", fields, proc.Tags())
```

Listing 2.4: The uprocstat plugin

2.2.1 Pfit-Jobinfo

```
1
2 // handle empty nodes with no jobs
3 if len(jobfilepath) == 0 {
4         jobids = append(jobids, "nojob", "nojob")
5 } else {
6         for i := range jobfilepath {
7                 _, jobfile = filepath.Split(jobfilepath[i])
8                 s := strings.Split(jobfile, delimiter)
9                 // check if the job info file is in the correct
    format
10                if len(s) == 4 {
11                        jobids = append(jobids, s[2], s[3])
12                } else {
13                        fmt.Println("Pfit-jobid processor plugin
    error: Unable to correctly recognise job ID's from the job info
    file!")
14                        jobids = append(jobids, "jobid_not_recognised
    ", "jobid_not_recognised")
15                }
16        }
17 }
18
```

```
19 for _, metric := range in {
20         // add jobid tags only when there is no jobid1 and jobid2 tag
       yet
21         if metric.HasTag("jobid"+strconv.Itoa(1)) == false &&
22         metric.HasTag("jobid"+strconv.Itoa(2)) == false {
23                 for i, j := range jobids {
24                         metric.AddTag("jobid"+strconv.Itoa(i+1), j)
25                 }
26         }
27 }
```

Listing 2.5: The uprocstat plugin

It determines the batch job ID's from a file with the name and format
$pfit_{<}userid >_{<} jobid1 >_{<} jobid2 >$
that hast do be created in the job prolog and stored in a local directory accessible by telegraf.
The plugin then annotates all node-wide metrics with usernames and jobids, that are currently performing calculations on the node.

2.2.2 Beegfs

The beegfs plugin is an exec plugin that reads and averages measurements from the beegfs-ctl, including operations read, operations write, MiB read and MiB write. To use the script, the nodenames of the storage nodes has to be configured in the plugin as drescribed in the README.md file.

2.2.3 Infiniband & Omnipath

The infiniband and omnipath plugins are exec plugins that can be activated or adjusted on the fly - without recompiling telegraf. The infiniband and omnipath plugin both use the output generated from the *perfquery* program. The omnipath2 plugin utilized the *opainfo* program to read out the port counters.

```
1 METRICS32="SymbolErrorCounter LinkErrorRecoveryCounter
      LinkDownedCounter PortRcvErrors PortRcvRemotePhysicalErrors
      PortRcvSwitchRelayErrors PortXmitDiscards PortXmitConstraintErrors
       PortRcvConstraintErrors LocalLinkIntegrityErrors
      ExcessiveBufferOverrunErrors VL15Dropped"
2 METRICS64="PortXmitData PortRcvData PortXmitPkts PortRcvPkts
      PortUnicastXmitPkts PortUnicastRcvPkts PortMulticastXmitPkts
      PortMulticastRcvPkts"
```

Listing 2.6: The uprocstat plugin

The collected metrics are then exported to the telegraf outputs by the normal exec plugin mechanics.

2.2.4 Nodeinfo

In the nodeinfo plugin, hardware information of the node is reported.

```
1 metrics["cpu_model"]=`grep "Model name" <<< "$ICPU" | awk -F: '{gsub
    (/^[ \t]+/,"",$2); print $2}' `
2 metrics["sockets"]=`grep "Socket(s)" <<< "$ICPU" | awk '{print $2}'`
3 metrics["cores_per_socket"]=`grep "Core(s) per socket" <<< "$ICPU" |
    awk '{print $4}'`
4 metrics["threads_per_core"]=`grep "Thread(s) per core" <<< "$ICPU" |
    awk '{print $4}'`
5 metrics["main_mem"]=`grep MemTotal <<< "$IMEM" | awk '{printf("%.0f",
     $2 * 1024)}'`
```

Listing 2.7: The uprocstat plugin

2.2.5 Nvidiatool

The nvidiatool plugin is an exec plugin that collects metrics from nvidia gpus, if there are any in the system. The plugin uses the smi tool to query gpu metrics, process information and also collects information on the associated cpu process.

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

Chapter 3

Middleware

3.1 Aggregator

The aggregator module is available at https://gitlab.gwdg.de/profit-hpc/aggregator. The code is written in Python and consists of following parts:

- configuration
- database wrappers
- recommendation system
- aggregator object
- interface

Configuration

The configuration is available globally in all Python modules of aggregator and are stored in `conf` directory and named `config.py`. It can be also extended with other values and objects without any limitation. The configuration file `influxdb.py` holds only the InfluxDB credentials and separated from the main configuration in order to exclude the password from being committed to the Git repository.

database wrappers

The database wrappers are core component of the aggregator module. The main purpose of wrappers are to fetch the data from the database and set the values in Aggregator object, which can be found at https://gitlab.gwdg.de/profit-hpc/aggregator/-/blob/master/db/aggrstruct.py. By default the Aggregator module comes only with InfluxDB wrapper. However, there is a possibility to develop a wrapper for any database. In wrappers the queries for database could be also combined together to gain more performance. The wrappers are stored in the separate folders in `db` directory and main contain their own configuration files. For instance, InfluxDB wrapper contains the configuration file called `metrics.py`, where the measurement and field names could be set according to the Database scheme.

recommendation system

The recommendation system is the module which analyses the metrics and in case if the issues are detected, returns corresponding recommendations. The recommendation system follows a rule based approach and has more verbose documentation in Git repository at https://gitlab.gwdg.de/profit-hpc/aggregator/-/tree/master/rcm. The recommendation system depends only on the Aggregator object.

aggregator object

This object is necessary for recommendation system and should be set by the database wrapper. It is located at https://gitlab.gwdg.de/profit-hpc/aggregator/-/blob/master/db/aggrstruct.py and extendable if further metrics are needed for reports or for the recommendation system.

interface

The Aggregator module returns JSON files with reformatted Aggregator object and configurable. To change the format of the JSON output, the following file can be used both for text and PDF reports https://gitlab.gwdg.de/profit-hpc/aggregator/-/blob/master/format/formats_json.py

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

Chapter 4

Reportgenerators

4.1 Text report

The source code of text report module of the toolkit is located at https://gitlab.gwdg.de/profit-hpc/text-report It contains all necessary documentation within the repository, please refer to that documentation for the latest updates.

Text report has its own data validation part, where it checks if the JSON file returned by Aggregator contains any errors. Then it formats the JSON in the plain text format with fixed width so it could be printed out in the user shell or sent via Email.

If the format of the text report should be changed, it can be done by editing the `format/text.py` file. It has also very flexible configuration `conf/config.py`.

4.2 PDF report

As a text report, the PDF report is the module which gets the JSON file from the Aggregator and reformats it into PDF. Unlike text report, the PDF report has time series data and therefore the input for the PDF report differs from the input of the text report.

The source code of the PDF report is located at https://gitlab.gwdg.de/profit-hpc/pdf-report.

In order to extend the PDF report, the file `pdfgen/pfit_pdfreport.py` should be used. For any other information, please refer to the documentation located in the repository of the module itself.

4.3 Statistics generator

Until now, only the user can get an overview of its jobs and its metrics (using the text or PDF report). It would be advantageous, if administrators and Fachberater can also get an overview of the users jobstatistics over a certain time intervall. For this purpose, a statistics generator (and the appropriate documentation) was written in the course of the project extension. In this section, the code of the statistics reportgeneraor will be described.

The statistics generator was written in Python and is divided into two source files:

- `create_init_summary.py` and
- `create_report_from_summary.py`

`create_init_summary.py` initially collects selected metrics of the job report files of a given time intervall and stores them in an intermediate format in the file `summary.txt`. The results in `summary.txt` can be evaluated with the python script `create_report_from_summary.py`. With this file an administrator can display a summary of all users over a specified time intervall with selected metrics and ratios for all metrics in a table format.

The principal workflow will be illustrated in the following lising of the main function (see listing 2.4) and starts with checking the number of program arguments (line 5 until 7). The configuration file `stats.conf` will be read out in line 11 with the given path from the command line (line 10). If the configuration file is not present then Python will throw an error and exits with an error code unequal 0 (line 12 until 15). Otherwise the data path and the path, where the summary file should be placed are read in (lines 17 to 20).

In line 24 the summary file `summary.txt` will be opened. In this file the job data of each job is stored in a single line. The source of this job data are the job text reports. For each job the text report will be read and selected metrics of the job are stored in one line of the file `summary.txt` int the below described intermediate format.

```
1  def main ( argv ):
2
3    sorting_key = 0
4
5    if len( sys.argv ) < 2:
6      print( "Usage: stats configfile ! Exiting!" )
7      sys.exit( 1 )
8
9    try:
10     path_to_config_file = str( sys.argv[ 1 ].strip() )
11     fp_conf_file = open( path_to_config_file + "stats.conf", "r" )
12   except IOError:
13     print( "No config file " + str( path_to_config_file +
14           "stats.conf" ) + " exists! Exiting!" )
15     sys.exit( 1 )
16   else:
17     path_to_data = str( fp_conf_file.readline().strip() )
18     path_to_summary = str( fp_conf_file.readline().strip() )
19
20     fp_conf_file.close()
21
22
23   try:
24     fp_stat = open( str( path_to_summary + "summary.txt" ), "a" )
25   except IOError:
26     print( "No summary file " + str( path_to_summary +
27           "summary.txt" ) + "exists! Exiting!" )
28     sys.exit( 1 )
```

```
29    else:
30      file_exists = 0
31
32      list_super_dirs = []
33      list_dir = []
34      list_dir_files = []
35
36      list_super_dirs = os.listdir( path_to_data )
37      list_super_dirs = [ int( x ) for x in list_super_dirs ]
38      list_super_dirs_sorted = sorted( list_super_dirs )
39
40      try:
41        fp_tmp_file = open( str(path_to_summary + "n_last_folder.txt"),
42                            "r" )
43      except IOError:
44        print( "No last folder file " + str( path_to_summary +
45               "n_last_folder.txt" ) + " exists! Exiting!" )
46        sys.exit( 1 )
47      else:
48        size_start = int( fp_tmp_file.read( 10 ) )
49        fp_tmp_file.close()
50
51
52      size_end = list_super_dirs_sorted[ -1 ]
53      del list_super_dirs_sorted[ -1 ]
54
55
56      try:
57        fp_tmp_file = open( path_to_summary + "n_last_folder.txt",
58                            "w" )
59      except IOError:
60        print( "No last folder file " + str( path_to_summary +
61               "n_last_folder.txt" ) + " exists! Exiting!" )
62        sys.exit( 1 )
63      else:
64        fp_tmp_file.write( str( size_end ) )
65        # writes the last folder number of the list to the file
66        fp_tmp_file.close()
67
68
69      for i in list_super_dirs_sorted:
70        if os.path.isdir( path_to_data + str( i ) ) and
71                        ( i >= size_start ) and ( i < size_end ):
72          list_dir.append( i )
73
74
75      for i in list_dir:
76        list_dir_files = os.listdir( path_to_data + str( i ) )
77        len_dir_files  = len( list_dir_files )
78        for j in range( len_dir_files ):
79            if os.path.isfile( str( path_to_data + str( i ) +
80                                    "/" + list_dir_files[ j ] ) ):
81              parse_file( str( path_to_data + str( i ) + "/" +
```

```
82                                  list_dir_files[ j ] ), fp_stat, i )
83      list_dir_files = []
84
85      fp_stat.close()
86
87    print( "---------- End of program ----------" )
88
89    return 0
90
91
92  if __name__ == "__main__":
93    main( sys.argv[1:] )
```

Listing 4.1: Listing of the *main* function of `create_init_summary.py`

In the following frame (see 2.5) the intermediate format will be described.

```
1  user123 | 1148102 | job123 | std | 1 | node112 | 3600 | 609 |
2  1588702752 | 1588729672 | 1588730281 | 73.1 | 11.7 | 16.0 |
3  0.0 | 0.0 | 16.0 | 18.9 | 11.9 | 62.8 | 0.0 | 0.0 | 1.9 |
4  0.0 | 0.0 | 0.0 |
5
6  user234 | 1148169 | job234 | std | 1 | node112 | 43200 | 40604 |
7  1588734913 | 1588734914 | 1588775518 | 68.4 | 10.9 | 16.0 |
8  0.0 | 0.0 | 16.0 | 91.0 | 57.1 | 62.8 | 0.0 | 0.0 | 1.9 |
9  0.0 | 0.0 | 0.0 |
10
11 user345 | 1148110 | job345 | std | 1 | node112 | 3600 | 657 |
12 1588702752| 1588730285 | 1588730942 | 71.8 | 11.5 | 16.0 |
13 0.0 | 0.0 | 16.0 | 17.1 | 10.7 | 62.8 | 0.0 | 0.0 | 1.9 |
14 0.0 | 0.0 |0.0 |
15
16 user456 | 1149118 | job456 | std | 1 | node044 | 9000 | 858 |
17 1588782442 | 1588782443 | 1588783301 | 43.0 | 6.9 | 16.0 |
18 0.0 | 0.0 | 16.0 | 22.4 | 14.1 | 62.8 | 0.0 | 0.0 | 1.9 |
19 0.0 | 0.0 | 0.0 |
```

Listing 4.2: Listing with example entries in the intermediate format

In every row selected metrics of every job are stored, separated with a pipe. This format is derived from the Slurm format (see e.g. the format of `sacct -p`). The meaning of every entry is as follows:

Username | JobID | Jobname | Partition | Number of nodes | Nodelist | Requested walltime of the job in seconds | Elapsed walltime of the job in seconds | Submit time of the job in seconds since the epoch | Start time of the job in seconds since the epoch | Endtime time of the job in seconds since the epoch | Ratio of averaged used cpu cores and averaged total available cpu cores | Averaged used CPU cores | Averaged total available CPU cores | Ratio of averaged used CPU hyperthreads and averaged total available CPU hyperthreads | Averaged used CPU hyperthreads | Averaged total available CPU hyperthreads | Ratio of averaged High Water Mark of main memory and averaged total available memory | Averaged High Water Mark of main memory | Averaged total available main memory | Ratio of averaged High Water Mark of swap memory and averaged total

available swap memory | High Water Mark of swap memory | Averaged total available swap memory | Ratio of High Water Mark of GPU memory and averaged total available GPU memory | Averaged High Water Mark of GPU memory | Averaged total available GPU memory |

If the summary file does exists (lines 25 - 28), 3 empty lists are created (lines 32 - 34). The list `list_super_dirs` is filled with the directory numbers, which sepearate the reports into thousands. This list is filled and sorted in the in lines 36 - 38. The last folder of the sorted list `list_super_dirs_sorted` will be cut, because this is the directory where the data of the current jobs will be stored. This folder is ommitted, to later avoid double entries. This last folder is stored in the file `n_last_folder` and this is the starting point for the next reading of those folders (see lines 40 - 66). At the beginning this file contains a 0. For every new run of `create_init_summary.py` all folders/foldernumbers between this startnumber and the last/current folder number. In the lines 74 until 82 the core of this file is located. Here for every superfolder (line 74) the files are parsed (line 80 f.) and the selected metrics are stored in the file `summary.txt` in the above described format.

To create an overview of the cluster usage of each user for an administrator (in a specified time), the script `create_report_from_summary.py` generates a table, where in each row the averaged metrics over a specified time interval of a user are listed (see **??**). To realize this table we choosed the following procedure, shown in listing 2.6. At the beginning (lines 3 until 11) the program tests, if the number of parameters is lesser than 5. If that is the point, the program will exit with an error message, which explains the structure of the input. It is as follows: `stats [y|m|d|h] [number] configfile sortingcriterion`. `stats` is the programname, the 4 following letters define the unit of the time interval (years, months, days and hours) and number is the concrete number of units. `configfile` is the name of the configurationfile, where the path to the summary file is in. The last argument is the sortingcriterion, that means, to which metric the values in the table have to be sorted. If the input of the command was correct, then the arguments will be parsed and stored in a list (lines 16 - 18). In line 25 to 35 the number of seconds of one unit are calculated (which depends of the unit which was taken) times the number of the units. In line 37 the difference between the current timestamp (in seconds) minus the number of seconds of the choosen time interval are calculated. This is the starting point for the extraction. All entries with a time stamp (end of job) older than this, will be ommitted. Line 42 f. reads the sorting key. The configuration file is read in line 49 f. and the contents of this file (the path to the data and the summary file) in line 56 f. The summary file (the data source of this script) will be opened and read in line 64 and 72 into a list (`list_acc_global`), which contains sublists. Every sublist refers to a job and all entries of a sublist are metrics averaged over one job of the user. Those part of the list, where the time stamp of the end of the job runtime is newer than the difference of the current time and the number of seconds of the choosen time interval, are extracted and stored into a new list (`extracted_list`). Finally with the function in line 81 the start and end point of the output interval is presented on the screen and in line 82 the function `output_data` the extracted list is sorted by the given sorting key(given metrics) and displayed on `stdout` as a table with a row for every user and columns containing selected metrics (for example requested and elapsed wallclock

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

time, CPU usage, main memory high water mark, please see ...).

```
1  def main( argv ):
2
3    if len( sys.argv ) < 5:
4      print( "\nUsage: stats [y|m|d|h] [number] configfile
5              sortingcriterion" )
6      print( "with sortingcriterion = user | requested | elapsed |
7              ratiowalltime | cpuusage | cpucores | ratiocpu |")
8      print( "memhwm | memtotal | ratiomem | swaphwm | swaptotal |
9              ratioswap | njobs | aveelapsed\n" )
10
11     sys.exit( 1 )
12
13   #
14   # store all command line arguments in the array args
15
16   args = []
17   for i in sys.argv:
18     args.append( i.strip() )
19
20   #
21   # Get current time in seconds since epoch and the seconds that to
       program should search in the past
22
23   time_since_epoch = int( time.time() )
24
25   if( args[ 1 ] == "y" ):
26     seconds_in_the_past = int( int( args[ 2 ] ) * 365.25 * 86400 )
27
28   if( args[ 1 ] == "m" ):
29     seconds_in_the_past = int( int( args[ 2 ] ) * 30.438 * 86400 )
30
31   if( args[ 1 ] == "d" ):
32     seconds_in_the_past = int( args[ 2 ] ) * 86400
33
34   if( args[ 1 ] == "h" ):
35     seconds_in_the_past = int( args[ 2 ] ) * 3600
36
37   start_date = time_since_epoch - seconds_in_the_past
38
39   #
40   # Get sorting key, for example high water mark (memhwm)
41
42   sorting_key = 0
43   sorting_key = get_sorting_key( args[ 4 ] )
44
45
46   #
47   # Open config file
48
49   path_to_config_file = str( sys.argv[ 3 ].strip() )
50   fp_conf_file = open( path_to_config_file + "stats.conf", "r" )
```

```
51
52
53    #
54    # Read data and summary file path
55
56    path_to_data = str( fp_conf_file.readline().strip() )
57    path_to_summary = str( fp_conf_file.readline().strip() )
58
59
60    #
61    # Open summary file
62
63    try:
64      fp_stat = open( str( path_to_summary + "summary.txt" ), "a" )
65    except:
66      print( "Cannot open summary.txt" )
67
68
69    #
70    # Read file into list
71
72    list_acc_global = read_summary_file( path_to_summary +
73                                         "summary.txt" )
74    extracted_list = extract_part_of_list_acc_global(
75          list_acc_global, time_since_epoch - seconds_in_the_past )
76
77    #
78    # Output of the collection interval and the sorted
79    # summarized list
80
81    output_timeinterval( start_date, time_since_epoch )
82    output_data( extracted_list, sorting_key )
83
84    print( "\n---------- End of program ----------\n" )
85
86    return 0
87
88
89  if __name__ == "__main__":
90    main( sys.argv[1:] )
```

Listing 4.3: Listing of the *main* function of `create_report_from_summary.py`.