# Tracking HPC Applications through Hashes of Executables

University of Basel
Faculty of Science
Department of Mathematics and Computer Science
HPC Research Group

Examiner: Prof. Dr. Florina M. Ciorba
Supervisor: Thomas Jakobsche

Author: Bilal Kocak
Email: bilal.kocak@stud.unibas.ch

June 16, 2022

**Abstract**

High-Performance Computing (HPC) provides considerable advantages in executing complex programs as parallel data processing on different nodes to gain performance on the run time of the applications. This report provides how to track HPC applications using their hashes of executables. We supply two bash scripts which are executed on the HPC systems, one of them for collecting all the hashes of executables through SLURM into named with labeled job id specific folders. The second bash script then extracts the information, which is the number of repeated executions of the same executable, user name, job id, runtime, resource usage (memory usage, CPU usage, etc.), and analyzes the processed data to find the correlations between them.

**Keywords:**

- HPC applications, job monitoring, hashes of executables

# Contents

# Chapter 1

# Introduction

In recent years, extreme amounts of digital data have been created day by day, and the complexity of the application increased. In order to solve this challenge, various concepts were developed depending on the area and need. The HPC systems are entering the area to bring up the light to the problems. HPC systems consist of many supercomputers linked together and are used in a wide range of parallel computing applications in the real world. It contains an extensive range of utilities from drug design to security industries. Thus, this brings many valuable advantages, such as reducing the price of the product manufacture, overcoming the difficulties of designing and building them, and saving time for evaluation of long-standing events like global climate change.

On the HPC system, supercomputers mostly use software programs such as applications and libraries already installed in the system. Users can use preinstalled programs for compiling and executing their executable files(i.e., application). Additionally, users can develop their applications for specific purposes. HPC jobs contain applications that are executed on the HPC systems. These applications are often in the form of compiled executables. Thus, the system operators or researchers do not know which application is inside a job. Nevertheless, it could be beneficial to know which application is inside a job. Furthermore, it could also be beneficial to track repeated executions of the applications. We have applied methods to get knowledge about the system resource usage of the applications.

Trying to recognize an application based on dynamic behavior during execution time has also been tried before by Ates et al. They have provided an automatic technique, which is called Taxonomist to detect applications with the usage of machine learning to classify applications and also detect unknown applications [3]. As described before, supercomputers consist of many nodes linked together within a network to achieve mass calculations. In the Taxonomist technique, monitoring data is periodically collected from the nodes during the run-time of applications. The monitoring data was later used for the analysis and training for classifying the algorithms to identify applications. Taxonomy is designed to detect unwanted applications such as cryptocurrency mining or password cracking [3]. Furthermore, by applying the Taxonomist technique, system administrators will learn about HPC systems and manage them conveniently and reliably.

Knowing the HPC systems and their facility environments' behavior will benefit HPC system administrators and system users. These benefits could be in wide-range characteristics to make the system more reliable and sustainable. Moreover, it makes the system more manageable in case of any problems. In 2017/2018, the Energy Efficient HPC Working Group (EE HPC WG) Dashboard Team conducted an analysis that assessed the current use of information dashboards for operational facility management in major supercomputing centers around the globe, resulting in the formalization of a process now referred to as Operational Data Analytics (ODA).[4]. Analyzing data from HPC systems is also called Operational Data Analytics, introduced by Bourassa et al. [4]. As described in the paper by Bourassa et al. [4], they have proposed the sophisticated ODA approach to increase the usage of energy consumption efficiency on the HPC facilities. Based on the results of the ODA measurements, the cooling equipment system is activated in the HPC system facility in case any heat load occurs. Various cooling systems optimize the total energy consumption of the HPC facility. Additionally, the use of ODA in testing and analyzing HPC and facility performance facilitates the practice of HPC design engineering and operations [4].

Monitoring and Operational Data Analytics (MODA) [2] is to provide insight into current trends in MODA, to identify potential gaps, and to offer an outlook into the future of the involved fields high performance-computing, databases, machine learning, and possible solutions for upcoming Exascale systems. MODA is a broader term that also includes aspects of data collection. It also provides an aspect of closing the gaps between the collected data and meaningful analyses to monitor the HPC systems effectively. Practical strategies for analyzing the collected data give HPC system users or end-users to make their system more efficient and maintainable and get practical/optimal resource usage of the HPC systems.

## 1.1 Motivation

HPC jobs typically contain applications that are executed on HPC systems. The application inside a job is in the form of executable (compiled source code) and often has non-descriptive names (e.g., a.out). However, HPC system operators, administrators, and HPC researchers do not know sufficient information about which application is inside a job, the number of executions, and the number of resources consumed for the applications. There is not any mechanism to track the executed applications. Although, getting a piece of knowledge about the system resource usage would be advantageous for HPC researchers and improve the job scheduling in the HPC systems. In order to get knowledge of which application is inside a job, to track repeated executions of the exact executions. In spite of the fact that our motivation is to get the knowledge of statistics about a specific application, such as execution time, resource usage( memory usage and CPU usage ), and the different node configurations that are used for the applications, and to know which applications are executed by which users and how often they are used and which job script configurations are used for which applications.

# Chapter 2

# Related Work

## 2.1 Classifying Jobs and Predicting Applications in HPC

This related paper [6], will be a summary of the related paper to this project with a different aspect of the requirements. In recent years, with increasing HPC supercomputers' usage, electric power consumption has also increased. In order to have a safe and reliable system, the prediction of such electric consumption of the HPC applications provides an advantage in such fluctuations. These predictions will avoid voltage drops in computer facilities. In order to be able to achieve this, Keiji Yamamoto et al. [6] provides a method for classifying applications on behalf of their power consumption, runtime, and resource usage in previous executions. Getting knowledge of the power consumption of the applications will interfere with any excessive use of electricity usage in the computer facility. In such situations, the system would be intervened quickly, and the system's stability would be ensured. The power consumption of the computers is application-specific and predicts the application's power consumption in the past. HPC applications are primarily executed in job script files. In the related paper, they have proposed a methodology for predicting the power consumption of the job scripts. Categorizing job scripts depends on the characteristics of the application in the job script and its use of statistics. Thus, many kinds of applications were executed on their HPC system to achieve this.

In the related paper, two methods exist

- Classifying an executed job into its application class using the feature of the executable file [6].
- Predicting the application class from a job script at the time of job submission [6].

Depending on the application's power consumption representing its classification, and when needed when performing high-power jobs, the generator is activated to power the system. In order to apply this procedure, we need to get prior knowledge of the executed jobs in the execution history.

A job contains the executable files (i.e., applications ) executed in HPC systems. So, the job is related to the application. The applications have features such as size, name, and the hash of the executable file. In HPC systems, there are environment variable Linux commands such as `LD`

PRELOAD used to extract these features. These all extracted features are collected for further usage. In one of their classification methods, Keiji Yamamoto et al. [6] uses the hash and notation of the executable. Classifying jobs using hashes is considered that if two of the executable files for the corresponding jobs are the same, then the hashes are also considered the same. Classifying jobs using symbol sets is considered that if two symbol sets of two executable files are identical, they are considered the same application.

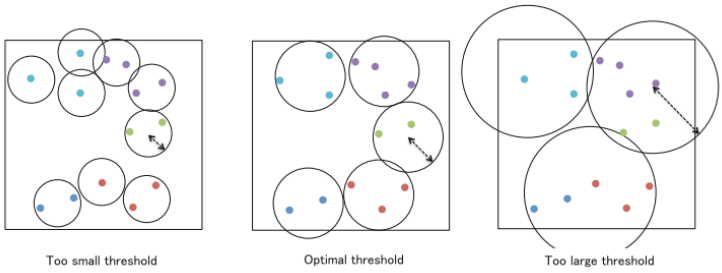


Figure 2.1: Difference of classification results at various thresholds "Reprinted from [6]".

Another method for classifying the applications is using similar symbol sets of the executable files. Defining the degree of the similarity of the symbol sets of the applications considered the same applications. All the applications are represented as a point in feature space, and if the distance between these points is small, then there is a high probability that they are the same applications. The cosine similarity method is used to calculate the distance between the vector. In order to get a good classification in this method, the threshold should be chosen very well. The high threshold is chosen to cause the problems, which are the different applications are considered the same. Also, it reduces application classification and vice versa. The low threshold was chosen to encounter many application classifications. Consequently, with the collection of statistical data, the number of executable files(i.e., application) classification differs with classifying by hash, classifying by symbol sets, and classifying by the similarity of symbol sets. Classification is done after the job is executed. It is assumed that one job script file contains only one executable file.

Another classification method with a different concept predicts an application class on time before submitting job execution. The features predicted before submission are electric power consumption, I/O traffic, and run-time. The classification is done with the features or content of the job script. After the collection of the data then, the evaluation of classification is done with the method called 10-fold cross-validation, the same as before chosen threshold is used in the method. When the threshold increases, the number of classes decreases and vice versa. In order to choose a better or optimal threshold, the number of thresholds is increased until getting better-collected data resulting in better accuracy. After applying the various classification methods, the statistical data is collected to gain knowledge and predict the power consumption of the applications on how much energy is consumed with corresponded application classes.

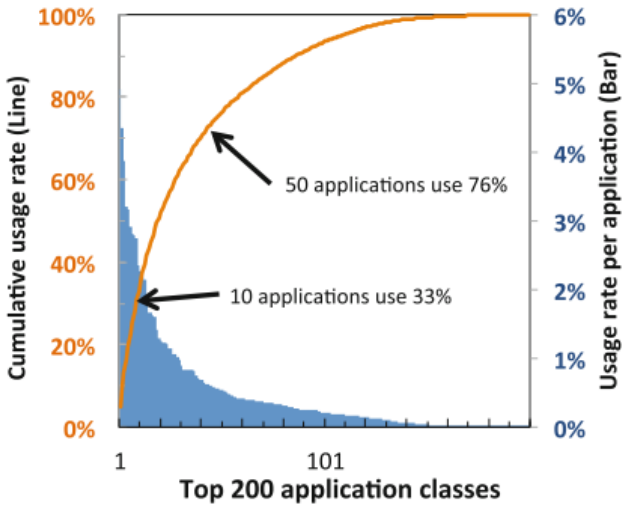


Figure 2.2: Statistics of each application "Reprinted from [6]".

This statistical information of each application and its classes is then used for the system's resource usage and to know when and how much energy is needed or would be consumed during the execution of the application and before the execution to make the system reliable, safe, and stability.

# Chapter 3

# Methods

In order to track the applications, we are using SLURM [5] commands to extract the submitted job metadata. The Slurm Workload Manager, formerly known as Simple Linux Utility for Resource Management (SLURM), or simply SLURM, is an open-source, fault-tolerant, and highly scalable cluster management and job scheduling system for Linux clusters of thousands of nodes. [5]. On the miniHPC [1] clusters, SLURM Workload Manager is already installed and used for researchers, students, and tutors. We then used this extracted metadata information that contains the job script's location. Furthermore, the job script contains the name and the path of the executable applications. We have then read that information from the job script file. Then, we computed the executable's hash and compared that with the other jobs.

## 3.1 Data Collection

### 3.1.1 Job Script Collection

We developed a program to collect job scripts. Users must execute their applications using a script we developed called collector.sh. The collector.sh script uses the command `scontrol show jobid -dd 'jobid'` to get the location of the job script. `Scontrol` is a tool meant for SLURM administration by user root.[5]. We copy the job script into a predefined folder to store all the information with a specific submitted job id. The job script is then processed further by the collector.sh script.

```
JobId=793610 JobName=srunjob
   UserId=kocbil00(75137) GroupId=unibas-users(1111) MCS_label=N/A
   Priority=4294894300 Nice=0 Account=(null) QOS=normal
   JobState=FAILED Reason=NonZeroExitCode Dependency=(null)
   Requeue=1 Restarts=0 BatchFlag=1 Reboot=0 ExitCode=2:0
   DerivedExitCode=2:0
   RunTime=00:00:01 TimeLimit=00:05:00 TimeMin=N/A
   SubmitTime=2022-05-16T10:47:28 EligibleTime=2022-05-16T10:47:28
   AccrueTime=2022-05-16T10:47:28
   StartTime=2022-05-16T10:47:28 EndTime=2022-05-16T10:47:29 Deadline=N/A
   SuspendTime=None SecsPreSuspend=0 LastSchedEval=2022-05-16T10:47:28 Scheduler=Backfill
   Partition=xeon AllocNode:Sid=dmi-cl-login:32522
   ReqNodeList=(null) ExcNodeList=(null)
   NodeList=cl-node002
   BatchHost=cl-node002
   NumNodes=1 NumCPUs=40 NumTasks=1 CPUs/Task=1 ReqB:S:C:T=0:0:*:1
   TRES=cpu=40,mem=62000M,node=1,billing=40
   Socks/Node=* NtasksPerN:B:S:C=1:0:*:* CoreSpec=*
   JOB_GRES=(null)
     Nodes=cl-node002 CPU_IDs=0-39 Mem=62000 GRES=
   MinCPUsNode=1 MinMemoryNode=62000M MinTmpDiskNode=0
   Features=(null) DelayBoot=00:00:00
   OverSubscribe=NO Contiguous=0 Licenses=(null) Network=(null)
   Command=/storage/shared/projects/MODA/collection-test/srun.jobscript
   WorkDir=/storage/shared/projects/MODA/collection-test
   StdErr=/storage/shared/projects/MODA/collection-test/srunoutput.txt
   StdIn=/dev/null
   StdOut=/storage/shared/projects/MODA/collection-test/srunoutput.txt
   Power=
```

Figure 3.1: Example of SLURM command output of the `scontrol`.

Listing 3.1: A code snippet for finding job script location on the collector.sh

```
exStorage="/storage/shared/projects/MODA/executable-collection/"
$dirOfCollector=$exStorage$1".folder"
cmdStr="Command="
value=$1
          ...
##executing the scontrol SLURM command to get information about the submitted job files
cmdPath=$(scontrol show jobid -dd $value | grep $cmdStr)
if [ $? -eq 0 ]; then
   ##setup the current directory and removing the "Command=" word from the path to get the
       pure path
   commandDir=${cmdPath/$cmdStr/}
   ##Copy job file to the CollectorDB directory
   tobeCopiedFileJS=$dirOfCollector"/"$value".jobscript"
   ##removing blanks from the string
   commandDir=${commandDir//[[:blank:]]/}
          ...
```

We are using code snippets in the collector.sh bash script to receive the submitted job ID when we submit the job for execution. It then uses a SLURM command called `scontrol` to get the information. The file location of the submitted job is saved under the keyword called `Command=`.

### 3.1.2 Application Executable Collection

After reading the job script file from the known folder, we start to collect the executable file name inside the job script file. As we describe in the previous section, we get the job script from the SLURM command called `scontrol`. We loop through the file and search for the `srun`, `mpirun`, and `mpiexec` keyword commands and then check which parameter is a file or not. After finding it is a file, we store the job script, executable, and executable on the miniHPC with its' submitted job id.

```
1 #!/bin/bash
2 #SBATCH -J srunjob
3 #SBATCH --time=00:05:00
4 #SBATCH --exclusive
5 #SBATCH --nodes=1
6 #SBATCH --ntasks-per-node=1
7 #SBATCH --cpus-per-task=1
8 #SBATCH --partition=xeon
9 #SBATCH --hint=nomultithread
10 #SBATCH --output=srunoutput.txt
11
12 ml load intel
13
14 srun sleep.sh
```

Figure 3.2: Example of SLURM job script.

Listing 3.2: A code snippet for reading job script file content on the collector.sh

```
        ...
while read -r lineByline; do
if [[ "$lineByline" == *"srun"* || "$lineByline" == *"mpirun"* || "$lineByline" ==
    *"mpiexec"* ]] ;then

   sentence=$lineByline
   stringArray=($sentence)
   ##looping the string array list which keeps the program name liste
   for val in "${stringArray[@]}"
   do
      if [ -f $val ];then
         echo "ITS FILE" $val
         nameOfProgram=${val}
         break
      else
         echo "ITS NOT A FILE" $val
      fi
   done
        ...
done < $dirOfCollector"/"$value".jobscript"
        ...
```

We read the job script file line by line until we find the program name. After the application name is founded, it copies the job script file and stores it in the specified location.

## 3.2 Data Processing

## 3.3 Hash Computation of Executables

After getting the executable file, we compute the executable applications' hash. A hash function is any function that can be used to map data of arbitrary size to fixed-size values.[Wikipedia]. A hash value is a unique value that corresponds to the file's content. Rather than identifying the contents of a file by its file name, extension, or other designation, a hash assigns a unique value to the contents of a file.[Microsoft]. In our project, we have used and applied the command called `sha1sum`, which is already installed on the miniHPC system. The `Sha1sum` command is used to calculate the SHA-1 message digest of the file. There are many other hashing algorithms used in encryption systems. Some standard hashing algorithms are SHA-1, SHA-2, SHA-256, SHA-512, MD5, NTLM, LANMAN, etc. One of the common problems of the hashing algorithms when applying to hash is a hash collision. Hash collision is when we apply a hashing algorithm to the different values and files, and the algorithm produces the same hash value for different pieces of data. This problem is not an ideal state in cryptography. Google broke an SHA-1 hashing algorithm that performs collision appending to different PDF files that produced the same value. Thus, hash collisions are inevitable, but there is a solution for collision resolutions. The SHA-256 hashing algorithm is one of the mainly used ones, which is still not yet cracked. So, we can say, for now, it is a collision-free algorithm.

Listing 3.3: A code piece applying the hash to executable file on the collector.sh

```
#code to be run if file does not exist
cp $progPath $tobeCopiedFileEX
echo "Executable file with name of " $value".executable copied!"
chmod +x $tobeCopiedFileEX
## also compute the hash
touch $dirOfCollector"/"$value".executable-hash"
sha1sum $progPath > $dirOfCollector"/"$value".executable-hash"
##echo "Executable hash file with name of " $value".executable-hash copied!"
chmod +x $dirOfCollector"/"$value".executable-hash"
```

We store the original executables, apply the `sha1sum` encryption algorithm to the executables, and save them to the specified location. Hash collisions are inevitable, but there are solutions to resolve them.

## 3.4 Data Analysis

### 3.4.1 Data Analysis Aggregated

To build the statistics, we developed custom bash scripts. We have collected the number of repeated executions of the same applications concerning elapsed time, memory usage, and CPU usage. We also collected data on which users and how often they executed the app and the number of app names that executed the most. Elapsed time statistics about repetitions are also saved in the system.

| Hash ID | User Name | App Name | Min Time | Avg Time | Max Time |
|---|---|---|---|---|---|
| feff87f2591466d41fbe2f3279689388448d4ddb | [kocbil00(2)jakth000(6)] | [sleep.sh(8)] | 00:00:00 | 00:00:03 | 00:00:08 |
| b012780364a1489f09a8d3ea1327ec129b4c8fb8 | [jakth000(7)] | [mandel.0(7)] | 00:15:02 | 00:15:12 | 00:15:27 |
| a8c8e943fff7cac8477e1080fd516873d4eb9dde | [kocbil00(6)] | [sleep.sh(6)] | 00:00:06 | 00:00:06 | 00:00:07 |
| c641e4c309db60bf705ebf9382b1d60986252889 | [kocbil00(2)] | [sleep.sh(2)] | 00:01:42 | 00:01:42 | 00:01:43 |
| 8e973ac3b72bea8c05ea03ada4e8fa87540a488a | [kocbil00(2)] | [sleep.sh(2)] | 00:05:12 | 00:05:20 | 00:05:29 |
| 7d3cd6e7b88dbfec560903e13432f6bf1bfc37e9 | [kocbil00(2)] | [sleep.sh(2)] | 00:00:05 | 00:00:05 | 00:00:05 |
| c42cd4238faf961b0617bd4b7db3656cbe03a065 | [kocbil00(1)] | [sleep.sh(1)] | 00:00:04 | 00:00:04 | 00:00:04 |
| 9957be0d479db6e6eaca11ff78cead0e19752335 | [kocbil00(1)] | [sleep.sh(1)] | 00:00:08 | 00:00:08 | 00:00:08 |
| 94d5c62433f9f2088ed18ec6179d63a557ef0a54 | [jakth000(1)] | [mg.E.128(1)] | 00:27:55 | 00:27:55 | 00:27:55 |
| 670d0f1a8aed3339a5798292df5a1ac587ee3734 | [jakth000(1)] | [bt.A.64(1)] | 00:15:49 | 00:15:49 | 00:15:49 |
| 43014d9889a376f075712ddc2f73cb78e7e32d4f | [jakth000(1)] | [cg.C.32(1)] | 00:13:36 | 00:13:36 | 00:13:36 |

Figure 3.3: Aggregated data of the time of the executions

Listing 3.4: A piece of code in analyzer-time.sh, for looping all stored folder files

```
#loop around the storage folders
cd $exStorage
for folder in * ;do
   if [ -d "$folder" ] ;then
      jobID=${folder%.*}
      cd $folder
      for f in * ;do
         if [ -f "$f" ] ;then
            ##removing blanks from the string
            f=${f//[[:blank:]]/}
            if [[ "$f" == *"$exHashFile"* ]];then
               pathOfExecutable=$exStorage$folder"/"$f
               if [ -f "$pathOfExecutable" ] ;then
                   #reading elapsed time of the submitted job regarding its' job id
                   elapsedTime=$(sacct -j $jobID --format=jobid,elapsed,user | sed '3q;d' |
                       awk '{ print $2 }')
                  ...
```

We have stored all files in the specified folder name associated with their job ID. Then, we loop through all the folders and extract their job IDs to get the elapsed time for executing the job file.

Listing 3.5: Comparison to find max and minimum value in analyzer-time.sh

```
##Comparison to find max and minimum value
check=true
for valSort in "${arrSortedHash[@]}"
do
   vlsHashID=$(echo $valSort | awk '{ print $2 }')
   check=true
   sumup=0
   for valNorm in "${arrHash[@]}"
   do
      vlnHashID=$(echo $valNorm | awk '{ print $2 }')
      time=$(echo $valNorm | awk '{ print $3 }')
      sec1=$(echo ${time} | awk -F: '{ print ($1 * 3600) + ($2 * 60) + $3 }')

      if [[ $vlsHashID == $vlnHashID ]];then
         if [ $check == true ];then
            max=$(echo $valNorm | awk '{ print $3 }')
            min=$(echo $valNorm | awk '{ print $3 }')
            check=false
         else
            if [[ "$time" > "$max" ]]; then
               max=$(echo $valNorm | awk '{ print $3 }')
            fi
            if [[ "$time" < "$min" ]]; then
               min=$(echo $valNorm | awk '{ print $3 }')
            fi
         fi
         sumup=$(($sumup + $sec1))
      fi
   done

   counter=$(echo $valSort | awk '{ print $1 }')
   diVal=$((sumup / counter))

   #convert sec to h:m:s
   ((hour=$diVal/3600))
   ((mins=($diVal-hour*3600)/60))
   ((sec=$diVal-((hour*3600) + (mins*60))))
   aptm=$(echo $(printf "%02d:%02d:%02d\n" "$hour" "$mins" "$sec"))

   userInfo="["
   while read -r lineByline; do
      arrIN=(${lineByline//|/ })
      if [[ $vlsHashID == ${arrIN[0]} ]];then
         userInfo+=${arrIN[1]}"("${arrIN[2]}")-"
      fi

   done < $groupUserInfoPathFile
   userInfo=${userInfo::-1}
   userInfo+="]"
```

```
   appInfo="["
   while read -r lineBylineApp; do
      arrAppIN=(${lineBylineApp//|/ })
      if [[ $vlsHashID == ${arrAppIN[0]} ]];then
         appInfo+=${arrAppIN[1]}"("${arrAppIN[2]}")-"
      fi
   done < $groupAppInfoPathFile
   appInfo=${appInfo::-1}
   appInfo+="]"

   vlH=$(echo $valSort | awk '{ print $2 }')
   arrGroupedJobINFO[${#arrGroupedJobINFO[@]}]=$vlH" # "$userInfo" # "$appInfo" # "$min" #
       "$aptm" # "$max
    ...
done
```

We have compared the minimum and maximum execution times of the job scripts and stored them in an array. First, we have assigned the first value of the array as a minimum and maximum time value. In the loop code snippet with the checking of the hash value's equality in the nested loop, then check the next time values of the array for checking the minimum and maximum. If the next value is smaller than the actual one, then we assign the next value as the minimum value. If the next value is bigger than the actual value, then we set to the maximum value the actual value. This process is applied to all the minimum and maximum value computations on time, CPU, and memory. We have stored all the usernames and application names who submitted the job script with related hash id in a separate text file. After that, we aggregate and group the data on the username and app name on different text files and select based on the unique hash id.

### 3.4.2 Data Analysis Individual

For individual data, we also developed custom bash scripts to build statistics. If the executable is known, this section lists each job along with statistics. This fact may show if a job is significantly different from previous executions.

| Job ID | User Name | This Time | Min Time | Avg Time | Max Time |
|---|---|---|---|---|---|
| 783570 | jaktho00 | 00:00:00 | 00:00:00 | 00:00:03 | 00:00:08 |
| 783571 | jaktho00 | 00:00:00 | 00:00:00 | 00:00:03 | 00:00:08 |
| 783574 | jaktho00 | 00:00:00 | 00:00:00 | 00:00:03 | 00:00:08 |
| 783607 | kocbil00 | 00:00:08 | 00:00:00 | 00:00:03 | 00:00:08 |
| 783609 | kocbil00 | 00:00:07 | 00:00:00 | 00:00:03 | 00:00:08 |
| 783621 | jaktho00 | 00:00:00 | 00:00:00 | 00:00:03 | 00:00:08 |
| 783790 | kocbil00 | 00:00:04 | 00:00:04 | 00:00:04 | 00:00:04 |
| 783923 | kocbil00 | 00:00:05 | 00:00:05 | 00:00:05 | 00:00:05 |
| 783924 | kocbil00 | 00:00:08 | 00:00:08 | 00:00:08 | 00:00:08 |
| 783925 | kocbil00 | 00:00:05 | 00:00:05 | 00:00:05 | 00:00:05 |
| 783926 | kocbil00 | 00:00:07 | 00:00:06 | 00:00:06 | 00:00:07 |
| 783940 | kocbil00 | 00:01:42 | 00:01:42 | 00:01:42 | 00:01:43 |
| 783950 | kocbil00 | 00:01:43 | 00:01:42 | 00:01:42 | 00:01:43 |
| 783952 | kocbil00 | 00:05:12 | 00:05:12 | 00:05:20 | 00:05:29 |
| 783972 | kocbil00 | 00:05:29 | 00:05:12 | 00:05:20 | 00:05:29 |
| 784788 | kocbil00 | 00:00:06 | 00:00:06 | 00:00:06 | 00:00:07 |
| 784789 | kocbil00 | 00:00:06 | 00:00:06 | 00:00:06 | 00:00:07 |
| 784841 | kocbil00 | 00:00:06 | 00:00:06 | 00:00:06 | 00:00:07 |
| 787736 | kocbil00 | 00:00:06 | 00:00:06 | 00:00:06 | 00:00:07 |
| 787737 | kocbil00 | 00:00:06 | 00:00:06 | 00:00:06 | 00:00:07 |
| 788212 | jaktho00 | 00:00:06 | 00:00:00 | 00:00:03 | 00:00:08 |
| 788217 | jaktho00 | 00:00:06 | 00:00:00 | 00:00:03 | 00:00:08 |
| ..... | | | | | |

Figure 3.4: Individual data of the time of the executions

Listing 3.6: A code piece for individual data of the time of execution in analyzer-time.sh

```
##looping the array list which keeps the hash id list
for valueJb in "${arrHash[@]}"
do
   jobID=$(echo $valueJb | awk '{ print $1 }')
   hashID=$(echo $valueJb | awk '{ print $2 }')
   user=$(echo $valueJb | awk '{ print $4 }')
   thisTime=$(echo $valueJb | awk '{ print $5 }')
   for val in "${arrGroupedJobINFO[@]}"
   do
      hashGroupedID=$(echo $val | awk '{ print $1 }')
      min=$(echo $val | awk '{ print $7 }')
      avg=$(echo $val | awk '{ print $9 }')
      max=$(echo $val | awk '{ print $11 }')

      if [[ $hashID == $hashGroupedID ]];then
         echo $jobID" "$user" "$thisTime" "$min" "$avg" "$max
         echo $jobID" "$user" "$thisTime" "$min" "$avg" "$max >> $fullPathFileStat
      fi
   done
done
```

# Chapter 4

# Results

## 4.1 Statistics and Visualization

We have visualized the aggregated data we obtained using the Python pandas and matplotlib library. The executable hashes' aggregated time, CPU, and memory usage are plotted with bar charts. For small values that look better on the chart, we apply a logarithm to these values, making the data look better than before.

For the plots, the x-axis is the execution time or resource usage (CPU or memory) of the application, and the y-axis is the executed application name.

### 4.1.1 Application Names

We have already collected all the statistical data we need and saved it to specified text files separately. After that, we read all these saved text files line by line and split the lines with the '#' symbol to get column names. In a nested loop, we again split the application names with the '-' symbol that we have already separated them with this symbol. After finding the application's name, we then select the most repeated application file name for plotting. The code snippet is shown in the Listing 4.1

Listing 4.1: A piece of code to select most repeated application file name in createBarChartTIME.py

```
...
# Read the data
with open('dtTIME.txt', 'r') as f:
    for line in f.readlines():

        # Store each line in the dictionary
        hashID, userName, progName, minTIME, avgTIME, maxTIME = line.split(' # ')
        # print(line)
        progList = progName.split('-')
        for pr in progList:
                subStr = (pr.split(startChar))[1].split(endChar)[0]
                count = int(subStr)
                dataDict[pr] = count
```

```
        maxValue = max(dataDict, key=dataDict.get)
        indexVal.append(maxValue)
        dataDict = {}
    ...
```

### 4.1.2 Time Plots

The graph shows the minimum, average, and maximum time to execute, using the most frequently repeated and aggregated application names. Additionally, we apply a logarithmic scale to visualize the data better.

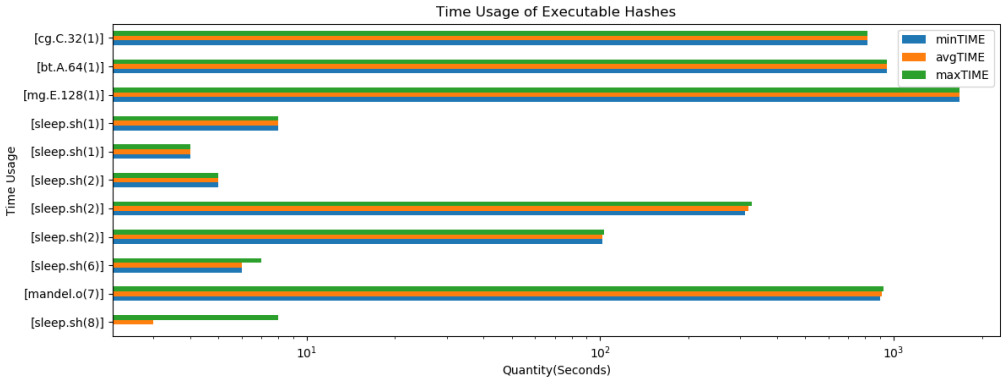


Figure 4.1: Time usage of Executable Hashes

### 4.1.3 CPU Plots

Using the most frequently repeated and aggregated application names, the graph shows the shortest, average, and longest CPU usage to execute the application. Even more than that, we apply a logarithmic scale to visualize the data better.

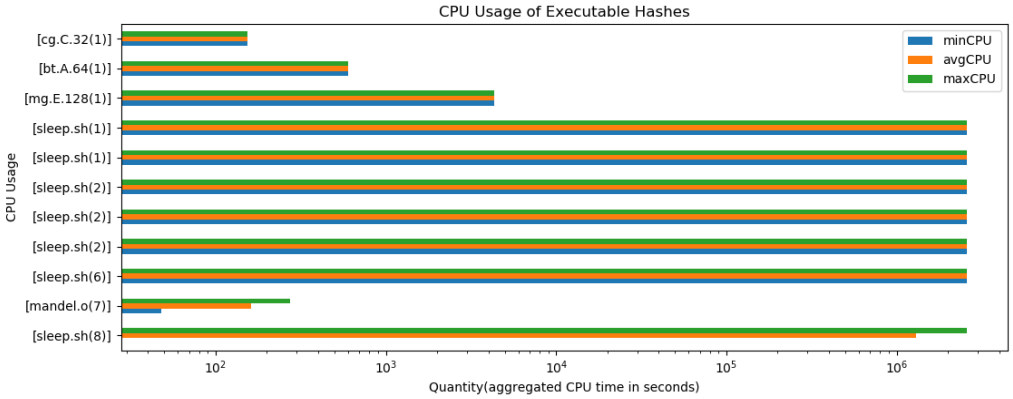


Figure 4.2: CPU Usage of Executable Hashes

### 4.1.4 MEM Plots

The graph shows the shortest, average, and longest Memory usage of executing applications using the most frequently repeated and aggregated application names. More importantly, we apply a logarithmic scale to visualize the data better.

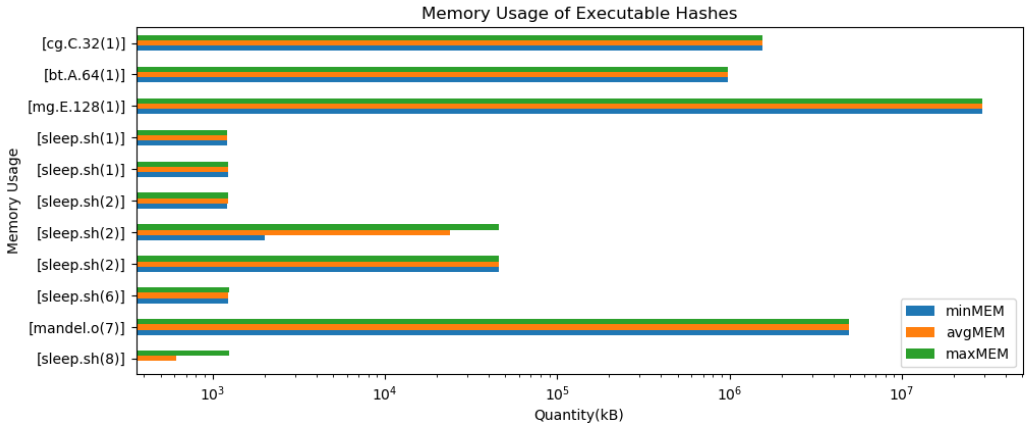


Figure 4.3: Memory usage of Executable Hashes

# Chapter 5

# Discussion

## 5.1 Benefits of Statistics about Executables

There are various advantages of tracking executables. Popular applications on the HPC systems get executed multiple times by various users. Tracking the execution of these applications can give us more insight into system usage, and we can build statistics about application behavior. Users can get information about repeated executions and see how their current execution compares to historical behavior. Administrators can understand how their system is used and that applications are frequently executed on the system. Furthermore, they can use these analyzed the monitored data to make the system more convenient and safe. For applications that had consistent behavior in the past, we could make predictions about the job behavior before it is finished.

## 5.2 Limitations of our approach

There are also some limitations to our scripts. Our approach is only working if any users execute the same application or if the same user executes the same application multiple times. If every user executes a different application, no users execute the same applications, then our method will not produce meaningful statistics. There are still some problems with collecting the executable; for example, we cannot collect executables that are just an alias (e.g., from a loaded module). Our approach is most suitable for applications with persistent behavior because if they behave differently and unpredictably every execution, then the statistics about repeated executions are not very meaningful(except maybe for estimates about maximum MEM usage).

# Chapter 6

# Conclusion and Future Work

## 6.1 Conclusion

We developed a collector.sh script that collects job scripts and hashes executables from user-submitted jobs in this project. We have also developed an analyzer.sh script that computes statistics about the historical behavior of several executed application executables, representing time, memory, and CPU usage. Our statistics provide insight into system usage and the historical behavior of application executables. So, with the data processed and collected, we have seen how many times the same application is executed on the user side. Moreover, we have seen the resource usages of the applications and users from the statistical data. As a general consequence, the statistical data of the systems give huge benefits to the HPC systems.

## 6.2 Future Work

The collector.sh script can also be improved to collect application executables that are only an alias. The collector.sh script can be developed as a `sbatch` or `srun` wrapper script and provides better functionality and immediate feedback on job performance at the end of the job. The analyzer.sh script can be improved to compute more metrics and analyze trends and seasonality in the data. Analyzer scripts can also be improved to take job script parameters into accounts, such as grouping executables by node configuration or user group.

# Bibliography


[1] minihpc. https://hpc.dmi.unibas.ch/en/research/minihpc/.

[2] Monitoring and operational data analytics conference. https://moda20.sciencesconf.org/resource/page/id/3.

[3] Emre Ates, Ozan Tuncer, Ata Turk, Vitus J Leung, Jim Brandt, Manuel Egele, and Ayse K Coskun. Taxonomist: Application detection through rich monitoring data. In *European Conference on Parallel Processing*, pages 92–105. Springer, 2018.

[4] Norman Bourassa, Walker Johnson, Jeff Broughton, Deirdre Carter, Sadie Joy, Raphael Vitti, and Peter Seto. Operational data analytics: Optimizing the national energy research scientific computing center cooling systems. pages 1–7, 08 2019.

[5] Morris A. Jette, Andy B. Yoo, and Mark Grondona. Slurm: Simple linux utility for resource management. In *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, pages 44–60. Springer-Verlag, 2002.

[6] Keiji Yamamoto, Yuichi Tsujita, and Atsuya Uno. Classifying jobs and predicting applications in hpc systems. In *International Conference on High Performance Computing*, pages 81–99. Springer, 2018.