Classifying jobs and predicting applications in HPC systems

Master Project

The 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: Hari Narayanan
Email: hari.narayanan@stud.unibas.ch

December 31, 2022

Abstract

High-performance computing has high resource usage with high-performance requirements. Understanding and optimizing resource usage while maintaining high performance is one of the biggest challenges in this area. In this project, we determine a way to classify jobs based on the application and predict which application will be executed based on the job. We also identify opportunities for using application knowledge to improve HPC resource consumption and save energy. Here, we generated hashes of nm commands, strings, and ldd and compared them. We first used the traditional hashing technique, which was more complex and failed to consider the changes that effects changes in a hash that could misclassify the applications. We use fuzzy hashing as context-based piece-wise hashing to compare hashes of different applications. Here, we demonstrate the classification of jobs based on a similarity percentage between the hashes. We used the executables from NAS parallel benchmarks and classified them based on the application. As mentioned, there are eight benchmarks, including five kernels and three pseudo applications. We could classify the applications in clusters using an appropriate threshold using fuzzy hashing.

Contents

1 Introduction **2**
1.1 Motivation 2
1.2 Challenges 2
1.3 Goals 2

2 Related Work **3**

3 Methods **4**
3.1 nm-command 4
3.2 ldd command 5
3.3 strings Command 5
3.4 Initial approach 6
3.5 Fuzzy Hashing 7

4 Results **13**

5 Discussion **16**

Chapter 1

Introduction

HPC systems submit various jobs in large quantities, leading to an increase in performance issues, which in turn consumes much power, increasing resource utilization. The idea is to classify the jobs application-wise before execution so we can predict the power usage and performance. As mentioned in the research paper by Yamamoto, they could predict jobs with 92 percent accuracy since resource usage and power consumption depend on the applications being executed. In the research paper, they proposed predicting the applications' power consumption. The proposed method was to extract the features of the executable and classify them based on the similarity of the hashes.

1.1 Motivation

In our current HPC systems, while we are submitting the jobs, the only details are the job name and the executable's name. The job names are chosen arbitrarily and need to be appropriately labeled. With the lack of the current practice of the jobs and executables needing to be labeled appropriately, we have the problem of identifying the application where the executable belongs. The idea is to label/classify the jobs before the execution, so we know the application and the resources required to run the same.

1.2 Challenges

The answer to the question is not trivial because of the security aspects of cryptocurrency mining in HPC systems. We also face resource bottlenecks if the same application being executed multiple times has a different performance which shouldn't be the case. In many applications, we could reduce CPU frequencies without losing performance. It can be best achieved if we know which application runs inside a job. If we know the application in the job, we could also co-schedule jobs on the same nodes.

1.3 Goals

This project aims to find a way to classify and cluster the executables based on the application or a programming paradigm, which can help us predict the applications that would be running on

the nodes. To achieve the mentioned goals, we need to extract the features of the executables and compare their similarity with each other. Further, we cluster the binaries based on their similarities.

Chapter 2

Related Work

There are a lot of methods and approaches to predicting and classifying jobs in HPC systems. We have used the method expressed in a research paper by Yamamoto, where they performed classification using the hash extracted from the executables. In this paper, the main idea is to extract the features of the file and place them in an application class. This helps determine the application class before the job is executed, which can, in turn, predict the application's resource usage. They have also seen its usage in the prediction of power usage of the application based on history.

In the paper by Yamamoto[1], they extract features and check the similarity of the symbol sets where they can be compared for the same application. The symbol sets generated from these features are placed in the feature vector. They measure the feature vectors' degree of similarity, called cosine similarity. Here, they identify the optimal threshold for classification.

We also came across a paper by Denis Shaikhislamov[2] where they use machine learning methods to detect similar supercomputer applications. Here they used autoencoders and Doc2Vec to detect similarities between the applications. They have also used static and dynamic analysis, which was used for sub-classification. The future work from this paper indicates using the techniques for further classification.

Our first idea was to go with the method described by Yamomoto and use the sci-kit learn library from python and generate similarities based on the distance between the vectors. We then switched to fuzzy hashing, which was relatively fast and effective, and hashes were easily categorized because of context-based piecewise hashing.

Chapter 3

Methods

We are using hashes of nm, ldd and strings command to compare and cluster the files.

3.1 nm-command

The nm command displays information about symbols in a specific file(executable or binary). The nm command reports numerical values in decimal notation by default. The nm command writes the following symbols in the output:

- Symbol Name
- Symbol Type
- Library or Object Name

Table 3.1: Symbol type description in nm

Symbol	Description
A	Global absolute symbol.
a	Local absolute symbol.
B	Global bss symbol.
b	Local bss symbol
D	Global data symbol
d	Local data symbol
f	Source file name symbol.
L	Global thread-local symbol (TLS).
l	Static thread-local symbol (TLS)
T	Global text symbol
t	Local text symbol
U	Undefined symbol

[3]

Output for nm Command

```
[naraya0001@dmi-cl-login nas-mpi-nmout]$ nm bt.A.x
                 U abort@GLIBC_2.2.5
000000000053b158 b abort_on_exit
0000000000517470 r ABS_MASK
                 U access@GLIBC_2.2.5
0000000000441700 T accumulate_norms_
000000000044fe40 t acquire_lub_resource
000000000043e8a0 T add_
00000000004e54a0 T __addq
00000000004e54f0 t addq_abs
00000000004e5540 t addq_abs.A
00000000004e4240 t addq_abs.L
...
```

3.2 ldd command

The ldd command lists the names of for all shared object dependencies like shared libraries.

Output for ldd Command

```
[naraya0001@dmi-cl-login nas-omp-lddout]$ ldd bt.A.x
        linux-vdso.so.1 => (0x00007ffda4975000)
        libmpifort.so.12 => /opt/apps/easybuild/software/impi/
        2021.4.0-intel-compilers-2021.4.0/mpi/2021.4.0/lib/
        libmpifort.so.12(0x00002b7c0e878000)
        libmpi.so.12 => /opt/apps/easybuild/software/impi/
        2021.4.0-intel-compilers-
        ...
```

3.3 strings Command

The strings command looks through all printable characters in the file.

Output for strings Command

```
[naraya0001@dmi-cl-login nas-omp-stringsout]$ strings bt.A.x
/lib64/ld-linux-x86-64.so.2
R88s
__gmon_start__
fcntl
__errno_location
lseek64
system
raise
sigaction
pthread_self
pthread_mutex_init
write
nanosleep
pthread_key_create
pthread_getspecific
pthread_setspecific
...
```

3.4 Initial approach

Our initial approach was to generate hashed from nm, string, and ldd. We compare these hashes, put them on feature vectors, and identify the distance between them. The hashes generated looked like the ones as presented in the Table 3.1. The distance would generate the degree of similarity, and further, we perform classification. We have a significant drawback using regular hashing, where the complete hash changes with a slight change in the executable. This method might need fixing with classification and would misclassify the executables. Hence, we used fuzzy hashing to move further with the research.

Table 3.2: Hash table for BT application from NAS benchmarks

BT			
Applicatio Modules	nm	strings	ldd
A	c3e29c4f08d5a5...	91322cec66c8a57...	8585df828902fc9...
B	c3e29c4f08d5a5...	af97c498e9d2005...	0ed2d9761f99b4...
C	c3e29c4f08d5a5...	642ec782f2d859c...	0290f4274b2937...
D	c3e29c4f08d5a5...	53dde80c22a9b0...	c06b2ac2fc35ee...
E	c3e29c4f08d5a5...	e8b5f577676ce42...	581042777df2b5...
F	c3e29c4f08d5a5...	03a023566119b53...	e9bbccd38a7a91...

3.5 Fuzzy Hashing

We are using Fuzzy hashing as the primary method for identifying the similarity between the hashes. This method is used in malware analysis to gather hashes and compare them. We use SSDeep for fuzzy hashing, which computes a signature based on context-triggered piece-wise hashes for each input file, also known as a fuzzy hash.

When we generate one hash of a complete file, we can see that any change in the file completely changes the hash, as seen in Figures 3.1 and 3.2.

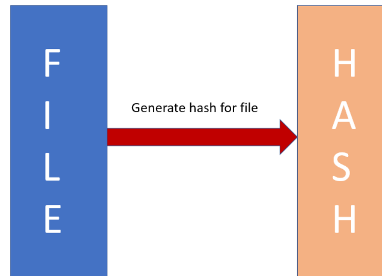

Figure 3.1: Hashing of a file

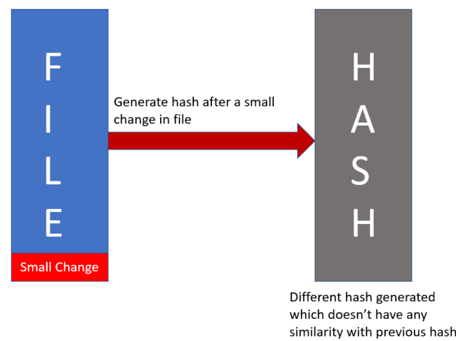

Figure 3.2: Complete change in the hash with a small change in file

We are using fuzzy hashes because it uses context-triggered piece-wise hashing, where a slight change in the executable only changes a part of the hash instead of generating a whole new hash, as you can see in Figures 3.3 and 3.4. This method helps us not generate a new hash that might be completely different and could also affect similarity results.

We have a dataset of NAS benchmarks[4] from different applications. This dataset has 8 applications

- IS - Integer Sort, random memory access
- EP - Embarrassingly Parallel

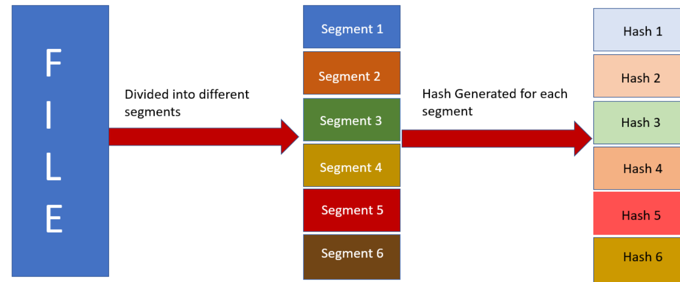

Figure 3.3: Context-triggered piece-wise hashing where hashes are segmented

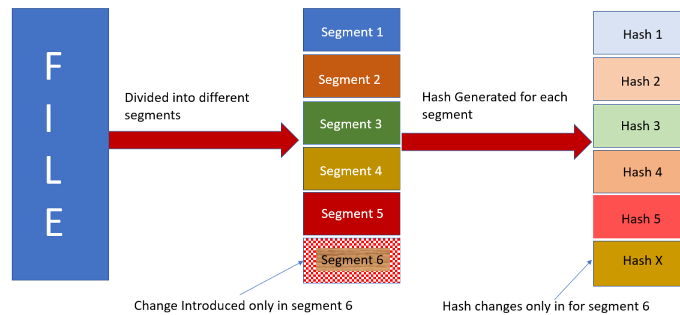

Figure 3.4: Context-triggered piece-wise hashing based on change of a part of file

- CG - Conjugate Gradient, irregular memory access and communication
- MG - Multi-Grid on a sequence of meshes, long- and short-distance communication, memory-intensive
- FT - discrete 3D fast Fourier Transform, all-to-all communication
- BT - Block Tri-diagonal solver
- SP - Scalar Penta-diagonal solver
- LU - Lower-Upper Gauss-Seidel solver

We applied fuzzy hashing on the above-mentioned executables using SSDeep. We used the hashes generated by the nm command, ldd command, and strings command. The outputs of these commands are placed in folders **nas-omp-stringsout**, **nas-omp-lddout** and **nas-omp-nmout**.

We used first used the commands to compare the files within the folder to check the similarity and have an idea about the same.

Command for comparison of files in the folder for stringsout

```
ml libtool
ssdeep-master/ssdeep -l -r -d nas-omp-stringsout/
```

Results for comparison in the strings folder

```
[naraya0001@dmi-cl-login hari-msc-project]$ bash compare-folder.sh
nas-omp-stringsout//bt.B.x matches nas-omp-stringsout//bt.A.x (90)
nas-omp-stringsout//bt.C.x matches nas-omp-stringsout//bt.A.x (94)
nas-omp-stringsout//bt.C.x matches nas-omp-stringsout//bt.B.x (91)
nas-omp-stringsout//bt.D.x matches nas-omp-stringsout//bt.A.x (91)
nas-omp-stringsout//bt.D.x matches nas-omp-stringsout//bt.B.x (90)
nas-omp-stringsout//bt.D.x matches nas-omp-stringsout//bt.C.x (90)
nas-omp-stringsout//bt.E.x matches nas-omp-stringsout//bt.A.x (91)
nas-omp-stringsout//bt.E.x matches nas-omp-stringsout//bt.B.x (88)
nas-omp-stringsout//bt.E.x matches nas-omp-stringsout//bt.C.x (88)
nas-omp-stringsout//bt.E.x matches nas-omp-stringsout//bt.D.x (90)
...
```

Command for comparison of files in the folder for lddout

```
ml libtool
ssdeep-master/ssdeep -l -r -d nas-omp-lddout/
```

Command for comparison of files in the folder for lddout

```
[naraya0001@dmi-cl-login hari-msc-project]$ bash compare-folder.sh
nas-omp-lddout//bt.C.x matches nas-omp-lddout//bt.A.x (74)
nas-omp-lddout//bt.E.x matches nas-omp-lddout//bt.A.x (61)
nas-omp-lddout//bt.F.x matches nas-omp-lddout//bt.B.x (63)
nas-omp-lddout//cg.A.x matches nas-omp-lddout//bt.A.x (69)
nas-omp-lddout//cg.A.x matches nas-omp-lddout//bt.E.x (61)
nas-omp-lddout//cg.B.x matches nas-omp-lddout//bt.E.x (63)
nas-omp-lddout//cg.C.x matches nas-omp-lddout//bt.A.x (69)
nas-omp-lddout//cg.C.x matches nas-omp-lddout//bt.E.x (61)
...
```

Command for comparison of files in the folder for nmout

```
ml libtool
ssdeep-master/ssdeep -l -r -d nas-omp-nmout/
```

Command for comparison of files in the folder for nmout

```
[naraya0001@dmi-cl-login hari-msc-project]$ bash compare-folder.sh
nas-omp-nmout//ep.B.x matches nas-omp-nmout//ep.A.x (99)
nas-omp-nmout//ep.C.x matches nas-omp-nmout//ep.A.x (96)
nas-omp-nmout//ep.C.x matches nas-omp-nmout//ep.B.x (96)
nas-omp-nmout//ep.D.x matches nas-omp-nmout//ep.A.x (99)
nas-omp-nmout//ep.D.x matches nas-omp-nmout//ep.B.x (99)
nas-omp-nmout//ep.D.x matches nas-omp-nmout//ep.C.x (96)
nas-omp-nmout//ep.E.x matches nas-omp-nmout//ep.A.x (99)
nas-omp-nmout//ep.E.x matches nas-omp-nmout//ep.B.x (99)
...
```

Command for clustering of files in the folder for stringsout

```
ml libtool
ssdeep-master/ssdeep -l -r -d -g nas-omp-stringsout/
```

Result of clustering of files in the folder for stringsout generated two clusters with sizes 42 and 9

```
[naraya0001@dmi-cl-login hari-msc-project]$ bash cluster-all.sh
** Cluster size 42
nas-omp-stringsout//bt.A.x
nas-omp-stringsout//bt.B.x
nas-omp-stringsout//bt.C.x
nas-omp-stringsout//bt.D.x
nas-omp-stringsout//bt.E.x
nas-omp-stringsout//bt.F.x
nas-omp-stringsout//cg.A.x
nas-omp-stringsout//cg.B.x
nas-omp-stringsout//cg.C.x
...

** Cluster size 9
nas-mpi-stringsout//dt.A.x
nas-mpi-stringsout//dt.B.x
nas-mpi-stringsout//dt.C.x
nas-mpi-stringsout//dt.D.x
nas-mpi-stringsout//dt.E.x
...
...
```

Result of clustering of files for nmout generated mulitple clusters with different sizes

```
[naraya0001@dmi-cl-login hari-msc-project]$ bash cluster-all.sh
** Cluster size 6
nas-omp-nmout//ep.F.x
nas-omp-nmout//ep.A.x
nas-omp-nmout//ep.B.x
nas-omp-nmout//ep.D.x
nas-omp-nmout//ep.C.x
nas-omp-nmout//ep.E.x

** Cluster size 3
nas-omp-nmout//mg.A.x
nas-omp-nmout//mg.B.x
nas-omp-nmout//mg.D.x

** Cluster size 3
nas-omp-nmout//ft.D.x
nas-omp-nmout//ft.E.x
nas-omp-nmout//ft.F.x
...
```

Result of clustering of files with ldd generated a single cluster of size 45 for OpenMP

```
[naraya0001@dmi-cl-login hari-msc-project]$ bash cluster-all.sh
ssdeep: Did not process files large enough to produce meaningful results
** Cluster size 45
nas-omp-lddout//bt.A.x
nas-omp-lddout//bt.B.x
nas-omp-lddout//bt.C.x
nas-omp-lddout//bt.D.x
nas-omp-lddout//bt.E.x
nas-omp-lddout//bt.F.x
nas-omp-lddout//cg.A.x
nas-omp-lddout//cg.E.x
nas-omp-lddout//cg.B.x
nas-omp-lddout//cg.C.x
nas-omp-lddout//cg.D.x
nas-omp-lddout//ep.A.x
nas-omp-lddout//cg.F.x
...
```

We generated the outputs for MPI and OpenMP executables. Based on the results obtained for clustering and classification, we realized that there are better options than using ldd. Since ldd is dynamic and uses dynamic memory allocation, the hashes generated are entirely different. It would always result in creating one distinct cluster with all the applications.

We use the results of nm and strings to apply thresholds on the given cluster for finer clustering and identify the optimal threshold for classification.

Command for clustering of files in the folder for stringsout with threshold of 85 percent

```
ml libtool
ssdeep-master/ssdeep -l -r -d -g nas-mpi-stringsout/ -t 85
```

Using this command we came to the conclusion that at 85 percent similarity MPI executables have the optimal classification which is further discussed in the results section.

Cluster generated for stringsout with threshold at 85 percent which results in optimal classification of applications

```
[naraya0001@dmi-cl-login hari-msc-project]$ bash cluster-all.sh
** Cluster size 6
nas-mpi-stringsout//bt.A.x
nas-mpi-stringsout//bt.B.x
nas-mpi-stringsout//bt.C.x
nas-mpi-stringsout//bt.D.x
nas-mpi-stringsout//bt.E.x
nas-mpi-stringsout//bt.F.x

** Cluster size 5
nas-mpi-stringsout//bt.A.x.ep_io
nas-mpi-stringsout//bt.B.x.ep_io
nas-mpi-stringsout//bt.C.x.ep_io
nas-mpi-stringsout//bt.D.x.ep_io
nas-mpi-stringsout//bt.E.x.ep_io

** Cluster size 6
nas-mpi-stringsout//cg.A.x
nas-mpi-stringsout//cg.B.x
nas-mpi-stringsout//cg.C.x
nas-mpi-stringsout//cg.D.x
nas-mpi-stringsout//cg.E.x
nas-mpi-stringsout//cg.F.x

** Cluster size 4
nas-mpi-stringsout//dt.A.x
nas-mpi-stringsout//dt.B.x
nas-mpi-stringsout//dt.C.x
nas-mpi-stringsout//dt.D.x

** Cluster size 6
nas-mpi-stringsout//ep.A.x
nas-mpi-stringsout//ep.B.x
nas-mpi-stringsout//ep.C.x
nas-mpi-stringsout//ep.D.x
...
```

Chapter 4

Results

We evaluated our solutions based on the similarity percentage. We had to set a threshold for the similarity percentage between the different hashes. The percentage is manually set where we can visually notice that the executables are clustered as per the applications.

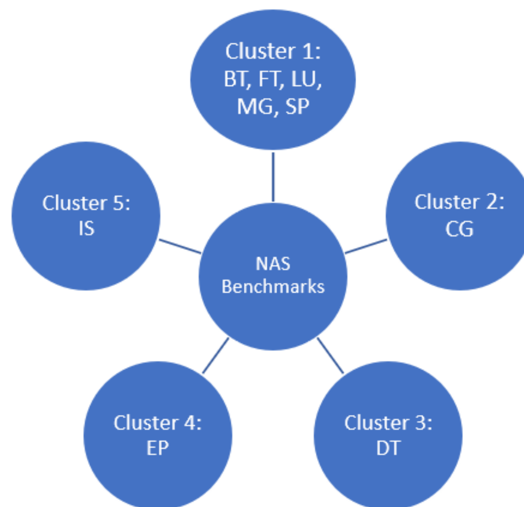

Figure 4.1: Clustering with a threshold at 55 percent(Clustered multiple applications together)

The hashes that generated the best similarity were the ones generated from **strings**. In our research, we classify binaries into application classes in different ways, with varying degrees of success, with ldd, nm, regular hashes, fuzzy hashes, etc. We found fuzzy hashes of strings to be most successful for our example data which had a success with 85 percent. Figure 4.1 depicts how decreasing the threshold combined multiple applications. It could be because they might be using similar libraries and functions, but they have entirely different functionalities, which are not ideal for classification. At 90 percent, as we see in Figure 4.3, executables of the same applications

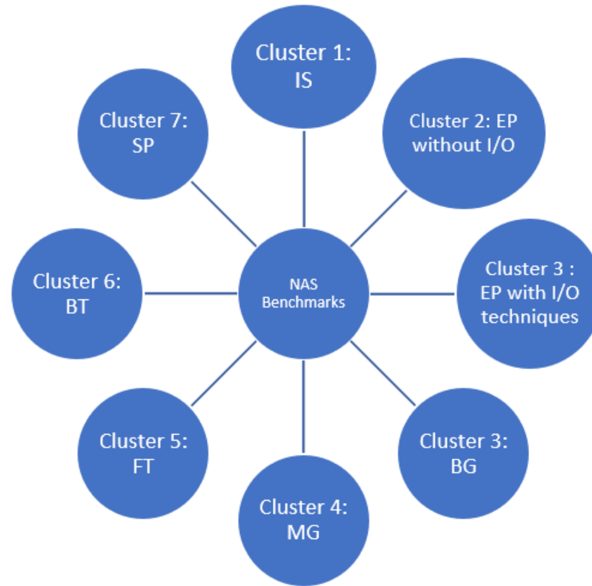

Figure 4.2: Clustering with a threshold at 85 percent(Most Optimal)

are clustered separately. It is because these executables might have different usages in the same application. Hence, a high threshold is overfitting for the whole dataset.

The advantages of classification are with the security (not being used for cryptocurrency). It would help in better understanding the system design and procurement and knowing which applications use the most resources and the characteristics of these applications. A minority of applications consumes many systems resources, and if we know which application it is, we can optimize the system for those specific applications. We can also perform optimization on applications that can be executed with lower CPU frequencies without losing performance, which would save energy(in terms of power consumption).

We could also understand performance variability and how the performance of specific applications changes over time. With the trends, we could figure out which applications are getting slower. It would need the applications to be labeled. HPC systems have pre-installed application labels that can help track which modules are used, identify modules that are not used anymore (potentially candidates to uninstall and free memory), or identify applications/codes that multiple

Figure 4.3: Clustering with a threshold at 90 percent(Separate clusters created for the same application i.e. for ft)

different users use, then include this app into the pre-installed list

Chapter 5

Discussion

The results indicate that we can predict the application without a proper naming convention. It will provide preliminary information for resource usage based on the application being executed. Some applications have consistent behavior or characteristics (CPU frequency, memory-bound, CPU-bound: co-scheduling, they may lead to failures than other applications, etc.)

One of the substantial limitations to this approach is that potentially some applications change their behavior drastically with input arguments, decreasing the usefulness of application binary labels (not affecting the security aspect)

The future aspect is to collect binary hashes of production jobs, matching or recognizing repeated executions. We need to investigate performance variation of production jobs to evaluate the potential success of job predictions for CPU frequency, co-scheduling, and backfilling.

Bibliography

[1] Keiji Yamamoto, Yuichi Tsujita, and Atsuya Uno, 2019, Classifying Jobs and Predicting Applications in HPC Systems.

[2] Denis Shaikhislamov and Vadim Voevodin, 2018, Solving the Problem of Detecting Similar Supercomputer Applications Using Machine Learning Methods.

[3] IBM documentation, 2022, nm Command, https://www.ibm.com/docs/en/aix/7.1?topic=n-nm-command

[4] NASA Advanced Supercomputing (NAS) Division, 2022, NAS Parallel Benchmarks, https://www.nas.nasa.gov/software/npb.html