# Record linkage for auto-tuning of High Performance Computing systems

Sophie Robert[1,2], Lionel Vincent[1], Soraya Zertal[2], and Philippe Couvée[1]

[1] Atos BDS R&D Data Management
sophie.robert@atos.net lionel.vincent@atos.net philippe.couvee@atos.net
[2] Li-PaRAD, University of Versailles
sophie.robert2@uvsq.fr soraya.zertal@uvsq.fr

**Abstract.** To become auto-adaptive, computer systems should be able to have some knowledge of incoming applications even before launching the application on the system, so that the runtime environment can be customized to the particular needs of this application. In this paper, we propose the architecture of an auto-tuner which relies on record linkage methods to match an incoming application with a database of already known applications. We then present a concrete implementation of this auto-tuner on High Performance Computing (HPC) systems, to submit unknown incoming applications with the best possible parametrization of a smart prefetch strategy by analyzing their metadata. We test this auto-tuner in conditions close to a production environment, and show an improvement of 28% compared to using the default parametrization. The conducted evaluation reveals a negligible overhead of our auto-tuner when running in production and a significant resilience for parallel use on a high-traffic HPC cluster.

**Keywords:** Record Linkage · Auto-tuning · Autonomous System · Performance · High Performance Computing

## 1 Introduction

Autonomous distributed and parallel computer systems should be able to provide run-time environments customized for the particular needs of each running application. When applications present regular characteristics and stable behaviors over time, which happens frequently in High Performance Computing (HPC), the system can be configured directly upon the submission of the application, by matching it with previously collected knowledge. This allows to configure the best possible runtime environment and make the most of the system's resources. In this paper, we propose the architecture of an auto-tuner which relies on record linkage methods to match the metadata of an incoming application collected by a workload manager with metadata collected on previously run applications, in order to take some auto-adaptive decisions before the submission of the application in a complete autonomous way. The main application of our auto-tuner that we present here is the development of auto-parametrizable runtime environments: once an application has been recognized as previously known, the

optimal running environment that had been computed for this application can be applied to the running platform. We will detail in depth the implementation of this auto-tuner for the case of a smart prefetch strategy for HPC systems. Upon the submission of the application, the auto-tuner uses the metadata to find the best possible parametrization for the prefetch strategy, given previously optimized applications. The main contributions of this paper are:

- The definition of the architecture of an auto-tuner to match incoming applications to previously run ones
- The original use of record linkage methods for matching applications through their metadata
- The validation of this auto-tuner in conditions close to production settings

In the remainder of the paper, we discuss in section 2 works that are related to record linkage and its use. Section 3 describes the architecture of the auto-tuner that we propose. In section 4 we describe in depth the methodology used for matching the incoming application with previously known applications. Section 5 describes the validation plan designed to test the performance of the matcher for an adaptive prefetch strategy and section 6 describes the results found through this validation plan. Conclusion and further works are detailed in section 7.

## 2 Related works

The automation of the tuning process using Record Linkage is absolutely original as no reference was found of close works in the literature. We can find works on prediction using the application's submission informations and the execution environment for HPC problems as the power-aware scheduling using inference models [16] or backfilling algorithms [5] and execution time using Decision Trees [17]. They exploit different Machine Learning methods but as far as we know, none of them used unsupervised method as we suggest, and especially record linkage techniques. While uncommon in computer science, record linkage is a very popular technique for matching data records across same individuals in the health and bio-statistics field.
*Record linkage* [2], also known as *data matching*, is a process which goal is to match several records corresponding to the same entity across difference data sources. The first idea of Record Linkage was introduced by Halbert Dunn in 1946 [4] as a book of life for each individual and since the 1950s several research domains explored, developed and applied this technique. The main popular ones are health [10] [6] [12] to improve the quality of clinical care and statistical studies as population census [7] to target the right development plans governments have to anticipate. This use of Record Linkage to develop many domains, goes with its own continuous optimization and a substantial work is achieved to preserve the record linkage privacy [13], cleaning and adjusting linkage errors [19]. While widely used in biostatistics and the medical field [7][8], record linkage has to our knowledge never been used to match computer records for application re-identification.

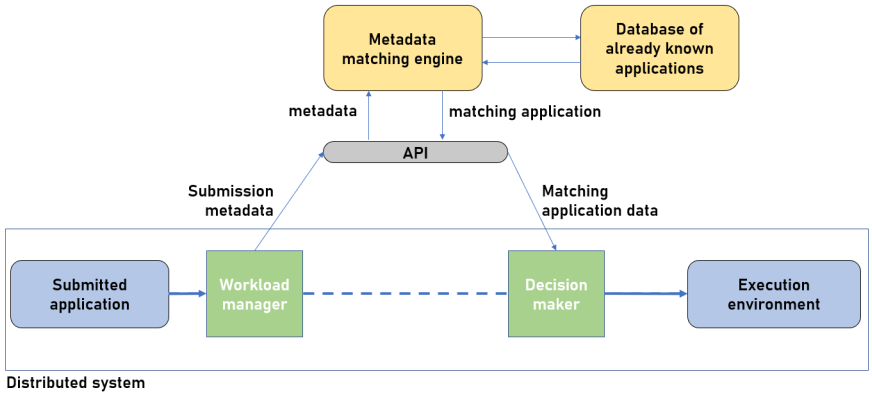


Fig. 1: Schematic representation of the matching pipeline

## 3 Suggested pipeline and practical implementation

We present in this section the workflow of the auto-tuner, as well as a practical implementation we use in order to quantify its efficiency.

### 3.1 Tuner workflow

Upon submission of an application by the user, the metadata of this application is collected, using environment variables in the submission environment such as the name of the user and the name of the program, but also the requested resources, such as the number of nodes. This data is then sent to the interface between the submission systems and the metadata matching engine. When this engine receives the data, it performs the matching using the records of applications that have been previously run on the system. These records contain the metadata as well as some information recorded after the run is finished. The application with the best match is then sent back to the API, which sends it to the module responsible for making decisions using this information. For example, this module could correspond to the scheduler that will adapt itself and select the most promising parametrization of the prefetcher. The application can then be run on the system and its information will be collected to enrich the knowledge base.
In our auto-tunner architecture, the submission of the application hangs until the matching process is over, thus requiring the matching process to be efficient enough to not affect the performance of the user.

### 3.2 Auto-tuning of HPC components using the tuner

Most modern hardware or software come with many configuration parameters, which have a strong impact on the performance. This is for example the case of hardware components, MPI collectives, I/O accelerators ... The impact of these

parameters are highly dependent on the characteristics of the running application, and running the system with the best possible configuration is crucial to make the most of the performance of the system. Several methods are available to find the optimal parametrization of a system given the application's input, whether relying on agnostic tuning methods such as black-box optimization or analytical models which predicts the behavior of the system. Once found, this configuration has to be set manually by the user each time the optimized application is run and this is a human error prone process. Furthermore, it did not allow to apply a configuration for new unseen incoming applications, although the knowledge of other applications that have already been evaluated on the system could lead to apply a better configuration than the default one. We adapt the tuner workflow of section 3.1 to use it to automatically set the appropriate configuration of a HPC component given its optimal configuration. As described in figure 2, the database of already known applications contains data collected from previously optimized applications: the application metadata, as well as their optimum parametrization, found through black-box optimization. This optimal parametrization is then used by the system to run the application.

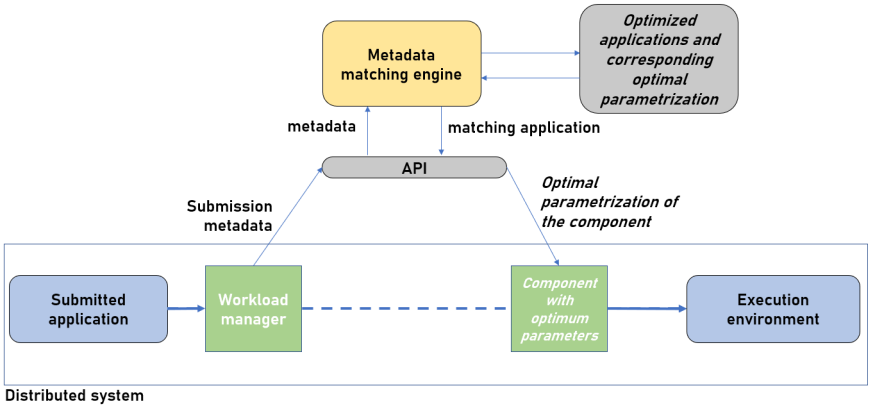


Fig. 2: Particular implementation of the auto-tuner for I/O accelerators

### 3.3 Practical implementation and use-case

We test the metadata matching system for auto-tuning of an I/O accelerator, called the *Small Read Optimizer* [1] acting as a prefetch strategy. The configuration of this accelerator has both a strong impact on the performance and a strong dependence on the particular profile of the application. Our particular implementation is done on a HPC cluster, using the Slurm workload manager [9]. The metadata collected by this workload manager are described in table 1. The op-

Table 1: Metadata collected by the workload manager and used for matching

| Variable name | Description | Variable type |
|---|---|---|
| Username | Name of the user that submitted the application | String |
| Jobname | Name of the job | String |
| Program name | Name of the running program | String |
| Command line | Command line used to submit the program | String |
| Start time | Time of submission of the application | Date |
| Node count | Number of nodes the application is running on | Integer |

timal parametrization is found using a black-box optimization framework called SHAMan [14], used to collect the metadata of the application as well as find the optimal parametrization of the accelerator. More details on this accelerator and its parametrization are available in [15].

## 4 Matching methodology

In this paper, we use deterministic record linkage [20] to perform the matching. The process of metadata matching is organized according to 3 steps, as represented in figure 3. Each step is linked to the other, hence the term of pipeline.

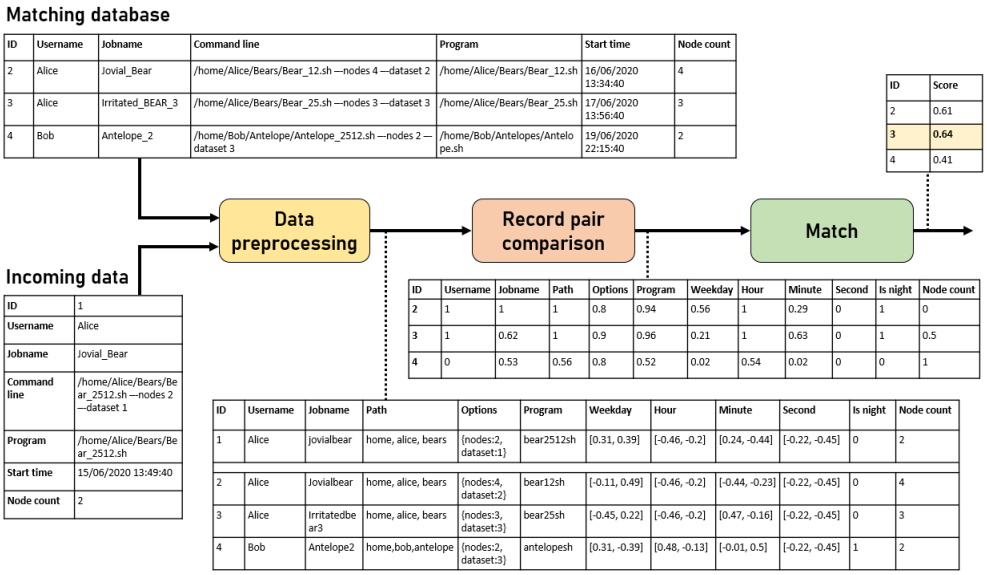


Fig. 3: Schematic representation of the matching pipeline

### 4.1 Data preprocessing

The goal of the pre-processing step is to transform the data into a standard format so that it can be optimally compared. In our matching engine, we use 3 distinct steps of data preprocessing before performing the matching:

- **Data cleaning**: it is necessary to remove possible unnecessary characters.
- **Data segmentation**: its goal is to ensure that at the output of the cleaning block, there is no redundancy in the data.
- **Extraction of new data**: it consists in adding a possible new field to the dataset.

The cleaning and extraction process created for each of the metadata variables described in table 1 are explicited in table 2. An example of extraction and cleaning are available in figure 3.

Table 2: Cleaning and extraction performed on the metadata

| **Variable name** | **Applied cleaning** | **Extracted variables** |
|---|---|---|
| Username | None | |
| Jobname | Removal of non-alphanumeric symbols, lowercase | |
| Program name | Removal of non-alphanumeric symbols, lowercase | |
| Command line | None | Path to the program Options of the program |
| Start time | Projection in a trigonometric circle (depending on the periodicity of each variable) | Second<br>Minute<br>Hour<br><br>Day<br>Weekday |
| Node count | None | |

Table 3: Comparison function applied for each of the available fields

| **Variable name** | **Comparison function** |
|---|---|
| Username | Exact comparison |
| Jobname | Levenshtein similarity |
| Program name | Levenshtein similarity |
| Path | Soft TFIDF comparison |
| Options | Default dict comparison |
| Weekday | Euclidean comparison |
| IsNight | Exact comparison |
| Minutes | Euclidean comparison |
| Hours | Euclidean comparison |
| Node count | Absolute comparison |

### 4.2 Record Pair comparison

The comparison step consists in comparing 2 records field by field. As the fields, described in table 1, are of heterogeneous types, an adapted similarity measure must be used for each field depending on its type, in order to compute the similarity between the 2 records. At the end of this step, we obtain a vector with a similarity score for each of the already available applications for matching with the incoming application. Because we do not want a particular field to weigh more than another, each of the used similarity score gives a value located between 0 and 1.
Depending on the nature of the field, one of the similarity functions bellow and summarized in table 3, will be applied:

- **Exact comparison**: Exact comparison consists in returning a boolean if the field is exactly equal for both records, regardless of their data type . The fields concerned by this similarity are the username as they are set by the system , and the boolean field which indicates if a job is running at night.
- **Single string comparisons**: To perform single string comparisons between the two string variables jobname and program name, we use the Levenshtein similarity measure [11]. It computes the similarity between two words as the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other. The Levenshtein distance between two strings $a$ and $b$ is defined as:

$$sim_{lev} = \begin{cases} |a| \text{ if } |b| = 0 \\ |b| \text{ if } |a| = 0 \\ sim_{lev}(a,b) \text{ if } a[0] = b[0] \\ 1 + min \begin{cases} sim_{lev}(\text{tail}(a), b) \\ sim_{lev}(a, \text{tail}(b)) \\ sim_{lev}(\text{tail}(a), \text{tail}(b)) \end{cases} \end{cases}$$

  with the function *tail* a function returning the string without its first element.
- **Option comparisons**: To compare the options of the application between the two records, we compute for each option the ratio of values in common given for this key, and attribute the value of this ratio to the score. When two applications have the same option name or key, we add a constant weight depending on each value to the score for each key in common in the option dictionary.
- **Multiple string comparison**: Soft Tf-Idf [3] consists in considering how frequently vario leter combinations appear in the whole dataset, instead of just taking into account the values for the two records. It allows to downplay strings that are repeated across every record (in our case, the /home/ suffix is very common among command lines) in order to focus on the less frequent ones. The frequency of the terms present in the records are computed and then divided by their frequency over the whole dataset.

– **Numerical comparisons**: To locate the similarity score between 0 and 1, we can only calculate traditional distance measures on values already located between -1 and 1, such as the coordinates extracted from the trigonometric projection. For the other numeric measures as the number of nodes, we use the *absolute* similarity, which sets the distance to zero if the two elements are too far away when compared to a threshold $d_{max}$, and otherwise to the distance between the two data points normalized by the threshold.

$$sim_{abs} = \begin{cases} 1 - \frac{|n_1 - n_2|}{d_{max}} & \text{if } |n_1 - n_2| \leq d_{max} \\ 0 & \text{else} \end{cases}$$

with:

- $n_1$: the first number to compare
- $n_2$: the second number to compare

In our case, we set the threshold to the number of maximum of nodes available in the cluster.

### 4.3 Matcher

The goal of the matcher is to determine the similarity output during the comparison step. We compute the average value of the similarity score, normalized by the number of extracted field per original fields so that fields like the submission time do not weigh more than others because of their many extracted fields. If no record go above a set threshold, then the application is considered to not have any match and their parametrization is set to the default one. Otherwise, the application with the highest score is set for matching.

In our case, the weighted matcher score is:

$$\begin{aligned} \text{score} = & \frac{1}{6} sim_{\text{username}} + \frac{1}{6} sim_{\text{jobname}} + \frac{1}{6} sim_{\text{program}} \\ & + \frac{1}{12}(sim_{\text{path}} + sim_{\text{options}}) \\ & + \frac{1}{24}(sim_{\text{second}} + sim_{\text{minutes}} + sim_{\text{hours}} + sim_{\text{weekDay}}) \\ & + \frac{1}{6} sim_{\text{nodeCount}} \end{aligned}$$

For our experiment, we have decided to put some even weights on every similarity and thus on every collected metadata to avoid any design bias, but in practice, it is advised to perform a careful examination of the behavior of the users on the particular cluster in order to select the weights of each similarity. These weights are chosen empirically and their relevance for each use-case has to be evaluated using performance metrics.

# 5 Validation plan

To validate our auto-tuner and appreciate its usefulness in a conditions close to those encountered in production clusters, we built a validation environment based on a small HPC cluster of 5 compute nodes and 1 master node. Three different users are created to represent several user behaviors observed from a real setting: each user launches their needed applications in their particular way.

## 5.1 Cluster usage scenario

We started by creating a pool of possible applications, defined by their I/O characteristics and their names. Each one corresponds to an HPC application which I/O patterns can be altered by a different parametrization or a different input dataset. These applications are generated thanks to a home-made highly configurable I/O generator. It allows to generate any combination of parallel and sequential I/O patterns representing the set of HPC applications that users are allowed to run on the production cluster. In our validation scenario, three different users are using the cluster, each with their own behavioral profile. Each user has access to every application in the defined pool and can run it on the cluster, possibly adding its particular variations (e.g. application options or input dataset) composing a job per variation. The I/O characteristics of the application remain the same regardless of the user who picked it, but the metadata used for the application depends on the user: for example the user can choose the jobname, he can edit the program name or add some options to the command line. The possible variation in the metadata is detailed in section 5.3. The users can choose to run the application once but they can also choose to use a black-box optimization tuning framework. These applications will then be included in the database used to perform the matching and will have their metadata stored as well as the optimal parametrization found by the black-box tuner. This validation scenario allows us to have a set of jobs owned by each different user, as well as a set of optimized job.

## 5.2 Tested applications and environment

In practice, we have designed a set of 9218 different possible applications the users can pick from. The selection of the applications per each user is done according to random distributions described in table 4, to model different type of user behaviors. In practice, we model the behavior of 3 different users as described in table 4. Table 5 gives the final number of each application selected by each user as well as the number of optimized ones.

## 5.3 Variation in metadata

As said in the previous section, each user launches an application with a different metadata, called job. The metadata that can be changed by the user are:

Table 4: Parametrization of the metadata matching experiment

| **User ID** | **Number of applications** | **Number of jobs per application** | **Number of optimized jobs per application** |
|---|---|---|---|
| 1 | 10 | $\mathcal{U}(5, 10)$ | $\mathcal{U}(1, 2)$ |
| 2 | 20 | $\mathcal{U}(1, 5)$ | $\mathcal{U}(1, 3)$ |
| 3 | 25 | $\mathcal{U}(5, 15)$ | $\mathcal{U}(2, 4)$ |

Table 5: Applications and experiment numbers per used ID

| **User ID** | **Number of applications** | **Total number of jobs** | **Total number of optimization** |
|---|---|---|---|
| 1 | 10 | 73 | 13 |
| 2 | 20 | 66 | 40 |
| 3 | 25 | 227 | 71 |

- **Username** The name of the user is set by the user ID using the application.
- **Jobnames** The only variation in the metadata performed by the user is the modification of the jobname, which is in practice highly dependent on the user behavior and his choices. We define two possible users' behavior. Each application has a root name, and the users can randomly add a prefix and a suffix containing the selected configuration. This mimics some guidelines that have been observed on systems' running in production. For example, if we have an application called `ant`, in two thirds of the cases, an adjective randomly picked is appended as a prefix to the jobname (becoming for example `angry_ant`) and in one third of the cases, the number of nodes is added at the end of the jobname (running on 2 nodes, `angry_ant_2`).
- **Nodes configuration** Once the user has selected an application, the topology of the system is randomly drawn from a list of available compute nodes. These nodes have different names and configurations and their choices have a strong impact on the behavior of the application and thus on the impact of the auto-tuner. This configuration is reflected by the number of nodes.

### 5.4 Evaluation metrics

We evaluate the usability of our auto-tuner according to two metrics:

1. **Impact of the auto-tuner on execution times**: We compare the time spent by the application when using the parametrization selected by the matcher, against the one using the default parametrization. This quantifies the positive impact of our auto-tuner on users' usage of the cluster.
2. **Matching elapsed time**: We evaluate the time needed to perform the matching between the incoming application and the previously executed applications, as a function of the number of applications in the matching

database. We also provide some insights in the load resilience of the suggested architecture, by testing the impact of having several users launching their application in parallel.

Additionally, we provide insight on the relevance of the matches performed by the auto-tuner and discuss how the quality of these matches impacts the performance gain as well as the matching score. We also examine the relationship between the users' behavior and the observed performance gain attributed to the tuning process.

## 6 Results

### 6.1 Impact of the auto-tuner on execution times

Over all runs we find a median improvement of 28.39% (26.06% in mean) over the 366 different jobs and 41 applications, compared to using the application with the default parametrization. When removing the 150 jobs who had been directly optimized by the user, we still have a median performance improvement of 27.50% (25.47%) over applications that have never been seen by the optimizer. The accelerability potential of each job has a strong dependence on the used application, but we found that overall, on average, no application was slowed down by using the tuner. When looking at individual runs, we find that only 7 runs out of the 366 were slowed down because of our tuner, compared to the default parametrization, for a median loss of only 3% in execution time. Overall, these results highlight the performance and usefulness of our matcher when running in production.

### 6.2 Matching quality

The study of the behavior of the matcher shows that the right application is matched in 92% of the cases and the right number of nodes are matched in 56% of the cases. As described in table 6, a matching application and a matching number nodes yield a better matching score and bring a better improvement in performance than applications that do not match (from 25% to 29%).

Table 6: Median value of the score and the auto-tuning gain

| **Matched application** | **Matched nodecount** | **Score** | **Auto-tuning gain (%)** | **Number of jobs** |
|---|---|---|---|---|
| False | False | 0.73 | 25.61 | 13 |
| | True | 0.74 | 24.73 | 15 |
| True | False | 0.80 | 25.93 | 148 |
| | True | 0.93 | 29.27 | 190 |

### 6.3 Impact of the user's behavior on the tuning process

When designing our experiment, we selected several different users' profile. The distribution of the score and auto-tuning gain per user profile is reported in table 7. We find that the user with the highest ratio of experiments per number of jobs (user 2) is the one who benefits the most from the auto-tuner. The user with the lowest ratio of experiment is the one with the lowest measured gain.

Table 7: Distribution of score and auto-tuning gain per user profile

| User ID | Average score | Avg. auto-tuning gain |
|---|---|---|
| 1 | 0.78 | 23.85 |
| 2 | 0.87 | 27.40 |
| 3 | 0.88 | 26.24 |

### 6.4 Elapsed time

**Time to match** The time required for matching a single application to databases of different size are described in figure 4. The tests are performed with an API running on a server with 94GB of RAM, with an Intel(R) Xeon(R) CPU E5-2470 0 @ 2.30GHz with 16 cores and only 4 nodes dedicated to running the matching API. We find that for a very large database of 2000 job, the submission time goes up to 20 seconds. While this response time can seem high, it has to be considered in the perspective of the HPC context: upon submission, the application is sent to the workload manager queue and can stay there for several hours until the cluster is available, especially for cluster under high traffic. Also, HPC applications often take hours to days to run.

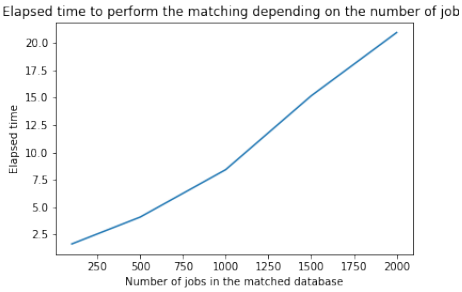


Fig. 4: Time to match applications per database size

**Load tests** To test the resilience of the API to parallel submission by different users, we performed some load testing on the system running the API, for a database of 150 jobs. We perform a series of load tests simulating the behavior of users running applications on the cluster. We test a population of up to 500 users submitting an application every second. The results of the tests are reported in figure 5. In the worst case scenario, we find a worst response time

of 180 seconds, when 500 users are submitting one application every second in parallel. While the number of users stays below 200, the response time is inferior to 30 seconds (submitting a request every second), which makes the use of the auto-tuner transparent to the user even in a very high traffic scenario.


Fig. 5: Load tests on the API

## 7 Conclusion and further works

In conclusion, we present in this paper the architecture of an auto-tuner which relies on record linkage techniques to match an unknown incoming application with a database of previously run applications. We evaluate this matcher by auto-tuning a smart prefetch strategy and show a median performance improvement of 28% compared to using the default parametrization over a dataset representative of conditions found in production. An analysis of required elapsed times show a negligible overhead for a user submitting his job, and load tests show that up to 200 users launching a new job every second do not have an impact on the user's experience when submitting a new job.
In terms of further works, we aim at testing this auto-tuner on a cluster running in production, to have a better understanding of the user's behavior. Another research axis is the development of a probabilistic record linkage rather than the deterministic to reduce the linkage errors [18] and improve the quality of our uto-tuning as the probabilistic method theoretically captures more matches.

## 8 Acknowledgments

This work was partially supported by the EU project "ASPIDE: Exascale Programming Models for Extreme Data Processing" under grant 801091. We would also like to acknowledge Thibaut Arnoux for his contributions to this paper.

## References

1. Atos: Tools to improve your efficiency (2018)

2. Christen, P.: Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection. 2012
3. Cohen, W.W., Ravikumar, P., Fienberg, S.E.: A comparison of string distance metrics for name-matching tasks. In: Proceedings of the 2003 International Conference on Information Integration on the Web. p. 73–78. IIWEB'03, AAAI Press (2003)
4. Dunn, H.: Record linkage. American Journal of Public Health **66**, 1412–1416 (1946)
5. Dutot, P., Georgiou, Y., Glesser, D., Lefevre, L., Poquet, M.and Rais, I.: Towards energy budget control in hpc. In: EEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). p. 381:390 (2017)
6. Haukka, J., Sankila, R., Klaukka, T., Lonnqvist, J., L., N., A., T., Wahlbeck, K., J, T.: Incidence of cancer and statin usage—record linkage study. International Journal of Cancer **126**, 279:84 (2010)
7. Jaro, M.A.: Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. Journal of the American Statistical Association **84**(406), 414–420 (1989)
8. Jaro, M.A.: Probabilistic linkage of large public health data files. Statistics in medicine **14**(5-7), 491–498 (1995)
9. Jette, M., Yoo, A., Grondona, M.: Slurm: Simple linux utility for resource management (07 2003)
10. Kelman, C.W.and Bass, J., Holman, D.: Research use of linked health data – a best practice protocol. Aust NZ Journal of Public Health **26**, 251–255 (2002)
11. Miller, F.P., Vandome, A.F., McBrewster, J.: Levenshtein Distance: Information Theory, Computer Science, String (Computer Science), String Metric, Damerau?Levenshtein Distance, Spell Checker, Hamming Distance. Alpha Press (2009)
12. Mitchell, R., Braithwaite, J.: Evidence-informed health care policy and practice: using record linkage to uncover new knowledge. Journal of Health services Research policy **26** (2021)
13. Ranbaduge, T., Chriten, P.: A scalable privacy-preserving framework for temporal record linkage. Journal of Knowledge and Information Systems **62**, 45–78 (2020)
14. Robert, S., Zertal, S., Goret, G.: Shaman: An intelligent framework for hpc autotuning of i/o accelerators (2020)
15. Robert, S., Zertal, S., Vaumourin, G., Couvée, P.: A comparative study of black-box optimization heuristics for online tuning of high performance computing i/o accelerators. Concurrency and Computation: Practice and Experience **n/a**(n/a), e6274. https://doi.org/https://doi.org/10.1002/cpe.6274, https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6274
16. T., S., Weill, J.C., Mougeot, M.: Predicting job power consumption based on rjms submission data in hpc systems. In: International conference on high Performance Computing. p. 63:82 (June 2020)
17. Tanash, M., Dunn, B., Andresen, D., Hsu, W., H., Y., Okanlawon, A.: Improving hpc system performance by predicting job resources via supervised machine learning. In: Proceedings of the PEARC. p. 1:8 (JUly 2019)
18. Tromp, M., C., R.A., Bonsel, A., H., B., R.J.: Results from simulated data sets: probabilistic record linkage outperforms deterministic record linkage. Journal of clinical epidemiology **64**, 565:572 (2021)
19. Winkler, W.E.: Cleaning and using administrative lists: Enhanced practices and computational algorithms for record linkage and modeling/editing/imputation. In: Administrative Records for Survey Methodology. Wiley Online Library (2021)
20. Zhu, Y., Matsuyama, Y., Y., O., S., S.: When to conduct probabilistic linkage vs. deterministic linkage? a simulation study. Journal of Biomedical informatics **56**, 80:86 (2015)