École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise

Mémoire d'ingénieur

« Analyse de la consommation des ressources des systèmes HPC. »

---

par
Tàzio Gennuso

Maître de stage :	Dr. Sébastien Varrette
Tuteur académique à l'ENSIIE :	Christophe Mouilleron
Diplôme préparé :	Ingénieur ENSIIE (promotion 2019)

Stage de fin d'études effectué du 1er mai 2019 au 31 octobre 2019
Présenté publiquement le 13 novembre 2019

Préparé au sein de :
University of Luxembourg
Interdisciplinary Centre for Security Reliability and Trust (SnT)
Parallel Computing and Optimization Group (PCOG)
6, avenue de la Fonte, L-4364 Esch-sur-Alzette, Luxembourg
HPC for Research hpc.uni.lu

supervisor : Dr. Sébastien Varrette

location :
University of Luxembourg
Interdisciplinary Centre for Security Reliability and Trust (SnT)
Parallel Computing and Optimization Group (PCOG)
6, avenue de la Fonte, L-4364 Esch-sur-Alzette, Luxembourg
HPC for Research hpc.uni.lu

RÉSUMÉ

L'ordonnancement des jobs est un point critique pour les systèmes HPC, qui doivent garantir une disponibilité optimale. Aujourd'hui les heuristiques d'allocation de ressources des ordonnanceurs pour déterminer l'ordre d'exécution des jobs sont basées sur les ressources demandées par l'utilisateur, en fonction des ressources disponibles. Cependant, un certain nombre d'allocations sont faites de manière non optimale puisqu'il apparaît que qu'une faible part des ressources allouées est finalement réellement utilisée. Mieux comprendre les besoins des utilisateurs de manière à les anticiper permettrait de proposer des allocations sur des systèmes optimisés qui correspondraient mieux à l'utilisation des ressources. C'est évidemment crucial dans l'optique de l'amélioration de l'efficacité énergétique mais aussi de l'utilisation des ressources de calculs mises à disposition.

Dans ce contexte, les travaux présentés dans ce rapport sont une première étape vers cet objectif, avec une analyse approfondie de l'utilisation actuelle du cluster `iris`, le principal supercalculateur de la plateforme HPC de l'Université du Luxembourg. Plusieurs heuristiques sont proposées pour permettre de catégoriser les jobs soumis afin de déterminer un nombre minimal de profils de soumission en vue d'une optimisation future. Dans ce cadre, une heuristique de classification automatique a été proposée aboutissant à la définition de 5 groupes de jobs partageant un profil d'utilisation de ressources HPC communes selon 7 dimensions représentatives. Dans l'optique d'améliorer la classification obtenue, d'autre pistes sont également proposées et ont été testées pour permettre d'étendre les sources d'information associées aux profils identifiés (notamment au niveau des modules logiciels utilisés lors des réservations).

Mots-clefs : HPC, Slurm, Partitionnement Automatique, Monitoring, Environment Module

REMERCIEMENTS

J'adresse ces remerciements à toutes les personnes que j'ai rencontrées durant ce stage de fin d'études et qui ont participées à en faire l'expérience que j'ai vécue.

Tout d'abord je remercie Sébastien Varrette, chercheur à l'Université du Luxembourg et chef de l'équipe HPC pour son suivi et ses conseils lors de ce stage, qui m'auront été très instructifs.

Je remercie également de façon générale tous les membres de l'équipe HPC (qui administrent et supportent les supercalculateurs de l'Université du Luxembourg) pour leur accueil chaleureux, et particulièrement Emmanuel Kieffer pour ses conseils avisés concernant la classification automatique, Hyacinthe Cartiaux et Valentin Plugaru pour m'avoir fait visiter les salles serveurs du supercalculateur Iris, et enfin Frédéric Pinel pour nos échanges aussi variés qu'intéressants.

Et finalement, merci à Pascal Bouvry et Sébastien Varrette (à nouveau) pour m'avoir en premier lieu donné l'opportunité de faire mon stage de fin d'étude au sein de leur équipe.

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

ABRÉVIATIONS

Voici la liste des sigles et abréviations utilisés dans ce rapport, avec leur signification respective :

DBSCAN Density-Based Spatial Clustering of Applications with Noise
DLC Direct Liquid Cooling
ENSIIE École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise
HPC High Performance Computing
PCA Principal Component Analysis
PCOG Parallel Computing and Optimisation Group
PUE Power Usage Effectiveness
RJMS Resource and Job Management System
UL University of Luxembourg
ULHPC Centre HPC de l'Université du Luxembourg

1 INTRODUCTION

Ce rapport a été écrit dans le contexte d'un stage de fin d'études d'ingénieur (équivalence master 2). Ce stage a pris place à l'Université du Luxembourg University of Luxembourg (UL), sous la supervision du Dr. Sébastien Varrette, chercheur au sein de l'équipe PCOG[1]. Conjointement avec le Prof. Pascal Bouvry, il dirige le développement de la plateforme de calcul haute performance (*High Performance Computing (HPC) High Performance Computing* (HPC)) de l'Université[2], ainsi que l'équipe d'experts qui la gèrent et assure le support auprès chercheurs exploitant cette infrastructure. Il poursuit également des recherches dans le domaine de la sécurité de la performance des plateformes de calcul parallèles et distribués, telles que le HPC ou les infrastructures de Cloud Computing.

Dans ce contexte, l'objectif de ce stage était de contribuer au projet Energumen qui vise à développer de nouveaux algorithmes d'allocation de tâches afin de permettre des économies d'énergie au sein des grandes plate-formes distribuées de type HPC. Une tâche est composée d'un ensemble de processus, potentiellement inter-communicants et exécutant une ou plusieurs applications. Une classification des différents modèles de tâches a été proposées dans [3] en les caractérisant par le niveau de flexibilité des tâches et du sous-ensemble de ressources physiques allouées pour l'exécution de ces tâches. On distingue ainsi :

- Les tâches *rigides* qui utilisent une quantité constante (et déterminée à l'avance) de ressources de calculs durant toute leur exécution. Ces tâches ne peuvent pas fonctionner avec une quantité inférieure de ressources, et n'utilisera pas toute nouvelle ressource accordée en supplément.
- Les tâches *malléables* qui s'adaptent à la quantité de ressources allouées au début et pendant leurs exécutions ;
- Les tâches *évolutives*, passant par différentes phases d'une durée variables. Lorsque la tâche change de phase, sa consommation en ressources est susceptible de varier ;
- Les tâches *préemptibles*, susceptible d'être suspendues dans un état consistant puis être relancées ultérieurement.

En particulier, le projet Energumen s'intéresse à l'ordonnancement efficace d'une point de vue énergétique de tâches *malléables* sur une platforme HPC à large échelle en tenant compte des transferts de données. Pour cela, il est nécessaire de coupler la politique d'ordonnancement au gestionnaire de ressources en place sur la plateforme de calcul[3], pour permettre un taux d'occupation des ressources le plus élevé possible, tout en satisfaisant les requêtes des utilisateurs et en minimisant la consommation énergétique.

Le but de ce stage était de contribuer aux objectifs du projet par 3 activités :

1. Établir un mécanisme de collecte de données relatives aux tâches exécutées sur le cluster HPC de l'Université de Luxembourg (via le gestionnaire de ressource SLURM).

1 Parallel Computing and Optimisation Group (PCOG) - voir `https://pcog.uni.lu`
2 Voir `https://hpc.uni.lu`
3 Appelé *Resource and Job Management System (RJMS)*

2. Identifier les variables influentes sur le temps d'exécution, et la consommation énergétique. Ces variables décrivent l'environnement d'exécution, et les tâches.

3. Construire un modèle prédictif (performance, énergie) par apprentissage statistique (machine learning).

1.1 L'UNIVERSITÉ DU LUXEMBOURG ET SON CENTRE HPC

Ce stage s'est donc déroulé au sein de l'University of Luxembourg (UL)[4]. Fondée en 2003, l'UL est la seule université publique au Grand-Duché de Luxembourg. Multilingue, internationale et centrée sur la recherche, elle se définit aussi comme une institution moderne et à visage humain. Elle compte 1960 collaborateurs (scientifiques et chercheurs) qui épaulent 270 professeurs, assistants-professeurs et chargés de cours, et se veut proche des institutions européennes et de la place financière du Luxembourg.
L'Université est structurée avec trois facultés, et trois centres interdisciplinaires :

- La Faculté des Sciences, de la Technologie et de la Communication (FSTC) ;
- La Faculté de Droit, d'Économie et de Finance (FDEF) ;
- La Faculté des Lettres, des Sciences humaines, des Arts et des Sciences de l'Éducation (FLSHASE) ;
- L'Interdisciplinary Centre in Security, Reliability and Trust (SnT) ;
- Le Luxembourg Centre for Systems Biomedicine (LCSB) ;
- Luxembourg Centre for Contemporary and Digital History (C²DH).

Disposant d'un budget de 251M€ en 2018, l'UL s'investit dans la recherche autour de six priorités et des domaines de recherche d'excellence interdisciplinaire en pleine expansion, onze unités de recherche, sept chaires de dotation et quatre écoles doctorales.
Enfin, l'Université est présente sur trois sites : le Campus de Belval, au sud du pays, le Campus de Kirchberg, dans la capitale, et le Campus de Limpertsberg, également dans la capitale.

J'ai effectué mon stage dans les locaux du Campus de Belval, au sein de l'équipe PCOG et HPC de l'Université. Depuis 2006, l'Université du Luxembourg a investi plus de onze millions d'euros dans ses équipements HPC au sein du Centre HPC de l'Université du Luxembourg (ULHPC). L'objectif était avant tout de développer une forte puissance de calcul, combinée à une immense capacité de stockage de données accélérant la recherche en calculs intensifs et en analyse de données massive (Big Data). Par cette caractéristique, le centre HPC de l'Université se distingue de la plupart des autres équipements HPC, qui se concentrent souvent uniquement sur l'un de ces deux piliers.

À l'heure actuelle, cette initiative fournit à plus de 500 chercheurs (informaticiens, ingénieurs, physiciens, chercheurs en sciences des matériaux, biologistes et même économistes) la possibilité d'effectuer des calculs intensifs et de stocker massivement les données nécessaires dans le cadre de leur recherche. Toutes les universités de classes mondiales nécessitent ce type de plate-forme pour accélérer les recherches effectuées en leurs sein. La plateforme est entre autres utilisée pour faire de la recherche en physique, cryptologie, en validation de

4 http://www.uni.lu

logiciels ou en simulation et analyse des réseaux de communication. La vaste capacité de stockage installée joue un rôle essentiel pour la recherche en physique des polymères ou en sciences des matériaux, pour les simulations en recherche économique ou encore la recherche biomédicale sur les maladies neurodégénératives telles que les maladies de Parkinson ou d'Alzheimer. Le centre HPC offre aux facultés et aux centres interdisciplinaires de l'Université un avantage comparatif considérable. La disponibilité des énormes ressources informatiques, combinée aux vastes installations de stockage des données, permet de créer un environnement de recherche attrayant qui motive les chercheurs à rejoindre l'Université du Luxembourg. Cet équipement HPC est un atout stratégique pour l'Université, mais aussi un facteur essentiel pour la compétitivité scientifique et donc économique du Luxembourg.

La plateforme HPC est gérée par une équipe d'experts sous la direction du Prof. Pascal Bouvry et du Dr. Sébastien Varrette qui a supervisé ce stage. Cette infrastructure a continué à évoluer grâce aux efforts continus de l'équipe UL HPC (composée de Dr. S. Varrette, V. Plugaru, S. Peter, H. Cartiaux, C. Parisot, Dr. F. Pinel, Dr. E. Kieffer, Dr. Ezhilmathi Krishnasamy). Elle fournit en 2019 une puissance de calcul totalisant 1262 TeraFlops (1 TeraFlops = 10^{12} opérations en virgule flottante par seconde) et près de 9 PetaByte de stockage de données partagées. Grâce à cette capacité, le centre HPC de l'Université du Luxembourg se positionne comme l'un des principaux acteurs de la Grande Région Saar-Lor-Lux en matière de HPC et de Big Data.

Un apercu des clusters proposés sur la plateforme ULHPC est proposé dans le tableau 1.

Cluster	**Localisation**	**#Nœuds**	**#Cores**	R_{peak} [TFlops]	GPU R_{peak}
`iris`	Belval CDC	196	5824	347.65	748.8
`gaia`*	Belval BT1	273	3440	69.296	76.22
`chaos`*	Kirchberg	81	1120	14.495	0
`g5k` (Grid5000)	Kirchberg	38	368	4.48	0
`nyx`* (experimental)	Belval BT1	102	500	2.381	0
TOTAL :		**690**	**11252**	**438.302**	**+ 825.02 TFlops**

TABLE 1 : Caratéristiques des clusters de la plateforme ULHPC
* : Mise hors service fin 2019

A noter qu'un nouveau cluster (`aion`) est en cours d'acquisition et sera déployé en mars 2020 pour venir compenser la mise hors service des clusters historiques de la plateforme (`gaia` et `chaos`) et compléter l'offre proposée par `iris`.

1.2 UTILISATION ET ORDONNANCEUR HPC

Un cluster de calcul, ou cluster HPC, et un ensemble d'ordinateurs (appelés *nœuds*) dédiés au calcul scientifiques, qui sont connectés entre eux qui travaillent tous ensembles, comme une machine unique du point de vue de l'utilisateur. Une telle configuration permet d'augmenter la disponibilité des ressources, de faciliter la montée en charge d'utilisation et la répartition de la charge d'utilisation, et de faciliter la gestion des ressources. La gestion et le partage de tâches et de ressources (RJMS comme indiqué précédemment) est assuré par un logiciel appelé *ordonnanceur* (*schedulers* en anglais). Lorsqu'un utilisateur souhaite exécuter une tâche sur le cluster, il soumet un *job* qui est placé dans une *queue* de jobs. L'ordonnanceur a alors pour objectif de gérer et de planifier l'exécution de ces jobs dans le temps pour maximiser le taux d'occupation des ressources, tout en réduisant au maximum le temps passé entre

sa soumission et son exécution. Les principaux ordonnanceur sont listés dans le tableau 2 ci-dessous.

RJMS		Type	Dernière Version
Slurm	(SchedMD)	Open-Source	19.05
OAR		Open-Source	2.5.8
Spectrum LSF	(IBM)	Commercial	10.1.0
Torque/MOAB	(AdaptiveComputing)	Commercial	6.1.2
Univa Grid Engine (formely SGE/OGE)	(Univa)	Commercial	8.6.7

TABLE 2 : Liste des principaux ordonnanceurs/RJMS HPC.

Slurm, pour *Simpl Linux Utility Resources Manager*, est un ordonnanceur libre et open-source pour Linux très utilisé dans le monde du HPC, en particulier sur la majorité des systèmes référencés sur la liste Top500[5] des 500 super-calculateurs les plus puissants au monde. *Slurm* est notamment utilisé par l'ULHPC comme ordonnanceur de référence sur ses clusters (en particulier `iris`) depuis 2017 (OAR était utilisé pour les clusters précédents)

Slurm utilise trois démons différents pour fonctionner. `slurmctld` est le démon central, qui orchestre l'ensemble des autres démons *Slurm* et les ressources. Ce premier démon communique avec le démon `slurmdbd`, qui gère la base de données *Slurm*. Enfin, sur chaque nœud du cluster tourne un démon `slurmd`. Ces multiples démons `slurmd` communiques de façon hiérarchique de sorte à ce que le sommet communique avec le démon `slurmctld`. Enfin, les commandes utilisateurs de *Slurm* s'adressent tantôt au démon `slurmd` du nœud, tantôt au démon contrôleur principal `slurmctld`. Parmi ces commandes, on peut trouver :

SCONTROL pour administrer la configuration ou l'état de *Slurm* ;

SINFO pour obtenir des informations sur l'état des nœuds du cluster ;

SQUEUE pour obtenir des informations sur les jobs en attentes ;

SACCT pour obtenir des informations sur les jobs passés ;

SRUN / SBATCH pour soumettre des jobs.

La figure 1 présente cette architecture.

Pour ordonnancer les jobs, *Slurm* se base sur les demandes d'allocations de ressources (nombre de cpus, nombre de nœuds quantité de mémoire, ...) formulées par les utilisateurs. La figure 2 présente les statistiques d'utilisation du cluster `iris` de l'Université depuis son ouverture, d'abord d'un point de vue global (en haut), puis en se concentrant sur la partition principale et la qos associée (appelée `qos-batch`).

Cependant, les allocations ne représentent pas nécessairement la consommation réelle en ressources du job, et des ressources sont alors « perdues » en étant pas utilisées de façon optimale. On aimerait pouvoir prévoir la consommation réelle en ressources d'un job lors de sa soumission, avant son exécution, pour pouvoir améliorer la répartition des ressources, et l'ordonnancement.

5 https ://www.top500.org/

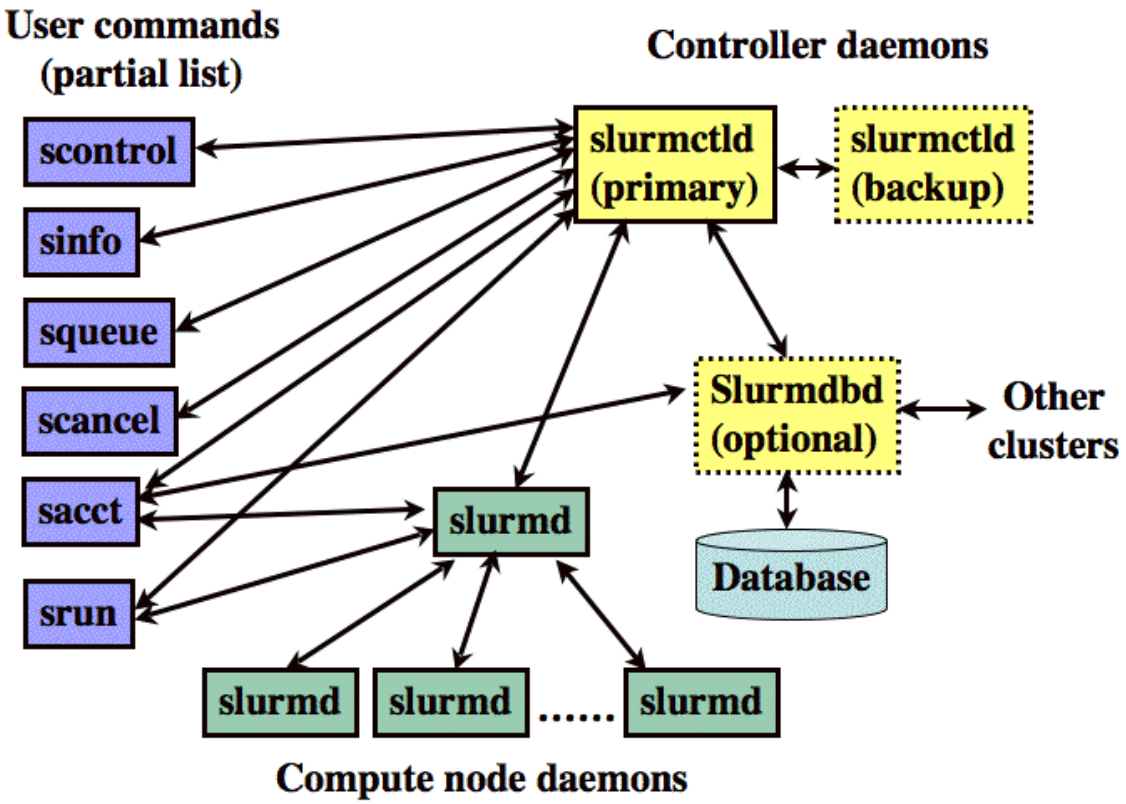


FIGURE 1 : Architecture de Slurm

1.3 REFORMULATION DU PROBLÈME

Pour réussir à prévoir la consommation d'un job avant son exécution, on suppose une approche à base de classification automatique. L'idée est de proposer une série de classes correspondant à des profils types de jobs, entraîner un modèle statistique à classifier les jobs passé dans ces classes, et utiliser ce modèle entraîné pour fournir une prévision à l'ordonnanceur (*Slurm* dans ce cas) qui pourra adapter les allocations de ressources à partir de cette prédiction. L'objectif à long terme est de coupler cet apprentissage et ces définitions de classes à une optimisation de la configuration des ressources améliorant l'efficacité énergétique de la plateforme de calcul.

Afin de réaliser cela, je me suis posé les questions suivantes, auxquelles j'ai tentées d'apporter des réponses durant ce stages :

1. Comment est utilisé le centre HPC de l'Université du Luxembourg ?
2. Quels sont les profils types de jobs soumis ?
3. Comment déterminer automatiquement le profil d'un job à sa soumission ?
4. Enfin, comment améliorer les informations collectées par l'ordonnanceur (en particulier sur le type de logiciel utilisés au sein d'un job) ?

Durant mon stage, j'ai tâché d'apporter des pistes de réponses à ces problèmes en travaillant sur une copie de la base de données de l'instance de *Slurm* du cluster `Iris` de l'ULHPC. Cette copie contenait tous les jobs ayant été alloués sur le cluster entre le 4 mai 2017 et le 13 mai 2019, à savoir 368142 jobs.

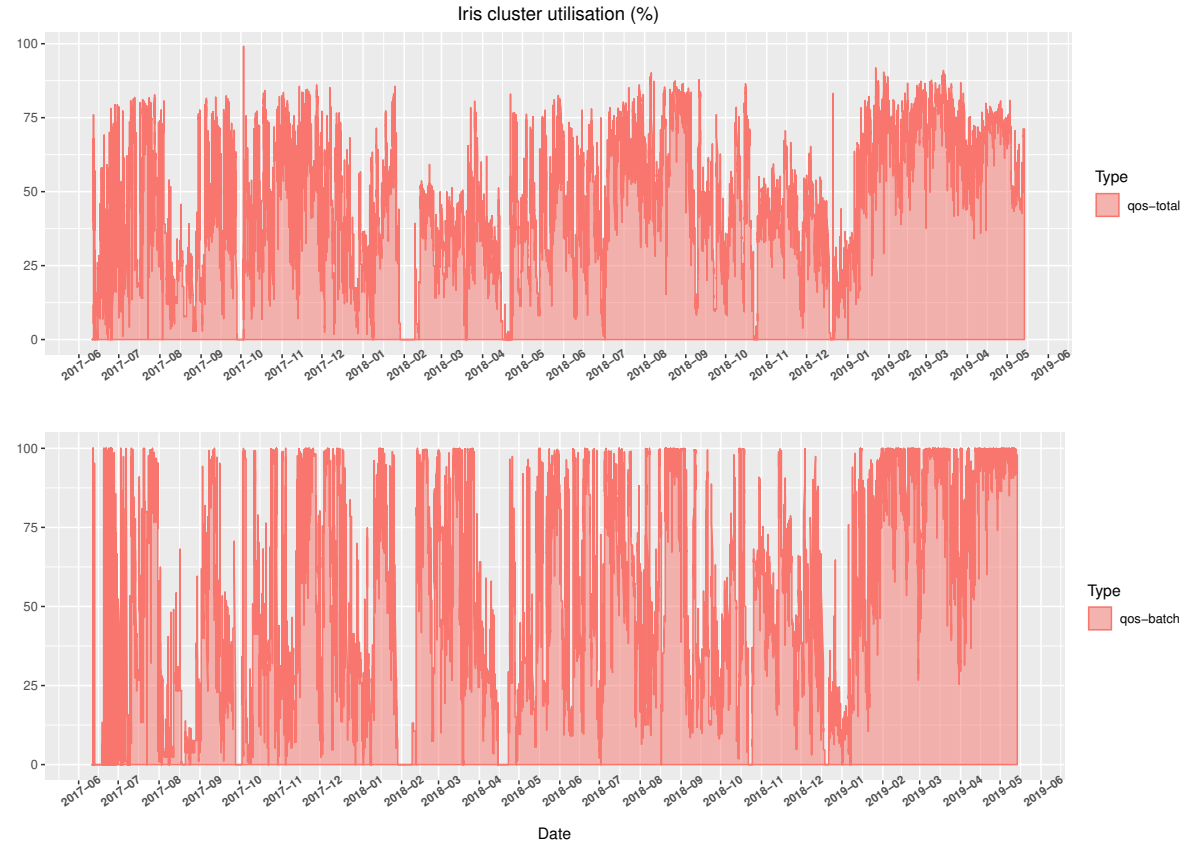


FIGURE 2 : Statistiques d'utilisation Slurm du cluster `iris` depuis son ouverture

A noter que ces travaux sont inspirés d'une première étude de classification de jobs (opérés à l'époque sur des traces du cluster `gaia`) proposés par J. Emeras et al., *Evalix : Classification and Prediction of JobResource Consumption on HPC Platforms*, 2015 [1].

1.4 ORGANISATION DU MANUSCRIT

Ce rapport est organisé de la façon suivante : ce premier chapitre introduit le sujet, en exposant les objectifs du stage, ainsi que le contexte et les solutions existantes. Le chapitre suivant détaillera les démarches que j'ai entreprises afin de réaliser les objectifs du stage et leurs implémentations. Le chapitre 3 se concentrera sur les résultats obtenus avec les implémentations décrites. Enfin, le dernièr chapitre 4 concluera ce rapport et proposera des ouvertures et poursuites du projet.

2 APPROCHE ET EXPÉRIMENTATIONS

Ce chapitre présente l'ensemble des démarches que j'ai menées avant d'obtenir des résultats qui seront montrés plus loin. Ces démarches comprennent la mise en place de l'environnement de tests et le développement d'outils d'analyse.

2.1 RÉPLICATION DU CLUSTER DE PRODUCTION `iris` PAR VIRTUALISATION

Pour mener mes expérimentations, j'ai eu besoin de disposer d'une instance de *Slurm* opérationnelle et plus généralement d'une réplication virtuelle de l'environnement de production qu'est le cluster `iris` afin de mener à bien mes tests sans impact sur les nombreux chercheurs utilisant la plateforme ULHPC. Il était en effet exclus d'utiliser le cluster `iris` de l'Université du Luxembourg pour faire mes expérimentationsafin de ne pas perturber son fonctionnement optimal. C'est pourquoi j'ai dû émuler un cluster virtuel, sur lequel j'ai fait tourner une instance de *Slurm*. La figure 3 donne un apercu du cluster `iris`. Bien entendu, il n'était pas nécéssaire de répliquer l'intégralité des éléments matériels qui composent le cluster pour mener mes expériences. Il était suffisant de se concentrer sur les composants liés à l'ordonnanceur *Slurm*.

Pour cela, je suis parti d'un précédent projet de l'université[1] qui m'a fourni un module *Puppet*[2] pour déployer *Slurm* sur un cluster. Ce projet proposait également une configuration pour générer un cluster virtuel avec *Vagrant*.

Vagrant[3] est un outil libre et Open-Source pour construire et gérer des environnements de développement qui soient portables et reproductibles. Ces environnements de développement gérés par *Vagrant* tournent sur des plateformes virtuelles locales, comme *VirtualBox*[4] ou VMWare[5]. *Vagrant* fournit une structure et un format de configuration pour provisionner l'environnement de manière automatique.

Le projet `puppet-slurm` de l'ULHPC proposait donc un ensemble de scripts de provisionnements pour *Vagrant* permettant de déployer *Slurm* (selon une configuration basique à titre d'exemple) sur un cluster virtuel de quatre machines telles que suit :

- `slurm-master`, la machine qui a le rôle de controlleur pour *Slurm*, sur laquelle tournent les démons `slurmdbd` et `slurmctld`, et qui héberge la base de données de *Slurm* ;
- `access`, la machine qui joue le rôle de nœud de login, à partir de laquelle les utilisateurs se connectent, y tourne un démon `slurmd` ;

1 Voir https://github.com/ULHPC/puppet-slurm
2 Puppet est un logiciel libre permettant la gestion de configuration à partir d'une machine maîtresse à des machines esclaves.
3 Voir https://www.vagrantup.com/
4 Voir https://www.virtualbox.org/
5 Voir https://www.vmware.com/

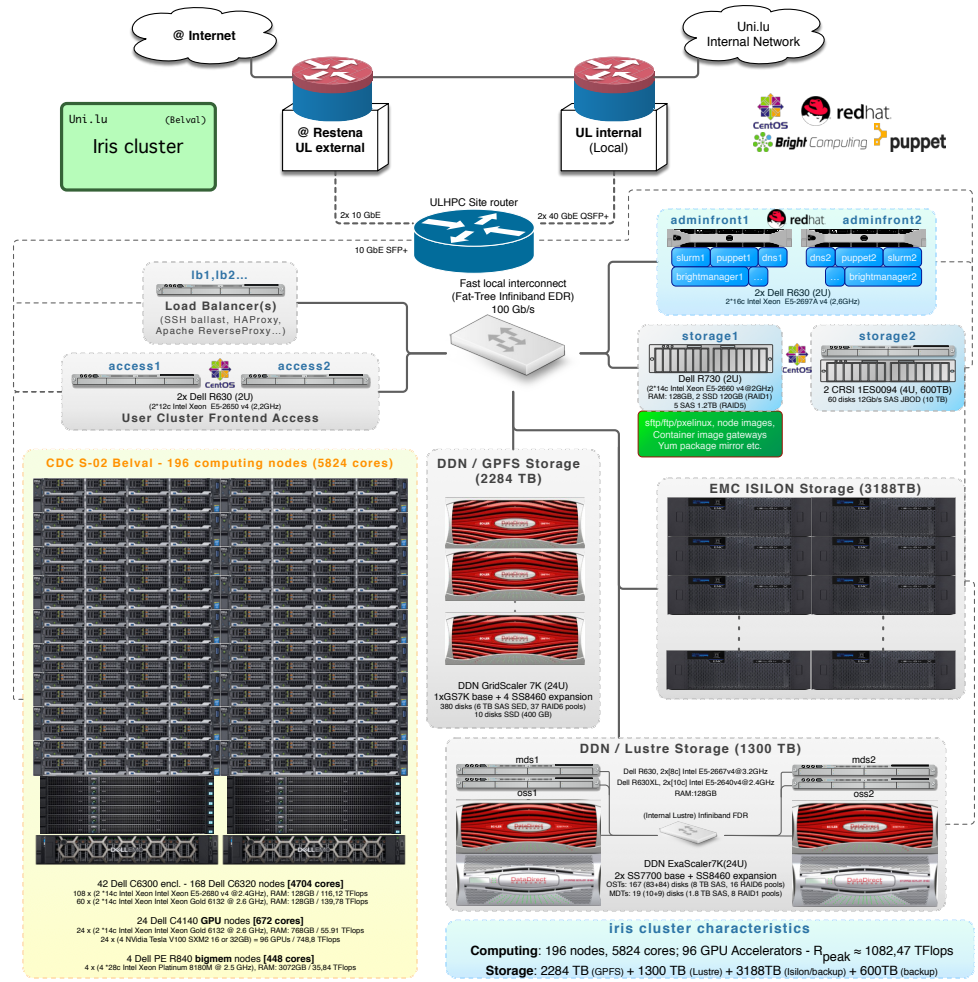


FIGURE 3 : Organisation du cluster `iris` (Source : https://hpc.uni.lu/systems/iris/)

- `node-1` et `node-2`, deux nœuds de calculs sur lesquels tournent un démon `slurmd`, et liés par défaut aux partitions `interactive` et `batch` (respectivements) de la configuration *Slurm*.

La figure 4 illustre l'architecture de ce cluster virtuel, et les connexions « licites » pour un utilisateur qui y existent. En réalité, depuis ma machine hôte (appelée `local` sur le schéma), je pouvais me connecter à tous les nœuds avec `ssh`. Et de la même façon, depuis l'intérieur du cluster, je pouvais me connecter à toutes les autres machines avec `ssh`. Cela n'est pas montré sur le schéma par souci de clarté.

Une fois cet environnement en place, j'ai souhaité y importer la base de donnée du cluster `iris` des deux dernières années. Cela se fait en quatre étapes :

ÉTEINDRE LE DÉMON SLURMDBD

Sur la machine `slurm-master` :

```
systemctl stop slurmdbd.service
```

MODIFIER LA CONFIGURATION

Il faut ensuite modifier le fichier de configuration `/etc/slurm/slurm.conf` de sorte à changer le nom du cluster pour « iris », et ajouter les définitions de nœuds, de partitions,

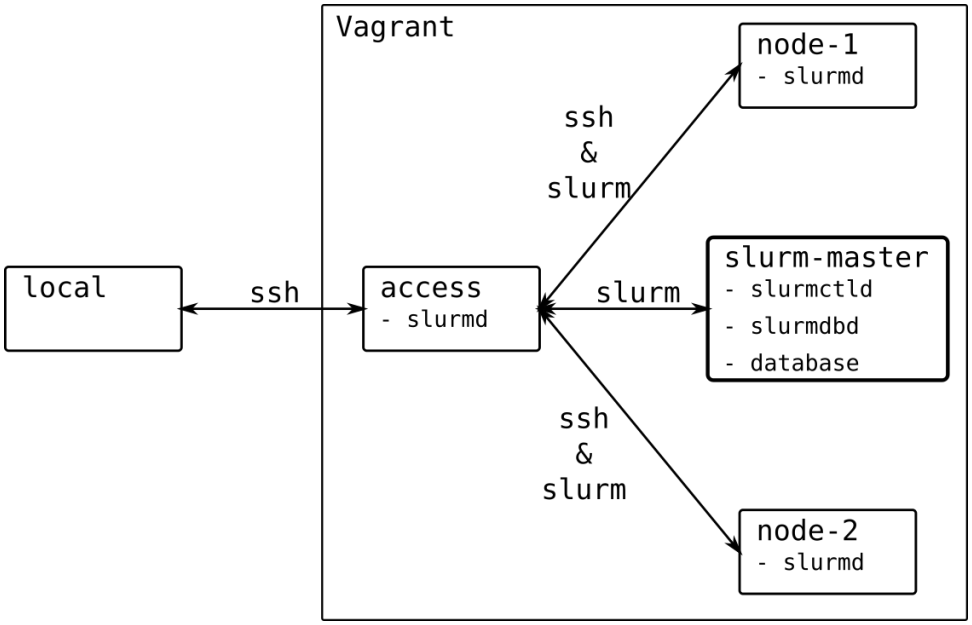


FIGURE 4 : Configuration *Vagrant* déployées pour répliquer l'environnement *Slurm* du cluster `iris`.

et de qos de la configuration du cluster `iris`. Le but est que la configuration de *Slurm* corresponde au contenu de la base de donnée. La structure réelle du cluster n'a pas tant d'importance pour cela, *Slurm* se contentant de considérer comme indisponible les nœuds renseignés dans la configuration mais qui n'existent pas.

IMPORTER LA BASE DE DONNÉE

On crée un dump de la base originelle avec la commande

```
mysqldump Slurm >  slurm_dump.sql
```

Sur la machine `slurm-master`, dans le prompt mysql, on réinitialise la base de données slurm :

```
(mysql)> drop database slurm; create database slurm;
```

Puis, à nouveau dans le shell, on rempli la base de donnée :

```
mysql slurm < slurm_dump.sql
```

REDÉMARRER LE DÉMON SLURMDBD

Sur la machine `slurm-master` :

```
systemctl start slurmdbd.service
```

Une fois cela fait, on a un cluster virtuel avec une instance de *Slurm* se reportant à la base de données des deux dernières années du cluster `iris` de l'Université du Luxembourg.

2.2 ANALYSE DE LA BASE DE DONNÉES slurm

À partir de l'environnement mis en place tel que décrit précédemment, je pouvais utiliser les commandes *Slurm*, et notamment la commande `sacct` pour interroger la base de donnée.

La commande `sacct` est une commande *Slurm* qui permet d'obtenir les données des jobs enregistrés dans la base *Slurm*. Contrairement à la commande `sstat` qui affiche diverses informations concernant un job en cours d'excécution, `sacct` ne prend en compte que les jobs terminés.

Dans une première approche pour comprendre l'utilisation des ressources par les jobs des utilisateurs, j'ai développé un script en Python3 autour de la commande `sacct`. Ce script automatise le tracé d'histogrammes représentant le nombre de jobs pour différentes valeurs d'une métrique. Ce script est pleinement documenté, suit les standards de style des PEP[6]

La commande `sacct` permet de sélectionner des jobs en fonction de divers critères (la date de début ou de fin, l'utilisateur propriétaire du job, le numéro du job, les qos, les partitions, etc), et offre 108 champs qui peuvent être spécifiés par l'utilisateur, fournissant divers informations pour l'ensemble des jobs sélectionnés. Le listing 2.1 contient la liste de ces champs.

6 PEP, pour Python Enhancement Proposals, est une sorte de RFC pour le langage Python. Les propositions de nouvelles fonctionnalités majeures passent par là, et la PEP8 est dédiée à définir un standard de style de code Python.

```
$ sacct --helpformat

Fields available:

Account             AdminComment        AllocCPUS           AllocGRES
AllocNodes          AllocTRES           AssocID             AveCPU
AveCPUFreq          AveDiskRead         AveDiskWrite        AvePages
AveRSS              AveVMSize           BlockID             Cluster
Comment             Constraints         ConsumedEnergy      ConsumedEnergyRaw
CPUTime             CPUTimeRAW          DerivedExitCode     Elapsed
ElapsedRaw          Eligible            End                 ExitCode
Flags               GID                 Group               JobID
JobIDRaw            JobName             Layout              MaxDiskRead
MaxDiskReadNode     MaxDiskReadTask     MaxDiskWrite        MaxDiskWriteNode
MaxDiskWriteTask    MaxPages            MaxPagesNode        MaxPagesTask
MaxRSS              MaxRSSNode          MaxRSSTask          MaxVMSize
MaxVMSizeNode       MaxVMSizeTask       McsLabel            MinCPU
MinCPUNode          MinCPUTask          NCPUS               NNodes
NodeList            NTasks              Priority            Partition
QOS                 QOSRAW              Reason              ReqCPUFreq
ReqCPUFreqMin       ReqCPUFreqMax       ReqCPUFreqGov       ReqCPUS
ReqGRES             ReqMem              ReqNodes            ReqTRES
Reservation         ReservationId       Reserved            ResvCPU
ResvCPURAW          Start               State               Submit
Suspended           SystemCPU           SystemComment       Timelimit
TimelimitRaw        TotalCPU            TRESUsageInAve      TRESUsageInMax
TRESUsageInMaxNode  TRESUsageInMaxTask  TRESUsageInMin      TRESUsageInMinNode
TRESUsageInMinTask  TRESUsageInTot      TRESUsageOutAve     TRESUsageOutMax
TRESUsageOutMaxNode TRESUsageOutMaxTask TRESUsageOutMin     TRESUsageOutMinNode
TRESUsageOutMinTask TRESUsageOutTot     UID                 User
UserCPU             WCKey               WCKeyID             WorkDir
```

Listing 2.1 : Liste des champs disponibles pour `sacct`

Parmi tous ces champs, j'en ai selectionné 8 pour leur pertinence par rapport à l'analyse que je souhaitais faire. Ces 8 champs correspondent chacun à une ressource consommée par le jobs. Ces champs sont les suivants :

- `AveCPU`, temps CPU moyen total utilisé ;
- `AveRSS`, la mémoire moyenne utilisée par toutes les tâches du job ;
- `MaxRSS`, la mémoire maximale utilisée par toutes les tâches du job ;
- `AveDiskRead`, le nombre moyen d'octets lus par toutes les tâches du job ;
- `AveDiskWrite`, le nombre moyen d'octets écrits par toutes les tâches du job ;
- `ConsumedEnergyRaw`, l'énergie totale consommée par toutes les tâches du job ;
- `AllocCPUS`, nombre de CPUs alloués au job ;
- `Elapsed`, le temps d'exécution du job.

Ces 8 métriques sont disponibles dans le script, et sont collectées (lorsque demandées) avec la commande sacct invoquée ainsi :

```
sacct {-u <user>|-a} -P -s CD -S {<starttime>|010170} [-E <endtime>] \
      --format jobid,Elapsed,{metric,...}
```

On voit que la commande `sacct` est appelée avec différents paramètres, qui sont :

- `-u` pour spécifier un utilisateur, sinon `-a` pour tous les utilisateurs ;
- `-P` pour obtenir une sortie au format csv avec le symbole « | » comme séparateur ;
- `-S` pour donner une date de début, le temps Epoch par défaut dans le script ;
- `-E` pour donner une date de fin, 'now' par défaut ;
- `-s CD` qui spécifie l'état de sortie des jobs, ici on garde les job à l'état COMPLETED (tous les process ont terminé avec un code de retour à 0) ;
- `--format` pour donner la liste des champs voulus.

On remarque que le jobID et le temps d'exécution du job sont toujours demandés à `sacct`. Le temps d'exécution est demandé pour pouvoir filtrer les jobs trop courts (par défaut moins de 10 secondes, cela est paramétrable en option) qui correspondent très probablement à des erreurs d'entrées des utilisateurs, et ne sont donc pas représentatifs de l'utilisation réelle du cluster. Le jobID quant à lui est nécessaire pour regrouper les données des étapes d'un job en une seule entrée.

Par ailleurs, pour plusieurs métriques, il n'est pas intéressant de s'intéresser à sa valeur brute, mais plutôt à son rapport sur le nombre de CPU ou le temps du job. Ainsi on s'intéresse plus à la consommation CPU moyenne par nombre de cœurs qu'à la consommation CPU moyenne totale. De la même manière, on s'intéresse à la vitesse d'écriture ou de lecture sur le disque (la quantité d'écriture ou de lecture par rapport au temps du job), plutôt qu'à la quantité. C'est pourquoi j'ai implémenté une fonctionnalité pour pouvoir demander à tracer le résultat de calculs sur les données, avec l'option `--compute`. Par exemple, pour obtenir la répartition en nombre de jobs de de la consommation mémoire moyenne par cœur, pour tous les jobs de la base, la commande est la suivante :

```
python spygraph.py AveRSS AllocCPUS --compute div 1,2
```

L'implémentation supporte en fait n'importe quelle succession d'addition(s), soustraction(s), multiplication(s) et division(s), et en déduit le titre du graphe et les unités par le même procédé que pour effectuer le calcul.

Enfin le script dispose de divers autres options pour régler autant de paramètres, qui sont détaillées dans le listing 1 que l'on trouvera en annexes.

Les graphes obtenus seront exposés et discutés dans le chapitre 3.

2.3 PARTITIONNEMENT AUTOMATIQUE

L'approche décrite précédemment est intéressante car cela m'a permis de mieux comprendre l'utilisation générale du cluster `iris`. Cependant, elle ne me permettait pas de trier les jobs par types. C'est pourquoi dans une seconde phase de mon stage, j'ai utilisé un algorithme de

partitionnement automatique pour déterminer des groupes de jobs partageant des qualités communes.

Il existe de multiples méthodes de partitionnement des données (*clustering* en anglais), qui dépendent également du *modèle* de cluster utilisés. Parmi les plus courantes, on peut citer :

- Les méthodes basées centroïdes telles que les algorithmes de type *K-mean* ;
- Les méthodes de regroupement hiérarchique se basant sur ces modèles de connectivité ;
- Des algorithmes de maximisation de l'espérance (EM) ;
- Des algorithmes basés sur un modèle de densité tels que DBSCAN ou OPTICS.

Nous verrons par la suite l'algorithme retenu. Dans tous les cas, j'ai travaillé en un premier temps sur un échantillon de 10K (10000) jobs pour tester les choix de variables et de paramètres, puis sur un échantillon de 100K jobs pour fournir des résultats expérimentaux. Ces résultats seront détaillés dans la section 3.

2.3.1 Choix des dimensions dans le partitionnement des jobs

Pour mener une analyse statistique, la première chose à faire est de déterminer l'ensemble des variables à utiliser.

Pour cela j'ai commencé par extraire de la base de données une dizaine de valeurs par job. Le critère de sélection de ces valeurs a été de savoir si elles étaient représentative du job ou de de sa consommation en ressources, et numériques (pour pouvoir exécuter des calculs dessus). Par ailleurs, certaines de ces métriques ont été par la suite abandonnées d'après l'observation des premiers résultats. Ainsi, les métriques gardée originellement ont été les suivantes[7] :

- le jobID ;
- l'état de sortie du job ;
- le code de retour du job ;
- le temps d'exécution du job ;
- le nombre de nœuds alloués ;
- le nombre de CPUs alloués ;
- le temps CPU moyen total du job ;
- la consommation mémoire moyenne totale du job ;
- la consommation mémoire maximale totale du job ;
- le nombre moyen d'octets lus par toutes les tâches du job ;
- le nombre moyen d'octets écrits par toutes les tâches du job ;
- l'énergie consommée.

7 La mention de « total » signifie qu'il existe une mesure pour chaque tâche du job, la métrique résultante pour le job est la somme de ces mesures.

La première étape du partitionnement a ensuite été de pratiquer l'*analyse en composantes principales* (appelée *Principal Component Analysis (PCA)*) du jeu de données. Cette analyse permet de montrer la part que prend chaque variable dans la variance de l'ensemble des données. Le but de cette analyse est de trouver les variables statistiques peu influentes (qui expliquent trop peu la variance, car elles sont liées à d'autres) que l'on peut retirer des données afin des les alléger tout en conservant leurs informations.

La figure 5 représente le tracé de la PCA pour un échantillon des 10K premiers jobs de la base de donnée avec les 12 métriques mentionnées ci-dessus. On observe une courbe arrondie, où l'importance des variables est de plus en plus faible. Les quatre dernières variables expliquent ensemble seulement 10% de la variance, alors que les trois premières en expliquent 50%.

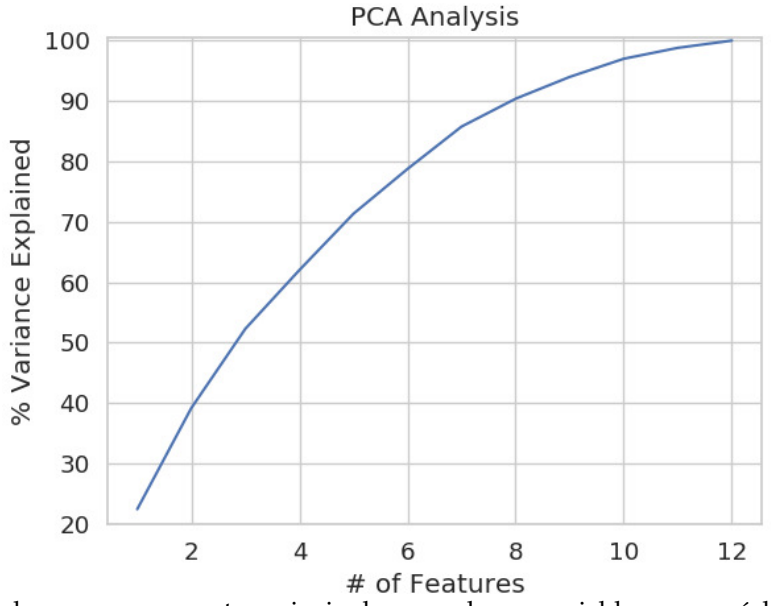


FIGURE 5 : Analyse en composantes principales pour les 12 variables sur un échantillon de 10K jobs

La PCA de l'échantillon de données pour 12 variables a révélé un mauvais choix des variables. En effet la sélection initiale insinuait naïvement qu'un maximum de variables expliquerait au mieux les données. Ce n'est cependant pas le cas, un choix de variables réduits mais judicieux est bien plus efficace. Aussi, j'ai retiré des 12 variables :

- le jobID, qui ne donnait pas d'informations sur la qualité du job ;
- l'état de sortie, le code de retour et le nombre de nœuds alloués, qui étaient sur un ensemble discret
- la consommation mémoire maximale totale du job, qui était proche en signification de la consommation mémoire moyenne totale du job.

Ainsi, les 7 variables restantes sont :

- le temps d'exécution du job ;
- le nombre de CPUs alloués ;
- le temps CPU moyen total du job ;
- la consommation mémoire moyenne totale du job ;
- le nombre moyen d'octets lus par toutes les tâches du job ;

- le nombre moyen d'octets écrits par toutes les tâches du job ;
- l'énergie consommée.

On remarque que le nombre de CPUs alloués a été conservé, malgré sa qualité discrète, car j'ai estimé qu'il s'agissait une information significative sur le job. Le graphe de la figure 6 représente le tracé de la nouvelle PCA pour le même échantillon de 10K jobs que précédemment, mais avec ces 7 variables. On constate une courbe qui semble presque affine pour les six premières variables. Chacune des six premières variables explique entre 10 et 15% de la variance des données (exceptée pour la première, le temps d'exécution, qui explique à lui seul près de 35% de la variance des données). La septième et dernière variable, l'énergie consommée, a une part très faible, moins de 5%. En effet, cela est cohérent, la consommation d'énergie est directement liée à la consommation d'autres ressources, cette variable est corrélée à d'autres.

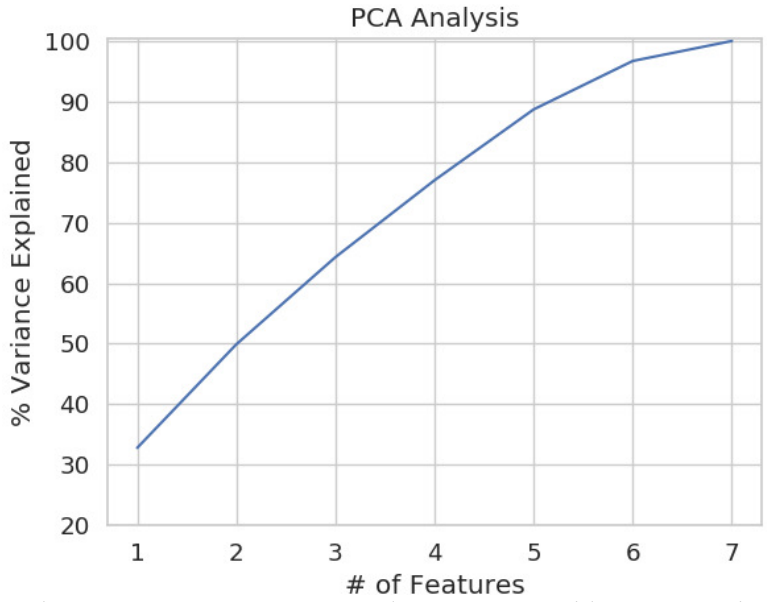


FIGURE 6 : Analyse en composantes principales pour 7 variables sur un échantillon de 10K jobs

La figure 7 représente la même PCA mais pour un échantillon plus représentatif de 100K jobs. On y observe la confirmation de la tendance du graphe précédent, les variables ont une part environ égale dans l'explication de la variance, et la consommation d'énergie est toujours très corrélée aux autres variables. Cette confirmation conforte le choix des 7 variables énoncées.

2.3.2 Détermination des paramètres de l'algorithme de partitionnement

L'algorithme de partitionnement sélectionné pour le problème est l'algorithme *DBSCAN* [2], pour *Density-Based Spatial Clustering of Applications with Noise*. Cet algorithme classe un ensemble de point selon des zones de densité, marquant comme bruit les points solitaires. La densité minimale est déterminée par les paramètres. L'intérêt de cet algorithme est qu'il n'a pas besoin de connaître le nombre de classes à former.

On trouvera en annexe, au listing 2, le détail de l'algorithme.
Pour déterminer la densité minimale, l'algorithme prend deux paramètres :

ε : la distance maximale entre deux points pour les considérer comme voisins

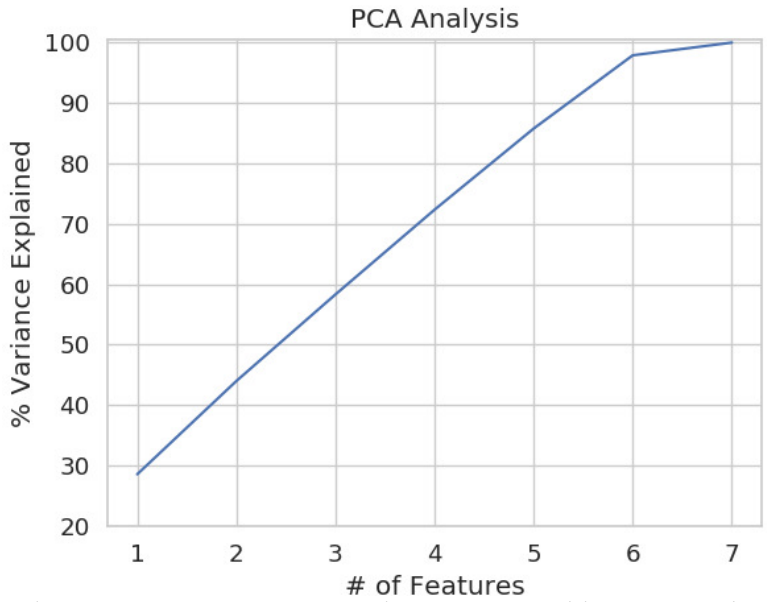


FIGURE 7 : Analyse en composantes principales pour 7 variables sur un échantillon de 100K jobs

MinPts : le nombre de points minimum de points à trouver dans l'ε-voisinage d'un point pour le considérer comme un point d'extension du groupe.

Finalement, on mesurera la qualité du partitionnement avec le *coefficient de silhouette*. Pour chaque point de l'ensemble des données, son coefficient de silhouette est la différence entre sa distance moyenne aux autres points des groupes voisins (séparation, ou distance moyenne extra-groupe b) et sa distance moyenne aux autres points de son groupe (cohésion, ou distance moyenne intra-groupe a), divisée par le maximum de ces deux distances, soit $(b-a)/max(a,b)$ [8]. Le coefficient de silhouette de l'ensemble du partitionnement est la moyenne du coefficient de silhouette de tous les points.

Ce coefficient de silhouette varie entre -1 et 1. Un coefficient de silhouette proche de -1 représente un mauvais partitionnement, une valeur proche de 0 montre un partitionnement dont les groupes se chevauchent, et une valeur vers 1 correspond à un partitionnement optimal, où les groupes sont bien distincts.

Afin d'obtenir un coefficient de silhouette intéressant, il est nécessaire de choisir au mieux les deux paramètres de l'algorithme. Pour cela, une heuristique par paramètre a été utilisée.

ε est choisi de telle sorte que 95% des points aient une distance à leur plus proche voisin inférieure à ε.

Pour MinPts, la méthode est plus compliquée. Une règle générique est de prendre MinPts comme étant le double de la dimension des données. Cependant, pour 12 ou 7 dimensions, cette valeur est faible, et n'est pas adaptée à la taille de l'échantillon. En effet, le jeu de données étant volumineux, une faible valeur de MinPts formerait de nombreux petits groupes peu significatifs. De plus, une valeur de MinPts trop petite pourra faire des groupes autour de valeurs aberrantes, ou de doublons de données. Une autre règle est de choisir MinPts tel que 95% des points aient plus de MinPts dans leur ε-voisinage. Cette règle est plus souple, mais fourni toujours une valeur de MinPts trop faible pour l'ensemble des données.

Ainsi la méthode choisie pour déterminer MinPts est une adaptation de cette seconde règle. Au lieu de travailler avec l'ε-voisinage, on travaille avec les $(n*\varepsilon)$-voisinages (avec n entier). Pour chaque valeur de n, on choisit MinPts tel que 95% des points aient plus de MinPts

8 voir la documentation de la bibliothèque scikit-learn

dans leur $(n * \varepsilon)$-voisinage, puis on calcule le coefficient de silhouette du partitionnement obtenu avec ε et cette valeur de MinPts. On choisit alors la valeur de MinPts pour laquelle le coefficient de silhouette est le plus haut.

Les figures 8 et 9 représentent le tracé des points (MinPts, coefficient de silhouette obtenu), pour le même échantillon de 10K jobs, avec, respectivement, les 12 métriques initiales, et les 7 métriques finalement conservées.

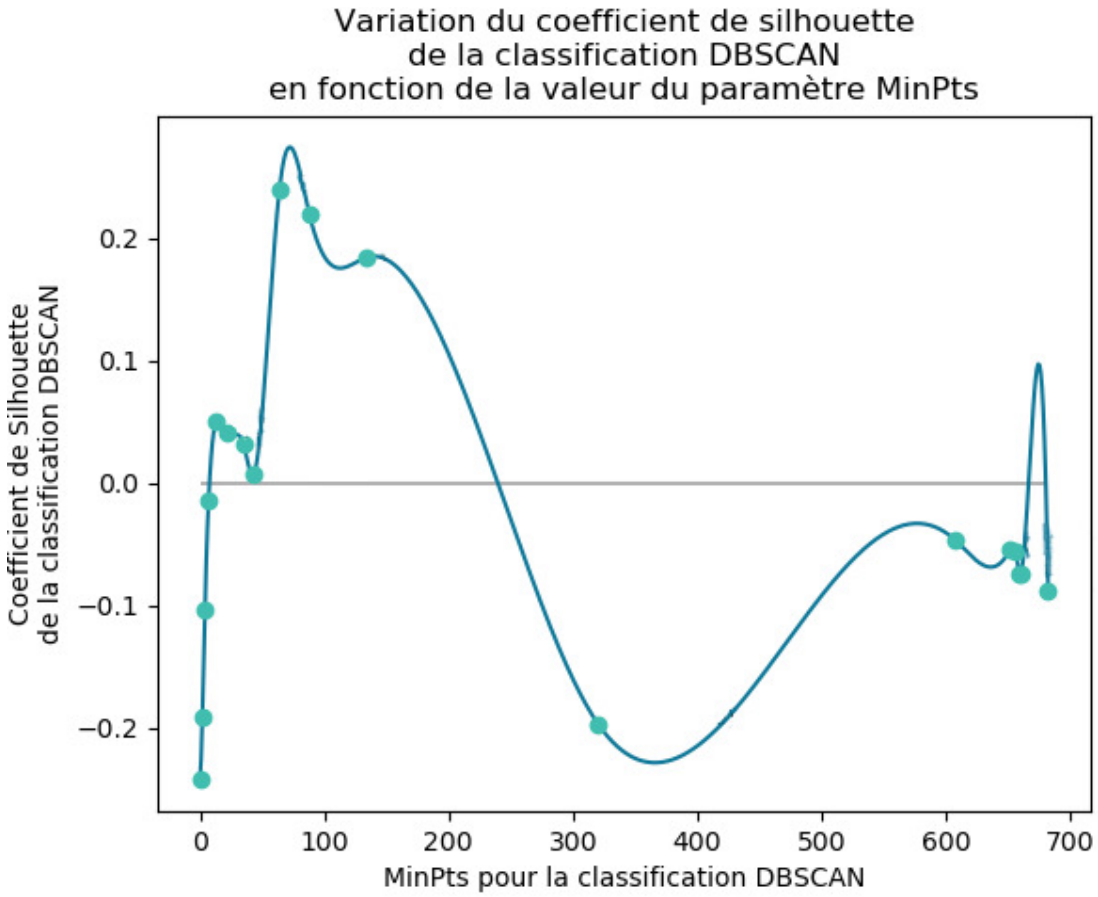


FIGURE 8 : Extrapolation de la variation du coefficient de silhouette du partitionnement DBSCAN en fonction de la valeur du paramètre MinPts, pour ε fixé. Échantillon de 10K jobs, 12 dimensions. Silhouette maximale de 0.24 calculée pour MinPts à 64.

La comparaison entre la meilleure silhouette obtenue à partir de l'échantillon de 10K jobs, avec 12 ou 7 variables., 0.24 contre 0.45, conforte bien la pertinence de la réduction du nombre de variables.

L'approche décrite ci-dessus permet d'obtenir une valeur du paramètre MinPts satisfaisante, mais à grands coûts. En effet, pour chaque itération n, on doit parcourir toutes les données pour calculer le $(n * \varepsilon)$-voisinage de chaque point. Avec un échantillon de 10K jobs, cela tournait sans problème en quelques dizaines de secondes sur un ordinateur de bureau, mais en passant à 100K jobs, le goulot d'étranglement est devenu critique.

J'ai donc procédé à paralléliser ce code avec la librairie Python *ipyparallel*[9], de sorte à lancer les calculs des n $(n * \varepsilon)$-voisinages en même temps, dans le but de le faire tourner sur le cluster `iris`. On trouvera en annexe C le détail de cette parallélisation.

Cela m'a permis d'obtenir une valeur de MinPts optimale pour le partitionnement des 100K jobs, dont les résultats seront exposés dans le chapitre **??**.

9 Voir https://ipyparallel.readthedocs.io/

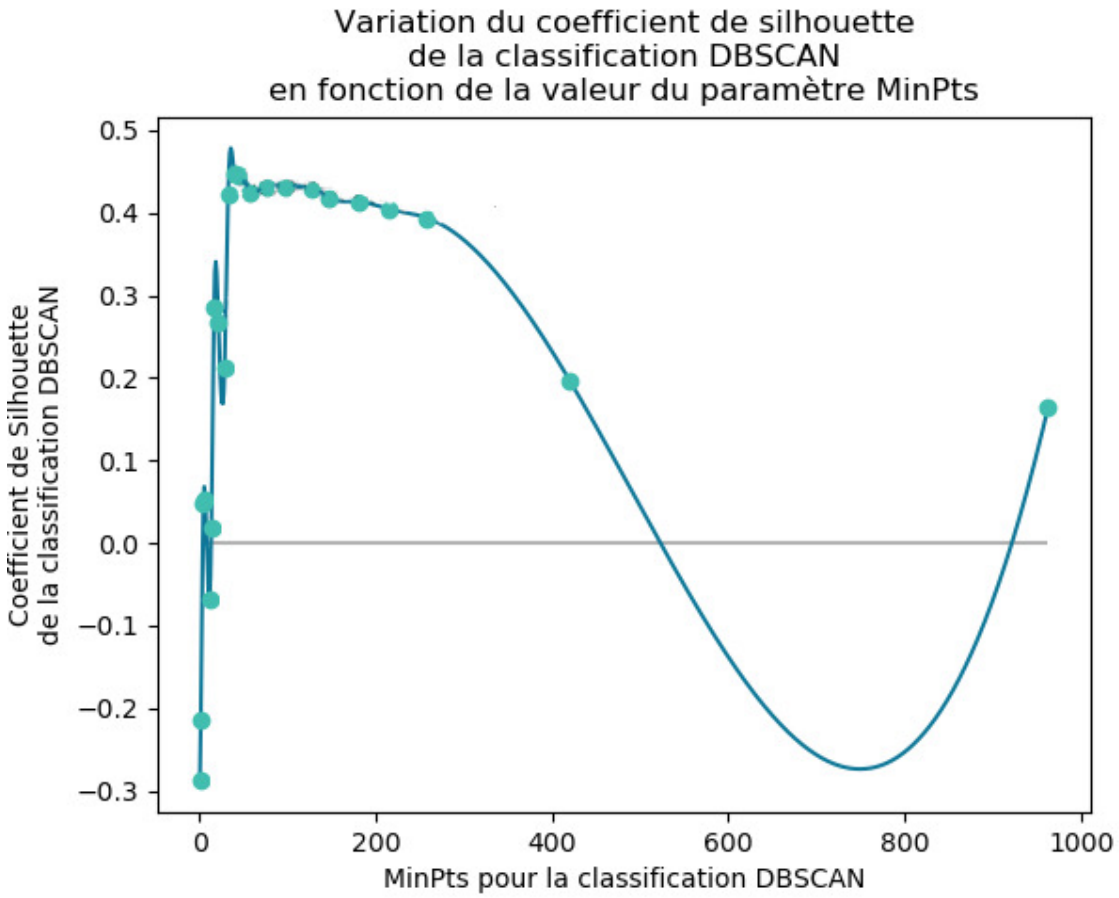


FIGURE 9 : Extrapolation de la variation du coefficient de silhouette du partitionnement DBSCAN en fonction de la valeur du paramètre MinPts, pour ε fixé. Échantillon de 10K jobs, 7 dimensions. Silhouette maximale de 0.45 calculée pour MinPts à 40.

2.4 RAPPORT D'UTILISATION DES MODULES LOGICIELS CHARGÉS AU SEIN DES JOBS

Les données collectées par *Slurm* et dont on a extrait les informations détaillées à la sections 3 par les moyens décrits plus haut, concernent surtout les mesures de ressources systèmes utilisées par le job lors de son exécution. Comme on a pu le constater, ce jeu de données permet déjà d'obtenir un bon modèle de partitionnement, néanmoins dans le contexte du projet ENERGUMEN, il apparaît nécessaire d'affiner le modèle de partitionnement par le biais d'informations complémentaires sur le contexte des jobs. Or, dans un souci d'optimisation, d'efficacité des démons *Slurm* mais aussi de protection de la vie privées, de nombreuses informations utiles pouvant être extraites de l'*environnement* d'un job ne sont pas aujourd'hui enregistré dans la base de données *Slurm*.

L'*environnement* d'un job contient, entre autre chose, l'ensemble des librairies chargées par le jobs, ce qui peut être un très bon indicateur du type de travaux effectués. En effet, une plateforme HPC accueille de nombreux utilisateurs ayant des besoins hétérogènes, et donc autant d'environnements différents, et souvent incompatibles. Afin de faciliter la gestion de ces environnements dans un contexte HPC, il existe le système de référence des *modules d'environnements*[10], qui proposent aux utilisateurs de charger dynamiquement des environnements

10 *Environment modules* en anglais. Voir https://modules.readthedocs.io/

logiciels définis, à volonté, et qui ont été compilés et optimisés pour les resources de calcul qui composent l'infrastructure de calcul. Un exemple typique d'utilisation est alors le suivant :

```
# Recherche des environnements de compilation existants sur le cluster. Ex:
# - FOSS: Free and Open Source Software
# - Intel: Intel Parallel Studio XE suit
$> module avail toolchain
------- /opt/apps/resif/data/stable/default/modules/all -------
 toolchain/foss/2019a                          toolchain/iccifortcuda/2019a
 toolchain/fosscuda/2019a                      toolchain/iimpi/2019a
 toolchain/gcccuda/2019a                       toolchain/iimpic/2019a
 toolchain/gompi/2019a                         toolchain/intel/2019a
 toolchain/gompic/2019a                        toolchain/intelcuda/2019a
 toolchain/iccifort/2019.1.144-GCC-8.2.0-2.31.1
$> which icc
icc not found
# Chargement de la toolchain intel qui permet
$> module load toolchain/intel
$> which icc
/opt/apps/resif/.../bin/intel64/icc
```

L'avantage de cette approche (et la raison pour laquelle elle est utilisée sur la quasi totalité des systèmes HPC à large échelle) est de décorréler la compilation et l'installation des logiciels optimisées pour les resources de calcul fournies (typiquement effectuée par l'équipe HPC en charge de maintenance opérationnelle du système[11]) de son utilisation qui se retrouve simplifiée pour les utilisateurs (à travers la commande `module load [...]`) qui n'ont plus à s'occuper des variables d'environnement et du répertoire d'installation (potentiellement compliqué) du logiciel utilisé.

Outre l'implémentation originale des *environnement modules*, il existe une suite logicielle alternative basée sur le langage de programmation LUA appelée *Lmod*[12] qui présente l'avantage de mieux gérer les dépendances entre les modules. C'est cette version qui, comme pour de nombreux sites HPC, est déployé au sein des clusters ULHPC. Il apparaissait dans tous les cas pertinent de sauvegarder l'information des modules logiciels chargés au sein des jobs soumis à la platforme (une information qui encore une fois n'est pas jugée pertinente pour le bon fonctionnement du RJMS). Nous verrons dans cette partie une proposition d'implémentation permettant de réaliser cette opération, i.e. de sauvegarder les modules chargés au sein d'un job dans une base de données, afin de pouvoir, à terme, exploiter ces informations. Cette implémentation se base sur l'utilisation de `syslog`, un utilitaire de gestion de journaux systèmes (ou *logs*) sous Linux, pour collecter les données, et une base de données MySql pour les conserver[13]. Chaque chargement de module générera un log sur le nœud, et chaque nœud enverra un message syslog à une machine qui centralisera ces logs pour les stocker dans une base de données. Dans cette implémentation, cette machine sera la machine virtuelle `slurm-master`, présentée à la section 2.1.

11 Au niveau de l'équipe HPC de l'Université, l'outil Easybuild est utilisé, comme pour de nombreux autres sites HPC, pour encadrer les étapes de compilation et d'installation des modules pour les logiciels proposées sur la plateforme - Voir https://easybuild.readthedocs.io/.

12 Voir https://lmod.readthedocs.io

13 Cette approche est proposée dans le documentation de *Lmod*, et est ici implémentée avec quelques modifications

Comme on a pu le voir, afin de charger un module logiciel, les utilisateurs exécutent la commande suivante :

```
module load <nom_du_module>
```

L'idée de base de l'implémentation décrite ici est de modifier la façon dont *Lmod* exécute cette commande, de manière à garder une trace extérieure du module chargé. En particulier, chaque fois que la commande `module` de *Lmod* est exécutée, tout le contenu du fichier `SitePackage.lua` (placé dans le dossier d'installation de *Lmod*) sera exécuté. C'est donc le bon endroit pour personnaliser le comportement de *Lmod*. Si on ne veut pas écrire dans le fichier déjà en place, on peut créer un nouveau fichier `SitePackage.lua` ailleurs, et indiquer à *Lmod* d'utiliser ce nouveau fichier en initialisant la variable `$LMOD_PACKAGE_PATH` avec le chemin du répertoire où se trouve notre `SitePackage` (on appelle ce répertoire *Site Directory*).

On trouvera le fichier `SitePackage.lua` utilisé pour cette implémentation en annexe, au listing 6. On y extrait six informations, que l'on écrit dans un log système avec un tag spécifique avec la commande système `logger`. Ces six informations sont les suivantes :

- le nom de l'utilisateur qui a chargé le module ;
- le jobID (donné par *Slurm*) du job concerné ;
- le host (c'est à dire le nom du nœud) ;
- la date actuelle ;
- le nom du module ;
- le chemin du module.

Le jobID est utile pour faire la correspondance avec la base *Slurm*. Le nom d'utilisateur et le nom de l'host ne sont pas absolument nécessaires pour notre objectif strictement, mais les informations sont disponibles et pourraient être utile à d'autres analyses, de même pour la date. Le nom du module et le chemin du module peuvent paraître des doublons mais si le nom est plus compact et facile à lire, le chemin informe de l'origine du module. On peut ainsi distinguer les modules proposés par les administrateurs, et les modules ajoutés par les utilisateurs dans leurs répertoires personnels. De même, cela n'est pas nécessaire à notre objectif, mais prodigue des informations intéressantes pour l'administration du système de modules.

L'étape suivante est de configurer *rsyslog* sur chaque nœud de manière à envoyer ces logs à la machine qui les centralisera. Dans le fichier `/etc/rsyslog.d/moduleTracking.conf` de chaque nœud de calcul, on ajoute la règle suivante :

```
# /etc/rsyslog.d/moduleTracking.conf
# [...]
if $programname contains 'ModuleUsageTracking' then @10.10.1.11:514
```

Dans cet exemple, l'adresse 10.10.1.11 correspond à la machine `slurm-master`, et le port 514 est le standard pour le protocole UDP.

On configure également *rsyslog* sur `slurm-master` afin de collecter ces logs, pour les écrire dans le fichier `/var/log/moduleUsage.log`. Dans le fichier `/etc/rsyslog.d/moduleTracking.conf`, on écrit :

```
$Ruleset remote
if $programname contains 'ModuleUsageTracking' then /var/log/moduleUsage.log
$Ruleset RSYSLOG_DefaultRuleset

# provides UDP syslog reception
$ModLoad imudp
$InputUDPServerBindRuleset remote
$UDPServerRun 514
```

On redémarre ensuite rsyslog sur tous les nœuds pour prendre en compte les changements de configuration :

```
sudo systemctl restart rsyslog
```

On peut ensuite ajouter une entrée à la configuration de *logrotate* de manière à faire une sauvegarde régulière du fichier `moduleUsage.log`

```
$ echo "\
/var/log/moduleUsage.log{
    missingok
    copytruncate
    rotate 4
    daily
    create 644 root root
    notifempty
}" > /etc/logrotate.d/moduleUsage.log
```

logrotate est une tâche planifiée avec *crontab*, pas un démon, il n'y a pas besoin de recharger sa configuration. Quand *crontab* exécute *logrotate*, il utilise automatiquement les nouvelles configurations.

Maintenant que l'on a une collecte de logs opérationnelle, on peut les stocker dans une base de données. Après avoir créé un utilisateur `lmod` dans mysql et lui avoir donné les droit sur la base de données qui va être créée, un script Python est disponible pour créer la base de donnée. Cette base contient cinq tables :

`jobt` : la table qui contient les jobIDs ;

`usert` : la table qui contient les utilisateurs ;

`modulet` : la table qui contient les informations relatives aux modules ;

`join_user_module` : une table de jointure qui associe utilisateurs et modules avec la date ;

`join_job_module` : une table de jointure qui associe jobs et modules avec la date.

Pour remplir la base de données avec les logs, un autre script Python est disponible. Ce script extrait les informations des logs formatés du fichier passé en paramètre (typiquement `/var/log/moduleUsage.log`). Afin d'automatiser le stockage, on peut préparer un script bash à faire exécuter par crontab à intervalle régulier :

```
#!/bin/bash

for logfile in /var/log/moduleUsage.log-*; do
    /path/to/python/script $logfile
    if [ $? -eq 0 ]; then
        rm -f $logfile
    fi
done
```

Enfin, un dernier script Python, dont les options d'aide sont montrées au listing 2.2, propose quelques analyses de la base de données. On peut voir :

- le nombre d'utilisateurs distincts utilisant chaque module ;
- le nombre de jobs distincts utilisant chaque module ;
- les utilisateurs de chaque module ;
- les modules utilisés par chaque utilisateurs ;
- les modules utilisés par chaque job ;
- les utilisateurs d'un certain motif dans le nom des module.

```
$ ./analyzeLmodDB --help
usage: analyzeLmodDB [-h] [--dbname DBNAME] [--syshost SYSHOST]
                     [--start STARTDATE] [--end ENDDATE]
                     [--sqlPattern SQLPATTERN]

                      ↪ {counts_users,counts_jobs,usernames,jobids,modules_used_by_user,modul

positional arguments:

  ↪ {counts_users,counts_jobs,usernames,jobids,modules_used_by_user,modules_used_by_job}
                        counts_users         : Report the number of distinct
                        ↪ users of a particular module
                        counts_jobs          : Report the number of distinct
                        ↪ jobs using a particular module
                        usernames            : Report users of a particular
                        ↪ pattern
                        jobids               : Report job ids of jobs using a
                        ↪ particular pattern
                        modules_used_by_user : Report the modules used by a
                        ↪ particular user
                        modules_used_by_job  : Report the modules used by a
                        ↪ particular job

optional arguments:
  -h, --help            show this help message and exit
  --dbname DBNAME       lmod db name
  --syshost SYSHOST     system host name
  --start STARTDATE     start date
  --end ENDDATE         end date
  --sqlPattern SQLPATTERN
                        sql pattern for matching
```

Listing 2.2 : options disponibles pour le script d'analyse de la base de données de l'utilisation des modules

3 RÉSULTATS EXPÉRIMENTAUX ET ANALYSES

Dans ce chapitre se trouvent les résultats obtenus validant les approches décrites au chapitre 2, accompagnés de quelques discussions autour de ces résultats.

3.1 ANALYSE DES JOBS DE LA BASE slurm

Cette première partie est dédiée aux résultats obtenus avec le script présenté en section 2.2.

La figure 10 représente la répartition du nombre de jobs en fonction du nombre de cœurs alloués. On peut y observer un grand pic à 28 cœurs : cela correspond à la réservation d'un nœud entier puisque la grande majorité des nœuds de calcul du cluster `iris` disposent de 2 processeurs de type Xeon E5-2680v4 ou Gold 6132 comportant 14 cœurs physiques chacun. On distingue ensuite très clairement un autre pic pour 2 nœuds, et 4 nœuds. On remarque également une large part de jobs utilisant une faible quantité de cœurs (i.e. jusqu'à 4). Ce graphe donne une première indication, certes naïve, de la part de jobs parallèles et séquentiels lancés sur le cluster.

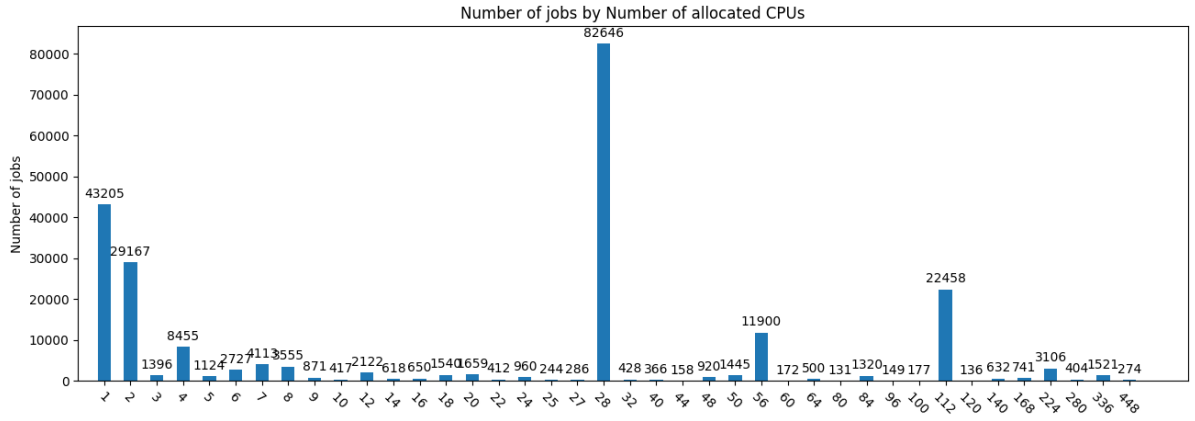


FIGURE 10 : Répartition du nombre de jobs en fonction du nombre de cœurs alloués.

La figure 11 représente la répartition du nombre de jobs en fonction de la part moyenne d'utilisation du CPU (temps CPU moyen par rapport au temps du job fois le nombre de cœurs). On observe que environ 40% des jobs (144000 sur 370000) n'utilisent que 5% du temps CPU total à leur disposition. On remarque également deux pics de même ordre de grandeur pour les jobs utilisant la moitié et la totalité des ressources CPU dont ils disposent.

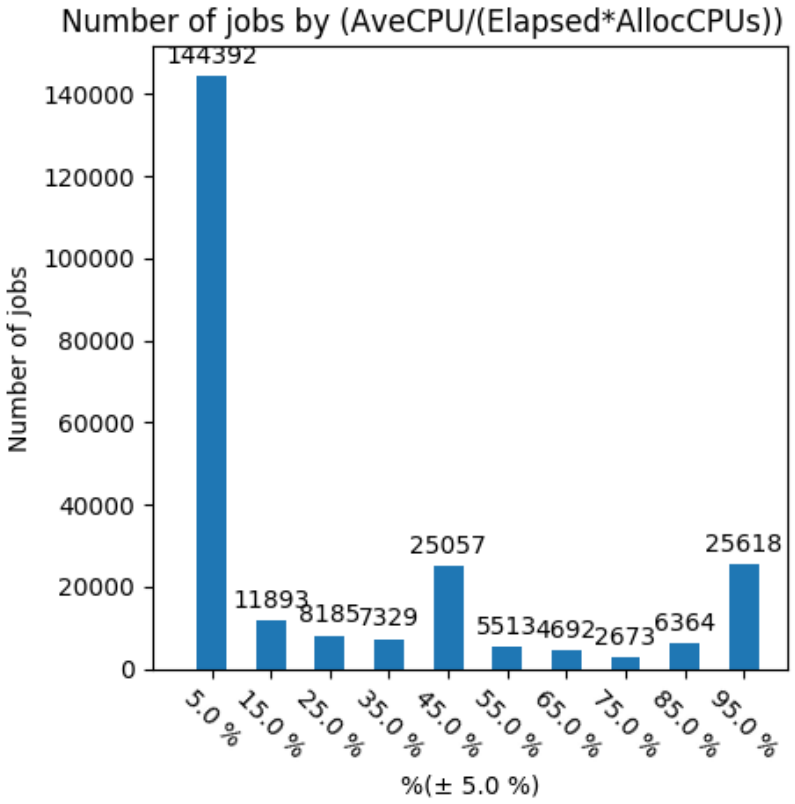


FIGURE 11 : Répartition du nombre de jobs en fonction de la proportion de temps CPU.

3.2 PARTITIONNEMENT AUTOMATIQUE DES PROFILS DE JOBS

On présentera dans cette section les résultats obtenus au terme du partitionnement automatique de 100K (100,000) jobs issus de la base de donnée *Slurm* du cluster `iris` de l'ULHPC répertoriant les statistiques des jobs exécutés sur la plateform au cours des deux dernières années. Ce partitionnement a été exécuté avec l'algorithme DBSCAN, et a pour paramètres $\varepsilon = 0.023$ (distance maximale entre deux points pour les considérer comme voisins) et $\mathtt{MinPts} = 990$ (nombre minimum de points à trouver dans un ε-voisinage d'un autre point pour considérer ce dernier comme point d'extension d'un groupe). Le coefficient de silhouette obtenu pour ce partitionnement est de 0.34.

Dans toutes les représentations graphiques présentées dans cette section, une même couleur et un même chiffre correspondent toujours au même groupe.

La figure 12 nous montre le nombre de groupes identifiés par le partitionnement, ainsi que leur répartition en nombre de points. Sur l'ensemble des 100K jobs, cinq groupes ont été identifiés dans lesquels se répartissent 73% des données. Les 27% restants (en gris) ont été considérés comme du bruit, et n'appartiennent à aucun groupe. Le groupe numéroté 0 est les le plus large de tous, avec 38% des données. Grâces aux résultats précédents, on peut intuiter que ce groupe contient les jobs peu consommateurs de ressources.

La figure 13 présente trois projections des données sur 3 plans différents, colorées par groupes identifiés. On observe que les groupes sont très liés au nombre de cœurs réservés. Cela était attendu, la métrique étant discrète. Cependant, elle ne gâche pas tout le partitionnement, puisqu'on voit que plusieurs groupes contiennent des jobs avec un nombre de cœurs variables

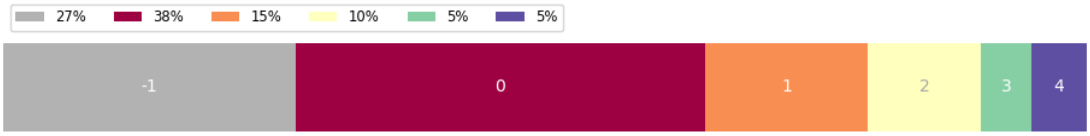


FIGURE 12 : Répartition des groupes identifiés

(les groupes 0 et 2 par exemple), et inversement, pour un même nombre de cœurs, des jobs peuvent être classés dans plusieurs groupes (1 et 3 par exemple).

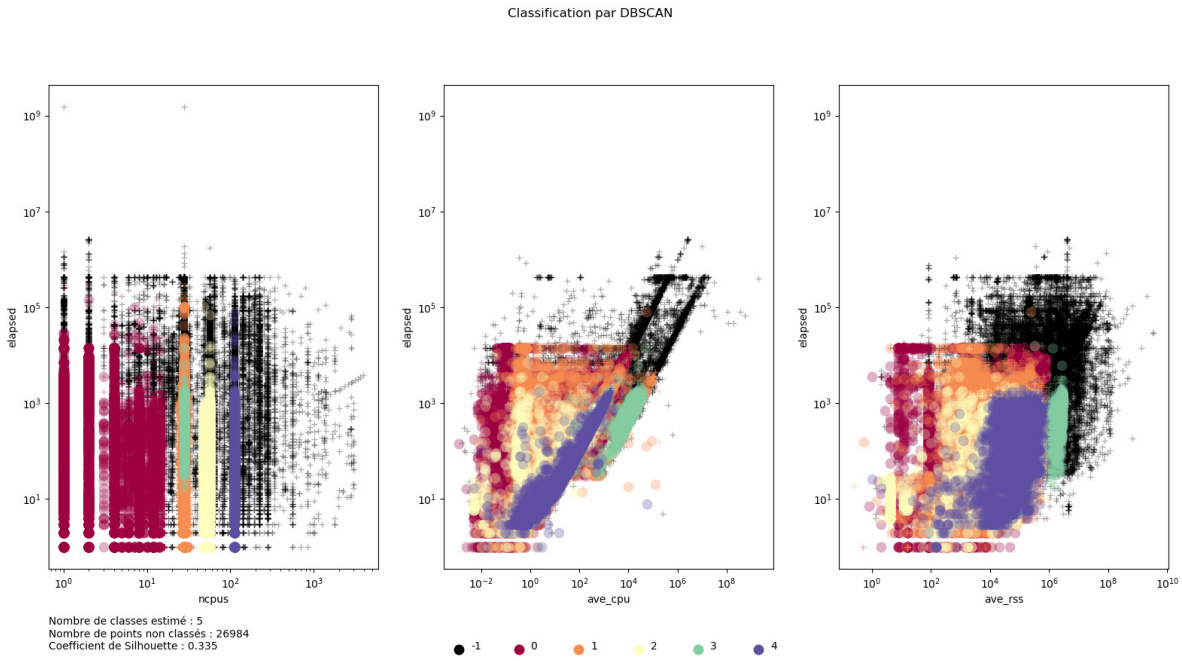


FIGURE 13 : Projections des données, colorées par groupes

La figure 14 présente les diagrammes en boîtes des cinq groupes identifiés pour les sept métriques, avec les courbes de densités.

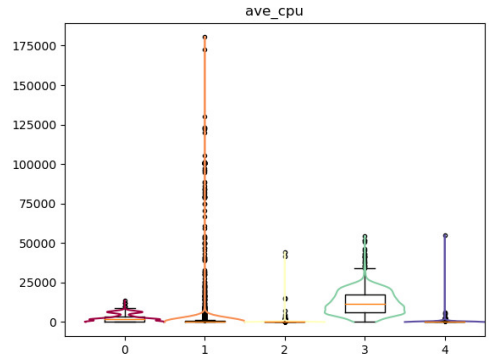
L'ensemble de ces représentations graphiques nous permettent d'identifier le contenu de chaque groupe.

Les groupes 3 et 4 sont les plus faciles à comprendre. Le groupe 3 semble correspondre à des jobs sur un nœud entier (28 cœurs), et plutôt intensifs en temps CPU et (surtout) en utilisation de la mémoire parmi l'ensemble de l'échantillon. Dans le groupe 4 se trouvent les jobs utilisant un nombre important de cœurs, et plus de mémoire que les autres groupes (excepté le 3). On voit que cela impacte la consommation d'énergie des jobs de ce groupe.

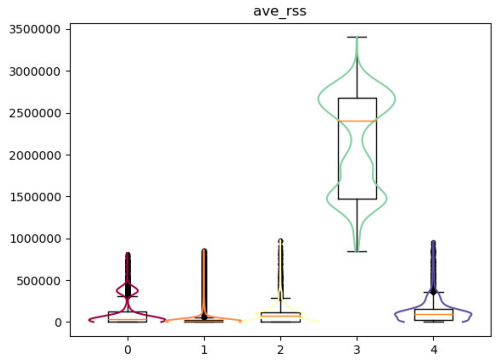
Le groupe 1 concerne également des jobs sur 1 nœud entier (comme le 3), mais moins intensifs en ressources. Cela nous montre bien la différence d'utilisation des ressources qu'il peut y avoir pour une réservation a priori similaire, et suggère un type de job différent.

Le groupe 0 contient les jobs utilisants peu de CPU, mais avec une consommation des autres ressources très variées. La plupart des jobs les plus intensifs en lecture/écriture disque se trouvent dans ce groupe (et également les jobs les plus consommateurs d'énergie).

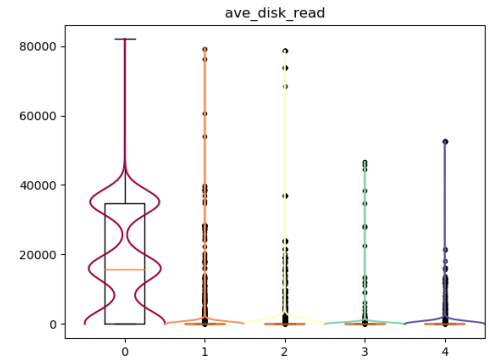
Le groupe 2 (malgré un nombre de cœurs supérieur et plus varié) semble assez proche du groupe 1 en terme de consommation de ressources. Généralement, les mesures pour le groupe



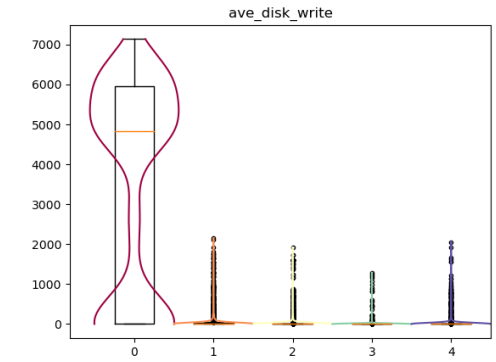
(a) Temps CPU moyen total



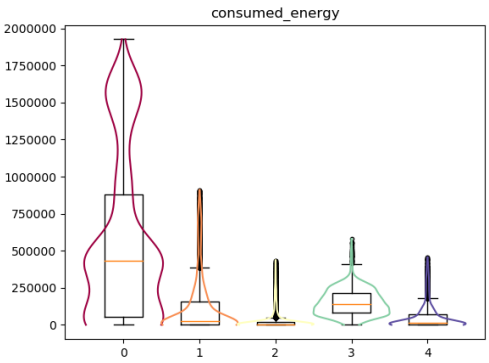
(b) Consommation mémoire moyenne totale



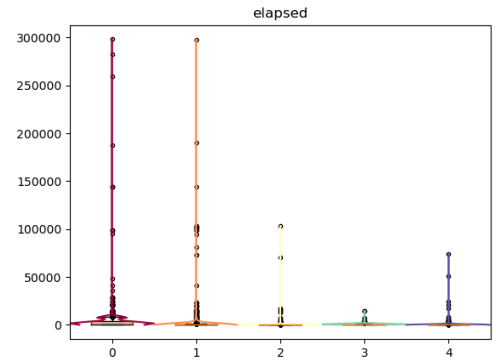
(c) Lecture disque moyenne totale



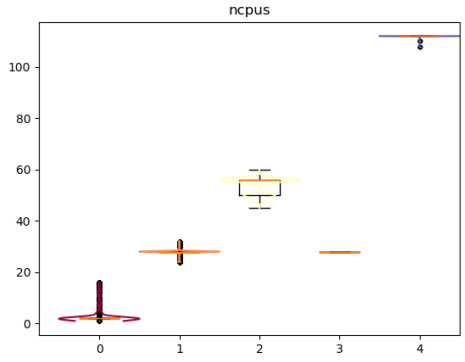
(d) Écriture disque moyenne totale



(e) Énergie consommée



(f) Temps d'exécution



(g) Nombre de CPUs alloués

FIGURE 14 : Diagrammes en boîtes des 7 métriques pour chaque groupe identifié

1 sont plus étalées que celle pour le groupe 2. Cela est probablement lié au fait que le groupe 1 est plus grand (en nombre de points) que le groupe 2.

La figure 14a est intéressante car un peu contre-intuitive. En effet, on penserait naturellement que plus un job a de cœurs, plus son temps CPU moyen total sera long (puisque ce temps est la somme du temps CPU moyen de chacune de ses tâches). Cependant, on n'observe aucune tendance de ce genre. En considérant que la plupart des jobs on un temps d'exécution constant (et proche de 0), comme on peut le voir à la figure 14f, alors les valeurs de la figure 14a peuvent être vues comme un « taux d'efficacité » (en utilisation des cœurs disponibles) des jobs. Il est clair alors que les jobs du groupe 3 sont très efficaces (en ce sens), mais c'est le groupe 2 qui contient les jobs les plus efficaces (bien qu'ils restent des valeurs extrêmes).

Une mesure qui aurait pu être pertinente, mais qui est absente ici, est le rapport du temps CPU moyen total sur le temps d'exécution du job. Cette mesure permettrait d'éviter l'approximation précédente d'un temps d'exécution constant (et proche de 0).

Grâce à ces observations, on peut avoir une meilleure compréhension de l'utilisation du cluster `iris`, et identifier des profils de jobs.

3.3 ANALYSE DES MODULES LOGICIELS CHARGÉS AU SEIN DES JOBS

Les tests ont montré que l'implémentation du traçage de l'utilisation des modules logiciels chargés au sein des jobs, telle que décrite à la section 2.4 est fonctionnelle. Des tests concluants ont été effectués pour des jobs interactifs, et des jobs passifs, dans l'environnement virtuel décrit à la section 2.1.

Cependant, aucun résultat intéressant ne sera montré ici, car cette solution n'a pas encore été implémentée sur le cluster `iris` en production. Afin de faciliter cette mise en production, j'ai préparé deux scripts shell (l'un pour les nœuds de calcul, l'autre pour la machine qui centralise les logs) qui exécutent les actions décritent pour la mise en place décrite précédemment. Ces scripts pourront être exécuté par un gestionnaire de configuration (tel que *Puppet*) de sorte à automatiser le déploiement de cette solution.

4 CONCLUSION ET PERSPECTIVES

Ce mémoire a été écrit dans le contexte d'un stage de fin d'études d'ingénieur au sein de l'Université du Luxembourg sous la supervision du Dr. Sébastien Varrette qui co-dirige le développement de la plateforme de calcul haute performance de l'Université.

Les travaux présentés dans ce manuscrit interviennent dans le cadre du projet ENERGUMEN qui vise à développer de nouveaux algorithmes d'allocation de tâches malléables afin de permettre des économies d'énergie au sein des grandes plate-formes distribuées de type HPC. De manière précurseur à tous travaux sur ce projet, il était nécessaire d'avoir une compréhension aboutie de l'utilisation et de la consommation des resources HPC mises à la disposition des chercheurs au sein du cluster `iris` de l'Université, en amenant une classification fine des jobs soumis sur cette plateforme depuis son ouverture en juin 2017 jusqu'à aujourd'hui.

A ce titre, j'ai travaillé sur une copie de la base de donnée de l'instance de l'ordonnanceur *Slurm* en place au niveau du cluster *iris* de l'Université du Luxembourg, qui contenait les informations détaillées sauvegardé pour les jobs soumis entre le 4 mai 2017 et le 13 mai 2019. Dans un premier temps, j'ai étudié la répartition des jobs en fonction de leur consommation de diverses ressources. J'ai ensuite proposé une approche en partitionnement automatique non supervisé pour tenter de classifier les types de jobs. Plusieurs étapes furent nécessaires pour optimiser cette classification, depuis le choix des métriques (*features*) à considérer comme dimensions pour l'algorithme de partitionnement avec la technique PCA, jusqu'à l'affinement des paramètres de l'algorithme de partitionnement selectionné (Density-Based Spatial Clustering of Applications with Noise (DBSCAN)). Le partitionnement obtenu par l'analyse de 100K jobs a permis d'identifier 5 groupes de jobs dépendant de leurs consommations en ressources HPC (depuis les plus intensifs aux plus légers), complétant ainsi les résultats obtenus lors de la première analyse des jobs soumis sur le cluster `gaia` présentés dans [1].

Finalement, le jeux de données stockées dans la base *Slurm* ne concerne que les métriques liées à l'utilisation des ressources systèmes des nœuds de calculs alloués, en particuliers les informations liées à l'environnement logiciel des jobs (et notamment les modules logiciels chargés au sein de chaque job qui fournissent pourtant une information critique sur le profil utilisateur et le type de job) ne sont pas sauvegardées. C'est pour cette raison qu'a été proposé ici une approche permettant le traçage des modules logiciels utilisés au sein des jobs et porteurs d'informations significatives sur le type de travail associé à ces jobs. L'objectif reste bien sûr d'affiner le modèle de partitionnement obtenu, même si l'implémentation proposée n'est pas encore effective sur le cluster de production `iris`. Lorsqu'elle le sera, il sera alors possible d'envisager une classification automatique supervisée exploitant cette nouvelle dimension.

Les perspectives de ces travaux sont nombreuses puisque la classification obtenue vise à proposer à l'ordonnanceur une prévision du profil de consommation en ressource d'un job directement lors de sa soumission, permettant ainsi d'orienter l'allocation vers des ressources dont la configuration aura été (ou sera) optimisée en fonction de ce profil et de la consommation associée.

5 BILAN

Travailler avec l'équipe qui gère et administre la plateforme HPC de l'Université du Luxembourg est une expérience unique. Ce stage m'a permis d'évoluer dans une structure académique, avec ses avantages et ses inconvénients. Ce nouveau contact avec le monde des systèmes d'information me permet d'être initié à ses règles et coutumes.

Ce stage a été une opportunité pour moi car la problématique HPC a été un axe central de mon parcours à l'ENSIIE. Il m'a permis de m'investir plus encore dans ce domaine, et de constater à nouveau la diversité des activités et des connaissances nécessaires qui gravitent autour de ce milieu.

De plus, ce stage m'a permis d'apprendre et d'exercer les méthodes de travail, les usages, conventions et outils en place ici. Cela m'apparaît comme un premier pas dans le monde de la recherche en informatique.

J'y ai rencontré divers difficultés, desquelles j'ai appris : le déploiement d'un environnement de tests virtuel pour répliquer le comportement de *Slurm*, ou encore le choix des variables et dimensions pour un partitionnement automatique des données.

Pour finir, c'est donc une expérience riche en nouveauté pour moi. Durant ce stage, j'ai eu à jongler entre les compétences, en faisant du développement, de la gestion de logs, et de l'analyse statistique. J'ai apprécié la diversité des tâches nécessaires à la réalisation de mon projet. Ce stage de fin d'étude aura été une bonne conclusion pour mon parcours à l'ENSIIE, qui me permettra de m'épanouir dans le domaine des systèmes d'information dans le domaine du HPC.

Note : Les expérimentations présentées dans ce documents on été menées avec les ressources HPC de l'Université du Luxembourg. [4] – voir `http://hpc.uni.lu`.

BIBLIOGRAPHIE

[1] Emeras, J., Varrette, S., Guzek, M., Bouvry, P. : Evalix : Classification and Prediction of Job Resource Consumption on HPC Platforms. In : Proc. of the 19th Intl. Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'15), part of the 29th IEEE/ACM Intl. Parallel and Distributed Processing Symposium (IPDPS 2015). IEEE, India (May 2015)

[2] Ester, M., Kriegel, H.P., Sander, J., Xu, X. : A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In : Proceedings of the Second International Conference on Knowledge Discovery and Data Mining. pp. 226–231. KDD'96, AAAI Press (1996), http://dl.acm.org/citation.cfm?id=3001460.3001507

[3] Feitelson, D.G., Rudolph, L. : Toward convergence in job schedulers for parallel supercomputers. In : Job Scheduling Strategies for Parallel Processing. pp. 1–26. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)

[4] Varrette, S., Bouvry, P., Cartiaux, H., Georgatos, F. : Management of an Academic HPC Cluster : The UL Experience. In : Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014). pp. 959–967. IEEE, Bologna, Italy (July 2014)

ANNEXES

A OPTIONS DU SCRIPT SPYGRAPH.PY

Le listing 1 présente l'ensemble des options disponibles pour le script présenté à la section 2.2

```
$ ./spygraph.py --help
usage: spygraph.py [-h] [--compute [COMPUTE [COMPUTE ...]]]
                   [--stepsize STEPSIZE] [--min MIN] [--max MAX]
                   [--logarithmic] [--starttime STARTTIME] [--endtime ENDTIME]
                   [--user USER] [--title [TITLE [TITLE ...]]]
                   metric [metric ...]

Plot a graph of the number of jobs for each value of the  given metric according
↪  to the slurm data.

positional arguments:
  metric                The metric(s) you want to plot. Available metrics are :
                                AveCPU -> Average CPU time (s)
                                AveRSS -> Average RSS (B)
                                MaxRSS -> Max RSS (B)
                                AveDiskRead -> Average Disk Read (B)
                                AveDiskWrite -> Average Disk Write (B)
                                ConsumedEnergyRaw -> Consumed Energy (J)
                                AllocCPUs -> Number of allocated CPUs ()
                                Elapsed -> Job's elapsed time (s)

optional arguments:
  -h, --help            show this help message and exit
  --compute [COMPUTE [COMPUTE ...]], -c [COMPUTE [COMPUTE ...]]
                        Operations to perform on asked metrics. Available
                        ↪  operations are :
                                div m1,m2  -> m1/m2
                                mul m1,m2  -> m1*m2
                                add m1,m2  -> m1+m2
                                sub m1,m2  -> m1-m2
                        'm1' and 'm2' must be replaced by the number
                        ↪  representing the position of the wanted metric as it
                        ↪  is given in positional argument, starting with 1.
                        Operations can be combined.
  --stepsize STEPSIZE, -s STEPSIZE
                        The step you want for the discretisation of continuous
                        ↪  data, in the unit of the final metric. Ignored if
                        ↪  the data is discrete, and applied to all graphs if
                        ↪  several are asked.
  --min MIN, -m MIN     The minimum value you want, in the unit of the final
  ↪  metric.
  --max MAX, -M MAX     The maximum value you want, in the unit of the final
  ↪  metric.
  --logarithmic, -l     Create the graph with a logarithmic scale.
  --starttime STARTTIME, -S STARTTIME
                        Select jobs after the specified time. Same format than
                        ↪  time formats of sacct command.
```

```
--endtime ENDTIME, -E ENDTIME
                      Select jobs before the specified time. Same format than
                      ↪  time formats of sacct command.
--user USER, -u USER  The user you want the metric to be plotted for.
--title [TITLE [TITLE ...]], -t [TITLE [TITLE ...]]
                      The title you want for the graph. Available only if one
                      ↪  graph is asked (one metric or -c option).
```

Listing 1 : Interface implémentée dans le script `spygraph`.

B ALGORITHME DBSCAN

Le listing 2 décrit (sous forme de pseudo-code) l'algorithme DBSCAN utilisé pour le partitionnement automatique non supervisé proposé dans le cadre de ces travaux.

```
DBSCAN(D, eps, MinPts)
   C = 0
   pour chaque point P non visité des données D
      marquer P comme visité
      PtsVoisins = epsilonVoisinage(D, P, eps)
      si tailleDe(PtsVoisins) < MinPts
         marquer P comme BRUIT
      sinon
         C++
         etendreCluster(D, P, PtsVoisins, C, eps, MinPts)

etendreCluster(D, P, PtsVoisins, C, eps, MinPts)
   ajouter P au cluster C
   pour chaque point P' de PtsVoisins
      si P' n'a pas été visité
         marquer P' comme visité
         PtsVoisins' = epsilonVoisinage(D, P', eps)
         si tailleDe(PtsVoisins') >= MinPts
            PtsVoisins = PtsVoisins U PtsVoisins'
      si P' n'est membre d'aucun cluster
         ajouter P' au cluster C

epsilonVoisinage(D, P, eps)
   retourner tous les points de D qui sont à une distance inférieure à epsilon
   ↪ de P
```

Listing 2 : Pseudo-code de l'algorithme DBSCAN utilisé pour le partitionnement automatique des jobs *Slurm* soumis sur le cluster `iris`.

C PARALLÉLISATION DU CALCUL DE minpts

Cette section présente le processus de parallélisation mentionné à la section 2.3.2.
Le fichier `calculs.py`, présenté au listing 3 contient l'algorithme qui calcule le $(n * \varepsilon)$-voisinage de chaque point, et renvoie la valeur de *MinPts* telle que 95% des points ont au moins *MinPts* $(n * \varepsilon)$ voisins, ainsi que la silhouette obtenue avec cette valeur de *MinPts*. C'est cette fonction qui sera appelée simultanément, avec des valeurs de n changeantes.

```python
#!/usr/bin/env python3

import numpy as np
from sklearn.metrics import pairwise_distances
from sklearn.cluster import DBSCAN
from sklearn import metrics

def get_minpts_silh(X, eps, mult):
    # this array count the number of neighbors
    # within radius of each line of X, in the same order
    nbrs = np.zeros(X.shape[0], int)
    for idx, vect_x in enumerate(X[:-1]):
        # distance is symmetric, we count only once
        distances = pairwise_distances([vect_x], X[idx+1:], metric='minkowski')
        ok_dist = (distances <= eps*mult)[0]
        nbrs[idx] += len(ok_dist.nonzero()[0])
        nbrs[idx+1:][ok_dist] += 1

    nbrs[::-1].sort() # reverse sort inplace
    # returns the number of neighbor such as 95% of all points
    # have more neighbors than this number
    minpts = nbrs[int(0.95*len(nbrs))]

    clust = DBSCAN(eps=eps, min_samples=minpts).fit(X)
    silh = metrics.silhouette_score(X, clust.labels_)

    return (minpts, silh)
```

Listing 3 : fichier calculs.py

Le fichier `parallelizer.py`, présenté au listing 4, est le point d'entrée du programme. C'est ici que les données sont récupérées et pré-traitées, avant d'être envoyées aux processus de calculs. Les retours des calculs sont ensuite regroupés, et écrit dans un fichier.

```python
#!/usr/bin/env python3
import argparse
import logging
import os
import sys
from sklearn.datasets.samples_generator import make_blobs
from sklearn.externals.joblib import Parallel, parallel_backend
from sklearn.externals.joblib import register_parallel_backend
```

```
from sklearn.externals.joblib import delayed
from sklearn.externals.joblib import cpu_count
from ipyparallel import Client
from ipyparallel.joblib import IPythonParallelBackend
import datetime

import pandas as pd
from sklearn.preprocessing import scale
from calculs import get_minpts_silh

FILE_DIR = os.path.dirname(os.path.abspath(__file__))
sys.path.append(FILE_DIR)

#prepare the logger
parser = argparse.ArgumentParser()
parser.add_argument("-p", "--profile", default="ipy_profile",
                help="Name of IPython profile to use")
args = parser.parse_args()
profile = args.profile
logging.basicConfig(filename=os.path.join(FILE_DIR,profile+'.log'),
                    filemode='w',
                    level=logging.DEBUG)
logging.info("number of CPUs found: {0}".format(cpu_count()))
logging.info("args.profile: {0}".format(profile))

#prepare the engines
c = Client(profile=profile)
#The following command will make sure that each engine is running in
# the right working directory to access the custom function(s).
c[:].map(os.chdir, [FILE_DIR]*len(c))
logging.info("c.ids :{0}".format(str(c.ids)))
bview = c.load_balanced_view()
register_parallel_backend('ipyparallel',
                          lambda : IPythonParallelBackend(view=bview))
# prepare the data
DATA_FILE = 'jobs_dump.dat'
EPS = 0.023

jobs = pd.read_csv(DATA_FILE, engine='python', sep=' ;; ', header=0,
↪  nrows=100000)
jobs = jobs._get_numeric_data() #keep only numeric features
jobs.drop(columns=['job_id', 'exit_code', 'state', 'nnodes', 'max_rss'],
↪  inplace=True)

X = jobs.values # convert the data into a numpy array
X = scale(X)

# parallel execution
with parallel_backend('ipyparallel'):
    mpt_silh = Parallel(n_jobs=len(c))(delayed(get_minpts_silh)(X, EPS, mult)
```

```
                                for mult in range(1,51))
#write down the results
mpt_silh.sort(key=lambda e:e[0]) # sort by minpts
with open(os.path.join(FILE_DIR,'minpts_silhouettes.csv'), 'w') as f:
    f.write('minpts silhouette\n')
    f.write('\n'.join(' '.join(str(v) for v in l) for l in mpt_silh))
    f.write('\n')
```

Listing 4 : fichier parallelizer.py

Le fichier `prl_launcher.sh`, présenté au listing 5, est un script sbatch, à destination de l'ordonnanceur *Slurm*. Les premières lignes, celles qui commencent par « `#SBATCH` » sont des directives pour *Slurm*, et correspondent aux options de réservation de ressources. Ensuite, on trouve des instruction bash pour préparer l'environnement et finalement lancer le script donné en paramètre.

```
#!/bin/bash -l

#SBATCH -p batch        # batch partition
#SBATCH -J ipy_engines  # job name
#SBATCH -n 10           # 10 cores
#SBATCH -N 1            # 1 node
#SBATCH --mem=64G       # 64G memory
#SBATCH -t 2:00:00      # killed after 2 hours

module load lang/Python/3.6.4-foss-2018a-bare

source scikit/bin/activate

#create a new ipython profile appended with the job id number
profile=job_${SLURM_JOB_ID}

echo "Creating profile_${profile}"
ipython profile create ${profile}

ipcontroller --ip="*" --profile=${profile} &

#srun: runs ipengine on each available core
srun ipengine --profile=${profile} --location=$(hostname) &

echo "Launching job for script $1"
python $1 -p ${profile}
```

Listing 5 : fichier prl_launcher.sh

Finalement, l'ensemble est placé en file d'attente sur le cluster avec la commande suivante :

```
  sbatch prl_launcher.sh parallelizer.py
```

D TRAÇAGE DES MODULES LOGICIELS CHARGÉS AVEC LMOD

On trouvera dans cette section les scripts mentionnés à la section 2.4, concernant l'implémentation d'un système de traçage de modules utilisés par les jobs à partir de *Lmod*.

```
require("strict")
require("lmod_system_execute")
local hook    = require("Hook")
local uname   = require("posix").uname
local cosmic  = require("Cosmic"):singleton()
local syshost = cosmic:value("LMOD_SYSHOST")
-- By using the hook.register function, this function "load_hook" is called
-- ever time a module is loaded with the file name and the module name.
local s_msgA = {}

function load_hook(t)
   -- the arg t is a table:
   --     t.modFullName:  the module full name: (i.e: gcc/4.7.2)
   --     t.fn:           The file name: (i.e
   ↪  /apps/modulefiles/Core/gcc/4.7.2.lua)
   -- Writing to syslogd:
   if (mode() ~= "load") then return end
   local user        = os.getenv("USER")
   local jobid       = os.getenv("SLURM_JOBID")
   local host        = syshost or uname("%n")
   local currentTime = epoch()
   local msg         = string.format("user=%s jobid=%s module=%s path=%s
   ↪  host=%s time=%f",
                                     user, jobid, t.modFullName, t.fn, host,
                                     ↪  currentTime)
   local a           = s_msgA
   a[#a+1]           = msg
end

hook.register("load",load_hook)

function report_loads()
   local sys         = os.getenv("LMOD_sys") or "Linux"
   if (sys == "Linux") then
      local a = s_msgA
      for i = 1,#a do
         local msg = a[i]
         lmod_system_execute("logger -t ModuleUsageTracking -p local0.info " ..
         ↪  msg)
      end
   end
end

ExitHookA.register(report_loads)
```

Listing 6 : fichier SitePackage.lua

E DÉVELOPPEMENT DURABLE ET RESPONSABILITÉ SOCIÉTALE

Sur le site de l'Université du Luxembourg, on peut lire[1] :

> *Au Luxembourg, une discipline essentielle de recherche et et d'enseignement a été mise en place pour décrire les défis de durabilités définis dans des systèmes spécifiques, et le développement de domaines de recherches interdisciplinaires clefs, incluant le transport/la mobilité, l'économie de la santé, l'économie écologique, l'économie géographique, la socio-économie et les études démographiques, et l'informatique verte est bien en cours.*
>
> *Il y a cependant relativement peu de recherches qui combinent les sciences naturelles et l'ingénierie avec des perspectives critiques issues des sciences sociales et humaines concernant le fonctionnement de notre société. C'est pourtant un prérequis pour mieux comprendre et agir sur sur des systèmes socio-écologiques complexes.*
>
> *Une nouvelle discipline de science de la transformation durable est en train d'émerger, en laquelle les chercheurs, la société civile, les entreprises privées, et les gouvernements collaborent dans le but de transformer la façon dont la société et l'environnement interagissent.*

L'Université semble donc avoir des ambitions à grande échelle et à long terme concernant le développement durable.
Un point pratique dans l'organisation actuelle de la mobilité va cependant à l'encontre de ces principes. L'Université, originellement présente dans la ville de Luxembourg, a ouvert le campus de Belval dans le sud du pays en 2015, et y déménage progressivement. Ce campus devient un centre important du développement du pays, et les flux quotidiens de personnes y sont très importants. Les parkings ont été rendus payants pour décourager l'utilisation de véhicules personnels, mais pourtant, l'offre des transport en commun n'a pas évolué. Beaucoup de travailleurs sur le campus viennent de la ville française Thionville, et très peu de possibilités d'usage des transports en commun leur sont possibles (une ligne régulière peu fréquente, et des mauvaises correspondances avec le réseau Luxembourgeois).
Cependant, à l'échelle du pays, le réseau de transports en communs est géographiquement large, régulier, et les billets unifiés et à un prix attractif. De plus, un réseau de tram est en construction dans la capitale, et il est prévu pour le printemps 2020 de rendre l'intégralité des transports en commun dans tout le pays gratuits.

À l'échelle de l'ULHPC (et dans le domaine du HPC de façon générale), réduire la consommation énergétique est un vrai défi. En effet, les machines doivent tourner en permanence pour répondre aux contraintes de disponibilités. Mais la plus grosse part de consommation d'un système HPC ne vient pas tant des machines elles-même que du système de refroidissement. Le système de refroidissement actuellement en place sur le cluster Iris de l'ULHPC est un système de ventilation par la porte des racks. Les racks sont disposés en rangées face à face deux à deux, créant une alternance de couloirs d'air frais (envoyé dans les machines) et de couloirs d'air chaud (expulsé des machines). Un tel système est efficace, mais représente près de la moitié de la consommation énergétique totale du centre.
Une nouvelle technologie de refroidissement va bientôt être mise en place sur le nouveau cluster `aion` qui doit compléter `iris` début 2020. Appelée *DLC*, pour *Direct Liquid Cooling*, ce dispositif fait circuler un liquide (de l'eau en général, bien meilleure conductrice thermique que l'air), à l'aide d'une pompe dans un circuit qui passe à travers les lames de calcul pour être mise en contact directement avec les composants électroniques fortement émetteurs de chaleurs (les processeurs, la RAM, les accélérateurs etc.).

1 traduction libre de l'anglais

Cette méthode est bien plus efficace pour transférer la chaleur dégagée par les composants vers l'extérieur et permet d'atteindre un indicateur d'efficacité énergétique (en anglais *Power Usage Effectiveness (PUE)*) proche de 1. Le PUE indique quel est le ratio entre l'énergie totale consommée par l'ensemble du centre d'exploitation (avec entre autres, le refroidissement, le traitement d'air, les UPS (onduleurs)...) et la partie qui est effectivement consommée par les systèmes informatiques que ce centre exploite (serveurs, nœuds de calcul, stockage, réseau). Le Direct Liquid Cooling (DLC) s'avère donc nettement plus économique en énergie que le refroidissement traditionnel à air (qui assure typiquement un PUE de 1.5).