Gestion de l'énergie sur la plate-forme de calcul scientifique PlaFRIM

Corentin Mercier

Stage de fin d'études - Rapport :
Gestion de l'énergie sur la plate-forme de calcul scientifique PlaFRIM

Corentin Mercier

Août 2022

université de BORDEAUX

Inria

Maître de stage : Brice GOGLIN

Table des matières

Table des figures

Remerciements

Pour commencer, je souhaite remercier **Brice GOGLIN**, mon maître de stage qui m'a apporté son savoir et ses précieux conseils ainsi que de nouveaux contacts professionnels qui m'ont apporté leur aide. Je lui suis très reconnaissant pour tout ce qu'il a fait afin de m'offrir cette opportunité de stage.

Ensuite, je remercie **Julien LELAURAIN** et **Loic SIRVIN**, les deux administrateurs de **PlaFRIM** qui ont fait tout leur possible pour mettre à ma disposition les outils nécessaires au fur et à mesure de mon stage afin que je puisse réaliser mes expériences dans les meilleures conditions.

Pour continuer, je tiens à remercier **Amina GUERMOUCHE**, pour m'avoir apporté ses connaissances sur les mécanismes de gestion de l'énergie sous **Linux** ainsi que son soutien lors de certains moments difficiles de mon stage.

Enfin, je souhaite remercier **les autres membres de l'équipe PlaFRIM** ainsi que **mes collègues chez TADaaM** qui m'ont apporté un cadre de travail exceptionnel, de bons conseils ainsi qu'une ouverture de qualité sur le monde de la recherche.

1 Introduction

Au cours de ma cinquième année à l'**Université de Bordeaux**, j'ai suivi la spécialité calcul intensif et sciences des données. Cette spécialité comprends deux domaines. Le premier, le calcul haute performance, s'intéresse à l'utilisation maximale des capacités des machines afin d'accélérer les calculs. Le second, se concentre sur l'exploitation de grandes masses de données afin d'en tirer des informations utiles.

Au cours de ma formation, qui se déroulait en grande partie à l'**ENSEIRB-MATMECA**, j'ai beaucoup utilisé la plate-forme de calcul **PlaFRIM** (plus exactement sa partition « **Formation** » dédiée aux enseignements). Le but était de nous familiariser avec le fonctionnement des centres de calculs qui sont au cœur de notre spécialité.

Pendant les cours, les enseignants nous ont montré la capacité de calcul des supercalculateurs modernes. Ces dernières vont de paire avec une consommation électrique toujours plus importante. Elle grandit à tel point qu'il devient de plus en plus difficile d'augmenter la puissance des supercalculateurs sans engendrer un projet irréaliste sur le plan énergie (pour donner un ordre de grandeur, le supercalculateur le plus puissant [capable de monter un peu au dessus de 10^{18} opérations par secondes] consomme **20 MW**, l'équivalent de deux fois la consommation électrique de tout le **Togo**).

Bien que l'enjeu climatique soit une urgence, la question de l'impact environnemental de notre domaine d'activité n'est que peu abordé dans le cursus. C'est l'envie d'acquérir des connaissances solides sur la question énergétique dans l'informatique couplé à la volonté de m'investir dans la plate-forme de calcul locale qui a servi pour ma formation que j'ai choisi de faire un stage de six mois sur **PlaFRIM**.

Dans ce rapport, je vous présenterai d'abord l'environnement ainsi que les objectifs du stage puis je parlerai du travail que j'ai réalisé ainsi que du chemin que j'ai pris avant de terminer sur le bilan de ce stage. Un glossaire est fourni en fin de rapport (en annexe, section A). Il définit la plupart des termes techniques liés au domaine.

2 Présentation du stage

Dans cette partie, nous poserons le stage dans son contexte ainsi que les motivations qui ont mené à sa création avant de détailler mon environnement de travail ainsi que les objectifs qu'il m'était demandé d'atteindre.

2.1 Entreprise d'accueil

J'ai effectué mon stage au sein du **Centre Inria de l'Université de Bordeaux**. Cette composante d'**Inria** est un laboratoire de recherche de renom et emploie plus de 260 chercheurs répartis dans 18 équipes-projets différentes. Travaillant sur **PlaFRIM**, au sein de l'équipe **TADaaM**, j'étais entouré de chercheurs dont l'objectif est de construire une couche service faisant le lien entre les besoins des applications et les caractéristiques bas niveau du système d'exécution.

Pour être plus précis, j'ai travaillé dans un *open-space* où se trouvaient des stagiaires (allant de première année à dernière année d'études), des doctorants, des post-doc et des ingénieurs. Ces derniers, quasiment tous formés sous **Linux** et maîtrisant souvent des langages bas niveau comme C ou C++ aidaient les stagiaires. Plus généralement, une ambiance d'entraide régnait au bureau. Les personnes les plus expérimentées (certains post-doc et ingénieurs) maîtrisaient bien plus **PlaFRIM** que moi et ont pu m'aider par moments.

Pour terminer, de nombreux séminaires sont donnés au sein du laboratoire afin que les chercheurs mais aussi les doctorants et les stagiaires puissent partager leur travail. Au cours de ce stage, j'ai pu assister à une dizaine de séminaires traitant de sujets divers tels que l'impact de la mémoire sur des systèmes hétérogènes, la répartition de tâches OpenMP au plus près possible des données ou la présentation de nombreux projets en partenariat avec des acteurs industriels tels que **Airbus** ou **Naval Group**.

2.2 Contexte

Le stage est focalisé sur la plate-forme de calcul commune à l'**IMB**, au **LaBRI** et hébergée par le centre : **PlaFRIM**. C'est une plate-forme très hétérogène servant principalement aux chercheurs souhaitant tester leurs applications sur des configurations matérielles très variées. Chaque configuration matérielle est représentée par un groupe de nœuds (de machines). Par exemple, les nœuds allant de **bora001** à **bora044** appartiennent au groupe **bora** et possèdent le même *hardware*.

Le stage se place dans un contexte de sobriété énergétique de **PlaFRIM**. En effet, sa consommation représente un peu plus de 30% de la consommation électrique du centre alors que son taux d'utilisation moyen se trouve entre 30 et 40 %. Pour avoir un ordre de grandeur, le laboratoire consomme environ **1600 MWh** par an dont **500** sont liés au fonctionnement de **PlaFRIM** [1].

D'un côté, la direction du centre souhaite réduire la consommation de **PlaFRIM** afin d'en amortir les coûts de fonctionnement ainsi que pour tendre le plus possible vers les objectifs climatiques fixés par le gouvernement.

De l'autre, les utilisateurs sont sensibles à la question écologique et aimeraient voir du changement au sein de la plate-forme qui consomme inutilement de l'énergie à maintenir allumé des machines inactives.

Enfin, les administrateurs, conscients du problème et acteurs du changement, ont proposé ce sujet de stage afin de satisfaire les demandes de la direction et des utilisateurs. Cela allait aussi dans le sens d'une chargée de mission récemment nommée pour baisser la consommation d'**Inria** de 10% à l'échelle nationale.

Pour terminer, ce stage devait également permettre aux administrateurs de mettre en place des mécanismes de réduction de la consommation électrique de **PlaFRIM** ainsi que d'avoir une vue d'ensemble des solutions disponibles pour l'avenir.

2.3 Environnement de travail

Pour réaliser mon travail, j'avais un bureau à ma disposition dans l'*open-space* de **TADaaM** avec un ordinateur portable et un poste de travail classique.

Ensuite, côté logiciel, je disposais d'un accès utilisateur à la partition principale de **PlaFRIM** ainsi qu'à la partition « **Formation** ». C'est sur cette dernière qu'allait se dérouler l'intégralité de mes

1. Cela inclut la consommation des machines ainsi que du groupe froid servant à les refroidir.

expériences et de mon développement. Tout devait être minutieusement testé avant d'être déployé sur la partition principale.

Au cours de mon stage, j'ai eu besoin d'installer de nouveaux outils et de disposer d'accès privilégiés afin de prendre des mesures de consommation (plus de détails en section 3.2.2) et d'utiliser certains outils système (voir section 3.4.1). J'avais également besoin d'accès spéciaux afin de pouvoir modifier le comportement de l'ordonnanceur de travaux (de *jobs*) soumis par les utilisateurs : **SLURM**. Cela s'est avéré nécessaire afin de mettre en place le système détaillé dans la section 3.7.

Côté matériel, j'avais à disposition 5 machines :

- **miriel087** et **miriel088** qui possèdent un processeur **2x 12-core Intel Haswell** et 128 Go de RAM. Les machines du groupe **miriel** sont en fin de vie et sont souvent instables ;
- **bora044** qui possède un **2x 18-core Intel CascadeLake** et 192 Go de RAM. Le groupe **bora** possède une configuration standard et permet de réaliser des expériences nécessitant beaucoup de machines du même type ;
- **diablo04** est une machine **AMD** et possède un processeur **2x 32-core AMD Zen2** ainsi que 256 Go de RAM ;
- **sirocco25** est aussi une machine **AMD**. Elle possède un **2x 32-core AMD Zen3**, 512 Go de RAM et 2 cartes **NVIDIA A100**. Les machines du groupe **sirocco** sont équipées de GPU et servent principalement pour les expériences utilisant ces derniers.

Enfin, j'étais libre dans le choix des outils pour mener à bien mes expériences. Cependant, les scripts que je devais produire devaient être écrits en C, Python ou Bash. Leur code devait être soigneusement écrit et documenté afin qu'il soit facilement maintenable et extensible par les administrateurs de la plate-forme.

2.4 Objectifs

Dans le sujet du stage, les objectifs ont clairement été définis par **M. GOGLIN**. Les objectifs principaux sont :

- Étudier différentes stratégies d'économie d'énergie pour les systèmes Unix ;
- Analyser les différences de consommation sur les machines en fonction de leur activité ;
- Analyser l'impact de la fréquence du processeur sur la consommation, en particulier via les *governors* du système d'exploitation ;
- Se renseigner sur les fonctionnalités de l'ordonnanceur **SLURM** afin d'éteindre les nœuds inactifs ;
- Estimer les gains potentiels en terme d'énergie suite à l'application des différentes stratégies.

Après avoir discuté avec mon maître de stage, nous avons défini des objectifs secondaires :

- Extraire des statistiques d'utilisation avancées de la plate-forme à partir des traces **SLURM** ;
- Comparer les outils de mesure de la consommation électrique ;
- Dans le cas où **SLURM** ne permette pas d'éteindre les nœuds, implémenter un système qui remplisse cette mission.

Le travail que j'ai réalisé porte uniquement sur la consommation électrique des machines. Cependant, cela aura un impact sur le groupe froid qui les refroidit par effet de bord.

3 Travail réalisé

Dans cette section, nous discuterons de différentes techniques utilisées pour économiser de l'énergie avant de s'intéresser aux outils mis à ma disposition pour évaluer la consommation des machines. Nous verrons ensuite comment l'étude des *governors* m'a mené à m'intéresser à l'impact de la fréquence des processeurs et aux systèmes de basse consommation intégrés dans le noyau **Linux**. Enfin, nous iront en profondeur dans le cœur du sujet avec l'extinction des machines inutilisées grâce à **SLURM**.

3.1 État de l'art

Au début de mon stage, je me suis intéressé aux recherches récentes en matière d'économie d'énergie afin de dresser une liste de techniques que l'on pourrait mettre en œuvre.

3.1.1 Une politique SLURM plus verte

J'ai commencé par l'article de **Marco D'Amico et Julita Corbalan Gonzalez** ([1]) qui propose une nouvelle politique d'ordonnancement des *jobs*. Cette dernière appelée **EAMC** analyse les *jobs* arrivants et détermine la combinaison application/matériel qui minimise la consommation électrique. Les autres combinaisons sont utilisées seulement si cela permet d'améliorer le taux d'utilisation de la plate-forme de calcul.

J'ai abordé cette première approche avec mon maître de stage qui m'a expliqué qu'on ne peut pas changer les demandes matérielles des utilisateurs puisqu'ils ont besoin que leurs conditions expérimentales soient reproductibles d'une expérience à l'autre.

3.1.2 Overprovisioning

Changer la politique de **SLURM** ne me semblait pas la priorité, alors je me suis penché sur d'autres techniques.

J'ai continué mes recherches avec une étude de 2013 ([4]) dont le but était de trouver le meilleur couple (*nb_ nœuds* x *nb_ cœurs/nœuds, énergie/nœud*) pour une application donnée. Certaines combinaisons donnent plus de ressources que ce dont l'application a besoin, on appelle cela : l'*overprovision-ing*. Cette technique consiste à donner plus de ressources à un *job* mais à les faire fonctionner au ralenti afin d'économiser de l'énergie sans réduire les performances de l'application.

Afin d'observer l'efficacité de cette technique, les auteurs de l'article ont notamment étudié l'impact de 4 configurations principales sur différents types d'applications.

Les 4 configurations principales sont :

- **packed-min** (resp. **packed-max**) : répartir tous les cœurs de calcul sur le moins de machines possibles et les faire fonctionner à fréquence minimale (resp. maximale) ;
- **spread-min** (resp. **spread-max**) : prendre le plus de machines possibles puis faire une répartition équitable des cœurs de calculs et les faire fonctionner à fréquence minimale (resp. maximale).

Les auteurs ont remarqué que les applications dites *Memory-bound* (où la mémoire est le facteur limitant quant à l'efficacité de leur exécution) sont accélérées sur les configurations de type *spread*. Cela s'explique par le fait que davantage de mouvements de données peuvent être faits simultanément puisque l'ensemble des données est réparti sur plus de nœuds. À l'inverse, les applications dites *Compute-bound* (celle dont la puissance de calcul est le facteur limitant) sont plus adaptées à des configurations *packed*. En effet, ces applications dépendent de leur capacité à effectuer de longs traitements sur leur données et profitent d'avoir plus de cœurs/nœud pour paralléliser les traitements. Malheureusement, ce constat n'est pas vrai pour toutes les applications et certaines se comportent mieux avec des combinaisons différentes des 4 principales.

J'en ai conclu que l'*overprovisioning* pouvait être envisagé sur **PlaFRIM** et j'en ai parlé à **M. GOGLIN**. Après discussion, il ne serait possible de le mettre en place que sur les *jobs* sans demande de cœurs de calcul précise. J'ai ensuite réfléchi à son application et cela conduirait à avoir un système capable de détecter quelle serait la meilleure combinaison (*nb_ nœuds* x *nb_ cœurs/nœuds, énergie/nœud*) pour chaque nouvelle application dans la file. Cela faisait partie du travail futur des auteurs de l'article de 2013 et il n'était pas possible d'implémenter un tel système sur **PlaFRIM**. En effet, la complexité de la mise en place de l'*overprovisioning* comparé aux gains limités que cette technique engendre [2] m'a poussé à chercher d'autres techniques d'économie d'énergie.

2. L'*overprovisioning* n'est pas compatible sur tous les *jobs* et fait l'hypothèse d'un bon passage à l'échelle des

3.1.3 Gérer la consommation du réseau

Quand on pense à réduire la consommation électrique d'un centre de calcul, on parle souvent d'éteindre les machines ou de diminuer l'énergie allouée aux composants mais la gestion des liens réseaux n'est pas souvent abordée.

En 2015, un article de **Saravanan et al.** s'intéressait aux nouveaux protocoles de basse consommation des liens réseaux. Ces derniers, intégrés dans Ethernet depuis 2014 font partie du standard *Energy Efficient Ethernet* qui est applicable aux liens allant jusqu'à 400 Gb/s.

Les deux protocoles basse consommation sont :

— **Deep-Sleep** : après que le lien réseau soit inutilisé pendant un temps $t1$, l'endormir afin de réduire sa consommation de 90%. Le temps de réveil du lien pour qu'il soit de nouveau opérationnel est long.

— **Fast-Wake** : après que le lien soit inutilisé pendant un temps $t2$, le passer dans un mode de basse consommation afin de réduire sa consommation de 40%. Son temps de réveil est bien plus court.

Dans les centres de calcul, les interconnexions des nœuds sont allumées sans interruption alors qu'elles ne sont pas beaucoup sollicitées (puisque limiter les communications à leur minimum est une des clés vers l'amélioration des performances). Par conséquent, il existe des gains conséquents en terme d'énergie si les liens inutilisés sont endormis.

Les auteurs de l'article se sont intéressés à trouver les meilleurs temps $t1$ et $t2$ afin de réduire la consommation au maximum tout en ayant un faible impact sur les performances des applications. Ils sont venus à la conclusion que les deux variables doivent être calibrées spécifiquement pour chaque application sinon les performances chutent et les gains en énergie s'en trouvent diminués ([8]).

Par conséquent, endormir les liens réseaux inutilisés me semblait intéressant mais devoir calibrer manuellement les deux paramètre $t1$ et $t2$ pour chaque application rendait cette solution inenvisageable. J'ai ensuite recherché le travail futur des auteurs et j'ai trouvé un de leurs articles paru en 2018 ([7]) qui propose un système de paramétrage automatique des paramètres qui garantit de bonnes économies d'énergie ainsi qu'un impact faible sur les performances des applications.

Une fois la contrainte de devoir calibrer manuellement les paramètres retirée, j'ai cherché comment mettre en place cette technique. Malheureusement, je me suis rendu compte que tous les équipements réseau de **PlaFRIM** devaient implémenter le standard *Energy Efficient Ethernet* afin de supporter les protocoles **Deep-Sleep** et **Fast-Wake**. Ce n'était pas le cas et par conséquent, cette solution ne pouvait pas être mise en place.

Pour conclure ces lectures d'articles, j'ai appris qu'il était possible d'influer la politique d'ordonnancement des *jobs* afin de favoriser les allocations de ressources optimales aux applications. J'ai compris qu'il était possible de faire de l'*overprovisioning* afin de réduire la consommation électrique mais que cette technique ainsi que la précédente n'engendreraient pas de gains importants et que le but de **PlaFRIM** ne permettait pas la mise en place de ces techniques. Enfin, je me suis intéressé à la réduction de la consommation au niveau des liens réseaux avant de me rendre compte que les équipements de la plate-forme ne supportaient pas la mise en veille des liens.

3.2 Analyse des outils de mesure de la consommation électrique

Après mes recherches, je me suis recentré sur les pistes proposées dans le sujet du stage. Cependant, pour observer leur efficacité, il me fallait des outils fiables pour mesurer la consommation électrique des machines. Cette section a pour but de présenter les différents outils à ma disposition et de les comparer.

applications.

3.2.1 Programmes de stress

Afin de comparer les outils de mesure, il me fallait faire varier la consommation des machines. Pour cela, des programmes de stress étaient nécessaire. Ces programmes servent à augmenter la charge de la machine et par conséquent leur consommation.

J'ai trouvé de nombreux programmes mais seuls trois d'entre eux sortaient du lot à mon sens :

— **stream** : un programme de stress principalement focalisé sur la mémoire ;
— **stress** : une référence en la matière, il est possible de choisir facilement le nombre de cœurs à stresser, quelles opérations font les *threads*, si l'on veut stresser la mémoire, etc ;
— **s-tui** : couplé avec **stress**, ce programme permet de visualiser la température des cœurs, leur consommation, etc. Cet outil sera plus détaillé en section 3.4.1.

J'ai également développé un petit programme semblable à **stress** appelé **melt** qui consiste à faire tourner tous les cœurs de la machine à 100% de leur capacité. Ce programme était destiné à remplacer **stress** si je ne pouvais pas l'installer.

3.2.2 Les différents outils

Mon maître de stage m'a présenté les différents outils qui étaient à ma disposition : **ipmitool**, les PDU avec **DCE** et **energy_scope**. Le but de cette section est de les présenter succinctement.

1. **PDU et DCE**
   Les PDU (*Power Distribution Unit*) sont les prises sur lesquelles les machines sont branchées. La particularité de ces prises est qu'il est possible de récupérer la consommation électrique d'une prise en particulier ou d'un groupe de prises pour les PDU dits « non manageable ».

   Les données de ces prises sont stockées dans une base consultable grâce à un outil donné par le fournisseur des PDU. Cet outil est **Data Center Expert** (**DCE**). Il est indispensable d'avoir un compte sur le serveur où sont stockées les données pour utiliser l'outil. Quelques problèmes sont survenus quand j'ai essayé d'installer **DCE** sur mon ordinateur de travail fonctionnant sous **Linux**. Étant bloqué, mon encadrant m'a donné un accès distant à une machine virtuelle sous **Windows** où l'outil est installé, ce qui m'a permis de m'en servir.

   DCE permet d'interroger les capteurs présents sur chaque PDU afin de générer des rapports. Sur la figure 1, on peut voir que l'on peut générer un rapport pour une période spécifique ou relative à l'heure actuelle. On peut choisir la (ou les) prise(s) et le type de capteur à interroger.

   Il est important de faire le lien entre l'identifiant de la prise et la machine correspondante. J'ai pu me rendre en salle machine pour identifier les prises des quelques machines mises à ma disposition. Une fois les prises identifiées, j'ai pu générer des rapports. Ces derniers sont présentés sous forme de graphiques dont un exemple est visible sur la figure 2.

   Voulant savoir comment sont prises les mesures, j'ai commencé par fouiller dans la documentation du logiciel. J'avais remarqué que les mesures étaient prises toutes les 3 minutes et n'ayant pas trouvé plus de réponses dans la documentation, j'ai effectué quelques expériences. Par exemple, j'ai stressé la machine pendant 90% de la période de 3 minutes puis je l'ai laissé inactive pour les 10% restants. En voyant sur le graphique que le point de mesure se trouvait au plus bas et après avoir répété d'autres expériences similaires, j'ai compris que les PDU ne prenaient que des mesures instantanées toutes les 3 minutes.

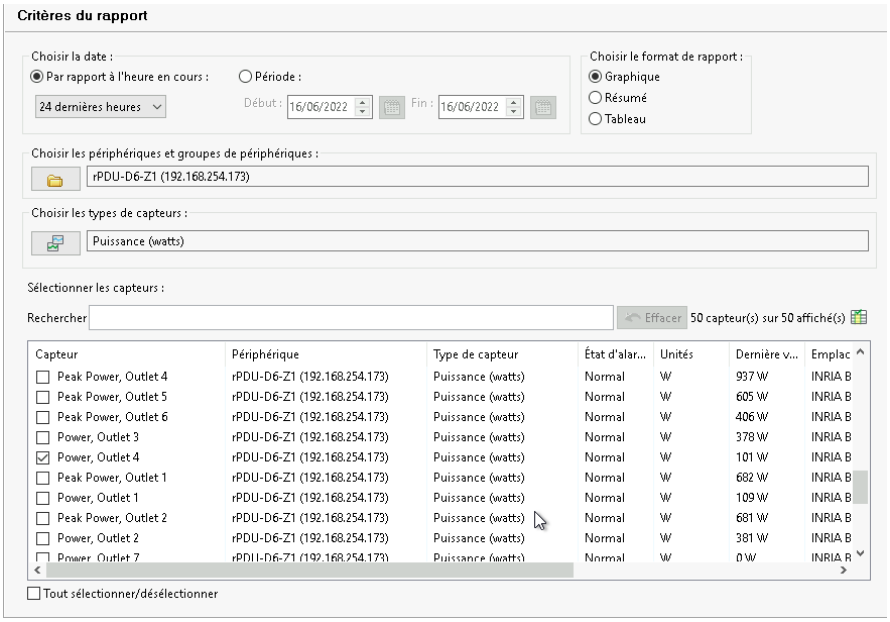

FIGURE 1 – Menu de création de rapports sur **DCE**

FIGURE 2 – Consommation électrique de C6-Z1,22 (**bora044**) le 21 Avril 2022 de 14 à 15 heures

2. **ipmitool dcmi/sensor**
   Ce premier outil s'appuie sur le système IPMI (*Intelligent Platform Management Interface*). Ce dernier est composé d'un contrôleur principal : le BMC (*baseboard management controller*). C'est par ce contrôleur que le système communique avec les capteurs de la machine, notamment ceux de température et de consommation électrique. Le BMC est aussi capable d'allumer et d'éteindre la machine via une connexion réseau. Les paquets IPMI utilisent le protocole *IPMI Remote Management Protocol* qui transitent en mode UDP sur le port 623.
   Pour interroger le système IPMI, il est possible d'utiliser la commande **ipmitool**. Cette dernière nécessite les privilèges *root*. Il existe 2 façons de mesurer la consommation électrique avec **ipmitool** :
   — **ipmitool sensor** :

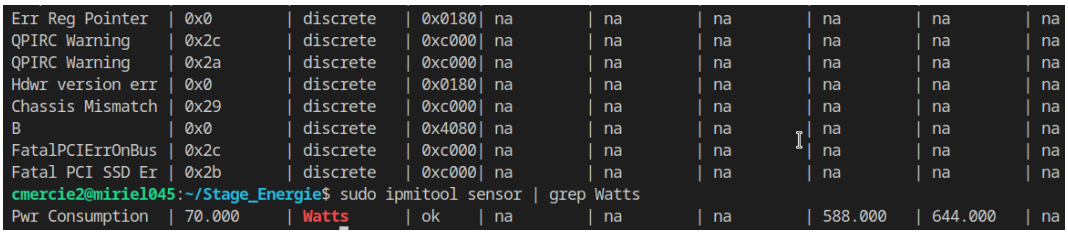

FIGURE 3 – Sortie de la commande **ipmitool sensor**

Comme on peut le voir sur la figure 3, la commande **ipmitool sensor** interroge tous les capteurs de la machine. Il est alors important de filtrer la sortie de la commande afin de ne récupérer que le capteur de consommation électrique.

— **ipmitool dcmi** :

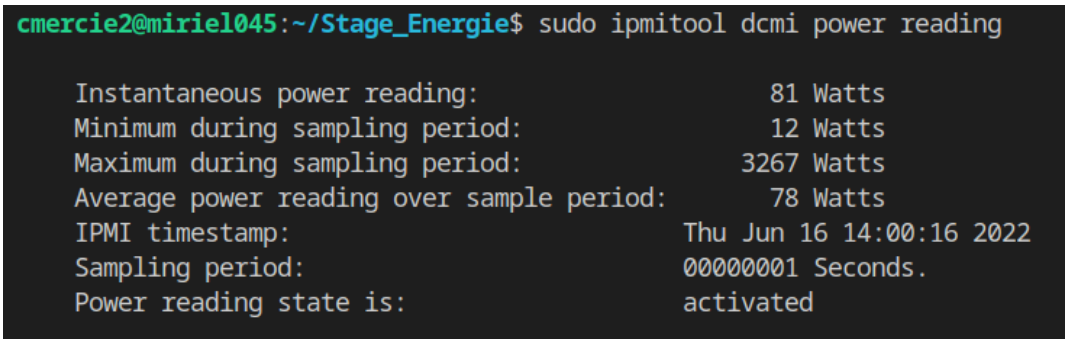

FIGURE 4 – Sortie de la commande **ipmitool dcmi power reading**

Ici, nous nous intéressons à la partie DCMI (*Data Center Manageability Interface*) d'**ipmitool**, plus particulièrement à la section *power*. Dans cette section, il est, par exemple, possible de fixer une limite de consommation à la machine mais ce qui nous intéresse est l'option *reading* qui permet de récupérer la consommation instantanée de la machine.

3. **energy_scope**

   energy_scope ([3]) est un outil développé dans le centre, principalement par **Hervé MATHIEU**. Ce logiciel permet de récupérer la consommation électrique de différents composants de la machine tels que le(s) CPU(s), le(s) GPU(s) et la mémoire. Il se base sur des registres privilégiés du processeur (les Model-specific register (MSR)) afin de mesurer la consommation instantanée de la machine. Seul un accès en lecture à ces registres est nécessaire pour le bon fonctionnement de l'outil.

   Pour utiliser **energy_scope**, il suffit de le lancer en lui donnant en paramètre l'application dont on veux mesurer la consommation. **energy_scope** génère alors des traces qu'il faut importer sur le site dédié à l'outil. Le serveur nous permet ensuite de visualiser les traces.

   L'interface présentée sur la figure 5 comporte une schématisation de la machine en haut à gauche et une courbe présentant l'évolution de la consommation électrique de la machine au fil de l'exécution de l'application en bas à gauche. Il est possible de zoomer sur une partie du graphe afin de lire un point précis de la courbe.

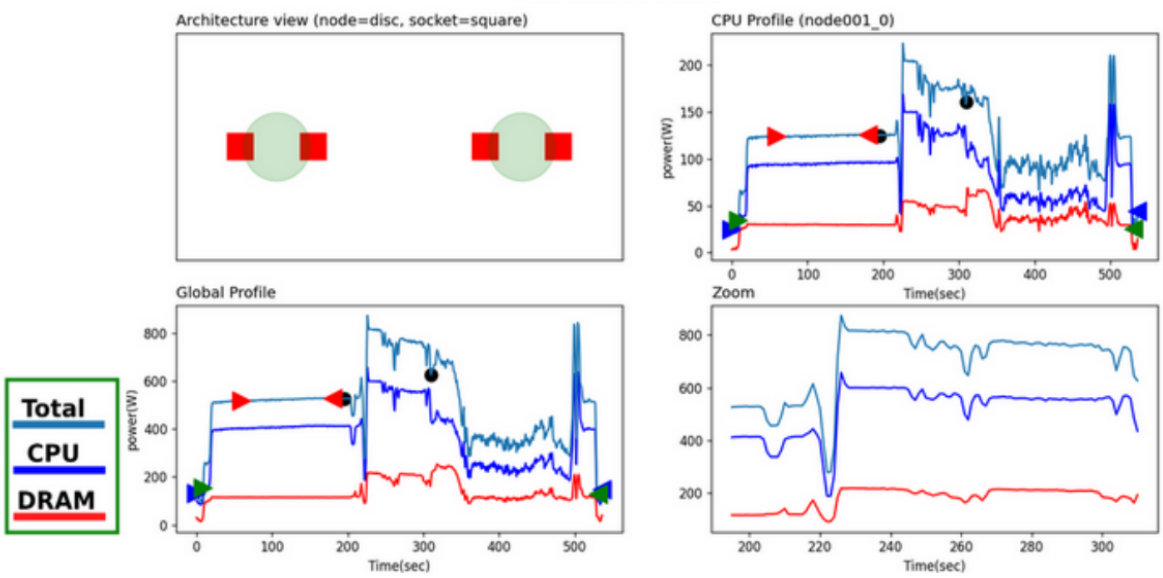

FIGURE 5 – Interface d'**energy_scope**

3.2.3 Expériences avec les outils

Après avoir pris en main tous les outils à ma disposition, il me fallait les comparer. **energy_scope** et les commandes **ipmitool** permettent de récupérer la consommation instantanée de la machine tandis que les PDU ne récupèrent cette information qu'une fois toutes les 3 minutes. Afin que mes expériences soient fiables, j'ai décidé de prendre 5 points de comparaison par expérience (ce qui représente des périodes de 15 minutes).

Lors de ma phase expérimentale, je pensais faire d'une pierre deux coups. Je pouvais faire une étude comparative des outils de mesure de la consommation électrique et observer l'impact des *governors* sur les machines. Par la même occasion, je pouvais observer si mes résultats étaient différents entre **intel** (machine **bora044**) et **AMD** (machine **diablo04**).

Plus précisément, pour chaque machine, j'ai effectué 5 prises de mesures pour chaque *governor* disponible (**performance** et **powersave** plus **conservative** pour **diablo04**).

Au début de mes expériences, je faisais en sorte de prendre les mesures de consommation avec les deux commandes **ipmitool** et les PDU en même temps. Comme les outils donnaient des résultats différents les uns des autres, je me suis dit que prendre les mesures en simultané pouvait perturber leur bon fonctionnement. J'ai donc fait de nouvelles expériences en utilisant les PDU et les deux commandes **ipmitool** en différé. Malheureusement, comme on peut le voir sur la figure 6, il n'y a aucune différence entre les mesures simultanées et en différé. Nous pouvons aussi voir que **ipmitool sensor** nous donne des mesures avec une mauvaise granularité. J'ai pu observer au cours de mes expériences que cet outil renvoyait des mesures qui étaient toujours des multiples de 5.

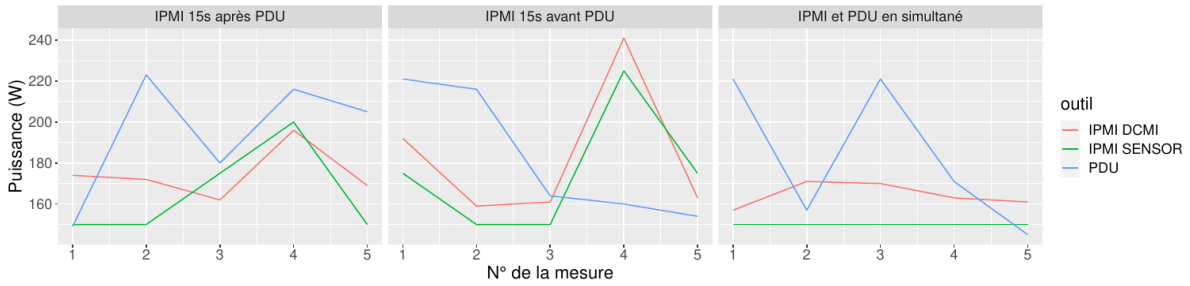

FIGURE 6 – Comparaison des outils de mesure sur **diablo04** à 0% CPU (*governor* : **performance**)

Le problème n'étant pas réglé, j'ai essayé d'utiliser les deux commandes **ipmitool** en différé pour voir si elles n'interféraient pas entre elles. Cependant, afin de pouvoir comparer les résultats, je continuais à mesurer la consommation avec les PDU : en simultané et en différé afin de voir si ces derniers ne causaient pas aussi des interférences. Avant de me lancer dans d'autres expériences, j'ai pris le temps de développer un script PYTHON afin d'automatiser la prise de mesure et pour m'assurer qu'elles soient prises au bon moment. J'ai également développé un script R pour générer facilement le graphique précédent ainsi que ceux que vous verrez par la suite.

Sur la figure 7, nous pouvons remarquer que les mesures des deux outils sont bien différentes. Bien que **ipmitool dcmi** soit utilisé seul et en différé par rapport aux PDU, les valeurs ne concordent pas.

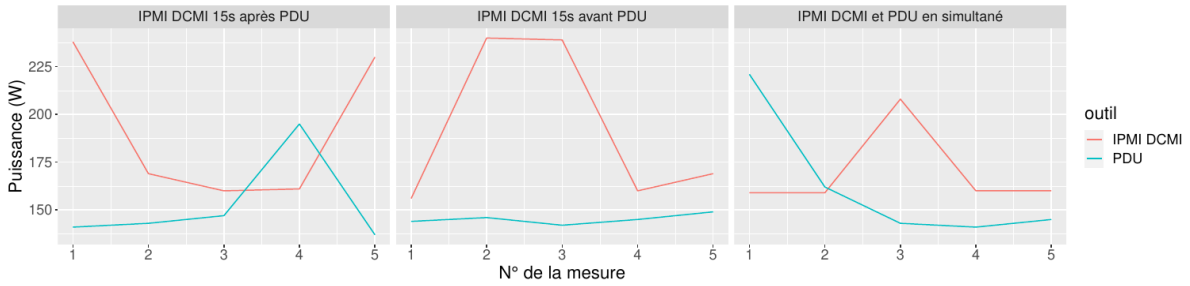

FIGURE 7 – Comparaison des mesures prises par **ipmitool dcmi** et par les PDU sur **diablo04** à 0% CPU (*governor* : **performance**)

Pour la suite de mes expériences, j'ai décidé de soumettre **energy_scope** aux mêmes tests que les autres outils. Les résultats présentés sur la figure 8 m'ont fait comprendre qu'**energy_scope** sous-estimait la consommation électrique (sûrement car il n'a pas accès à tous les composants de la machine) mais qu'il présentait des résultats cohérents par rapport aux PDU.

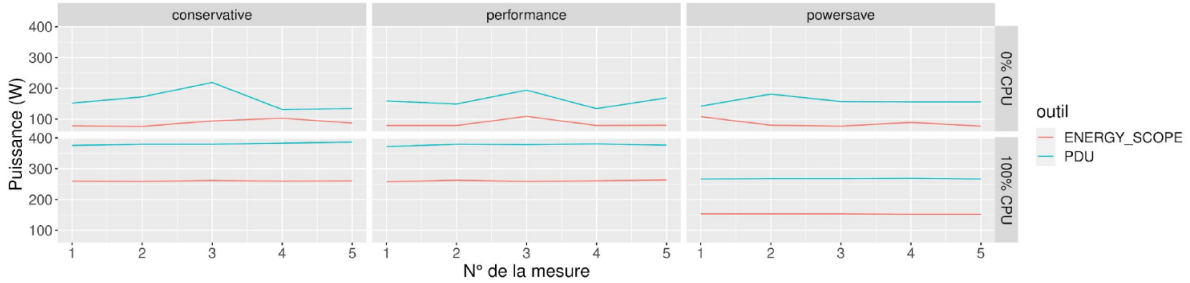

FIGURE 8 – Comparaison **energy_scope** / PDU sur **diablo04** en fonction de l'utilisation du CPU et du *governor*

À la fin de mon premier mois de stage, j'ai pu dresser un tableau récapitulatif quant aux caractéristiques des outils. J'ai également généré des graphiques comparatifs de ces derniers.

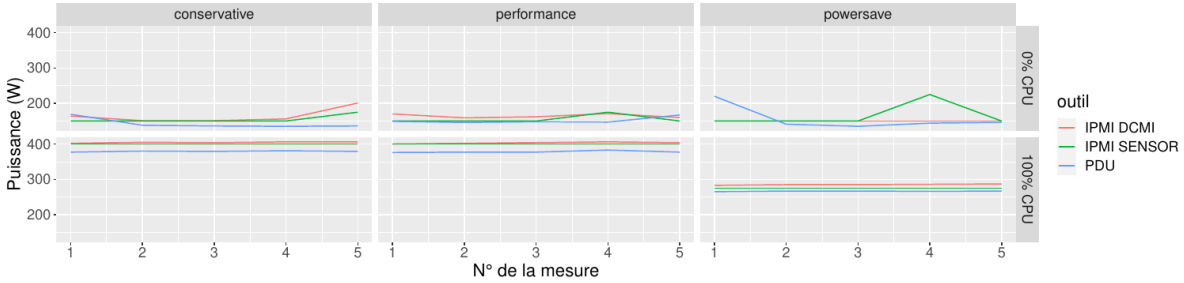

FIGURE 9 – Comparaison de la consommation électrique sur **diablo04** en fonction de l'utilisation du CPU et du *governor*

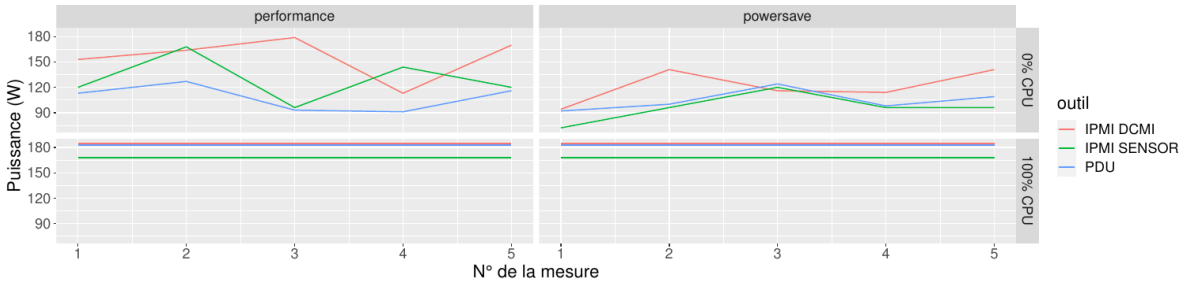

FIGURE 10 – Comparaison de la consommation électrique sur **bora044** en fonction de l'utilisation du CPU et du *governor*

Sur la figure 10, avec une utilisation du processeur à 0% et sur la partie 0% CPU en **powersave** de la figure 9, nous pouvons voir que l'ensemble des mesures est instable. À l'inverse, quand l'utilisation du processeur est à son maximum, les 3 outils s'accordent sur quasiment les mêmes résultats. J'ai ensuite vérifié la consommation des machines inactives avant même que mon stage ne commence et j'ai pu constater ces mêmes instabilités (plus de détails en section 3.2.4). J'ai donc décidé de ne pas prendre en compte les mesures prises lorsque les machines étaient inactives pour comparer les outils entre eux.

Après avoir fait toutes ces expériences, j'ai pu présenter un rapport aux administrateurs comparant les différents outils ainsi qu'une première discussion sur les *governors*.

Pour résumé, voici un tableau de comparaison des différents outils :

Outil \ Critère	Fréquence des mesures	Granularité des mesures	Niveau de privilèges requis	Vision des composants de la machine
PDU	Toutes les 3 minutes	Au Watt près	Compte d'accès à la base de données PDU	Accès direct à l'alimentation
IPMI DCMI	Mesure instantanée	Au Watt près	Accès root	Tous les composants
IPMI SENSOR	Mesure instantanée	Par multiple de 5 Watt.	Accès root	Tous les composants
ENERGY_SCOPE	Toutes les 0,1 secondes	Au Watt près	Accès en lecture des MSR	CPU, GPU, DRAM

FIGURE 11 – Tableau comparatif des 4 outils de mesure de la consommation électrique

Suite à ces résultats d'expérience, nous avons décidé de ne mesurer la consommation électrique qu'avec les PDU et **ipmitool dcmi** pour le reste de mon stage. Ce choix fut motivé par leur précision, leur facilité d'utilisation (interface graphique pour exploiter les PDU / commande revoyant la consommation instantanée pour **ipmitool dcmi**) et le fait que je dispose d'accès suffisants pour les utiliser.

Par la même occasion, ces expériences ont mis en valeur des comportements étonnants que je devais comprendre.

3.2.4 Comportement inattendu sur bora044

Sur la figure 9, nous avons pu voir que le *governor* **powersave** permet effectivement de réduire la consommation de **diablo04** de 100 W quand la machine est très chargée. Étonnamment, ce n'est pas le cas pour **bora044** (voir figure 10). Au début, j'ai pensé que comme l'une d'elle est **intel** (**bora044**) est l'autre est **AMD**, cela pouvait s'expliquer par la différence de *driver*[3].

Nous pouvons aussi voir sur la figure 10 que la consommation électrique de la machine n'est pas stable quand elle est inactive. Je suis donc allé mesurer l'importance de ces instabilités et me suis rendu compte de l'amplitude des pics de consommation présents sur la machine en période d'inactivité (voir figure 12).

3. J'ai exploré plus en profondeur le fonctionnement de ces *drivers* en section 3.3.

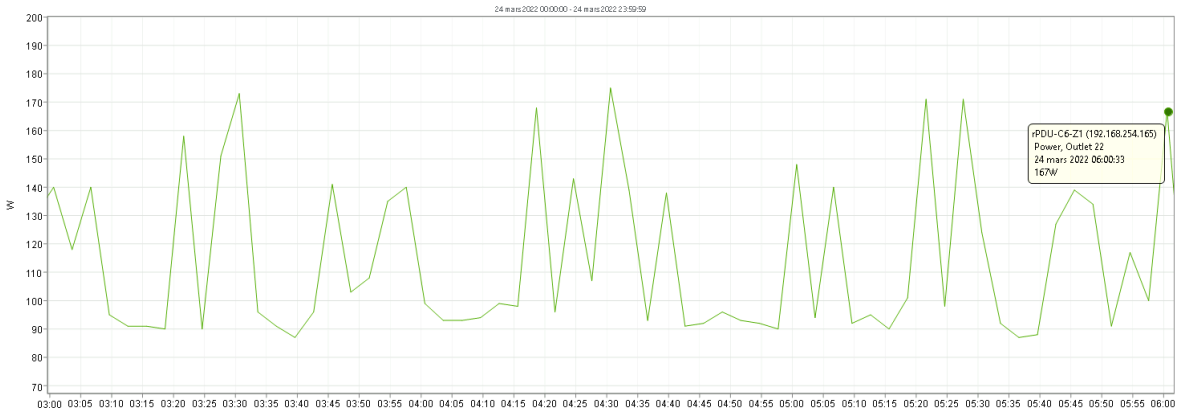

FIGURE 12 – Pics de consommation sur **bora044** lors d'une période d'inactivité (variance des pics de 80 W)

Le premier mois de stage touchait à sa fin et j'avais dorénavant deux nouveaux objectifs à remplir : analyser les *governors* pour comprendre pourquoi ils étaient sans effet sur **bora044** et essayer de trouver la source de ces anomalies.

3.3 Governors

Après la comparaison des différents outils, **M. GOGLIN** m'a donné un nouvel objectif pour le deuxième mois de stage : comprendre l'impact de la fréquence CPU sur la consommation électrique. Pour modifier facilement la fréquence, **Linux** expose différents *governors* (**performance**, **powersave**, etc). Afin de bien comprendre comment ils étaient gérés, je me suis plongé dans la documentation du noyau **Linux**.

3.3.1 La gestion des *P-states*

La partie de la documentation traitant de la gestion des *governors* ([12]) commençait par décrire ce que sont les *P-states* (Operating Performance Points).

Un *P-state* est une configuration (fréquence,voltage). Plus il est élevé, plus la fréquence d'horloge du CPU est haute ce qui entraîne de meilleures performances au prix d'une plus grande consommation électrique. Comme les processeurs modernes peuvent fonctionner dans un large spectre de fréquences, il est possible de changer le *P-state* d'un CPU.

C'est le rôle du sous-système **CPUFreq** de gérer le changement de *P-state*. Ce dernier est formé de 3 composants :

- **le noyau (core)** qui contient la base de code en commun pour les deux autres composants ainsi que l'interface utilisateur ;
- **le driver** qui sert d'intermédiaire entre le CPU et le *governor*. Il fournit au *governor* la liste des *P-states* disponibles ainsi que des informations propres au CPU. C'est aussi ce composant qui commande au CPU de changer de *P-state* ;
- **le *governor*** qui implémente un algorithme servant à estimer la fréquence CPU la plus adaptée à un instant donné. Chaque *governor* représente un algorithme.

En image, cela donne :

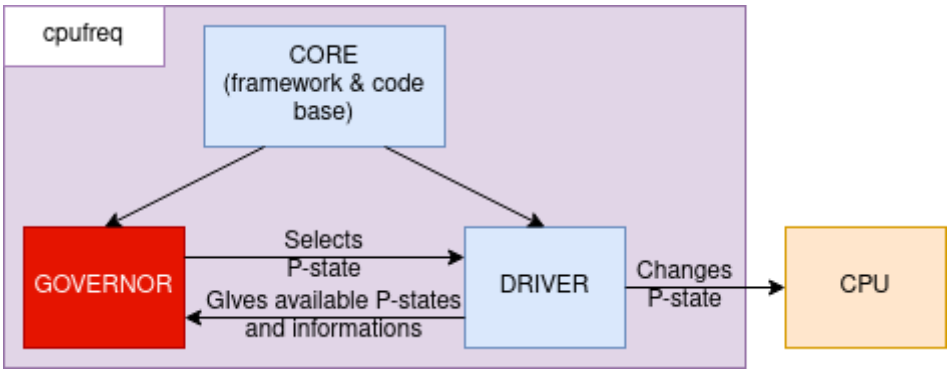

FIGURE 13 – Schéma de fonctionnement de **CPUFreq**

Il existe deux *drivers* principaux : **acpi-cpufreq** et **intel_pstate**. Les deux sections suivantes détaillerons leurs caractéristiques.

3.3.2 acpi-cpufreq

Après lecture de la documentation de **acpi-cpufreq** ([12]), j'ai compris que ce *driver* est utilisé sur les anciennes machines **intel** (architectures anciennes à SANDY BRIDGE[4]) et les machines **AMD**.

Il expose de nombreux attributs que l'on peut trouver dans le dossier **/sys/devices/system/cpu/cpu**x**/cpufreq** (où x est le numéro du cœur concerné). Les attributs les plus importants sont :

- **scaling_max_freq** qui représente la fréquence maximale du CPU ;
- **scaling_min_freq** qui représente la fréquence minimale du CPU ;
- **scaling_cur_freq** qui représente la fréquence actuelle du CPU ;
- **scaling_driver** qui contient le nom du *driver* actuellement utilisé ;
- **scaling_available_governors** qui contient la liste des *governors* disponibles sur le système ;
- **scaling_governor** qui contient le nom du *governor* actuellement utilisé.

acpi-cpufreq fournit aussi des *governors* génériques. Les 3 qui reviennent le plus sur **PlaFRIM** sont :

- **performance** : le CPU est bloqué à sa fréquence maximale autant que possible ;
- **powersave** : le CPU est bloqué à sa fréquence minimale autant que possible ;
- **conservative** : la fréquence CPU s'adapte en fonction de la charge du système.

3.3.3 intel_pstate

Ce *driver* est présent sur toutes les machines **intel** récentes. Il ne peut pas être déchargé (mais il peut être désactivé) et doit être configuré soit par la ligne de commande du noyau soit par le système de fichiers. Ensuite, ce *driver* s'appuie sur une technologie récente : HWP (hardware-managed *P-states*). Cette technologie permet au processeur de choisir lui-même son *P-state*. Enfin, **intel_pstate** fournit ses propres *governors* et dispose de différents modes de fonctionnement :

- **Mode actif** (par défaut) : **intel_pstate** outrepasse la couche *governor* et fournit ses propres algorithmes de sélection de *P-state*. Ces deux algorithmes sont **performance** et **powersave**. Ils portent le même nom que les *governors* présents dans **acpi-cpufreq** mais ne font pas la même chose ! De plus, leur fonctionnement change selon l'état d'HWP :
  - **si HWP est activé** (le processeur supporte HWP et ce dernier est activé), le processeur choisit son *P-state*. Cependant, **intel_pstate** peut influencer ce choix grâce à un biais. Ce dernier, représenté par un entier, permet d'orienter le processeur vers les performances

4. SANDY BRIDGE date de Janvier 2011.

ou vers l'économie d'énergie (en réduisant sa fréquence). Le biais d'**intel_pstate**, appelé **Energy-Performance Preference** (EPP) ou **Energy-Performance Bias** (EPB), peut prendre différentes valeurs parmi celles présentes sur la figure ci-dessous :

EPB value	String
0	performance
4	balance-performance
6	normal, default
8	balance-power
15	power

FIGURE 14 – Signification de chaque valeur du biais d'**intel_pstate**

Quand HWP est activé, le *governor* **performance** consiste à mettre le biais à 0 forçant le CPU à ne choisir que les plus hautes fréquences. De son côté, le *governor* **powersave** place le biais à la valeur définie par le système de fichiers (**/sys/devices/system/cpu/cpu**x**/cpufreq/energy_performance_preference** pour le cœur n°x).

— **si HWP est désactivé**, **intel_pstate** sélectionne l'algorithme à utiliser entre **performance** et **powersave**. Par défaut, **performance** est utilisé si l'option de configuration du noyau **CONFIG_CPU_FREQ_DEFAULT_GOV_PERFORMANCE** est présent.
En **performance**, **intel_pstate** cherche le *P-state* le plus haut qu'il peut utiliser et le sélectionne. En **powersave**, le *driver* utilise les registres privilégiés du CPU afin de déterminer la charge du processeur et d'utiliser un *P-state* adapté.

— **Mode passif** : c'est le mode utilisé quand l'option **intel_pstate=passive** est mis dans la ligne de commande du noyau [5]. Dans ce mode, **intel_pstate** fonctionne comme un *driver* classique. Il n'outrepasse pas la couche *governor* et transmet les informations du CPU à **acpi-cpufreq**. De ce fait, tous les *governors* d'**acpi-cpufreq** sont disponibles et ce sont eux qui sélectionnent le *P-state* d'après les informations données par **intel_pstate**. Ces derniers feront appels à **intel_pstate** pour changer le *P-state* du CPU.

3.3.4 Expériences sur les *drivers*

Après avoir bien compris le fonctionnement des *governors*, j'ai eu l'occasion de discuter avec **Amina GUERMOUCHE**, chercheuse chez **Inria** dans le domaine de l'énergie. Elle m'a conseillé de mesurer la fréquence du processeur en même temps que mes relevés de consommation. Je lui ai expliqué qu'il semblerait que les *governors* n'aient aucun impact sur **bora044** (une machine **intel** récente utilisant **intel_pstate**) et elle m'a conseillé de vérifier si le programme de stress que j'utilise ne fait pas de vectorisation. En effet, quand un programme fait de la vectorisation, le CPU voit sa fréquence diminuée afin d'éviter la surchauffe. Elle m'a aussi conseillé de voir si le comportement est le même en désactivant **intel_pstate**.

Pour mes tests sur les *drivers*, j'ai utilisé le programme de stress **melt** dont je possède les sources afin de m'assurer qu'aucune vectorisation n'est faite. Afin que mes résultats soient les plus représentatifs possibles, j'ai utilisé les PDU pour relever la consommation sur une période d'une heure.

5. L'option qui désactive HWP doit être présente également pour entrer dans ce mode.

J'ai commencé par tester chaque biais sur **bora044** sous un stress maximum (100% d'utilisation CPU) et j'ai remarqué que la consommation restait la même. Je me suis dit qu'HWP pourrait être la cause et j'ai demandé sa désactivation par les administrateurs (un accès *root* est nécessaire pour modifier la ligne de commande du noyau). Après avoir refait les expériences avec **intel_pstate** dans le mode actif mais sans HWP, j'ai remarqué que les résultats étaient les mêmes : la consommation reste constante à environ **183 W**.

Ne comprenant pas ces résultats, j'ai effectué la même expérience sur l'ordinateur portable fourni par **Inria**. Après avoir changé le biais d'**intel_pstate**, j'ai remarqué une baisse drastique de la consommation et des performances. Par conséquent, quelque chose sur **bora044** empêchait les *governors* et/ou le biais d'**intel_pstate** de fonctionner correctement.

3.3.5 Expériences sur les *governors*

Puisque le biais ne semblait pas avoir d'influence sur la machine **intel**, je me suis recentré sur les *governors*. Afin d'évaluer l'impact de ces derniers, j'ai simulé une journée de travail sur **PlaFRIM** sous différentes politiques. Comme la plate-forme est utilisé à 30%, j'ai décidé de faire mes expériences sur des périodes de 30 minutes dont 20 sont passées à 0% CPU et le reste à pleine puissance (soit 33% d'activité). Chaque expérience dure 6 heures afin d'obtenir 12 périodes.

Le *governor* n'est changé qu'en période d'inactivité puisque l'on veut conserver les performances pendant les *jobs*. À ce moment là, je voulais aussi vérifier si changer le *governor* revenait à changer la fréquence du processeur. J'ai donc fixé manuellement la fréquence de **bora044** pour une de mes expériences. Plus précisément, en période d'activité, je l'ai fixé à la fréquence maximale (2.6 GHz) puis à la fréquence minimale en période d'inactivité.

La figure 15 résume l'ensemble des résultats de mes expériences.

Machine	Governor & biais (100% CPU)	Governor & biais (0% CPU)	Consommation minimale (W)	Consommation maximale (W)	Consommation moyenne (W)
bora044	performance	performance	84	185	134
		powersave	88	184	129
	2.6 GHz	1 GHz	86	185	130
diablo04	performance	performance	135	380	225
		powersave	126	381	212

FIGURE 15 – Comparaison de la consommation électrique sur **bora044** et **diablo04** en fonction du *governor*

En conclusion de ces expériences, on remarque que changer le *governor* en **powersave** sur **bora044** [6] n'a aucun effet signifiant (gain moyen de 5 W). De même, changer la fréquence manuellement revient à faire le même travail que le *governor* et mène au même résultat. Cependant, comme je soupçonnait que **bora044** avait un problème, j'ai aussi effectué mes expériences sur **diablo04** où je savais que les *governors* auraient un impact (rappel en figure 9). J'ai été surpris de voir que le gain moyen sur cette machine est de 13 W. En réalité, ce faible impact s'explique par le fait que le *governor* n'est changé qu'en période d'inactivité du CPU. Or, nous avons pu voir sur les figures 9 et 10 que les *governors* n'ont quasiment aucun impact à 0% d'utilisation CPU.

3.4 Fréquence CPU : un premier levier pour réduire la consommation

En parallèle de mes tests sur les *drivers* et les *governors*, j'ai suivi le conseil de **Mme. GUERMOUCHE** et je me suis intéressé à la fréquence CPU.

6. Le *driver* utilisé est **intel_pstate** en mode actif. HWP est activé afin que le biais ait un sens.

3.4.1 Fréquence CPU : recherche d'outils de mesure

Dans la section précédente, j'ai essayé de fixer manuellement la fréquence CPU sur **bora044**. Cependant, je n'étais pas convaincu que mes changements aient réellement fonctionné. Je me suis alors mis à chercher différents moyens de mesurer la fréquence du processeur et j'ai trouvé les méthodes suivantes :

1. **filtrer la sortie de lscpu sur « MHz »**, ce qui me donne la moyenne des fréquences actuelles des différents cœurs du processeur ;
2. **lire depuis le système de fichiers** permet d'obtenir la fréquence actuelle de chaque cœur que ce soit dans **/proc/cpuinfo** ou dans **/sys/devices/system/cpu/cpu**x**/cpufreq/scaling_cur_freq** ;
3. **utiliser s-tui** qui est un outil graphique permettant de surveiller de nombreuses métriques telles que la température et la fréquence CPU (les compteurs de fréquence se trouvent en bas à gauche sur la figure 16).

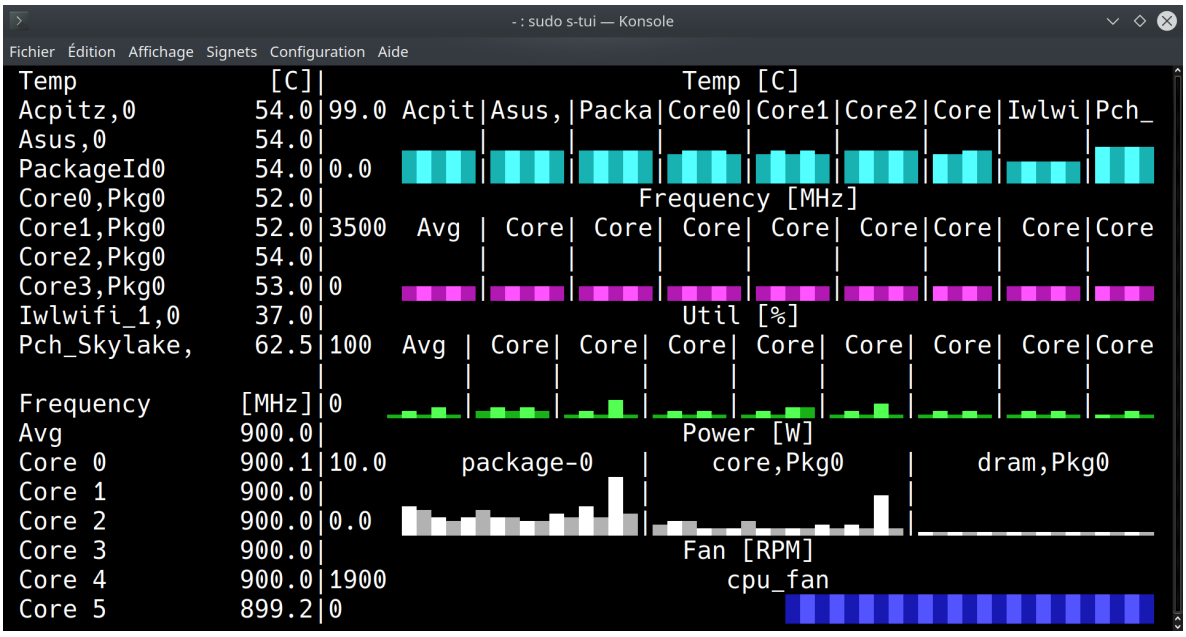

FIGURE 16 – Interface graphique de l'outil **s-tui**

Après mes recherches, j'ai à nouveau mesuré la fréquence de **bora044** lors d'une activité intense (via la commande **stress**). Malgré tout, les nouveaux outils renvoyaient une fréquence de 1 GHz alors que je l'avais fixé à 2.6 GHz. Pensant que ces derniers fonctionnaient mal, j'ai essayé de calculer la fréquence avec l'aide de mon maître de stage.

Pour ce faire, nous voulions compter le nombre de cycles CPU utilisé pour l'exécution d'un programme et diviser ce nombre par le temps d'exécution. Afin d'y arriver, nous avons essayé différentes approches :

— **faire appel à rdtsc()**, une instruction qui renvoie le compteur de cycles. Malheureusement, les résultats renvoyés étaient incorrects. En effet, rien n'empêchait le processeur de préempter le programme entre deux appels à **rdtsc** ce qui faussait nos résultats ;

— **faire appel à cpuid()**. Comme je n'arrivais pas à m'en servir, j'ai cherché sa définition et je me suis rendu compte que cette instruction interroge les mêmes registres qu'**rdtsc** ce qui nous mène au même résultat ;

— **compter les cycles d'un programme x86 simple**. Pour cela, j'ai écrit un programme C qui ne consiste qu'en une boucle *for* déroulée le plus possible (999 incrémentations dans le corps de boucle). J'ai ensuite regardé le code généré afin de m'assurer que la boucle dont j'allais mesurer les cycles comportait bien 1000 instructions, chacune s'exécutant pendant 1 cycle. Ce fut bien le cas, les 999 incrémentations furent traduites en 999 instructions **addq** et une instruction **jump** était présente à la fin pour effectuer le contrôle de la boucle. Je pensais alors pouvoir calculer simplement le nombre de cycles utilisé pour exécuter la boucle et diviser ce nombre par le temps nécessaire à son exécution afin de calculer la fréquence du CPU. Malheureusement, les mêmes problèmes que précédemment sont survenus ce qui faussait mes résultats.

À la recherche d'un nouveau moyen de mesurer la fréquence du processeur, j'ai exposé mon problème à **Mme. GUERMOUCHE** qui m'a conseillé d'utiliser **Likwid** ([10]). C'est un outil qui permet de mesurer différentes métriques après l'exécution d'un programme. Grâce à **Likwid**, j'ai pu mesurer la fréquence du processeur. Les premiers résultats pris sur un seul cœur correspondaient avec la valeur que j'avais fixé mais après avoir essayé sur l'ensemble des cœurs, l'outil me renvoyait à nouveau 1 GHz.

Ces recherches m'ont pris quelques semaines et après avoir remis les outils en cause (pensant que les valeurs de fréquence fournies par le noyau étaient forcément exactes), j'ai décidé d'étudier l'impact de la fréquence du processeur sur d'autres machines.

3.4.2 Expériences sur grid5000 : un tournant décisif

Rapidement, d'autres membres de l'équipe m'ont proposé de faire mes expériences sur **grid5000** qui est une plate-forme expérimentale répartie en différents sites. Chacun a ses particularités et **Nancy** était le mieux adapté pour mes tests. En effet, il dispose de PDU « manageable » ainsi que de Watt-mètres ce qui me permet de récupérer la consommation individuelle de chaque machine. Ces métriques sont collectées et mises à disposition des utilisateurs grâce à un outil fourni par **grid5000** : **Kwollect**. Pour continuer, mon maître de stage m'a conseillé ce site car il dispose de machines ayant un processeur similaire à celui de **bora044**. En effet, les machines du groupe **gros** sont équipés de processeurs **Intel(R) Xeon(R) Gold 5220 CPU @ 2.20GHz** qui suivent aussi l'architecture Cascade Lake.

La plate-forme **grid5000** a aussi deux avantages importants pour mes tests : il est possible d'être *root* (via **sudo-g5k**) sur les machines afin d'y installer les paquets que l'on veut et d'y déployer facilement une image système. Par chance, l'image **centos7** (qui est la distribution utilisée sur **PlaFRIM**) est disponible dans la liste des images préinstallées.

Après avoir eu la validation du chef de **TADaaM** pour la création de mon compte **grid5000**, je me suis connecté au site **Nancy** pour réaliser des tests sur une machine du groupe **gros**.

Mes premières expériences ont été faites sur la machine **gros-4** tournant sur une distribution **Debian 10.2.1-6** dont le noyau **Linux** est en version **5.10.0-13**.

Mon premier réflexe a été de comprendre **Kwollect**, l'outil de relevé de la consommation électrique. J'ai remarqué qu'il était possible de régler la fréquence des mesures. Par conséquent, comme pour **DCE** (présenté en section 3.2.2), je me suis assuré du type des mesures prises : une moyenne sur la période ou une mesure instantanée. J'ai choisi une période de 5 minutes puis j'ai stressé la machine sur 90% de la période et l'ai laissé inactive sur la fin. En voyant la consommation à son minimum, j'ai compris que **Kwollect** prenait des mesures instantanées.

Ensuite, j'ai débuté mes expériences. Le but était de vérifier le comportement des *governors* et du biais tout en mesurant l'impact de la fréquence sur la consommation électrique. Pour cela, j'ai mesuré la consommation en faisant varier la fréquence manuellement, en changeant le *governor* et le biais de façon cohérente à ce qui serait fait sur **PlaFRIM**.

Ces expériences m'ont mené aux résultats présentés ci-après sous forme de courbe (voir figure 17) et d'un tableau récapitulatif (présent en figure 18). Afin d'être concis, seuls les résultats pris avec le *governor* **powersave** et le biais sur *power* sont montrés.

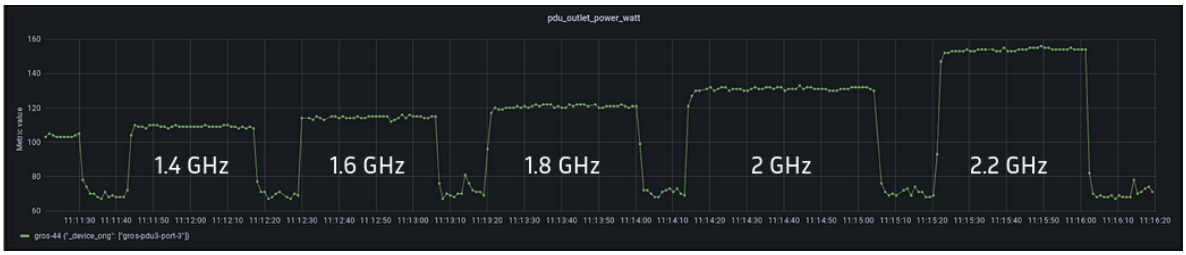

FIGURE 17 – Consommation électrique de **gros-44** en fonction de la fréquence par pas de 200 MHz

Fréquence (GHz)	Governor & biais	Consommation moyenne (W)
1	performance	101
	powersave	98
1.2	performance	107
	powersave	103
1.4	performance	112
	powersave	109
1.6	performance	118
	powersave	114
1.8	performance	124
	powersave	121
2	performance	130
	powersave	130
2.2	performance	153
	powersave	153

FIGURE 18 – Comparaison de la consommation électrique sur **gros-44** en fonction de la fréquence, du *governor* et du biais **intel_pstate** à 100% d'utilisation CPU

Les résultats précédents ont été pris sur une autre machine : **gros-44**. En effet, le lendemain de mes expériences, il a été signalé que le PDU de **gros-4** était défectueux. J'ai été contraint de refaire mes expériences par précaution et j'ai trouvé exactement les mêmes résultats. Pour continuer, j'ai effectué les mêmes expériences en déployant l'image de **centos7** (qui possède quasiment la même version de **Linux** que **PlaFRIM**) sur **gros-44** grâce à **kadeploy3** (l'outil de **grid5000** permettant de charger une image système) ce qui n'a eu aucun impact sur les résultats. J'ai été rassuré de voir que changer le système d'exploitation n'ait aucun effet sur la consommation électrique.

Pour revenir au but principal de l'expérience, j'ai remarqué que les *governors* et le biais n'ont qu'un faible impact sur la consommation (environ 5 W) lorsque la machine est chargée au maximum. Cela montre qu'**intel_pstate** permet au système d'utiliser les plus hautes fréquences lorsque la charge est élevée même si le biais est tourné vers l'économie d'énergie. Ceci n'est pas vrai lorsque **acpi-cpufreq** est utilisé comme nous l'ont montré les expériences sur **diablo04** (rappels sur la figure 9).

Enfin, j'ai effectué les mêmes expériences lorsque la machine était inactive et j'ai remarqué que la consommation moyenne était d'environ 80 W quel que soit le biais ou le *governor*. Après avoir compris que les commandes et les outils que j'utilisais n'étaient pas la source du problème, j'ai appliqué mes nouvelles connaissances sur **PlaFRIM**.

3.4.3 Application sur PlaFRIM

Au départ, j'ai voulu voir si **bora044** n'était pas mal configuré. J'ai donc demandé l'aide des administrateurs pour installer exactement la même configuration système que **gros-44**.

Après avoir refait mes expériences et observé que la fréquence n'avait aucun impact sur **bora044**, j'ai compris que la machine était en cause depuis le début. Afin de prouver définitivement cela, j'ai demandé à récupérer une autre machine du même groupe que **bora044** : **bora040**. J'ai effectué les mêmes expériences de fréquences et rien qu'à l'apparence de la courbe (disponible en figure 19), j'ai su que **bora040** n'avait pas le même problème que **bora044**.

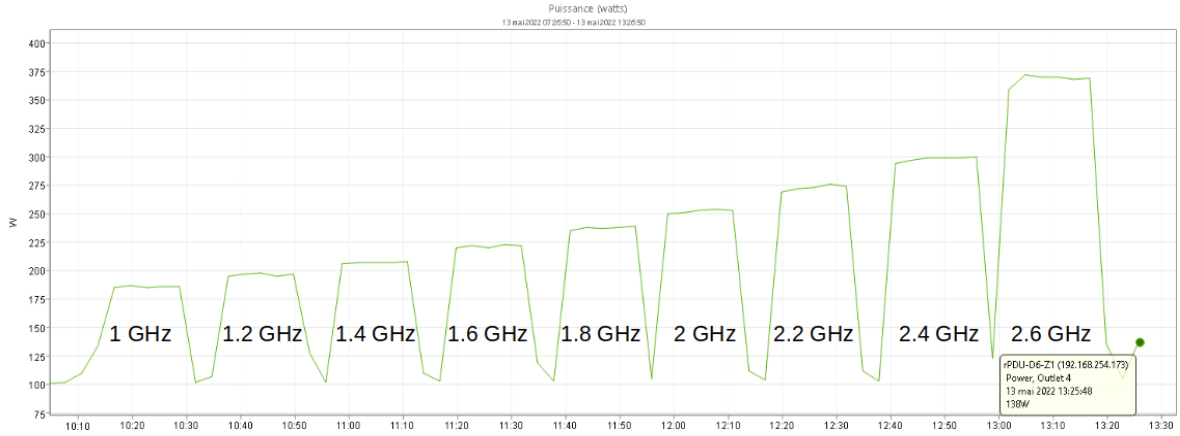

FIGURE 19 – Tracé de la consommation électrique de **bora040** en fonction de la fréquence CPU avec le *governor* **performance**

Par conséquent, j'ai dû refaire mes expériences sur les *governors* puisque les précédentes sur **bora044** ont été faussées par le comportement étrange de la machine. Les résultats sont résumés dans le tableau suivant :

Fréquence (GHz)	Governor & biais	Consommation moyenne (W)
1	performance	186
	powersave	178
1.2	performance	197
	powersave	190
1.4	performance	207
	powersave	200
1.6	performance	222
	powersave	215
1.8	performance	237
	powersave	236
2	performance	252
	powersave	252
2.2	performance	272
	powersave	272
2.4	performance	298
	powersave	300
2.6	performance	370
	powersave	370

FIGURE 20 – Comparaison de la consommation électrique sur **bora040** en fonction du *governor* et de la fréquence du processeur

Tout d'abord, on peut clairement voir l'impact de la fréquence sur la consommation électrique sur la figure 19. Il est intéressant de noter que plus la fréquence est élevée, plus la consommation augmente rapidement. En effet, l'écart de consommation entre 1 GHz et 1.2 GHz est d'environ **10 W** tandis que l'écart entre 2.2 GHz et 2.4 GHz est d'environ **25W** et celui entre 2.4 GHz et 2.6 GHz (la fréquence maximale de la machine) est de **70 W**. Cela signifie qu'une faible réduction de la fréquence CPU permet une diminution importante de la consommation électrique tout en ayant un faible impact sur les performances. Cela est d'autant plus vrai que la machine est capable de monter haut en fréquence.

Ensuite, les résultats présentés en figure 15 combinés à ceux en figure 20 montrent que le changement de *governor* a un faible impact sur la consommation électrique (environ 10 W). Cela s'explique par le fonctionnement des *governors*, en particulier ceux fournis par **intel_pstate**. Que ce soit en **performance** ou en **powersave**, si le système est très chargé, le processeur sera capable d'atteindre sa fréquence maximale dans les deux cas et sa consommation électrique sera identique. Il en va de même lorsque la machine est inactive, son processeur tombe dans les plus basses fréquences quel que soit le *governor*. Cependant, j'ai remarqué qu'en **powersave**, le processeur descend plus rapidement dans les basses fréquences et qu'il y est plus facilement maintenu. Au contraire, de petits pics de fréquence ont été observés avec le *governor* **performance**. Cela permettrait d'expliquer la petite différence de consommation entre les deux *governors* lorsque la machine est inactive.

3.4.4 Conséquences pour PlaFRIM

Après cette révélation, je me suis demandé si d'autres machines ne souffraient pas du même problème. J'ai donc commencé à développer un petit script afin de détecter rapidement si une machine a sa fréquence bloquée comme **bora044**.

Pour détecter ce problème, j'ai commencé par calculer la consommation de la machine. J'ai trouvé

qu'elle obéissait à la formule $consommation = capacitance$[7] $* voltage^2 * frequence$ ce qui venait étayer mes résultats sur le lien entre consommation et fréquence. Ensuite, je me suis rendu compte que calculer la consommation dans un script n'était pas une bonne approche puisqu'il faudrait récupérer les valeurs des variables de la formule à l'aide d'un processus. De plus, je disposais d'une meilleure façon de récupérer la consommation instantanée de la machine avec un sous-processus : appeler la commande **ipmitool**. Cette façon de procéder m'a permis de revoir le module **subprocess** qui m'a servi plus tard (voir section 3.7).

Une fois la consommation récupérée, j'ai développé un algorithme capable de détecter les blocages de fréquence :

Algorithm 1 Algorithme de détection du blocage de la fréquence

```
1: min_freq, max_freq ← read_frequency_bounds()
2: steps ← get_frequency_steps(min_freq, max_freq) ▷ Calcule 3 étapes intermédiaires en plus
   de la fréquence minimale et de la fréquence maximale
3: stress_machine(seconds=60)
4: measures ← []
5: for frequency in steps do
6:     Fixer la fréquence à frequency
7:     consumption ← subprocess("ipmitool dcmi power readings")
8:     measures.append(consumption)
9:     Attendre 10 secondes
10: check_increasing_order(measures)
```

L'algorithme 1 est exécuté plusieurs fois afin de compter le nombre d'occurrences où l'ordre croissant est respecté. En effet, il est attendu que la consommation augmente de façon strictement croissante en même temps que la fréquence augmente. Si cet ordre n'est pas respecté, la machine a un comportement anormal et il est possible que sa fréquence soit en réalité bloquée comme pour **bora044**.

Après avoir fini ce script, j'ai demandé aux administrateurs de le faire tourner sur chaque machine de la plate-forme afin de détecter la présence d'autres cas comme **bora044**. Le premier constat fut que certaines machines n'exposent pas leur fréquence minimale et maximale. Par conséquent, le test ne fonctionnait pas sur ces dernières. Ensuite, les résultats du script montraient que beaucoup de machines avaient le même problème que **bora044**.

Surpris par le nombre de machines signalées comme défectueuses par le script, j'ai reconduit le test manuellement sur certaines d'entre elles et j'ai découvert deux choses.

La première est qu'il est assez courant que les outils du système (en l'occurrence **cat /proc/cpuinfo**) permettant de donner la fréquence actuelle ne fonctionnent pas bien. En fixant manuellement la fréquence, j'ai pu observer un taux d'erreur variable d'une machine à l'autre (souvent jusqu'à 90% d'erreurs !). Ce comportement a été retrouvé sur bon nombre de machines, montrant que le comportement des outils de mesure de fréquence sur **bora044** n'est pas un cas isolé.

La deuxième chose que j'ai découverte est que certaines machines ont une « fréquence bloquée par paliers ». Pour être plus précis, le processeur de certaines machines ne peut prendre que certaines valeurs de fréquence et non pas n'importe laquelle dans l'intervalle [fréquence_minimum, fréquence_maximum]. Ces valeurs précises sont décrites dans **/sys/devices/system/cpu/cpu**x**/cpufreq/scaling_available_frequencies** (pour le cœur n°x). Par exemple, le script signalait que la machine **diablo04** avait un problème. Les différentes étapes de fréquences à tester étaient : 1.5, 1.7, 1.9, 2.1 et 2.35 GHz alors que les fréquences autorisées étaient 1.5 GHz, 2 GHz et 2.35 GHz. Les fréquences non autorisées sont ramenées au palier inférieur ce qui fait qu'en réalité, les mesures ont été prises aux fréquences 1.5, 1.5, 1.5, 2 et 2.35 GHz. Par conséquent, l'ordre croissant n'a pas pu être respecté et le test n'est pas passé.

7. La capacitance représente la capacité d'un composant ou d'un circuit à recevoir et à stocker de l'énergie sous la forme d'une charge électrique.

Pour conclure cette section, j'ai passé un mois et demi à bien comprendre le fonctionnement des *governors* et à mesurer l'impact de la fréquence CPU sur la consommation électrique. J'ai perdu énormément de temps pour essayer de comprendre pourquoi les résultats que j'avais sur **bora044** étaient si incohérents alors que j'aurai pu changer de machine bien plus tôt et essayer mes expériences sur une des **miriel** à ma disposition (elles aussi ont un processeur **intel**). Malgré la perte de temps, j'ai pu approfondir mes connaissances sur les *governors* et j'ai pu comprendre mes résultats surprenants du mois de Mars (notamment ceux présents sur la figure 10).

Enfin, pour réduire la consommation de la plate-forme,
changer le *governor* quand la machine est inactive n'a que peu d'intérêt. En effet, comme nous voulons les meilleures performances pendant les *jobs*, le *governor* reste sur **performance** et la fréquence est à son maximum. Quand la machine est inactive, la fréquence est mise au minimum quel que soit le *governor*. Cependant ce constat est moins vrai sur les machines utilisant **acpi-cpufreq**. Le *governor* change bien la fréquence du processeur mais nous avons vu que le gain moyen est faible (environ 10 W, voir figure 15) sur une journée de travail classique.

Finalement, j'ai retiré des leçons essentielles après tout ce parcours : il ne faut pas faire confiance de manière aveugle aux outils du système et réduire un peu la fréquence permet de réaliser de grandes économies d'énergie sans trop impacter les performances.

3.5 Recherche d'autres techniques

Après avoir accompli les objectifs de mon deuxième mois de stage avec deux semaines de retard, je me suis focalisé sur d'autres pistes de recherche proposées dans le sujet du stage.

3.5.1 Analyse des C-states

L'une de ces pistes parle des *C-states*. Afin de les comprendre, je me suis tourné vers la documentation du noyau **Linux** ([11]).

Cette sous-section a pour but de présenter les connaissances que j'ai acquises en 3 points : comment sont gérés les *C-states*, quels sont les *governors* permettant de choisir ces derniers et en quoi diffère chaque *C-state*.

1. **Comment sont gérés les *C-states* ?**
   Pour commencer, les *C-states* sont des états de basse consommation du processeur et n'entrent en jeu que lorsqu'il est inactif. Ce dernier est considéré comme inactif quand il n'y a plus de tâches et que la *idle task* est exécutée. Le sous-système **CPUIdle** expose différents *drivers* et *governors* utilisés dans la *idle task* dont le fonctionnement est décrit par l'algorithme 2.

Algorithm 2 Fonctionnement de la *idle task* résumée en pseudo-code

```
1: while True do
2:     selected_cstate ← governor()    ▷ Fourni par le CPU idle time management subsystem appelé CPUIdle
3:     driver(selected_cstate)         ▷ Fait partie de CPUIdle, il demande au CPU d'aller dans le C-state sélectionné
4:     if no activity predicted soon then
5:         stop_scheduler_tick()
```

   Chaque *C-state* possède deux paramètres : la **target residency** qui indique le temps minimum que le système doit passer dans cet état (le temps pour entrer dans l'état est inclus) et l'**exit latency** qui donne le temps maximum pour sortir de l'état (le temps pour entrer dans l'état est inclus au cas où le CPU est réveillé pendant qu'il rentre dans l'état choisi).

2. **Présentation des différents *governors***
   À l'instar de **CPUFreq**, **CPUIdle** possède une couche *driver* et une couche *governor*. Sur les

machines **intel** récentes, le *driver* est **intel_idle** tandis que sur les anciens **intel** et les machines **AMD**, le *driver* est **acpi_idle**.

Les *governors* sont choisis en fonction du type du système. Si la ligne 5 de l'algorithme précédent peut être effectuée, il est alors possible d'arrêter l'ordonnanceur de tâches du CPU et le système est dit *tickless*.

Par défaut, pour les systèmes *tickless*, le *governor* **Menu** est utilisé. Ce *governor* tente de prédire le prochain réveil du CPU ce qui lui permet d'estimer le temps d'inactivité (le **sleep length**). Cette estimation est corrigée par un facteur qui varie suivant la précision des prédictions du *governor*. Grâce à un historique récent des périodes d'inactivité du CPU, le *governor* choisi le *C-state* qui satisfait le mieux $target\ residency < sleep\ length\ \&\&\ exit\ latency < extra\ latency\ limit$. L'**extra latency limit** permet de poser une contrainte sur la disponibilité du CPU. Une faible latence permet de sélectionner des *C-states* peu profond. Ils consomment plus d'énergie mais il est possible d'en sortir plus rapidement.

Par défaut pour les systèmes *non-tickless*, le *governor* **ladder** est utilisé. Ce dernier prend d'abord le *C-state* le moins profond disponible. Une fois dans cet état, si le CPU est inactif assez longtemps, il entre dans l'état suivant. La procédure est répétée jusqu'à ce que le CPU soit réveillé.

Pour continuer, il existe un autre *governor* pour les systèmes *non-tickless* : le *governor* **Timer Events Oriented** (**TEO**) mais pour une question de concision, il ne sera pas abordé dans ce rapport.

Ensuite, il existe un quatrième *governor* disponible pour les deux systèmes : **haltpoll**. Ce *governor* fait de l'attente active. Le CPU exécute des instructions inutiles jusqu'à ce qu'il soit sollicité. Ce *governor* permet de maintenir le CPU disponible. Cependant, c'est le pire choix en matière de consommation électrique.

Pour terminer cette présentation des *governors*, il est bon de savoir qu'il est possible de changer le *governor* de **CPUIdle** ou de désactiver directement **CPUIdle** avec la ligne de commande du noyau. Par ailleurs et de manière générale, les systèmes *tickless* permettent d'économiser plus d'énergie que les systèmes *non-tickless*.

3. **Les différents *C-states***
   Chaque *C-state* possède un nom et un numéro. Plus le numéro est grand et plus l'état est profond. Plus l'état est profond, moindre est la consommation électrique mais plus le CPU mettra de temps à se réveiller. Au lieu d'écrire la description de chaque *C-state*, voici un résumé assez complet dont vous trouverez la source ici : [2].

Mode	Name	What it does
C0	Operating State	CPU fully turned on.
C1	Halt	Stops CPU main internal clocks via software ; bus interface unit and APIC are kept running at full speed.
C1E	Enhanced Halt	Stops CPU main internal clocks via software and reduces CPU voltage ; bus interface unit and APIC are kept running at full speed.
C1E	–	Stops all CPU internal clocks.
C2	Stop Grant	Stops CPU main internal clocks via hardware ; bus interface unit and APIC are kept running at full speed.
C2	Stop Clock	Stops CPU internal and external clocks via hardware.
C2E	Extended Stop Grant	Stops CPU main internal clocks via hardware and reduces CPU voltage ; bus interface unit and APIC are kept running at full speed.
C3	Sleep	Stops all CPU internal clocks.
C3	Deep Sleep	Stops all CPU internal and external clocks.
C3	AltVID	Stops all CPU internal clocks and reduces CPU voltage.
C4	Deeper Sleep	Reduces CPU voltage.
C4E/C5	Enhanced Deeper Sleep	Reduces CPU voltage even more and turns off the memory cache.
C6	Deep Power Down	Reduces the CPU internal voltage to any value, including 0V.
C7	Deep Energy Saving	The CPU tries to flush its L3 cache. If the L3 cache **is able to** be entirely cleared, the CPU cuts its power to save energy. The power from the system agent is removed too.
C7s	–	When an MWAIT(C7) command is issued with a C7s sub-state hint, the entire L3 cache is flushed in one step as opposed to flushing the L3 cache in multiple steps. This also allows the system to send I/O devices to low power mode to reduce unnecessary power consumption when the system idles down.
C8	–	The L3 cache is flushed in a single step. The power to the PLL is cut.
C9	–	The VCCIN (VCC Input Voltage) gets lowered to a minimum.
C10	–	The single phase core management system, VR12.6, goes into a low-power state. The CPU is almost shut down.

FIGURE 21 – Tableau récapitulatif des différents *C-states* existants

Dans le dossier **/sys/devices/system/cpu/cpu**x**/cpuidle**, on peut trouver un sous-dossier **state**y pour chaque état supporté par le cœur n°x. Chaque sous-dossier possède différents attributs dont voici les plus intéressants :

- **above** / **below** qui représente le nombre de fois où cet état était trop profond (le CPU a été réveillé plus tôt) / pas assez profond ;
- **desc** / **name** qui contient une description de l'état y / le nom de l'état y (souvent Cy) ;
- **latency** : l'**exit latency** de l'état ;
- **residency** : la **target residency** de l'état ;
- **disable**, c'est le seul paramètre modifiable. S'il est mis à 1, l'état est désactivé.

Il est important de noter que pour désactiver un état, il faut qu'il soit désactivé sur tous les cœurs. Enfin, il faut faire attention, si le *governor* est ladder, il ne sera plus en mesure de sélectionner les états plus profonds que l'état désactivé.

Après avoir acquis de solides connaissances sur les *C-states*, j'ai compris qu'il ne serait pas possible d'agir sur ces derniers puisque la seule chose que l'on puisse faire est de changer le *governor* ou de désactiver certains *C-states* (ce qui ne ferait qu'augmenter la consommation des machines quand elles sont inactives). Pour la plupart de nos machines, le système est *tickless* et donc seul un *governor* est disponible. Par conséquent, le système d'exploitation gère de lui même les *C-states* et le seul moyen de récupérer un gain d'énergie via ces états serait de trouver une façon de forcer le CPU à entrer rapidement dans les états les plus profonds lorsque la machine est inactive. J'ai estimé que le rapport entre le temps nécessaire pour mettre en place cette solution et le gain potentiel était trop faible et j'ai préféré m'intéresser aux autres pistes du sujet de stage.

3.5.2 Les S-states

Après m'être intéressé aux *C-states*, j'ai commencé mes recherches sur les *S-states* avec un autre chapitre de la documentation du noyau **Linux** ([13]). Il faut bien faire la différence entre les *S-states* qui sont les états du système tels que la veille ou l'hibernation et les *C-states* qui sont des états de basse consommation du CPU.

Mes premières recherches m'ont permis de dresser la liste des *S-states* couramment disponibles :

- **Suspend to Idle** (S0) : gèle l'espace utilisateur et place les périphériques d'entrées/sorties en mode basse consommation ;
- **Standby** / **Power-On Suspend** (S1) : comme S0 mais place les cœurs du CPU qui ne sont pas utiles à la reprise du système en mode basse consommation. Le système peut reprendre rapidement ;
- *CPU Poweroff* (S2) : comme S1 mais tout le CPU est éteint ainsi que certains composants de la carte mère. Ce *S-state* n'est pas supporté mais je vous le présente à titre d'information ;
- **Suspend-to-RAM** (S3) : tous les composants sont éteints sauf la RAM où tout l'état du système est sauvé ;
- **Suspend-to-disk** / **Hibernate** (S4) : absolument tous les composants sont hors tension et l'état du système est sauvé sur le disque ;
- **Shutdown** (S5) : extinction pure et simple de la machine, rien n'est sauvé.

J'ai aussi pu trouver les différents paramètres accessibles depuis le système de fichiers. Ces derniers se trouvent dans le dossier **/sys/power** mais le seul qui nous intéresse ici se trouve dans le fichier **state**. Ce dernier contient un mot pour chaque *S-state* disponible. Pour résumer, **freeze** équivaut à S0, **standby** correspond à S1 et **disk** à S4.

Rapidement après cela, je suis tombé sur un article de **Issam Rais et al.** ([5]) que j'ai trouvé particulièrement intéressant puisqu'il couvre les *S-states* et analyse l'impact de l'arrêt des machines sur le matériel.

Pour résumer les contributions des auteurs, l'état S3 n'est pas conseillé car de nombreuses erreurs (notamment des erreurs réseau) surviennent lors du réveil du système. C'est pour cela que cet état

n'est généralement pas activé dans les centres de calcul. Ensuite, leurs résultats les ont mené à ne pas considérer S0 et S1 parce-qu'ils ne sont pas assez profond pour être intéressant en terme d'économie d'énergie.

Par conséquent, les chercheurs se sont concentrés sur S5, l'extinction des machines. Ce qui m'a beaucoup intéressé, c'est le fait qu'ils ont effectué leurs expériences en partie sur **grid5000**. De plus, les auteurs ont comparé deux stratégies d'extinction des machines inactives : l'une agressive où les machines sont éteintes dès la fin d'un *job* et l'autre plus intelligente qui n'éteint les nœuds que s'il y a un gain à le faire. Comme leurs résultats se basent sur des traces, leur algorithme intelligent connaît le futur de façon exacte. Après avoir effectué leurs simulations, les auteurs sont arrivés à la conclusion que les gains engendrés par une politique intelligente d'extinction des nœuds étaient trop faibles par rapport à la mise en place d'un tel algorithme. Ceci dit, d'après eux, une politique intelligente peut avoir un avantage si les périodes d'inactivité sont courtes et fréquentes (ce qui n'est pas le cas sur **PlaFRIM**).

Enfin, les auteurs de l'article ont compté le nombre de cycles de démarrage et d'allumage des nœuds afin de voir si cela avait un impact négatif sur le matériel. Ils sont arrivés à la conclusion que même une politique d'extinction très agressive n'endommage pas le matériel (les disques en particulier).

Après avoir acquis ces connaissances, je suis allé sur **PlaFRIM** pour voir quels *S-states* sont disponibles. La figure suivante montre leur disponibilité et les classe en fonction de la consommation qu'ils engendrent et du temps nécessaire pour la remise en service des machines.

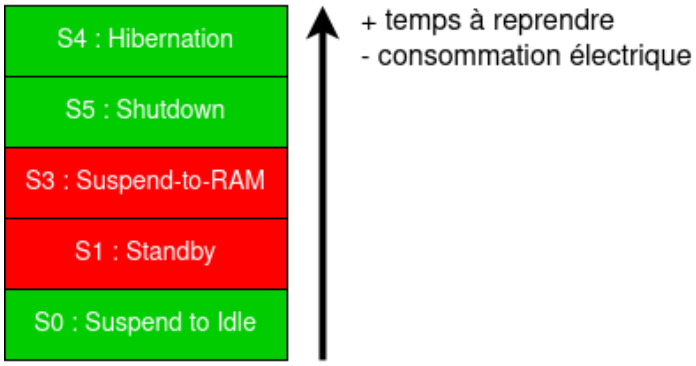

FIGURE 22 – Présentation des *S-states* disponibles sur **PlaFRIM** (en vert) triés par ordre décroissant de la consommation électrique qu'ils engendrent

Comme vu précédemment, S0 n'est pas un état intéressant pour économiser de l'énergie. S1 et S3 ne sont pas disponibles donc je n'avais le choix qu'entre S4 et S5. Ces deux états ne consomment pas d'énergie puisque la machine est éteinte. Cependant, S4 met plus de temps à reprendre son activité que S5 car il est nécessaire de charger l'état du système depuis le disque. Par conséquent, le seul *S-state* intéressant pour économiser de l'énergie sur **PlaFRIM** est S5, soit une extinction pure et simple des machines. Cette technique est d'autant plus viable qu'elle ne semble pas abîmer le matériel.

3.5.3 Autres techniques

Avant d'approfondir davantage le sujet de l'arrêt des machines, j'ai cherché les pratiques courantes en terme d'économie d'énergie dans les *data centers*.

J'ai trouvé un article de **Huigui Rong et al.** ([6]) qui propose différentes solutions logicielles et matérielles pour réduire la consommation électrique des *data centers*. Voici les plus intéressantes que j'ai pu lister :

— **éteindre les liens réseaux inutilisés** : cette solution a été explorée en section 3.1.3 et n'est pas applicable sur **PlaFRIM** ;

— **virtualiser les jobs** : cela permettrait de mettre plusieurs *jobs* sur une même machine et donc de diminuer le nombre de machines à garder allumé pour servir les utilisateurs. Cette solution n'est pas envisageable car les machines virtuelles n'offrent pas un environnement reproductible pour les expériences scientifiques ;
— **placer les jobs nécessitant peu de ressources sur des machines peu puissantes** : cette solution n'est pas applicable car il nous faudrait un moyen d'estimer la demande en énergie des *jobs*. De plus, cela nous forcerait à changer les allocations des utilisateurs ce que nous ne pouvons pas faire pour ne pas altérer leurs conditions expérimentales ;
— **organiser les serveurs de sorte à créer des couloirs chauds** : la disposition de la salle machine suit déjà ce principe. Cela permet de ne refroidir que les couloirs chauds au lieu de la pièce entière ;
— **réserver les machines proches de la climatisation en priorité** : comme ces dernières sont mieux refroidies, il vaut mieux les utiliser en premier. Cette solution est envisageable en modifiant les paramètres de priorité de **SLURM** pour qu'il alloue facilement les machines proches de la climatisation [8].

Malheureusement, il n'était pas possible de mettre en place la plupart de ces techniques sur **PlaFRIM** et la seule solution disponible dans la liste ne me semblait pas nous procurer un grand gain en terme d'énergie.

Pour conclure, après avoir étudié les *C-states*, les *S-states* et d'autres techniques d'économie d'énergie, seul l'arrêt des nœuds inutilisés m'a paru être la solution ayant le plus d'impact sur le bilan énergétique de la plate-forme.

3.6 Arrêt des nœuds : les fonctionnalités de SLURM

Le troisième mois de mon stage touchait à sa fin et j'allais commencer à explorer la piste la plus prometteuse de mon stage : éteindre les nœuds inactifs. Cependant, avant de savoir comment éteindre et allumer les machines de façon automatique via **SLURM**, je me suis intéressé aux statistiques d'utilisation de **PlaFRIM** afin de savoir quelles machines éteindre.

3.6.1 Préparation : statistiques d'utilisations avancées

Ma mission consistait à donner le taux d'utilisation de chaque groupe de nœuds en fonction de l'heure, de la journée et des périodes de vacances.

Pour ce faire, j'avais à ma disposition l'ensemble des traces d'utilisation de **PlaFRIM** fournies par **SLURM** sur l'année 2021. Ces traces contiennent des informations sur chaque *job*, à savoir :
— son **identifiant unique (JobIDRaw)** ;
— son **nom (JobName)** ;
— sa **date soumission (Submit)** ;
— sa **date de début (Start) et de fin (End)** ;
— la **liste des nœuds utilisés (NodeList)** ;
— le **nombre de nœuds utilisés (NNodes)**.

Afin de mener à bien ma mission, je me suis appuyé sur mes connaissances en Spark que j'ai développé lors du cours de support de données enseigné au semestre précédent. En effet, une partie de ce cours était dédié à la manipulation de *dataframes* avec Spark. Pour réaliser mes traitements, j'ai choisi d'utiliser un module très populaire en Python pour manipuler ce genre de structures de données : **pandas**.

Mon premier réflexe fut de réduire mes données aux informations nécessaires et j'ai retiré toutes les lignes concernant des *jobs* vides.

Première version : total des nœuds utilisés en fonction des heures

8. Ces machines doivent être identifiées au préalable.

Une fois mes données prêtes, j'ai pensé à une première façon de les regrouper : pour chaque *job*, prendre sa date de soumission et l'identifier par un tuple (**IsHoliday, Weekday, Hour**). Pour réaliser la majorité de mes transformations sur mes données, j'utilisais la méthode **apply** afin d'appliquer une fonction à chaque ligne du *dataframe*. La création de chaque tuple d'identification temporel nécessitait de lire une chaîne de caractères représentant une date et d'en extraire le jour de la semaine, l'heure et de tester si le jour se trouve dans une période de vacances définie par mon maître de stage. Pour réaliser les traitements, je me suis tourné vers le module **datetime** qui offre des méthodes permettant d'analyser des chaînes représentant des dates pour les convertir en objets. Ensuite, ce module expose un nombre considérable de méthodes dont certaines permettant de comparer des dates, de récupérer le jour de la semaine, l'heure, etc.

Pour continuer, il fallait que je compte le nombre de nœuds en fonction de leur groupe. Dans un premier temps, j'ai été heurté à un problème de taille : la colonne **NodeList** contient une chaîne représentant la liste des nœuds du *job* en suivant le format **SLURM**. Pour lire ce format, il existe la commande **scontrol show hostnames <list>**. Cepdendant, je ne voulais pas utiliser de **subprocess** pour exécuter cette commande afin de ne pas ralentir le script. Je me suis alors mis à chercher un utilitaire Python pour lire les listes de nœuds **SLURM**. Malheureusement, je n'ai pas pu trouver tel outil.

J'ai ensuite essayé de créer le mien mais la tâche n'était pas simple, il fallait être capable de transformer des chaînes complexes (ex. : "miriel[004-008],bora030,sirocco[05,08]") en liste de nœuds au format Python (exemple : ["miriel004","miriel005","miriel006","miriel007","miriel008","bora030","sirocco 05","sirocco08"]). J'ai exposé mon problème à **M. GOGLIN** qui m'a montré des scripts servant à effectuer d'autres statistiques sur **PlaFRIM**. Certains d'entre eux possédaient des fonctions faisant exactement ce que je cherchais. J'ai pu les récupérer pour analyser la liste de nœuds de chaque *job* et ainsi en déduire le nombre de nœuds utilisés.

Voici un exemple d'entrées du *dataframe* produit par la première version de mon script :

JobIDRaw	Start	NodeList
499167	2021-01-17T16:01:04	diablo04,miriel[087-088]

↓

IsHoliday	Weekday	Hour	NodeCount
False	6	16	3

Figure 23 – Transformation des données dans la première version du script

*NB : Le jour 0 correspond au lundi et le jour 6 correspond au dimanche. Pour créer la colonne **NodeCount**, j'aurai pu directement utiliser la colonne **NNodes** du dataframe de départ. J'ai préféré me servir de la liste de nœuds car je devais me frotter à son analyse tôt ou tard pour la seconde version du script.*

Après cette transformation, il ne me restait plus qu'à grouper les lignes possédant les mêmes valeurs pour les colonnes **IsHoliday, Weekday** et **Hour**. Pour ce faire, j'ai utilisé la méthode **groupby** qui permet de regrouper les lignes en fonction de la valeur de certaines colonnes données en paramètre. Cette méthode est très commune aux *dataframes* et j'avais déjà pu l'utiliser en cours. Une fois les données regroupées, il ne me restait plus qu'à sommer les valeurs de la colonne **NodeCount** pour

déterminer le nombre total de nœuds utilisés pour chaque tuple d'identification temporel (rappel : (**IsHoliday, Weekday, Hour**)).

Cette première version du script était un bon début mais les résultats n'étaient pas vraiment exploitables. Nous pouvions voir le nombre de nœuds utilisés en fonction des heures de la journée mais la contribution de chaque groupe au total n'était pas montrée et il était donc impossible de savoir quels groupes de nœuds garder allumés.

Seconde version : séparer la contribution de chaque groupe de nœuds

L'objectif principal était de transformer le total des nœuds pour chaque *job* en une colonne contenant le groupe et une autre contenant le nombre de nœuds du groupe.

En regardant les autres scripts présents pour construire les graphiques à partir d'autres statistiques, je suis tombé sur une méthode intéressante : **pivot_table**. Cette méthode prend 3 paramètres, chacun étant une liste de colonne. Le premier paramètre *index* donne les colonnes utilisées pour grouper les données, le deuxième permet de transformer chaque valeurs des colonnes désignées par *columns* en colonnes (une nouvelle colonne par valeur) et le dernier paramètre *values* sert à désigner la ou les colonne(s) contenant les valeurs du nouveau *dataframe*. Par défaut, la moyenne est faite sur les valeurs et c'est exactement ce dont j'avais besoin à terme.

Le nouvel objectif principal était de générer, pour chaque *job*, une ligne par groupe de nœuds utilisé. Chaque ligne devait contenir le tuple d'identification temporel du *job*, le groupe et le nombre de nœuds afin d'appliquer **pivot_table** et d'obtenir mes résultats finaux. Mon premier problème fut de séparer les différents groupes de nœuds présents dans la colonne **NodeList**. En fouillant la documentation, j'ai trouvé la méthode **explode** qui prend en paramètre le nom d'une colonne et qui, pour chaque élément de la liste contenue dans la valeur de la colonne, crée une ligne correspondante à chaque valeur. Ceci m'a été très utile pour créer une ligne pour chaque machine utilisée par le *job* en transformant la colonne **NodeList** en liste Python et en appelant **explode** sur cette dernière.

De cette manière, il m'a été facile de gérer le cas où une même machine est partagée par des *jobs* différents. En effet, en ajoutant le numéro de la semaine au tuple d'identification temporel, si une machine apparaissait plusieurs fois à la même heure du même jour de la même semaine, je savais que cette machine ne devait être comptée qu'une fois puisqu'elle était partagée par plusieurs *jobs*. Afin de retirer toutes les lignes dupliquées de mon *dataframe*, j'ai utilisé la méthode intégrée : **drop_duplicates**.

Cela marchait plutôt bien et après un **pivot_table**, je disposais d'une colonne **NodeGroup** et d'une colonne **NodeCount** pour chaque nœud utilisé par le *job*.

Ensuite, pour que mes résultats soient vraiment représentatifs de l'activité de **PlaFRIM**, il me fallait oublier la date de soumission des *jobs* (d'autant plus qu'un *job* peut être soumis et lancé des heures plus tard). Je devais compter les machines utilisées par un *job* pour chaque heure où le *job* a été exécuté (en arrondissant à l'heure supérieure). Pour l'implémentation, j'ai pris la date de début de chaque *job* puis j'ai avancé par pas d'une heure jusqu'à atteindre la fin du *job*. À chaque tour de boucle je générais un tuple d'identification temporel. Chacun d'eux correspondait à une ligne contenant la liste de nœuds complète du *job*. L'appel à **explode** et à **drop_duplicates** était fait après.

J'avais enfin mes résultats séparés pour chaque groupe de nœuds mais quelque chose n'allait pas : je voyais que même les machines très peu utilisées (comme **arm01**) avaient une moyenne d'utilisation très élevée. Après inspection, je me suis rendu compte que mes moyennes n'avaient pas de dénominateur commun.

D'un côté, les machines les plus utilisées apparaissent quasiment tous les jours de l'année ce qui me donne 52 valeurs différentes pour chaque couple (jour, heure). De l'autre, ce nombre est bien plus faible si la machine n'apparaît que très peu de fois dans l'année. Afin de diviser toutes mes sommes par 52, j'ai créé un *dataframe* « minimal » contenant tous les tuples d'identification temporel de l'année et pour chaque tuple, je générais une ligne par groupe de nœuds présent dans les données originales avec une valeur de 0. De ce fait, après concaténation du *dataframe* « minimal » avec le *dataframe* principal, tous les groupes de nœuds étaient représentés 52 fois et le résultat de mes moyennes était correct.

Pour vous aider, le résumé illustré de ces transformations est fourni en annexe, section D.

Lors du développement, je pensais que faire simplement la division par 52 n'était pas une bonne pratique à cause de l'introduction d'une « constante magique ». J'ai alors opté pour cette alternative lente et peu élégante en pensant bien faire. Avec du recul, je n'aurai pas dû m'obstiner dans mon erreur et accepter le prix d'une constante dans le code.

Présentation des résultats

Une fois mes données prêtes, il ne me manquait plus qu'à les afficher. Pour cela, je me suis inspiré des autres scripts d'affichage écrits en Python. Ces derniers utilisent le module **matplotlib** pour faire leur graphiques. J'ai eu l'occasion de m'entraîner à utiliser ce module au cours de mon cursus et écrire ce script fut une bonne piqûre de rappel.

Pour réaliser mon affichage, j'ai commencé par lire le *dataframe* à partir du fichier généré par le script **get-usage-per-hour.py** précédent puis je n'ai gardé que les données correspondantes au jour donné en paramètre de mon script d'affichage. Ensuite, j'ai créé un *stacked bar plot* avec les données filtrées. Ce dernier est enfin affiché à l'écran ou est stocké dans un fichier.

Au cours de la réalisation de ce script d'affichage que j'ai nommé **plot-usage-per-hour.py**, j'ai remarqué que les barres que l'on peut voir sur la figure 24 ne correspondent pas toujours aux marques de la légende. C'était causé par le manque de certaines valeurs dans le *dataframe*. Afin de m'assurer que toutes les valeurs soient présentes et dans le bon ordre, j'ai trié les lignes de mon *dataframe* selon le tuple d'identification temporel puis j'ai rajouté les éventuels zéros manquants.

Au final, ces deux scripts me permettent d'obtenir des statistiques détaillées sur l'utilisation de **PlaFRIM** et d'afficher élégamment les résultats.

Pour illustrer, la figure 24 est produite à l'aide de mes scripts et montre l'utilisation de la plate-forme pendant les vendredis ouvrés de l'an 2021.

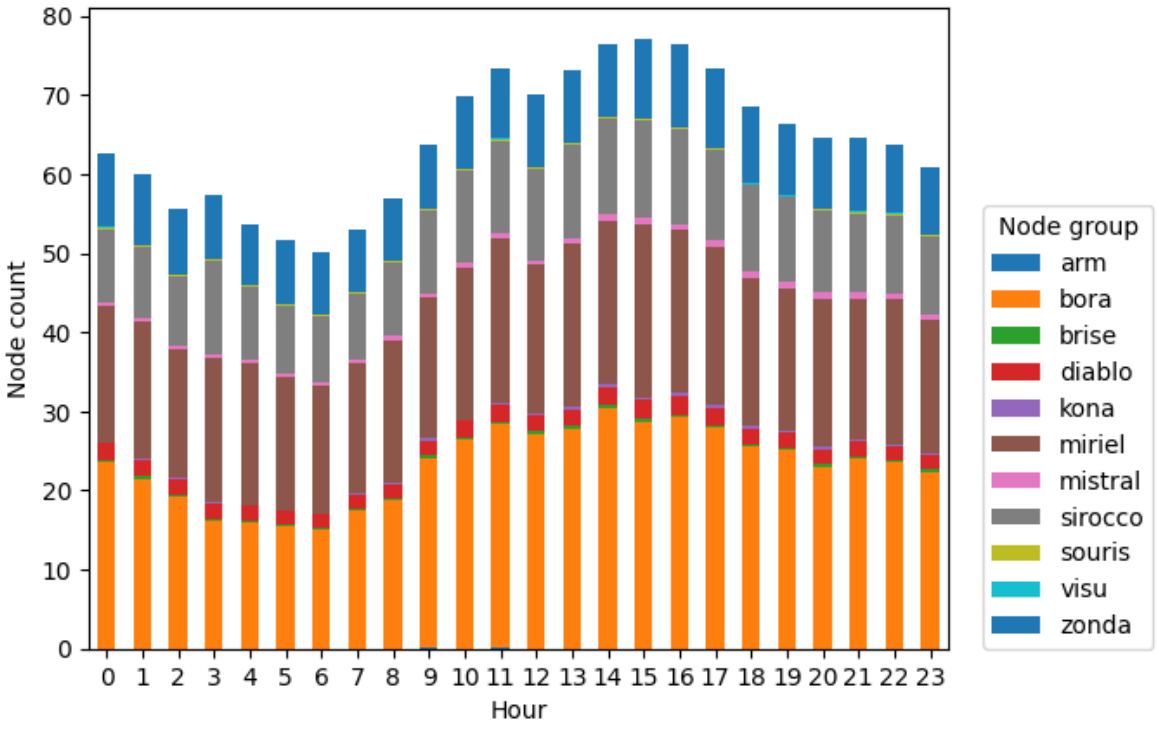

Figure 24 – Utilisation de **PlaFRIM** pendant les vendredis ouvrés en 2021

Par ailleurs, j'ai utilisé le module **argparse** de Python afin d'analyser la ligne de commande

invoquant les scripts. Cela m'a permis de récupérer facilement les options, de gérer les erreurs et de fournir un message d'usage très bien formaté dont voici un exemple :

```
cmercier@CORE-System:~/Stage_Energie/python_scripts$ ./get-usage-per-hour.py -h
usage: get-usage-per-hour.py [-h] [-f FILENAME] [-s SEPARATOR] [-y YEAR] database

positional arguments:
  database              the csv file containing the data to parse

optional arguments:
  -h, --help            show this help message and exit
  -f FILENAME, --filename FILENAME
                        the parsed data will be stored to the given file. If not provided they will be saved to 'output.csv'.
  -s SEPARATOR, --separator SEPARATOR
                        use a custom separator instead of ','
  -y YEAR, --year YEAR  the year of the data to parse (default : 2021)
cmercier@CORE-System:~/Stage_Energie/python_scripts$
```

FIGURE 25 – Message d'aide du script **get-usage-per-hour.py**

Enfin, je me suis aussi assuré que mes scripts soit bien commentés et documentés puis j'ai écrit une petite documentation technique fournissant quelques exemples d'utilisation pour les administrateurs de **PlaFRIM**.

3.6.2 État de l'art : fonctionnalités de SLURM et de ses *plugins*

Il m'a fallu deux semaines pour réaliser l'exploitation des statistiques de **PlaFRIM**. Ceci nous amenait au début du mois de Juin et je me suis focalisé sur la préparation de la soutenance de mi-stage qui avait lieu le 24.

Malgré tout, j'ai pu progresser pendant les trois premières semaines de Juin puisque j'ai parcouru la documentation de **SLURM** ([9]) afin de connaître ses fonctionnalités concernant la consommation électrique.

Pour résumer, voici les différentes variables disponibles à partir du fichier de configuration **SLURM** (**/etc/slurm/slurm.conf**) qui concernent son mode économie d'énergie :

- **SuspendTime** : en secondes, c'est le temps d'inactivité requis avant qu'un nœud soit mis en mode économie d'énergie. Si cette variable vaut -1, le système d'économie d'énergie est désactivé ;
- **SuspendRate** / **ResumeRate** : c'est le nombre de nœuds pouvant entrer / sortir du mode économie d'énergie à chaque minute (particulièrement utile pour les grosses plate-formes) ;
- **SuspendProgram** / **ResumeProgram** : cette variable contient le chemin vers le script qui sera exécuté par **SLURM** quand il placera / sortira des nœuds du mode économie d'énergie. La liste des nœuds concernés est passée en paramètre de ces scripts ;
- **SuspendTimeout** / **ResumeTimeout** : définit le temps en secondes pour laisser un nœud entrer / sortir du mode économie d'énergie ;
- **SuspendExcNodes** / **SuspendExcParts** : liste des nœuds / des partitions à exclure du mécanisme d'économie d'énergie. Il est possible de dire à **SLURM** de laisser certaines machines inactives. Par exemple, **bora[001-040]:4** indique à **SLURM** de garder 4 nœuds **bora** inactifs. Il est bon de noter que **SLURM** ne démarrera pas les 4 nœuds s'ils sont éteints. Ils doivent d'abord être allumés par les utilisateurs.

La documentation explique également le fonctionnement du mode économie d'énergie. Ce dernier est assuré par le démon **SLURM slurmctld** qui exécute périodiquement l'algorithme suivant.

Tout d'abord, il identifie les nœuds inactifs depuis au moins **SuspendTime** secondes. Ensuite, il exécute le **SuspendProgram**, avec en argument, la liste des nœuds inactifs à passer en mode économie d'énergie.

Pour continuer, le démon regarde les nœuds qui sont demandés pour un *job* et qui sont en mode économie d'énergie. Il exécute le **ResumeProgram**, avec en argument, la liste des nœuds concernés. Si un nœud n'est pas en service après le **ResumeTimeout**, il est placé dans l'état **down**~ et est

considéré comme hors d'usage. Les *jobs* ayant demandé le nœud hors d'usage sont remis dans la file d'attente.

Cela signifie que **SLURM** est capable d'éteindre les nœuds inactifs si le **SuspendProgram** contient les instructions nécessaires à l'arrêt des nœuds. Cela doit aller de paire avec un **ResumeProgram** capable de démarrer les nœuds concernés. **SLURM** est aussi capable de gérer des *jobs* dont certains nœuds sont inactifs et il se charge également du changement d'état des nœuds.

En effet, chaque nœud possède un état dont les principaux sont :

— **alloc** : tous les cœurs du nœud sont alloués ;
— **mix** : une partie des cœurs du nœud sont alloués ;
— **idle** : l'entièreté du nœud est disponible pour allocation ;
— **down** : le nœud est hors service.

Avec le mode économie d'énergie, trois nouveaux symboles peuvent venir en suffixe des états précédents. Un « ∼ » signifie que le nœud est en mode économie d'énergie ; un « # » signifie que le nœud est en train de sortir du mode économie d'énergie et enfin un « % » marque l'entrée du nœud dans le mode économie d'énergie.

Les capacités intégrées à **SLURM** sont puissantes mais elles ne suffisent pas pour nos besoins. En effet, mon maître de stage aimerait pouvoir garder certaines machines allumées en avance afin d'éviter que les *jobs* interactifs doivent attendre l'allumage des nœuds demandés. Hors **SLURM** ne sait garder éveillé que les machines déjà allumées par les utilisateurs. **M. GOGLIN** aimerait également avoir un **SuspendTime** différent pour certaines machines, notamment **arm01** qui n'est active que rarement mais qui est très utilisée lorsqu'elle est sollicitée. Par exemple, avoir un **SuspendTime** d'une semaine sur cette machine uniquement permettrait aux chercheurs qui la sollicitent de ne pas devoir la rallumer chaque matin de la semaine.

Garder certaines machines allumées en prévision de *jobs* futurs constitue une **marge de réactivité**. Et comme nous venons de le voir, **SLURM** ne sais pas la gérer correctement et ne permet pas d'avoir un **SuspendTime** différent pour certains nœuds.

Je me suis alors mis à chercher du côté des *plugins* officiels sur le **GitHub** de **SLURM**, en vain. En effet, les seuls *plugins* en rapport avec l'énergie que j'ai trouvé sont :

— **acct_gather_energy** qui interroge les MSR (introduits en section 3.2.2) afin de récupérer la consommation du CPU et du GPU des nœuds et de consigner les résultats dans une base de données ;
— **power** qui sert à mesurer la consommation globale de la plate-forme.

Les autres *plugins* que j'ai trouvé permettent de gérer d'autres technologies telles que les **burst buffers**, de collecter des données utilisateurs pour en faire des statistiques, de mettre en place différentes files avec des priorités différentes, de gérer la topologie des nœuds et de MPI, etc.

J'ai également tenté de chercher des *plugins* non-officiels sur Internet, sans succès. En dernier recours, j'ai fait appel à la communauté via la liste de diffusion publique des utilisateurs **SLURM**. La seule réponse que j'ai reçu m'indiquait qu'il n'existait aucun *plugin* permettant de mettre en place une marge de réactivité ou un **SuspendTime** différent pour certains nœuds. Dans cette réponse, il m'était conseiller d'implémenter ces systèmes moi-même.

3.6.3 Allumer et éteindre les machines

Après avoir compris le mécanisme d'économie d'énergie de **SLURM** et avant d'implémenter moi-même la marge de réactivité, j'avais décidé de mettre en place le système basique permettant d'éteindre et d'allumer les nœuds. Cela serait une v0 de la piste consistant à éteindre les nœuds inactifs.

Au départ, j'ai commencé par modifier la configuration de **SLURM** sur **Formation** pour activer le mécanisme d'économie d'énergie. J'ai placé le **SuspendTime** à 60 secondes afin de voir le comportement du **SuspendProgram** sans trop attendre. Après avoir définit les chemins du **SuspendProgram** et du **ResumeProgram**, j'ai essayé d'écrire ces scripts dans mon *home*. Cependant, cela ne menait à rien car même si j'avais les droits pour modifier la configuration de **SLURM** sur **Formation**, **SLURM** n'avait pas les droits de lire les scripts dans mon *home* et cela provoquait une erreur.

Avec l'aide des administrateurs de **PlaFRIM**, nous avons placé les scripts dans le dossier partagé présent sur tous les nœuds et accessible à **SLURM**. C'est également là où se trouvent d'autres scripts propres à **SLURM** tels que les prologues / épilogues qui sont des scripts exécutés au début / à la fin d'un *job*.

Après avoir réglé quelques *crashs* de **SLURM** avec **M. LELAURAIN**, mes scripts s'exécutaient et les machines passaient bien dans l'état **idle**~. Le problème fut qu'on ne puisse pas remettre les machines en service. Lorsqu'on faisait une demande, les nœuds concernés atteignaient le **ResumeTimeout** et passaient **down**~. La cause du problème résidait dans le fait que mes scripts ne faisaient qu'afficher des messages. Lorsque les nœuds étaient placés en mode économie d'énergie, le démon **SLURM** était tué et quand ils sortaient du mode économie d'énergie, le démon **SLURM** n'était pas redémarré puisque les nœuds n'étaient pas réellement redémarrés.

Afin de régler ce problème, il fallait que mes scripts agissent vraiment sur la mise sous tension des nœuds. Pour gérer cela à distance, les administrateurs utilisent IPMI ou **rpower**. Au départ, ils ne voulaient pas me donner accès à ces commandes [9] pour des raisons de sécurité [10]. Par conséquent, **M. LELAURAIN** a fait en sorte que lorsque **SLURM** se connecte en **ssh** sur un nœud, cela exécute la commande **poweroff**. Une fois la partie extinction réglé, il a mis en place le protocole **Wake-on-LAN** afin de rallumer les machines à distance sans passer par IPMI.

Après l'installation de ces mécanismes, j'ai pu écrire mes scripts **SLURM** (**SuspendProgram** et **ResumeProgram**).

Faire en sorte que les nœuds s'éteignent était facile, il suffisait d'invoquer **ssh** avec l'utilisateur **SLURM** sur la machine cible. Pour l'allumage, c'était un peu plus compliqué. Comme la commande **wol** (**Wake-on-LAN**) prend l'adresse MAC de la machine à réveiller, il a fallu me fournir la liste de toutes les adresses des nœuds de **PlaFRIM**. J'ai pu mettre en œuvre mes connaissances en Bash pour récupérer l'adresse MAC de la machine voulue dans un fichier à l'aide d'une combinaison de **sed** et de **grep**.

Au final, le mécanisme d'économie d'énergie de **SLURM** fonctionnait très bien. Les machines s'allumaient et s'éteignaient comme prévu, au départ. En effet, nous nous sommes aperçus que faire un **poweroff** coupe parfois la communication **ssh** avant qu'elle ne soit fermée proprement ce qui fait que le **SuspendProgram** renvoie une erreur. Pour corriger cela, j'ai proposé de remplacer **poweroff** par **shutdown** avec un délai suivi d'un **exit** afin de terminer correctement la connexion **ssh**.

À la fin du mois de Juin, nous avions mis en place le système d'économie d'énergie de **SLURM** et je pouvais m'attaquer à la marge de réactivité et aux délais personnalisés des nœuds.

Pendant le développement de la marge de réactivité, nous avons remarqué que certaines machines ne redémarraient pas. Parfois, le paquet **Wake-on-LAN** n'arrivait pas à sa destination ou la machine ne supportait pas ce protocole. Après réflexion, les administrateurs m'ont accordé l'accès à IPMI via un script préparé par **M. SIRVIN**. Ce dernier me permettait uniquement d'allumer les machines et de vérifier l'état de leur alimentation. Cette solution fut très efficace et fonctionnait bien mieux que le **Wake-on-LAN**.

3.7 Arrêt des nœuds : marge de réactivité et délais personnalisés

Quand j'avais compris qu'il fallait que je programme la marge de réactivité à la main, je pensais à avoir affaire à une tâche extrêmement difficile. Sans accès aux structures de **SLURM** gérant les temps d'inactivité, je pensais devoir tenir une base de données enregistrant les dates de début d'inactivité des nœuds, tenir le compte des nœuds disponibles pour chaque groupe et surveiller l'activité de la plate-forme pour savoir quels nœuds maintenir éveillés en fonction de la marge demandée.

9. J'avais uniquement accès aux commandes **ipmitool** me permettant de lire la consommation électrique des machines.

10. Le réseau IPMI est séparé du réseau classique et de prime abord, il n'était pas question de permettre à mes scripts d'interagir avec ce dernier. Entre de mauvaises mains, un intrus pourrait éteindre les machines brutalement ou accéder à bien d'autres fonctionnalités privilégiées.

Heureusement, pendant la mise en place du mode d'économie d'énergie de **SLURM**, j'ai discuté de mon problème lié à la marge de réactivité aux administrateurs de **PlaFRIM**. Grâce à leur aide, j'ai compris que choisir les nœuds à maintenir éveillés ne serait pas si difficile. Par exemple, pour maintenir 3 nœuds **miriel** éveillés, je peux simplement demander une allocation de 3 nœuds **miriel** à **SLURM**. C'est lui qui se chargera de me donner les nœuds déjà disponibles et de réveiller davantage de nœuds si nécessaire. Même si je ne peux pas accéder aux chronomètres mesurant l'inactivité des différents nœuds, je peux toujours envoyer des allocations courtes pour réinitialiser ce dernier.

Suite à ça, j'avais trouvé une idée simple pour mener à bien ma mission et les administrateurs étaient plutôt d'accord avec cette dernière : utiliser un **cron** pour envoyer régulièrement des petits *jobs* aux machines à maintenir éveillées (que ce soit pour la marge de réactivité ou les délais d'inactivité avant extinction personnalisés). Le début de l'implémentation du système débuta début Juillet, il me restait environ 1 mois et demi pour remplir ma mission du mieux que je pouvais.

3.7.1 Marge de réactivité : premières versions

Pour commencer, mon but était de créer un script PYTHON qui lancerait des petits *jobs* (via la commande **SLURM srun**) afin de maintenir certains nœuds éveillés. Pour choisir quels nœuds garder, l'utilisateur du script pourrait écrire ce qu'il souhaite dans un fichier de configuration qui serait lu par le script **reactivity_margin.py**.

Afin de ne pas trop m'embêter à vérifier le format du fichier de configuration, j'ai opté pour le module **configparser**. Ce dernier permet de lire facilement des fichiers au format INI avec la méthode **read_file**. De ce fait, un fichier contenant :

```
[Section]
    variable = valeur
    miriel = 4
```

pouvait être lu par le module et il me suffisait d'interroger la variable contenant le résultat de la lecture comme un dictionnaire pour obtenir mes valeurs. Dans notre exemple, config["Section"]["miriel"] renvoie 4.

Ensuite, j'ai rapidement trouvé un algorithme qui me permettrait d'implémenter la marge de réactivité : faire en sorte que chaque section représente une période et après avoir trouvé la première section contenant la date actuelle, lire la définition de la marge correspondante et l'appliquer.

Au départ, pour trouver facilement la section définissant la marge de réactivité d'une section décrivant une période de temps, son nom devait être <section_temporelle>**_margin**.

Ensuite, dans le but de trouver la section contenant la date actuelle, chacune d'elle devait contenir la variable **days** qui représente une liste de plages de dates (au format jj/mm to jj/mm) séparées par des virgules. Il était également possible de définir une contrainte optionnelle sur les heures (au format hh to hh) afin que la section ne soit choisie que si la date et l'heure corresponde.

Au final, j'ai pu implémenter cette première version en trois semaines (en parallèle du script présenté en section 3.7.3). Mon script était capable de lire un fichier de configuration simple dont un exemple est donné ci-après :

```
[Business days]
    days = 01/01 to 15/07, 16/08 to 31/12
    hours = 8 to 9

[Holidays]
    days = 15/07 to 15/08
    #hours = 9 # hours isn't defined -> section active all day

# MARGIN DEFINITION BELOW
```

```
[Business days_margin]
    miriel = 4
    bora = 2

[Holidays_margin]
    miriel = 1
```

Dans cet exemple, si nous sommes le 2 Septembre à 8h30, la section **Business days** sera sélectionnée et le contenu de la section **Business days_margin** sera lue. Pour chaque paire (variable, valeur), un *job* sera lancé en demandant l'allocation de <valeur> nœuds du groupe <variable>.

Vers la fin du mois de Juillet, j'ai eu un retour de mon maître de stage sur mes scripts. Ce fut le début du développement de la deuxième version de **reactivity_margin.py** incorporant les changements suivants :

- **le format des dates inclut l'année** (nouveau format : aaaa/mm/jj) ;
- **fusion des sections** qui décrivent une période et celles définissant une marge ;
- **fusion des fichiers de configurations** des deux scripts. Cela provoquait un conflit de noms de variables et j'ai dû renommer **days** et **hours** en **date_range** et **hour_range** ;
- **création d'un script principal** en Bash : **keep_awake_policy.sh** qui appelle les deux scripts **reactivity_margin.py** et **custom_node_timeout.py** (décrit en section 3.7.3).

De ma propre initiative, j'ai effectué les changements suivants pour simplifier la tâche des administrateurs :

- **ajout du support des jours de la semaine** dans la variable **date_range**. Par exemple, écrire « Mondays » équivaut à « tous les lundis » ;
- **ajout du mot-clé « NOT »** suivi d'un nom de section dans la variable **date_range**. Cette expression est reconnue comme l'inverse de la plage de dates de la section donnée ;
- **le script principal inscrit les évènements** dans un fichier de *log*.

Après ces améliorations, la deuxième version du script était capable de lire des fichiers de configuration comme celui de l'exemple suivant :

```
[Business days]
    days = NOT Holidays
    hours = 8 to 9
    miriel_margin = 4
    bora_margin = 2

[Holidays]
    days = 2022/07/15 to 2022/08/15, Sundays
    #hours = 9 # hours isn't defined -> section active all day
    miriel_margin = 1
```

Si nous sommes le 2 Septembre 2022 à 8h30, la section **Business days** sera sélectionnée car l'inverse de la variable **date_range** de la section **Holidays** est vrai et que nous sommes entre 8 et 9 heures. Toutes les variables finissant par **_margin** seront lues et pour chacune d'entre elles, un **srun** sera exécuté. Dans notre exemple, un *job* demandant 4 nœuds **miriel** et un *job* demandant 2 nœuds **bora** seront lancés.

Rapidement après, **M. GOGLIN** m'a soumis d'autres ajouts à intégrer aux scripts afin qu'ils soient plus faciles à déboguer. Cela consistait à ajouter une option **–check-config** pour vérifier la validité du fichier de configuration et une option **–dry-run** afin d'afficher les actions des scripts sans les exécuter.

3.7.2 Marge de réactivité : version finale

Pour la dernière version du script, j'ai décidé de l'améliorer autant que possible que ce soit au niveau des performances ou de ses fonctionnalités. Afin de faciliter la lecture, voici une liste des optimisations réalisées :

- **limiter le nombre de subprocess utilisés** : seuls les **srun** utilisent **subprocess**, les recherches dans les fichiers sont faites sans faire appel à **grep**. Le fichier est ouvert à la place ;
- **revoir la structure des boucles et des *if*** afin d'accélérer les traitements. Pour réaliser ces optimisations, je me suis basé sur les notions apprises en cours de projet d'algorithmique numérique qui nous présentait certaines optimisations de boucles et de conditions (notamment sortir les conditions le plus possible des boucles) ;
- **utiliser les list comprehensions de PYTHON** pour créer mes listes au lieu des boucles traditionnelles ;
- **tuer les srun 1 seconde après leur lancement** [11] afin d'éviter qu'ils ne s'accumulent dans la file d'attente et ennuient les utilisateurs.

Et voici une liste des changements fonctionnels de la version finale :

- **les srun utilisés portent le nom « keepIdle »** afin d'être plus reconnaissables et utilisent l'option -X afin de s'interrompre dès le premier signal **SIGINT** ;
- **ajout d'une variable optionnelle keep_nodes** dont la valeur est une liste de nœuds au format **SLURM**. Un *job* sera créé pour chaque nœud de cette liste afin de les maintenir éveillés s'ils sont inactifs ;
- **ajout d'une nouvelle option (–set-time aaaa/mm/jj HH:MM) aux deux scripts**. Cela permet de régler leur horloge interne et facilite leur débogage (surtout couplé avec l'option –**dry-run**).

Ensuite, j'ai effectué une refonte totale du système de **date_range** et de **hour_range** pour donner un contrôle extrêmement poussé à l'utilisateur. Maintenant, la valeur de **date_range** est une expression booléenne. Les opérateurs **OR, AND** et **NOT** sont disponibles et leur évaluation est paresseuse. La priorité des opérateurs est (du plus haut au plus bas) : **NOT->AND->OR**. Les utilisateurs ont accès à différentes expressions :

- **une date seule** (au format aaaa/mm/jj) ;
- **deux dates séparées par « to »** ;
- **un jour de la semaine (au pluriel)** ;
- **le nom d'une autre section** pour représenter sa **date_range**.

Il est possible de mettre des parenthèses dans les expressions mais elles n'ont aucun sens pour le script, elles sont là pour faciliter la lecture des expressions. Le plus dur dans cette refonte à été d'écrire le *parser* (l'analyseur syntaxique) de ce pseudo-langage. Comme je ne voulais pas utiliser de module, ce qui aurait alourdi le code avec des grammaires compliquées, j'ai écrit mon propre *parser* dont voici l'algorithme :

11. À l'aide d'un signal **SIGINT** envoyé au processus.

Algorithm 3 *Parser* de la variable **date_range**

```
1: function CHECK_AND_OPERAND(and_operand)
2:     expression ← and_operand.split("NOT")
3:     if len(expression) == 2 then
4:         return not evaluate_expression(expression[1])
5:     return evaluate_expression(expression[0])
6:
7: function EVALUATE_OR_OPERAND(or_operand)
8:     for and_operand in or_operand.split(" AND ") do
9:         if not evaluate_and_operand(and_operand) then
10:            return False
11:    return True
12:
13: function PARSE_DATE_RANGE(section)
14:    date_range ← config[section]["date_range"]
15:    for or_operand in date_range.split(" OR ") do
16:        if evaluate_or_operand(or_operand) then
17:            return True
18:    return False
```

L'astuce réside dans le fait qu'il n'y ait pas de parenthèses (bien qu'on puisse les simuler en utilisant les noms de sections). Il suffit alors de séparer toutes les opérandes en suivant la priorité opératoire avant d'évaluer les expressions. La part la plus maligne du système se trouve dans l'analyse des expressions. L'algorithme 4 résume le fonctionnement de cette analyse.

Algorithm 4 Fonction servant à évaluer les expressions de base

```
1: function EVALUATE_EXPRESSION(expression)
2:     weekdays ← ["mondays","tuesdays","wednesdays","thursdays","fridays","saturdays",
   "sundays"]
3:     if expression.lower() in weekdays then        ▷ L'expression représente un jour de la semaine
4:         return weekdays.index() == now.weekday()
       ▷ Une expression sans « / » ou ne commençant pas par un chiffre n'est pas une date, c'est un
   nom de section
5:     if not "/" in expression or not expression[0].isdigit() then
6:         return parse_date_range(expression)
7:     return check_day(expression)          ▷ Vérifie que le jour actuel est dans la plage de dates
   représentée par expression
```

Le plus dur a été de savoir comment reconnaître le type d'une expression entre une date, un nom de section ou un jour de la semaine. J'ai commencé par tester les jours de la semaine car il suffit de tester l'appartenance de l'expression à une liste de 7 éléments. Ensuite il fallait faire la différence entre une date et un nom de section pouvant contenir des espaces, des « / », etc. L'astuce réside dans le fait qu'une expression sans « / » ou ne commençant pas par un nombre ne pourra jamais être une chaîne de date valide. Par conséquent, il faut traiter l'expression comme un nom de section et relancer le *parser* avec le nom de section représenté par l'expression. Enfin, par élimination, si aucune des conditions précédentes n'est remplie, l'expression est une date ou deux dates séparées par « to ».

De nombreuses difficultés sont venues avec le nouveau format, notamment la gestion des erreurs. En effet, il faut vérifier que les expressions ne sont pas vides, que les dates sont dans le bon format, que les noms de sections existent et soient valides, etc. Générer un message d'erreur approprié à chaque cas peut vite devenir difficile et le code peut perdre en lisibilité. Afin de gérer efficacement les erreurs,

la fonction **parse_date_range** est celle qui rattrape toutes les *exceptions* renvoyées par les autres fonctions en aval et se charge de les afficher. Ainsi, à chaque fois qu'une erreur est détectée, les fonctions plus spécialisées se contentent de lever une *exception*. Ensuite, avec un langage aussi complexe, il existe de très nombreuses façons de faire dérailler le *parser* que j'ai écrit. Dans le but d'éviter cela, j'ai passé beaucoup de temps à tester toutes les variations les plus étranges qu'un utilisateur puisse écrire dans la valeur de **date_range**. À chaque fois, je découvrais un nouveau cas que je n'avais pas géré. Que ce soit l'utilisation d'opérateurs dans le nom des sections ou les problèmes liés à des noms de sections ressemblant fortement à des dates ; la gestion de chaque nouveau cas devenait de plus en plus dure. D'autant plus que l'évaluation paresseuse faisait que certaines erreurs n'étaient pas détectées si elles étaient dans la partie ignorée par l'évaluation de l'expression. Dans le même temps, il fallait que je vérifie bien que l'option permettant de vérifier la configuration détecte les nouveaux bogues que je trouvais. J'ai dû implémenter cette option de sorte que l'évaluation de la variable **date_range** ne soit pas paresseuse afin de ne rater aucune erreur.

Des problèmes similaires sont apparus lors de la refonte du *parser* de **hour_range** mais en moins problématique. En effet, j'ai réécrit le système d'heures afin qu'il soit possible d'écrire des plages d'heures au format HH:MM to HH:MM. Cela permet de contrôler l'activation des sections à la minute près. De plus, l'opérateur **OR** est disponible, permettant une encore plus grande flexibilité dans la définition des périodes.

Ce *parser* est bien moins sophistiqué, il sépare la chaîne à partir des « **OR** » et évalue les heures comme avant. La seule chose qui change est le format de lecture de la chaîne que je donne à **datetime** qui incorpore les minutes. Par ailleurs, afin de me simplifier la tâche, à chaque invocation du script, la date actuelle que je récupère est arrondie à la minute près (les secondes sont mises à 0).

Pour conclure cette partie sur la marge de réactivité, le script est capable de lire des fichiers de configuration très complexes (dont un exemple est donné en annexe, section B), de sélectionner la première section contenant la date et l'heure actuelle afin d'appliquer la marge décrite dans ladite section. Cette marge peut contenir un nombre de nœuds par groupe souhaité ou une liste de nœuds spécifiques à laisser allumés. Pour réaliser cette marge, **reactivity_margin.py** lance les *jobs* nécessaires. Bien que ces derniers n'exécutent rien et terminent en moins d'une seconde, le script possède un garde-fou l'empêchant de générer des *jobs* inutilement qui viendraient engorger la file d'attente. Son code est documenté, séparé en parties thématiques et dispose d'un système de vérification d'erreur très fort, capable d'intercepter et de prévenir toute mauvaise manipulation de la configuration.

3.7.3 Délais personnalisés

Le développement de ce script s'est fait en parallèle que **reactivity_margin.py** mais le chemin fut moins laborieux.

Très rapidement, l'algorithme du script m'est venu en tête ainsi que le format du fichier de configuration.

Pour ce dernier, chaque section aurait comme nom une liste de nœuds **SLURM** et définirait au moins une variable temporelle parmi les deux disponibles. Ces variables sont **days** et **hours**. Elles désignent le temps de la période de maintien en éveil des machines concernées par la section en jours et en heures.

Le premier problème s'est vite présenté puisque les listes de nœuds **SLURM** contiennent des crochets ce qui entre en conflit avec la syntaxe du format INI où le nom des sections se trouve entre crochets. Heureusement, j'ai trouvé rapidement de l'aide en ligne pour régler ce problème en changeant certaines options de **configparser**.

Pour l'algorithme, j'ai pensé à un système basé sur un enregistrement. Les machines présentes dans la configuration sont surveillées et lorsque le script **custom_node_timeout.py** détecte la mise sous tension d'une des machines à surveiller, il lit la valeur en heures et en jours associée à sa section dans le fichier de configuration et ajoute à l'heure actuelle le nombre correspondant de jours et d'heures. La date résultante est enregistrée dans un fichier : **keep_awake_deadlines.txt**. Il s'agit de la *deadline* du nœud. Tant que l'heure actuelle est inférieure ou égale à cette *deadline*, le script enverra un petit *job*

rien qu'à ce nœud afin de le garder éveillé. Si au contraire, la *deadline* est dépassée, elle est supprimée du registre.

Pour fonctionner, ce script a besoin de connaître l'état des nœuds afin de savoir si l'une des machines à surveiller est allumée. Pour ce faire, j'ai choisi d'appeler la commande **sinfo -N -o %N=%t** à l'aide de **subprocess**. Ceci me donne l'état de tous les nœuds sous la forme <nœud>=<état>. Pour savoir si une machine est allumée, le script cherche l'absence des symboles « % » et « ~ » dans l'état de cette dernière.

L'implémentation de la première version s'est passée sans trop de difficultés. Après avoir développé la deuxième version de **reactivity_margin.py**, j'ai remarqué quelques problèmes dans mon script et j'ai commencé à corriger et à améliorer ce dernier.

Cela a induit les changements suivants :

— la fonction **erase_exceeded_deadlines** retire, au fur et à mesure, les nœuds de la liste lorsque leur *deadline* est supprimée afin d'accélérer la procédure ;
— les *jobs* ne sont envoyés que si la machine est inactive (état **idle**) ;
— les *jobs* créés sont nommés « **keepIdle** » afin de les identifier facilement.

NB : Contrairement au script précédent, les jobs générés ne sont pas tués car ils sont envoyés alors que la machine est disponible. Par conséquent, le seul moyen qu'ils soient bloqués dans la file d'attente est que quelqu'un ait réservé le nœud en même temps que mon script ce qui ne devrait pas arriver souvent.

Ensuite, j'ai remarqué un bogue important dans mon script. Comme ce dernier est invoqué par un **cron** toutes les minutes, il ne se passe que deux minutes entre le moment où la machine reçoit son dernier petit *job* (lorsque $now == deadline$) et le moment où elle est de nouveau réenregistrée. Cela ne laisse pas assez de temps aux machines pour atteindre le **SuspendTime** et elles restent allumées, prisonnières d'un cycle infini où elles sont toujours enregistrées. Dans le but de régler ce problème, j'ai instauré une date d'expiration sur les *deadlines* égales à $deadline + SuspendTime + expiration\_margin$. Les *deadlines* ne sont supprimées qu'après leur expiration ce qui fait qu'aucun *job* n'est envoyé entre le moment où la *deadline* est passée et le moment où elle est expirée. Par conséquent, il faut que la *deadline* soit supprimée puis que la machine soit de nouveau enregistrée pour recevoir à nouveau des *jobs* de la part du script.

Le paramètre « expiration_margin » permet d'ajouter un délai supplémentaire à l'expiration des *deadlines* afin de laisser une machine s'éteindre même si sa dernière activité dépassait un peu sa *deadline*. Cela permet d'éviter de repartir pour un autre enregistrement où une machine serait maintenue éveillée inutilement pendant des heures parce-qu'elle aurait été utilisée un peu plus que prévu sans pour autant être réellement demandée par la suite. Vous pourrez trouver en annexe, section C, un exemple de situation illustrant le fonctionnement de cette marge.

Avant de conclure cette partie, voici l'algorithme de **custom_node_timeout.py** dans sa version finale :

Algorithm 5 Délais personnalisés des nœuds - version finale

```
1: function MAIN
2:     nodes_state ← sinfo()
3:     suspend_time ← get_suspend_time()  ▷ Nécessite de lire dans la configuration de SLURM
4:     deadlines ← get_deadlines()
5:     exceeded_deadlines ← []
6:     for section in config do
7:         node_list, days, hours ← get_section_information(section)
8:         for node in node_list do
9:             node_deadline ← get_node_deadline(deadlines, node)
10:            if is_powered_up(node, nodes_state) and node_deadline == None then
11:                register_node_deadline(node, days, hours)
12:                if is_idle(node, nodes_state) then ▷ Pour empêcher la machine de s'éteindre avant
   la prochaine invocation du script
13:                    execute_srun(node)
14:            else if node_deadline != None and now <= node_deadline and is_idle(node,
   nodes_state) then
15:                execute_srun(node)
16:            else if node_deadline != None and now >= get_deadline_expiration(node_deadline,
   suspend_time, expiration_margin) then
17:                exceeded_deadlines.append(node)
18:    erase_exceeded_deadlines(exceeded_deadlines)
```

La gestion des erreurs de ce script était beaucoup plus facile à mettre en oeuvre par rapport au script précédent. En effet, seules deux variables sont à vérifier : **days** et **hours** et les autres variables présentes sont ignorées. Il faut également s'assurer que les valeurs de ces variables soient des entiers. Pour cela, j'utilise **configparser** qui fournit la méthode **getint** permettant de récupérer la valeur d'une variable et de la convertir en entier. Une *exception* est levée si la valeur n'est pas un entier et c'est cette *exception* que je récupère pour détecter toute erreur de type.

Pour conclure, tous les scripts que j'ai écrit ont utilisé le module **argparse** ce qui m'a permis de gérer facilement les arguments de la ligne de commande et de générer d'élégants messages d'aides. Je me suis efforcé au maximum de rendre le code le plus lisible et maintenable possible. Pour cela, j'ai suivi au mieux les conventions PYTHON, j'ai documenté mon code et écrit une documentation complète se trouvant dans le même dossier que mes scripts. Ce document contient toutes les informations sur le fonctionnement de ces derniers ainsi que la procédure d'installation détaillée pour déployer le système avec facilité. Des conseils sont également donnés pour déboguer la configuration [12] ou pour effectuer certaines actions de maintenance comme supprimer la *deadline* d'un nœud en cours de production.

Pour continuer, j'ai fait en sorte que mes scripts produisent le moins de *jobs* possibles afin de limiter la charge sur **SLURM** et j'ai étudié leur comportement pour qu'ils soient les plus invisibles possibles aux yeux des utilisateurs. Les scripts ont été améliorés de sorte que l'exécution des deux en série dure moins d'une demi-seconde. Enfin, je les ai longuement testé afin de gérer le maximum de cas d'erreurs possibles pour que toute panne soit repérée rapidement à l'aide de messages d'erreurs pertinents. À la fin, le script principal n'écrit des *logs* qu'en cas d'erreur afin de ne pas générer des Giga-octets de fichiers journaux inutilement.

3.7.4 Limites et perspectives d'améliorations

Malgré tout, le système que j'ai implémenté présente des limites :

— **Augmentation du compteur central de *jobs* SLURM : le job ID** dû au fait que le système s'appuie sur la création de *jobs* ;

12. Avec l'aide des options –**check-config,** –**dry-run** et –**set-time** disponibles sur les deux scripts.

— **Pollution des statistiques d'utilisation SLURM** : pour les mêmes raisons que précédemment. Cependant, il est très facile de filtrer les *jobs* lancés par **SLURM** ou portant le nom « **keepIdle** » ;
— **Augmentation de la taille de la base de données contenant les statistiques SLURM** : il pourrait s'avérer nécessaire de retirer régulièrement les entrées concernant les *jobs* **keepIdle** afin de limiter l'impact sur les performances lors du traitement de la base ;
— **Incertitude quant au passage à l'échelle sur de très grosses plate-formes**. Chaque *job* constitue une requête à la machine **SLURM**. En cas d'un afflux soudain et important de *jobs*, cette dernière pourrait ne pas suivre les demandes. Ce risque est davantage présent s'il est demandé au système de maintenir éveillé de nombreux nœuds différents.

Il est également possible d'améliorer le confort utilisateur en créant un système permettant de suivre la progression de l'allumage des différentes machines demandées lors d'une allocation. Il serait aussi intéressant de communiquer les temps de démarrage de chaque type de nœud afin de permettre aux utilisateurs d'avoir une estimation quant au démarrage de leur *job*.

Enfin, j'ai développé les scripts **reactivity_margin.py** et **custom_node_timeout.py** parce que je n'ai pas trouvé de moyens d'accéder aux structures internes de **SLURM** contenant les chronomètres d'inactivité des machines. De plus, il me semblait impossible d'obtenir un système fonctionnel dans le temps imparti s'il fallait modifier le code source de **SLURM**.

Par conséquent, une grande amélioration serait de porter le système des scripts PYTHON vers les sources de **SLURM**, ou mieux, de créer un *plugin* **SLURM**. Cela rendrait le système plus transparent et faciliterait sa distribution. En effet, la création de marges de réactivité est demandée par la communauté **SLURM** mais aucune solution publique n'existe.

4 Estimation de la quantité d'énergie économisée

Pendant les derniers jours de mon stage, j'ai tenté de donner une première estimation de l'énergie économisée si le système d'arrêt des nœuds inactifs était mis en place.

Pour réaliser mes calculs, je me suis basé sur les statistiques produites en section 3.6.1. Pour rappel, avant de faire appel à **pivot_table**, je dispose du nombre total de nœuds utilisés par groupe pour chaque heure dans l'année.

J'ai commencé par retirer le critère **IsHoliday** du *dataframe* afin de considérer l'utilisation sur toute l'année. Ensuite, j'ai calculé le nombre moyen de nœuds utilisés par groupe dans une journée de l'année 2021. Il m'a suffit de regrouper mes données par les valeurs de **NodeGroup** et d'appliquer la moyenne (avec un **groupby(["NodeGroup"].mean()**) pour arriver à mes fins.

Après avoir calculé le nombre moyen de nœuds utilisés par groupe pour une journée de 2021, je pouvais déduire le nombre de nœuds inutilisés en moyenne chaque jour (arrondi à l'entier supérieur). En me basant sur la consommation de **bora040** en période d'inactivité qui est de **80 W** et n'ayant pas l'information pour les autres groupes de nœuds, j'ai fait mes estimations en faisant l'hypothèse que tous les nœuds consomment en moyenne **80 W** lorsqu'ils sont inactifs.

Sachant qu'il y a $24 * 365 = 8760$ heures dans une année et connaissant le nombre moyen de machines inutilisées chaque jour, nous pouvons estimer la quantité d'énergie gaspillée grâce à la formule : $nombre\_machines\_inutilisees * 80 * 8760$. Le résultat de la formule nous donne le nombre de Wh consommés chaque année pour alimenter les machines inactives. Il suffit alors de diviser le résultat par 1000 pour convertir les Wh en kWh et en considérant que 1 kWh était facturé **0,11 €** en 2021 [13], il est possible de chiffrer le prix de l'électricité gaspillée en alimentant les machines inactives.

Enfin, j'ai réalisé différentes simulations pour mesurer l'impact de **SuspendTime** sur les économies réalisées. Pour modéliser un **SuspendTime** d'une heure, il me suffit d'ajouter une heure à tous les *jobs* et de considérer que **SuspendTime** vaut 0. Chaque machine est alors comptée dans un créneau horaire supplémentaire comme si elle était inactive, en attente de son extinction.

13. Ce tarif sera très probablement revu à la hausse en 2022.

Le tableau suivant représente le taux d'utilisation moyen des machines suivant leur groupe pendant l'année 2021 en fonction de **SuspendTime**.

SuspendTime	bora	diablo	kona	miriel	sirocco	zonda	kWh économisés	€ gagnés
0	50%	23%	25%	20%	40%	39%	92 505	10 175,62
1 heure	57%	34%	25%	21%	44%	39%	88 300	9 713,09
2 heures	62%	34%	25%	22%	44%	43%	85 497	9 404,74
4 heures	69%	34%	25%	24%	48%	48%	80 592	8 865,12

TABLE 1 – Taux d'utilisation des machines sur une journée moyenne de 2021 en fonction de **SuspendTime**

Les valeurs des groupes **arm, brise, mistral, souris** et **visu** ont été omises car ces groupes ne contiennent qu'une ou deux machines.

5 Bilan

En résumé, voici les différentes stratégies d'économie d'énergie applicables sur **PlaFRIM** :

Stratégie	Mise en place	Impact
Éteindre les nœuds inutilisés.	Mise en place détaillée en section 3.6.	Impact élevé dû au nombre élevé de nœuds inactifs et à leur consommation.
Abaisser légèrement la fréquence des nœuds en activité.	Côté administrateur : **sudo CPUpower frequency-set -u** <**freq**> pour fixer la fréquence maximale de manière permanente. Côté utilisateur : utiliser –**cpu-freq HighM1** avec **salloc** / **srun** afin de fixer la fréquence à son maximum moins un palier.	Impact fort sur les machines capables de monter haut en fréquence.
Placer le *governor* en **powersave** lorsque les machines sont inactives.	Modification des prologues / éplilogues administrateurs : **PrologSlurmctld** et **EpilogSlurmctld**.	Impact faible, permet une stabilisation de la fréquence au minimum lorsque la machine est inactive.
Faire de l'*overprovisioning* sur certains *jobs*.	Identifier les *jobs* adéquats et leur donner plus de ressources que nécessaire mais réduire leur fréquence.	Impact faible, n'est applicable que sur certains *jobs* et tous les *jobs* ne s'y prêtent pas.
Réserver en priorité les machines proches de la climatisation.	Identifier les machines en question et modifier la politique de **SLURM** pour les allouer en priorité.	Impact faible car toutes les machines d'un même bloc partagent le même couloir chaud.

TABLE 2 – Tableau récapitulatif des différentes stratégies d'économie d'énergie applicables sur **PlaFRIM**

Ensuite, voici les différentes stratégies d'économie d'énergie non-applicables sur **PlaFRIM** :

Stratégie	Raison de sa non-application
Changer la politique d'allocation de **SLURM** pour trouver la meilleure combinaison machine / *job*.	Impossibilité de changer les conditions expérimentales des utilisateurs.
Mettre les liens réseaux inutilisés en mode basse consommation.	Nécessite que tous les équipements réseaux implémentent le standard *Energy Efficient Ethernet* ou que les liens réseaux possèdent un mode économie d'énergie.
Placer les *jobs* sur des machines virtuelles afin de garder moins de nœuds actifs.	Met en péril la reproductibilité des expériences ainsi que leur bon déroulement.
Déplacer les *jobs* nécessitant une faible puissance de calcul sur des machines peu puissantes.	Impossibilité de changer les conditions expérimentales des utilisateurs et nécessité d'estimer les besoins du *job* en avance.

TABLE 3 – Tableau récapitulatif des différentes stratégies d'économie d'énergie non-applicables sur **PlaFRIM**

Enfin, la majorité des objectifs principaux du stage ont été atteints. En six mois, j'ai pu :
— Étudier différentes stratégies d'économie d'énergie. Ces dernières sont résumées plus haut ;
— Analyser les différences de consommation sur les machines qui m'étaient données sur **Formation** en fonction de leur activité ;
— Faire une analyse fine des *governors* et de l'impact de la fréquence CPU sur la consommation électrique ;
— Maîtriser les fonctionnalités de **SLURM** afin d'éteindre les nœuds inactifs ;
— Estimer les gains potentiels en terme d'énergie suite à l'application de l'arrêt des nœuds inactifs.

Tous les objectifs secondaires ont été accomplis, à savoir :
— Extraire des statistiques d'utilisation avancées de la plate-forme à partir des traces **SLURM** (objet de la section 3.6.1) ;
— Comparer les outils de mesure de la consommation électrique (traité en section 3.2.2) ;
— Mise en place d'un système de marge de réactivité et de délais d'inactivité personnalisés pour les nœuds en complément du système d'extinction / allumage fourni par **SLURM** afin d'améliorer le confort utilisateur (les détails sont donnés en section 3.7).

Enfin, parmi toutes les stratégies étudiées, seule la plus importante a pu être mise en place : l'arrêt des nœuds inactifs complété par le système de marge de réactivité et de délais personnalisés. Ceci fut ma priorité afin de m'assurer de l'impact de mes travaux après mon départ. D'après mes premières estimations, ceci devrait permettre au centre d'économiser environ **90 000 kWh / an** en fixant **SuspendTime** autour de 1 heure.

6 Conclusion

En définitive, ce stage m'a beaucoup appris. J'ai pu découvrir l'informatique sous un angle nouveau qu'est la consommation électrique des machines. J'ai appris le fonctionnement des mécanismes connus du grand public tels que les mécanismes de veille (les *S-states*) et les modes « d'économie d'énergie » (les *governors* et leur lien avec la fréquence du processeur). J'ai pu également me renseigner sur les pratiques actuelles en matière de sobriété énergétique au sein même des processeurs (les *C-states*) et des plus grands supercalculateurs / *data centers*.

Ensuite, j'ai pu m'intéresser aux outils servant à mesurer la consommation électrique des machines avec leurs différents avantages et inconvénients. J'ai été également confronté aux difficultés que pose la mesure de la fréquence CPU.

Ce stage m'a également permis d'agrandir mes connaissances en PYTHON, que ce soit pour gérer correctement les arguments de la ligne de commande avec **argparse**, pour manipuler de grands volumes de données avec **pandas** ou lire des fichiers de configuration avec **configparser**. Je suis resté dans l'esprit de mon stage de première année de master en fournissant un code clair et documenté afin qu'il soit facilement repris et étendu par la suite. Pour continuer, ce stage m'a également permis d'affûter grandement ma maîtrise de **SLURM**, un ordonnanceur très répandu dans le domaine de ma spécialité : le calcul haute performance. J'ai pu développer des compétences en tant qu'utilisateur et les connaissances que j'ai acquises sur son fonctionnement me permettent de pouvoir le configurer plus efficacement ainsi que de pouvoir contribuer au projet via la création de *plugins*.

Par ailleurs, j'ai été immergé dans le monde de la recherche qui était nouveau pour moi jusqu'à cette année. Je suis reconnaissant des opportunités qui m'ont été offertes au contact des chercheurs, des doctorants et de mes collègues pour comprendre leur métier, leurs recherches, leurs enjeux ainsi que les joies et les déboires qui sont associés à cette activité.

Enfin, bien que mon projet professionnel ne s'oriente pas davantage dans la recherche, j'espère que mon passage chez **Inria** permettra au laboratoire de réduire sa consommation électrique et que les utilisateurs de **PlaFRIM** seront enthousiastes vis-à-vis des changements apportés à la plate-forme.

Bibliographie

[1] Marco D'Amico et Julita Corbalan Gonzalez. « Energy hardware and workload aware job scheduling towards interconnected HPC environments ». In : *IEEE Transactions on Parallel and Distributed Systems* (2021). [Dernier accès : 14 août 2022]. URL : https://arxiv.org/pdf/2106.12007.pdf.

[2] golinuxhub. *What are the CPU c-states ? How to check and monitor the CPU c-state usage in Linux per CPU and core ?* [Dernier accès : 14 août 2022]. 2018. URL : https://www.golinuxhub.com/2018/06/what-cpu-c-states-check-cpu-core-linux/.

[3] M. Hervé. *Energy Scope : a tool for measuring the energy profile of HPC and AI applications.* [Dernier accès : 14 août 2022]. URL : https://sed-bso.gitlabpages.inria.fr/datacenter/energy_scope.html.

[4] Tapasya Patki, David K. Lowenthal, Barry Rountree, Martin Schulz et Bronis R. de Supinski. « Exploring hardware overprovisioning in power-constrained, high performance computing ». In : *ICS '13*. [Dernier accès : 14 août 2022]. 2013. URL : https://digital.library.unt.edu/ark:/67531/metadc832063/m2/1/high_res_d/1084707.pdf.

[5] Issam Raïs, Anne-Cécile Orgerie, Martin Quinson et Laurent Lefèvre. « Quantifying the Impact of Shutdown Techniques for Energy-Efficient Data Centers ». In : *Concurrency and Computation : Practice and Experience* 30.17 (2018). [Dernier accès : 14 août 2022], p. 1-13. DOI : 10.1002/cpe.4471. URL : https://hal.archives-ouvertes.fr/hal-01711812/file/paper-wiley.pdf.

[6] Huigui Rong, Haomin Zhang, Sheng Xiao, Canbing Li et Chunhua Hu. « Optimizing energy consumption for data centers ». In : *Renewable and Sustainable Energy Reviews* 58 (2016). [Dernier accès : 14 août 2022], p. 674-691. ISSN : 1364-0321. URL : https://www.sciencedirect.com/science/article/pii/S1364032115016664.

[7] Karthikeyan Saravanan et Paul Carpenter. « PerfBound : Conserving Energy with Bounded Overheads in On/Off-Based HPC Interconnects ». In : *IEEE Transactions on Computers* PP (jan. 2018). [Dernier accès : 14 août 2022]. DOI : 10.1109/TC.2018.2790394. URL : https://www.researchgate.net/publication/322322342_PerfBound_Conserving_Energy_with_Bounded_Overheads_in_OnOff-Based_HPC_Interconnects.

[8] Karthikeyan Saravanan, Paul Carpenter et Alex Ramírez. « Exploring Multiple Sleep Modes in On/Off based Energy Efficient HPC Networks ». In : [Dernier accès : 14 août 2022]. Oct. 2015. URL : https://www.researchgate.net/publication/283121952_Exploring_Multiple_Sleep_Modes_in_OnOff_based_Energy_Efficient_HPC_Networks.

[9] SchedMD. *Slurm Power Saving Guide.* [Dernier accès : 14 août 2022]. 2022. URL : https://slurm.schedmd.com/power_save.html.

[10] Jan Treibig, Georg Hager et Gerhard Wellein. *LIKWID : A lightweight performance-oriented tool suite for x86 multicore environments.* [Dernier accès : 14 août 2022]. 2010. URL : https://arxiv.org/abs/1004.4431.

[11] Rafael J. Wysocki. *CPU Idle Time Management.* Documentation noyau Linux : les cstates [Dernier accès : 14 août 2022]. 2018. URL : https://www.kernel.org/doc/html/latest/admin-guide/pm/cpuidle.html.

[12] Rafael J. Wysocki. *CPU Performance Scaling.* Documentation noyau Linux : CPUFreq [Dernier accès : 14 août 2022]. 2017. URL : https://www.kernel.org/doc/html/v4.14/admin-guide/pm/cpufreq.html.

[13] Rafael J. Wysocki. *System Sleep States.* Documentation noyau Linux : les sstates [Dernier accès : 14 août 2022]. 2017. URL : https://www.kernel.org/doc/html/v4.18/admin-guide/pm/sleep-states.html.

A Glossaire

— **CHP** (HPC (en)) :
Calcul haute performance ou *High Performance Computing* en anglais. C'est un domaine de l'informatique visant à optimiser les applications pour quelles soient les plus efficaces possibles (en temps, en mémoire, etc).

— **Nœud de calcul** :
Une machine sur laquelle s'exécute une application. Elle peut contenir des GPU, des CPU, plus ou moins de RAM, etc.

— **Ordonnancement / ordonnanceur** :
L'ordonnanceur est le programme qui gère le placement des *jobs* dans la file d'attente en vue de leur exécution. L'ordonnancement est le fait de choisir la place d'un *job* dans la file d'attente ainsi que les machines qui lui seront allouées tout en respectant les contraintes données par l'utilisateur.

— **Parser** :
Terme couramment utilisé pour désigner l'analyse d'une expression afin de récupérer des informations ou de savoir si elle correspond à une certaine syntaxe.

— **Plate-forme de calcul** (*Cluster* (en)) :
Ensemble de machines interconnectées dont le but principal est d'exécuter des applications très demandeuses en temps et en ressources.

— **Tâche** (*Job* (en)) :
Une tâche définit l'application à exécuter ainsi que les contraintes (matérielles, logicielles, durée, etc) relatives à cette dernière. Les tâches sont placées dans une file jusqu'à leur exécution.

— **Vectorisation** :
La vectorisation consiste à vectoriser les traitements. C'est-à-dire, appliquer le même traitement à plusieurs données en une seule instruction au lieu d'une instruction par donnée.

B Fichier de configuration - zoom sur la marge de réactivité

Imaginons que nous sommes le 4 Septembre 2022 à 8h30. Voici un exemple de configuration pour la marge de réactivité.

```
[My section]
    date_range = 2022/01/01 to 2022/04/20 AND Tuesdays OR 2022/12/25
    hour_range = 8:00 to 17:30 OR 22:14 to 23:59
    group1_margin = 4 # will keep 4 nodes of group1 idle
    group2_margin = 2

[Other section]
    date_range = Saturdays OR Sundays
    #hour_range = 4:00 to 5:00 # no hour_range defined -> section valid all day
    group1_margin = 6
    keep_nodes = group2[1-4],my_node04

[No margin]
    date_range = My section OR Mondays AND 2022/01/17 AND NOT Other section
    # no margin variables nor keep nodes -> no nodes will be kept idle
```

Dans cet exemple, la section **My section** est lue en premier. La variable **date_range** est évaluée comme (2022/01/01 to 2022/04/20 AND Tuesdays) OR 2022/12/25. Nous ne sommes pas entre le 1er Janvier 2022 et le 20 Avril 2022 inclus donc le AND est évalué à faux sans regarder si nous sommes un mardi. L'autre partie du OR est ensuite évaluée et comme nous ne sommes pas le 25 Décembre 2022, la **date_range** est évaluée à faux et le *parser* analyse la section suivante.

Dans la section **Other section**, la plage de dates est évaluée à vrai puisque le 4 Septembre 2022 est un dimanche. Le *parser* vérifie ensuite la plage horaire mais comme cette dernière n'est pas définie (puisque commentée), elle est automatiquement évaluée à vraie et la section est sélectionnée. Le script **reactivity_margin.py** va alors lancer un *job* contenant 6 machines du groupe **group1**, un *job* avec les machines 1 à 4 du groupe **group2** ainsi qu'un *job* allouant **my_node04**.

Ces petits *jobs* ne font rien et quittent la file dès que possible mais, grâce à eux, le chronomètre d'inactivité des machines concernées est réinitialisé !

C Le but de « expiration_margin »

Pour comprendre l'effet de la variable **expiration_margin** (qui représente un nombre de minutes) dans **custom_node_timeout.py**, il nous faut prendre un cas de base. Soit **SuspendTime** = 120 secondes et soit un nœud ayant un délai personnalisé d'une heure. La figure 26 montre le comportement de **SLURM** et du script lorsque le nœud n'est pas alloué lors de la fin de son enregistrement se trouvant à 9:00.

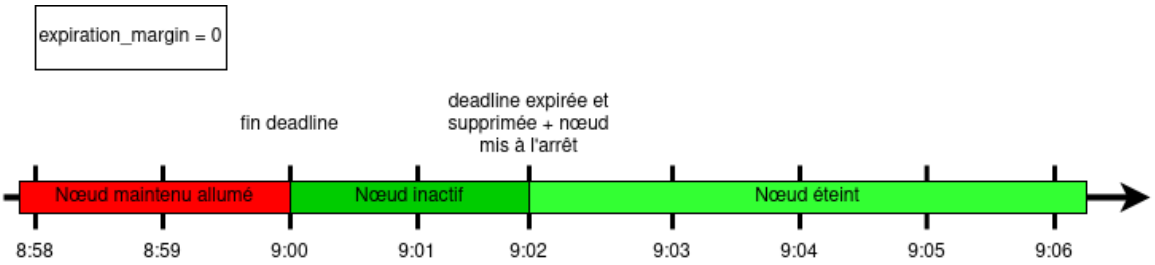

FIGURE 26 – Évolution de l'état d'un nœud inactif dont la période de maintien en éveil touche à sa fin

Lorsque **expiration_margin** est à 0, si le nœud est alloué un peu après la fin de sa *deadline*, il sera forcément réenregistré pour une nouvelle période de maintien en éveil comme le montre la figure 27. Cela pourrait maintenir le nœud allumé alors qu'aucun utilisateur ne souhaite s'en servir de si tôt.

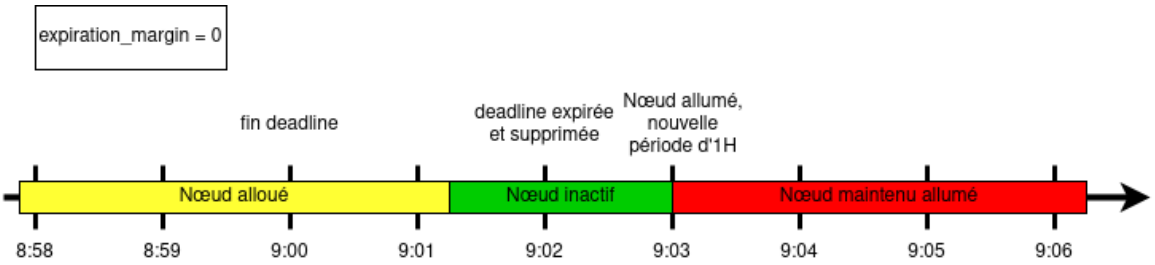

FIGURE 27 – Évolution de l'état d'un nœud alloué peu après la fin de sa période de maintien en éveil

Augmenter l'**expiration_margin** permet de rajouter un peu de délai afin de ne pas repartir inutilement dans une nouvelle période de maintien en éveil. Dans l'exemple suivant, il faut que le nœud soit actif après 9:03 pour obtenir un nouvel enregistrement. S'il n'est sollicité que jusqu'à 9:02, le nœud n'est pas réenregistré inutilement et s'éteindra 120 secondes plus tard si aucune allocation n'est faite.

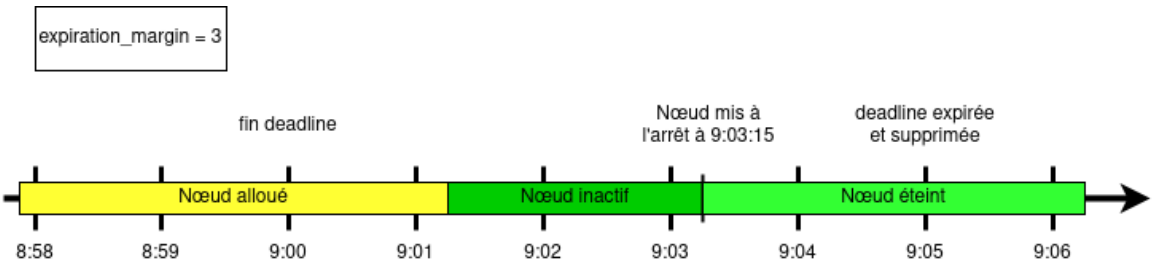

FIGURE 28 – Évolution de l'état d'un nœud alloué peu après la fin de sa période de maintien en éveil avec une **expiration_margin** égale à 3 minutes

D Résumé illustré des transformations appliquées aux statistiques de PlaFRIM

JobIDRaw	Start	End	NodeList
499167	2021-01-17T16:01:04	2021-01-17T17:01:04	diablo04,miriel[087-088]

↓ uncompress_slurm_nodelist()

JobIDRaw	Start	End	NodeList
499167	2021-01-17T16:01:04	2021-01-17T17:01:04	["diablo04","miriel087","miriel088"]

↓ compute_job_timeline()

IsHoliday	WeekNumber	Weekday	Hour	Node
False	2	6	16	["diablo04","miriel087","miriel088"]
False	2	6	17	["diablo04","miriel087","miriel088"]

↓ explode("Node")

IsHoliday	WeekNumber	Weekday	Hour	Node
False	2	6	16	diablo04
False	2	6	16	miriel087
False	2	6	16	miriel088
False	2	6	17	diablo04
False	2	6	17	miriel087
False	2	6	17	miriel088

↓ drop_duplicates()

[...]

↓ compute_node_group_and_node_count()

IsHoliday	WeekNumber	Weekday	Hour	NodeGroup	Count
False	2	6	16	diablo	1
False	2	6	16	miriel	1
False	2	6	16	miriel	1
False	2	6	17	diablo	1
False	2	6	17	miriel	1
False	2	6	17	miriel	1

↓ create_and_concat_minimal_dataframe()

[...]

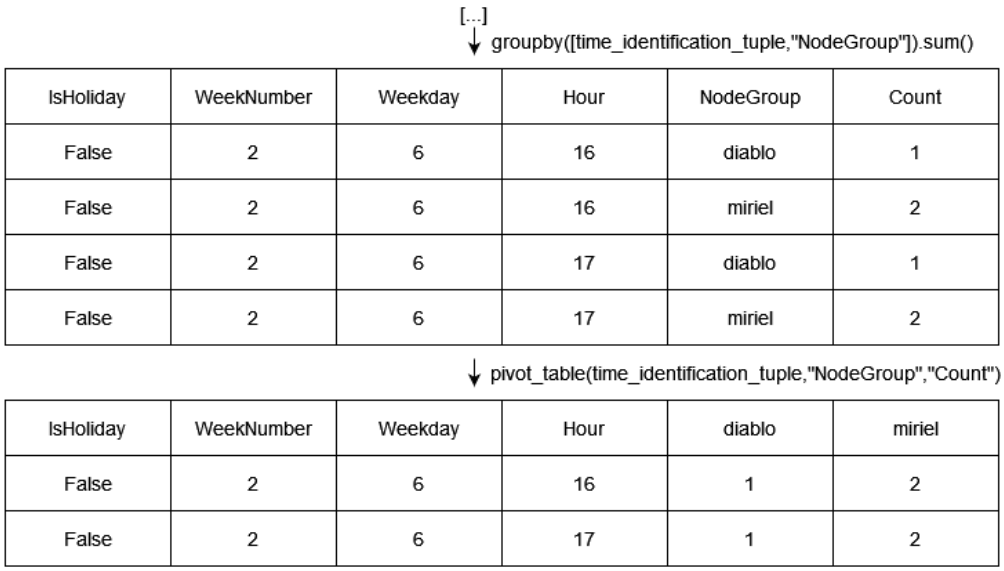

FIGURE 29 – Résumé illustré de l'extraction de l'utilisation des machines pour chaque heure de chaque jour à partir des statistiques de **PlaFRIM**