Élasticité des infrastructures HPC conteneurisées : résonance entre le HPC et le Cloud

Nicolas Greneche

Université Sorbonne Paris Nord
Laboratoire d'Informatique de Paris Nord (LIPN) - UMR CNRS 7030, France

Elasticité des infrastructures HPC conteneurisées : Résonance entre le HPC et le Cloud

Nicolas Greneche

Thèse de doctorat en Informatique

11 décembre 2023

Composition du Jury

Eric Renault	Professeur (LIGM, Université Gustave Eiffel)	Rapporteur
Christian Perez	Directeur de Recherche (LIP, INRIA)	Rapporteur
Christophe Cérin	Professeur (LIPN, Université Sorbonne Paris Nord)	Directeur
Roberto Wolfler-Calvo	Professeur (LIPN, Université Sorbonne Paris Nord)	Examinateur
Bruno Raffin	Directeur de Recherche (LIG, INRIA)	Examinateur
Alba Cristina Magalhaes Alves de Melo	Professeure (Université de Brasilia)	Examinatrice
Luciana Arantes	Maitre de conférence (Sorbonne université)	Examinatrice

REMERCIEMENTS

Je remercie mon directeur de thèse, Christophe Cérin, pour son encadrement, sa disponibilité et ses conseils durant l'ensemble du travail. Je mesure combien l'encadrement d'un doctorant demande d'investissement en temps et Christophe Cérin s'est toujours montré disponible quand j'en ai eu besoin, tout au long de mon parcours de trois ans. Il a su partager son expérience sur tous les aspects d'une thèse, depuis l'élaboration des idées jusqu'à la rédaction de solutions claires et compréhensibles par le plus grand nombre.

Je remercie Éric Renault et Christian Perez pour avoir accepté d'être les rapporteurs du manuscrit de thèse ainsi que pour les remarques constructives qu'ils ont tous deux émises. Je les remercie sincèrement de s'être acquitté de ces tâches.

Je remercie Roberto Wolfer-Calvo, responsable de l'équipe AOC du LIPN, pour avoir facilité mes démarches et s'être rendu disponible afin de présider le jury de soutenance. Je tiens également à remercier chaleureusement Bruno Raffin, Alba Magalhaes Alves de Melo ainsi que Luciana Arantes qui ont accepté d'examiner mon travail.

Je remercie mes collègues de la DSI pour leur soutien moral durant cette épreuve, et notamment Youssef Abdechchafiq, Saber Mghezzi Chaa, Mohamed Sait et Mourad Zerrouki. Je remercie également Dominique Bascle, mon responsable, qui m'a placé dans les meilleures conditions pour accomplir ce travail.

J'apprécie également en ce moment l'aide de Patrice Darmon et Tarek Menouer. C'est notamment grâce à Umanis / CGI que j'ai pu présenter nos travaux à Glasgow et recevoir le « Best Artifact Award ». Qu'ils en soient remerciés.

Enfin je remercie Mustapha Lebbah, mon tuteur qui m'a suivi sur ces trois ans ainsi que Julien Sopena et Lionel Pournin qui ont accepté de superviser mon travail de mi-parcours.

A la mémoire de Georges Constantinescu, surnommé affectueusement « Papa Georges » par ses étudiants dont j'ai eu le plaisir de faire partie.

TABLE DES MATIÈRES

CHAPITRE 1

PRÉAMBULE

FranceTerme est une base de données terminologique mise en place par la délégation générale à la langue française et aux langues de France, rattachée au ministère de la Culture. Cette base de données recueille les néologismes récents qui ont été validés par la Commission d'Enrichissement de la Langue Française (CELF) et publiés au Journal Officiel. Elle vise à remplacer les termes empruntés à d'autres langues, dans le but de favoriser l'usage de mots d'origine française. L'objectif premier de FranceTerme est de renforcer la langue française en évitant qu'elle ne perde du terrain à l'échelle mondiale. FranceTerme propose la définition de l'informatique suivante [1] :

Ensemble des opérations réalisées par des moyens automatiques, relatif à la collecte, l'enregistrement, l'élaboration, la modification, la conservation, la destruction, l'édition de données et, d'une façon générale, leur exploitation.

Cette définition est raccourcie dans le langage courant en *traitement automatique de l'information* qui perd un peu en précision en omettant les parties acquisition, transport et stockage.

Nous avons ici deux idées fortes, à savoir traitement automatisé et données. Un traitement (automatisé ou non) et une action d'un sujet utilisant des ressources pour agir sur un ou plusieurs objets. En informatique, le sujet est un processus, le matériel constitue les ressources et les données sont l'objet. Cependant, en filigrane de l'automatisation explicite, l'idée d'efficacité dans le traitement est sous entendue. Cette idée s'incarne dans la discipline du calcul à haute performance ou HPC pour « *High Performance Computing* ». Cette discipline propose aux processus des directives permettant de fédérer et d'agir finement sur les ressources. En conséquence, le HPC s'est construit comme un domaine très proche du matériel et le reste encore aujourd'hui. Chaque niveau d'indirection entre le processus et ses ressources diminue l'efficacité du traitement. Ainsi, l'adjonction de chaque couche logicielle tels que la virtualisation ou la conteneurisation doit donc être soigneusement pesée avant d'être appliquée.

1. https ://www.culture.fr/franceterme/terme/INFO240

Au niveau du matériel, la topologie des ressources du HPC est différente des machines classiques de type ordinateur personnel. La mémoire est plus abondante, les technologies d'interconnexion beaucoup moins sujettes à la latence i.e., délai de communication etc. Mais surtout, les files d'exécution (ou cœurs) offertes par les processeurs sont beaucoup plus nombreuses mais moins rapides. En effet, au moment de la rédaction de ce manuscrit les processeurs courant dans le HPC embarquent quelques dizaines de cœurs cadencés autour de 2,5 GHz alors que les processeurs à destination des particuliers disposent de moins d'une dizaine de cœurs mais cadencés à plus de 4 GHz. L'écart s'explique par le dégagement de chaleur induit par la densité. Les machines HPC (qu'on appelle également nœuds de calcul) peuvent exécuter de nombreux processus participant de façon concurrente au traitement. C'est ce qu'on appelle la parallélisation du calcul.

Dans un contexte HPC, les processus sont distribués par un ordonnanceur sur des machines dites machines parallèles. Un fois qu'ils sont arrivés sur la bonne ressource, tout l'art du calcul parallèle est d'orchestrer un accès intègre, c'est-à-dire respectant l'ordre souhaité des interactions sur les ressources par les processus concurrents du traitement travaillant sur les mêmes données. En effet, lorsque deux processus travaillent sur la même variable, il ne faudrait pas que l'un la modifie alors que l'autre est en train d'y accéder. La somme des spécificités aussi bien matérielles, logicielles ou d'usage font que ces machines sont extrêmement spécialisées donc peu accessibles aux populations peu versées dans l'informatique et plus particulièrement le calcul parallèle. En effet pour monter en puissance, les fabricants ont fait croître la fréquence des processeurs selon la loi de Moore [1] (le nombre de transistors double environ tous les deux ans), combiné avec le « Dennard Scaling » [2] (une loi d'échelle qui stipule que, lorsque les transistors deviennent plus petits, leur densité de puissance reste constante) résultant à ce que les performances par joule doublent environ tous les 18 mois, ce qui favorise l'apparition de co-processeurs.

Après avoir été cantonné aux sciences de la terre, essentiellement prévisions météo et étude du climat [3], et à l'astrophysique (étude des propriétés des objets de l'univers), le calcul scientifique s'est étendu à l'ingénierie mathématique (dynamique des fluides) et aux sciences du vivant. Ici, il s'agit non seulement de la chimie mais également de toutes les sciences dites omiques (la génomique, la protéomique, la métabolomique, la métagénomique et la transcriptomique). Le calcul haute performance est maintenant présent dans le domaine des sciences humaines et sociales mais aussi dans celles du monde naturel. C'est pour cette raison qu'au fil du temps des couches d'abstractions sont apparues pour gérer non seulement les problèmes d'accès concurrents aux données mais aussi les communications réseau ainsi que l'hétérogénéité des matériels. Ces abstractions troquent une « faible » quantité de performances pour une accessibilité accrue.

Un autre souci, conséquence directe de la spécialisation inhérente au calcul haute performance, est qu'il est extrêmement difficile de reproduire les expérimentations effectuées sur tel ou tel matériel. En effet, le matériel et la couche logicielle étant fortement couplés pour obtenir des performances maximales, les conditions d'expérimentations sont extrêmement coûteuses à mettre au point et compliquées à reproduire.

Plus récemment, une nouvelle tendance a émergé : le *Cloud Computing* ou informatique en nuage. L'objectif est assez analogue au calcul haute performance, à savoir, orchestrer la répartition de processus sur une grande quantité de ressources. En revanche, la nature

des traitements (et donc des processus qui en découlent) est très différente. Pour le cas du calcul scientifique, les processus issus du traitement sont fortement couplés entre eux et avec le matériel. On pourrait faire ici un parallèle avec les organes d'un corps humain. Si un organe défaille, le corps risque de mourir. Pour les traitements Cloud, c'est différent.

En effet, les traitements effectués dans le Cloud ne se résument pas à des calculs sur des données pour en tirer un résultat. Un Cloud sert également à instancier des services hétérogènes, persistants et éventuellement tolérants aux pannes. Ainsi un traitement peut s'incarner par un ensemble de processus, éventuellement tout ou partie redondée, communiquant entre eux de façon occasionnelle ou soutenue couplés à des niveaux divers avec le matériel. Le découplage du Cloud entre les processus et les ressources matérielles favorisent l'application de certaines propriétés sur les traitements au prix d'un certain coût de ressources.

Notre contribution est de transposer aux infrastructures HPC certaines caractéristiques des environnements Cloud. Nous ciblons trois propriétés en particulier :

— L'élasticité ;
— La reproductibilité ;
— La cohabitation.

L'élasticité est la capacité des services Cloud à passer à l'échelle via un accroissement horizontal (plus de processus) ou vertical (plus de ressources par processus) en fonction de la charge exercée sur eux. La reproductibilité est l'art de dupliquer des expérimentations de façon idempotente (peu importe le nombre de fois où elles sont rejouées, le résultat sera toujours identique en laissant les données dans le même état). Cette idempotence est garantie par un ensemble de mécanismes de confinement de processus reposant sur deux méthodes. La première méthode consiste à présenter un matériel émulé aux noyaux des processus confinés. Cette méthode répond au nom de virtualisation et consiste à exécuter plusieurs noyaux en parallèle sur du matériel émulé. La seconde méthode est la conteneurisation qui multi-instancie un ou plusieurs objets du noyau et les dédie aux processus confinés. Enfin, la cohabitation est l'habileté des fournisseurs de Cloud à instancier efficacement sur leur infrastructure une grande diversité d'applications allant du *workflow* scientifique au serveur web.

L'approche proposée dans cette thèse consiste à instancier les ordonnanceurs HPC directement sur les infrastructures Cloud pour bénéficier des trois propriétés susnommées. L'orchestrateur Cloud agit comme un système de provisionnement pour notre ordonnanceur HPC en nuage. Il convient dès maintenant d'insister sur un point. Notre approche ne vise pas à remplacer les infrastructures HPC actuelles. En effet, notre approche reposant sur les couches d'isolation inhérentes au Cloud, les ordonnanceurs HPC se voient amputés d'une partie de leurs capacités de prise en compte du matériel à un niveau très fin. En revanche, cette approche convient aux nouveaux usages qui ont émergé avec la démocratisation du HPC.

Notre approche vient donc compléter le paysage HPC existant en apportant la notion de HPC « mou » par opposition aux infrastructures HPC historiques que nous qualifierons de « dures ». Nous pouvons donc compléter un cluster HPC classique avec une partie des ressources dans le Cloud dans un contexte d'hybridation. Mais nous pouvons également

instancier un cluster HPC de A à Z dans le Cloud.

En définitive, notre travail a donné lieu aux publications qui suivent.

Publications dans des conférences internationales à comités de lecture :

- Cérin, C., Greneche, N. and Menouer, T. (2020, September). Towards pervasive containerization of HPC job schedulers. In 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) (pp. 281-288). IEEE, doi : 10.1109/SBAC-PAD49847.2020.00046. https://ieeexplore.ieee.org/document/9235080 (2020)
- Menouer, T., Greneche, N., Cérin, C., and Darmon, P. (2021, November). Towards an optimized containerization of HPC job schedulers based on namespaces. In IFIP International Conference on Network and Parallel Computing (pp. 144-156). Cham : Springer International Publishing, doi :10.1007/978-3-030-93571-9_12. https://link.springer.com/chapter/10.1007/978-3-030-93571-9_12 (2021)
- Greneche, N., Menouer, T., Cérin, C. and Richard, O. (2022, August). A methodology to scale containerized HPC infrastructures in the Cloud. In European Conference on Parallel Processing (pp. 203-217). Cham : Springer International Publishing, doi.org/10.1007/978-3-031-12597-3_13 https://link.springer.com/chapter/10.1007/978-3-031-12597-3_13 (2022)

Publication dans des ateliers :

- Greneche, N. and Cérin, C. (2022, November). Autoscaling of Containerized HPC Clusters in the Cloud. In 2022 IEEE/ACM International Workshop on Interoperability of Supercomputing and Cloud Technologies (SuperCompCloud) (pp. 1-7). IEEE, doi : 10.1109/SuperCompCloud56703.2022.00006 https://ieeexplore.ieee.org/document/10027497 (2022) Artefact : https://github.com/Nyk0/chsc

Chapitre de livre :

- Cérin, C., Grenèche, N. and Menouer, T. (2023). Executing Traditional HPC Application Code in Cloud with Containerized Job Schedulers. In High Performance Computing in Clouds : Moving HPC Applications to a Scalable and Cost-Effective Environment (pp. 75-97). Cham : Springer International Publishing (2023). https://link.springer.com/chapter/10.1007/978-3-031-29769-4_5

Récompense :

- Prix de la meilleure expérimentation : Greneche, Nicolas ; Cerin, Christophe ; Menouer, Tarek ; Richard, Olivier (2022). Artifact and instructions to generate experimental results for Euro-Par 2022 conference proceedings : A methodology to scale containerized HPC infrastructures in the Cloud. figshare. Online resource. doi :10.6084/m9.figshare.19952813.v1 (2022). `https://springernature.figshare.com/ndownloader/files/35504759`

Nous donnons dans les annexes les sources de nos artefacts d'expérimentations. Nous croyons très fortement à l'accès libre à ces éléments car ils permettent une science reproducible. Malheureusement, la vérification indépendante des résultats de recherche publiés n'a pas toujours été systématique ni même habituelle en informatique, bien que l'innovation dans ce domaine repose largement sur l'expérimentation avec des logiciels, du matériel et des données. La situation évolue, et cela est notable, notamment sur le plan des questions de recherches reproductibles [2].

En outre, un nouvel élan pour remédier à cette situation a été donné par différents gouvernements qui exigent que des plans de gestion des données accompagnent les demandes de subvention pour le partage des résultats expérimentaux. De même, il est de plus en plus accepté que l'accès aux artefacts expérimentaux qui sous-tendent les résultats rapportés dans les publications est le meilleur moyen de garantir l'intégrité scientifique et de faire progresser le domaine.

Il convient de noter également que les sociétés savantes internationales en informatique ACM et IEEE travaillent depuis dix ans sur ces problématiques [3]. Les différents travaux initiaux ont convenus qu'il est sans doute trop tôt pour être prescriptif sur les meilleures pratiques, mais que des recommandations spécifiques sont certainement nécessaires pour :

1. Clarifier les définitions de base, les critères d'évaluation pour la réplicabilité, la répétabilité, la reproductibilité, la réutilisation, la disponibilité ;
2. Motiver et inciter les auteurs, les évaluateurs, les comités de programme, les comités éditoriaux ;
3. Adopter/inventer des descriptions de métadonnées standard pour les logiciels, les données, les méthodologies ;
4. Permettre les processus d'évaluation des artefacts dans les flux de travail de soumission automatisés ;
5. Encourager le partage des artefacts ;
6. Définir des formats de stockage et d'empaquetage acceptables ;
7. Soutenir et intégrer : les dépôts de données et de logiciels internes et externes ;
8. Identifier, citer et lier les artefacts en tant qu'objets de publication de première classe ;

2. `https://reproducibility.gricad-pages.univ-grenoble-alpes.fr/web/pages/Ressources\%20diverses.html`

3. `https://www.ieee.org/content/dam/ieee-org/ieee/web/org/ieee_reproducibility_workshop_report_final.pdf`

9. Conserver et préserver les artefacts en vue d'une réutilisation future.

1.1 Présentation du sujet

De façon très générale, notre sujet consiste à explorer les possibilités offertes par les environnements Cloud pour les appliquer aux infrastructures HPC. Ce rapprochement n'est pas tout à fait un nouvel axe de recherche mais il a été peu exploré à ce jour, de notre point de vue. Dans le chapitre 8 consacré à l'état de l'art, nous verrons que les deux mondes ont déjà plusieurs intersections. L'originalité de notre travail est que nous allons explorer les possibilités offertes par le provisionnement dans les environnements Cloud pour voir dans quelle mesure nous pouvons en bénéficier dans les infrastructures HPC. Cette problématique est relative à l'administration système des infrastructures HPC.

Notre sujet s'inscrit totalement dans ce domaine. Nous ne sommes pas du tout sur les aspects performances, Intelligence Artificielle, BigData, la conteneurisation de logiciels scientifiques ou encore le développement de services HPC en mode SaaS (*Software as a Service*). Nous nous positionnons à un niveau plus bas pour nous intéresser aux bénéfices que peuvent retirer les ordonnanceurs HPC (et par conséquent les infrastructures physiques qui les accueillent) des technologies issues du monde du Cloud. Dans notre travail, l'élément clé est le provisionnement des ressources, à savoir les nœuds de calcul dans le cas présent. Il s'agit donc d'une thèse profondément ancrée dans le système.

Pour donner un définition générale du provisionnement, celui-ci consiste à créer et mettre en place une infrastructure informatique physique ou virtuelle de façon automatique ou manuelle. Ce terme regroupe les différentes étapes nécessaires pour gérer la mise en place d'un système fonctionnel. Nous noterons que lorsque le provisionnement est mentionné, il sous-entend généralement un déploiement automatique par lot. L'idée de notre travail est de rendre les ordonnanceurs HPC élastiques, ce qui implique la mise à disposition, à la demande, de nœuds de calcul dans un temps raisonnable pour permettre à notre infrastructure de croître ou réduire pour s'adapter à la demande. Pour la phase de croissance, il est donc nécessaire de provisionner, éventuellement en temps réel, des nœuds de calcul. Nous allons maintenant faire une revue rapide des différentes méthodes de provisionnement.

Historiquement, le provisionnement consistait à déployer des systèmes d'exploitation sur un serveur physique (aussi appelé serveur *bare-metal*). Ce déploiement automatisé visait à éviter les erreurs de configuration en proposant d'instancier des systèmes d'exploitation à la configuration homogène sur un lot de machines. Nous noterons que le premier outil de provisionnement libre nommé xCAT [4] a été publié le 31 octobre 1999 par IBM, ce qui consécutif à l'adoption des clusters HPC tels que nous les connaissons actuellement. Nous reviendrons sur l'historique du HPC dans la section 3.1 qui lui est dédiée.

Cette méthode de provisionnement présente deux limitations. La première est que la mise à disposition de nouveaux nœuds de calcul est lente car le système entier doit être

4. https ://xcat.org

descendu sur la machine. De plus, il implique des redémarrages de la machine ce qui impacte d'autant plus le temps de mise à disposition des nouveaux nœuds de calcul. Le second problème est que les machines physiques sans systèmes d'exploitation ne disposent pas de connectivité IP. Pour recevoir un nouveau système, elles ne peuvent compter que sur le niveau liaison pour accrocher un serveur de déploiement via, par exemple, le protocole PXE. Dans un contexte traditionnel, c'est-à-dire si les routeurs n'offrent pas de flexibilité comme relayer le trafic de niveau 2 ou l'encapsuler dans du niveau 3, cette limitation prive le déploiement d'une quelconque capacité de routage pour joindre un autre réseau dans un contexte d'hybridation. Nous serons dans l'incapacité d'aller piocher dans des ressources distantes type fournisseur Cloud. Nous excluons donc cette direction de recherche.

Toujours dans la fin des années 90 / début des années 2000, la virtualisation a commencé à se démocratiser. Nous ne parlerons pas ici des mécanismes car la section 4.1 les évoque. Mais, en résumé la virtualisation consiste à instancier plusieurs systèmes d'exploitation invités isolés les uns des autres au-dessus d'un système d'exploitation hôte en gérant les aspects d'accès exclusif au matériel afin d'optimiser la répartition des ressources. Dans ce cas, le problème de provisionnement se déplace vers la mise à disposition des images des systèmes invités. Cependant, ces images sont lourdes car elles incluent un système complet et leur lancement revient à en jouer toute la séquence de démarrage ce qui implique une latence d'instanciation assez élevée. De plus, à l'exception des fondeurs de processeurs qui ont pris en compte la virtualisation au niveau matériel en ajoutant des instructions dédiées comme Intel VT-x où AMD-V, le niveau d'indirection nécessaire pour le partage des ressources matérielles entre les systèmes invités induit une dégradation des performances peu souhaitable dans le contexte HPC. Nous excluons également cette direction.

Un peu après la virtualisation, une seconde méthode d'isolation s'est également démocratisée. Cette méthode nommée conteneurisation vise à isoler un processus plutôt qu'un système complet. A l'instar de la virtualisation, on retrouve des processus invités isolés s'exécutant sur un système d'exploitation hôte. Cependant, dans ce cas, un seul noyau (donc un seul système d'exploitation) est en exécution. De plus, les images sont beaucoup plus légères car elles incluent l'exécutable associé au processus ainsi que ses dépendances. Enfin, le temps d'instanciation du point de vue de la perception humaine d'un conteneur est égale à celui d'un processus non isolé. Dans un contexte à la demande, cette méthode de provisionnement est tout à fait acceptable.

En conséquence, notre approche développée dans cette thèse est complètement centrée autour de la notion de processus. Cependant, avant toute chose, il convient de se poser une question essentielle : est ce que ce travail a un intérêt dans le monde réel ? Dans la section 1.2 qui suit, nous avons mené une étude sur plusieurs centres de calcul qui officient à des échelles variées pour répondre à cette question.

1.2 Motivation du sujet par l'étude du profil des travaux HPC

La première étape de notre travail fut d'évaluer l'opportunité d'une telle approche. En effet, si les infrastructures HPC actuelles donnent une pleine satisfaction aux usagers en terme d'utilisation ainsi qu'aux administrateurs sur l'optimalité de l'utilisation des ressources, alors il n'est pas nécessaire de développer des mécanismes complémentaires. Nous avons donc commencé par faire une étude de l'utilisation des ressources CPU. Pour cette étude nous avons lancé un appel sur la liste calcul du CNRS [5] pour collecter les logs d'ordonnanceurs HPC. Nous avons reçu une dizaine de réponses de centres de calcul de différentes échelles :

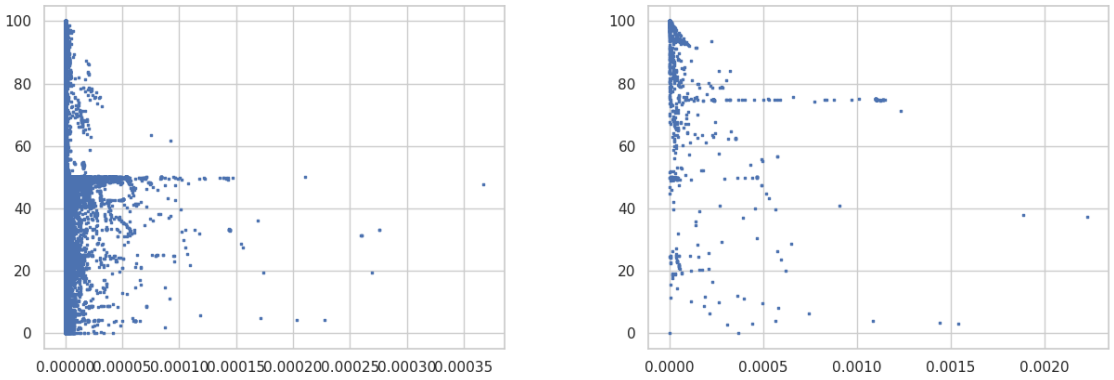
— La machine de laboratoire ;
— La machine d'université / mésocentre ;
— Le centre national ;
— La machine spécialisée.

La machine de laboratoire constitue le premier barreau sur l'échelle de la mutualisation. Ces machines sont partagées par les équipes d'un laboratoire de recherche. L'usage est orienté vers un domaine scientifique, cependant la diversité des équipes partageant la machine introduit une certaine hétérogénéité au niveau des pratiques (contrairement à la machine que nous avons qualifié de spécialisée qui est taillée pour un usage donné). La machine d'université ou mésocentre est une machine partagée par différentes entités de recherches officiant dans des thématiques extrêmement variées. Il en résulte une hétérogénéité des usages bien plus vaste que pour les machines de laboratoire. Le centre national recoupe les mêmes usages mais à une autre échelle en termes de quantité de ressources disponible. Enfin, la machine spécialisée est une machine dédiée à une thématique scientifique avec des usages homogènes. Par exemple le cluster de l'IPGP (Institut de Physique du Globe de Paris) qui est dédié aux sciences de la terre et donc optimisé pour un nombre restreint de logiciels issus de la discipline.

Les administrateurs des différentes machines nous ont communiqué les enregistrements des travaux lancés sur leur infrastructure sur une période de six mois. Nous les avons traités pour déterminer l'efficacité de leurs travaux, c'est-à-dire la quantité de CPU réservée par rapport à l'utilisation réelle mesurée sur les machines. Nos résultats ont été présentés au workshop WAMCA (*Workshop on Applications for Multi-Core Architectures*) en 2020 [4]. Ce travail a mis en lumière le fait que pour les machines de la population hétérogène, l'efficacité moyenne d'un travail par rapport au processeur est de 38% avec une médiane à 48% pour les centres nationaux. Nous sommes dans les mêmes ordres de grandeur pour les mésocentres. D'autres études, telles que [5] qui est moins focalisées sur la partie CPU mais plus mémoire, retombent sur le même chiffre. Les auteurs ne considèrent que la machine de niveau national mais s'intéressent aussi à la mémoire.

Pour aller un peu plus loin dans la taxonomie, nous avons cherché à caractériser plus finement le profil des travaux. L'efficacité moyenne est déjà un indicateur mais pour deux

5. https ://calcul.math.cnrs.fr/



(a) Centres nationaux

(b) Machines spécialisées

Figure 1.1 – Impact négatif des travaux (en abscisses le coefficient d'impact négatif et en ordonnées l'efficacité du travail)

travaux ayant une efficacité équivalente, le plus long va être le plus impactant sur le cluster. Nous avons donc calculé un facteur d'impact négatif basé sur le temps CPU du travail. Le temps CPU est une métrique qui prend en compte le nombre de CPU affectés au travail. Ainsi, si un travail dure 200 secondes et que l'utilisateur a réservé quatre CPUs alors le temps CPU sera de 800 secondes. Cet impact négatif équivaut au temps CPU total perdu du travail (soit le complément du temps CPU passé à calculer) divisé par le temps CPU total consommé par tous les autres travaux durant l'exécution du travail ciblé.

Dans la figure 1.1, nous comparons l'impact négatif des machines de centres nationaux avec celui d'une machine dédiée à une population bien ciblée. Sur l'axe des ordonnées on retrouve l'efficacité du travail et sur l'axe des abscisses le coefficient d'impact négatif. On constate que, sur le graphique représentant les machines type centre nationaux, les points (chaque point représente un travail HPC) sont répartis tout le long des ordonnées et certains vont assez loin sur les abscisses. Cette répartition met en lumière le fait qu'il existe des emplacements de temps disponibles au niveau CPU pour d'autres processus (pas nécessairement HPC d'ailleurs).

Au-delà de l'aspect ressources, les usages autour du HPC ont évolué à travers le temps avec l'hétérogénéisation de la population utilisatrice de moyen de calcul. Cette diversification a entraîné une sous-utilisation de certaines ressources lors de l'exécution d'un calcul. Il paraît donc opportun de rechercher des mécanismes de cohabitation afin d'arriver à mobiliser ces ressources libres pour d'autres travaux. Avec la conteneurisation, il est dorénavant possible de partager une machine physique (ou virtuelle) entre plusieurs processus isolés. Un axe de recherche dans lequel nous nous inscrivons consiste à compléter les machines HPC traditionnelles avec des ressources en Cloud pour décharger les clusters HPC *bare-metal* des travaux qui ne nécessitent pas un tel niveau d'optimisation.

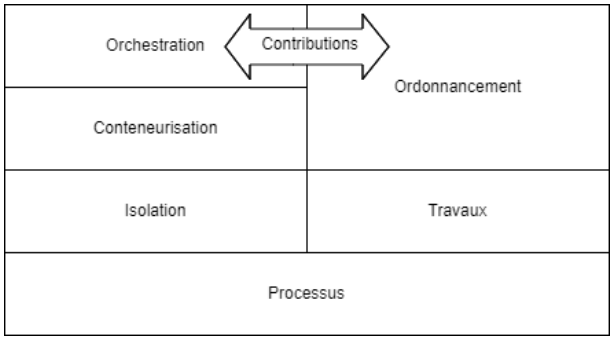


Figure 1.2 – Organisation et dépendances des notions

1.3 Annonce du plan

Comme nous l'avons précisé dans la section 1.1, notre approche est centrée sur les processus et leur isolation pour faire office de nœuds de calcul conteneurisés. Le manuscrit s'organise en deux grandes parties. La première présente tous les éléments sur lesquels nous nous appuyons pour notre travail. Il s'agit de composants relativement matures, utilisés sur des systèmes en production. La figure 1.2 présente l'articulation des notions à aborder.

La première partie démarre avec le chapitre 2 dans lequel nous mettons en évidence quelques points saillants de la gestion des processus par les systèmes Linux. Nous nous limitons aux aspects essentiels à la compréhension de notre sujet. Dans le chapitre 4, nous dressons un état de l'art des méthodes d'isolation actuelles sous Linux. Notre travail s'inscrit dans une thématique HPC et plus précisément sur les ordonnanceurs de travaux. Ainsi dans le chapitre 3, nous parlons de l'histoire du HPC avec un focus sur les ordonnanceurs de travaux.

Une fois créés, nos nœuds de calcul conteneurisés doivent pouvoir être distribués sur les machines pour s'exécuter. Le chapitre 5 comporte les éléments nécessaires à la compréhension de cette problématique. Cette distribution se passe en deux phases. Tout d'abord, des conteneurs simulant des nœuds de calcul sont créés en utilisant les mécanismes de bas niveau présentés dans le chapitre 4. Ils sont ensuite ils sont distribués sur l'infrastructure par un orchestrateur. Le chapitre 6 reprend tous les éléments essentiels présentés dans cette première partie. Ce chapitre viendra clôturer la première partie destinée à mettre en place tous les éléments techniques qui sous-tendent notre contribution.

Le chapitre 7 ouvre la seconde partie en présentant les défis de la communauté travaillant autour de la convergence Cloud / HPC. Cet chapitre est assez général au niveau des problématiques abordées. Le chapitre 8 le complète en resserrant l'état de l'art autour de notre positionnement par rapport à la communauté HPC. Une fois la position clarifiée, nous détaillons nos contributions dans les chapitres 9, 10 et 11 qui se suivent chronologiquement dans la réalisation de notre travail. En effet, le chapitre 9 présente une expérimentation autour de la conteneurisation de l'ordonnanceur Slurm et son intégration dans une infrastructure HPC réelle, à savoir le centre de calcul de l'USPN (Université Sorbonne Paris Nord).

Le chapitre 10 présente une méthode pour ajouter ou retirer dynamiquement des nœuds de calcul conteneurisés dans un cluster HPC. Nous avons éprouvé cette méthode avec trois ordonnanceurs majeurs : Oar, OpenPBS et Slurm. Le chapitre 11 se situe dans la continuité du précédent en ajoutant la notion d'automatisme à l'ajout ou au retrait de nœuds de calcul conteneurisés. Nous avons réalisé l'expérimentation avec Oar.

Le chapitre 12 est le dernier. Il fait office de conclusion et synthétise notre contribution pour présenter les perspectives de prolongement de notre travail. Nous distinguons des perspectives à moyen et long terme visant principalement à intégrer notre travail de façon efficace dans le monde du HPC.

Enfin, Nous retrouvons les annexes qui se décomposent en trois parties. La première reprend les éléments systèmes centrés autour du processus qui contribuent à son isolation par la conteneurisation. La seconde partie présente l'artefact primé à Europar'22. La dernière partie présente l'artefact fourni pour l'atelier SuperCompCloud'22.

Première partie

Notions clés et éléments de contexte du HPC et du Cloud

CHAPITRE 2

LES PROCESSUS

Les processus sont les entités actives d'un système d'exploitation. Ils sont le résultat du lancement d'un programme exécutable (parallèle ou non). Le programme en lui-même est une entité latente, présente sur le système de fichiers mais qui doit être exécutée pour passer à l'état actif de processus. Le processus va s'appuyer sur le noyau pour accéder aux ressources matérielles. Cependant, la gestion des processus n'est pas uniforme selon le système d'exploitation concerné.

Dans ce chapitre nous allons commencer par présenter les familles de systèmes d'exploitation via une grille d'évaluation comprenant notamment leur gestion des processus ainsi que l'articulation par rapport aux autres mécanismes nécessaires au fonctionnement du système d'exploitation. Ensuite nous resserrons la discussion sur le système d'exploitation cible (à savoir Linux) en précisant le fonctionnement général des processus et quelques mesures de sécurité concourant à l'isolation pré-conteneurisation.

Nous documentons également en annexe A les aspects fins de la notion de processus pour le lecteur qui souhaiterait approfondir les notions évoquées. En définitive, la figure 2.1 rappelle l'importance et l'articulation de ces notions qui sont le socle sur lequel repose notre sujet, d'où cette incise en annexe.

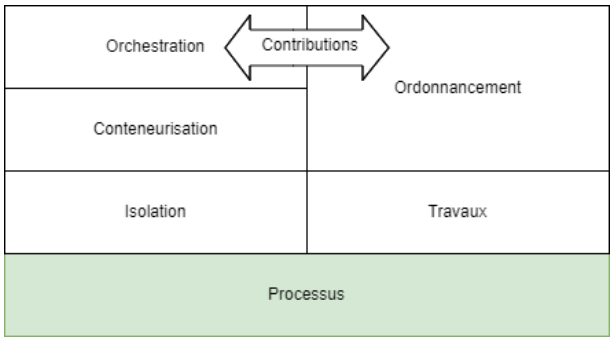


FIGURE 2.1 – Organisation et dépendances des notions

2.1 Familles de systèmes d'exploitation

Dans un système d'exploitation, l'espace utilisateur et l'espace noyau sont deux espaces mémoire disjoints. Ces domaines sont séparés pour des raisons de sécurité et de stabilité. Nous allons présenter ces espaces via trois propriétés liées aux processus qui s'y exécutent à savoir le niveau d'accès au matériel, l'isolation mémoire offerte et les privilèges accessibles.

L'espace utilisateur est la zone de mémoire où s'exécutent les applications et les processus créés par les utilisateurs. Ces applications sont les programmes typiques des utilisateurs comme les éditeurs de texte, les navigateurs Web, etc. Les programmes de l'espace utilisateur ne peuvent pas accéder directement aux ressources matérielles. Ils doivent passer par des interfaces fournies par le noyau pour accéder aux services système. Les caractéristiques de l'espace utilisateur sont :

— Accès restreint : Les programmes de l'espace utilisateur n'ont pas un accès direct aux ressources matérielles et ne peuvent pas exécuter d'instructions sensibles. Cela garantit que les applications ne peuvent pas causer de dommages irréparables au système ;
— Isolation : Chaque application s'exécute dans son propre espace mémoire isolé, ce qui évite les conflits de mémoire et les interférences entre les processus ;
— Privilèges limités : Les processus de l'espace utilisateur ont des privilèges limités en termes de contrôle sur le système. Ils ne peuvent effectuer que des opérations approuvées et autorisées par le noyau (grâce à une interface d'utilisation normée).

L'espace noyau, quant à lui, est la partie du système d'exploitation qui gère les ressources matérielles (via les pilotes) et fournit les services aux applications de l'espace utilisateur comme la gestion des processus, de la mémoire, des opérations d'entrée / sortie (affichage, réseau ou fichier), de l'ordonnancement, des systèmes de fichiers, etc. Ainsi, un processus travaille dans l'espace utilisateur mais est défini dans l'espace noyau. Les caractéristiques de l'espace noyau sont :

— Accès privilégié : Le noyau a un accès complet aux ressources matérielles et peut exécuter des instructions sensibles. Cela lui permet d'accomplir des tâches critiques et de gérer efficacement grâce à un accès direct (c'est-à-dire sans contrôle d'accès) au matériel ;
— Isolation : Le noyau est isolé de l'espace utilisateur pour éviter que les erreurs où les accès indésirables des processus en espace utilisateur n'affectent directement le noyau en compromettent la sécurité et la stabilité du système ;
— Privilèges élevés : Le noyau a des privilèges élevés pour prendre des décisions et exécuter des opérations qui affectent l'ensemble du système.

Ces deux espaces sont communs à tous les systèmes d'exploitation actuels. La différence entre les différentes familles réside dans l'étendue des composants s'exécutant dans l'un ou l'autre des espaces. Plus il y a de composants dans l'espace noyau, plus le système est rapide mais potentiellement instable. On distingue ainsi trois familles de systèmes d'exploitations ordonnées de façon croissante par le nombre de composants en espace noyau :

1. Les systèmes d'exploitation à noyau monolithique ;
2. Les systèmes d'exploitation à micro-noyau ;
3. Les systèmes d'exploitation à noyau hybride (que nous n'évoquerons que très rapidement dans la synthèse de la section 2.1.3).

2.1.1 Les systèmes d'exploitation à noyau monolithique

C'est la famille de systèmes d'exploitation la plus représentée. Dans un noyau monolithique, tous les composants partagent le même espace mémoire. Ils communiquent directement entre eux via des accès directs à la mémoire sans passer par des systèmes type IPC (*Inter Process Communications*). Ainsi, ces systèmes offrent une meilleure performance au détriment du contrôle et de l'isolation des communications.

— Structure : Dans un noyau monolithique, toutes les fonctionnalités du système d'exploitation (gestion des processus, de la mémoire, les pilotes de périphériques, etc.) sont implémentées comme des composants intégrés directement dans le noyau. L'espace noyau et l'espace utilisateur coexistent dans un seul espace d'adressage ;
— Processus : Les processus sont gérés par le noyau directement. Les transitions entre les différents états des processus (en cours d'exécution, en attente, terminé, etc.) sont gérées par les mécanismes internes du noyau ;
— Communication entre les composants : Les composants du noyau monolithique communiquent généralement via des appels systèmes avec des accès directs à la mémoire. Cela permet une communication rapide au détriment de l'isolation des interactions ;
— Complexité : Les noyaux monolithiques sont généralement plus simples à concevoir et à développer initialement car toutes les fonctionnalités sont centralisées. Cependant, ils peuvent devenir plus complexes à mesure que de nouvelles fonctionnalités sont ajoutées ;
— Performance : En raison de l'exécution directe des fonctions du noyau, les noyaux monolithiques ont tendance à offrir de meilleures performances en termes de vitesse d'exécution. Cependant, une erreur dans une partie du noyau peut potentiellement affecter tout le système.

Le système star à noyau monolithique est Linux. En réalité, l'appellation Linux correspond à un noyau libre. Il est en général enrobé d'outils (Glibc, interpréteurs de commandes, compilateurs etc.) souvent sous licence GNU [1]. C'est pour cette raison que pour désigner un système basé sur Linux, on parle de système GNU / Linux. Il existe différents assemblages noyau Linux / outils GNU. Ces assemblages sont appelés distributions. Les principales distributions sont Debian, RedHat, Fedora, SuSE, Ubuntu, Gentoo et Arch.

La parenté de Linux avec Unix se joue au niveau de l'implémentation de la norme POSIX (*Portable Operating System Interface*). POSIX est une norme qui définit un ensemble de fonctions permettant de solliciter les services de base d'un système d'exploitation. L'objectif est de permettre aux applications de fonctionner sur différents systèmes

1. https ://www.gnu.org/licenses/licenses.html

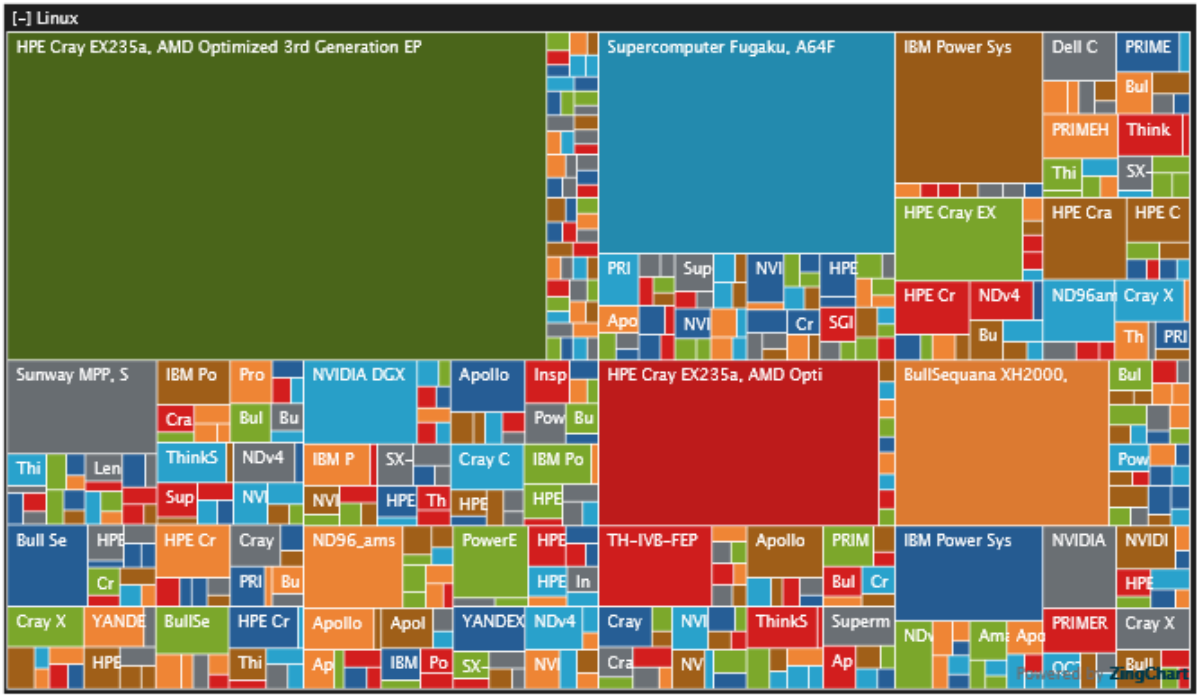


Figure 2.2 – Adoption de Linux parmi les machines du TOP500

d'exploitation sans avoir à être modifiées. POSIX est le standard officiel qui définit les interfaces communes à tous les systèmes de type Unix

On notera que l'interface POSIX ne définit pas un système d'exploitation en tant que tel. Elle fournit simplement une liste de fonctions et de comportements qui doivent être pris en charge par les systèmes d'exploitation compatibles avec POSIX. Les bibliothèques POSIX permettent aux développeurs d'utiliser les fonctions répondant à la norme dans leurs applications. Ces bibliothèques masquent les spécificités des différents systèmes d'exploitation, ce qui facilite le développement d'applications portables d'un système POSIX à un autre.

Linux représente donc 96,3% du million de sites web à plus gros trafic [2], 90% des systèmes qui sous-tendent les infrastructures publiques de Cloud [3] et 100% des 500 plus grosses machines de calcul comme le montre la figure 2.2 issue du générateur de statistiques [4] du TOP500 [5] pour juin 2023. Cette figure représente la distribution des systèmes d'exploitation pour les machines du TOP500. Chaque système d'exploitation de l'image est représenté par un rectangle contenant lui-même les machines basées sur lui. On observe qu'il n'y a qu'un seul système représenté, Linux, avec toutes les machines dedans. Linux est donc un standard de facto dans le monde du Cloud et du HPC.

2. https ://truelist.co/blog/linux-statistics
3. https ://www.linuxfoundation.org/blog/blog/linux-runs-all-of-the-worlds-fastest-supercomputers
4. https ://www.top500.org/statistics/treemaps
5. https ://www.top500.org

2.1.2 Les systèmes d'exploitation à micro-noyau

Les systèmes basés sur les micro-noyaux tentent d'implémenter le principe de la séparation de privilèges et du moindre privilège [6] aux noyaux de systèmes d'exploitation. Par opposition aux noyaux monolithiques où tous les composants tournent dans l'espace noyau, les développeurs de micro-noyaux tentent de découper les fonctionnalités en bloc autonomes qui communiquent entre eux via des mécanismes type RPC, comme suit :

— Structure : Un micro-noyau sépare les fonctionnalités du système en composants essentiels (gestion des processus, de la gestion de la mémoire, etc.) qui sont exécutés dans l'espace noyau restreint. Les fonctionnalités non essentiels comme les pilotes de périphériques sont quant à elles exécutées dans l'espace utilisateur, distinct de l'espace noyau. c'est un recours massif au principe du moindre privilèges ;
— Processus : Dans un micro-noyau, les processus s'exécutent dans l'espace utilisateur. Les transitions entre les états des processus impliquent des appels au noyau via des mécanismes comme les RPC ;
— Communication entre les composants : Les composants d'un micro-noyau communiquent généralement via des RPC, ce qui est plus lent que les appels systèmes, mais offre une meilleure isolation. Les services utilisateur et noyau communiquent à travers des mécanismes définis par le noyau ;
— Complexité : Les micro-noyaux implémentent nativement la séparation de privilèges en développant chaque composant de façon modulaire. Cette méthode est plus complexe à mettre en place en raison de la nécessité de gérer les communications entre les composants et l'interaction entre l'espace noyau et l'espace utilisateur. Cependant, cette modularité facilite la maintenance des composants à mesure que le système évolue ;
— Performance : En raison des communications inter-composants via des RPC et de la séparation de l'espace noyau et de l'espace utilisateur, les micro-noyaux peuvent avoir une légère perte de performance par rapport aux noyaux monolithiques. Cependant, les avantages en termes de fiabilité et de sécurité peuvent compenser cela.

Les micro-noyaux offrent donc certains avantages en termes de fiabilité et de sécurité. Certaines implémentations comme seL4 [7] de la famille des L4 [8] sont prouvés formellement. Ces avantages ont un coût en terme de performances et de développement. Cependant, pour des systèmes opérant dans un domaine spécialisé, cette catégorie de systèmes d'exploitation est intéressante. On pourra citer en exemple QNX [9] qui est très présent dans les systèmes embarqués [10], notamment dans l'industrie des véhicules autonomes [11].

2.1.3 Synthèse sur les familles de systèmes d'exploitation

Dans cette présentation nous n'avons pas détaillé les noyaux hybrides. En effet, ceux-ci sont utilisés principalement chez Apple avec XNU et Microsoft avec Windows et consistent principalement à faire varier le curseur entre les micro-noyaux qui n'ont que la gestion des IPC basiques, de la mémoire et de l'ordonnancement des processus dans l'espace noyau et les noyaux monolithiques où tout se fait dans l'espace le plus privilégié. Au niveau des

concepts, ils n'apportent pas grand chose au sujet ni en terme de représentation dans nos domaines à savoir le Cloud et le HPC. Ils sont peu présents.

Dans notre travail, nous utilisons Linux qui, comme nous l'avons vu précédemment, est prépondérant dans notre domaine. Nous avons vu que Linux est monolithique, c'est-à-dire que tous les composants du système fonctionnent dans l'espace noyau. Ainsi, de par sa conception, l'isolation des processus sous Linux est un sujet sensible. Dans la suite de ce chapitre nous synthétiserons les notions essentielles de la gestion des processus sous Linux puis dans le chapitre suivant, nous ferons une revue des méthodes d'isolation qui leurs sont associées.

2.2 Les processus sous Linux

Dans cette section nous allons modérer l'argument amené précédemment sur les systèmes monolithiques du point de vue de la sécurité en présentant les mécaniques d'isolation natives des processus sous Linux. En effet, MS-Dos et Linux sont tous les deux monolithiques mais il existe, en terme de sécurité, un monde entre les deux systèmes.

Il faut avoir à l'esprit que, concernant l'isolation des processus sous Linux, on trouve deux grandes ères : la première antérieure à l'apparition des espaces de noms, la seconde postérieure à celle-ci.

Dans cette section, nous donnerons quelques généralités sur le fonctionnement des processus ainsi que leurs mécanismes d'isolation natifs. Dans la section suivante nous aborderons le contrôle d'accès associé aux processus. Les mécanismes d'isolation post espaces de noms seront abordés dans le chapitre 4 qui suit.

2.2.1 Fonctionnement des processus

Un processus est avant tout une structure de données nommée `task_struct` définie dans les sources du noyau Linux (fichier d'en-tête `sched.h`) qui est le descripteur de processus (*process descriptor*). Cette structure de données contient tous les attributs du processus comme son identifiant (PID, Process IDentifier), son identité d'exécution, les descripteurs ouverts, etc.

L'annexe A.1 donne tous les détails sur la structure de données associée aux processus. Les descripteurs de processus sont organisés dans le noyau via une liste doublement chaînée circulaire utilisée par l'ordonnanceur nommé CFS (*Completely Fair Scheduler*). CFS va placer les processus sur les différents cœurs du processeur en fonction d'une priorité calculée dynamiquement. L'annexe A.2 présente tous les détails sur l'ordonnancement des processus.

Durant son exécution, le processus dispose d'un espace mémoire virtuel qui lui est propre. En conséquence, le processus n'a pas d'accès direct à la mémoire. Il voit un espace d'adressage dont il est le seul utilisateur et qui n'a de sens que pour lui. La translation vers la mémoire réelle (mémoire vive) se fait par le couple MMU (*Memory Management*

Unit) et TLB (*Translation Lookaside Buffer*) géré dans le noyau. Dans l'annexe A.3 on trouve tous les détails d'allocation mémoire. C'est une mesure d'isolation pour éviter que les processus n'aient accès à la mémoire des uns des autres. Les échanges de données doivent être « consentis » via des communications IPC (*Inter Process Communications*) de différents types (mémoire partagée, *sockets* locaux, etc.) listés dans les annexes A.4. Cependant, avant les espaces de noms, il n'existait pas de mécanisme particulier pour isoler les IPC.

Au cours de son cycle de vie, le processus va passer par différents états. L'état courant est stocké dans l'attribut `state` du descripteur de processus. La plupart du temps, le processus navigue entre deux états : TASK_RUNNING et TASK_[UN]INTERRUPTIBLE respectivement en exécution et en attente d'être ordonnancé. Les signaux peuvent également influer sur l'état d'un processus. En effet, les processus implémentent tous un gestionnaire de signal (*signal handler*) qui déclenche certaines actions en fonction du signal reçu.

L'annexe A.5 donne plus de détails sur les états et les signaux. Les signaux sont adressés au processus par leur PID. Avant l'apparition des espaces de noms, il n'existait qu'un seul ensemble de PID. En conséquence, les processus n'étaient pas isolés les un des autres. Chaque processus pouvait potentiellement accéder au PID des autres.

Dans cette section, nous avons dessiné les contours des mécanismes d'isolation structurels proposés par le noyau Linux aux processus. Ces isolations concernent les processus entre eux dans l'espace utilisateur. Nous allons maintenant dire un mot sur les appels systèmes qui proposent une interface de communication entre les processus en espace utilisateur et le noyau afin, par exemple, d'accéder au matériel. Ici l'isolation repose sur la normalisation de cette interface.

2.2.2 Les appels systèmes

Dans la section précédente, nous avons dit que le processus consomme les ressources matérielles de la machine pour s'exécuter. Cependant, le matériel est piloté par le noyau, la structure de données du processus est dans le noyau, l'ordonnanceur est dans le noyau alors que les processus sont créés en espace utilisateurs. Il existe donc une interface de communication entre l'espace utilisateur et l'espace noyau. Cette interface de communication constitue les appels systèmes (*syscalls*). Les appels systèmes sont nécessaires à la stabilité et à la sécurité du système. Le fait de proposer une interface mandataire entre l'espace utilisateur et l'espace noyau contraint les actions que l'espace utilisateur peut demander au noyau.

Un appel système est une fonction fournie par le noyau d'un système d'exploitation et utilisée par les programmes s'exécutant dans l'espace utilisateur. Il permet à un programme d'effectuer une tâche qui ne peut être effectuée que par le noyau, comme l'accès à un périphérique matériel ou la gestion de la mémoire. Le fonctionnement d'un appel système est le suivant :

— Le programme appelant utilise une instruction d'appel système pour indiquer au noyau qu'il souhaite effectuer une tâche ;

— Le noyau interrompt le programme appelant et passe en mode noyau ;
— Le noyau effectue la tâche demandée par le programme appelant ;
— Le noyau renvoie le contrôle au programme appelant.

Lorsqu'un programmeur développe une application, il utilise des API (*Application Programming Interface*). Une API est une interface d'utilisation d'une librairie donnée. Par exemple l'API d'une librairie mathématique lambda propose des prototypes de fonctions classiques telles l'addition, soustraction, division ou multiplication. Ces API sont disponibles dans une multitude de langages de plus ou moins haut niveau. Cependant, elles sont toutes cantonnées à l'espace utilisateur. Un programmeur n'invoquera quasiment jamais jamais l'appel système en direct, il se contentera d'utiliser les fonctions de haut niveau qui, lorsqu'elles auront besoin d'interagir avec le matériel, déclencheront des appels systèmes.

La chaîne d'exécution d'un appel système diffère donc de l'exécution d'une fonction classique dans la mesure où elle effectue un changement de contexte de l'espace utilisateur à l'espace noyau. Ce changement de contexte est obligatoire car le noyau travaille dans un espacé mémoire protégé. Les variables concernées par le recours à un appel système doivent donc être copiées de l'espace utilisateur vers l'espace noyau. Après le traitement, une nouvelle copie est effectuée de l'espace noyau vers l'espace utilisateur. Les variables en espace noyau ne sont jamais manipulées en direct depuis l'espace utilisateur, il y a toujours une copie en préambule pour que la manipulation de la variable soit faite depuis l'espace de l'acteur. Dans l'annexe A.6, nous allons suivre l'exécution de la fonction `getentropy(void *buffer, size_t length)` de la GlibC. Cet exemple illustre les interactions entre l'espace utilisateur et l'espace noyau lors de l'invocation d'un appel système.

2.2.3 Création d'un processus

Le dernier pré-requis à évoquer concernant les processus est leur mécanique de création. C'est un point clé dans la mesure où la décision d'isoler un processus se prend au moment du lancement du processus. Le chapitre 4 est dédié aux mécanismes d'isolation. Dans cette section nous allons présenter les mécanismes de création des processus dits « lourds » et des *threads* (aussi appelés processus légers). Nous verrons également la phase d'exécution du code dans le processus nouvellement créé.

La première chose à savoir est que sous Linux, les processus sont organisés sur le modèle parent / enfant. L'idée générale est qu'à un moment de son cycle de vie, un processus va créer un fils qui est une copie de lui-même et qui va déclencher un recouvrement de ses pages d'exécution pour y projeter, par exemple, le code machine d'un fichier exécutable. C'est typiquement ce qui se passe au démarrage d'un système Linux. A la fin du lancement du noyau, le premier processus invoqué, `systemd` (le lanceur) va ensuite créer autant de fils que de services à démarrer. Un service est un processus qui s'exécute en mode boucle infinie en arrière plan invoqué par init dont le but est de répondre à des requêtes initiées par des processus soit 1) localisés sur des machines distantes (on parle alors de services réseaux) ou 2) depuis des processus locaux (on parle alors de services systèmes). La figure 2.3 présente cette hiérarchie.

On y retrouve aussi bien le processus `systemd` possédant le PID 1 avec comme fils

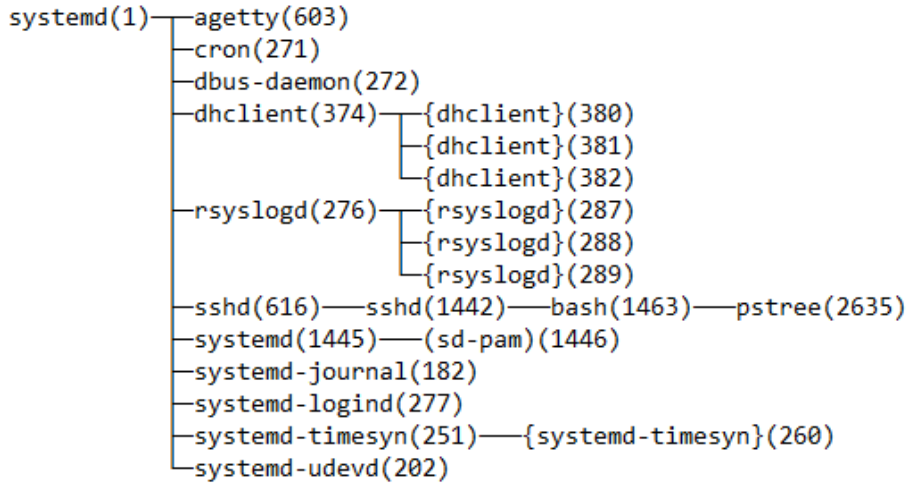


Figure 2.3 – Hiérarchie des processus depuis `systemd`

les services essentiels à savoir le planificateur de tâches `cron`, le service `dhclient` pour la configuration DHCP, le service de journalisation des événements `rsyslogd`, etc. On notera que c'est la même mécanique qui est mis en place lorsque l'on exécute une commande depuis un shell. Toujours dans la figure 2.3, nous pouvons voir le processus `pstree` qui est le résultat du lancement du fichier exécutable `/usr/bin/pstree`. Pour le lancer, nous sommes connectés en SSH (Secure SHell) sur la machine. En conséquence, si on se réfère à la figure 2.3, on voit que `systemd` a enfanté le service `sshd`. A la connexion de l'utilisateur, un nouveau processus `sshd` fils du précédent est créé et ce nouveau processus `sshd` enfante un shell `bash` pour accueillir l'utilisateur. Lorsque l'on tape la commande pstree dans le terminal, un processus `pstree` est créé et affiche la hiérarchie des processus.

La fonction chargée de créer un processus fils, copie du père, se nomme `fork()` et elle est disponible, par exemple, dans la GlibC. Ici on parle bien de la fonction mise à disposition par la GlibC, pas de l'appel système. Il n'est pas rare de confondre les deux. En effet, historiquement, l'appel système réalisant la copie de la structure `task_struct` (cf section 2.2.1) se nommait également `fork()`. Dorénavant, l'appel système chargé de cette action est `clone()`. Le fonctionnement de la fonction `fork()` est détaillée dans l'annexe A.7. Lorsqu'un processus est créé de cette manière, il dispose d'un espace mémoire totalement disjoint avec le père. C'est la différence entre les processus dits « lourds » et les *threads*.

Comparons les arguments de `clone()`, d'abord pour un processus lourd :

```
clone(child_stack=NULL, ...) = 1120
```

Puis pour un *thread* :

```
clone(child_stack=0x7f1cfc833fb0, ...) = 743
```

La différence est que l'argument `child_stack` est à NULL dans le cas du processus « lourd » alors qu'il contient une adresse pour le *thread*. Les *threads* partagent donc une zone mémoire avec le père. En effet, lorsque l'appel système *clone()* est invoqué, il prend un argument `child_stack` qui vaudra NULL si on crée un fils type processus « lourd ».

Si c'est un *thread*, il contient le nouveau *stack pointer* (registre esp) du processus fils. Dans le cadre d'un processus lourd, il n'est pas nécessaire de le préciser puisque le fils récupérant une copie du père, le *stack pointer* sera le même. Dans le cadre d'un processus léger, l'espace d'adressage virtuel du père et du fils est le même. Il faut donc deux piles différentes d'où la nécessité de préciser où se trouve la nouvelle pile

Il s'agit de l'unique différence entre un *thread* et un processus « lourd ». Ils sont tous les deux représentés par la structure `task_struct`. L'annexe A.8 propose des exemples illustrant la création des *threads* en C. Nous avons vu comment fait le système d'exploitation pour créer des processus. Cependant, le processus créé est la copie de son père. Or, lorsque l'on exécute une commande avec un interpréteur comme bash, c'est bien la commande qui est exécutée et pas une copie de bash. Nous allons maintenant aborder `exec()` qui est souvent consécutif à un `clone()` pour recouvrir le code exécutable du père projeté en mémoire au moment de la duplication par le code exécutable du programme visé.

Lors de l'exécution du programme fils, la fonction `execvp()` est utilisée pour recouvrir les pages exécutables du processus courant avec le code machine du fichier passé en paramètre. Le détail de la mécanique d'exécution du programme passé en paramètre dans le processus est définie dans l'annexe A.9. Cette fonction repose sur l'appel système `execve()` :

```
execve("/usr/bin/hostname", ["/usr/bin/hostname"], 0x7fff716cf3a8 ...) =
0
```

Cet appel système prend l'exécutable en paramètre et l'emplacement mémoire où le projeter.

2.3 Contrôle d'accès

Le contrôle d'accès est un élément essentiel pour limiter les interactions du processus avec les autres entités du système. Nous verrons deux modèles de sécurité basés sur l'identité du processus pour évaluer l'accès aux objets du système : le DAC (*Discretionary Access Control*) et le MAC (*Mandatory Access Control*). Nous verrons ensuite les capacités (*capabilities*) qui sont directement attachées au processus et lui donnent un certain nombre de privilèges sur le système. Enfin nous évoquerons un mécanisme d'isolation historique `chroot()` qui, bien que limité, est utilisé par plusieurs programmes pour assurer un niveau d'isolation.

La déclinaison du DAC pour les systèmes Linux se base sur l'identité du processus telle que présentée dans l'annexe A.10 ainsi que sur un jeu de droits positionnés sur le fichier accédé par le processus (sous réserve que le système de fichiers sous-jacent supporte le stockage de ces droits dans les métadonnées). Ce contrôle est très simple car il se limite à un triplet lecture / écriture et exécution pour trois ensembles d'utilisateurs à savoir utilisateur et groupe propriétaire du fichier et les autres utilisateurs pouvant accéder au système. En fonction des systèmes de fichiers, on peut éventuellement trouver des ACLs (*Access Control List*) étendues proposant une matrice de contrôle d'accès plus fine. La caractéristique

essentielle du DAC est que la politique de sécurité est décentralisée car elle est gérée par le propriétaire de la ressource (d'où le qualificatif de discrétionnaire). Elle évolue donc dans une configuration non prédictible par l'administrateur des ressources. Ainsi, pour compléter le DAC, il existe un autre modèle MAC (*Mandatory Access Control*) qui applique une politique de sécurité fixée une fois pour toute par le noyau à son démarrage et qui ne peut être modifiée durant l'exécution du système même par l'administrateur de la machine. La déclinaison la plus connue du MAC est SELinux.

Les capacités sont complémentaires au DAC dans la mesure où lorsqu'un processus tourne avec une identité banalisée, elles lui donnent la possibilité d'acquérir des privilèges précis sur le système. Par exemple, lorsque l'on effectue une capture de paquet réseau sur un système, la configuration par défaut de la pile réseau du noyau Linux est d'ignorer les trames qui ne lui sont pas destinées (c'est-à-dire avec une adresse MAC destination qui n'est portée par aucune de ses interfaces). Pour changer ce comportement, il faut passer l'interface en mode promiscuité (*promiscuous*). Les utilisateurs banalisés (autre que "root") n'ont pas le droit de réaliser cela. Cependant, le descripteur de processus dispose d'un attribut `cred`. Dans cet attribut, il est possible de définir des capacités comme, par exemple, la capacité CAP_NET_ADMIN lui permettant de passer une interface réseau en mode promiscuité.

Nous conclurons cette partie sur le contrôle d'accès avec une fonction historique d'isolation des processus nommée `chroot()`. Cette fonction permet de changer la racine du système de fichiers pour un processus en modifiant un attribut associé à la structure `task_struct`. En conséquence, le processus n'a qu'une vision partielle du système de fichiers. En revanche, cette isolation est limitée car comme le rappelle Alan Cox sur un fil de discussion de la liste des développeurs du noyau Linux [6] :

chroot is not and never has been a security tool. People have built things based upon the properties of chroot but extended (BSD jails, Linux vserver) but they are quite different.

En effet, comme nous le détaillons dans l'annexe A.14, il est nécessaire d'adjoindre à `chroot()` d'autre mesures, *a minima* une transition vers une identité banalisée avec `setuid()`, pour garantir un peu de sécurité à un processus isolé du système de fichiers.

2.4 Synthèse sur les processus sous Linux

Dans ce chapitre et l'annexe A associée, nous avons donné toutes les clés relatives au processus et nécessaires à notre sujet. Il nous paraît extrêmement important d'avoir les idées claires sur leur fonctionnement car la notion de processus est au cœur de notre travail. Nous avons ciblé les concepts clés qui vont nous permettre d'appréhender tous les aspects d'isolations inhérents aux conteneurs. En effet, un conteneur est avant tout un processus auquel s'ajoutent des propriétés d'isolation.

Cependant, il y a bien eu un avant conteneur dans l'isolation des processus. Nativement, un processus est isolé des autres car il possède son propre espace mémoire (cf annexe A.3). Il est également isolé du noyau car il est obligé de recourir à une interface

6. https ://lkml.org/lkml/2007/9/26/87

de communication nommée appel système (cf section 2.2.2 et annexe A.6) qui limite ses interactions sur les ressources. De plus, ses interactions avec le système de fichiers sont généralement limitées par un modèle discrétionnaire se fondant sur l'identité du processus (cf section 2.3, notamment la partie DAC dans l'annexe A.11) pour lui appliquer une matrice de règles (ACL).

Concernant toujours les systèmes de fichiers, le noyau peut en limiter la vue via le mécanisme des chroot (cf annexe A.14). Les systèmes disposent également de mécanismes de sécurité basés sur la responsabilité, pour déléguer, si besoin, une partie de privilèges élevés, ou capacités, sur le système à certains processus (cf annexe A.12). Les conteneurs s'inscrivent donc comme un mécanisme d'isolation supplémentaire dans la continuité des précédents.

CHAPITRE 3

ÉLÉMENTS DU HPC : ORDONNANCEMENT DE TRAVAUX

Le HPC est la discipline qui étudie l'efficacité des traitements de données automatisés et la meilleure référence du moment est sans contexte l'ouvrage de David Barkai [12] publié en 2023. Ce livre intitulé « Unmatched – 50 Years of Supercomputing »reprend tous les éléments historiques, techniques et scientifiques du HPC des cinquante dernières années. À coup sur, les réponses aux questions que le lecteur pourrait se poser à la suite de la lecture de ce chapitre, s'y trouveront et de manière complémentaire. Nous invitons donc le lecteur à consulter cette référence bibliographique en cas d'interrogations sur le sujet.

La discipline du HPC entremêle les aspects matériels et logiciels en optimisant l'utilisation du premier par le second. Notre approche est clairement orientée vers l'aspect logiciel du HPC. Dans [13] les auteurs dressent un panorama synthétique et surtout présentent des perspectives dans lequel notre travail s'inscrit. D'un point de vue historique, il y a eu plusieurs tendances matérielles qui se sont succédées au fil du temps. Nous distinguons cinq grandes ères :

— Des années 40 à 70 : les premiers super-ordinateurs ;
— Des années 70 au milieu des années 80 : c'est le règne des machines dites CPU (*Central Processing Unit*). C'est l'ère du calcul à mémoire partagée ;
— Milieu des années 80 à 2000 : l'émergence des clusters de calcul. C'est l'ère du calcul à mémoire distribuée. C'est également au début des années 90 que le TOP500 est né ; Le TOP500 est un projet de classification, par ordre décroissant, des 500 super-ordinateurs les plus puissants au monde ;
— De 2000 à 2010 : les architectures hybrides GPU / CPU ;
— De 2010 à maintenant : les accélérateurs et l'informatique en nuage.

3.1 Historique du HPC

Dans notre contexte, il est crucial de noter qu'aux quatre premières ères il faut donc ajouter une tendance qui a émergé au milieu des années 2010, le HPC dans le Cloud. Plusieurs entreprises telles que R-HPC, Amazon Web Services, Univa, Silicon Graphics International, Sabalcore, Gomput, et Penguin Computing se sont attelées à proposer des services de HPC en mode Cloud. Cette nouvelle tendance démocratise le HPC pour les entités n'ayant pas forcément les moyens (ou des besoins journaliers) de financer une infrastructure de calcul en local.

En examinant l'histoire du HPC sur les 50 dernière années, la typologie précédente peut également être délimitée par cinq « époques » décennales définies par les thèmes architecturaux suivants des systèmes : processeurs vectoriels, multiprocesseurs, microprocesseurs, grappes, accélérateurs et informatique en nuage. Nous y reviendrons à la section 3.1.5, après avoir détaillé la typologie introduite au dessus.

3.1.1 Les années 40 à 70

Dans les années 40 à 70, le développement des super-ordinateurs fut largement piloté par les instances militaires des différents protagonistes. Durant la seconde guerre mondiale, deux codes étaient utilisés par les Nazis pour chiffrer leurs échanges : Enigma [14] et Lorenz [15]. Ces deux codes furent cassés par les britanniques avec le perfectionnement de la Bombe électromécanique et les supercalculateurs Colossus Mark I et II. La Bombe historique est développée en 38 en Pologne puis améliorée par les Britanniques. En résumé, cette Bombe est composée de plusieurs machines Enigma en parallèle pour casser le code. C'est en quelque sorte un ancêtre du *multi-thread*. Les supercalculateurs Colossus Mark I et II furent utilisés pour casser le code de Lorenz.

Ces deux machines étaient extrêmement spécialisées sur le cassage de code. Le couplage entre le matériel et l'usage était au maximum. En 1945, les États-Unis ont développé l'ENIAC (Electronic Numerical Integrator and Computer). Cette machine est le premier ordinateur entièrement électronique pouvant être Turing-complet. Turing-complet signifie que la simulation d'une machine de Turing universelle (compter, comparer, lire, écrire, etc.) est possible. Ainsi, il peut être reprogrammé pour résoudre, en principe, tous les problèmes calculatoires. C'est ce premier découplage entre les ressources et l'exécution, c'est-à-dire entre le matériel et le logiciel, qui fait dire à certains que l'ENIAC est le premier ordinateur conçu [16].

Juste après la seconde guerre mondiale, les États-Unis et la Russie entrent dans une guerre larvée qui s'installe progressivement entre les années 1945 et 1947 et dure jusqu'à la chute des régimes communistes en Europe en 1989. Durant cette période, deux machines posèrent d'autres bases de l'informatique moderne : le CDC 6600 [17] en 1964 et l'ILLIAC-IV [18] en 1966. Le CDC 6600 produit par la société américaine « Control Data Corporation » et conçu par Seymour Cray fut le premier ordinateur utilisant un processeur multi-cœurs superscalaire. Avec sa performance de crête à 1 MFLOP, il fut le premier ordinateur qualifié de super-ordinateur (*supercomputer*). En 1966, l'ILLIAC IV

est la première machine massivement parallèle avec ses 256 FPUs (*Floating Point Units*) sur 64 bits et ses quatre CPUs capables de traiter un milliard d'opérations par seconde. En dépit de ses 50 MFLOP, cette machine SIMD (*Single Instruction, Multiple Data*) fut considérée comme un échec en terme de rapport puissance / investissement (31 millions de dollars). Les résultats sont en dessous des projections initiales.

En résumé, cette ère a jeté les bases des architectures que nous connaissons actuellement. La transition du tout mécanique vers l'électronique a permis de dé-corréler l'usage et le matériel ouvrant la voie aux architectures que nous connaissons actuellement à savoir des systèmes plus ou moins généralistes exploitant un matériel (un système d'exploitation en somme).

3.1.2 Les années 70 au milieu des années 80

Les années 70 à 90 furent les années que certains appellent l'ère de Cray. Cray est le nom d'une entreprise américaine fondée en 1972 par Seymour Cray (anciennement CDC), sous le nom de « Cray Research ». L'année 1972 fut charnière car Seymour Cray était bloqué dans une voie sans issue sur la série des CDC X600. D'ailleurs Jim Thornton qui était son partenaire chez CDC était parti sur une architecture radicalement différente nommée CDC STAR-100 [19]. Le CPU principal était moins complexe que la série des X600. En revanche, il était épaulé par plusieurs coprocesseurs dédiés à tes tâches précises. C'était également la première machine à processeurs vectoriels. Les architectures vectorielles utilisent un pipeline optimisé pour traiter une opération isolée à virgule flottante sur un gros volume de données. Pour que les performances soient optimales, il faut qu'un flux de données arrive de façon ininterrompue au processeur vectoriel. Malgré ses 100 MFLOP, cette machine fut jugée décevante principalement du fait que peu de programmes peuvent être vectorisés efficacement sous forme d'instructions isolées. En effet, quasiment tous les traitements au moment n s'appuyaient sur les résultats des traitements $n-1$, les résultats devaient nettoyer les pipelines avant de pouvoir être réinjectés.

En parallèle, en 1976 Cray Research accoucha du Cray-1 [20] installé au LANL (Los Alamos National Laboratory). Dans la suite du STAR-100, le Cray-1 implémenta un processeur vectoriel mais avec plus de succès que son ancêtre. Le Cray-1 disposait également d'un processeur scalaire. Le design particulier en forme de « C » du Cray-1 permettait aux parties sensibles à la vitesse de cadençage d'être proches les une des autres réduisant la distance de câbles. Ainsi la fréquence obtenue était de 80 MHz pour un total de 133 MFLOP. Grâce à toutes ces qualités, Cray Research vendit 80 copies du Cray-1 ce qui est un record pour les machines de type super-ordinateur.

Dans la lignée du Cray-1, Cray Research sort respectivement les modèles Cray X-MP [21] et Y-MP [22] en 1982 et 1988 respectivement. Le X-MP était cadencé à 105 MHz et embarquait deux processeurs vectoriels de 200 MFLOP chacun. Le Y-MP était cadencé à 167 MHz et embarquait 2, 4 ou 8 processeurs vectoriels de 333 MFLOP chacun. Sur ces deux machines la mémoire était partagée entre les processeurs. Ce principe est fondateur du calcul haute performance. En effet, ce paradigme a fait école et se nomme calcul à mémoire partagée ou « *shared memory* ».

3.1.3 Du milieu des années 80 à l'an 2000

Ces années entraînent une rupture assez nette dans le monde du HPC. En effet, la complexité d'organiser un accès concurrent à une mémoire partagée augmente avec le nombre de processeurs consommateurs de celle-ci. Le paradigme unique de la mémoire partagée constituait ainsi une impasse dans l'accroissement des performances de calcul. Un second paradigme est donc venu le compléter : la mémoire distribuée ou « *distributed memory* ». Par opposition à la mémoire partagée, dans la mémoire distribuée les processeurs ont chacun leur propre mémoire. Ainsi, la difficulté n'est plus de synchroniser les accès concurrents en mémoire mais d'implémenter un système d'échange de messages performant car la communication en mémoire partagée n'est plus suffisante. Dès le milieu des années 80, plusieurs machines à mémoire distribuées fleurissent. Durant ces années, le TOP500 [23] est également créé. Depuis 1993 et deux fois par an, il donne la liste des 500 machines les plus puissantes du monde (hors machines sensibles type militaire) classées par ordre décroissant par leur résultat au *benchmark* LINPACK [24] qui mesure la performance de calcul en virgule flottante des super-ordinateurs.

En 1985, Intel lança l'iPSC/1 Hypercube. Un cluster basé sur ce modèle pouvait connecter de 32 à 128 nœuds. Chaque nœud embarquait un processeur 80286 accompagné d'un coprocesseur arithmétique 80287. La connexion entre les processeurs se faisait selon un modèle Hypercube. En effet, chaque nœud embarquait huit interfaces réseaux. L'une d'elle était dédiée à l'administration par un nœud maître, tandis que les sept autres assuraient le raccordement avec les autres pour constituer l'hypercube ($2^7 = 128$). Le média de connexion physique était l'Ethernet et Intel fournissait un logiciel nommé NX pour gérer les communications entre les nœuds. En 1987, Intel lance son successeur l'iPSC/2 [25] qui, en plus de proposer un accroissement des performances, proposait également un réseau d'interconnexion basé sur le protocole Direct-Connect propriétaire. De plus, la conception de l'iPSC/2 était modulaire et certains composants étaient interchangeables. Enfin, il intégrait un système de fichiers distribué pour partager les données sur les différents nœuds. En 1990, Intel sort la dernière génération de cette gamme, le iPSC/860 [26], qui accroît les performances de la génération précédente.

Cette machine pose les bases des architectures de cluster en mode mémoire distribuée. Les composants essentiels sont :

— Des nœuds de calcul pour exécuter les programmes ;
— Un noeud maître qui ordonnance le travail sur les nœuds de calcul ;
— Un système d'échange de messages ;
— Un système de fichiers distribué pour l'accès aux données sur le cluster.

De plus, l'iPSC est un assemblage de composants courants « sur étagères » et développés ou adaptés pour la machine. Cette hétérogénéité est toujours de mise. Par exemple, la machine Fugaku [27] lancée en 2020 utilise un processeur sur base ARM, l'A64FX, fondu pour la machine et nécessitant des compilateurs adaptés.

En 1985 également, la société TMC (*Thinking Machines Corporation*) lançait le CM-1 (Connection Machine 1) et en 1987 son successeur, la CM-2 [28]. La machine CM-1 embarquait des processeurs très simples à 1 bit avec 4 Ko de RAM. Elle prenait la forme

d'un cube divisé en huit sous-cubes. Chaque sous-cube contenait 16 circuits imprimés et un processeur contrôleur. Chaque circuit imprimé contient 32 puces. Chaque puce disposait de 16 de ces processeurs simples et un routeur pour les communications. Ce qui donne un total de 65 536 processeurs simples.

En 1987, TMC sortait le CM-2 qui intégrait un coprocesseur arithmétique et plus de RAM. En 88, TMC sort une version allégée le CM-2a à 4 096 ou 8 192 processeurs simples et une version plus rapide le CM-200. Ces trois machines étaient adossées à un stockage « Datavault » qui prenait deux interfaces pour les entrées sorties : une E/S Ethernet pour piloter le stockage et un E/S propriétaire à haute vitesse pour les données. A contrario des iPSC manufacturées par Intel, cette génération de machines distribuées développées par TMC étaient entièrement propriétaires. En 91, TMC a pris un virage avec l'adoption du processeur RISC (*Reduced Instruction Set Computing*) SPARC pour le CM-5 (Super SPARC pour le CM-5E) passant du modèle SIMD au MIMD. Cependant des possibilités d'émulation du SIMD étaient offertes aux utilisateurs. Cette génération marqua l'abandon du stockage Datavault.

En 1994, à la NASA, Donald Becker et Tom Stirling ont monté un cluster composé d'ordinateurs et matériels réseaux standards [29]. Il s'agissait de 16 ordinateurs 486 DX interconnectés avec un réseau Ethernet à 10 Mo/s. Ce cluster dégageait 1 GFLOP pour un investissement de 50 000 dollars ce qui en fait le meilleur rapport qualité / prix de l'époque. Stirling a nommé cette catégorie de cluster Beowulf qui est un vieux poème epic Anglais contenant l'extrait suivant : *« Because my heart is pure, I have the strength of a thousand men. »* soit *« Car mon cœur est pur, j'ai la force de mille hommes »*. Pour tirer parti de ces clusters, deux choix s'offraient aux utilisateurs. Le premier est l'adoption d'un système à image unique (ou SSI pour *Single System Image*) comme Mosix, OpenMosix ou plus récemment Kerrighed développé en France. Ces systèmes offraient une image unifiée du matériel agrégé. On avait l'impression d'avoir une seule machine qui était la somme des ressources du cluster. Le second choix était de monter une pile logicielle avec un ordonnanceur, un système de fichiers distribué et un système de transmission de messages. Nous discuterons ces derniers points plus en détails dans la section 3.2 dédiée aux clusters Beowulf.

3.1.4 Depuis 2000

Depuis les années 2000, les clusters sont construits sur un modèle Beowulf dans le sens ou ils sont construits sur du matériel standardisé. Les déclinaisons HPC que les constructeurs proposent de leurs produits sont généralement des modèles adaptés de leurs architectures standards. Sur ce matériel, on retrouve un ordonnanceur, un système de fichiers distribué et un système de transmission de messages. La montée en puissance des machines ne passe plus par l'accroissement de la cadence des processeurs mais plutôt par celui du nombre de cœurs. Avec le développement des GPUs [30] (*Graphical Processing Unit*) et autres accélérateurs matériels polyvalents, les supercalculateurs du Top 500 sont des clusters hybrides mixant un grand nombre de noeuds embarquant des processeurs standards multi-cœurs disposant d'un cache individuel (L1 + L2) et partagé (L3) ainsi que de la mémoire vive avec des nœuds GPU (ou autres accélérateurs) utilisés pour décharger

les CPU de certains types d'opérations.

En 2018, la machine Summit [31] était première au TOP500. Manufacturée par IBM, elle est localisée à l'OLCF (*Oak Ridge Leadership Computing Facility*). Cette machine est composée de 4 608 nœuds. Chaque nœud possède 2 processeurs IBM Power 9 associé à 512 Go de DDR4 SDRAM. Chaque nœud embarque également 6 GPU Tesla avec 96 Go de mémoire HBM2. Ces deux familles de mémoires sont agrégées sur le modèle de la mémoire cohérente pour proposer au total environ 600 Go adressable à la fois par le CPU et le GPU. Enfin, cette mémoire peut être complétée par 800 Go de mémoire RAM non volatile qui se positionne entre le nœud et le stockage distribué. Les nœuds sont interconnectés par un réseau Infiniband, organisé en *Fat-tree* non bloquant, à 200 Go/s entre les nœuds. Le stockage total de la machine s'élève à 250 Po offert par un cluster de 77 nœuds IBM ESS Storage [32]. Au total, la machine dégage 200 PFLOP.

En 2020, Fugaku [27] succède à Summit à la tête du TOP500. Manufacturée par Fujitsu et soutenue par le Riken, la machine s'appuie sur un processeur ARM A64FX embarquant 48 cœurs de calcul accompagnés de quatre cœurs pour les E/S. Ce processeur propose des extensions propres à Fujitsu (*Hardware barrier*, *Sector cache* et *Prefetch*). En conséquence, chaque nœud tourne avec un système d'exploitation basé sur un Linux modifié pour prendre en considération ces extensions. En effet, sur chaque nœud, deux noyaux cohabitent sur le modèle *Multi-kernel*. Le premier est un noyau Linux modifié pour implémenter l'interface IHK (*Interface for Heterogeneous Kernels*) qui lui permet de démarrer un noyau McKernel [33] extrêmement léger ne contenant que du code développé par le Riken. Ce noyau possède ses propres mécanismes de gestion de la mémoire, du *multi-threading* ainsi que ses compteurs de performance. Il implémente certains appels systèmes critiques pour le calcul hautes performances en laissant le noyau Linux gérer le reste.

Le *Multi-Kernel* est différent de la virtualisation (qui est, en somme, un autre moyen de faire fonctionner plusieurs noyaux en parallèle sur le même matériel) dans la mesure où l'hyperviseur est absent et que l'accès est direct au matériel réel sans émulation. Une autre originalité de Fugaku est qu'il n'intègre aucun GPU ou autre accélérateur. La machine compte au total 158 976 nœuds avec un stockage décomposé en trois niveaux L1, L2 et L3. Le L1 est un NVMe SSD de 1,6 To partagé par 16 nœuds. Le L2 est un système de fichiers distribué Lustre de 150 Po partagé par tout le cluster. Enfin, le L3 est un stockage élastique en mode Cloud. Au total, la machine dégage 537 PFLOP.

En 2021, Frontier prend la première place du TOP500 [34]. Manufacturée par HPE (via l'acquisition de Cray), elle est opérée à l'OLCF. C'est la première machine à accéder à l'exaflops avec 1 685 PFLOP. Cette machine est composée de 9 408 nœuds. Chaque nœud embarque un processeur AMD Epyc 7453s « Tento » avec 64 cœurs accompagné de quatre GPU Radeon Instinct MI250X AMD et 512 Go de RAM. Le CPU Epyc inclut deux personnalisations propres à Frontier. La première est que des interconnexions xGMI (Socket to Socket Global Memory Interface) ont été ajoutées pour proposer une mémoire cohérente à destination du CPU et du GPU. La seconde est que des ports PCIe ont été ajoutés pour accueillir 4 To de stockage en NVM sur chaque nœud. De plus, chaque GPU a un accès direct aux quatre interfaces réseaux embarquées sur le nœud. En conséquence, la mémoire HBM de l'accélérateur est au plus près du réseau. Les nœuds sont connectés

selon la topologie « *Dragonfly* » [35]. Cette topologie est motivée par le coût. En effet, 90% des câbles sont des cuivres standards pour 10% en optique. Enfin, Frontier s'appuie sur un stockage Lustre de 700 Po.

3.1.5 L'histoire du HPC sous l'angle de l'architecture des systèmes

De manière complémentaire et comme nous l'avons brièvement signalé aux points précédents, l'histoire du HPC peut se décliner selon la typologie des processeurs vectoriels, multiprocesseurs, microprocesseurs, grappes, accélérateurs et informatique en nuage. Il s'agit d'un point de vue à la fois matériel et architectural. Nous ne détaillons ici qu'une partie de l'histoire, vue sous cet angle.

Un processeur vectoriel met en œuvre un jeu d'instructions conçu pour fonctionner efficacement sur de grands tableaux unidimensionnels. Ils peuvent améliorer considérablement les performances de certaines applications régulières comme la simulation numérique et l'algèbre linéaire. Les premiers super-ordinateurs vectoriels ont vu le jour au début des années 70 chez les grands constructeurs, dont Cray.

Dans les années 90, est apparue la notion de « *Cluster Computing* » et également de grille de calcul. L'idée était de concevoir des super-ordinateurs avec du matériel généraliste en grand nombre. A cette époque, on a vu l'émergence du calcul pair-à-pair (*Volunteer Computing*) ou quiconque ayant des PCs connectés à Internet pouvaient participer à une expérience scientifique en donnant du temps de calcul. Le projet Seti@Home [36] a alors connu un succès notoire. Il a permis de montrer la faisabilité d'un super-ordinateur spécialisé dont les nœuds de calcul étaient géographiquement distants et dont l'interconnexion reposait sur un réseau grand public.

La synthèse *Desktop Grid Computing* [37], publiée en 2012 retrace quinze ans de recherche dans ce domaine. L'ouvrage présente les techniques communes utilisées dans de nombreux modèles, algorithmes et outils développés au cours de la décennie 1990 pour mettre en œuvre l'informatique en grille. Ces techniques permettent de résoudre de nombreux sous-problèmes importants pour la conception d'intergiciels, notamment l'ordonnancement, la gestion des données, la sécurité, l'équilibrage des charges, la certification des résultats et la tolérance aux pannes. La première partie du livre couvre les idées initiales et les concepts de base de l'informatique en grille. La deuxième partie explore les problèmes que la communauté présentait comme actuels et futurs.

A partir de 2010, les machines se sont dotées d'accélérateurs qui consistent à confier une fonction spécifique à un coprocesseur dédié pour l'effectuer de façon plus efficace. Aujourd'hui, tous les processeurs grand public récents intègrent plusieurs unités dédiées. Cela est rendu possible grâce à la gravure toujours plus fine des circuits intégrés, libérant ainsi des transistors pour les accélérateurs. Dans les super-ordinateurs actuels, les accélérateurs sont hérités des GPUs (eux même héritiers des supercalculateurs SIMD, cf. section 3.1.1). Leur parallélisme poussé les rend très performants pour du calcul matriciel. Il peut aussi s'agir d'accélérateurs pour certaines opérations en apprentissage automatique comme les TPU (*Tensor Processing Unit*) de Google. Il s'agit d'une tendance forte en 2023. Apple a prévu des processeurs en gravure 3nm pour 2024.

3.1.6 Synthèse de l'historique du HPC

Une tendance émerge de cette histoire. Au fil du temps les super-ordinateurs et cluster HPC sont de moins en moins spécialisés laissant la place à des composants matériels et logiciels de plus en plus courants. La spécialisation a plutôt glissé du côté des fondeurs qui proposent des déclinaisons HPC de leurs produits. Cette standardisation matérielle a amené une certaine homogénéité des piles logicielles favorisant le développement de librairies de haut niveau ce qui a considérablement démocratisé le calcul scientifique. Ces machines sont évaluées dans un classement bi-annuel nommé TOP500. Cependant, le *benchmark* Linpack ne mesure l'efficacité que d'un type d'opération. D'après Jack Dongara : *« That is because it only tests the resolution of dense linear systems, which are not representative of all the operations usually performed in scientific computing. »*[1]. Un autre *benchmark* nommé HPCG [38] est donc venu le compléter, notamment en générant des données sur disque (1To). Cependant ces deux *benchmarks* ne mesurent que la puissance de calcul des machines. Avec la prise de conscience du coût énergétique d'un calcul, le GREEN500 [39] est né en 2007. Il complète le score du TOP500 avec un score d'efficacité énergétique.

Pour conclure, les architectures informatiques n'ont cessé d'osciller entre des architectures centralisées et décentralisées. Le *cloud computing* d'aujourd'hui est une architecture centralisée. Celle qui précédait, le *grid computing*, l'était beaucoup moins, tout comme les architectures *Edge* et *Fog* qui se déploient depuis 2020 pour lesquelles le calcul est effectué en bordure de réseau plutôt que dans un centre de données. Cependant, en 2023, les centres de calcul HPC pour la production fonctionnent toujours selon une architecture de type cluster, c'est-à-dire avec des processeurs et accélérateurs dédiés au HPC et contrôlés par un ordonnanceur de tâches qui alloue les travaux aux nœuds de calcul.

3.2 Les clusters HPC Beowulf

Comme énoncé dans la section 3.1.4, les clusters Beowulf sont des agrégats de machines interconnectées par des réseaux plus ou moins performants pour collaborer à la résolution de problèmes impossibles à traiter sur une machine isolée et standard. Les difficultés pour maintenir une telle infrastructure sont triples :

— Administrer une machine utilisée par des populations aux besoins orthogonaux. Certains utilisateurs vont développer des codes et, selon la maturité de leur développement, chercher à les déboguer ou bien à les étalonner pour évaluer leur passage à l'échelle. D'autres utilisateurs vont faire tourner des codes sur étagère éprouvés pour la production dans le cadre de leur travail expérimental. Pour chacune de ces deux populations, soit l'objectif est de produire les résultats expérimentaux finaux, soit de monter une preuve de concept pour solliciter des heures de calcul sur un cluster d'échelle supérieure ;
— Proposer un environnement de développement accessible à une population d'utilisateurs très hétérogène ayant des affinités plus ou moins fortes avec le développement

1. https ://www.top500.org/news/jack-dongarra-on-top500-past-present-and-future

d'applications parallèles et l'utilisation des outils informatiques. Un tel environnement est composé d'un cocktail de librairies dont la difficulté d'utilisation croît avec leur proximité par rapport aux couches basses du système ;
— Proposer un accès au cluster suffisamment ergonomique pour que les utilisateurs puissent lancer leurs travaux simplement en définissant leurs contraintes pour les ventiler sur les ressources libres.

3.2.1 Architecture d'un cluster HPC typique

Dans cette section, nous allons aborder la typologie type des centres de calcul. Au niveau fonctionnel, on distingue quatre types de machines qui composent une architecture standard de centre de calcul :

— Le(s) nœud(s) frontal (frontaux) sur le(s)quel vont se connecter les utilisateurs afin de récupérer ou charger leurs fichiers, compiler leurs codes ou encore exécuter les travaux sur le cluster ;
— Le(s) nœud(s) d'administration accessible(s) uniquement à l'administrateur et qui héberge les services de base du cluster type service de déploiement, annuaire des utilisateurs etc. ;
— Le(s) nœud(s) de stockage qui ser(ven)t un système de fichiers distribué sur l'ensemble du centre de calcul ;
— Les nœuds de calcul qui exécutent les travaux des utilisateurs. Ces nœuds embarquent les périphériques liés au calcul type CPU avec de nombreux cœurs, GPU, une quantité de mémoire importante etc.

En conséquence, un centre de calcul est une machine partagée entre de multiples utilisateurs. Chaque utilisateur dispose d'un compte sur le système avec un espace personnel qui lui est dédié et distribué sur les nœuds de calcul. En d'autres termes, les fichiers présents dans l'espace personnel de l'utilisateur sur le frontal sont également présents au même endroit sur chaque nœud de calcul.

Le contrôle d'accès aux fichiers est basé sur le DAC (cf. annexe A.11). Quelques expérimentations ont été faites sur l'intégration d'un contrôle d'accès MAC (cf. section A.13) aux clusters HPC [40]. Cependant, superposer une politique de contrôle d'accès mandataire à un système dont les programmes évoluent sans cesse et de façon spontanée (chaque utilisateurs peut installer ses propres codes) est beaucoup trop compliqué. Ainsi, le code s'exécute avec les droits de l'utilisateur sur le système. Il est donc soumis au contrôle d'accès UGO standard des systèmes Linux basé sur l'UID du possesseur du fichier exécutable. Les UID doivent être consistants sur tout le cluster car dans le contexte du HPC, ces codes peuvent se répartir sur plusieurs nœuds. Nous allons maintenant discuter les deux paradigmes d'exécution des codes sur un cluster HPC à savoir les exécutions à mémoires partagées et distribuées.

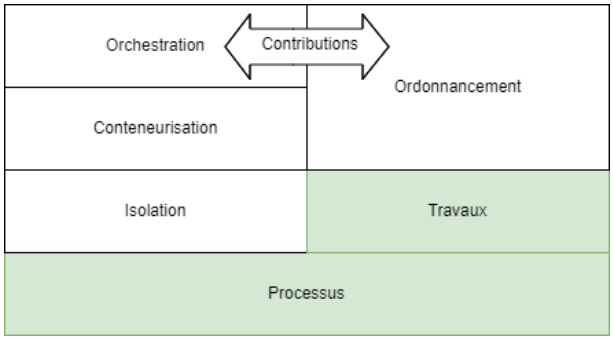


Figure 3.1 – Organisation et dépendances des notions

3.2.2 Types de travaux

Comme mentionné dans la section précédente, il y a deux paradigmes d'exécution des codes. Le premier est le mode mémoire partagée ou le code s'exécute localement sur la machine. Le second est le mode mémoire distribuée ou le code est éclaté et s'exécute sur plusieurs nœuds. Cette distribution implique des communications inter-nœuds qui ajoutent une couche de complexité de synchronisation des échanges par le réseau. Les travaux peuvent être vus comme des processus à lancer accompagnés d'une description des ressources qu'ils requièrent pour s'exécuter sur le cluster HPC. Ces travaux sont le plus souvent parallèles, c'est-à-dire qu'ils occupent plusieurs cœurs localisés sur un même nœud de calcul (mémoire partagée) ou sur des cœurs répartis sur plusieurs nœuds de calcul (mémoire distribuée). La figure 3.1 représente l'avancement de la discussion du sujet.

Mémoire partagée

Comme mentionné dans l'annexe A.8, lorsqu'un programme doit travailler de façon concurrente sur plusieurs cœurs de la même machine, il utilisera des *threads* plutôt que des processus systèmes indépendants au niveau espace mémoire. En effet, les *threads* partagent des zones mémoires ce qui leur permet de communiquer de façon efficace au prix d'un travail de synchronisation d'accès aux variables. En conséquence, ils peuvent communiquer par des moyens à très faible latence telle que la mémoire partagée (cf. section A.4). Cependant, ces exécutions concurrentes impliquent un travail de gestion des accès aux variables partagées. Sous Linux, la librairie *pthread* fournit les fonctions de bas niveau pour créer et manipuler les *threads* . Dans les versions de la Glibc antérieures à la 2.34 cette librairie était incarnée par un objet externe (libpthread.so). Dans les versions supérieures les fonctions associées aux *threads* sont incluses directement dans la librairie de base `libc.so.6`.

Cependant, la manipulation des mécanismes d'exclusions mutuelles ou de sémaphores peut être compliquée pour les utilisateurs peu familiers avec les problématiques de programmation système. Il existe donc une API de plus haut niveau nommée OpenMP [41] permettant d'abstraire ces difficultés au prix d'une légère dégradation des performances [42]. OpenMP comporte trois composants distincts : les directives de compilation, les routines de la bibliothèque d'exécution et les variables d'environnement. Avec OpenMP, il

suffit d'indiquer au compilateur via des balisages dits `pragmas` que l'on entre dans une boucle parallélisable, c'est-à-dire dont les itérations sont indépendantes les unes des autres.

La figure 3.2 montre la séquence d'exécution d'un programme sur une machine disposant d'un processeur physique doté de quatre cœurs *hyperthreadés*. Un cœur hyperthreadé peut activer un second fil d'exécution partageant son cache. Cela permet de doubler le nombre de processus en exécution en parallèle. Cependant, les performances peuvent se dégrader avec l'activation des hyperthreads si le calcul fait un usage intensif du cache. Au total nous avons donc $1 \times 4 \times 2 = 8$ fils d'exécution. Nous voyons que le programme démarre sur l'*hyperthread* du second cœur. Sur les processeurs modernes, il est possible d'activer le mode *hyperthread* qui active un second fil d'exécution au niveau du cœur partageant son cache. L'activation est optionnelle selon que vous faites tourner des processus intensifs en cache ou non. Il s'exécute dans un seul *thread* jusqu'à ce que le code arrive dans la section « *#pragma parallel for* ».

A ce moment là, des *threads* vont virtuellement se créer (dans les implémentations efficaces, les *threads* sont créés à l'initialisation du programme). Le nombre de nouveaux *threads* est égal à la valeur de la variable OMP_NUM_THREAD). Les occurrences du contenu de la boucle sont traitées quatre par quatre sur des cœurs / *hyperthread* différents. Le *thread* maître effectue un *wait()* qui est une fonction permettant de placer le processus appelant en attente de la terminaison d'autres processus. En conséquence, le père se retrouve en attente de la fin de l'exécution de ses fils. Lorsque ses fils se terminent, il continue son exécution sur un seul cœur / *hyperthread*.

On notera que OpenMP n'est pas limité à l'exécution sur une seule machine physique. Il existe des systèmes à image unique dont le but est d'agréger plusieurs machines par le réseau pour en faire une seule grosse machine. On citera les projets Mosix [43] et ses alternatives libres OpenMosix, OpenSSI et Kerrighed [44]. Dans ce cas, un travail OpenMP peut se retrouver distribué sur plusieurs machines physiques. Une expérimentation avec Kerrighed est présentée dans [45]. Il existe également des implémentations OpenMP distribués via SDSM (*Software Distributed Shared Memory*). SDSM est un *middleware* fournissant des mécanismes permettant de présenter aux processus une mémoire partagée consistante physiquement répartie sur plusieurs machines. Enfin, nous trouvons le *framework* CAPE (*Checkpointing-Aided Parallel Execution*) [46] qui propose de transformer le code source OpenMP à mémoire partagée en code CAPE. Ce code est compilable avec les compilateurs standards et s'exécute sur des machines distribuées instrumentées avec le *framework* CAPE.

Mémoire distribuée

Lorsqu'un programme fonctionne en mode mémoire distribuée cela signifie qu'il va s'exécuter sur différentes machines interconnectées par un réseau plus ou moins efficace. Les différents processus doivent donc avoir la possibilité de communiquer entre eux pour s'échanger des variables. Afin que ces programmes soient portables d'un centre de calcul à un autre, il a été nécessaire de normaliser ces échanges de messages. C'est ainsi qu'est né le standard MPI (*Message Passing Interface*) faisant suite à PVM (*Parallel Virtual Machine*) dont la première version MPI-1.0 fut publiée en 1994. Le dernier standard MPI-

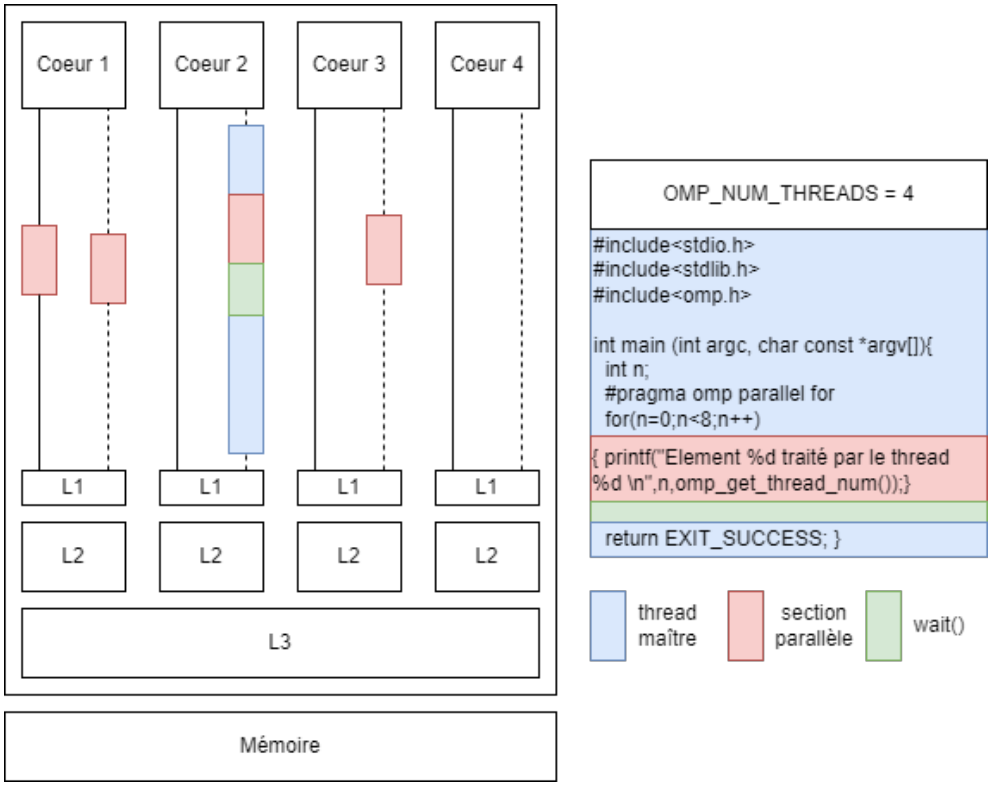


Figure 3.2 – Architecture OpenMP

4.0 date de 2021. Il existe plusieurs implémentation du standard MPI :

— Intel MPI qui est fournie avec la suite de compilateurs Intel ONE API ;
— MPICH qui est une implémentation libre du standard MPI portée par l'ANL (Argonne National Laboratory) et L'université de l'état du Mississippi. MPICH a servi de base à de nombreuses implémentations propriétaires de MPI : IBM MPI (pour Blue Gene), Intel MPI, Cray MPI, Microsoft MPI, Myricom MPI, OSU MVAPICH/MVAPICH2 etc. ;
— MVAPICH/MVAPICH2 est une implémentation portée par l'université de l'état d'Ohio sous licence BSD ;
— OpenMPI qui est l'implémentation libre la plus répandue développée par un consortium d'industriels et d'académiques.

L'idée derrière MPI est d'abstraire le réseau du point de vue du développeur. L'utilisateur choisit le nombre de « *workers* » à démarrer au lancement du programme. Chaque « *worker* » est un processus (cf. annexe A). Lors de l'initialisation du programme, une infrastructure de communication est créée rassemblant tous les nœuds de calculs demandés par l'utilisateur. Le média de communication (TCP, Infiniband, shm etc.) est totalement abstrait pour l'utilisateur car les processus sont ordonnés par rang et on utilise le numéro de rang pour les adresser. Les développeurs n'ont pas à se poser les questions de couples IP/Port pour les communications TCP ou alors d'adresses mémoires pour les shm au cas ou les « *workers* » soient situés sur le même nœud de calcul. La figure 3.3 présente la structure de l'implémentation libre OpenMPI. On y retrouve trois couches d'abstractions successives OPAL, ORTE et OMPI.

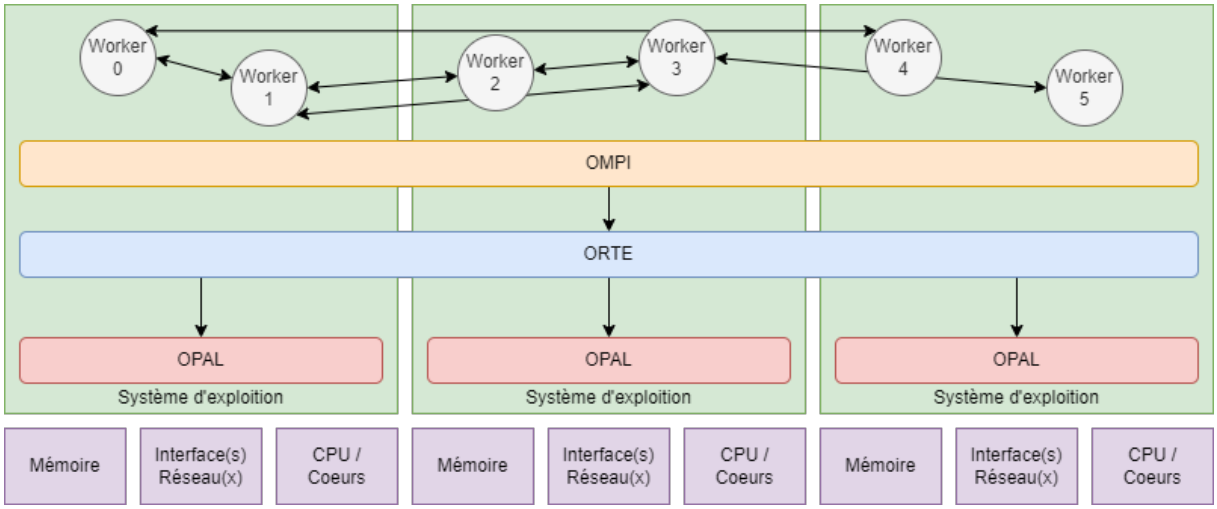


Figure 3.3 – Architecture de OpenMPI

OPAL (*Open Portable Access Layer*) constitue la couche la plus basse, c'est-à-dire la plus proche du système d'exploitation, qui fournit les briques de bases au processus telles que les directives de débogage, les manipulations de chaînes de caractères, listes chaînées etc. OPAL gère également la portabilité d'OpenMPI en implémentant les mécanismes de découvertes d'interfaces réseau, les primitives de communication en mémoire partagée, les horloges disponibles pour l'horodatage des messages ou encore la gestion de l'affinité entre les processus « *worker* » et processeurs (au sens cœurs éventuellement *hyperthreadés*) physiques en collaboration avec l'ordonnanceur du système d'exploitation (cf. annexe A.2).

ORTE (*OpenMPI RunTime Environment*) s'appuie sur les services mis en place par OPAL pour lancer les processus « *worker* » MPI, par exemple par SSH, RSH ou encore directement via l'ordonnanceur HPC (cf. section 3.2.3) sous réserve qu'il le supporte. Une fois les processus lancés, c'est également ORTE qui va les superviser et les tuer. L'existence d'une couche dédiée à la gestion des processus MPI s'explique par la distribution de ceux ci sur une multitude de nœuds de calcul. Les mécanismes standards de supervision des processus natifs sur les systèmes d'exploitation ne sont pas suffisants car ils partent du principe, tout à fait légitime, que les processus tournent sur la même machine, c'est-à-dire dans le même espace utilisateur. Avec MPI, il faut ajouter dans l'équation que ces processus sont localisés sur des machines différentes d'où la nécessité d'une couche de supervision ad-hoc.

OMPI (OpenMPI) est la couche la plus haute manipulée par les développeurs. Cette couche fournit les bibliothèques à lier à l'exécutable ainsi que les bibliothèques à inclure dans les codes sources. On y trouve également le compilateur mpicc (*MPI C compiler*) et le lanceur mpirun (*MPI runtime*). On notera que l'API proposée par OpenMPI implémente le standard MPI spécifié par le MPI forum. Ainsi, chaque code MPI peut utiliser telle ou telle implémentation de MPI.

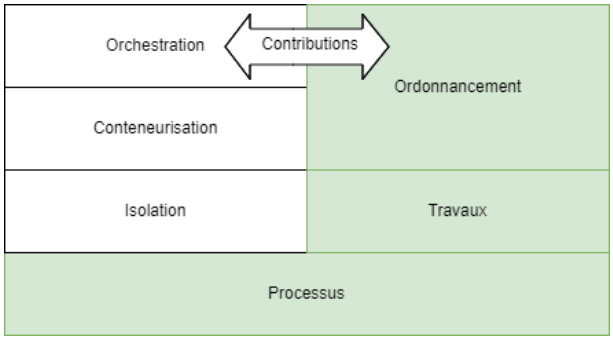


Figure 3.4 – Organisation et dépendances des notions

3.2.3 Ordonnanceur HPC

L'ordonnanceur HPC est le chef d'orchestre du cluster. C'est lui qui va placer les travaux des utilisateurs sur les nœuds de calcul. Il connaît l'état du cluster HPC en temps réel. Il sait quel travail de quel utilisateur est en cours ainsi que la quantité de ressources qu'il consomme. L'ordonnanceur HPC s'exécute sur le(s) nœud(s) d'administration (cf. section 3.2.1) et va distribuer le travail à ses agents installés sur les nœuds de calcul. La place de l'ordonnancement dans notre sujet est précisée dans la figure 3.4

Dans cette section, nous allons donner les dénominateurs communs propres à tous les ordonnanceurs HPC. En définitive, un ordonnanceur HPC embarque quatre composants essentiels :

— Le maître qui va connaître l'état du cluster. Il sait quel travail tourne sur quel(s) nœud(s) et surtout il intègre le(s) algorithme(s) d'ordonnancement. Selon les implémentations des ordonnanceurs HPC, il peut s'agir d'un exécutable monolithique ou de plusieurs processus qui communiquent avec des IPC ;
— L'agent tourne sur les nœuds de calcul du cluster HPC. La mission de cet agent est double. D'une part, il informe le maître sur l'état du nœud de calcul. D'autre part, il lance les travaux sur les nœuds de calcul en suivant la séquence `fork()` $->$ `setuid()` $->$ `exec()`. En conséquence, le travail est un fils de l'agent et il s'exécute avec les privilèges de l'utilisateur qui a soumis le travail ;
— Les outils clients installés sur le nœud frontal permettant d'interagir avec le maître pour soumettre un travail, lister les travaux en exécution, vérifier l'état des nœuds de calcul etc. ;
— Le dernier composant n'est pas obligatoire au sens strict, cependant il est activé en pratique dans quasiment tous les cas. Il s'agit du comptage des ressources consommées par les utilisateurs (*accounting*). C'est ce composant qui va permettre non seulement la facturation des heures de calcul, mais aussi d'utiliser les algorithmes d'ordonnancement se basant sur la consommation passée de ressources (par exemple le *fairshare* [47]).

Le figure 3.5 présente l'intégration des composants d'un ordonnanceur dans un cluster HPC. La séquence typique est que l'utilisateur se connecte au frontal du centre de calcul. Il y charge son script de soumission contenant les contraintes de ressources ainsi que la façon

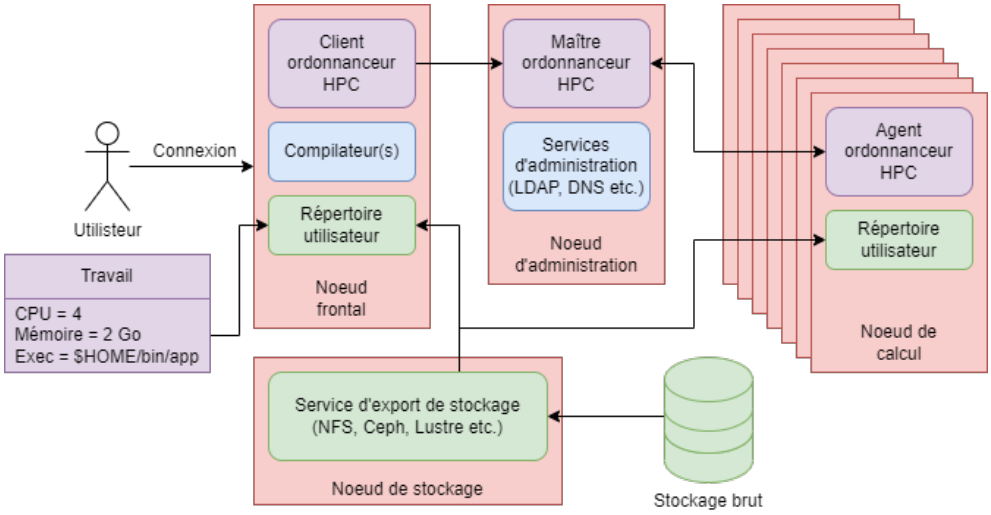


Figure 3.5 – Intégration des ordonnanceurs HPC

de lancer son travail. Éventuellement, il peut également y compiler un code personnel. Ensuite, il va utiliser les outils clients pour envoyer son script de soumission au maître de l'ordonnanceur HPC. Le maître interprète le script soumission du travail et va, soit le mettre en file d'attente si les ressources demandées ne sont pas encore disponibles, soit donner l'ordre à ses agents sur les nœuds de calcul d'exécuter le travail de l'utilisateur. En général l'agent va d'abord créer un processus fils dédié à la supervision du travail de l'utilisateur (notamment les entrées / sorties) qui va lui-même invoquer le(s) processus du code à exécuter. Une fois le travail en exécution le maître met à jour l'état du cluster.

3.2.4 Synthèse de l'ordonnancement de travaux HPC

Dans toute la discussion précédente, nous avons vu que l'architecture physique du cluster HPC est imbriquée avec l'ordonnanceur qui le pilote. Tous les clusters Beowulf sont construits sur le modèle présenté dans la section 3.2.1. Ils sont tous pilotés par un ordonnanceur HPC tel que présenté dans la section 3.2.3. Les ordonnanceurs HPC partagent tous les mêmes propriétés et passer de l'un à l'autre n'est pas très compliqué pour les utilisateurs. Ce qui va vraiment faire la différence entre les clusters HPC c'est leur choix d'architectures matérielles et la ventilation des types de ressources de calcul (CPU, GPU, stockage etc.).

CHAPITRE 4

ISOLATION DE PROCESSUS SOUS LINUX

Il est maintenant temps de détailler un concept clé de notre travail appelé l'isolation. Il existe deux grands paradigmes d'isolation en informatique : la virtualisation et la conteneurisation. La différence entre les deux est que la virtualisation isole un système d'exploitation complet en permettant à plusieurs noyaux de tourner de façon concurrente sur le même matériel tandis que la conteneurisation isole les processus en multi-instanciant certains objets du noyaux. Cependant, lorsque l'on parle d'isolation il faut bien distinguer deux choses : la pondération des interactions et la limitation des interactions. Ces notions se définissent comme suit :

1. La pondération des interactions consiste à associer une qualité de service (ou QoS, *Quality of Service*) par rapport aux ressources consommées par l'entité isolée ;
2. La limitation des interactions, quant à elle, concerne le contrôle d'accès de l'entité isolée sur le reste du système.

Nous commencerons par une brève discussion sur la virtualisation qui va essentiellement nous servir à la positionner par rapport à l'isolation de processus. Nous poursuivrons avec les mécanismes de bas niveau associés aux processus afin des les confiner. Enfin nous verrons comment les conteneurs tirent parti de ces mécanismes d'isolation à travers un état de l'art des moteurs de conteneurs actuellement utilisés. La figure 4.1 présente la position de la section actuelle dans la discussion.

4.1 La virtualisation

La virtualisation est la capacité de faire tourner plusieurs noyaux de façon concurrente sur un même matériel. Le principe général est qu'un composant nommé l'hyperviseur, potentiellement localisé à différents endroits comme présenté dans la figure 4.2 va présenter un matériel émulé sur le matériel réel à différents noyau invités. Ainsi, il est possible de faire cohabiter sur la même machine des systèmes d'exploitation de familles différentes :

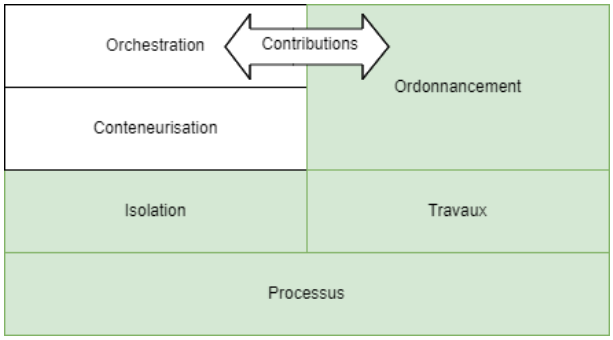


Figure 4.1 – Positionnement de l'isolation dans le sujet

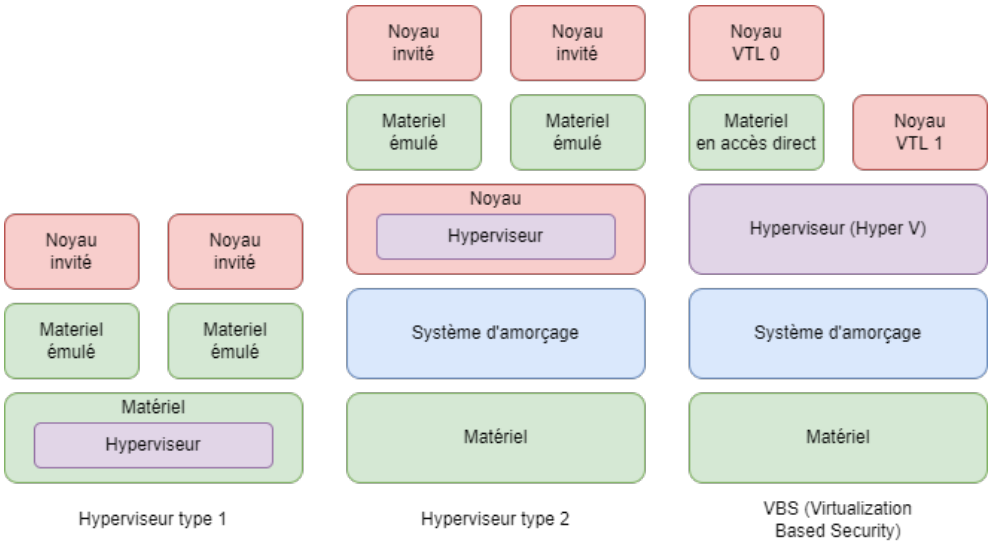


Figure 4.2 – Les familles d'hyperviseurs

Linux, Windows, BSD etc.

La gestion du matériel réel par l'hyperviseur est liée à son type. Dans les hyperviseurs de type 1, un module noyau donne directement aux systèmes invités (virtualisés) l'accès au matériel. Dans certains cas, l'hyperviseur peut même être localisé dans un composant matériel spécialisé connecté à la carte mère. Dans ce cas, Il n'y a même pas de système d'exploitation sur le nœud de virtualisation. Les hyperviseurs de type 2 présentent un matériel totalement émulé au système d'exploitation invité. Par rapport au type 1, nous avons donc un niveau d'indirection supplémentaire entre les noyaux invités et le matériel réel.

Concernant le matériel émulé, certains hyperviseurs proposent une collection de drivers optimisant l'accès à certaines ressources matérielles en rendant l'isolation plus poreuse via l'installation d'un pilote spécifique côté noyau invité. On pourra citer l'exemple de VirtIO qui permet d'adresser certains périphériques de façon plus directe en faisant collaborer le noyau de l'hôte avec le noyau invité via des pilotes spécialisés. Cet accès plus direct est possible sous réserve que le pilote du périphérique en question sur le noyau hôte supporte l'accès via VirtIO. En conséquence, le noyau invité a conscience d'être virtualisé du fait de la présence de ce pilote qui effectue des *hypercalls* à destination de l'hyperviseur hôte. Dans

ce cas de figure on ne parle plus de virtualisation complète mais de para-virtualisation. Enfin, selon les hyperviseurs, il est également possible de donner un accès direct et donc exclusif au noyaux invités à certains matériels de l'hôte. Dans ce cas, l'hyperviseur se laisse volontairement contourner.

Les hyperviseurs de type 2 implémentés dans le noyau du système hôte constituent un code complexe privilégié. D'un point de vue strictement sécurité, il présente donc d'une surface d'attaque conséquente. Cependant, en plaçant l'hyperviseur juste après le système d'amorçage, Microsoft utilise la virtualisation dans une optique strictement sécurité (VBS pour *Virtualization Based Security*). L'idée est que l'hyperviseur démarre en premier pour lancer deux noyaux avec des VTL (*Virtual Trust Level*) différents (0 et 1). Le noyau VTL 0 va embarquer les pilotes du matériel de la machine et accueillir les processus utilisateurs courants type navigateurs web, traitement de texte etc. Le noyau VTL 1 n'embarque que du code Microsoft sans aucun pilote. Le VTL 1 est dédié à héberger les processus manipulant des données sensibles. Pour y accéder, le noyau en VTL 0 doit demander au noyau en VTL 1 de lui transmettre les données (pour cette raison, le VTL 1 est aussi nommé *Proxy kernel*). Ainsi, pour un processus comme LSASS (*Local Security Authority Subsystem Service*) qui manipule des crédentiels d'authentification dans le VTL 0 n'est plus obligé de les stocker dans son espace mémoire mais peu collaborer avec un processus dans le VTL 1 (en l'occurrence LSAISO pour *LSA ISOlated*) qui contiendra ces crédentiels dans un espace mémoire totalement séparé. En conséquence, une extraction de mémoire sauvage du processus LSASS en VTL 0 ne donnera aucun résultat.

La virtualisation est donc un moyen d'isoler un système d'exploitation complet, c'est-à-dire son espace noyau et son espace utilisateur, en lui présentant un matériel plus ou moins émulé. Cependant cette isolation n'est pas forcément adaptée à l'usage ciblé. Par exemple, si le besoin est d'isoler un service web, l'encapsulation dans une machine virtuelle n'est pas nécessairement l'isolation la plus pertinente car la cible est une application unique. Nous allons évoquer deux mécanismes complémentaires appelés Cgroups et espaces de noms (*namespaces*) qui vont respectivement limiter la consommation de ressources du processus et ses interactions avec le système et les autres processus.

4.2 Limitation de ressources avec les Cgroups

Les cgroups sont implémentés dans le noyau Linux et permettent de créer des groupes arbitraires de processus transversaux à l'historique hiérarchie père / fils (cf. section 2.2.3). Chaque groupe peut se voir affecter des sous-systèmes (*subsystem*) qui vont lui imposer des limites de ressources. On parle aussi de contrôleurs (*controllers*), les deux dénominations sont équivalentes. Chaque sous-système superpose plus ou moins un type de ressource. La version actuelle des Cgroups est la V2. Dans la discussion qui suit, nous considérons directement la V2. On peut distinguer deux types de contrôleurs. D'une part, nous avons des contrôleurs qui vont limiter la quantité de ressources utilisables et d'autre part des contrôleurs qui vont autoriser ou non l'accès au ressources. Les cgroups sont vus comme un système de fichiers virtuel monté dans `/sys/fs/cgroup`. Nous allons commencer par explorer les limitations de ressources.

4.2.1 Limiter l'usage des ressources

Dans cette section nous allons illustrer la façon dont les contrôleurs proposés par les cgroups limitent les ressources des processus. Il existe un certain nombre de contrôleurs :

- cpuset : influe sur le placement des processus sur les CPU au moment de leur ordonnancement (cf. annexe A.2) ;
- cpu : limite les cycles CPU alloués au processus ;
- io : limite les ressources d'entrées / sorties typiquement sur les périphériques en mode bloc ;
- memory : limite la consommation mémoire ;
- hugetlb : limite l'utilisation des pages mémoire hautes lors de l'allocation mémoire (cf. annexe A.3) ;
- pids : contrôle la natalité des processus ciblés ;
- rdma : limite la quantité d'accès direct à la mémoire des processus faisant, par exemple, des communications sur Infiniband ;
- misc : permet de créer des compteurs personnalisés pour les ressources qui ne seraient pas représentées par les contrôleurs précédents.

Nous allons dérouler un exemple pour illustrer le fonctionnement des cgroups. Nous allons créer deux cgroups s'appuyant sur les contrôleurs cpu et memory. Le but étant, sur une machine de huit CPUs, de créer un cgroup cghigh limitant l'usage des ressources à quatre CPUs (soit 50%) et 1 Go de mémoire et cglow limitant l'usage des ressources a deux CPU (soit 25%) et 512 Mo de mémoire. Nous lancerons un programme qui va récupérer la limite mémoire de cghigh et la limite CPU de cglow. Pour créer cette hiérarchie, nous exécutons le listing 4.1 qui génère la topologie de Cgroups définie dans la figure 4.3. On retrouve la vue physique telle qu'elle est lorsque l'on consulte le contenu de /sys/fs/cgroup et la vue logique. Notons que les moteurs de conteneurs génèrent ces Cgroups dynamiquement lors de l'instanciation du conteneur.

```
$ cgcreate -g cpu:/cghigh
$ cgcreate -g memory:/cghigh
$ cgcreate -g cpu:/cglow
$ cgcreate -g memory:/cglow

$ cgset -r cpu.max="500000 1000000" cghigh
$ cgset -r memory.high="1024M" cghigh
$ cgset -r cpu.max="250000 1000000" cglow
$ cgset -r memory.high="512M" cglow

$ cgexec -g cpu:cghigh -g memory:cglow ./monprogramme
```

Listing 4.1 – Instanciation de notre exemple

L'idée derrière les cgroups est qu'il faut provisionner des structures de données de type `cgroup`. Chaque `cgroup` possède un attribut `subsys` qui est un tableau de taille égale au nombre de sous-systèmes activés par le noyau. Chaque case du tableau est un pointeur

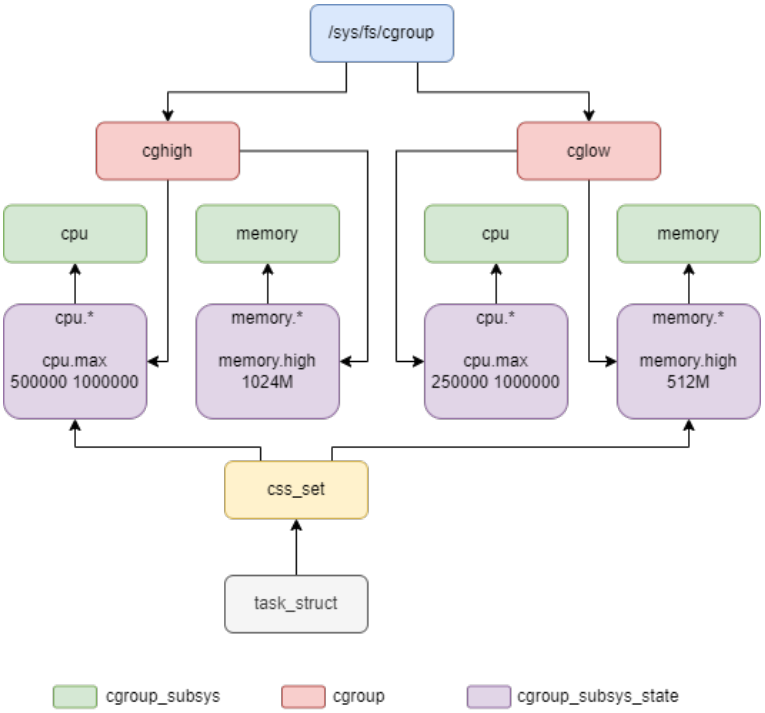


Figure 4.3 – Topologie de notre exemple

sur une structure `cgroup_subsys_state` contenant les paramètres de configuration de chaque sous-système.

Chaque structure `cgroup_subsys_state` contient un pointeur vers la structure `cgroup_subsys` qui est le sous-système associé aux paramètres. Ainsi, lorsque l'on lance le programme, on le fait souscrire à un certain nombre de `cgroup_subsys_state`. La structure `css_set` réalise l'association entre la `task_struct` résultante du lancement du programme et les `cgroup_subsys_state` auxquels elle souhaite souscrire. Ensuite, selon les sous-systèmes choisis, des fonctions sont exécutées par le noyau pour renforcer les limites.

Par exemple pour la limitation CPU, le noyau a créé une structure `task_group` correspondant au `cgroup_subsys_state` cghigh qui va accueillir notre `task_struct` et il aura appliqué la fonction `tg_set_cfs_bandwidth(struct task_group *tg, u64 period, u64 quota)` avec `tg` qui pointe sur le groupe de tâche dont le processus fait partie, `period` qui est la référence de temps (ici 1000000) et `quota` qui est la portion de référence de temps qu'on autorise au processus (ici 500000, soit la moitié).

On notera que si les cgroups sont activés dans le noyau, la `task_struct` dispose d'un attribut `sched_task_group` qui pointe directement sur sa `task_group` pour éviter que l'ordonnanceur perde trop de temps. Les moteurs de conteneurs font tout ce travail pour l'utilisateur en fonction des paramètres avec lesquels l'utilisateur invoque son ou ses conteneur(s).

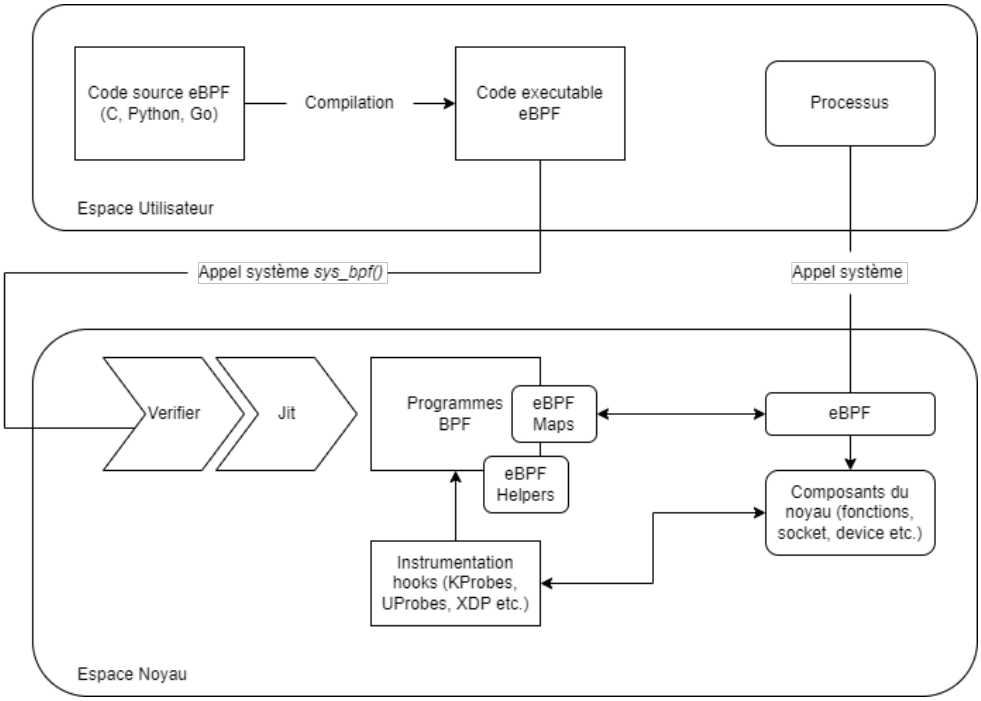


Figure 4.4 – Le framework eBPF

4.2.2 Contrôle d'accès aux ressources

Les cgroups offrent également des propriétés de contrôle d'accès via eBPF (*extended Berkeley Packet Filter*). Le *framework* eBPF permet de développer des programmes qui déclenchent des actions à la levée d'événements sur différents points d'instrumentation à la fois sur des composants du noyau (KProbes, XDP etc.) ou sur des fonctions invoquées en espace utilisateur (UProbes) comme défini dans la figure 4.4. Pour comprendre le fonctionnement du *framework*, nous allons dérouler un très court exemple s'appuyant sur le *package* bcc fourni avec Python dans le listing 4.2.

Cet exemple place un point d'instrumentation en espace utilisateur (donc via UProbe) sur la fonction `readline()` de bash. Ce point d'instrumentation est donc évalué à chaque validation de commande via l'interpréteur bash. Ce programme BPF utilise une fonction helper eBPF `bpf_get_current_uid_gid()`. Le programme d'instrumentation est ensuite lié à bash avec la fonction `attach_uretprobe()` qui s'attache au retour d'une fonction d'un exécutable donné.

Dans le noyau Linux, la structure `cgroup` contient un attribut `bpf` de type `cgroup_bpf` utilisé pour stocker les programmes eBPF. Actuellement les sous-systèmes `net_cls` et `net_prio` utilisés respectivement pour marquer des paquets avec un identifiant de catégorie et fixer une priorité utilisent eBPF. Le sous-système `devices` utilise également eBPF pour autoriser ou interdire l'accès à un périphérique. En effet, certains développeurs se basent sur ce *framework* pour isoler leur conteneur en appliquant un contrôle d'accès mandataire avec une politique définie dans un programme eBPF. RedHat le fait avec OpenShift pour isoler des conteneurs privilégiés de développement.

```
#!/usr/bin/python3

from bcc import BPF
from time import sleep

bpf_text="""
#include <linux/sched.h>

int printCmd(struct pt_regs *ctx){

        char command[16] = {};
        uid_t uid = bpf_get_current_uid_gid() & 0xffffffff;
        bpf_probe_read_user_str(&command, sizeof(command), (void *)
            ↪ PT_REGS_RC(ctx));
        bpf_trace_printk("Command from %d: %s",uid,command);
        return 0;
}
"""

b = BPF(text=bpf_text)
b.attach_uretprobe(name="/bin/bash", sym="readline", fn_name="printCmd")

while(1):
        sleep(1)
```

Listing 4.2 – Exemple eBPF en Python

4.3 Les espaces de noms (Namespaces)

Les espaces de noms, ou *namespaces* (souvent abrégés ns), constituent un modèle de multi instanciation d'objets du noyau liés au processus. L'idée derrière les espaces de noms est que chaque processus puisse disposer, s'il le souhaite, d'instances privées d'objets qu'il est amené à manipuler. Les espaces de noms sont arrivés dans les noyaux Linux vers 2006. Auparavant, les processus partageaient tous les mêmes objets. Les espaces de noms offrent donc une isolation entre les processus. Plusieurs objets sont éligibles à la privatisation :

— Les points de montage du système de fichiers ou "mount namespaces" (`mount_ns`) ;
— Les identifiants de processus : "pid namespaces" (`pid_ns`) ;
— Les noms de machine et de domaine : "uts namespaces" (`uts_ns`) ;
— Les ressources réseau (pile, protocoles, tables de routage, interfaces...) : "network namespaces" (`net_ns`) ;
— L'arborescence des cgroups : "cgroup namespaces" (`cgroup_ns`) ;
— Les identifiants de sécurité (identifiants d'utilisateur, identifiants de groupe, etc.) :

"user namespaces" (`user_ns`) ;
— Les IPC (cf. annexe A.4 ainsi que les queues de message POSIX : "ipc namespaces" (`ipc_ns`).

D'un point de vue réalisation dans le noyau, ces différents objets gravitent autour de la structure `task_struct` et sont manipulés par différents appels systèmes :

— `clone()` : création d'un processus en l'associant à de nouveaux espaces de noms ;
— `setns()` : association du processus appelant à un espace de nom existant ;
— `unshare()` : association du processus appelant à de nouveaux espaces de noms ;
— `stat()` : obtention de l'identifiant d'un espace de nom ;
— `ioctl()` : opérations diverses spécifiques aux espaces de noms.

Les deux appels systèmes sur lesquels nous allons passer un peu de temps sont `clone()` et `unshare()`. Ces deux appels partagent le même jeu de drapeaux (flags) qui superpose les namespaces disponibles :

— `cgroup_ns` (CLONE_NEWCGROUP) ;
— `ipc_ns` (CLONE_NEWIPC) ;
— `net_ns` (CLONE_NEWNET) ;
— `pid_ns` (CLONE_NEWPID) ;
— `user_ns` (CLONE_NEWUSER) ;
— `uts_ns` (CLONE_NEWUTS) ;
— `mount_ns` (CLONE_NEWNS).

Dans la section 2.2.3, nous avons vu qu'un processus invoquait l'appel système `clone()` pour créer un fils. Dans l'annexe A.7, nous avons mis en lumière que l'appel système capturé au moment de la création du processus est invoqué comme suit :

```
   clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID
| CLONE_CHILD_SETTID | SIGCHLD,
child_tidptr=0x7f1868362810) = 1120
```

Le second argument `flags` contient les paramètres passés à la création du fils. Si on y ajoute CLONE_NEWUTS et CLONE_NEWIPC on se retrouve avec la topologie d'espaces de noms présentée dans la figure 4.5.

Le processus fils va partager cinq *namespaces* sur sept avec son père. En revanche, il aura ses propres espaces de noms `ipc_ns` pour ses sémaphores, mémoires partagées etc. et `uts_ns` pour son propre nom de machine et nom de domaine. L'appel système `unshare()` permet au processus d'arrêter de d'utiliser les espaces de noms passés en paramètre pour s'en créer de nouveaux. La figure 4.6 illustre l'appel à `unshare()` avec les flags CLONE_NEWUTS et CLONE_NEWIPC.

Au niveau des structures de données dans le noyau, tout tourne autour de la structure `task_struct`. Il existe deux familles d'espaces de noms : hiérarchique ou non. Les espaces de noms non-hiérarchiques sont tous groupés dans un structure de données `nsproxy`. Chacun des champs de la structure de données `nsproxy` pointe sur la structure de données dédiée à la caractérisation de l'espace de nom soit `uts_namespace`, `mnt_namespace`, `cgroup_namespace`, `ipc_namespace` et `net`.

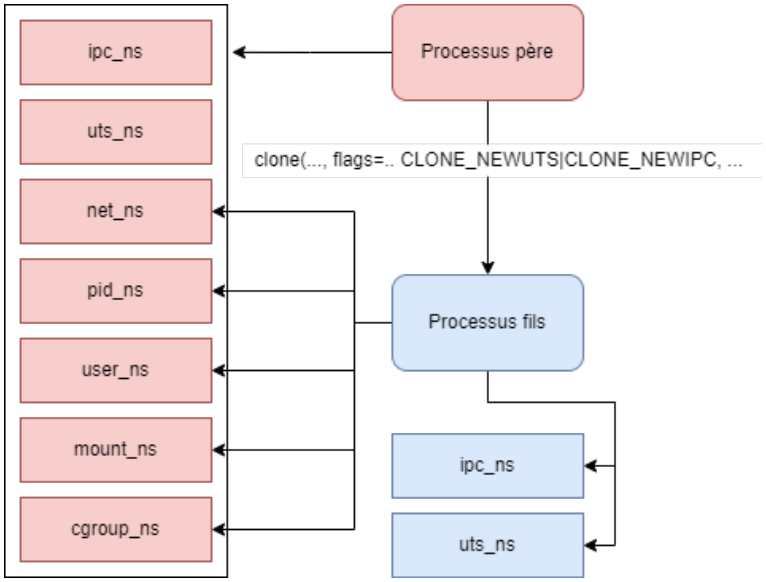


Figure 4.5 – *clone()* avec les espaces de noms

Les espaces de noms `pid_namespace` et `user_namespace` sont quant à eux hiérarchiques. En effet, un `user_ns` ou un `pid_ns` est le fils du `user_ns` ou du `pid_ns` associé à la tâche qui l'a créé. De plus, tout *namespace* est la propriété du `user_ns` associé à la tâche qui l'a créé. Enfin, une tâche à un identifiant (PID) différent dans chaque `pid_ns` et par conséquent la structure `task_struct` doit permettre d'accéder à tous ces identifiants et aux `pid_ns` associés.

L'aspect hiérarchique est caractérisé par deux champs dans la structure décrivant l'espace de nom :

1. `parent` : pointeur sur la structure de l'espace de nommage père. Quand ce champ est NULL, cela indique le sommet de la hiérarchie donc un `user_ns` ou un `pid_ns` initial (c'est-à-dire crée par `systemd` au démarrage de la machine comme dans la figure 2.3) ;

2. `level` : entier désignant le niveau dans la hiérarchie. Le premier niveau est 0 (la racine ou niveau initial). La valeur augmente séquentiellement pour chaque nouveau *namespace* fils.

Dans la figure 4.7, nous remarquons que la structure `task_struct` a un champ `thread_pid` qui pointe sur une structure `pid`. Cette structure `pid` contient, en plus de `level`, un tableau `numbers[]`. Ce tableau numbers contient une ou plusieurs structure(s) `upid` présentées dans le listing 4.3. Les membres de `upid` sont `nr` qui est le PID du processus et `ns` qui pointe sur le `pid_namespace` dans lequel `nr` a du sens. Nous avons donc une relation 1 à N entre la `task_struct` et le `pid_namespace` à travers la structure `pid`.

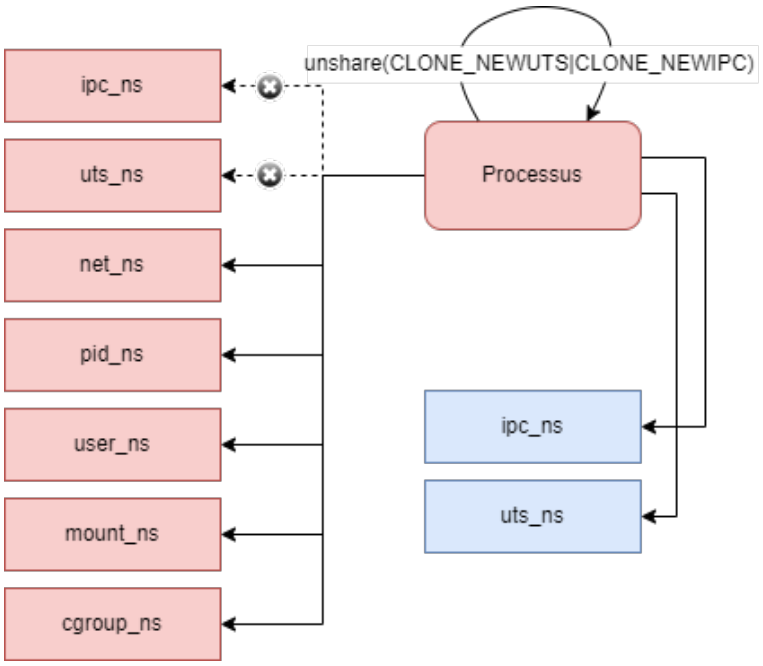


Figure 4.6 – `unshare(CLONE_NEWUTS|CLONE_NEWIPC)`

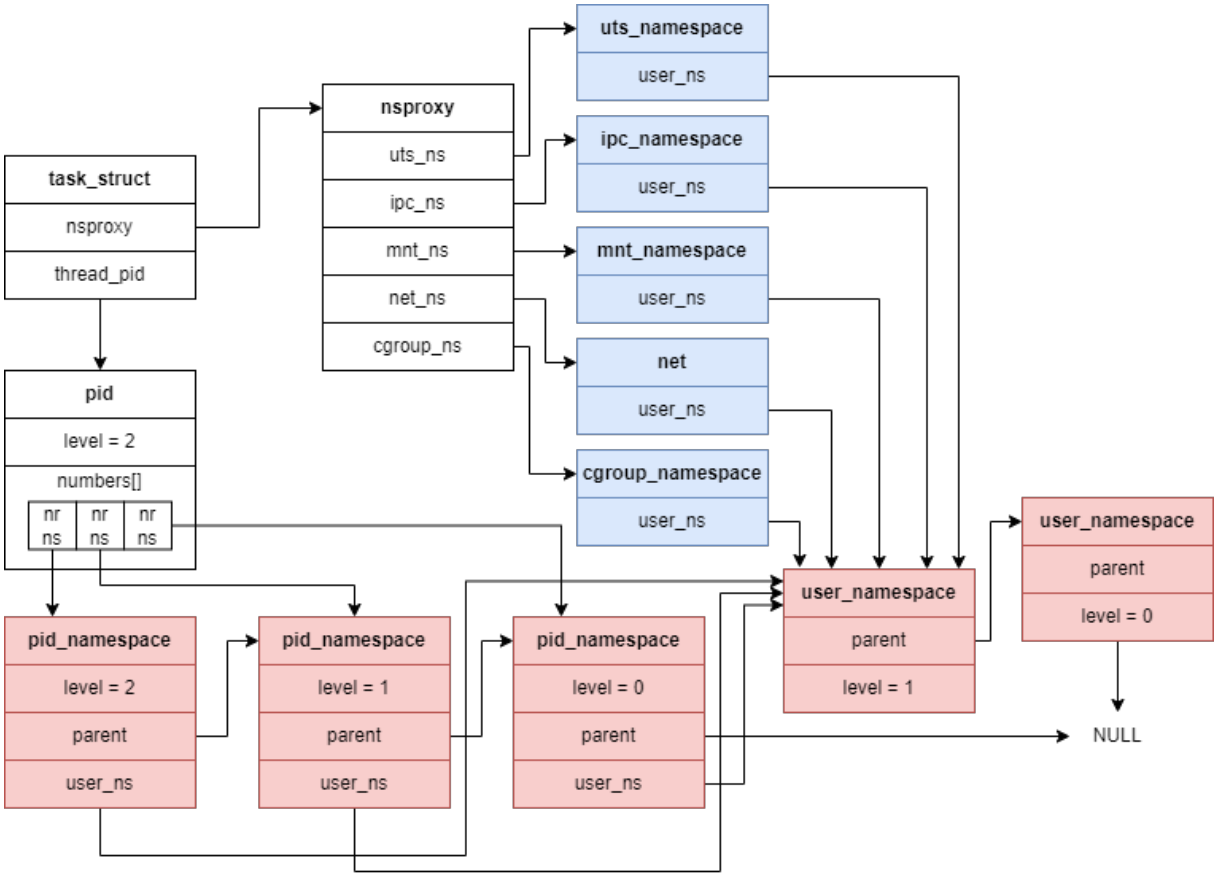


Figure 4.7 – Les espaces de noms dans le noyau

```
#Fichier : /include/linux/pid.h
struct upid {
       int nr;
       struct pid_namespace *ns;
};

struct pid
{
       refcount_t count;
       unsigned int level;
[...]
       struct upid numbers[1];
};
```

Listing 4.3 – Les structures de données PID

Nous noterons que dans les distributions Linux actuelles, tous les processus sont par défaut dans des espaces de noms dits initiaux car créés par `systemd`. Les espaces de noms complètent les cgroups pour réaliser une isolation complète du processus. Les cgroups limitent la consommation de ressources du processus tandis que les espaces de noms isolent les ressources que le noyau met à disposition des processus. Les moteurs de conteneurs s'appuient sur ces deux mécanismes en proposant des directives de haut niveau à l'utilisateur pour configurer ces deux couches.

4.4 Les moteurs de conteneurs

4.4.1 Généralités sur les conteneurs

Les conteneurs sont des systèmes d'isolation orientés applications. La discussion sur les conteneurs est consécutive à la présentation des couches basses de l'isolation comme le montre la figure 4.8 représentant l'avancement de la présentation de notre travail. Nous pouvons maintenant donner quelques cas d'utilisation typiques des conteneurs. Il s'agit de :

— Déploiement d'applications : les conteneurs permettent de créer des environnements d'exécution isolés pour les applications, ce qui facilite leur déploiement sur différentes plates-formes et infrastructures. Nous pouvons créer un conteneur qui contient tous les composants nécessaires à l'application, tels que le code, les bibliothèques, les dépendances, etc. Cela garantit que l'application fonctionnera de manière cohérente, indépendamment de l'environnement dans lequel elle est déployée ;
— Orchestration et gestion des infrastructures : les outils d'orchestration de conteneurs tels que Kubernetes permettent de gérer et de mettre à l'échelle facilement des clusters de conteneurs. Nous pouvons définir des règles de déploiement, de mise

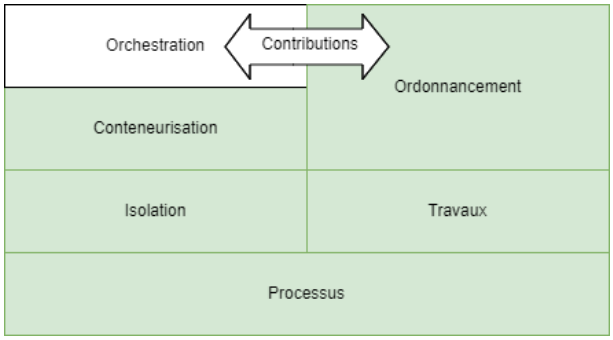


Figure 4.8 – Positionnement de l'isolation dans le sujet

à l'échelle automatique, de répartition de charge, etc., pour garantir que l'application est disponible et fonctionne de manière fiable. Cela facilite également les mises à jour et les déploiements progressifs de nouvelles fonctionnalités ;

— Intégration continue et déploiement continu (CI/CD) : les conteneurs sont utilisés dans les pipelines CI/CD pour automatiser le processus de construction, de test et de déploiement des applications. Les développeurs peuvent créer des images de conteneurs qui encapsulent leur code et les tests associés. Ces images peuvent ensuite être déployées dans des environnements de test, de pré-production et de production de manière cohérente, garantissant ainsi la fiabilité du processus de livraison logicielle ;

— Isolation et portabilité : les conteneurs offrent une isolation des ressources entre les applications et les environnements d'exécution. Cela signifie que nous pouvons exécuter plusieurs conteneurs sur le même hôte physique sans qu'ils n'interfèrent les uns avec les autres. De plus, les conteneurs sont portables (modulo le noyau du système d'exploitation hôte), ce qui signifie que nous pouvons les exécuter sur différentes infrastructures, qu'il s'agisse de machines locales, de serveurs *bare-metal*, de machines virtuelles ou de cloud public ;

— Collaboration et partage : les conteneurs facilitent la collaboration entre les membres de l'équipe de développement et d'exploitation. Nous pouvons partager facilement des images de conteneurs contenant des applications, des configurations et des dépendances spécifiques. Cela garantit que tous les membres de l'équipe utilisent le même environnement, ce qui facilite la résolution des problèmes et la collaboration ;

— Analyse des malwares : les fournisseurs de solution de sécurité type EDR / XDR proposent des services de Cloud d'analyse antivirale. Les EDR / XDR sont des antivirus modernes capables de communiquer avec les équipements de sécurité type pare-feux afin de corréler les évènements de sécurité pour déterminer l'effectivité d'une compromission. Lorsqu'un fichier est analysé par l'antivirus, il est remonté vers un Cloud géré par le fournisseurs et mis en conteneur pour subir plusieurs tests type étirage dans le temps pour voir si une charge malveillante n'est pas programmée pour exploser à un moment donné, analyse des tentatives d'interactions avec l'extérieur etc. Si un problème est détecté, une empreinte du fichier est créée et redescend vers tous les clients connectés à ce Cloud.

La figure 4.9 représente les étapes d'une instanciation de conteneur. Pour mettre son application en conteneur, le développeur doit fournir deux choses : son application et

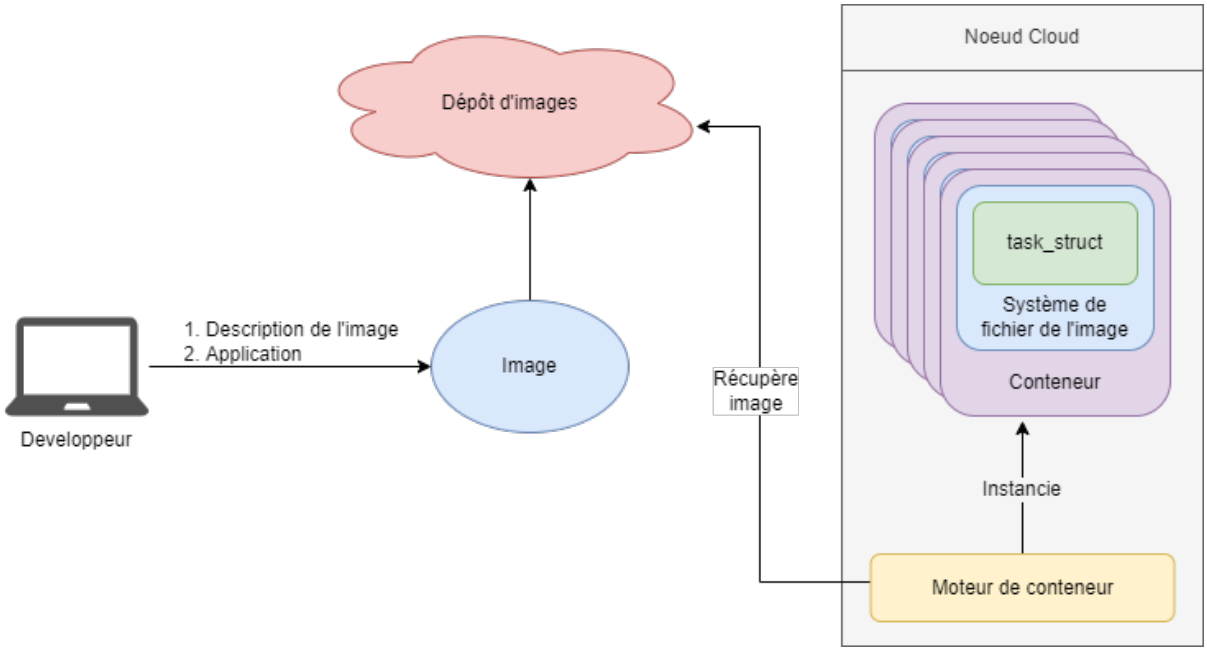


Figure 4.9 – Schéma général d'instanciation d'un conteneur

une recette de construction de l'image associée. Cette image contiendra non seulement l'exécutable de l'application mais également ses dépendances (fichiers de configuration, librairies etc.). Elle est donc totalement autosuffisante pour l'exécution, un peu à la manière d'un `chroot()` (cf. annexe A.14). Le développeur lance ensuite le processus de création de l'image. Une fois celle-ci construite en local, il est très courant de la pousser vers un dépôt public ou privé pour la publier à destination d'une certaine audience. On notera que l'image inclut l'application et ses dépendances. Elle est donc transportable d'un système à l'autre pourvu que le moteur de conteneur soit présent. Elle est également figée, ce qui en fait un environnement logiciel reproductible (modulo le matériel sur lequel cet environnement s'exécute). On considère deux types de conteneurs : jetables et persistants. Les conteneurs jetables n'ont pas de persistance au niveau du système de fichier. Lorsque l'application se termine, le conteneur est détruit et si on le relance, on se retrouve dans l'état initial de l'image. Ce type de conteneur volatile est exécuté par des moteurs tels que CRI-O et Docker. Les conteneurs persistants installent un système de base sur le système hôte à la manière d'un `chroot()`. Le programme conteneurisé s'exécute dans ce contexte. Ainsi lorsqu'il modifie des fichiers dans son contexte, ils sont modifiés de façon persistante au redémarrage de l'application conteneurisée. Il s'agit de LXC que nous discuterons dans la section 4.4.2.

La machine souhaitant exécuter le programme conteneurisé doit tout d'abord disposer d'un moteur de conteneur. Il en existe plusieurs, les plus connus sont Docker et CRI-O. Le moteur va récupérer l'image sur le dépôt et lancer l'exécutable de l'application dans le contexte de système de fichier de l'image c'est-à-dire que la racine du système de fichier pour l'exécutable conteneurisé est celle de l'image. Au lancement de l'image, le moteur va appliquer un certain nombre de restrictions au processus conteneurisé. Ces restrictions sont de deux types : limite de consommation de ressources et isolation avec le système hôte et les autres conteneurs. En résumé, on peut voir un conteneur comme un ensemble

de restrictions appliquées à la structure `task_struct` vue dans la section A.

Il existe de nombreux moteurs de conteneurs dans le paysage. Nous allons limiter la discussion à cinq moteurs qui offrent des fonctionnalités nous permettant d'illustrer les différentes architectures mais aussi d'explorer les usages de chacun. L'idée générale est que plus un moteur de conteneurs est portable d'une famille de noyau à un autre, plus il nécessite de niveaux d'indirection avant d'interagir avec les couches basses d'isolation tels que les cgroups et les espaces de noms sous Linux ou les *jobs* sous Windows.

En Juin 2015, Docker a initié un projet au sein de la fondation Linux nommé *Open Container Initiative* (OCI). L'OCI est une organisation à but non lucratif dont l'objectif est de développer des standards ouverts pour les conteneurs d'applications. Voici quelques-unes de leurs principales contributions :

— Spécifications des formats de conteneurs : L'OCI a publié les spécifications techniques pour les formats de conteneurs d'applications, notamment l'*OCI Image Format* (format d'image OCI) et l'*OCI Runtime Specification* (spécification d'exécution OCI). Ces spécifications définissent les normes pour la création, la distribution et l'exécution des conteneurs, garantissant ainsi leur portabilité et leur interopérabilité entre différents fournisseurs et plates-formes ;
— Encouragement de l'interopérabilité : L'OCI s'engage à promouvoir l'interopérabilité entre les différents acteurs de l'écosystème des conteneurs. En établissant des normes communes, l'OCI permet aux développeurs et aux utilisateurs de créer et de distribuer des conteneurs sans être liés à une plate-forme spécifique ;
— Gestion ouverte : L'OCI adopte une approche transparente et ouverte pour le développement des spécifications. Les contributeurs, qu'ils soient des individus ou des entreprises, peuvent participer activement aux discussions et aux travaux de l'OCI pour faire évoluer les standards des conteneurs ;
— Certification OCI : L'OCI propose un programme de certification qui permet aux fournisseurs de logiciels et de services de démontrer leur conformité aux spécifications OCI. Cela aide les utilisateurs à identifier les produits et les solutions qui respectent les normes OCI et qui offrent une compatibilité garantie ;
— Collaboration avec d'autres organisations : L'OCI collabore avec d'autres organismes et projets, tels que la *Cloud Native Computing Foundation* (CNCF), pour favoriser l'adoption des standards OCI dans l'écosystème des conteneurs et encourager l'innovation continue.

En résumé, ce projet travaille sur deux axes : la spécification de l'environnement d'exécution des conteneurs (*runtime-spec*) et la normalisation de la définition des images (*image-spec*). En préambule de la présentation de Docker, nous allons brièvement aborder les LXC (*LinuX Containers*) apparus en 2008. LXC a été le premier moteur de conteneurs à interagir directement avec les mécanismes disponibles dans le noyau Linux sans avoir à le patcher (contrairement par exemple a OpenVZ apparu 2005). Nous allons ensuite retracer l'histoire de Docker car elle est entremêlée avec la standardisation et la modularisation des briques de conteneurisation. Nous aborderons également CRI-O qui est maintenant le moteur de conteneur par défaut de Kubernetes. Enfin, nous mentionnerons les moteurs gVisor et Katacontainers qui sont deux systèmes durcis s'appuyant respectivement sur `ptrace()` et KVM.

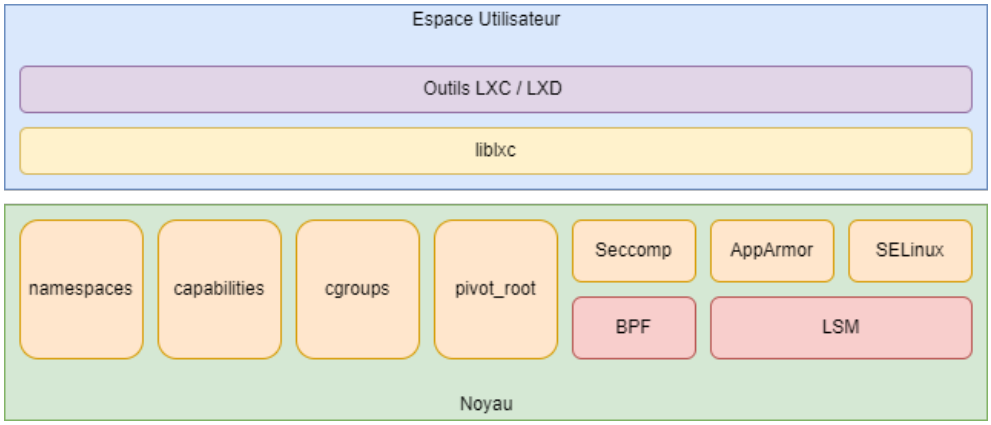


Figure 4.10 – Architecture de LXC

Enfin, peu importe le moteur de conteneur retenu, la bonne pratique et de n'avoir qu'un processus et ses fils dans un conteneur. C'est la nature du processus conteneurisé qui va conditionner le choix du moteur. Si on conteneurise `systemd` (et donc par rebond tous les services d'une machine car `systemd` est le service de démarrage) alors on s'oriente vers des conteneurs dit lourds (LXC). Si on conteneurise un service unitaire type Nginx, on parle alors de conteneur légers et il s'agit plutôt de Docker. Évidemment, tout fonctionnerait à la place de tout. Cependant, les outils constituant chaque moteur sont plus ou moins adaptés à tel ou tel usage.

4.4.2 LXC

LXC (*LinuX Containers*) est un moteur de conteneur lancé en 2008 qui s'appuie directement sur les mécanismes disponibles dans le noyau pour isoler des processus. Son architecture est présentée dans la figure 4.10. LXC fournit à la fois une librairie servant d'interface d'utilisation de ces mécanismes avec le noyau (liblxc) mais également des outils de haut niveau pour les utilisateurs qui sont un ensemble de commandes préfixés par lxc-*.

La `liblxc` s'appuie sur des mécanismes de contrôle d'accès mandataire (MAC cf. A.13), notamment SELinux et AppArmor, pour créer des politiques d'isolation des conteneurs. Elle peut également tirer parti de `Seccomp` pour limiter les appels systèmes. `Seccomp` s'appuie sur le framework BPF (cf. section I.4.2.1). Si le conteneur a besoin de privilèges particuliers, `liblxc` peut s'interfacer avec les capacités (cf. section 4.2.2). Évidemment, `liblxc` interagit également avec le couple cgroups / espaces de noms (cf. section 4.2 et 4.3). Enfin, le système de fichier du conteneur LXC est isolé grâce à `pivot_root()` qui est un appel système comparable à `chroot()` au niveau fonctionnel. La différence majeure est que `pivot_root()` requiert la nouvelle racine de notre processus soit sur une partition différentes, ce qui mitige les risques inhérents à `chroot()`.

Au niveau de l'usage LXC est assez semblable à une machine virtuelle (attention, nous parlons bien de l'usage, pas des mécanismes sous-jacents). Il n'est pas vraiment utilisé par des développeurs mais plus par des administrateurs pour créer des conteneurs de systèmes Linux complets beaucoup moins volatiles que les conteneurs type Docker.

4.4.3 Docker

Docker est un moteur de conteneurs publié en open-source en mars 2013. La conception originale de Docker était totalement monolithique. Il était constitué d'un unique service `dockerd` qui interagissait directement avec la couche librairie LXC (`liblxc`) pour créer ses conteneurs. Le processus `dockerd` s'occupait à la fois du cycle de vie des conteneurs, de la gestion des images, des volumes, du réseau etc. En conséquence, la moindre mise à jour de `dockerd` entraînait l'arrêt des conteneurs associés. Le pilotage du service `dockerd` se fait par les outils utilisateur Docker pour instancier un conteneur de façon unitaire ou `docker-compose` si on doit en instancier plusieurs ayant des interactions entre eux (par exemple un couple Apache / MySQL).

En juillet 2015, la première version de l'environnement runC (qui se nommait libcontainer à l'époque) proposant une interface d'utilisation des couches basses en vue d'exécuter le conteneur vit le jour. Cette interface répond au nom de CRI pour *Container Runtime Interface*. En pratique, c'est un outil en ligne de commande qui va d'abord préparer l'environnement du conteneur, c'est-à-dire créer les espaces de noms, les cgroups et éventuellement y adjoindre des contraintes de quota d'utilisation des ressources. Une fois l'environnement préparé, il réalise la séquence `fork()` $->$ `exec()` pour exécuter le processus conteneurisé dans le contexte de son image.

En décembre 2015, Docker cède `containerd` à la CNCF (*Cloud Native Computing Foundation*). Ce sous ensemble du code du Docker monolithique original est un service chargé de gérer le cycle de vie du conteneur. Il va non seulement créer, démarrer, arrêter ou détruire le conteneur mais aussi gérer le transfert des images, l'attachement à des stockages divers et la configuration du réseau. De par ses origines, containerd est conforme aux spécifications de l'OCI. On notera que `containerd` est disponible pour Linux et Windows.

Un autre intérêt de `containerd` est qu'il vient avec un service `containerd-shim` qui va être le père du processus conteneurisé. Il va gérer le retour du conteneur, les entrées sorties types tty, entrées / sorties standards etc. ainsi que le changement de père. Le processus `containerd-shim` est directement rattaché à `systemd` ce qui fait qu'il est possible de mettre à jour Docker et `containerd` sans interrompre les conteneurs en exécution. C'est ce qu'on appelle les conteneurs sans démons ou *daemonless containers*.

Le service `containerd-shim` utilise la CRI (*Common Runtime Interface*) pour demander à runc d'instancier le conteneur de façon effective. La CRI agit comme un pont entre le système de gestion de conteneurs et le *runtime* de conteneur. Elle fournit une interface standardisée permettant au système de gestion de conteneurs de communiquer avec le *runtime* de conteneur (runc). Ainsi, le système de gestion de conteneurs n'a pas besoin de connaître les détails spécifiques du *runtime* de conteneur utilisé. La CRI facilite également l'intégration de nouveaux *runtimes* de conteneurs dans des systèmes de gestion de conteneurs existants, car il suffit de développer un adaptateur (CRI Shim) qui implémente l'interface CRI pour que le *runtime* de conteneur puisse être utilisé avec le système de gestion de conteneurs compatible CRI. C'est ainsi qu'on peut multiplier les *runtime* tels que ceux que nous verrons dans les sections 4.4.4 et 4.4.5.

Au final, c'est donc runc (le *runtime*) qui va interagir avec tous les mécanismes de bas

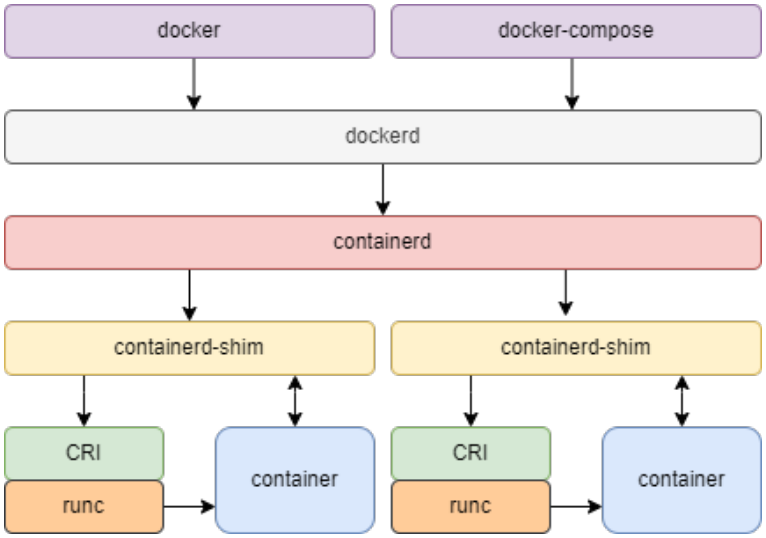


Figure 4.11 – Vue logique de Docker

```
systemd─┬─...
        ├─containerd───6*[{containerd}]
        ├─containerd-shim─┬─bash
        │                 └─9*[{containerd-shim}]
        ├─dockerd───18*[{dockerd}]
        └─...
```

Figure 4.12 – Parenté des processus Docker

niveau pour instancier le conteneur. Les figures 4.11 et 4.12 présentent respectivement une vision logique et physique des composants de Docker. En conclusion sur Docker, nous voyons que l'architecture à évoluée pour poser les bases de structuration des moteurs actuels.

4.4.4 CRI-O

CRI-O est un équivalent de `containerd`. Cependant il ne s'intègre pas avec les commandes de haut niveau Docker. En revanche, il est capable de récupérer les images créées avec Docker et disponibles sur le Docker Hub (ou tout autre dépôt). Le projet est porté par RedHat, IBM et OpenSuse. Il va également gérer le réseau, le stockage et le système de fichier des conteneurs. CRI-O est arrivé en 2015. A cette époque, Docker n'était pas encore pleinement modulaire et Google souhaitait un moteur de conteneur amputé des fonctions inutiles faisant doublon avec son orchestrateur Kubernetes (cf. section 5.1). Cependant, un peu plus tôt la même année, Docker décida d'ajouter son propre orchestrateur Swarm à son produit.

CRI-O est donc très lié à Kubernetes contrairement à Docker qui fournit un service certes plus conséquent en termes de codes mais autonome au niveau des fonctionnalités. La raison est que Docker intègre tous les composants de gestion du cycle de vie des conteneurs (gestion des images, instanciation et supervision des conteneurs, orchestration avec swarm etc.) tandis que CRI-O ne fait qu'instancier un conteneur à partir d'une image. CRI-O

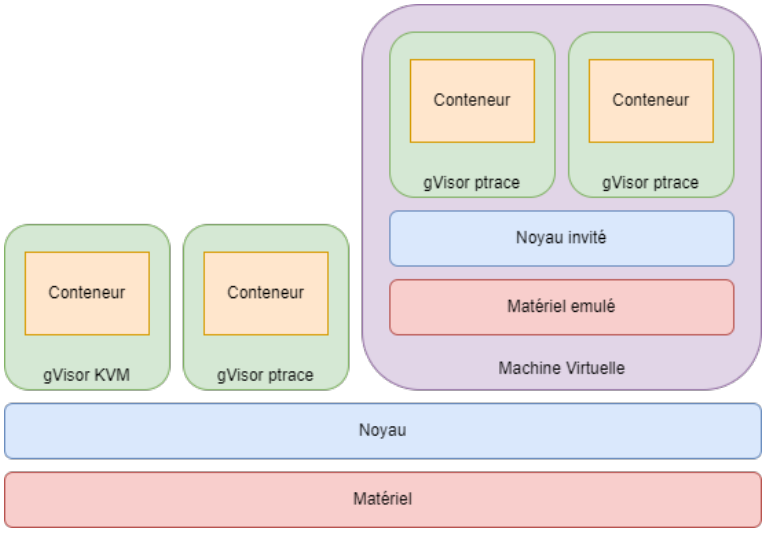


Figure 4.13 – Plateforme d'interception

est maintenant le moteur par défaut de Kubernetes.

4.4.5 gVisor

Dans la continuité de la discussion sur CRI-O, Google propose une seconde alternative à runC nommé gVisor. Cette alternative est un système de bac à sable (*sandbox*) pour conteneurs. L'idée est que gVisor va instancier les conteneurs (comme le runC standard) mais il va également instrumenter ses interactions. Comme présenté dans la figure 4.13, gVisor peut s'appuyer sur l'appel système `ptrace()` ou KVM pour intercepter les interactions et les autoriser ou non.

L'appel système `ptrace()` est invoqué par un processus en cours d'exécution pour surveiller et contrôler un autre processus. En d'autres termes, `ptrace()` permet à un processus de prendre le contrôle d'un autre processus et de l'observer ou de modifier son comportement. Le processus appelant de `ptrace()` peut réaliser les actions suivante sur le processus supervisé :

— Lire et écrire dans la mémoire du processus cible ;
— Exécuter une seule instruction dans le contexte du processus cible ;
— Obtenir des informations sur l'état du processus cible, comme les registres de la CPU et la pile d'appels ;
— Mettre le processus cible en pause ou le reprendre après une pause.

L'appel système `ptrace()` est souvent utilisé pour le débogage de programmes ainsi que l'analyse de logiciels malveillants. Cependant, il peut également être utilisé pour d'autres tâches comme la surveillance des performances système et surtout, ce qui nous intéresse plus particulièrement dans notre contexte, l'application de politique de sécurité sur le processus cible.

Le second mécanisme d'instrumentation est de s'appuyer sur KVM (*Kernel-based Vir-*

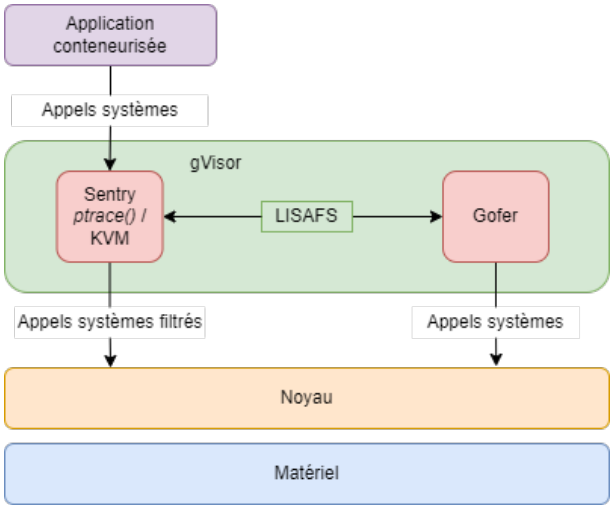


Figure 4.14 – Services de gVisor

tual Machine). Pour bien comprendre comment KVM intervient, il faut aborder brièvement les services qui composent gVisor. La figure 4.14 représente l'architecture de gVisor. On distingue deux services : Sentry et Gofer. Le composant réalisant l'interception est Sentry. C'est le composant principal de gVisor. Dans le paragraphe précédent, nous avons discuté de l'interception d'interaction via `ptrace()`. Sentry peut également s'appuyer sur KVM pour assumer le rôle d'hyperviseur (cf. section suivante 4.1) sans toutefois présenter de matériel émulé au processus conteneurisé.

L'objectif est de tirer partie des extensions de virtualisation des processeurs pour instrumenter plus efficacement les processus conteneurisés. En effet, l'instrumentation strictement logicielle par `ptrace()` implique de nombreux changements de contexte (cf. section A.2). Cependant, KVM et `ptrace()` se complètent car :

— Si la machine ne supporte pas les extensions de virtualisation, il faut utiliser `ptrace()` ;
— Si les conteneurs sont dans des machines virtuelles alors `ptrace()` donne de meilleurs résultats que KVM en *nesting* (c'est-à-dire de l'hypervision à l'intérieur d'une machine virtuelle).

Une fois les événements capturés, Sentry peut les filtrer. On notera que Sentry reprend un peu le fonctionnement de UML (*User Mode Linux*) [48] qui était un système permettant d'avoir des machines virtuelles totalement en espace utilisateur en interceptant les appels systèmes via `ptrace()` et en les jouant sur un noyau en espace utilisateur.

Sentry est accompagné d'un second service nommé Gofer. Dans un souci de séparation de privilèges, Sentry n'accède pas au système de fichier. Cette tâche incombe à Gofer qui agit comme un *proxy* vers le système de fichier pour Sentry. Les deux processus communiquent via un *socket* local ou une mémoire partagée par le protocole LISAFS (*LInux SAndbox File System protocol*) qui a remplacé 9P (Plan 9) utilisé jusqu'en janvier 2023.

CHAPITRE 5

ÉLÉMENTS DU CLOUD : ORCHESTRATION AVEC KUBERNETES

Dans le chapitre 4, nous avons vu comment les conteneurs sont gérés de façon unitaire, c'est-à-dire comment un processus est isolé via différents mécanismes. Cependant, une application est souvent composée de N conteneurs. Par exemple, un site marchand sera composé d'un frontal *proxy* inversé, d'un serveur web, d'une base de données etc. Parmi ces services, certains ont des données persistantes (serveur de base de données) d'autres non (frontal *proxy* inversé), certains ont besoin d'être exposés à l'extérieur du réseau tandis que d'autres non.

Les conteneurs n'ont pas tous les mêmes besoins. De plus, nous rappelons que le Cloud est surtout affaire d'élasticité. L'infrastructure conteneurisée doit pouvoir croître et décroître pour passer à l'échelle de la demande. Dans l'absolu, il y a deux façons de faire croître des ressources : 1) en augmentant leur capacité d'un point de vue unitaire, 2) en les multipliant. Dans le monde des processus conteneurisés, cela se traduit par : 1) augmenter le plafond du quota des ressources utilisables par le processus (plus de mémoire, plus de CPU etc.), 2) créer de nouveaux conteneurs. On parle respectivement de passage à l'échelle vertical (*vertical scaling*) pour 1) et passage à l'échelle horizontal (*horizontal scaling*) pour 2). L'orchestrateur est responsable de la cohérence de la répartition et de la gestion des conteneurs composants les différentes applications. C'est également la dernière notion à discuter sur les pré-requis de nos contributions comme le montre la figure 5.1.

La philosophie des orchestrateurs est de permettre à l'utilisateur de décrire un déploiement. Un déploiement est une description d'un état cible. Par exemple, l'utilisateur peut formuler un souhait tel que « je veux déployer un conteneur Apache qui doit être redémarré en cas de panne, un serveur Nginx en *reverse proxy* et un serveur MySQL avec un volume persistant ». L'orchestrateur va placer les conteneurs au mieux sur les nœuds disponibles sur l'infrastructure de Cloud. Nous distinguerons deux grandes familles d'orchestrateurs en fonction de leur étendue. Il existe des orchestrateurs tels que Docker-Compose ou Minikube qui orchestrent des machines sur un seul nœud. Leur étendue est dite locale. On

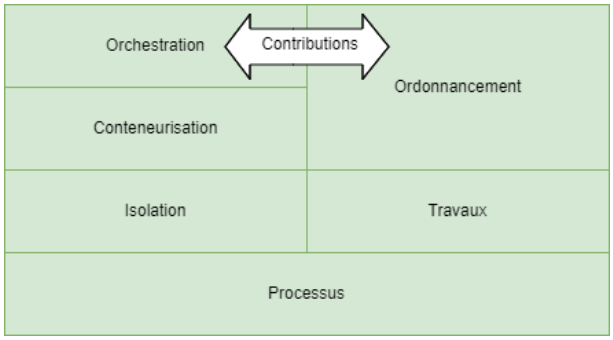


Figure 5.1 – Organisation et dépendances des notions

trouve également des orchestrateurs distribués tels que Kubernetes, Docker Swarm ou Nomad qui répartissent les conteneurs sur plusieurs nœuds. Notre contribution s'appuie sur l'orchestrateur distribué Kubernetes (souvent abrégé K8s). Nous allons identifier les points saillants de Kubernetes pertinents avec notre approche. Nous évoquerons également quelques unes de ces alternatives, à savoir Docker Swarm et Nomad.

5.1 Architecture générale de Kubernetes

Un cluster Kubernetes est composé d'un (ou plusieurs) nœud(s) de contrôle, aussi appelés « *control plane* », et de nœuds de travail accueillants les conteneurs. Dans la suite de ce chapitre, lorsque nous parlerons de nœuds, ce seront des nœuds de travail. Lorsqu'il s'agira du nœud de contrôle nous l'expliciterons. Sur la figure 5.2, nous retrouvons les deux composants susmentionnés. Commençons par le nœud de contrôle qui héberge cinq composants :

— Le service `etcd` qui est une base de données clés / valeurs (*hashtable*) distribuée sur tout le cluster. On y retrouve les informations sur les nœuds, les configurations des déploiements, les secrets etc. Ce service peut être multi-instancié pour garantir sa disponibilité ;
— L'ordonnanceur qui va placer au mieux les conteneurs sur les noeuds du cluster Kubernetes en fonction des contraintes formulées par l'utilisateur ;
— Le gestionnaire de contrôleurs qui va piloter les déploiements des conteneurs sur les nœuds. Nous y reviendrons dans la section 5.3 consacrée au sujet car c'est un composant critique pour notre contribution ;
— Le contrôleur Cloud qui gère la connexion avec des fournisseurs externes en cas d'architectures hybrides ;
— Le serveur d'API (API Server) qui est le point d'entrée pour communiquer avec le cluster Kubernetes. Il va interagir avec tous les composants susmentionnés pour satisfaire les requêtes qui lui arrivent. Ses correspondants peuvent être les nœuds pour les affaires de gestion des conteneurs ou les utilisateurs (via la commande `kubectl` par exemple) pour solliciter des déploiement ou en récupérer les informations.

Sur les nœuds de travail, nous trouvons essentiellement deux services : Kubelet et K-Proxy (Kube-Proxy). K-Proxy s'occupe de la couche réseau dédiée aux conteneurs.

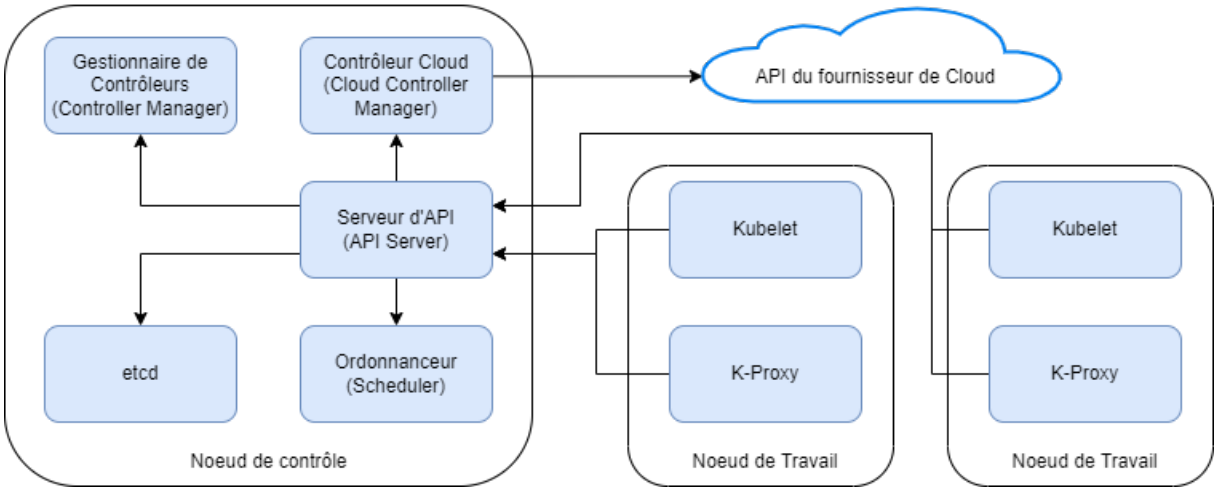


Figure 5.2 – Architecture générale de Kubernetes

Il va interagir avec le système de filtrage de paquet pour créer des règles adaptées à chaque conteneur pour ses interactions aussi bien avec les autres conteneurs du cluster qu'avec l'extérieur. Kubelet quant à lui s'occupe de dialoguer avec le moteur de conteneurs pour gérer ce qu'on appelle des Pods et qui sont des ensembles de conteneurs. En effet, comme nous allons le voir dans la section 5.2, Kubernetes ne manipule pas directement les conteneurs mais des groupes de conteneurs appelés Pods.

5.2 Les Pods

Les Pods sont placés sur les nœuds par l'orchestrateur. En conséquence, tous les conteneurs d'un Pod sont localisés physiquement sur le même nœud. C'est une manière élégante de résoudre le problème des applications composés de plusieurs processus systèmes. En effet, la règle d'or dans la création des conteneurs est d'isoler seulement un processus et ses fils. Le principe du Pod colle à cette réalité dans la mesure où l'on peut créer plusieurs conteneurs qui seront localisés sur le même nœud donc susceptibles de partager des volumes locaux (fichiers de configuration, bases de données etc.), de communiquer via des IPC ou encore s'envoyer des signaux en fonction des espaces de noms qu'ils vont partager ou non et des capacités qu'ils vont posséder.

Le second intérêt du groupement des conteneurs en Pods est qu'il est possible de les organiser en séquences avec, par exemple, un conteneur chargé de préparer le terrain avant de démarrer les suivants. Ces conteneurs d'initialisation (initContainers) sont chargés de vérifier certaines conditions (services réseaux distants accessibles, disponibilité d'une ressource en particulier, etc.) avant le lancement du conteneur principal.

La figure 5.3 présente le cycle de vie d'un Pod. L'utilisateur doit tout d'abord décrire son Pod. Le listing 5.1 présente une définition de Pod très simple. Ce fichier est divisé en deux sections séparées par « — ». La première section décrit le Pod tandis que la seconde le réseau associé. Nous commencerons par le Pod. Le champ `kind` donne le type de déploiement à effectuer. Ici il s'agit ici du déploiement d'un Pod unitaire simple. Nous

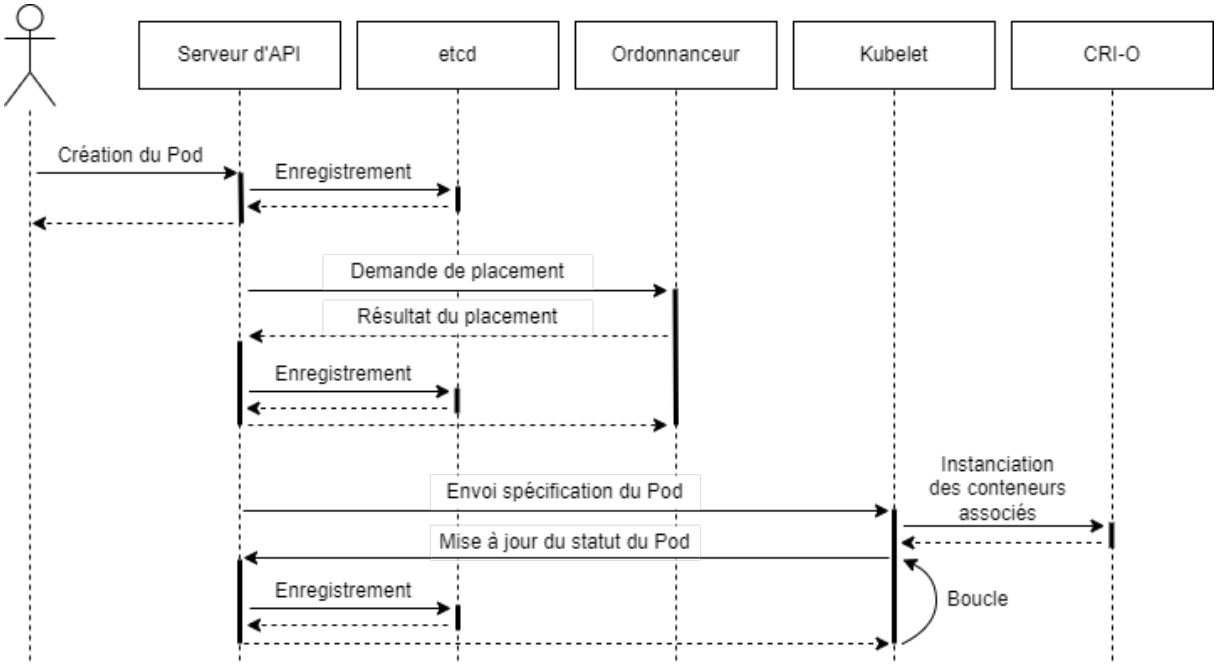


Figure 5.3 – Acteurs du cycle de vie d'un Pod

voyons que la section `spec` contient une sous-section containers pouvant accueillir de multiples couples `name` / `image` qui donnent les différentes images des conteneurs constituant le Pod. Ici il s'agit d'un Pod déployant un seul conteneur isolant un service Nginx. Une fois instanciés, les conteneurs de ce Pod pourront communiquer avec d'autres conteneurs de Pod différents (sauf application d'une politique de sécurité particulière). Cette précision est importante pour la suite de notre travail. Le serveur d'API reçoit cette définition et l'inscrit dans la configuration globale du cluster. Il effectue ensuite une demande de placement à l'ordonnanceur. Une fois la décision rendue, le serveur d'API transmet au service Kubelet du nœud. Enfin, le Kubelet va s'adresser au moteur de conteneurs pour instancier tous les conteneurs définis dans la description de l'utilisateur.

La seconde section est un objet de type service `kind: Service`. La définition que nous avons donné est la plus simple, les Pods (associés au service via le `selector` défini qui va coïncider avec les labels des metadata du Pod) vont s'enregistrer dans le DNS interne à Kubernetes avec le nom pleinement qualifié webserver.`<namespace>.svc.cluster.local`. Les services permettent de mettre en place des topologies assez complexes au niveau des Pods cependant les éléments donnés ici suffisent pour comprendre notre travail.

Les Pods sont rarement déployés de façon unitaire. Ici le champ `kind` prend la valeur Pod ce qui signifie que la ressource à contrôler est un Pod unitaire et lié à aucun autre en particulier. Ce champ `kind` conditionne l'application de ce que l'on appelle un contrôleur au(x) Pod(s) déployé(s). Le contrôleur est en quelque sorte le thermostat de la ressource déployée, nous lui définissons un état cible et il fait tout ce qui est en son pouvoir pour le maintenir.

```
apiVersion: v1
kind: Pod
metadata:
  name: webserver
  labels:
        name: webserver
        net: headless
spec:
  containers:
  - name: webserver
    image: nginx:latest
apiVersion: v1
---
kind: Service
metadata:
  name: netsimple
spec:
  selector:
    net: headless
  clusterIP: None
```

Listing 5.1 – Définition d'un Pod simple

5.3 Les contrôleurs

Kubernetes embarque un gestionnaire de contrôleur. Il va nous permettre de définir différents types de contrôle de Pods. Dans la section précédente, nous avons vu le type Pod unitaire ainsi que service associé mais il en existe d'autres, nécessaires, notamment `replicaset`, `deployment` et `statefulset`, à discuter pour la suite du travail. Ces trois contrôleurs partagent la propriété de gérer une flotte de Pods aux spécifications identiques en leur offrant certaines propriétés, notamment le passage à l'échelle. Les Pods gérés sont appelés `replicas` et leur nombre peut s'accroître ou diminuer de façon dynamique assurant un passage à l'échelle horizontal à l'application déployée.

5.3.1 ReplicaSet

Le `ReplicaSet` est le contrôleur le plus simple lorsqu'il s'agit de maintenir un ensemble stable de Pods à un moment donné pour assurer la disponibilité d'un service. Le `ReplicaSet` fonctionne sur le principe de l'adoption des Pods. En effet, le `ReplicaSet` va définir un sélecteur qui sera par la suite référencé par les spécifications des Pods. Lorsque cela coïncide, le(s) Pod(s) est / sont adopté(s) par le `ReplicaSet`.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: webservers
  labels:
     tier: rs-webservers
spec:
  replicas: 3
  selector:
     matchLabels:
        tier: rs-webservers
  template:
        metadata:
           labels:
           tier: rs-webservers
        spec:
        containers:
           - name: webserver
             image: nginx:latest
```

Listing 5.2 – ReplicaSet simple

Le listing 5.2 reprend le Pod défini dans le listing 5.1 pour l'intégrer dans un `ReplicaSet` (`kind: ReplicaSet`) nommé webservers et qui va créer trois (`replicas: 3`) Pods répondant à la spécification coïncidant avec le label `tier: rs-webservers`. A partir de ce moment, le gestionnaire de contrôleurs va s'efforcer de maintenir trois instances actives des Pods du `ReplicaSet` qui coïncident avec les sélecteurs définis. Le schéma de nommage est <nom du Pod>-<hash>. Les noms des Pods ne sont donc pas prévisibles.

La limite principale des `ReplicaSet` est que si vous le détruisez en laissant les Pods actifs (par une opération d'orphelinisation) et que vous en recréez un avec des spécifications différentes, alors ces nouvelles configurations ne sont pas répercutées sur les Pods existants. Les `ReplicaSet` ne supportent pas les « *Rolling Updates* » qui détruisent les Pods avec la configuration antérieure pour les recréer avec la nouvelle sans interruption de service. Ces mises à jour sont supportées par les contrôleurs `Deployment` et `StatefulSet`. Ils sont très rarement employés directement mais plutôt de façon indirecte par des sur-couches que nous allons maintenant examinées.

5.3.2 Deployment

Le contrôleur `Deployment` est une surcouche du `ReplicaSet` qui va gérer les mises à jour de configuration en mode « *Rolling Updates* ». De plus, chaque changement de configuration est versionné ce qui permet de repasser sur une ancienne version en cas de dysfonctionnement de la mise à jour. Le listing 5.3 présente l'instanciation du `ReplicaSet`

sous forme de `Deployment`. Il y a très peu de différences de configuration. Les replicas sont supprimés en mode LIFO (*Last In First Out*). En conséquence, lors d'une réduction du nombre de `replica`, il est impossible de cibler précisément tel ou tel Pod du Deployment. Comme dans les `ReplicaSet`, le schéma de nommage est <nom du Pod>-<hash>. Les noms des Pods ne sont donc pas prévisibles.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webservers
  labels:
        app: webserver
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webserver
  template:
    metadata:
      labels:
        app: webserver
    spec:
        containers:
        - name: nginx
          image: nginx:latest
```

Listing 5.3 – Deployment simple

Enfin, les états de certains attributs du Pod ne sont pas stables à travers les créations et suppressions de Pods replicas. Nous avons déjà évoqué la génération du nom mais il s'agit également des adresses IP, attachements à des volumes etc. Le dernier contrôleur `StatefulSet` ajoute de la stabilité dans les attributs des Pods gérés par le contrôleur.

5.3.3 StatefulSet

Le `StatefulSet` crée les Pods avec un identifiant unique persistant au ré-ordonnancement. En cas de suppression et de recréation du Pod ses attributs sont préservés. En conséquence, les Pods d'un `StatefulSet` ne sont pas interchangeables. A leur re-création, ils récupèrent la même IP, les mêmes volumes etc. Le schéma de nommage des Pods est différent des `ReplicaSet` et des `Deployment`, leur nom est suffixé par un numéro incrémenté par ordre de création. Le nom des Pods gérés par un `StatefulSet` est donc prédictible. Tout comme les `Deployment` les « *rolling updates* » sont également gérés. Le listing 5.4 définit un `StatefulSet` simple pour déployer trois Pods Nginx. La définition n'est pas tellement différente d'un `Deployment`. La différence notable est la présence d'un service en

mode « `headless` » (`clusterIP: None`) c'est-à-dire sans adresse d'équilibrage de charge. C'est nécessaire car les adresses doivent rester stables dans le temps. Enfin, les Pods sont supprimés et recréés en mode LIFO comme les `Deployment`.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx
  serviceName: "nginx"
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
```

Listing 5.4 – StatefulSet simple

5.4 Alternatives

Il existe deux grandes alternatives à Kubernetes s'agissant de l'orchestration de conteneurs : Docker Swarm et Nomad. Au passage, notons que ces trois orchestrateurs de conteneurs sont open-source. Bien qu'ils partagent tous un objectif similaire, à savoir distribuer des conteneurs sur une infrastructure, chacun a ses propres avantages et inconvénients concernant la facilité d'utilisation, la scalabilité, la flexibilité et la complexité.

Nous allons commencer par synthétiser les points clés de l'orchestrateur de référence Kubernetes. Il est largement adopté pour sa flexibilité et sa capacité à gérer des clusters de grande taille. Kubernetes peut gérer des milliers de conteneurs sur plusieurs nœuds et gérer la haute disponibilité, le passage à l'échelle automatique et des couches de communication réseau relativement complexes. Les forces de Kubernetes sont donc :

— La haute disponibilité : Kubernetes est conçu pour garantir une disponibilité maximale en cas de défaillance de nœuds ou de conteneurs (sous réserve de configurer son déploiement de façon adéquate) ;
— L'interopérabilité : Kubernetes prend en charge une grande variété de moteurs de conteneurs, notamment Docker, rkt et CRI-O. Il offre également une grande flexibilité dans la configuration des ressources en s'intégrant avec les principaux systèmes de stockage, *backend* réseaux etc. Du point de vue de la sécurité, il intègre différents modèles de contrôle d'accès comme RBAC (*Role Based Access Control*) et peut s'appuyer sur plusieurs composants de sécurité comme seccomp, SELinux etc. ;
— L'extensibilité : Kubernetes possède un écosystème riche de *plugins* et de modules complémentaires pour étendre ses fonctionnalités.

En revanche, un orchestrateur aussi complet que Kubernetes vient avec son lot de difficultés comme :

— Sa complexité : Kubernetes peut être assez complexe à configurer et à gérer pour les débutants ;
— Surdimensionnement : Kubernetes est souvent considéré comme étant surdimensionné pour les petites applications.

Face au mastodonte (dans tous les sens du terme) Kubernetes, Docker Swarm est un orchestrateur de conteneurs intégré à Docker Engine. Il est beaucoup plus facile à configurer et à utiliser. Il offre également des fonctionnalités similaires comme le passage à l'échelle automatique et la gestion du réseau. L'intégration est native à Docker Engine ce qui facilite la gestion des conteneurs et des images Docker. Cependant, Docker Swarm peut gérer moins de conteneurs, le passage à l'échelle comprend beaucoup moins de métriques et surtout elles ne sont pas configurables (au contraire du service metrics de Kubernetes). De plus, l'écosystème est plus pauvre que Kubernetes. Enfin, Docker Swarm ne s'interface qu'avec `containerd`.

L'autre alternative est Nomad qui est développé par la société Hashicorp. En terme de complexité de configuration et de quantité de conteneurs maximale gérée, Nomad est entre Kubernetes et Docker Swarm. Sa force est de savoir gérer des *workloads* hétérogènes comme le déploiement de conteneurs, le lancement de travaux simples mais surtout le provisionnement de serveurs bare-metal.

Pour conclure ce comparatif, nous mentionnerons également deux orchestrateurs très légers visant les déploiement locaux (c'est-à-dire cantonnés à une seule machine) : Docker Compose et Minikube. Docker Compose fait partie de la suite Docker et permet de faire des déploiements locaux de conteneurs. L'orchestration est assez légère, nous pourrions plus parler de coordination. En effet, la notion de définition d'un état cible de déploiement n'est pas présente dans Docker Compose. Il faut individuellement fixer des politiques de

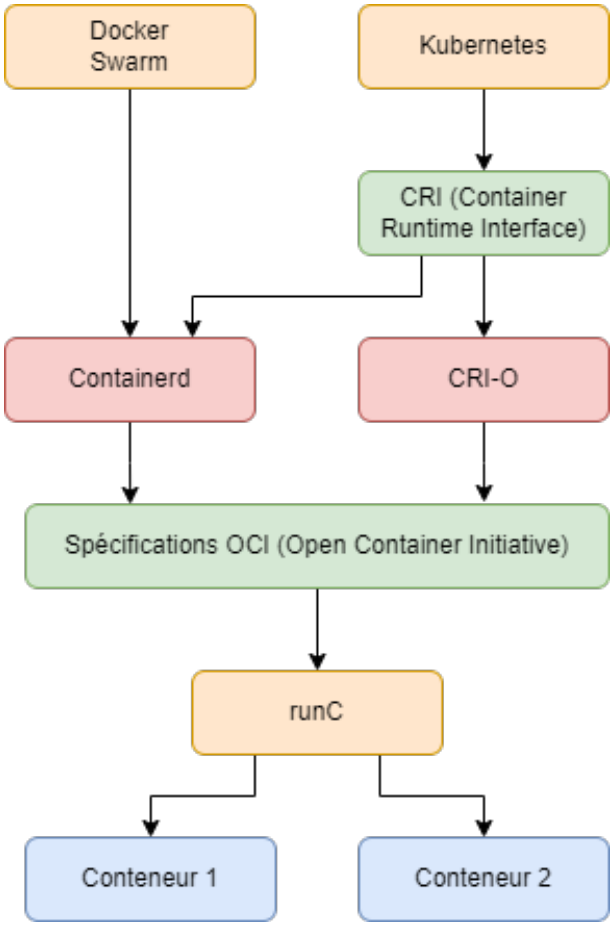


Figure 5.4 – Hiérarchie des composants d'une infrastructure Cloud

redémarrage pour chaque conteneur. Il n'est pas possible de faire des automates de dépendances. L'autre orchestrateur local est Minikube qui propose toutes les fonctionnalités de Kubernetes sur un seul nœud. La distribution n'est pas gérée. En revanche, cette distribution de Kubernetes permet de tester ou développer des déploiements.

5.5 Synthèse sur l'orchestration de conteneurs

Les orchestrateurs de conteneurs pilotent les déploiements locaux (Docker Compose, Minikube) ou distribués (Kubernetes, Docker Swarm ou Nomad) de conteneurs. La figure 5.4 montre les relations de dépendances entre les différents composants d'une infrastructure Cloud. Chaque « étage » offre son lot de propriétés qui s'additionnent pour définir ce qu'on appelle le Cloud. Dans cette synthèse, nous reprendrons également des éléments de la section 4.4 à propos des conteneurs pour donner une vue complète des propriétés intrinsèques des infrastructures de Cloud.

A l'étage le plus bas, les conteneurs offrent des propriétés de reproductibilité. Comme les conteneurs sont basés sur une image, chaque instanciation aura le même environnement de système de fichier (exécutable, bibliothèques etc.). Les conteneurs sont instanciés par

runC (ou un équivalent en mode bac à sable, cf. section 4.4.5) qui se pilote via la CRI (Container Runtime Interface). Ainsi, les conteneurs sont portables d'un moteur à l'autre pourvu que ce dernier implémente la CRI.

Au-dessus, nous retrouvons les moteurs de conteneurs. Leur travail est de créer l'environnement du conteneur à savoir le réseau, le stockage, la gestion de son cycle de vie etc. Cet étage peut être un frein à la portabilité car comme nous l'avons vu dans la section 4.4, tous les moteurs ne sont pas équivalents en termes de fonctionnalités. Le transportabilité du contexte d'exécution est donc assurée quel que soit le moteur par contre l'environnement peut changer. Si on veut assurer la reproductibilité sur un autre moteur, il faut donc s'assurer que l'environnement peut également être reproduit.

Au plus haut, on trouve les orchestrateurs de conteneurs. Ils sont chargés de gérer les déploiements. L'objectif de ces déploiement est double : la disponibilité et le passage à l'échelle (composant centrale de l'élasticité). La disponibilité est assurée par le fait que l'orchestrateur fera tout ce qui est en son pouvoir pour maintenir l'état cible défini dans le déploiement quitte à ré-instancier des conteneurs défaillants sur des nœuds différents. Le passage à l'échelle essentiel à l'élasticité repose sur la capacité de l'orchestrateur à gérer plusieurs instances de services conteneurisés sous réserve d'adapter l'environnement (essentiellement réseau) pour équilibrer la charge entre ceux ci.

En résumé, nous pouvons dire que le choix de l'orchestrateur dépend de l'échelle et du niveau de service attendu de l'infrastructure cible. C'est d'ailleurs le choix déterminant dans la construction d'une infrastructure Cloud car plus on remonte dans les couches de la figure 5.4, plus les composants sont spécialisés car proches du besoin de l'utilisateur.

CHAPITRE 6

CONCLUSION SUR LES ÉLÉMENTS HPC ET CLOUD

Après avoir explicité et analysé le fonctionnement des processus sous Linux et leurs usages sur des infrastructures de Cloud et HPC, nous pouvons observer, d'une part, que les problématiques sont assez similaires. Dans les deux cas, il s'agit de placer des processus sur une ou plusieurs machines en fonction d'une contrainte de ressources souhaitée par l'utilisateur. D'autre part, l'usage est tout de même différent.

L'ordonnanceur HPC va placer des travaux avec un début et une fin. Le calcul démarre, prend des entrées et produit des sorties. Les processus ne sont pas nécessairement isolés car un contrôle d'accès basé sur le DAC est jugé suffisant sur les clusters HPC actuel du fait que les architectures sont très fermées au réseau extérieur par nature. Les calculs ne créent traditionnellement pas de *sockets* réseaux (excepté pour les calculs à mémoire distribuée, sur un réseau dédié à cet usage), ils ne communiquent pas sur des réseaux publiques et tournent sur des machines derrière un frontal en coupure du réseau interne de l'entité. C'est un réseau privé à l'intérieur du réseau interne. Les ordonnanceurs HPC ont cependant recours à l'isolation de processus comme nous le verrons dans le chapitre 8. Nous avons décidé de placer cette discussion dans la partie relative à l'état de l'art car de nouvelles fonctionnalités émergent régulièrement dans les moteurs de conteneurs utilisés par les ordonnanceurs. C'est un sujet à mi-chemin entre le mode de la production et celui de la recherche. Cependant, le recours actuel à la conteneurisation par les ordonnanceurs vise la commodité de pouvoir utiliser des images prédéfinies plutôt qu'une optique de sécurité. L'isolation est effectivement très partielle car il n'est fait usage que de deux espaces de noms sur les sept disponibles comme illustré dans la figure 6.1.

L'orchestrateur de conteneurs est arrivé plus tard chronologiquement. Il va déployer des services potentiellement accessibles sur les réseaux publics type Internet. Ainsi, ces services exposés se doivent d'être totalement isolés du reste de l'infrastructure pour éviter les déplacements latéraux et / ou les escalades de privilèges de potentiels pirates. Contrairement à l'ordonnanceur HPC, l'orchestrateur sait ajouter ou retirer des conteneurs afin

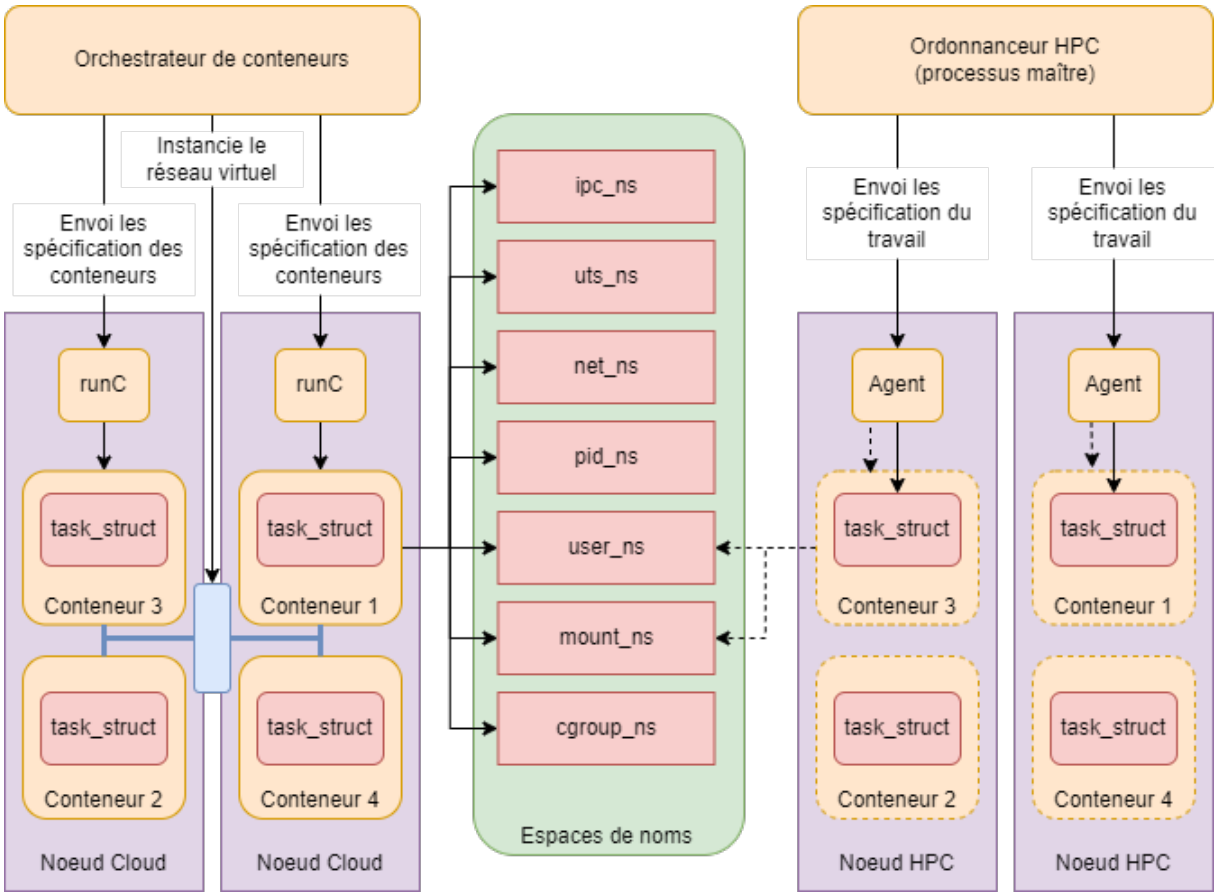


Figure 6.1 – Ordonnancement et orchestration

de s'adapter à la charge à laquelle le service est soumise. Ces conteneurs sont persistants.

Il existe cependant des intersections entre les deux mondes dans le sens où chacun entre légèrement dans le monde de l'autre. Ainsi certains ordonnanceurs HPC tels qu'OAR ou OpenPBS sont dorénavant capables d'intégrer dynamiquement des nœuds de calculs dans leur topologie sans redémarrage. Certains tels que SLURM sont aussi capables de modifier dynamiquement leur configuration. Enfin, les ordonnanceurs HPC intègrent également quelques technologies utilisées en conteneurisation telles que les cgroups et les espaces de nom pour respectivement affiner la granularité des réservations de ressources et exécuter des images de logiciels scientifiques. Ces facultés plus ou moins avancées selon l'ordonnanceur considéré concourent à acquérir la capacité de gérer le « *burst* », c'est-à-dire faire déborder les travaux HPC sur le Cloud en cas de saturation des ressources. Les orchestrateurs de conteneurs font également un pas vers le HPC en intégrant le support des équipements type GPU ainsi que des contrôleurs dédiés au *workflow* calculs (par exemple le contrôleur « *jobs* » dans Kubernetes qui crée un ou plusieurs Pods et s'assure qu'un certain nombre d'entre eux se terminent avec succès).

Les deux mondes ne doivent ni être opposés ni comparés. En effet, un utilisateur disposant d'un code éprouvé et optimisé pour une architecture tirera bien parti d'une infrastructure HPC « classique » sans aucun niveau d'indirection entre le processus de calcul et le matériel. Un utilisateur souhaitant tester le bon fonctionnement d'un code en mode distribué pourra le déployer sur une infrastructure de Cloud avant de le porter sur un cluster HPC. Enfin, certains usagers de calcul scientifique peuvent tout simplement n'avoir aucun besoin d'utiliser un cluster HPC car soit ils utilisent des logiciels interactifs en mode SaaS (*Software as a Service*). Leurs codes sont peu parallèles et les déployer sur un cluster HPC serait de la gourmandise.

Deuxième partie

Contributions scientifiques

CHAPITRE 7

LA COMMUNAUTÉ FACE À L'ENJEU DU HPC DANS LE CLOUD

7.1 Introduction et vue macroscopique

Quand nous examinons, sur le temps long, l'écosystème du HPC [12] et sous l'angle du génie logiciel i.e, la science qui étudie les méthodes de travail et les bonnes pratiques des personnes qui développent des logiciels, nous pouvons affirmer qu'il y a plus de 30 ans, les utilisateurs finaux des machines HPC commençaient par exprimer leurs besoins et leurs souhaits, puis ces besoins étaient ensuite exprimés en termes de fonctionnalités nécessaires au niveau du noyau ou du micro-noyau, puis, il était nécessaire de spécifier la manière dont les noyaux interagissaient en dehors de l'unité centrale, par exemple via le réseau et les entrées / sorties.

Notons au passage, l'intrication forte entre matériel et logiciel, notamment au niveau du système. Comme nous l'avons déjà souligné (voir la section 3.1.5), nous nous inscrivons dans cette perspective historique en mettant au premier plan les problématiques système. Cependant, nous approchons le système via les artefacts de 2023, notamment ceux du DevOps. Reprenons et rappelons maintenant le fil de l'histoire du HPC.

La situation précédente a vite été reconnue comme problématique et a conduit à une ère de co-conception. En effet, dès que le matériel changeait, il était nécessaire de repartir de zéro. Il est donc apparu nécessaire de compter sur des couches d'abstraction, si possible standardisées, afin de résister à l'usure du temps., par exemple. C'est alors que les utilisateurs/développeurs d'applications, les développeurs de logiciels et de bibliothèques systèmes ainsi que les architectes informatiques se sont réunis pour concevoir, simuler et, dans certains cas, construire des modèles réduits des différents composants logiciels et matériels pour le HPC.

Cette approche a permis à un petit nombre d'utilisateurs (ceux considérés comme re-

présentant des applications importantes, principalement la simulation et la modélisation scientifiques et techniques) d'avoir un rôle plus direct et plus intégré dans le développement des systèmes HPC. Ces systèmes, tels qu'ils ont été développés, se concentrent habituellement sur une utilisation et un débit élevés de « gros travaux » (on sous entend ici que les travaux durent longtemps et font un usage intensif des ressources matérielles). Les utilisateurs se sont alors habitués à utiliser un ordonnanceur de travaux. Ce modèle de soumission et d'attente a été bien accepté car les applications ne sont généralement pas soumises à des contraintes de temps, par exemple de temps réel.

Les architectures matérielles à partir des années 90 sont aussi traditionnellement homogènes, les développeurs portant leurs applications sur chaque système individuellement et pouvant ensuite idéalement utiliser la plus grande partie ou la totalité du système.

A partir des années 2000, en parallèle à l'évolution précédente, de grandes entreprises telles qu'Amazon et Google se sont efforcées de construire des systèmes pour répondre à leurs besoins en matière d'analyse de données. Les charges de travail de ces entreprises ont entraîné un ensemble différent de choix pour les processeurs, la connectivité réseau, les Entrées/Sorties et les autres composants logiciels et matériels, avec une hétérogénéité accrue, pour prendre en charge de multiples charges de travail.

Initialement, ces entreprises avaient des besoins ponctuels, comme par exemple, lancer les facturations en fin de journées, et pour faire simple. Les ressources matérielles (serveurs) se sont alors trouvés inutilisés en journée si bien qu'un nouveau modèle économique est apparu. Il s'agit de payer quand on en a besoin (*pay-as-you-go*), modèle qui comprend l'achat d'instances à bas coût (instance Spot chez Amazon). Le modèle de Cloud est alors vite apparu comme intéressant pour les informaticiens en général, un peu moins aux informaticiens du HPC car ceux-ci ont toujours souhaités, au fil de l'histoire, garder la main sur les niveaux matériels jusqu'aux niveaux haut du système et des applications.

Cependant, le modèle du Cloud vient aussi avec une automatisation du déploiement, de la configuration et de l'approvisionnement en ressources, et il est facile à utiliser. Il est donc très attrayant, en particulier pour les personnels des grands centres de calcul qui ont été habitués aux longs processus d'examen des procédures d'attribution des heures de calcul pour déterminer les allocations de temps sur les systèmes HPC ainsi que les processus manuels pour mettre en œuvre ces décisions d'allocation. Par ailleurs, l'idée de ressources faciles à transporter d'une architecture à une autre qui est nécessaire à cause de l'hétérogénéité sous-jacente du Cloud est permise par les technologies de conteneurs.

Aujourd'hui, il est clair qu'il existe de nombreuses applications HPC qui fonctionnent bien sur les Clouds commerciaux et les Clouds privés. Il est également clair que d'autres applications ne fonctionnent pas bien, pour une variété de raisons comme les changements dans les interconnexions réseau, les systèmes de virtualisation et les niveaux optimaux de précision numérique.

La communauté travaillant sur la convergence HPC-Cloud doit comprendre cet état de fait, et se poser les questions des changements qui pourraient être apportés au niveau de l'application, du logiciel système et du matériel pour augmenter la proportion d'applications qui fonctionnent mieux sur le Cloud ou sur un système dédié HPC. Le principal défi de cette communauté est de combler le fossé entre les besoins de l'utilisateur et ceux

du système à grande échelle.

Examinons maintenant quelques approches, au niveau macroscopique. Les avancées scientifiques, dites du niveau microscopique, seront présentées au chapitre 8 et seront plus spécifiquement relatives à notre positionnement. Ainsi, le chapitre courant et le suivant constituent deux états de l'art en matière de HPC dans le cloud. Dit autrement, le chapitre courant donne une vue générale des problématiques du HPC dans le cloud alors que le chapitre suivant discutera plus spécifiquement des travaux en connexion directe avec notre travail.

Donc, pour l'instant, examinons ce que la communauté investi, en dehors de notre propre vue. Nous passons successivement en revue les travaux au Lawrence Livermore National Laboratory, en particulier ceux du groupe autour de Daniel Milroy (*Center for Applied Scientific Computing*), puis certains travaux de synthèse présents dans [49], puis d'autres travaux publiés dans le workshop SuperCompCloud de la conférence internationale Supercomputing, workshop qui porte sur « l'Interoperability of Supercomputing and Cloud Technologies », et enfin un travail lié à l'ordonnancement des conteneurs.

Nous avons sélectionnés ces travaux pour être le plus exhaustif possible, et surtout présenter une vue de haut niveau des problématiques du domaine. Même s'ils ne sont pas tous liés directement à notre direction de recherche dans la thèse, ils seront, autant que possible, mis en perspective. La discussion autour de MPI et de la tolérance aux fautes sera reprise, par exemple, un peu plus tard en remarquant que le besoin d'un MPI plus « dynamique »devient prégnant dans le cadre de la convergence HPC et Cloud. On entend ici par communications dynamiques, des schémas de communications qui devraient évoluer dans le temps imparti au programme parallèle puisque nous pouvons ajouter et supprimer des nœuds, dynamiquement.

7.2 Quelques travaux au LLNL (*Lawrence Livermore National Laboratory*)

L'orchestration de travaux HPC dans le Cloud est confrontée à plusieurs limites. En effet, les ordonnanceurs de conteneurs intégrés dans les orchestrateurs ne sont pas prévus pour déployer des travaux HPC tels que nous les avons définis dans la section 3.2.2. Dans cette section consacrée aux travaux de LLNL et IBM sur le sujet, nous allons d'abord mettre en évidence les limites de l'orchestrateur Kubernetes, puis nous présenterons les travaux basés sur KubeFlux pour y remédier.

7.2.1 Les limites de l'orchestration dans le Cloud

La gestion des workflow HPC dans le Cloud est différente de la gestion des travaux HPC dans une configuration centre de calcul. La différence provient du fait que dans le Cloud, un travail HPC est simplement un sous ensemble des travaux que doit exécuter l'orchestrateur. Une telle contrainte nécessite de repenser l'ordonnancement des travaux.

Dans cette section, nous allons présenter quelques méthodes pour gérer cette hétérogénéité.

Dans [50], les auteurs présentent une évaluation de la possibilité de mettre en place des *workflows* HPC sur des infrastructures Cloud. Les auteurs parlent essentiellement de l'impact de trois granularités de topologies de gestion des processus : *bare-metal* (le processus s'exécute directement sans isolation), conteneur Docker (le processus est conteneurisé) et avec Kubernetes (processus conteneurisé et ordonnancé par Kubernetes). Ils ont évalué (1) l'accès direct au matériel, (2) les communications inter-conteneurs dans les trois topologies d'ordonnancement et enfin (3) les limitations de configuration. Pour le (2), ils ont considéré les communications MPI sur TCP/IP et RDMA. Leur conclusion est que Kubernetes induit des surcoûts sur les communications TCP/IP. Cependant, Kubernetes supporte une multitude de réseaux d'*Overlay* utilisant des mécanismes très différents au niveau TCP/IP. On devine dans le papier que seul Flannel a été évalué.

Le lancement de travaux HPC dans le Cloud se confronte à certaines limites. Dans [51], les auteurs font un retour d'expérience sur le lancement de travaux HPC directement sur l'orchestrateur Kubernetes. Leurs travaux ont été lancés sur un cluster Kubernetes hétérogène et dynamique. Ils ont identifié que l'absence d'algorithme de type *fairshare* est impactante pour un placement efficace des Pods accueillant les travaux HPC.

Les ordonnanceurs HPC et les orchestrateurs Cloud partagent le même objectif de placer au mieux les travaux sur les ressources demandées. Cependant, les ordonnanceurs HPC sont peu habiles avec la gestion de l'élasticité du fait que les ressources sont stockées à plat de façon statique sous forme d'une *bitmap*. C'est très rapide pour trouver un nœud libre, en revanche la gestion de ressources dynamiques est très compliquée. Oar [52] fait un peu exception à la règle car il propose une gestion des ressources plus dynamique et permet de définir des ressources abstraites. L'orchestrateur Kubernetes possède un algorithme d'ordonnancement assez simple. Il place les Pods de façon séquentielle sur le premier nœud qui donne satisfaction. Ensuite, il surveille le déploiement pour s'assurer que l'état courant reste conforme avec l'état cible.

Dans [51], les auteurs montrent que Kubernetes est un peu trop simple pour pouvoir exécuter des travaux HPC de façon efficace. Les points qu'ils soulèvent sont : 1) un Pod est fait pour tourner jusqu'à ce qu'on l'arrête contrairement à un travail HPC qui a un début mais aussi une fin 2) Les travaux interactifs ne sont pas supportés 3) Les orchestrateurs ne sont pas très contraignant pour demander des contraintes de ressources aux utilisateurs, ces derniers sont tentés de surestimer leurs besoins et 4) Il n'y a pas d'implémentation du *fairshare*. Cependant des travaux sont en cours, comme ceux discutés à la prochaine section, pour permettre à Kubernetes de rattraper son retard sur l'orchestration de travaux HPC.

7.2.2 KubeFlux : vers une orchestration des travaux HPC efficace

Dans [53] les auteurs introduisent Flux, un framework s'interfaçant avec Kubernetes et proposant 1) Une représentation générique des ressources 2) Une prise de décision passant à l'échelle 3) Une architecture tolérante aux pannes et 4) Un ordonnancement à plusieurs étages (l'allocation peut transiter d'un ordonnanceur père à un fils plus « local »). Kube-

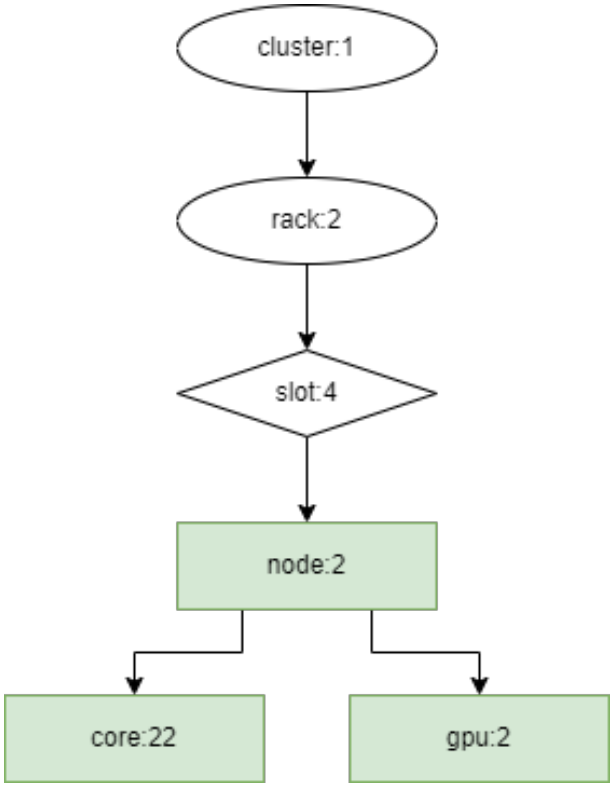


Figure 7.1 – Réservation de ressources avec KubeFlux

Flux [54] et [55] s'appuie sur ce *framework* pour permettre à Kubernetes d'ordonnancer des travaux HPC sous forme de Pods sur une infrastructure Cloud standard. Les noeuds de calcul sont enregistrés dynamiquement dans l'ordonnanceur KubeFlux via NFD [1] (Node Feature Discovery). Ensuite l'utilisateur formule sa demande de réservation de ressources sous forme d'un graphe comme dans la figure 7.1. Pour cette demande de réservation, l'utilisateur demande quatre slots de deux nœuds chacun. Chaque nœud doit posséder au moins 22 cœurs et deux GPUs. Avec la directive de haut niveau « rack », ces *slots* doivent être répartis sur deux racks.

Le travail [56] est une expérimentation de KubeFlux par les auteurs du travail précédent [54] et [55]. Ils ont éprouvé l'algorithme de placement de Kubernetes par défaut avec KubeFlux sur un cluster de 34 nœuds dans le Cloud IBM sous RedHat OpenShift. Ils ont lancé des travaux GROMACS, une application bien connue de simulation en dynamique moléculaire. Les auteurs ont constaté que l'utilisation de KubeFlux améliore l'efficacité du travail GROMACS lancé.

L'orchestration de *workload* HPC dans le Cloud ne se limite pas à Kubernetes. Dans [57] les auteurs utilisent Docker Swarm pour déployer un cluster MPI conteneurisé. L'idée est de créer des images contenant à la fois la couche MPI et un jeu de clés SSH. Le processus conteneurisé est le serveur sshd qui attend les connexions MPI. Ces conteneurs sont ensuite recrutés dans le cluster MPI pour y exécuter les travaux à mémoire distribuée. Cette approche rappelle un peu le travail publié dans [58], et que nous avons présenté dans

1. https ://github.com/kubernetes-sigs/node-feature-discovery

la section 8.3.1, mais avec une approche totalement Cloud.

7.3 Quelques travaux présentés dans un ouvrage de synthèse

L'ouvrage intitulé « High Performance Computing in Clouds - Moving HPC Applications to a Scalable and Cost-Effective Environment » [49], publié à la mi 2023 chez Springer constitue une vue synthétique de nombreux problèmes touchant à la convergence, mais aussi la divergence, du HPC dans le Cloud, en particulier sur le plan des applications.

Les chapitres qui nous intéressent ici sont les chapitres 1 (Why Move HPC Applications to the Cloud ? [59]), puis 4 (Deploying and Configuring Infrastructure [60]), puis 6 (Designing Cloud-Friendly HPC Applications [61]), puis 9 (Harnessing Low-Cost Virtual Machines on the Spot [62]), et enfin 10 (Ensuring Application Continuity with Fault Tolerance Techniques [63]). Notons que la partie III de l'ouvrage est dédiée aux retours d'expérience pour des domaines applicatifs de bioinformatique, d'apprentissage machine ou encore de déploiement et de configuration de systèmes Cloud réels.

Le chapitre 1 discute des motivations qui sous tendent le titre de l'ouvrage. Alors que dans les années 1990 les principes d'économies d'échelle ont favorisé la conception de microprocesseurs pour des applications générales, principalement en raison de l'industrie du PC, les tendances actuelles montrent que les exigences des centres de données du Cloud vont orienter la conception de ces puces [64, 65, 66].

Bien que nous n'ayons pas encore complètement atteint ce stade, l'exécution des charges de travail HPC dans le Cloud présente déjà plusieurs avantages bien connus, à savoir (a) il n'est pas nécessaire d'acquérir, d'installer et d'entretenir du matériel coûteux, ni d'investir dans des installations physiques pour héberger ce matériel ; (b) il n'y a pas de contraintes énergétiques ou de refroidissement qui entravent l'augmentation de la capacité de calcul et de stockage. Le Cloud donne l'impression de ressources infinies alors que d'un point de vue de la soutenabilité il faut bien construire, donc extraire des matières rares, qui sont finies, pour fabriquer les composants électroniques, mais ceci est une autre histoire ; (c) il n'est pas nécessaire de faire des efforts sur des obligations afférentes aux grands systèmes, comme l'embauche et l'entretien de professionnels qualifiés en informatique, la gestion de licences logicielles ou la résolution de problèmes d'obsolescence du matériel, ce qui, une nouvelle fois, d'un point de vue soutenabilité, est problématique ; en outre, (d) il n'y a pratiquement pas de retards associés à l'acquisition et à l'installation du système, ni de temps d'attente dans les files d'attente de travaux causés par une utilisation élevée du système. L'informatique du Cloud a en effet été optimisée à l'extrême, alors que nous pourrions nous poser la question de faire aussi bien mais avec moins de ressources, afin de respecter certains critères de soutenabilité. Mais, ici encore, c'est une autre histoire.

Néanmoins, pour les auteurs du chapitre, plusieurs défis doivent encore être relevés pour faciliter ce processus d'économie d'échelle à plus grande échelle. Trouver l'infra-

structure adéquate (c'est-à-dire l'ensemble des ressources informatiques du Cloud) pour chaque application ou adapter l'application pour tirer parti de nouvelles fonctionnalités, telles que l'élasticité du Cloud, sont quelques exemples de ces défis. Les différents chapitres de l'ouvrage traitent des stratégies et des techniques permettant d'optimiser l'utilisation de l'informatique du Cloud, en particulier sur le plan des applications.

Le chapitre 4 présente les éléments d'infrastructure clés proposés par les fournisseurs de Cloud dans le cadre du modèle IaaS (Infrastructure as a Service), et examine les moyens de les déployer et de les configurer pour créer des clusters HPC. La problématique est donc ici de déterminer une infrastructure Cloud, la plus adéquate aux problèmes HPC, ou la plus générique possible.

Le cycle de vie d'une infrastructure HPC dans le Cloud suit les étapes suivantes qui sont maintenant bien comprises et documentées. Les clusters haute performance hébergés chez les fournisseurs de services en nuage sont construits avec un ensemble de composants et de services virtuels. Les nœuds de connexion et de calcul sont mis en œuvre avec des machines virtuelles AWS si l'on est abonné chez Amazon. Le nœud de connexion est par exemple de type *m5.xlarge* tandis que les nœuds de calcul sont plus puissants, par exemple de type *c6i.32xlarge*. Chaque machine virtuelle est associée à un dispositif de stockage par blocs, qui stocke l'image de la machine virtuelle et les données locales. Le principal composant de stockage est mis en œuvre par un service de stockage Cloud basé sur des fichiers, le système de fichiers élastiques AWS, ou via le service EFS. Ce système de fichiers est monté sur tous les nœuds, y compris les nœuds de connexion et de calcul, ce qui permet à l'utilisateur et aux applications d'accéder de manière transparente aux fichiers de données - l'utilisateur accède à ces fichiers à partir du nœud de connexion (par exemple, via une connexion à distance), tandis que les applications y accèdent à partir des nœuds de calcul. Enfin, tous les composants sont reliés par un réseau privé virtuel (VPC dans la terminologie consacrée du Cloud).

Une fois l'infrastructure déployée et configurée, un outil de gestion de grappe de machine et un ordonnanceur (par exemple SLURM) peut être installé sur tous les nœuds pour ce qui est des *daemons* SLURM, ce qui permet aux utilisateurs d'exécuter leurs tâches HPC comme s'ils se trouvaient sur un cluster HPC traditionnel.

Une fois que l'utilisateur dispose d'un compte chez un fournisseur de Cloud, il peut déployer l'infrastructure, automatiquement, via des recettes d'installation et de configuration. Ces recettes peuvent commencer par créer un réseau VPC et un système de fichiers partagés, avant de les associer aux machines virtuelles du cluster virtualisé. Finalement, quand le cluster n'est plus nécessaire, l'utilisateur peut détruire l'infrastructure pour éviter des dépenses inutiles puisqu'il opère sous le mode « *pay-as-you-go* ».

Une des difficultés techniques pour ce schéma de fonctionnement est de déterminer les machines virtuelles adéquates ainsi que les ressources associées (en terme de nombre de cœurs, réseau, stockage). Mais les fournisseurs de Cloud ont fait des efforts importants pour masquer les difficultés du processus complet si bien que la création du cluster HPC dans le Cloud est, de nos jours, à la portée d'un non informaticien. Cependant, la sélection des ressources qui optimisent le rapport coût/bénéfice de l'utilisation de l'environnement de Cloud peut ne pas être une tâche triviale, en raison de la quantité de configurations de

ressources différentes proposées par les fournisseurs de nuage. Il existe plusieurs stratégies pour traiter ce problème et déterminer la meilleure configuration adaptée à une charge de travail donnée. Il s'agit d'un problème d'optimisation combinatoire... qui n'est pas abordé dans les chapitres de notre thèse, qui s'intéresse avant toutes choses aux problématiques purement système de la construction d'un cluster HPC dans le Cloud. Cependant, la dernière section de ce chapitre introduit les problèmes et les solutions d'un article auquel nous avons participé, lié au placement de conteneurs.

Le chapitre 6, quant à lui, explore la problématique de comprendre comment les applications HPC sont écrites, ainsi que les détails concernant l'architecture informatique. Les auteurs rappellent que nous avons à faire avec cinq modèles d'application : Bag-of-Tasks, Master-Slave, Pipeline, Divide-and-Conquer et Bulk-Synchronous Parallel. Chacun de ces modèles présente des particularités en matière de couplage et d'interaction des processus, qui ont un impact considérable sur les spécifications de l'unité centrale (CPU/cœurs), de la mémoire et du réseau. Le choix d'un modèle (ou d'une combinaison de modèles) doit également tenir compte de la manière dont nous pouvons exploiter les installations Cloud et l'infrastructure matérielle fournie. Nous pouvons ensuite extraire du parallélisme au niveau de l'application, parallélisme de type multi-processeur (multi-cores) et/ou de type multi-ordinateur, en combinant des bibliothèques appropriées comme OpenMP et MPI (Message-Passing Interface).

Parmi les caractéristiques essentielles du Cloud, les auteurs du chapitre s'intéressent à l'élasticité qui est mis au regard des cinq modèles d'application. Pour cela, ils reprennent l'étude de Kehrer et Blochinger [67], qui introduit trois stratégies d'adaptation au Cloud qui peuvent être appliquées à différentes applications parallèles : (1) le copier-coller, (2) le remaniement en fonction du Cloud et (3) le remaniement en fonction du Cloud et le contrôle de l'élasticité.

La stratégie du copier-coller consiste à migrer les applications parallèles existantes sans les modifier. Dans ce cas, les utilisateurs peuvent remplacer leur infrastructure HPC par du matériel virtuel, en exploitant les ressources immédiatement disponibles dans les environnements Cloud proposés et le modèle de paiement à l'utilisation. Les options pertinentes sont les solutions IaaS ou les solutions de conteneurs, car elles peuvent être personnalisées pour héberger la pile logicielle nécessaire à l'exécution de ces applications et coordonnées avec des intergiciels de calcul distribué capables d'interagir avec des infrastructures basées sur le Cloud. Les principaux inconvénients de cette stratégie sont les suivants (1) l'élasticité n'est pas utilisée, étant donné que l'application n'a pas été modifiée pour gérer des ressources variables. Ainsi, le nombre d'unités de traitement doit être sélectionné de manière statique ; et (2) certaines caractéristiques du Cloud, telles que les vitesses de traitement hétérogènes, les latences de réseau variables, les faibles largeurs de bande de réseau et les frais généraux de virtualisation, peuvent affecter les performances des applications qui exigent des communications synchrones (celles qui sont étroitement couplées).

La stratégie de remaniement en fonction du Cloud (*Cloud-aware Refactoring*) propose un remaniement architectural pour rendre les applications parallèles existantes compatibles avec le Cloud, c'est à dire les rendre moins affectées par les caractéristiques des environnements Cloud standard. Dans ce contexte, Fehling et al [68] ont identifié les propriétés des applications qui permettent effectivement de tirer parti des environnements

Cloud. Pour les auteurs, il s'agit de l'isolation de l'état, distribution, élasticité, gestion automatisée et couplage souple. L'isolation de l'état est liée à la conception des parties d'une application Cloud pour qu'elles soient sans état, ce qui permet d'isoler l'état dans de petites parties de l'application. La distribution est obtenue en décomposant une application en composants distincts qui peuvent être répartis entre les ressources disponibles. L'élasticité se concentre sur l'ajout et la suppression dynamiques de ressources informatiques et exige que l'application gère la quantité variable de ressources.

La stratégie de remaniement en fonction du coud et le contrôle de l'élasticité (*Cloud-aware Refactoring and Elasticity Control*), consiste en l'utilisation de l'élasticité pour traiter les charges de travail HPC dans le Cloud conjointement avec la prise en compte, en plus du remaniement de l'application. Cette capacité à exécuter des applications parallèles avec des ressources dynamiques peut offrir plusieurs avantages, notamment l'amélioration des performances et de l'efficacité des applications, la réduction des coûts, la tolérance aux pannes, l'équilibrage des charges et une meilleure utilisation des ressources [68]. Plusieurs articles relatifs à l'informatique du Cloud traitent de l'exploration de cette caractéristique et étudient les mécanismes d'élasticité. Les auteurs du chapitre 6 mettent alors l'accent sur les travaux présentés dans les références [69], [70] et [71]. Remarquons enfin que des solutions d'élasticité sont proposées dans cette thèse, au niveau système, et que ces mécanismes pourront être utilisés pour implémenter de l'élasticité pour les applications HPC s'exécutant sous SLURM (par exemple) conteneurisé dans un Cloud.

Dans le chapitre 9 de l'ouvrage les auteurs abordent les questions d'utilisation de ressources de Cloud qui ne sont pas disponibles tout le temps mais sujettes à des interruptions. Chez les fournisseurs de Cloud, afin de maximiser l'utilisation du Cloud et donc de leurs revenus, par exemple chez Amazon, il est d'usage de proposer des instances du marché Spot. Dans ce cas de figure, les ressources inutilisées du fournisseur sont disponibles pour des tarifs avec une réduction pouvant aller jusqu'à 90% par rapport au modèle à la demande, mais à la condition que ces ressources soient libérées à la demande du fournisseur, à tout moment.

Aujourd'hui, bien que les principaux fournisseurs proposent des *VM Spot*, seul AWS offre également la fonction d'hibernation des *VM Spot*. Dans ce cas, lorsque la VM est mise en veille par AWS, sa mémoire et son contexte sont sauvegardés, et lorsque AWS réactive la VM, le contexte est restauré et les tâches interrompues sont redémarrées à partir du point d'hibernation.

Les auteurs du chapitre 9 présentent alors une vue d'ensemble d'un mécanisme d'ordonnancement développé pour programmer des applications de type « sac de tâches » (BoT) soumises à des contraintes de délais sur des machines virtuelles Spot avec hibernation, dans le but de minimiser les coûts financiers pour l'utilisateur. Il s'agit donc bien d'un problème qui s'intéresse à faire du HPC dans le Cloud. Notons que les applications « sac de tâches » se prêtent bien à l'utilisation de ressources sur le marché Spot car elles correspondent à des tâches qui ne communiquent pas entre elles, alors qu'elles durent très longtemps. Toujours est-il que la problématique d'ordonnancement est intéressante à plus d'un titre.

L'ordonnanceur (HADS) commenté par les auteurs du chapitre 9 planifie les applica-

tions BoT avec des contraintes de délai dans les *VM Spot* (pour des raisons de coût) et les VM à la demande. Par conséquent, si une *VM Spot* hiberne à temps pour respecter le délai d'une application, les tâches déjà assignées à cette instance Spot hibernée sont reprises à partir du point de sauvegarde lorsqu'elle sera à nouveau disponible. Toutefois, si ce n'est pas le cas, une défaillance temporelle risque de se produire et le délai de l'application ne sera pas respecté. Par conséquent, l'objectif de HADS est d'offrir une solution d'ordonnancement dynamique qui garantisse l'exécution des tâches des applications avec des contraintes de délai, en évitant les défaillances temporelles même en présence d'hibernations multiples avec un coût monétaire minimal en ce qui concerne les prix d'allocation des VMs. À cette fin, le cadre général de l'algorithme fournit des mécanismes pour migrer les tâches d'une *VM Spot* hibernée chaque fois qu'elle ne reprend pas suffisamment tôt pour garantir les contraintes de délai de l'application. Si les *VM Spot* existantes allouées ne sont pas suffisantes pour exécuter ces tâches, de nouvelles VM à la demande doivent être déployées.

Nous pouvons maintenant déduire que l'algorithme d'ordonnancement a besoin de deux modules principaux : (1) le module heuristique d'ordonnancement primaire qui définit une liste initiale d'ordonnancement des tâches, et (2) un module d'ordonnancement dynamique piloté par les événements qui, si nécessaire, migre les tâches vers d'autres machines virtuelles afin que le délai de terminaison soit respecté. En outre, pour réduire les coûts ou équilibrer la charge, ce module peut également migrer les tâches des machines virtuelles occupées vers les machines virtuelles inutilisées en appliquant une procédure de vol de travail. Il s'agit d'un mécanisme de consolidation, technique bien connue par ailleurs. Enfin, pour éviter d'exécuter les tâches migrées depuis le début, les tâches sur les *VM Spot* prennent des points de contrôle périodiquement. Ainsi, celles qui étaient exécutées dans une VM qui a hiberné sont migrées vers d'autres VM et commencent leur exécution à partir de leur dernier point de contrôle respectif. Ceci est relativement facile à implémenter car les tâches ne communiquent pas, donc nous n'avons pas à sauvegarder « l'état des communications via le réseau ». D'autre part, le point de contrôle d'une tâche induit des frais généraux, augmentant le temps d'exécution de la tâche, qui doivent être pris en compte par le module d'ordonnancement primaire lors du placement des tâches sur les *VM Spot*.

Pour conclure avec ce schéma algorithmique, nous pouvons remarquer qu'il est complexe dans le sens où deux modules interagissent, mais il est relativement générique, laissant libre court aux développeurs d'instancier une technique particulière pour une sous-brique de l'algorithme général. C'est le cas par exemple pour le choix de la technique de consolidation.

Dans le chapitre 10, les auteurs relèvent que la probabilité que les défaillances aient un impact sur les services Cloud et les applications des utilisateurs est extrêmement élevée. En particulier, dans les applications HPC, généralement composées de tâches de longue durée dont l'exécution peut durer des jours, voire des mois, les défaillances peuvent avoir des conséquences négatives importantes sur l'exécution correcte de l'application et même entraîner une perte de travail. Il est donc fondamental de connaître le niveau de tolérance aux pannes qu'une application HPC donnée exige et quelle est la technique de tolérance aux pannes idéale pour minimiser la perte de travail.

Il convient de noter que, dans ce chapitre 10, les auteurs considèrent également la révocation des services comme un type de défaillance et ils examinent comment les techniques de tolérance aux fautes (FT) peuvent être utilisées pour extraire les avantages économiques maximaux de ce modèle d'exécution tout en garantissant l'exécution correcte des applications. Il convient d'entendre ici le terme révocation dans le sens ou il a été discuté précédemment avec les instances Spot.

Les auteurs considèrent également les fautes de type *crash*. Dans le contexte du Cloud, un *crash* se produit lorsqu'une ressource s'arrête de manière inattendue. Par exemple, lorsqu'une VM à la demande cesse de fonctionner avant que le client ne la libère. D'autre part, une révocation se produit lorsqu'une ressource, telle qu'une machine virtuelle préemptible, est intentionnellement arrêtée par le fournisseur. Dans les deux cas, les techniques les plus courantes utilisées pour les tolérer sont les points de contrôle, le retour en arrière et la réplication.

Les auteurs passent alors en revue les techniques de tolérance aux fautes, discutent de leur implémentation et de leurs limites, en particulier pour le cas de MPI-FT, le MPI qui tolère les fautes. La partie concernant le Cloud discute des bonnes pratiques pour implémenter la tolérance aux fautes de manière effective. Reprenons ici le cas des points de contrôle.

Un point de contrôle peut être mis en œuvre à l'aide de l'un des trois niveaux distincts suivants, en fonction du degré de transparence souhaité vis à vis de l'utilisateur et de l'emplacement dans la pile logicielle [72, 73, 74] : (1) au niveau de l'application, (2) au niveau de l'utilisateur et (3) au niveau du système.

Au niveau de l'application, le code de l'application doit indiquer quand les points de contrôle doivent être effectués. Ensuite, la procédure de point de contrôle capture l'état de l'application par une interaction directe avec elle. Cette approche devrait être la plus efficace puisque le programmeur sait quelles structures de données et quelles variables doivent être préservées et lesquelles peuvent être supprimées. Toutefois, son applicabilité est limitée au cas où le code de l'application peut être modifié. En outre, le temps de récupération peut être un problème car il correspond au temps nécessaire pour demander, démarrer et configurer une nouvelle machine virtuelle pour charger l'application.

Au niveau de l'utilisateur, les points de contrôle sont mis en œuvre dans l'espace utilisateur et fournissent une transparence à l'application en virtualisant les appels système. Selon [73], une telle virtualisation permet aux outils de point de contrôle de capturer l'état de l'ensemble du processus sans être liés au noyau, offrant ainsi une plus grande portabilité entre les plates-formes, mais au prix d'un surcoût constant de virtualisation. En outre, les points de contrôle au niveau de l'utilisateur sont généralement plus importants que ceux au niveau de l'application, car ils ne peuvent pas profiter de l'optimisation de la mémoire basée sur la sémantique de l'application.

Les procédures de point de contrôle au niveau du système sont mises en œuvre soit dans le noyau, soit sous la forme d'un module du noyau. Dans ce cas, toute la pile de mémoire de l'application est sauvegardée. Contrairement à la mise en œuvre au niveau de l'utilisateur, le point de contrôle au niveau du système n'a pas d'effet sur la continuité de l'application car nous n'avons pas besoin de virtualiser les interfaces d'appel système

puisqu'il a un accès direct aux structures du noyau [73]. Cependant, elles sont souvent liées à la version du noyau, ce qui les rend non portables entre différentes plates-formes.

CRIU [75], un outil de « checkpointing » très populaire, a été utilisé dans plusieurs travaux pour garantir la fiabilité des applications fonctionnant dans les Cloud [76] et [77]. Il s'exécute au niveau de l'utilisateur et sauvegarde l'état complet du processus sans aucune modification du code de l'application. Dans Teylo et al [78], les auteurs ont appliqué CRIU pour enregistrer les points de contrôle des applications BoT fonctionnant dans un nuage Amazon EC2. Les auteurs bouclent alors sur l'ordonnateur HADS que nous avons précédemment introduit. Les résultats des expériences utilisant des applications synthétiques [79] et l'application de référence NAS [80], s'exécutant à la fois dans des VM ponctuelles et à la demande sujettes à l'hibernation, confirment l'efficacité de HADS en termes de coûts monétaires par rapport à une approche basée uniquement sur les VM à la demande Amazon EC2. Ils montrent également que le système HADS évite les défaillances temporelles, même en présence de plusieurs événements d'hibernation de VM ponctuelles, et que la stratégie de point de contrôle/récupération est capable de réduire l'impact sur le temps d'exécution de l'application, en cas de migration des tâches.

Il convient de noter que la définition de la bonne fréquence des points de contrôle dans les environnements en nuage n'est pas une tâche simple, en particulier du côté du l'utilisateur. Par conséquent, dans les Clouds, les intervalles de contrôle sont généralement soit des intervalles fixes définis par l'utilisateur, soit des intervalles adaptatifs [81].

7.4 Quelques travaux présentés aux workshops SuperComp-Cloud

La série d'ateliers SuperCompCloud se tient soit pendant la conférence internationale ISC High Performance soit pendant SC (SuperComputing), deux évènements scientifiques reconnus et dont le cœur des discussions est le HPC. La première édition de SuperComp-Cloud date de 2019. Son contour est l'intersection des technologies HPC et de Cloud. SuperCompCloud, comme atelier spécifique au domaine de la convergence HPC et Cloud, réunit des experts et des praticiens du monde universitaire, des laboratoires nationaux et de l'industrie pour discuter des technologies, des cas d'utilisation et des meilleures pratiques afin de définir une vision et une orientation holistique pour tirer parti de la combinaison du calcul à haute performance et à échelle extrême et des écosystèmes de Cloud à la demande.

Dans l'édition 2020 de SuperCompCloud nous pouvons noter deux articles relatifs au problème de rendre l'utilisation facile de ces systèmes HPC / Cloud, soit du point de vue de l'utilisateur final, soit du point de vue des API de programmation. L'article intitulé « Exosphere - Bringing The Cloud Closer » [82] s'intéresse à l'écosystème de OpenStack, l'orchestrateur Cloud particulièrement utilisé aux États Unis, mais aussi au CERN.

Les services de Cloud pour la recherche doivent fournir des interfaces conviviales afin d'élargir leur adoption au sein des communautés scientifiques, en particulier par les chercheurs qui n'ont pas de solides connaissances en informatique. Malheureusement, l'inter-

face utilisateur par défaut d'OpenStack (appelée Horizon) a été développée pour les administrateurs de systèmes informatiques. Pour effectuer des tâches courantes (par exemple, créer et se connecter à une instance de Cloud), Horizon exige une certaine familiarité avec les groupes de sécurité des pare-feu, les paires de clés SSH et les réseaux privés virtuels. Nous pouvons cependant noter que les interfaces utilisateurs de Amazon et Microsoft, quant il s'agit de demander le déploiement d'une instance, procèdent de la même manière. Cela veut peut-être dire qu'il y a maintenant un relatif consensus sur ce qu'il est requis pour utiliser et déployer une VM chez un fournisseur de cloud public.

Toujours est-il que les auteurs, ici, proposent le système Exosphere qui peut supporter de nombreuses applications informatiques de la recherche et de nombreuses manières. En effet, Exosphere permet aux utilisateurs de créer et de gérer des serveurs persistants pour les bases de données et les passerelles scientifiques, des clusters évolutifs pour le calcul scientifique, les salles de classe et les ateliers, et des serveurs bac à sable jetables pour une flexibilité maximale pour les travaux de test et de développement. Un intérêt pratique est que Exosphere est écrit avec Elm [83], un langage de programmation fonctionnel pour la construction d'interfaces utilisateur. Il est compilé en JavaScript, HTML et CSS. La même base de code est utilisable dans un navigateur web standard, ou en tant qu'application de bureau multiplateforme (utilisant actuellement le cadre Electron [84]). Autrement dit, le code repose sur un certain nombre de standards du Web, même si certaines limitations sont discutées dans l'article. Par exemple, Exosphere communique avec les API REST/HTTP d'OpenStack au nom de l'utilisateur, mais les navigateurs modernes imposent la politique de même origine, une fonction de sécurité qui empêche Exosphere de faire ces appels API à moins que l'administrateur d'OpenStack ne configure le partage des ressources entre les origines.

Dans l'édition 2020 de SuperCompCloud, nous trouvons également l'article intitulé « FirecREST : a RESTful API to HPC systems » [85]. La problématique est de nouveau ici les passerelles entre systèmes de calcul scientifique. Comme précédemment, il est important que les centres de calcul à haute performance fournissent une interface moderne et accessible à l'extérieur (du système), telle que des API compatibles avec le Web. Une telle interface permet d'accéder aux ressources du centre de calcul à haute performance pour permettre aux portails web scientifiques de soumettre un travail ou de déplacer des données à l'intérieur et à l'extérieur du centre de calcul à haute performance. Le travail des auteurs présente l'API FirecREST, une infrastructure d'API Web RESTful qui permet aux communautés scientifiques d'accéder aux divers services et ressources intégrés disponibles sur les systèmes HPC. Rappelons qu'une interface RESTful est une interface de programmation d'application qui fait appel à des requêtes HTTP pour obtenir (GET), placer (PUT), publier (POST) et supprimer (DELETE) des données. Les capacités de FirecREST ont été définies sur la base des exigences de cas d'utilisation décrits dans l'article.

Le cœur de l'article est donc une présentation de l'architecture et les capacités fonctionnelles de l'API FirecREST. Avant de décrire l'API, les auteurs présentent aussi les cas d'utilisation qui ont motivé les exigences. Ainsi les auteurs ont identifié trois exigences majeures (a) la nécessité pour l'API de s'intégrer à divers fournisseurs d'identité externes au centre de calcul. En dehors des services internes tels que le calcul interactif, les utilisa-

teurs n'ont pas nécessairement de comptes dans le centre HPC ciblé. Cette caractéristique dépend de la politique globale de gestion de l'accès à l'identité du centre et de l'utilisation de protocoles d'authentification standard ; (b) la capacité d'exécuter des charges de travail sur des systèmes HPC ; (c) la capacité d'effectuer des transferts externes de données vers/depuis les systèmes de fichiers du centre attachés au système HPC.

Pour terminer, revenons au premier Workshop SuperCompCloud, de 2019, qui a lancé la thématique de la convergence HPC et Cloud. Rappelons ici que le terme « convergence » ne signifie pas que le Cloud absorbe le HPC, mais plutôt un rapprochement entre le meilleur des deux mondes. Dans cette édition de 2019, nous trouvons la présentation intitulée « Scalability and data security : deep learning with health data on future HPC platforms » [86] qui est clairement ancrée dans domaine applicatif mêlant l'IA et le Cloud à des fins de santé. L'analyse des données de santé à grande échelle présente en effet plusieurs défis pour les environnements HPC classiques. Les ensembles de données contiennent des informations personnelles sur la santé et sont régulièrement mis à jour, ce qui complique l'accès aux données sur les systèmes HPC accessibles au public. En outre, du côté de l'IA, l'ensemble diversifié des techniques pour les tâches à accomplir et les modèles - allant des réseaux neuronaux pour l'extraction d'informations aux bases de connaissances pour la modélisation prédictive - a des échelles, des préférences matérielles et des exigences logicielles qui varient considérablement, constitue un système complexe.

Les systèmes exascales et les environnements de Cloud ont la possibilité de jouer un rôle important en assurant la sécurité des données et la portabilité des performances. Les plateformes en nuage fournissent des solutions prêtes à l'emploi pour maintenir la sécurité des données, tandis que des travaux récents ont étendu les environnements informatiques sécurisés à des systèmes tels que l'OLCF Summit. Dans son exposé, l'auteur explique comment son groupe de recherche répond au besoin de ressources HPC évolutives et aux exigences de sécurité des données inhérentes au traitement des informations personnelles sur la santé, dans le cadre du partenariat inter-agences entre le ministère de l'énergie et l'Institut national du cancer, aux États-Unis. Dans le cadre de ce partenariat, le groupe a développé des modèles d'apprentissage profond pour extraire des informations des rapports de pathologie cancéreuse afin d'établir des rapports sur l'incidence du cancer en temps quasi réel.

7.5 Ordonnancement de conteneurs

Nous présentons maintenant le travail [87] auquel nous avons participé. Le travail a été piloté par Tarek Menouer et les idées de la partie expérimentale ont été reprises pour certaines de nos expérimentations ultérieures. Voir à ce sujet le chapitre 9 intitulé « Conteneurisation avec Slurm ».

Le titre de l'article contient le terme *namespaces* (espace de nommage) mais il ne s'agit pas ici du concept lié à l'isolation dont nous avons préalablement parlé. Dans notre étude, un espace de nommage et une notion générique qui peut être interprétée de plusieurs façons. Par exemple, nous regroupons les conteneurs de la même application dans le même espace de nommage. Nous pouvons également regrouper les conteneurs qui appartiennent

à une ou plusieurs applications mais d'un même utilisateur dans le même espace de nommage. Notre étude suppose qu'un expert connaît le regroupement pertinent à satisfaire, et nous proposons un moyen de regrouper les conteneurs et de les ordonnancer.

La majorité des stratégies d'ordonnancement des conteneurs ne tiennent pas compte de la quantité de données transmises entre les conteneurs. L'article présente une nouvelle stratégie d'ordonnancement des conteneurs qui regroupe automatiquement les conteneurs appartenant au même groupe (*Namespace*) sur le même nœud. L'objectif est de compacter les nœuds avec des conteneurs du même groupe afin de réduire le nombre de nœuds utilisés, les coûts de communication entre les nœuds et d'améliorer la qualité de service (QoS) globale des applications conteneurisées.

L'ordonnancement est une heuristique qui fonctionne de la manière suivante. Chaque fois qu'un nouveau conteneur est soumis, notre approche sélectionne, parmi tous les nœuds de l'infrastructure en nuage, le nœud ayant le plus grand nombre de conteneurs appartenant à la même application afin de réduire les coûts de communication entre conteneurs. Notre politique compacte également les nœuds avec des conteneurs afin de réduire le nombre de nœuds de l'infrastructure en nuage utilisés. Il y a donc deux étapes, séquentielles, dans l'heuristique.

La stratégie proposée est mise en œuvre dans le cadre de Kubernetes. Des expériences démontrent le potentiel de notre stratégie dans différents scénarios. Plus important encore, nous montrons d'abord que la cohabitation entre notre nouvelle stratégie d'ordonnancement et la stratégie par défaut de Kubernetes est possible et bénéfique pour le système. Grâce aux espaces de noms, la cohabitation n'est pas limitée à deux méthodes d'ordonnancement des travaux par lots ou des services en ligne. Enfin, grâce à l'automatisation du déploiement, nous démontrons également que plusieurs clusters Slurm peuvent être instanciés à partir d'un pool de nœuds *bare-metal*. Cette réalité contribue au concept de « HPC as a Service ».

Pour conclure, ce chapitre illustre certains travaux de la communauté HPC dans le cloud et réalise des parallèles, quand cela est possible, avec notre sujet. L'idée est de montrer au lecteur que cette communauté est active sur les plans des applications scientifiques (*workflows* ou de bioinformatique), des communications rapides et sûres, de convivialité et de programmabilité, et enfin d'ordonnancement des conteneurs. L'idée est aussi de convaincre le lecteur que le paysage du HPC dans le cloud est vaste. Nous le visitons sous l'angle système.

CHAPITRE 8

ÉTAT DE L'ART ET POSITIONNEMENT DU TRAVAIL

Contrairement au chapitre précédent qui illustrait un large éventail de travaux généraux de la communauté « HPC dans le Cloud » à travers une vue macroscopique, dans ce chapitre, nous spécialisons notre positionnement scientifique (vue microscopique) vis à vis de travaux scientifiquement proches des nôtres. Cette section nous fait avancer à la dernière étape de la présentation des notions autour de notre travail comme montré dans la figure 8.1.

Nous allons commencer par présenter les intersections entre le monde de la conteneurisation et les ordonnanceurs HPC. Il existe d'ores et déjà des liens entre les deux thématiques sous forme de moteurs de conteneurs adaptés aux contraintes du HPC. Ceux-ci sont au nombre de trois : Singularity, Charliecloud et Shifter. Il est important de les présenter car ce sont des briques assez récurrentes dans tous les travaux sur la conteneurisation des travaux HPC comme nous le verrons dans la section 8.1.

Cependant, dès qu'on élargit la thématique au Cloud, il devient important de considérer un autre moteur de conteneur qui domine le monde du Cloud : Docker. Comme nous l'avons vu dans la section 4.4.3, le moteur Docker est assez complet. Il intègre notamment des fonctions de virtualisation réseau qui montent encore plus en complexité lorsqu'on lui ajoute de l'orchestration. Si nous utilisons les couches réseau standards des moteurs de conteneurs, cette complexité à un coût pour les communications réseau. Cet état de fait impacte directement les calculs à mémoire distribuée comme nous le verrons dans la section 8.2. Ainsi, si nous voulons continuer sur le sujet, il faut accepter que Docker soit considéré. On notera qu'il existe cependant des travaux pour annuler ce coût dans les grilles de calcul et qui pourraient être réutilisés dans le Cloud [88].

Dans la section 8.3 qui suit, nous allons passer en revue les interfaçages du HPC dans le Cloud. Nous avons distingué deux grandes tendances. La première concerne les ordonnanceurs HPC. L'idée principale est de leur faire gérer les travaux à la mode Cloud sur des moteurs type Docker. La seconde tendance est d'embarquer un *middleware* entre l'ordonnanceur HPC et un orchestrateur pour aller piocher dans le Cloud des ressources

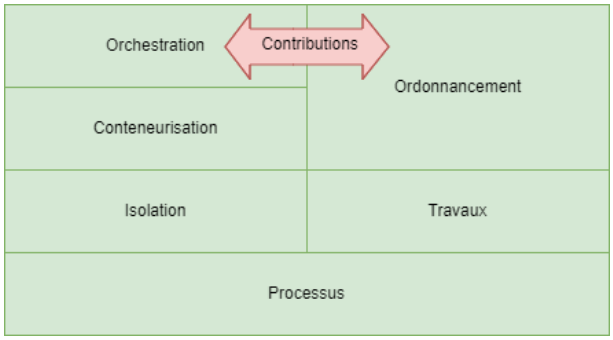


Figure 8.1 – Organisation et dépendances des notions

complémentaires. Ensuite, qui dit Cloud dit forcément provisionnement. Le provisionnement sera discuté dans la section 8.4 car c'est un élément essentiel de toute stratégie de gestion élastique des ressources car il s'agit d'amener les environnements d'exécution sur les ressources. Enfin nous préciserons quelques éléments essentiels sur la cohabitation dans la section 8.5. La cohabitation consiste à permettre à différentes entités de partager la même infrastructure. Chaque entité a évidemment des attentes différentes.

8.1 Interfaçages entre les ordonnanceurs et les moteurs de conteneurs

Avec l'adoption des conteneurs, il est tentant pour les utilisateurs d'infrastructures HPC de mettre à profit ces propriétés d'isolation pour créer ce qu'on appelle des UDSS pour « *User Defined Software Stack* », soit des piles logicielles paramétrables par les utilisateurs. Les avantages sont les mêmes que pour les images des conteneurs (cf. section 4.4, pour mémoire itinérance et reproductibilité). Cependant, certains moteurs de conteneurs sont plus adaptés que d'autres dans un contexte HPC. En effet, Docker et son démon `dockerd` qui tourne avec les droits administrateurs sur la machine pose de sérieux problèmes en termes de sécurité. Ces services qui tournent avec l'identité de l'administrateur offrent des moyens d'élever les privilèges du conteneur sur les nœuds de calcul. Cela constitue donc un problème en terme de sécurité car l'utilisateur peut se retrouver administrateur du noeud de calcul via l'utilisation de conteneurs privilégiés. Il existe cependant des alternatives sécurisées à Docker comme 1) Socker [89], un *wrapper* pour Docker utilisable par les ordonnanceurs HPC, 2) Udocker [90] (un Docker en espace utilisateur) et 3) rkt [1] qui malgré des fonctions prometteuses pour le HPC [91] est arrêté.

Des alternatives ont été développées spécifiquement pour le monde du HPC (même si elles peuvent être utilisées dans un autre contexte). Ces alternatives s'appuient sur le fait que l'exécution d'un travail lancé par un ordonnanceur (cf. section 3.2.3) n'a besoin que d'un petit sous-ensembles d'espaces de noms (contrairement à Docker). Ces alternatives fonctionnent en mode sans démons (*daemonless*). En réalité, l'exécution d'un travail a juste besoin de déplacer la racine du processus vers le système de fichier associé à l'image

1. https ://github.com/rkt/rkt

du logiciel utilisé et d'accéder à certains périphériques. Finalement, rien que la combinaison de l'utilisation des espaces de noms `user_ns` et `mount_ns` (cf. section 4.3) et quelques « *bind mount* » ne puissent résoudre.

Nous allons discuter trois moteurs de conteneurs orientés pour cet usage [92] : Charliecloud qui est le plus minimal, Singularity qui propose plus de fonctionnalités et Shifter qui présente une originalité dans la gestion des images.

8.1.1 Charliecloud

Charliecloud [93] est le moteur de conteneur sans démon le plus minimal. Il est supporté par le LANL (*Los Alamos National Laboratory*). Il comprend quelques centaines de lignes de C pour le lanceur ch-run (équivalent de runC, cf. section 4.4.3) et à peine plus d'un millier de lignes de Python pour les *wrappers* divers (notamment ch-image pour la manipulation des images). La partie ch-image n'est pas vraiment pertinente pour notre sujet, nous allons par contre discuter le fonctionnement de ch-run qui exécute un programme dans le contexte de son image.

La figure 8.2 présente le fonctionnement de l'exécutable ch-run qui invoque un exécutable nommé « app ». Il commence par invoquer un `unshare()` (cf. section 4.3) afin de créer un espace de nom `user_ns` dédié à notre programme. A partir de ce moment là, le programme tourne dans son propre contexte par rapport aux UID. Il est maître dans son contexte. En revanche, dès qu'il doit agir avec l'extérieur (c'est-à-dire avec des objets de l'espace de nom original), sont UID est associé avec le EUID de l'appelant. Dans un contexte HPC, l'EUID est l'UID de l'utilisateur banalisé qui lance son code. Ainsi, le programme lancé via ch-run ne peut pas escalader ses privilèges sur le système hôte.

Ensuite, ch-run effectue une série de « *bind mount* » afin de monter l'image et certains répertoires de l'hôte à l'intérieur (par exemple, le répertoire utilisateur). Une fois le système de fichier de l'image en place, ch-run invoque `pivot_root()` (cf. section 4.4.2) pour changer la racine du processus courant. Enfin il invoque `execvp()` pour recouvrir son code courant avec celui de « app » (cf. annexe A.9).

8.1.2 Singularity / Apptainer

Singularity [94] est maintenu depuis 2017 par la société Sylabs qui continue à proposer le produit en libre sous le nom de SingularityCE. Il est beaucoup plus populaire que Charliecloud et notamment déployé a l'ALCF (*Argonne Leadership Computing Facility*), OLCF (*Oak Ridge Leadership Computing Facility*), NERSC (*National Energy Research Scientific Computing Center*) et sur le cluster Perlmutter, au LLNL et à Sandia. En 2021, un *fork* est créé pour des raisons idéologiques et une version amputée des greffons commerciaux ajoutés par Sylabs voit le jour sous le nom de Apptainer.

Nous parlerons d'Apptainer dans la suite de la discussion. Apptainer est construit sur le même modèle que Charliecloud. Il est « *daemonless* » et se base sur le couple `user_ns`

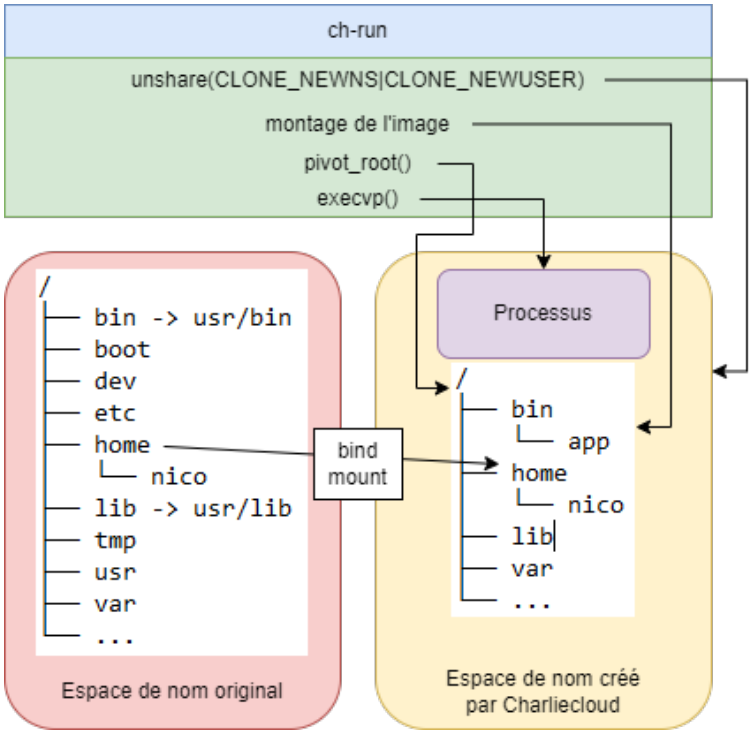


Figure 8.2 – Architecture de Charliecloud

et `mount_ns` pour isoler l'exécution de programmes dans un contexte de systèmes partagés (au sens multi-utilisateurs) tels que les clusters HPC. Une différence avec Charliecloud est que Apptainer est plus intégré aux clusters HPC. Il propose des interfaces d'intégration avec les principaux ordonnanceurs HPC disponibles. L'accès aux périphériques type GPU est également plus simple soit directement en CLI (*Command Line Interface*) ou via des fichiers de configuration (c'est également possible avec Charliecloud mais la configuration est plus manuelle).

L'exécutable apptainer qui lance les conteneurs peut avoir le bit SUID (SetUID) positionné ou pas. Ce bit fait partie de ce que l'on appelle les bits spéciaux qui viennent compléter le système UGO (cf. annexe A.11) en ajoutant des droits supplémentaires. SUID en est un et se place sur les fichiers exécutables. Avec ce bit positionné, l'exécutable tournera avec l'identité de son possesseur et non celle du lanceur ce qui ouvre de nombreuses possibilités dans son utilisation. Utilisé en mode SUID les conteneurs créés par Apptainer pourront acquérir des capacités supplémentaires (cf. annexe A.12), utiliser des images chiffrées (en s'appuyant sur LUKS2), autoriser les bits SUID sur les exécutables à l'intérieur de l'image, appliquer des politiques appArmor, seccomp ou SELinux ou encore utiliser d'autres espaces de noms que `user_ns` et `mount_ns`.

Enfin, Apptainer peut s'appuyer sur les cgroups (cf. section 4.2) pour limiter l'utilisation de ressources du conteneur. Charliecloud ne propose pas cette fonctionnalité dans la mesure ou c'est de la responsabilité de l'ordonnanceur HPC de limiter les ressources utilisées par les travaux en exécution. C'est l'agent de l'ordonnanceur qui doit créer la hiérarchie du cgroup ou placer le travail.

En conclusion, avec l'utilisation du bit SUID et au prix d'une surface de code (donc

d'attaque) beaucoup plus conséquente, Apptainer est plus versatile que Charliecloud. Le bit SUID permet de donner des privilèges au conteneur et également d'appliquer des mécanismes de sécurité pour les limiter. L'ambition de Apptainer est de proposer une alternative complète « *daemonless* » à Docker.

8.1.3 Shifter

La force de Shifter [95] en plus de son aspect assez minimal est l'optimisation de l'utilisation des images. Il propose de convertir les images dans un format qui lui est propre nommé UDI (*User Defined Images*). Ce format met à plat les couches des images standard Docker ce qui permet de les monter localement via Squashfs et donc de faire porter toutes les opérations sur les métadonnées des fichiers par le nœud local même dans un contexte de système de fichier distribué pour limiter les surcoûts [96]. A l'instar des deux protagonistes précédents, Shifter utilise l'espace de nom `user_ns` pour réaliser ce montage.

8.1.4 Performance de ces moteurs de conteneurs HPC

L'impact sur les performances de ce type de conteneurisation dans le HPC est inexistant. Dans [97], les auteurs démontrent que les moteurs de conteneurs HPC susmentionnés n'ont quasiment aucun impact sur les performances CPU et mémoire des travaux HPC à mémoire partagée ou distribuée. Ils ont éprouvé ces différents moteurs avec les *benchmarks* industriels standards à savoir SysBench, STREAM et HPCG. La seule très légère variation concerne l'empreinte mémoire. Leur conclusion est que les applications peuvent être conteneurisées (avec ce type de moteurs taillés pour le HPC) sans trop se soucier des impacts sur les performances.

Avec le travail que nous avons réalisé dans la section 4.3, notamment sur les structures de données dans le noyau des espaces de noms, le résultat de leur travail n'est pas surprenant. En effet, même si nous ne les utilisons pas de façon consciente, à partir du moment où les espaces de noms sont activés dans le noyau, ils sont évalués lors des interactions du processus. Par défaut, les espaces de noms sont activés dans les distributions Linux. Dans la section suivantes nous allons voir quelques problèmes relatifs à l'utilisation de MPI dans le code conteneurisé.

8.2 La problématique MPI

Dans cette section, nous allons discuter des adaptations de la couche MPI aux conteneurs dans une optique HPC. Nous verrons que la prise en compte de MPI dans les conteneurs peut se faire de plusieurs manières en fonction de la topologie sur laquelle le travail à mémoire distribuée va s'exécuter.

Pour cette discussion, il faut distinguer deux configurations de moteurs de conteneurs

1) Soit ils utilisent le réseau de l'hôte, ce qui est le cas pour la famille des moteurs HPC discutés dans la section 8.1 2) Soit ils instancient un réseau virtuel permettant d'adresser les conteneurs individuellement via une pile réseau privée pour le processus isolé. Pour les deux configurations, il faudra adapter l'image du conteneur pour y placer la couche MPI en accord avec celle de l'hôte.

En revanche, pour l'impact sur la performance des communications, nous verrons que la question se pose pour la configuration 2), à savoir avec un réseau virtuel. Cette section se divisera donc en deux parties. La première présentera le travail réalisé autour de l'adaptation de MPI au contexte des conteneurs. La seconde présente l'impact de la virtualisation du réseau sur la performance des communications MPI inter-conteneurs.

8.2.1 Adaptation des travaux à mémoire distribuée aux conteneurs

Lorsqu'il s'agit d'adapter MPI au monde des conteneurs il faut distinguer deux cas de figure. Pour bien comprendre chacun des deux cas, il est nécessaire de faire un petit point sur le lancement des processus MPI. Le principe est qu'un maître récupère la liste des nœuds et s'y connecte pour lancer les processus MPI. La connexion peut se faire de plusieurs manières. Si on utilise un ordonnanceur, il peut prendre en charge cette tâche. Sinon, l'implémentation de MPI peut avoir un serveur qui écoute sur le réseau. Enfin, le moyen le plus courant est de recourir à SSH. Il faut donc un serveur sshd en écoute sur le nœud et que le maître MPI puisse s'y connecter de façon non interactive (souvent par clés). Lorsque l'on transpose cette observation au monde des conteneurs, nous obtenons les cas de figure de suivants :

— Cas 1 : Les processus MPI s'exécutent dans des conteneurs déjà provisionnés en exécution sur une infrastructure de Cloud avec leur propre configuration réseau. Dans ce cas, il faut rendre le conteneur adressable par le lanceur MPI (mpiexec) ;
— Cas 2 : Les processus MPI tournent dans un conteneur HPC instancié par un ordonnanceur HPC comme ceux de la section 8.1. Dans ce cas, le défi est d'aligner la version de MPI du conteneur avec celle de l'environnement d'exécution.

Cas 1 : Dans [98], l'objectif des auteurs est de définir un modèle pour intégrer des applications parallèles MPI dans des *workflows* DevOps permettant de transporter une application encapsulée dans des conteneurs Singularity et Docker sur différentes plateformes matérielles et virtuelles. Leur but est d'évaluer la portabilité d'une application afin de tester ses propriétés dans différents environnements. Premièrement, ils ont testé la combinaison conteneur Singularity / Cray XC-series. Ensuite, ils ont testé le couple Docker / EC2 (*Elastic Compute Cloud*). Les valeurs références sont celles de l'exécution du code *bare-metal* compilé avec la librairie MPI Cray. Sans surprise, le code conteneurisé avec Singularity et compilé avec la librairie MPI Cray obtient les mêmes performances que la référence. De plus, en dépit de performances moindres, l'expérimentation sur EC2 a mis en lumière que cette plateforme est très intéressante pour tester le code, notamment son comportement lorsqu'il passe à l'échelle.

Dans [99], les auteurs amènent la réflexion que si la performance est recherchée, alors les conteneurs ne peuvent être agnostiques par rapport aux composants logiciels de la

couche du dessous. Les auteurs proposent de s'appuyer sur le mécanisme de labels pour gérer différentes saveurs (*flavors*) d'images en fonction de la configuration du système hôte sur lequel le conteneur va s'exécuter. Ils proposent également d'ajouter un « *hook* » au lancement pour sélectionner l'image appropriée par rapport au système d'accueil. Ce modèle d'optimisation des images demande un peu plus de travail au développeur et casse le modèle d'indépendance des couches. Ce travail est dans la continuité de leur contribution précédente [98] autour de la personnalisation des images. une fois de plus, le gros du travail est d'aligner le contexte MPI du conteneur avec celui de l'environnement d'exécution.

Dans [100], les auteurs présentent une étude assez complète de l'intégration de MPI dans des conteneurs Singularity dans le cadre de PRACE (*Partnership for Advanced Computer in Europe*). Leur conclusion est que la principale difficulté est d'aligner la configuration MPI du conteneurs avec les capacités de support du moteur. Une fois passé cette difficulté, il montrent que les performances des codes conteneurisés sont très similaires à une exécution native.

Cas 2 : Dans [101], les auteurs présentent une intégration de MPI en créant un cluster conteneurisé dédié à MPI. Ils commencent par créer une interface réseau de type pont sur laquelle ils demandent au moteur de conteneur d'y attacher les conteneurs. Ensuite, ils créent une image Docker contenant les librairies MPI ainsi qu'un serveur SSH qui sera le processus conteneurisé. A leur démarrage, les nœuds s'enregistrent sur un service nommé Consul [2] chargé de maintenir l'inventaire des nœuds MPI conteneurisés à destination du processus maître MPI. L'utilisateur peut ensuite lancer ses programmes MPI sur cette infrastructure. Ce travail propose donc une intégration de MPI intrusive au niveau de l'infrastructure mais transparente pour les utilisateurs. L'inconvénient est que même lorsque les conteneurs MPI ne font rien, ils sont en exécution et occupent donc des ressources inutilisées.

L'article [102] apporte une solution au problème de ressources gâchées du travail précédent. Les auteurs proposent un *wrapper* pour le lanceur mpiexec de MPICH permettant de définir une image Docker sur laquelle lancer leurs travaux à mémoire distribuée. Les conteneurs MPI sont donc instanciés à la demande sur l'infrastructure. Contrairement à [101], les conteneurs ne lancent pas un serveur SSH mais un service *hydra_pmi_proxy*. Les auteurs indiquent que la partie équilibrage de charge des conteneurs sur les nœuds est prise en charge par Docker Swarm. Cependant, ils donnent très peu de détails sur le procédé de localisation des conteneurs MPI par le maître. On notera qu'il existe une approche assez similaire pour OpenMPI nommée *dssh* [3].

8.2.2 Impact des réseaux virtualisés sur les code à mémoire distribuée

Dans [103], les auteurs cherchent à mesurer l'impact des moteurs de conteneurs sur différents *benchmarks*. Ils font varier le nombre de nœuds et de conteneurs par nœud mais aussi les versions des noyaux. Les auteurs mentionnent que la couche réseau instanciée

2. https ://www.consul.io
3. https ://qnib.org/blog/2016/03/31/dssh-proof-of-concept-for-a-ssh-less-docker-native-mpi

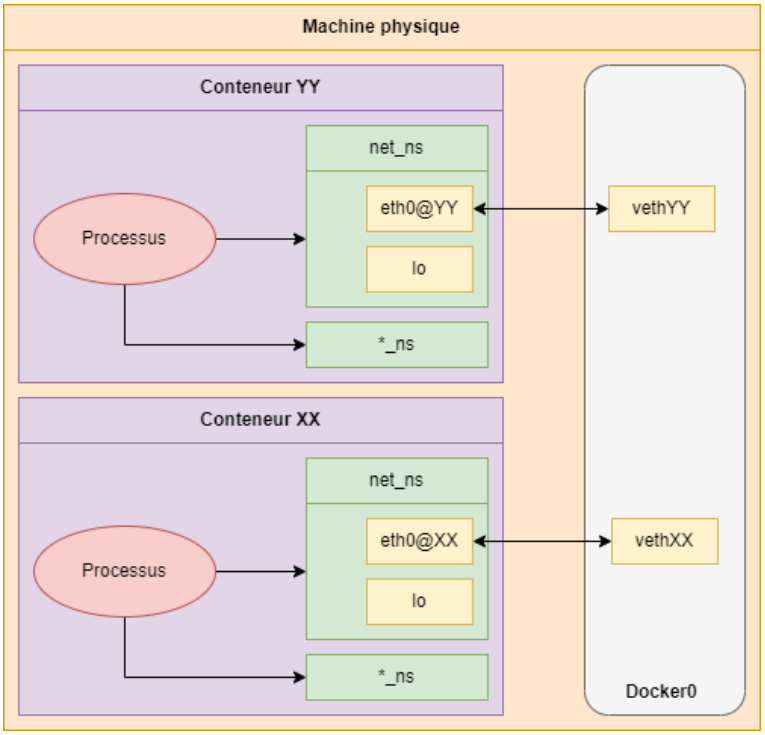


Figure 8.3 – Réseau virtuel dans Docker

par le moteur de conteneurs pénalise les communications MPI. Une étude complémentaire [104], confronte plusieurs moteurs de conteneurs à savoir LXC, Docker et Singularity. Cette étude montre que Singularity s'approche des performances natives tandis que LXC et Docker ont une dégradation des performances dès que le réseau est impliqué du fait de leur recours à l'espace de noms `net_ns` et à la couche réseau virtuelle mise en place pour joindre la pile réseau privée résultante comme le montre la figure 8.3. Pour Docker, ils mettent également en évidence que les entrées / sorties sur disque sont également impactées par la gestion des images en couche via AUFS (*Advanced multi layered Unification FileSystem*).

Dans [105] les auteurs proposent une étude récente (2022) et très complète sur les impacts réseau en TCP/IP sur Ethernet traditionnel mais ils étendent leur étude avec l'exploration des performance d'Infiniband en mode RDMA (*Remote Direct Memory Access*) et IPoIB (*Internet Protocol over InfiniBand*). Ils évaluent à la fois Singularity et Docker. Pour Docker, ils testent un certain nombre de configuration réseau (Host, Overlay et MACVLAN). Lorsqu'ils sont en mode multi-conteneurs sur une machine (c'est-à-dire que plusieurs conteneurs tournent sur la machine, ce qui implique une couche réseau virtuelle pour les adresser individuellement via leur `net_ns` privé), ils appliquent des limitations de ressources via les cgroups 4.2. Les auteurs passent plusieurs *benchmarks* sur ces combinaisons de configuration : OSU [4] ainsi que ceux de la suite HPCC [106]. Leur conclusion est qu'avec singularity il n y a aucun impact, en revanche Docker coûte cher (jusqu'à 70% de latence en plus) dans un contexte multi-conteneurs sauf en RDMA, ce qui est logique car le réseau virtuel est juste utilisé pour la phase d'instanciation.

4. https ://mvapich.cse.ohio-state.edu/benchmarks

8.3 Passerelles pour le HPC dans le Cloud

Dans cette section, nous allons voir comment des ordonnanceurs HPC peuvent être connectés à des infrastructures Cloud pour y lancer des travaux conteneurisés La problématique est différente de la section précédente 8.1. En effet, il s'agissait d'encapsuler un travail dans un conteneur ne proposant aucune virtualisation du réseau dont l'essentiel bénéfice était d'être capable de tirer parti des images de logiciels scientifiques pour les utiliser sur un cluster HPC.

Ici, nous parlons d'ordonnanceurs qui vont eux même instancier des conteneurs pour les travaux HPC sur des moteurs proposant une virtualisation du réseau. En conséquence, comme les conteneurs sont placés sur un réseau IP routable, cela ouvre la possibilité au centre de calcul de déborder sur des fournisseurs de Cloud. Dans ce cas, c'est bien l'ordonnanceur HPC qui crée les conteneurs. Une autre approche est de passer la main à un orchestrateur pour s'occuper de provisionner les conteneurs pour les travaux HPC. Dans ce cas, un connecteur est requis.

8.3.1 Intégration des problématiques Cloud dans les ordonnanceurs HPC

Avec Spectrum LSF Suites [5], IBM a décidé de développer une solution de gestion de travaux générique adaptée à l'instanciation de travaux HPC, machines virtuelles, conteneurs etc. LSF se connecte avec les principaux moteurs de conteneurs : Docker, Shifter ou Singularity. Il gère le provisionnement des images et machines virtuelles ainsi que les étapes de construction, test et livraison des applications. Ainsi, LSF est une application complète qui n'est pas réservée aux conteneurs. Le fait que LSF gère tout le cycle de vie du *workload* du provisionnement jusqu'à l'exécution lui permet de supporter le « burst » vers le Cloud [107]. Le « *burst* » consiste à permettre au travail de prendre plus de ressources durant son exécution pour les relâcher par la suite. En effet, avec la gestion actuelle par les ordonnanceurs HPC, une quantité fixe de ressources est assignée au travail pour toute sa durée de vie. En général, le « *burst* » consiste à aller chercher des ressources dans un Cloud public depuis un Cloud privé donc dans un contexte de Cloud hybride.

Dans [58], les auteurs présentent une méthode combinant l'ordonnanceur HPC PBS et un script maison afin de déployer des travaux conteneurisés sur des nœuds exécutant le moteur de conteneur Docker. Leur script se charge de démarrer le service dockerd en préambule de l'exécution de leur travail. Ils présentent deux approches pour l'exécution du travail à mémoire distribuée : 1) un conteneur par machine exécutant tous les processus MPI et 2) un conteneur par processus MPI. Ils évaluent leur approche avec un *benchmark* LINPACK lancé respectivement dans un contexte physique ou virtualisé avec KVM pour chaque scénario de conteneurisation. La conclusion est que les écarts de performances sont minimes excepté lorsque l'on exécute un LINPACK optimisé pour Intel sur KVM.

5. https ://www.ibm.com/downloads/cas/VEO91OVO

8.3.2 Ponts entre les ordonnanceurs HPC et les orchestrateurs

Dans [108], les auteurs présentent une utilisation originale de l'orchestrateur Kubernetes. Celui-ci sert à agréger des clusters HPC basés sur Slurm. Les auteurs ont implémentés un type de travail *SlurmJob* permettant à Kubernetes d'envoyer le travail isolé dans un conteneur Singularity à un des clusters HPC Slurm disponible. Finalement, dans cette architecture, Kubernetes fait office de gestionnaire de grilles de calcul [109]. Cependant, le cluster HPC est un type de ressource comme un autre pour l'orchestrateur Kubernetes. Il n'est pas limité à l'orchestration de cluster HPC, cela en fait un gestionnaire de grilles plus polyvalent (mais moins expert sur les spécificités HPC).

Dans [110], les auteurs ajoutent un module nommé Torque-Operator à l'ordonnanceur HPC du même nom. Ce module est un pont entre un cluster Big Data géré par un orchestrateur Kubernetes et un cluster HPC traditionnel piloté par Torque. Il s'exécute sur le frontal du cluster HPC. Le frontal HPC est également un nœud Kubernetes. L'idée est que tous les travaux sont soumis sur le cluster Big Data Kubernetes. L'utilisateur peut encapsuler un script Torque dans le YAML de son déploiement Kubernetes. Dans ce cas, le script est passé au module Torque-Operator qui soumet le travail au cluster HPC.

Dans [111] les auteurs sortent de l'évaluation par des *benchmarks* pour se concentrer sur la conteneurisation de simulation en biologie. A la différence des travaux précédents, cette fois ci, ce ne sont pas des *benchmarks* bien connus qui sont passés mais des codes réels. Ils évaluent les performances de leurs codes CFD et FSI conteneurisés avec Docker, Singularity et Shifter. Les évaluations sont déclinées sur quatre clusters HPC possédants des architectures processeurs différentes (Intel Skylake, IBM Power9 et Arm-v8) avec une échelle allant jusqu'à 256 nœuds pour 12K cœurs. Leur conclusion est que Singularity est le meilleur choix concernant les performances.

Le papier [112] compare les différentes couches de virtualisation réseau des moteurs de conteneurs. Ils font tourner leurs conteneurs dans des machines virtuelles et physiques. Ils confirment le surcoût de la conteneurisation pour les communications MPI en mettant en évidence que le plus adapté est Calico en mode BGP car il évite le recours au NAT qui est le pire cas [113].

8.4 Provisionnement

Le provisionnement est un élément essentiel dans les stratégies de passage à l'échelle. En effet, une mise à disposition efficace des environnements d'exécution est une brique indispensable dans toutes les stratégies de déploiement à la demande. Cependant, la vitesse n'est pas le seul facteur. Il y a également la commodité de configuration, la transportabilité de l'environnement et la mise à disposition des images.

Dans [114], les auteurs présentent une méthode de création d'un cluster HPC dédié avec PBS pour un utilisateur en piochant dans les nœuds d'une grille hétérogène. L'idée est que l'utilisateur demande l'instanciation d'un cluster à son usage exclusif en piochant dans les ressources libres des différents participants de la grille. L'utilisateur peut alors

y lancer ses travaux. D'après les résultats de cette étude, il semble que cette approche permette d'instancier assez rapidement des petits clusters HPC privés, ce qui optimise l'utilisation des ressources globales de la grille. Dans cette approche, la granularité de provisionnement est le nœud physique. Cette approche ne s'appuie sur aucune brique d'isolation des processus. Ici, PBS passe la main à Globus pour identifier un serveur physique libre et pour déployer l'agent de PBS dessus afin de le recruter dans le cluster.

Dans [115], les auteurs présentent une infrastructure dans laquelle les ordonnanceurs HPC (Moab dans leur expérimentation) communiquent avec un système de provisionnement (OpenStack dans leur expérimentation) pour déployer des machines virtuelles embarquant un code de physique spécifique. L'utilisateur demande le déploiement de ce cluster virtualisé au sein de son travail. A la fin du travail, le nœud est libéré pour accueillir tout autre type de travail.

Kadeploy [116] développé à l'INRIA permet également de déployer de la machine virtuelle [117] ou physique [116] en vue d'y exécuter des expérimentations type travaux en mode batch ou des manipulations en mode interactif. Il est couplé avec Oar [52] qui intègre les problématiques de provisionnement directement dans le fichier de soumission. Ainsi, l'utilisateur peut instancier son environnement expérimental sur la machine physique avant d'y exécuter son travail. On notera que le déploiement s'appuie sur PXE (*Pre-boot eXecution Environment*), les nœuds doivent donc être sur le même domaine de *broadcast* Ethernet.

Dans [118], les auteurs présentent SEC (*Semi-Elastic Cluster*). SEC propose d'adapter la taille d'un cluster HPC en instanciant de nouveau nœuds dans un Cloud externe en fonction des données de consommation de ressources des travaux passés. Le but est de dégager une tendance pour y adapter la quantité de ressources disponible. En conséquence, ce système de passage à l'échelle est non intrusif par rapport à l'ordonnanceur HPC (Slurm dans leur expérimentation). Les auteurs vont piocher dans différents fournisseurs d'infrastructures Cloud : EC2 et GCE. D'après les spécifications de l'offre IaaS de ces deux fournisseurs, on devine que les nœuds Slurm ajoutés de cette façon sont des machines virtuelles.

8.5 Cohabitation

La cohabitation était un des objectifs de notre travail. Cependant quand on parle de cohabitation il faut bien faire la distinction entre deux problèmes distincts : 1) Faire cohabiter des systèmes distincts gérés par différents utilisateurs sur la même infrastructure 2) Faire cohabiter des travaux aux paradigmes différents sur la même infrastructure. Ces deux problèmes ne s'excluent pas, au contraire ils se complètent. La cohabitation de systèmes (problème 1) permet d'avoir une bonne agilité au niveau de la granularité des ressources mise à disposition, tandis que la gestion des travaux aux besoins différents (problème 2) fait un usage optimisé de ces ressources en fonction des paradigmes du travail qui va les utiliser.

Le premier étage de cohabitation repose sur la virtualisation et la conteneurisation déjà abondamment discutée respectivement dans les sections 4.1 et 4.4. Nous allons ici

nous intéresser au second problème en évoquant Mesos, YARN et Omega.

L'objectif de Mesos [119] est de permettre le partage d'une infrastructure par divers systèmes de gestions de travaux distribués. Mesos propose deux services complémentaires. D'une part, il propose un langage de haut niveau permettant de modéliser le *framework* à satisfaire. D'autre part, il dispose d'un service permettant de piloter les nœuds de l'infrastructure. En effet, Mesos possède un service coordinateur et des agents déployés sur chaque nœud de l'infrastructure. Ainsi, lorsqu'une demande de travail est formulée depuis le *framework* pris en charge par Mesos, le service coordinateur consulte ses agents sur les machines pour formuler des offres de ressources (*resource offers*). Lorsque le *framework* reçoit les offres de Mesos, il en accepte une et envoie à Mesos une description de la tâche à exécuter. Mesos envoie ensuite le travail à ses agents sur les nœuds participant au travail soumis par le *framework*.

Cette architecture fait de Mesos un ordonnanceur à deux étages. Il existe effectivement quatre types d'ordonnanceurs :

— Monolithiques : par exemple l'ordonnanceur du noyau Linux. Il ne manipule que des structures `task_struct` sur un ensemble de ressources fixes ;
— A partitions statiques : c'est typiquement le cas des ordonnanceurs HPC actuels. Les ressources sont divisées en partitions distinctes dédiées à certains types de travaux (distribués, SMP, GPU etc.). L'ordonnanceur place donc les travaux sur les ressources des partitions en fonction de leurs types ;
— A deux étages : c'est la mécanique que nous avons détaillé avec Mesos ;
— A état partagé : c'est le cas d'Omega que nous allons maintenant examiner.

Omega [120] est donc un ordonnanceur à état partagé développé par Google. Le fonctionnement est comparable à la famille à deux étages excepté que le *framework* pris en charge a une vue de l'intégralité des ressources. Il formule donc sa demande en fonction de cette vision globale. Omega se comporte comme un arbitre qui se contente d'accepter ou refuser la demande du *framework* qui doit la réitérer jusqu'au succès. C'est pour cette raison qu'on parle de gestion optimiste des ressources pour les ordonnanceurs à état partagé et pessimiste pour ceux à deux étages. Il serait tentant de faire une comparaison entre Mesos et Omega mais comme le mentionne l'article [121], qui présente une étude entre les quatre types d'ordonnanceur, Omega est très lié aux usages de Google donc très difficile à évaluer dans un contexte plus généraliste.

YARN (*Yet Another Resource Negotiator*) [122] est, quant à lui, un ordonnanceur monolithique dédié au *framework* Hadoop [123]. Il est adapté aux longs travaux en mode batch. Il est donc disqualifié pour les services avec une exécution non bornée dans le temps (par exemple un service web) ou basés sur des requêtes interactives courtes. YARN et Mesos peuvent collaborer via Myriad [6]. Myriad est porté par des grandes sociétés comme eBay. Lorsqu'un travail arrive sur YARN, il est pris en charge par Myriad qui va demander à Mesos de lui faire une offre de ressources. Mesos va ensuite invoquer un agent Myriad sur les nœuds sélectionnés. L'agent Myriad va quand à lui lancer l'agent YARN. Ce dernier se manifeste ensuite au gestionnaire de ressource YARN qui va pouvoir consommer les ressources du nœud.

6. https ://incubator.apache.org/projects/myriad.html

8.6 Positionnement

Dans cette dernière section du chapitre, nous allons reprendre les points de l'état de l'art et préciser pour chacun d'eux comment nous nous positionnons par rapport au travail de la communauté. Nous discuterons donc de notre orientation scientifique par rapport au schéma classique de l'ordonnanceur invoquant des conteneurs spécialisés pour le HPC (cf. section 8.1). Nous verrons de quelle façon notre travail est impacté par les problématiques MPI (cf. section 8.2). Nous préciserons l'articulation que nous proposons entre les ordonnanceurs HPC et les orchestrateurs Cloud (cf. section 8.3). Nous nous positionnerons également par rapport aux *workflow* HPC instanciés directement sur le Cloud (7.2). Enfin, nous expliciterons ce que nous proposons au niveau du provisionnement (cf. section 8.4) et de la cohabitation (cf. section 9.3).

Ordonnanceurs et conteneurs HPC. Notre contribution vient en complément des infrastructures classiques. Comme nous le verrons dans la section 9.4.1, nous avons implémenté un scénario d'hybridation entre un cluster HPC classique disposant de nœuds de calcul physiques et notre approche à base de nœuds de calcul conteneurisés. Notre contribution vient donc en complément de ce qui existe actuellement.

La problématique MPI. Notre contribution est concernée par les soucis de performances dans la mesure où nous fournissons des nœuds de calcul conteneurisés qui doivent disposer de leur propre pile réseau. Nous nous appuyons donc sur un réseau virtuel tel que présenté dans la section 8.2.2. En revanche, nous n'avons pas de soucis particulier au niveau de l'environnement d'exécution MPI.

HPC dans le Cloud. Notre contribution utilise les infrastructures Cloud existantes pour y déployer des conteneurs simulant des nœuds de calcul. Ces nœuds de calcul conteneurisés embarquent l'agent d'ordonnanceurs HPC. Les objectifs sont 1) d'étendre les clusters HPC existants avec des nœuds dans le Cloud (cf. section 9.4.1) et 2) de fournir des clusters HPC complets conteneurisés sur des infrastructures Cloud (cf. section 9.4.2). Nous ne modifions ni l'ordonnanceur HPC ni l'orchestrateur de Cloud. Nous nous limitons également à n'utiliser que des configurations élémentaires pour transporter facilement nos environnements d'un système à un autre.

Provisionnement. Nous réalisons un provisionnement dynamique des nœuds de calcul conteneurisés en utilisant Kubernetes comme un *broker* entre l'ordonnanceur HPC et la plateforme de Cloud. Nous n'utilisons Kubernetes qu'à des fins de provisionnement pour supporter les changements d'échelle de nos clusters HPC conteneurisés tels que présentés dans le chapitre 10.

Cohabitation. Nous avons vu dans la section 8.5 que la cohabitation peut se passer à deux niveaux. Nous nous plaçons clairement dans le premier cas car nous n'utilisons pas de *framework* particulier, nous faisons cohabiter des nœuds de calculs avec d'autres services en mode conteneurs qui n'ont peut être rien à voir avec le HPC.

CHAPITRE 9

CONTENEURISATION AVEC SLURM

Ce travail de thèse est effectué au sein du pôle support à la recherche de l'USPN. Une des missions de ce pôle est de maintenir un cluster HPC en état de fonctionnement. Les premières années du pôle furent dédiés à l'installation du cluster ainsi qu'à la formation des utilisateurs. Une fois cette phase d'installation technique et organisationnelle passée, parmi les actions plus prospectives permises par l'assise du centre de calcul, il y avait le projet de proposer la mise à disposition de l'environnement HPC à des unités d'enseignement (master spécialisé en calcul scientifique, ingénieur instrumentation etc.) pour que chaque étudiant puisse se former.

Les deux possibilités étaient de proposer soit de la machine virtuelle, soit du conteneur. Dans la section 3.2.3, nous avons démontré que les processus composants les ordonnanceurs HPC fonctionnent sur le mode père / fils, c'est-à-dire que le travail lancé par l'utilisateur est le fils de l'agent de l'ordonnanceur sur le nœud de calcul. Dans la section 4.4, nous indiquons que la bonne pratique est d'avoir un père et ses fils dans un conteneur. Dans le contexte qui nous intéresse, le conteneur semble tout indiqué pour la distribution de l'environnement HPC pédagogique. En effet, les salles de travaux pratiques embarquent des machines plutôt orientées bureautiques n'ayant pas forcément la capacité de faire tourner un hyperviseur et ses machines virtuelles pour simuler un cluster HPC.

Cet environnement HPC pédagogique comprenait un ordonnanceur HPC accompagné de quatre nœuds de calcul conteneurisés. Pour ces cinq conteneurs, une orchestration locale très légère est suffisante. Il n'y avait aucune contrainte d'efficacité. Ce qui était privilégié était la reproductibilité et la facilité de distribution et d'installation. Dans la section 5.4 nous avons évoqué Docker-compose qui satisfait ces contraintes d'orchestration locale légère. Nous avons alors décidé d'aller plus loin dans cette approche en utilisant l'orchestrateur Kubernetes pour distribuer les conteneurs HPC sur plusieurs nœuds Cloud. Ce travail vise à évaluer la possibilité de profiter de certaines propriétés du Cloud dans les environnements HPC. Comme indiqué dans la section 3.1 sur l'historique du HPC, les machines sont de plus en plus denses en termes de ressources (essentiellement mémoire et CPU). En conséquence, nous souhaitons créer des noeuds de calcul conteneurisés pour,

d'une part, segmenter les ressources et les partager avec d'autres service que le calcul HPC (cohabitation) et, d'autre part, permettre d'accroître la reproductibilité des expérimentation en intégrant l'ordonnanceur HPC dans l'image de l'environnement expérimental. Dans cette section, nous allons commencer par présenter l'ordonnanceur HPC Slurm. Ensuite, nous discuterons de sa conteneurisation. Enfin, nous présenterons le PoC (*Proof of Concept*) mis en place.

Ce travail a été présenté au Workshop WAMCA (Workshop on Applications for Multi-Core Architectures) de la conférence SBAC-PAD en 2020 dans un article intitulé « Towards pervasive containerization of HPC job schedulers » [4].

9.1 Présentation de Slurm

Slurm est un ordonnanceur HPC opensource supporté par SchedMD. Il existe depuis une petite vingtaine d'années et il est composé d'environ 500 000 lignes de codes C dont 90% produites par l'équipe de SchedMD. Il est extrêmement répandu dans le monde du HPC car il gère six des dix plus gros clusters du Top500. Si on élargit aux 100 plus grosses machines, il représente 42% de la base installée et 35% des parts du marché des ordonnanceurs HPC.

Les deux composants principaux de Slurm sont slurmctld et slurmd. Le service slurmctld embarque l'ordonnanceur. Il connaît la topologie des ressources disponibles sur le cluster HPC et distribue les travaux en fonction de la configuration d'ordonnancement sélectionnée par l'administrateur (linéaire, partage équitable etc.). Il tourne sur le système avec une identité banalisée. Il peut être mis en cluster pour garantir la haute disponibilité du centre de calcul. Le second composant est slurmd. Ce service qui tourne souvent avec les droits d'administrateur est instruit par slurmctld pour exécuter de façon effective les travaux sur les nœuds de calcul. L'instanciation réelle du travail soumis par l'utilisateur se passe en deux étapes. D'abord slurmd va créer un processus slurmstepd qui va faire office de processus de supervision pour le travail de l'utilisateur en gérant ses entrées / sorties, le suivi de sa consommation de ressources (*accounting*) et les signaux. Ensuite, slurmstepd invoque le travail avec l'identité de l'utilisateur.

Il existe d'autres services optionnels dans la suite logicielle de Slurm. On citera slurmdbd qui gère une base de données contenant tout l'historique des travaux des utilisateurs. C'est sur cette base de données que slurmctld s'appuie lorsqu'il doit fixer la priorité du travail d'un utilisateur dans un contexte d'ordonnancement en partage équitable (*fair-share*). C'est également à partir de cette base de données que les administrateurs peuvent créer des rapports sur l'utilisation du cluster. Ce composant est donc optionnel sur les petits clusters se satisfaisant d'un ordonnancement linéaire. En revanche, à une certaine échelle, il est de facto obligatoire. Nous pouvons également mentionner slurmrestd qui est le serveur de l'API Slurm. Le service slurmrestd est notamment utilisé par le mode sans configuration (*configless*) de Slurm qui permet à slurmd de récupérer sa configuration dynamiquement évitant le déploiement d'un fichier de configuration sur chaque nœud. La figure 9.1 présente les interactions entre ces composants mandataires et optionnels.

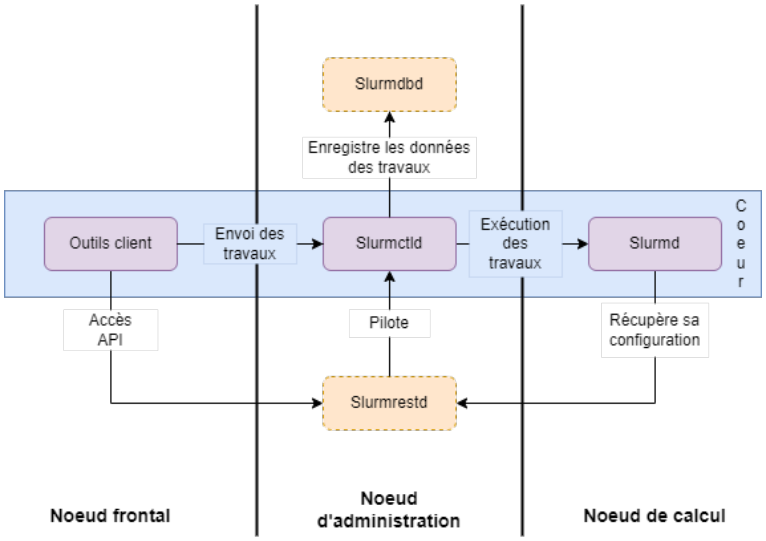


Figure 9.1 – Architecture de Slurm

9.2 Conteneurisation de Slurm

Chacun des services présentés dans la section précédente sont des processus systèmes isolés qui communiquent via un *socket* réseau. D'après les éléments de la section 4.4, ce sont des candidats idéaux pour la conteneurisation dans la mesure où un processus équivaut à un service. La figure 9.2 présente la topologie de conteneur cible. Le conteneur slurmd client embarque un service slurmd sur lequel aucun travail ne peut être envoyé. Le service slurmd sert juste à récupérer dynamiquement la configuration de Slurm pour les outils clients comme sbatch, squeue etc. Le conteneur slurmctld embarque le service slurmctld qui gère l'ordonnancement. Ce conteneur pourra éventuellement être interfacé avec un conteneur slurmdbd qui fera lui-même appel à un autre conteneur Mariadb / MySQL pour stocker les données de facturation (*accounting*). Nous pouvons également ajouter un conteneur slurmrestd pour proposer un accès à l'API de Slurm aux autres conteneurs. Enfin, il reste le service slurmd chargé qui reçoit les travaux pour les exécuter sur les nœuds de calcul. Dans ce dernier conteneur, on trouve la chaîne d'invocation qui, schématiquement, se note slurmd $->$ slurmstepd $->$ travail.

La première étape est de créer une image embarquant l'ordonnanceur HPC. Dans le listing 9.1, nous voyons que cette image est basée sur Debian et embarque également OpenMPI.

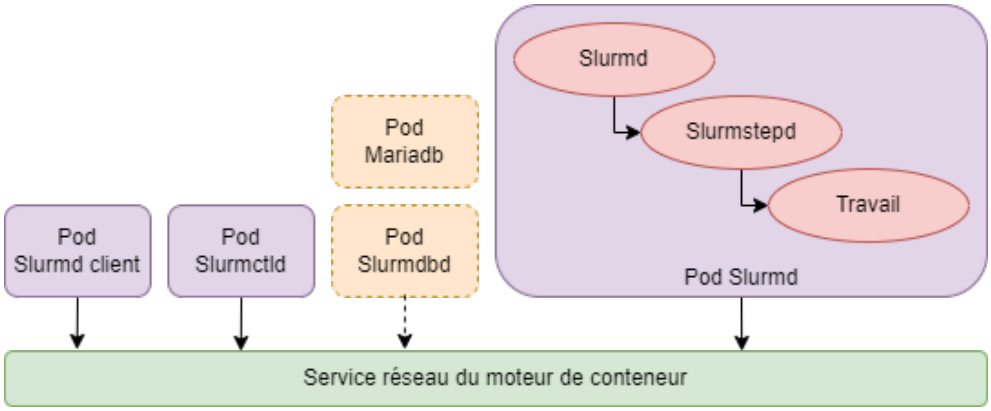


Figure 9.2 – Les services de Slurm

```
FROM debian:latest

RUN apt-get update && apt-get install -y ... #dependances# && \
 wget #openmpi# && tar -xjf #openmpi && \
 wget #slurm# && tar -xjf #slurm


WORKDIR "/slurm"
RUN ./configure --enable-multiple-slurmd --with-munge --enable-slurmrestd
    ↪  && make && make install && make contrib && make install-contrib

WORKDIR "/openmpi"
RUN ./configure --with-slurm --with-pmi=/usr/local && make && make
    ↪ install
WORKDIR "/"

CMD exec /bin/bash -c "trap : TERM INT; sleep infinity & wait"
```

Listing 9.1 – Dockerfile Slurm

Cette image dispose de tous les exécutables mentionnés dans la figure 9.2. À partir de cette image, nous avons instancié un conteneur slurmctld, slurmrestd et n conteneurs slurmd. Ces conteneurs ont fait l'objet d'une orchestration locale très légère avec Docker Compose (cf section 5.4). Ce premier travail visait à vérifier en pratique que la combinaison des assertions des sections 3.2.3, à savoir que les composants d'un ordonnanceur HPC sont indépendants et que les processus les incarnant fonctionnent sur la hiérarchie parent/enfant, et section 4.4, à savoir qu'un conteneur ne doit embarquer qu'un seul processus système et ses fils, est valide pour isoler des ordonnanceurs HPC. Nous avons réussi à instancier Slurm sous forme de conteneurs, et nous avons passé différents code pour vérifier notamment la couche réseau. Ce premier travail a mis en évidence qu'il était possible d'ajouter l'ordonnanceur HPC dans la panoplie des outils d'expérimentation reproduc-

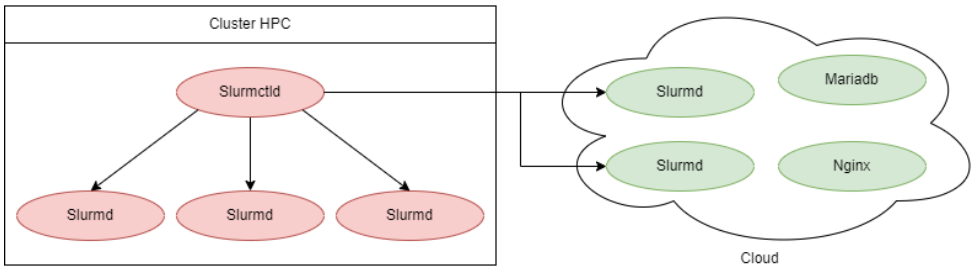


Figure 9.3 – Objectif de cohabitation

tible, du moins pour l'environnement logiciel. Cependant, dans ce travail nous visions une seconde propriété, la cohabitation.

9.3 Orchestration de Slurm

Dans cette section nous allons définir précisément la propriété de cohabitation et nous allons démontrer de quelle manière l'orchestration de conteneur pourrait la satisfaire. La figure 9.3 représente notre cible, à savoir une infrastructure hybride composée de nœuds de calcul classiques et conteneurisés. La partie gauche de la figure représente un cluster HPC physique traditionnel. Sur la partie droite, nous voyons des processus *slurmd* conteneurisés cohabitant avec d'autres services à destination d'autres populations et d'autres usages tels que Nginx, Mariadb, etc. Nous donnerons également les éléments techniques de l'orchestration de Slurm avec Kubernetes.

9.3.1 Cohabitation

La cohabitation est la capacité pour des conteneurs d'origine hétérogène à partager la même infrastructure. La cohabitation implique plusieurs choses :

— L'infrastructure est composée de plusieurs noeuds (physiques ou virtuels) ;
— L'infrastructure est utilisée par des zones géographiques, populations, domaines applicatifs etc. diverses ;
— L'attribution des ressources est orchestrée par un service tiers.

Dans la section 1.2 consacrée à la motivation du sujet, nous avons décrit une étude menée auprès de centres nationaux de différentes échelles. La conclusion est que dans les centres généralistes, les travaux sont très hétérogènes et ne mettent pas forcément la mémoire vive et le CPU à 100% en continu. En effet, certains utilisateurs cherchent juste à lancer un travail pas forcément très optimisé quelque part ou il va pouvoir s'exécuter longtemps. Sur le centre de calcul de l'USPN (anciennement centre de l'USPC avant la fin de la COMUE), nous avons également observé cette tendance. Pour ces utilisateurs, il n'est pas nécessaire de lancer leur travail sur des nœuds HPC physiques coûteux (processeurs puissants, mémoire vive rapides, réseau faible latence, scratch SSD etc.). Il serait plus opportun de proposer des nœuds plus économiques. Cependant, la tendance n'est plus

du tout à entasser des machines physiques peu puissantes. Le rapport encombrement / consommation électrique / puissance est catastrophique.

Ce qu'il faut bien comprendre quand on investit dans des machines c'est qu'il existe un « *sweet pot* » en terme de rapport puissance / prix. Si on considère que 100 est la puissance de la meilleure machine du monde, alors arriver à la puissance de 80 va coûter 40. Pour arriver a 84, cela va coûter 50, pour arriver a 90 cela va coûter 80 etc. On voit que le coût augmente énormément pour un gain de puissance qui n'est plus du tout en rapport. Pour des machines de calcul très spécialisées cela peut valoir le coup, par contre pour une utilisation plus standard on peut se contenter de machines à la puissance de 80.

À cette étape de notre travail, notre objectif était de permettre aux travaux HPC moins demandeurs de ressources spécifiques de déborder sur des machines plus standard dans une optique de délestage. La virtualisation (cf section 4.1) a explosé grâce à ce nouveau paradigme de cohabitation. L'idée était de faire cohabiter différents systèmes d'exploitations sur un même serveur pour saucissonner les ressources de la machine entre les différents invités virtualisés. C'est une première étape vers une meilleure granularité d'utilisation des ressources d'une machine physique.

Les conteneurs sont venus ajouter une granularité supplémentaire en confinant non plus un système d'exploitation complet mais un processus système (cf section 4.4). Comparer les deux n'a aucun sens car ce sont des technologies radicalement différentes. En revanche, elles se complètent dans les infrastructures Cloud dans le sens ou les orchestrateurs instancient les conteneurs dans les machines virtuelles elles-mêmes localisées sur des nœuds physiques. En effet, cela permet de simplifier la problématique du provisionnement. Les nœuds Cloud étant des machines virtuelles, il est beaucoup plus simple de les arrêter, déplacer, restaurer etc. La seule chose à maintenir sur le nœud physique est l'hyperviseur. Notre travail est d'arriver à instancier tout ou partie d'un cluster HPC sous forme de conteneur avec un orchestrateur. En conséquence, nos éléments conteneurisés vont cohabiter avec les autres services gérés par l'ordonnanceur. Nous allons maintenant discuter des aspects techniques du travail.

9.3.2 Intégration de Slurm avec Kubernetes

L'instanciation de Slurm dans Kubernetes repose sur le déploiement de plusieurs composants. En premier lieu, il faut déployer un composant de type `Service` pour avoir un service réseau entre les Pods. Le listing 9.2 décrit ce service. C'est un service réseau simple. Chaque Pod possédera son IP, on ne crée pas d'IP virtuelle. Chaque Pod aura une entrée de type A dans le DNS de Kubernetes. C'est le service réseau le plus simple disponible. Les Pods assurant le déploiement d'un Slurm conteneurisé vont tous utiliser ce service réseau.

```
apiVersion: v1
kind: Service
metadata:
  name: nodes
  namespace: hpc-nico
```

```
spec:
  selector:
    net: headless
  clusterIP: None
```

Listing 9.2 – Instanciation du service réseau

Une fois ce service réseau défini, nous allons construire l'infrastructure conteneurisée représentée à la figure 9.4. Nous avons 3 types de Pods : Client, Slurm et Slurmlctld. Le Pod Slurmctld, défini dans le listing 9.3, contient trois conteneurs : un conteneur d'initialisation (`initContainer`) et deux conteneurs standards. Le conteneurs d'initialisation nommé « slurmconf » sert à interroger le serveur d'API de Kubernetes pour connaître les propriétés des Pods Slurmd (qui représentent les nœuds de calcul conteneurisés) pour générer le fichier de configuration `slurm.conf` utilisé par Slurm.

Ce fichier généré est écrit sur un volume partagé nommé « slurmconf-volume ». Ce volume est de type `emptyDir`, ce qui correspond à un volume local aux conteneur du Pod qui apparaît et disparaît avec lui. Une fois le fichier généré, le conteneur d'initialisation s'arrête. Ensuite, les conteneurs applicatifs (`container`) « slurmctld » et « munge » sont démarrés. Le conteneur nommé « slurmctld » fait tourner le processus slurmctld ainsi que les *threads* slurmrestd lorsque des appels à l'API se font, notamment par les processus slurmd pour récupérer la configuration inscrite dans le `slurm.conf`. Il est accompagné d'un conteneur nommé « munge ».

Le service Munge gère l'authentification et le chiffrement des communications entre les composants de Slurm. La sécurité de Munge repose sur la cryptographie symétrique dans la mesure où chaque instance de munged (le démon de Munge) possède la même clé. La communication avec le démon slurmd ou slurmctld se fait via un *socket* local. C'est pour cette raison qu'on trouve un volume partagée de type `emptyDir` nommé « sock » entre les conteneurs « slurmd », « slurmctld » et « munge ».

```
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: control-node
  namespace: hpc-nico
  labels:
       name: control-node
spec:
  selector:
       matchLabels:
       app: slurmctld
  serviceName: "nodes"
  replicas: 1
  template:
```

```
metadata:
  labels:
    app: slurmctld
    role: control-node
    net: headless
spec:
  initContainers:
  - name: slurmconf
 image: nyk0/slurmconf
    volumeMounts:
    - mountPath: /etc/slurm
   name: slurmconf-volume
    command: ["/usr/local/bin/python"]
    args: ["generate_slurm_conf.py"]
  containers:
  - name: slurmctld
 image: nyk0/slurmcontainer
    volumeMounts:
 - mountPath: /etc/slurm
   name: slurmconf-volume
    - mountPath: /run/munge
name: sock
    command: ["/bin/bash"]
    args: ["start-slurmctld.sh"]
  - name: munge
 image: nyk0/slurmmunge
    volumeMounts:
 - mountPath: /run/munge
   name: sock
    command: ["/bin/bash"]
    args: ["start-munge.sh"]
  volumes:
  - name: slurmconf-volume
    emptyDir: {}
  - name: sock
    emptyDir: {}
  - name: home
  dnsConfig:
     searches:
 - nodes.hpc-nico.svc.cluster.local
```

Listing 9.3 – Le Pod Slurmctld

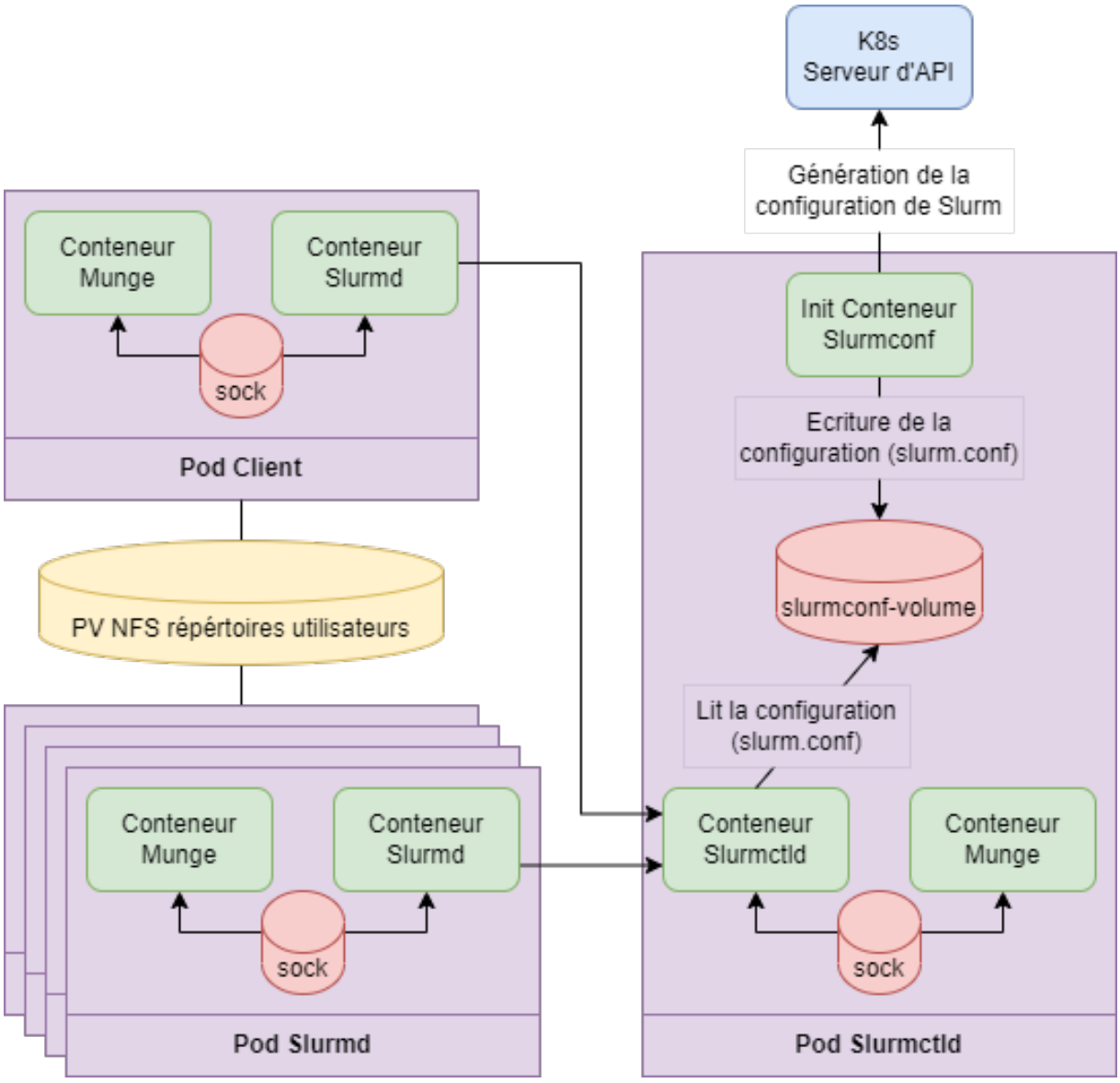


Figure 9.4 – Interactions des composants de Slurm

Le Pod nommé « Slurmd » décrit dans le listing 9.4 représente le nœud de calcul conteneurisé. Le premier point saillant à noter est que le déploiement de ces instances de slurmd conteneurisées s'appuie sur un contrôleur `StatefulSet` (cf section 5.3.3). Chaque instance de Pod déployé a donc :

— Un nom fixe : si on déploie X Pod slurmd, alors le premier se nommera hpc-node-0, le second hpc-node-1 etc. jusqu'à hpc-node-(X - 1). Les noms sont donc incrémentaux (donc prédictibles). Ce schéma de nommage est tout à fait compatible avec les outils de gestion / configuration des clusters et ordonnanceurs HPC dans la mesure où ils sont adressées par des rangs (exemple : hpc-node-[5-8] pour les machines hpc-node-5 à hpc-node-8) ;
— Les Pods apparaissent et disparaissent en mode LIFO (Last In First Out) lors des passages à l'échelle horizontaux. Le nombre de Pods du `StatefulSet` est conditionné par la valeur associée à l'attribut `replicas`. Sur un déploiement avec `replicas` avec une valeur de deux, nous aurons deux Pods nommés hpc-node-[0-1]. Si on ajoute un Pod (c'est-à-dire si on passe replicas à trois), alors on se retrouve avec trois Pods hpc-node-[0-2]. Si on repasse à deux, alors on retombe dans la situation initiale soit deux Pods hpc-node-[0-1] ;
— Les adresses IP sont stables par rapport au Pod. Si par exemple on crée un `StatefulSet` avec trois Pods (hpc-node-[0-2]) et qu'on le redimensionne à deux (hpc-node-[0-1]), alors si on le repasse de nouveau à trois (hpc-node-[0-2]), le dernier Pod (hpc-node-2) conservera la même IP à travers ses instanciations successives.

A l'instar du Pod slurmctld, le Pod slurmd comporte un conteneur slurmd et un conteneur munge. Le conteneur munge fonctionne exactement comme dans la section précédente. Il communique avec le conteneur slurmd via un volume partagé `emptyDir` qui accueille le *socket* local. Dans la configuration du conteneur slurmd, nous avons configuré une section `resources` avec deux sous rubriques limits et requests. Cette section va configurer un cgroup (cf section 4.2) pour y placer le processus conteneurisé. Dans notre exemple, nous limitons la ressource cpu à deux. Nous demandons au moins deux CPUs (`requests`) et nous limitons la consommation à deux CPUs (`limits`).

En conséquence, notre conteneur aura toujours 200% de quota cpu. C'est effectivement différent en termes d'affinité processus / processeur que d'avoir deux cœurs dédiés. Nous rappelons que notre travail ne vise pas à remplacer l'écosystème HPC (ce qui serait d'une incroyable prétention) mais à modestement le compléter. Pour en finir avec le conteneur slurmd, il dispose d'un volume home qui est un partage NFS hébergeant les répertoires utilisateurs accessibles de tous les Pods slurmd faisant office de nœuds de calcul conteneurisés.

```
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: hpc-node
  namespace: hpc-nico
  labels:
```

```
      name: hpc-node
spec:
  selector:
      matchLabels:
      app: slurmd
  serviceName: "nodes"
  replicas: 2
  template:
      metadata:
      labels:
      app: slurmd
      net: headless
      role: compute-node
      partition: COMPUTE
      spec:
      containers:
      - name: slurmd
      image: nyk0/slurmcontainer
      volumeMounts:
      - mountPath: /run/munge
      name: sock
      - mountPath: /home
      name: home
      resources:
      limits:
             cpu: "2"
      requests:
             cpu: "2"
          command: ["/bin/bash"]
          args: ["start-slurmd.sh"]
      - name: munge
      image: nyk0/slurmmunge
      volumeMounts:
      - mountPath: /run/munge
      name: sock
      command: ["/bin/bash"]
      args: ["start-munge.sh"]
      volumes:
      - name: sock
      emptyDir: {}
      - name: home
      persistentVolumeClaim:
```

```
      claimName: pvc-nfs-home
      dnsConfig:
      searches:
      - nodes.hpc-nico.svc.cluster.local
```

Listing 9.4 – Le Pod Slurmd

9.4 Expérimentation

Nous avons fait deux expérimentations. Une pour réaliser une hybridation du centre de calcul de l'USPN. Notre objectif était d'ajouter des nœuds de calcul conteneurisés orchestrés dans un cluster Kubernetes à l'infrastructure physique existante. La seconde venait en soutien de la partie expérimentale d'un article à propos d'un algorithme d'ordonnancement de Pods soumis à NPC en 2021 nommé « Towards an Optimized Containerization of HPC Job Schedulers Based on Namespaces » [87]. Cette fois ci, nous avons instancié un cluster HPC complet.

9.4.1 Scénario 1 : Hybridation

L'idée est d'ajouter une partition peuplée de nœuds conteneurisés en suppléments des nœuds de calculs physiques existants. Le défi dans le provisionnement de nœuds de calcul conteneurisés pour proposer le service hybride est de les connecter en réseau extérieur. En effet, on travaille à une connexion dans les couches basses de la pile réseau, il faut donc mettre en correspondance les latitudes de configuration qu'on a du côté du cluster HPC physique et le fonctionnement de l'*Overlay* réseau (Calico dans nos expérimentations). A l'USPN, notre réseau est segmenté en VLAN (*Virtual Local Area Network*) et le centre de calcul en utilise trois : un pour le réseau de déploiement des nœuds, un pour le réseau hors bande et le dernier pour les contrôleurs IPMI. Les communications entre les services slurmd et slurmctld passent par le réseau hors bande. Les interfaces de VLAN constituants les passerelles par défaut des différents réseaux IP sont portées par un pare-feu de site Palo Alto.

Calico, l'*overlay* réseau que nous utilisons avec notre infrastructure Kubernetes, créé un réseau de tunnels IPIP (*IP in IP*) pour interconnecter les Pods. Il s'agit d'une méthode utilisée pour encapsuler un paquet IP complet dans un autre paquet IP pour recréer un réseau IP logique au-dessus du réseau IP existant. Calico crée donc un maillage au-dessus du réseau IP des nœuds Cloud de Kubernetes pour interconnecter les Pods. Les méthodes permettant d'acheminer les paquets réseaux entre les Pods ne sont pas normées par Kubernetes. Chaque *Overlay* implémente ses propres méthodes de connexion.

Pour notre couple Palo Alto / Calico, la méthode la plus efficace aurait été de créer une interface IPIP sur le Palo Alto et d'y router le trafic à destination du réseau géré par l'*Overlay* Calico. Cette solution est la plus efficace en terme de latence, cependant le pare-feu Palo Alto ne supporte pas les interfaces IPIP. Nous avons donc mis en place une

machine tierce sous Linux sur le VLAN dédié au cluster Kubernetes avec une interface IPIP, comme exhibée dans la figure 9.5, qui sert de point d'entrée sur le réseau. C'est souvent ce type de topologie qu'on retrouve chez les fournisseurs de Cloud. Les Pods sont derrière un équilibreur de charge qui porte les IP « réelles » et qui effectue la correspondance avec les IP affectées aux Pods par l'*Overlay*. Il faut noter que la publication des Pods via un équilibreur de charge frontal est propre à chaque hébergeur. En plus de cette gestion de la connectivité, nous avons dû ajouter un conteneur d'initialisation se lançant avant le conteneur slurmd pour enregistrer dynamiquement le nœud conteneurisé dans la zone DNS du cluster HPC.

Nous rappelons qu'avec Slurm, chaque travail s'exécute dans une partition. Une partition est un ensemble (souvent homogène) de nœuds de calcul. Un travail ne déborde pas d'une partition à une autre. Ainsi, dans la mesure du possible, il est souhaitable de regrouper les nœuds conteneurisés sur dans une partition dédiée. De cette manière, les communications inter-noeuds n'ont à traverser qu'un seul niveau de translation, celui de l'*Overlay*. Dans le cas de nœuds éparpillés à l'intérieur et à l'extérieur du cluster Kubernetes, il faudra d'abord passer par le mécanisme d'exposition puis pas l'*Overlay*.

9.4.2 Scénario 2 : l'instanciation complète

Cette expérience a été réalisée dans le cadre d'un article soumis à NPC en 2021 nommé « Towards an Optimized Containerization of HPC Job Schedulers Based on Namespaces » [87]. L'objet de ce papier est de présenter un algorithme d'orchestration basé sur les *namespaces* pour Kubernetes. Un *namespace* dans Kubernetes (à ne pas confondre avec les *namespaces* du noyau Linux tels que présentés dans la section 4.3) est un cluster virtuel permettant d'isoler les ressources attachées à ce *namespace* des autres ressources. Par défaut, toutes les ressources créées sur le cluster (Pods, services etc.) sont placées dans un *namespace* nommé « default ».

L'algorithme présenté dans l'article vise à tenter d'entasser les Pods du même *namespace* sur les mêmes nœuds Kubernetes. Pour le HPC c'est particulièrement intéressant car si les nœuds de calcul conteneurisés sont localisés sur la même machine, alors la latence des échanges réseau lors d'un travail à mémoire distribuée (cf section 3.2.2) est réduite. Contrairement à l'hybridation présentée précédemment, cette fois ci nous instancions également slurmctld. Le cluster HPC est donc totalement conteneurisé. Pour cette expérience, nous avons automatisé la création de 280 Pods slurmd répartis aléatoirement dans six *namespaces* (au sens Kubernetes). Nous les distribuons dans un ordre totalement aléatoire sur quatre nœuds physiques. La répartition est présentée dans la figure 9.6. Nous constatons que les nœuds conteneurisés appartenant au même *namespace* sont entassés sur la même machine physique.

9.5 Résultats et perspectives

D'après ces premiers résultats, il est possible de conteneuriser l'ordonnanceur HPC Slurm. Nous avons éprouvé cette conteneurisation dans un contexte d'hybridation avec

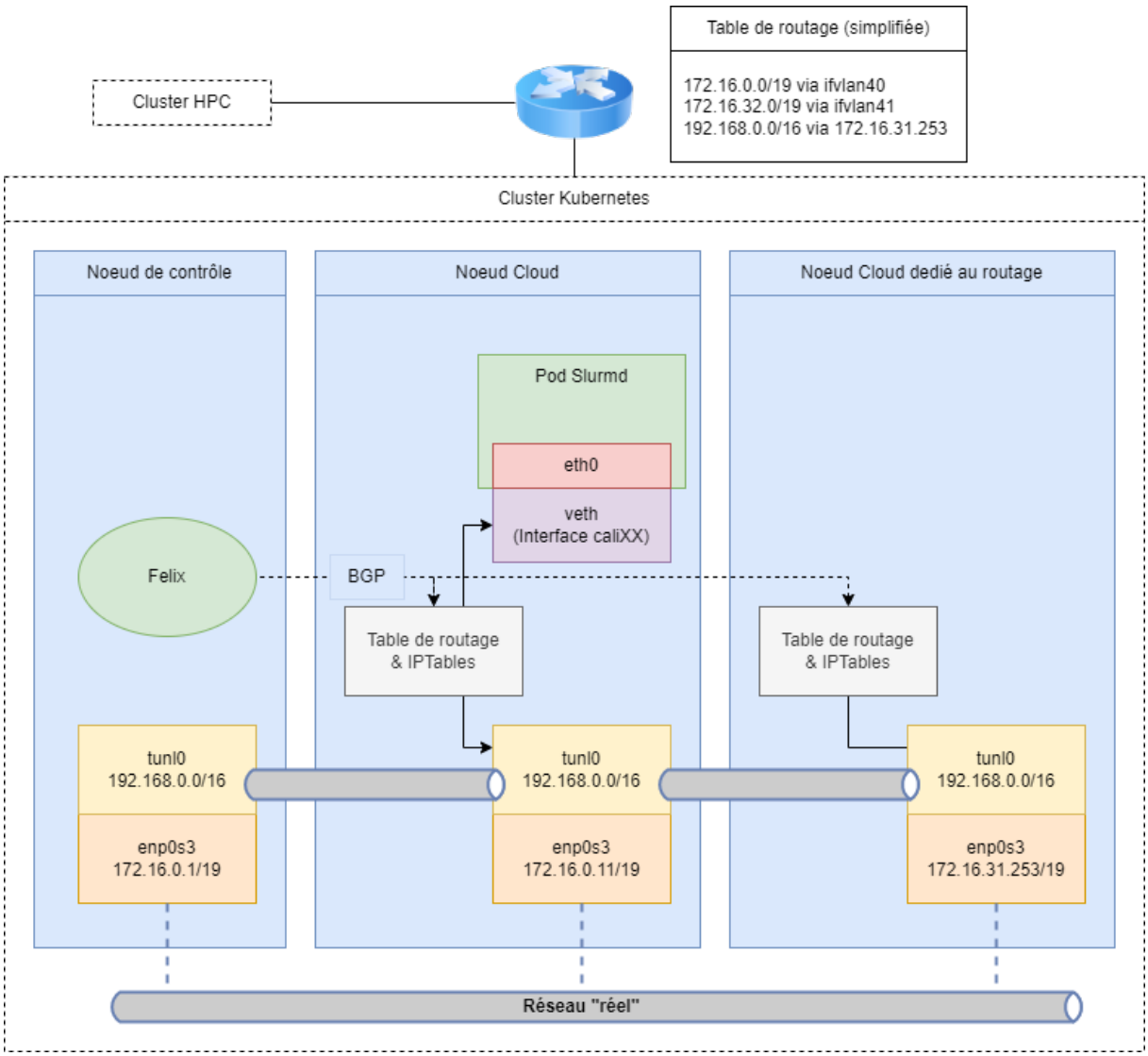


Figure 9.5 – Architecture hybride

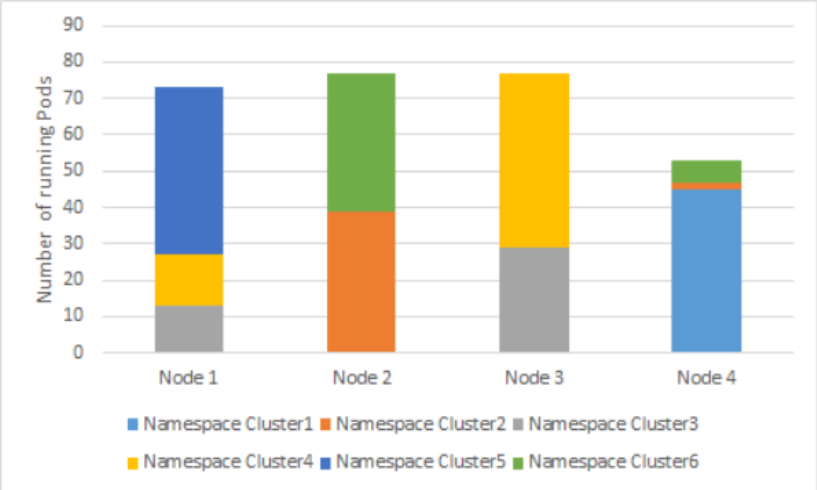


Figure 9.6 – Distribution de 280 Pods

un cluster réel et dans un contexte totalement Cloud. Nous avons vu dans la section 8.1 que les conteneurs sont intégrés aux ordonnanceurs HPC pour fournir des images directement utilisables en s'appuyant le moins possible sur les dépendances (librairies, configurations etc.) via l'espace de nom (cette fois ci au sens système, cf section 4.3) `mount_ns`.

De cette façon, la reproductibilité est accrue en fixant l'environnement et la version du logiciel scientifique utilisé sur le cluster HPC. En intégrant l'ordonnanceur HPC directement dans le conteneur, nous ajoutons une couche à cette reproductibilité. Les applications d'un tel travail sont multiples : fournir un environnement de test plus souple et moins volumineux qu'un ensemble de machines virtuelles, intégrer les ordonnanceurs HPC dans les chaînes de compilation et validation ou encore versionner les configurations du centre de calcul etc.

L'orchestration permet également de placer les nœuds conteneurisés parmi d'autres services plus standards. L'hybridation combinée à la capacité de cohabitation entre des nœuds conteneurisés et les autres services orchestrés par Kubernetes permettraient de limiter le gâchis de ressources mis en évidence dans la section 1.2 sur la motivation du sujet. En effet, les configurations de nœuds de calcul physiques sont figées. Les infrastructures Cloud proposent ce que l'on appelle des micro services. Les micro services sont des fonctionnalités que l'on peut attacher aux conteneurs telles que du stockage objet, de la base de données etc. Ainsi, si un service consomme finalement peu de CPUs mais qu'il est gourmand en stockage, il est possible de lui en attacher un plus performant. Les nœuds de calcul conteneurisés peuvent donc piocher dans les micro-services pour composer des configurations adaptées au travail à exécuter.

Nous avons donc transposé deux propriétés du Cloud vers le monde du HPC. Dans la suite du travail, nous visons une troisième propriété : le passage à l'échelle. Appliquée à notre travail, cette propriété vise la capacité d'ajouter dynamiquement des nœuds de calcul conteneurisés dans le cluster HPC. Cependant, le travail effectué est très empirique et basé sur un seul ordonnanceur HPC : Slurm. Nous devons étendre notre travail à d'autres ordonnanceurs HPC pour vérifier que notre travail est transposable.

CHAPITRE 10

MÉTHODE DE PASSAGE À L'ÉCHELLE DES ORDONNANCEURS CONTENEURISÉS

La seconde partie de notre travail a été de réfléchir au moyen d'appliquer la propriété de passage à l'échelle horizontale (*horizontal scaling*) courante dans les environnements Cloud. Cette propriété est d'ailleurs l'argument de vente numéro 1 des fournisseurs tels que GKE (Google), AWS (Amazon), Azure (Microsoft). La promesse est d'adapter la capacité de votre infrastructure à la charge (moyennant un coût supplémentaire, évidemment). Nous avons commencé par réinvestir le travail présenté à WAMCA afin de conteneuriser deux ordonnanceurs HPC supplémentaires : OpenPBS et OAR. Ensuite, nous avons testé l'ajout et le retrait de nœuds HPC conteneurisés à chaud en suivant une méthode générique transversale aux différents ordonnanceurs. L'article issu de ce travail nommé « A Methodology to Scale Containerized HPC Infrastructures in the Cloud » [124] a été présenté à la conférence Europar édition 2022. L'artefact associé a, de plus, remporté le prix de la meilleure expérimentation [125].

10.1 Redimensionnement de cluster HPC conteneurisé

Dans cette section, nous allons présenter une méthode générale pour permettre aux ordonnanceurs HPC conteneurisés de se redimensionner en fonction de l'ajout ou du retrait de nœuds de calcul (également conteneurisés). Nous commencerons par introduire la méthode générale et sa déclinaison sur trois ordonnanceurs HPC : Slurm, OpenPBS et OAR. En effet cette méthode possède un niveau « macro » qui est le principe général et un niveau « micro » qui est son application aux différents ordonnanceurs HPC. En effet, chacun possède des particularités qui nécessitent une adaptation de la méthode.

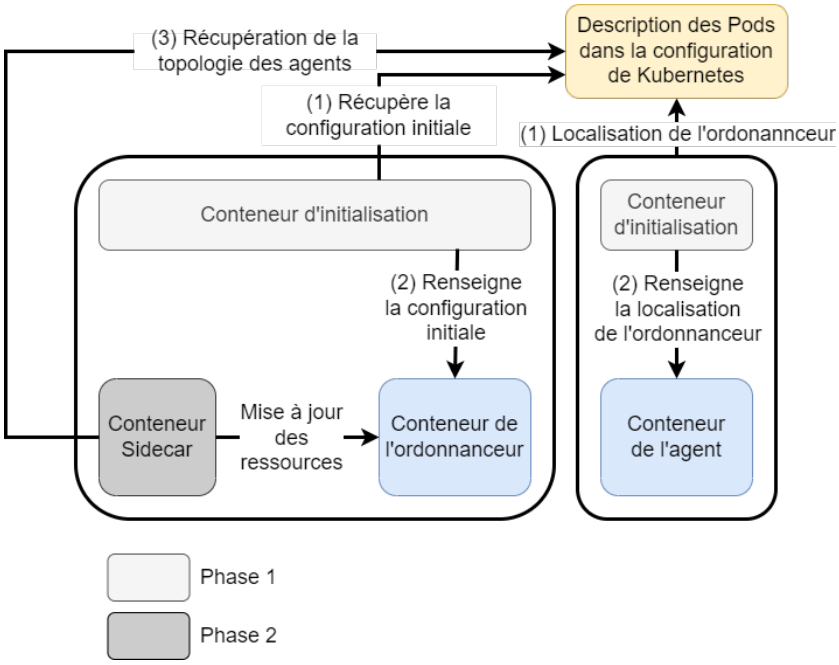


Figure 10.1 – Méthode au niveau »macro »

10.1.1 Méthodologie niveau macro

Au niveau macro, tous les ordonnanceurs HPC sont plus ou moins architecturés de la même façon. Nous avons présenté les grandes lignes du fonctionnement général d'un ordonnanceur dans la section 3.2.3. Les étapes de la méthode ainsi que leur articulation sont présentées dans la figure 10.1. L'idée générale est que l'ordonnanceur se cale sur la configuration du déploiement pour générer sa configuration. En cas de changement de celle ci, par exemple ajout / suppression d'agent conteneurisés, il doit régénérer sa configuration et l'appliquer. Cette dernière étape peut avoir une incarnation différente en fonction des ordonnanceurs.

Sur la figure 10.1, on voit que le conteneur de l'agent et celui de l'ordonnanceur sont chacun accompagnés d'un conteneur d'initialisation. Dans le cas de l'ordonnanceur, il sert à interroger la configuration du déploiement dans Kubernetes pour générer un fichier de configuration initial. En effet, l'orchestrateur connaît la description de chacun des Pods constituants les agents conteneurisés de l'ordonnanceur HPC, notamment la mémoire demandée, le nombre de CPUs etc. La configuration d'un ordonnanceur se résume à :

— des paramètres propres à l'ordonnanceur tels que l'algorithme d'ordonnancement choisi pour les travaux, l'identité utilisée par le programme etc. ;
— une description des ressources disponibles, soit la topologie des nœuds de calcul conteneurisés.

Du côté de l'agent, le conteneur d'initialisation interroge également la configuration du déploiement de Kubernetes, la plupart du temps pour localiser l'ordonnanceur. La récupération de la configuration est une action qui sera faite peu importe l'ordonnanceur (macro), cependant son application est propre à chacun (micro). Cette configuration ini-

tiale est la première phase de la méthode. A ce moment la, le cluster HPC conteneurisé est prêt à fonctionner. La seconde phase consiste à suivre les changements de configuration dans le déploiement tels que les ajouts / suppression de nœuds.

Pour réaliser cette surveillance, l'ordonnanceur est accompagné d'un autre conteneur qu'on appelle généralement un *sidecar*. Son rôle est d'assister le conteneur principal pour, par exemple, mettre sa configuration à jour en cas de changement de contexte. En cas de changement de topologies des nœuds HPC conteneurisés, le conteneur *sidecar* change la configuration de l'ordonnanceur et procède aux actions nécessaires à la prise en compte de la nouvelle configuration. Le processus de surveillance des changements de configuration du déploiement sera le même pour tous les ordonnanceurs (macro) en revanche les attributs à surveiller ainsi que l'action à réaliser pour leur prise en compte sont spécifiques à chaque ordonnanceur (micro). Nous allons maintenant discuter des détails au niveau micro.

10.1.2 Méthodologie niveau micro

Dans cette section nous allons surtout discuter du fonctionnement du conteneur *sidecar* qui est au cœur du processus de passage à l'échelle. Nous verrons comment chacun des trois ordonnanceurs passent à l'échelle dans un contexte Cloud. Nous allons commencer avec Slurm que nous avons déjà pas mal couvert dans la section précédente. Dans cette section, nous appellerons ordonnanceur le ou les processus s'exécutant sur le nœud maître chargés de prendre les décisions d'affectation des travaux aux nœuds de calcul.

Slurm repose sur deux services slurmctld et slurmd. Le service slurmctld implémente l'ordonnanceur et connaît les ressources du cluster HPC tandis que slurmld se charge d'exécuter les travaux sur les nœuds. La configuration de Slurm se fait avec un fichier plat de configuration. Le conteneur d'initialisation se charge de générer ce fichier. Le service slurmctld est configuré pour supporter le mode « *configless* ». Ce mode permet aux agents slurmd de récupérer dynamiquement leur configuration sans avoir besoin de pousser le fichier de configuration sur chaque nœud. Cette configuration requiert forcément le service Munge pour authentifier les communications entre les agents slurmd et l'ordonnanceur slurmctld.

En conséquence, nous nous retrouvons avec deux Pods : slurmd et slurmctld. Le Pod slurmd embarque un conteneur d'initialisation chargé de localiser le service slurmctld afin de télécharger la configuration ainsi qu'un conteneur Munge et un dernier conteneur slurmd. Le Pod slurmctld embarque quant à lui quatre conteneurs. Le premier est le conteneur d'initialisation qui génère la configuration de l'ordonnanceur. Le second est le conteneur Munge. Le troisième est le service slurmctld. Enfin, le dernier est le conteneur *sidecar* chargé de gérer les changements de configuration de l'ordonnanceur.

Ce conteneur *sidecar* surveille en continu les paramètres du déploiement via le serveur d'API Kubernetes. Lorsqu'un changement est détecté (tels que l'ajout ou la suppression de nœuds), le *sidecar* conteneur met à jour la configuration. En cas de modification de la configuration, le service slurmctld doit être redémarré. Ce redémarrage pose problème car son arrêt entraîne la destruction du Pod ce qui entraîne la perte des informations à propos des travaux en cours. Une solution pourrait être de créer un volume persistant

pour y stocker les informations des travaux en cours. Cependant cela crée une dépendance à la création du Pod. En effet, si la demande de volume physique (`pvclaim`) n'aboutit pas, le Pod ne peut pas démarrer. Nous avons opté pour une autre solution qui est d'utiliser les daemontools [1] développés par DJ Bernstein.

Les daemontools implémente un service nommé « supervise » qui est le processus réellement démarré par le conteneur slurmctld. Ce service lance à son tour le processus slurmctld (qui devient donc son fils) et le supervise. Le processus « supervise » ressuscite les processus fils en les redémarrant en cas d'arrêt. En effet, le conteneur *sidecar* doit tuer le processus slurmctld lors de modification de configuration. Cette procédure nécessite donc que le conteneur *sidecar* puisse envoyer un signal SIGTERM (cf. annexe A.5). Il faut donc que le conteneur *sidecar* et slurmctld partagent le même *namespace* PID (*pid_ns*). Ainsi le conteneur *sidecar* envoie le SIGTERM au processus slurmctld qui est surveillé par « supervise ». Une fois décédé, le processus slurmctld est relancé par « supervise ».

OpenPBS est un ordonnanceur HPC opensource. Il est composé de plusieurs services. Le service pbs_sched est l'ordonnanceur à proprement parler. Le service pbs_comm gère la haute disponibilité. Enfin, le service pbs_server.bin communique d'une part avec pbs_sched pour récupérer les décisions d'ordonnancement et d'autre part avec les agents déployés sur les nœuds de calcul pour leur distribuer le travail. Il communique également avec une base de données Postgres pour y stocker la description des ressources disponibles, notamment les spécifications des nœuds de calcul associés et les descriptions des travaux des utilisateurs. L'agent déployé sur les nœuds de calcul se nomme pbs_mom. C'est lui qui va exécuter les travaux sur les nœuds de calcul (à l'instar de slurmd).

Une fois de plus, nous avons deux types de Pods : l'ordonnanceur et l'agent. Le Pod accueillant l'ordonnanceur contient trois conteneurs, un conteneur d'initialisation qui génère la configuration d'OpenPBS, un conteneur qui exécute les processus nécessaires pilotant le cluster (Postgres, pbs_sched, pbs_comm et pbs_server.bin) et un conteneur *sidecar* qui ajoute ou retire des nœuds conteneurisés de la base de données d'OpenPBS. La démarche de conteneurisation d'OpenPBS ressemble fortement à celle de Slurm. La différence notable est que la partie ordonnancement est composée de plusieurs processus indépendants communiquant via des socket locaux ou réseaux alors que Slurm a un unique processus slurmctld qui crée des threads au besoin (tel que slurmrestd lorsqu'il y a des appels à l'API). Cela rend la conteneurisation de cette partie un peu plus complexe mais la vision de haut niveau (macro) reste la même.

OAR est un ordonnanceur opensource développé à l'INRIA (Institut National de Recherche en Informatique et en Automatique) et à Grenoble (LIG). À l'instar de n'importe quel autre ordonnanceur, nous avons une partie qui gère l'ordonnancement et un agent sur chaque nœud de calcul qui reçoit les travaux à exécuter. Le processus gérant l'ordonnancement s'appelle Almighty et interagit avec une base de données Postgres pour stocker les informations telles que la topologie des ressources disponibles, l'état du cluster et les informations des travaux soumis. Une originalité d'OAR est qu'il s'appuie sur le protocole SSH (Secure SHell) et son implémentation libre OpenSSH pour agir en tant qu'agent. OpenSSH est un service personnifié par un processus nommé sshd permettant

1. https ://cr.yp.to/daemontools.html

d'exécuter un Shell sur une machine distante via un canal chiffré. La configuration de l'agent consiste à déployer un fichier de configuration OpenSSH accompagné d'une clé sur le nœud de calcul.

OAR est plutôt simple à conteneuriser. La raison principale est que les agents sont très faiblement couplés avec l'ordonnanceur. En effet, pour OpenPBS ou Slurm il faut que l'ordonnanceur soit disponible lorsque l'agent démarre. Cela implique un travail de synchronisation à gérer dans le conteneur d'initialisation. Une fois de plus, rien de nouveau au niveau macro. Le Pod de l'ordonnanceur contient un conteneur d'initialisation chargé de générer la configuration d'OAR, un conteneur qui exécute Almighty et un conteneur *sidecar* chargé de gérer l'ajout et la suppression des nœuds de calcul conteneurisés. Ce Pod est accompagné d'un Pod Postgres. Côté agent, le Pod contient un conteneur d'initialisation chargé de descendre la configuration d'OAR et un conteneur démarrant sshd. Il faut noter que OAR requiert quelques capacités (cf. annexe A.12). Almighty doit posséder la capacité CAP_NET_RAW pour générer des paquets ICMP afin de déterminer le statut des nœuds de calcul. Le conteneur sshd de l'agent doit posséder la capacité CAP_SYS_CHROOT car l'exécution du travail est isolée au niveau système de fichier.

10.2 Expérimentation de passage à l'échelle

Dans cette section nous allons utiliser les environnements définis précédemment pour mettre en place une expérimentation sur le passage à l'échelle de nos clusters HPC conteneurisés. Ici, nous discutons d'un passage à l'échelle manuel, c'est-à-dire que les nœuds conteneurisés sont ajoutés manuellement. Nous discuterons du passage à l'échelle automatique dans la section suivante. Tous les déploiements de cluster HPC conteneurisés se font via le contrôleur `StatefulSet`. Nous rappelons que ce contrôleur présenté dans la section 5.3.3 créé X instances d'un Pod donné (X étant fixé dans la configuration du déploiement par le paramètre `replica`) nommées nom-0, nom-1 etc. jusqu'à nom-(X-1). Ces Pods ont des IPs persistantes à travers leur cycle de création / destruction. Enfin, si le nombre de Pods est revu à la baisse, ils sont supprimés sur le mode LIFO (*Last In First Out*).

10.2.1 Mode opératoire

Nous allons éprouver la capacité des ordonnanceurs à prendre en compte des changements de topologies de ressources. Nous n'allons pas mesurer l'impact de la conteneurisation sur les performances dans la mesure où dans la section 4.3 nous avons montré que les espaces de noms responsable du confinement sont profondément entremêlés avec les mécanismes de gestion des processus. De plus, toutes les distributions Linux viennent avec des noyaux pré-compilés supportant les espaces de noms. Enfin, le système de démarrage des services systemd est prévu pour ajouter les processus dans un cgroup (cf. section 4.2) par défaut. Retirer le support des espaces de noms et des cgroups d'un système basé sur Linux est donc un travail fastidieux et important, qui n'est en pratique jamais réalisé. Le sujet de l'expérimentation est plutôt de caractériser l'impact du passage à l'échelle sur les travaux en exécution ou en attente.

Nous avons joué plusieurs scénarios en lançant des travaux en mémoire distribuée et partagée. Pour la partie distribuée, il s'agit d'un calcul de Pi basé sur la méthode de Monte Carlo compilé avec OpenMPI. L'objectif est d'observer le potentiel impact du passage à l'échelle sur les communications MPI. Pour la partie mémoire partagée, il s'agit d'un calcul infini *multi-threadé* essentiellement chargé d'occuper les nœuds. Pour chaque scénario, nous avons donc exécuté les deux types de travaux. Nous avons dégagé quatre scénarios. Deux scénarios concernent les travaux en cours de traitement. Les deux autres concernent les travaux en attente. Les scénarios sont incarnés dans un artefact disponible dans les annexes B. Dans cet artefact, un ensemble de machines virtuelles sont fournies. Sur ces machines, une recette Ansible est présente permettant de déployer le cluster Kubernetes de test. Enfin, un dépôt Git publique localisé sur Github contient tous les Dockerfile de création des images ainsi que les manifests YAML des trois ordonnanceurs évalués. Le Readme.md précise l'intégralité des manipulations effectuées. Nous n'avons pas inclus les manifestes YAML dans le présent document car cela n'est pas nécessaire à la compréhension de cette section. Nous irons plus en détails sur cet aspect dans la section suivante consacrée à OAR.

10.2.2 Résultats

Nous allons commencer par présenter les résultats des scénarios concernant les travaux en attente pour les trois ordonnanceurs évalués. Tous les résultats sont centralisés dans le tableau 10.1. Le premier scénario (1) consistait à mesurer les conséquences de l'ajout de nœuds sur les travaux en attente. Nous commençons par lancer suffisamment de travaux pour saturer les ressources du cluster. Une fois saturé, nous lançons un autre travail, une fois MPI et une autre fois non-MPI, qui se retrouve en attente. Nous ajoutons un nouveau nœud de calcul en incrémentant le paramètre `replica` du déploiement. Le travail en attente est envoyé sur le nouveau nœud et s'exécute sans soucis sur les trois ordonnanceurs.

Ensuite, nous réalisons une seconde manipulation qui va occuper les ressources du cluster de façon à laisser un seul nœud libre. Ensuite, nous soumettons un travail MPI qui va nécessiter deux nœuds. À l'instanciation du nouveau nœud, nous nous retrouvons dans la situation où les ressources sont suffisantes pour exécuter le travail MPI en attente. Sur OpenPBS et OAR, cela fonctionne. Sur Slurm, le travail échoue. En revanche, si on relance le travail juste après, cette fois ci il fonctionne. La raison pour Slurm est que le travail MPI a été lancé avec srun car OpenMPI a été compilé avec le support de Slurm. C'est donc srun qui instancie la couche de communication du travail MPI. L'implémentation actuelle de Slurm ne supporte pas cette opération. En effet, en cas de changement de topologie, il faut redémarrer non seulement slurmctld mais aussi slurmd. Dans les versions à venir, l'ajout dynamique sera géré.

Le second scénario (2) s'intéresse aux comportements des travaux en attente lorsque l'on retire des nœuds libres. Nous lançons quelques travaux pour occuper le cluster en laissant quelques ressources libres. Nous soumettons un travail qui demande plus de ressources que celles disponibles. Nous décrémentons l'attribut `replica` pour réduire le nombre de nœuds de calcul conteneurisés disponibles. Cette manipulation n'a aucun impact sur les travaux en attente car si on recrée suffisamment de ressources pour exécuter les travaux,

Scénarios	SLURM	OpenPBS	OAR
(1) Impact sur les travaux en attente lors de l'ajout de ressources	Echec	Succès	Succès
(2) Impact sur les travaux en attente lors du retrait de ressources	Succès	Succès	Succès
(3) Impact sur les travaux en exécution lors de l'ajout de ressources	Succès	Succès	Succès
(4) Impact sur les travaux en exécution lors du retrait de ressources	Succès	Succès	Succès

Table 10.1 – Résultats du passage à l'échelle

on retombe sur les résultats du scénario (1). Les deux derniers scénarios vont concerner les travaux en exécution. Dans le scénario (3), nous lançons des travaux MPI et non-MPI puis nous ajoutons des nœuds. L'exécution continue sans soucis pour les trois ordonnanceurs. Enfin, dans le scénario (4), nous lançons à nouveau des travaux MPI et non-MPI et nous retirons des nœuds libres. Une fois de plus, aucun effet de bord a été constaté pour les trois ordonnanceurs.

10.2.3 Limitations du contrôleur `StatefulSet`

Dans la section précédente, les travaux ont volontairement été ordonnancés pour mettre en évidence les points potentiellement problématiques dans le passage à l'échelle du centre de calcul. Les scénarios étaient très contrôlés car le contrôleur `StatefulSet` utilisé pour gérer les Pods n'est pas forcément adapté à la gestion de Pods pour le HPC.

Les contrôleurs fournis avec Kubernetes sont très orientés gestion de conteneurs jetables. C'est logique comme approche car ce qui est visé est de garantir l'état de disponibilité d'un service. Les Pods qui sous-tendent ce service se doivent d'être totalement indifférenciés. Kubernetes garantit uniquement un nombre de Pods cible pour un service donné. Dans un contexte HPC, la problématique est un peu différente. Les nœuds conteneurisés ont une identité forte.

En effet, contrairement au contexte traditionnel de l'orchestration de conteneurs, le service à rendre est très lié à l'identité du conteneur. Le `StatefulSet` maintient l'association entre un Pod et son IP. Cependant, on ne peut pas cibler l'ajout ou le retrait d'un Pod en particulier. Par exemple, si nous avons un déploiement de cinq Pods et que le Pod numéro trois est libre et pourrait être supprimé, il est impossible de le faire. Nous rappelons que la gestion des Pods avec le `StatefulSet` se fait en mode LIFO. Si nous voulons être capables de proposer un service HPC avec passage à l'échelle automatique, il va falloir réaliser un développement supplémentaire pour piloter les Pods.

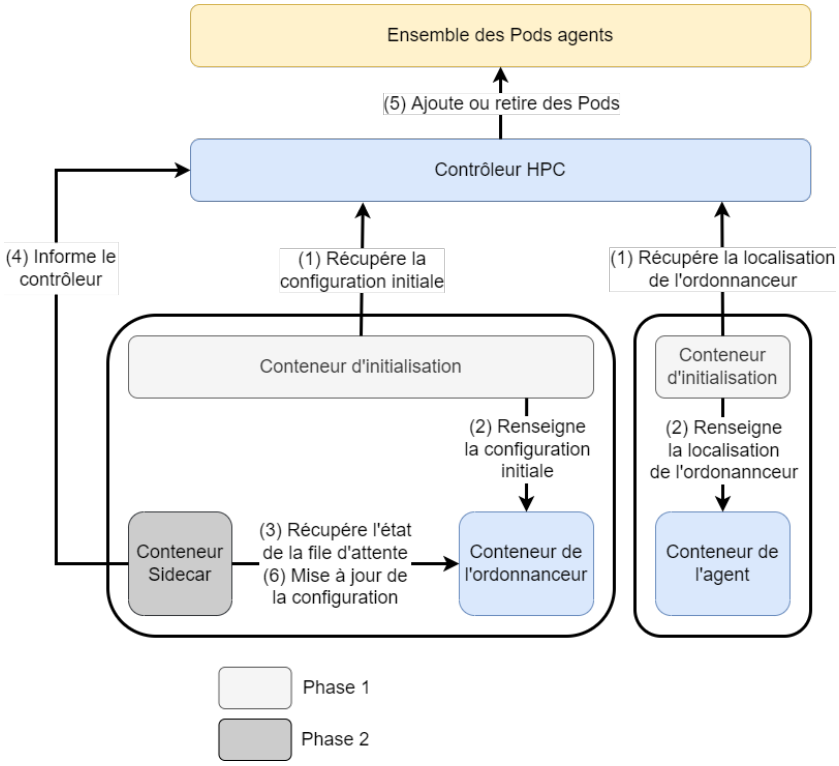


Figure 10.2 – Proposition d'une méthode de passage à l'échelle automatique

10.3 Perspectives

Tout service de passage à l'échelle automatique repose avant tout sur la collecte de métriques afin de prendre la décision de croître ou rétrécir. Dans l'écosystème Kubernetes, ce service de collecte de métriques est souvent assuré par Prometheus. Il est interconnecté avec le contrôleur pour qu'il puisse prendre la décision d'ajouter ou retirer des Pods (dans une certaine mesure). L'objectif suivant est d'arriver au même résultat avec des Pods HPC. Nous allons essayer d'implémenter la méthodologie définie dans la figure 10.2.

Dans cette figure, nous voyons que la métrique observée est la file d'attente des travaux. Par rapport à la méthodologie macro, nous voyons qu'un contrôleur HPC est apparu. Ce contrôleur est renseigné par le conteneur *sidecar* de l'ordonnanceur. Le conteneur *sidecar* surveille la file d'attente. Si des travaux sont en attente et que le nombre maximum de nœuds HPC conteneurisés n'est pas atteint, alors le conteneur *sidecar* demanderait la création de Pods supplémentaires au contrôleur HPC. A contrario, en cas de nœuds HPC conteneurisés libres, le conteneur *sidecar* de l'ordonnanceur demanderait leur suppression. Dans la suite de notre travail, nous avons implémenté cette méthode avec OAR.

CHAPITRE 11

PASSAGE À L'ÉCHELLE AUTOMATIQUE D'OAR

Ce travail fait suite aux conclusions du chapitre 10 précédent. Nous allons décrire comment nous avons implémenté le passage à l'échelle, cette fois ci automatique, avec l'ordonnanceur Oar. Ce travail a été publié dans un article nommé « Autoscaling of Containerized HPC Clusters in the Cloud » [126] présenté au Workshop SuperCompCloud'22 organisé conjointement avec la conférence SuperComputing, édition 2022 (SC'22). Dans l'annexe C, nous présentons un cas d'utilisation de notre travail.

Dans un premier temps, nous présenterons la conteneurisation d'Oar. En effet, nous ne sommes pas rentrés dans le détail dans le chapitre 10. Ensuite, nous discuterons de l'implémentation du contrôleur dédié au HPC. Enfin, nous présenterons l'expérimentation.

11.1 Conteneurisation d'Oar

Nous avons volontairement passé sous silence les détails de configuration des déploiements Kubernetes. Nous profitons de cette section pour donner plus de détails techniques qui sont représentatifs de notre état de connaissance technique sur cet orchestrateur. Ce mode opératoire est complètement transposable aux trois ordonnanceurs étudiés (et sans aucun doute aux autres).

Nous allons commencer par présenter le Pod qui contient le conteneur Almighty (pour mémoire le processus qui ordonnance les travaux et qui pilote le cluster) dont la configuration est présentée (de façon tronquée) dans le listing 11.1. Dans la section containers nous avons les paramètres classiques de l'instanciation d'un conteneur à savoir l'image (image), la commande à exécuter (combo command et args) et une autorisation particulière pour utiliser la capacité CAP_NET_RAW (`securityContext`). En effet Almighty en a besoin pour faire des requêtes ICMP afin de vérifier la disponibilité des nœuds de calcul. Une fois le conteneur Almighty du Pod défini, nous devons ajouter deux niveaux de configuration.

Le premier est au niveau du Pod via les `annotations` et concerne la topologie des

ressources du cluster HPC conteneurisé. Les `annotations` permettent d'ajouter des métadonnées au Pod. Nous rappelons que dans les clusters HPC les nœuds sont groupés par ensembles homogènes. La section des `annotations` contient des couples clés / valeurs. Les clés sont composées de deux parties séparées par « / ». La partie de gauche est le nom de l'ensemble de machines et la partie droite le type de ressource. De cette manière, nous définissons un patron de nœuds conteneurisés pour cet ensemble. Dans notre exemple, les nœuds conteneurisés appartiennent à l'ensemble « default ». On fixe un nombre maximum de trois nœuds (*nodes*) dans le cadre du passage à l'échelle automatique. Chaque nœud de cet ensemble dispose de deux vCPUs (cpuspernode).

Le vCPU est une notion assez floue dans le monde de l'isolation, que ça soit pour la virtualisation ou la conteneurisation. Pour la virtualisation, en général un vCPU représente un cœur de la machine physique en correspondance un pour un. Le monde physique fait de sockets et de cœurs tel que présenté dans la figure 11.1 est translaté vers la dimension plus éthérée des vCPUs comme dans la figure 11.2. Cependant, il peut aussi s'agir de CPUs émulés artificiellement, peu importe le nombre de cœurs réels dédiés à la machine virtuelle. Dans le monde des conteneurs, on parle plutôt d'un quota CPU de 100% au sens des cgroups. Enfin, on trouve l'image à instancier (image) et la racine nom de court de la machine (hostnamebase).

```
apiVersion: v1
kind: Pod
metadata:
  name: hpc-scheduler
  namespace: oar
  annotations:
        default/nodes: "3"
        default/cpuspernode: "2"
        default/image: "nyk0/chsc-oar"
        default/hostnamebase: "hpc-node"
spec:
  containers:
  - image: nyk0/chsc-oar
    name: oar-server
    envFrom:
    - configMapRef:
      name: oarconf
    command: ["/bin/bash"]
    args: ["/start-almighty.sh"]
    securityContext:
      capabilities:
        add: ["NET_RAW"]
```

Listing 11.1 – manifest du Pod Almighty

La seconde directive de configuration est une `configMap` au niveau du conteneur (et

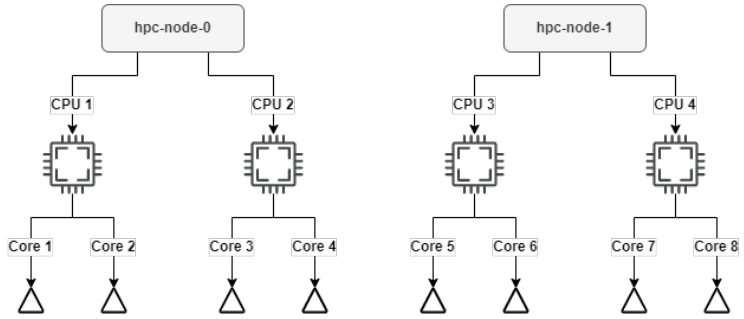


Figure 11.1 – Topologie réelle des ressources

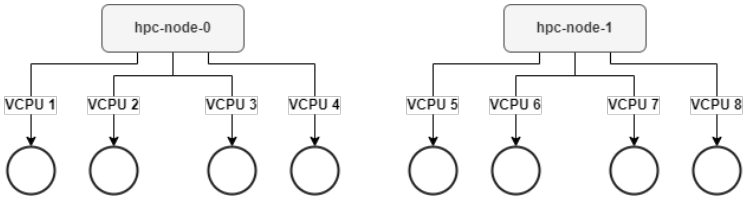


Figure 11.2 – Topologie transposée aux conteneurs des ressources

non plus du Pod). Une `configMap` est un objet dans Kubernetes qui stocke des données de configuration accessible via des variables d'environnement dans le contexte de l'application conteneurisée. Cela facilite la gestion et la modification de la configuration sans nécessité de modifier l'image utilisée dans le déploiement de l'application. Dans le listing 11.2, la `configMap` est attachée au conteneur hébergeant Almighty. On y reconnaît les directives de configuration du fichier `oar.conf` qui est utilisé par Almighty.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: oarconf
  namespace: oar
data:
  DB_TYPE: "Pg"
  DB_HOST: "db-server"
  DB_PORT: "5432"
  DB_Oar_BASE_NAME: "oar"
  DB_Oar_BASE_USERNAME: "oar"
  DB_Oar_BASE_USERPASS: "azerty"
  DB_Oar_BASE_USERNAME_RO: "oar_ro"
  DB_Oar_BASE_USERPASS_RO: "azerty"
  Oar_SERVER_HOSTNAME: "hpc-scheduler"
  ...
```

Listing 11.2 – configMap d'Oar

La dernière brique du déploiement d'Oar est le serveur de base de données. Le listing

11.3 présente le déploiement d'un serveur Postgres de base.

```
apiVersion: v1
kind: Pod
metadata:
  name: db-server
  namespace: oar
spec:
  hostname: db-server
  subdomain: nodes
  containers:
  - image: postgres
    name: postgres
    env:
    - name: POSTGRES_PASSWORD
      value: "azerty"
    - name: POSTGRES_USER
      value: "root"
```

Listing 11.3 – Manifest du serveur Postgres

Contrairement aux travaux précédents, nous ne définissons pas les nœuds de calcul conteneurisés dans le manifeste. Nous exhibons juste leurs spécifications dans les `annotations` du Pod de l'ordonnanceur. En effet, les nœuds sont instanciés à la volée par notre contrôleur HPC qui est notre contribution dans cette section.

11.2 L'implémentation du contrôleur HPC

Dans cette section, nous allons discuter de l'implémentation du contrôleur HPC. Le fonctionnement général du contrôleur est présenté dans la figure 11.3. Le changement majeur par rapport au travail présenté à Europar est que cette fois ci l'instanciation des nœuds est automatique en fonction de la charge du centre de calcul en collaborant avec l'ordonnanceur.

Le rôle de ce contrôleur est double. Il doit d'une part surveiller les `annotations` du Pod Almighty pour voir si des changements sont réalisés dans la topologie des ressources. D'autre part, il doit surveiller la file d'attente de l'ordonnanceur pour instancier ou détruire des nœuds de calcul conteneurisés au besoin. Dans un souci de séparation de privilèges, nous avons choisi d'implémenter ces deux rôles dans deux *threads* distincts. Le *thread* d'initialisation s'occupe de répercuter les changements d'`annotations` du Pod Almighty sur la configuration des ressources dans l'ordonnanceur. Le *thread* de surveillance interroge la file d'attente des travaux dans l'ordonnanceur pour instancier ou supprimer des nœuds de calcul conteneurisés.

Le thread d'initialisation est une boucle infinie qui est initialisée avec un état vide

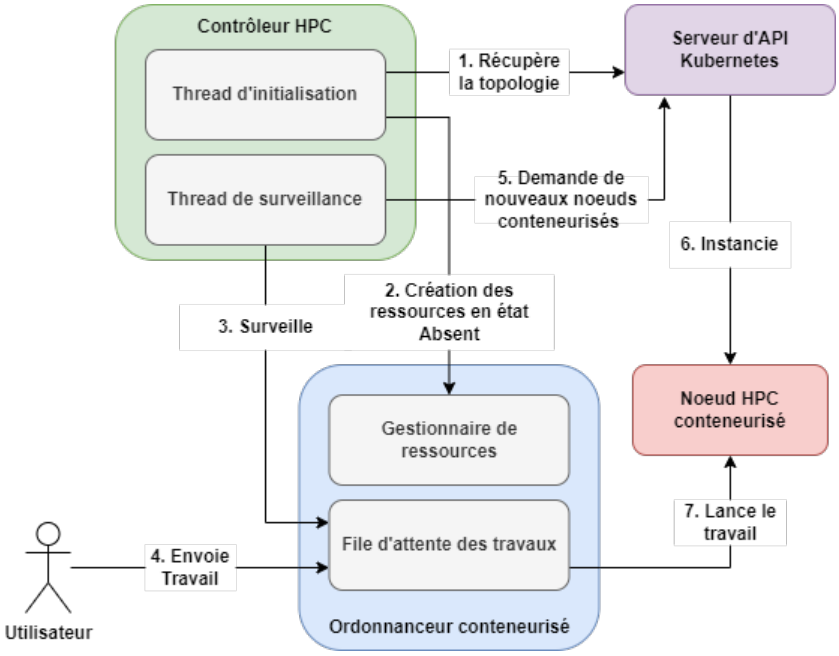


Figure 11.3 – Architecture du contrôleur HPC

et qui compare l'état courant avec l'état précédent. En cas de différence, deux cas se présentent : soit nous ajoutons des ressources dans un ensemble de machines, soit nous en retirons. Si on ajoute des ressources c'est simple, il suffit d'ajouter les nœuds dans la configuration de l'ordonnanceur. Si on en retire, plusieurs choix s'offrent à nous. Soit on redimensionne brutalement le cluster conteneurisé peu importe si des jobs tournent sur les nœuds HPC conteneurisés. Dans ce cas, il faut d'abord placer les nœuds à l'état « Absent », cela va automatiquement interrompre les travaux en cours. C'est le choix que nous avons fait. Une autre option serait de soumettre un travail d'extinction en ciblant les nœuds HPC conteneurisés à éteindre. L'inconvénient est que la suppression des nœuds HPC conteneurisés est différée dans le temps. Une fois que toutes les ressources sont libres, la configuration visée remplace celle en exécution de l'ordonnanceur.

Le thread de surveillance est une boucle infinie sur la file d'attente qui va ajouter des nœuds conteneurisés (dans la limite de la valeur nodes) en cas d'encombrement de la file d'attente. En revanche, si des nœuds HPC conteneurisés sont inoccupés, alors ils sont détruits.

Ce contrôleur HPC est lui-même conteneurisé dans un Pod. Le listing 11.4 présente le manifeste du Pod contrôleur HPC. Il récupère la `configMap` pour localiser le serveur d'API d'Oar et interagir avec la file d'attente et les ressources définies dans l'ordonnanceur. Le script `start-controller.sh` invoqué au démarrage du conteneur démarre les deux *threads* du contrôleur HPC.

```
apiVersion: v1
kind: Pod
metadata:
  name: controller
  namespace: oar
spec:
  containers:
  - image: nyk0/chsc-oar
    name: controller
    envFrom:
    - configMapRef:
      name: oarconf
    command: ["/bin/bash"]
    args: ["/start-controller.sh"]
```

Listing 11.4 – Manifest du contrôleur HPC

Le listing 11.5 présente la fonction d'ajout d'un Pod représentant un nœud de calcul conteneurisé. Le paramètre important est `kind='Pod'` dans la variable `pod_body`. Auparavant, nous utilisions le type un `StatefulSet`. Le type Pod représente une instance simple de Pod. Il n'y a plus de notion de déploiement comme précédemment. Les Pods sont instanciés individuellement et suivis par le contrôleur HPC. En effet, dans la section 5.3.3, nous avons vu que les propriétés du `StatefulSet` n'étaient pas adaptés à la conteneurisation de clusters HPC. L'implémentation de notre contrôleur est disponible en ligne [1].

1. https ://github.com/Nyk0/chsc

```
def createPod(k8sConnect, queueName, queuesDict, k8sNamespace, podName,
    ↪ containerEnv, containerCommand, containerArgs):

 containerResources = client.V1ResourceRequirements( \
  requests={"cpu" : queuesDict[queueName]['cpuspernode']}, \
  limits={"cpu" : queuesDict[queueName]['cpuspernode']})
 containers = []
 container1 = client.V1Container( \
  name='hpc-worker', \
  image=queuesDict[queueName]['image'], \
  resources=containerResources, \
  env=containerEnv, \
  command=containerCommand, \
  args=containerArgs)
 containers.append(container1)
 pod_spec = client.V1PodSpec(containers=containers)
 pod_metadata = client.V1ObjectMeta(
  name=podName, \
  namespace=k8sNamespace)
 pod_body = client.V1Pod(
  api_version='v1', \
  kind='Pod', \
  metadata=pod_metadata, \
  spec=pod_spec)
 k8sConnect.create_namespaced_pod(namespace=k8sNamespace, body=pod_body)
```

Listing 11.5 – Patron du Pod généré par le contrôleur HPC

11.3 Perspectives immédiates

L'implémentation actuelle du contrôleur est un premier jet. Nous allons essayer de monter un projet pour le développer en décorrélant de façon claire le processus de décision de son application à l'ordonnanceur. De plus, dans son implémentation actuelle, le contrôleur interroge directement la file d'attente de l'ordonnanceur pour collecter les métriques. Le processus de collecte devrait être capable de s'interfacer avec différente sources de données potentielles : la file d'attente de l'ordonnanceur en direct (par exemple via son API), les remontées de Prometheus (sous réserve qu'un exporter existe pour l'ordonnanceur concerné) ou tout autre système de supervision. Prometheus est un service centralisant les remontées de métriques diverses et variées issues de contrôleurs. Le contrôleur s'accole à la ressource pour la requêter et récupérer les valeurs brutes. Le contrôleur est un composant spécialisé, c'est-à-dire développé spécialement pour le service à requêter, chargé d'alimenter Prometheus. Le processus de collecte devrait donc implémenter plusieurs modules en fonction de la source de données. La figure 11.4 présente l'architecture

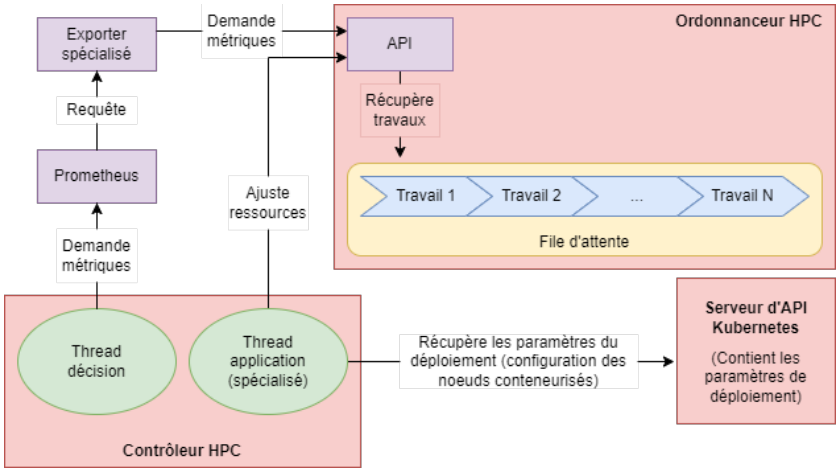


Figure 11.4 – Architecture du contrôleur cible

cible du contrôleur à mettre en place.

On y retrouve un processus représentant le contrôleur HPC. Ce contrôleur est divisé en deux *threads*. Le *thread* de décision requêterait un service Prometheus pour collecter les métriques (données de la file d'attente). Une fois les informations collectées, le *thread* de décision pourra demander la création de ressources au *thread* d'application. Le *thread* d'application connaît l'état actuel du provisionnement via l'interrogation du serveur d'API de Kubernetes. Lorsqu'il reçoit une demande de création de ressources supplémentaires par le *thread* de décision, il peut déclencher la création de ressources, c'est-à-dire l'ajout de nœuds HPC conteneurisés (si la limite dure de passage à l'échelle n'est pas atteinte).

Troisième partie

Conclusion et perspectives

CHAPITRE 12

BILAN, SYNTHÈSE ET ÉLÉMENTS PROSPECTIFS

Dans ce chapitre, nous allons commencer par résumer nos contributions à la fois sur la partie des infrastructures de calcul mais aussi des bénéfices apportés à leurs utilisateurs. Nous insisterons sur l'apport méthodologique de notre travail. Ce résumé sera suivi de la description des perspectives proches et plus lointaines dans le temps de notre travail. La conteneurisation des ordonnanceurs HPC ouvre des perspectives en terme de provisionnement de nœuds de calcul conteneurisés. En effet, nous avons vu que notre travail consistait à offrir un service type « HPC mou », c'est-à-dire que la performance n'est pas forcément notre objectif premier pour privilégier d'autres propriétés. Pour les utilisateurs cela se traduit par une grande souplesse de configuration et l'accès à des services « à la demande » pour compléter l'infrastructure HPC « *on-premise* ». Pour les administrateurs, les perspectives vont surtout tourner autour du provisionnement.

12.1 Résumé des contributions

Notre travail est parti d'observations de terrain et plus précisément de la volonté de conteneuriser Slurm (cf. chapitre 9) afin de proposer un environnement de travail pratiques pour les étudiants que nous formons au HPC. Il prenait donc la forme d'un fichier Docker Compose à distribuer aux étudiants. Nous avons ensuite décidé d'initier un travail complémentaire autour de ce travail empirique. Nous avons commencé par intégrer Slurm à Kubernetes pour évaluer la possibilité de faire fonctionner un Slurm conteneurisé dans le cadre d'une orchestration distribuée [4]. Nous avons instancié l'intégralité d'un cluster HPC dans le cadre d'une expérimentation pour le compte d'un autre travail sur les algorithmes d'ordonnancement de conteneurs dans Kubernetes [87]. Nous avons aussi mené une seconde expérimentation avec une conteneurisation partielle de Slurm. L'objectif était de compléter l'infrastructure physique existante avec des nœuds conteneurisés. Dans ce travail, nous avons mis en avant les aspects cohabitation et reproductibilité. Cependant, une propriété importante du Cloud est l'élasticité. Nos deux contributions suivantes vont dans ce sens.

Ainsi, après le travail autour de Slurm, nous nous sommes posés la question de savoir s'il était possible de redimensionner à chaud un cluster HPC en utilisant la propriété d'élasticité du Cloud pour provisionner dynamiquement des nœuds conteneurisés. Nous avons décidé de formaliser les choses en appliquant une méthode de passage à l'échelle (dans un premier temps manuelle) sur trois ordonnanceurs majeurs du marché OpenSource : Slurm, OpenPBS et Oar. Nous avons mis en lumière que ce passage à l'échelle était possible en suivant la méthode décrite dans [124]. Enfin, nous sommes allés un peu plus loin dans cette approche en ajoutant une dimension automatique au passage à l'échelle en interconnectant le contrôleur avec l'ordonnanceur afin de collecter les données des travaux en attente et ajouter des nœuds de calcul conteneurisés au besoin [126].

12.2 Au-delà de la technique, la méthodologie

Comme nous l'avons vu dans les chapitres 7 (section 7.2) et 8 (section 8.3), il existe plusieurs méthodes pour intégrer le HPC dans le Cloud. Dans la section 7.2, les auteurs décident d'adapter l'ordonnancement des travaux HPC aux orchestrateurs [55]. Notre appréciation est qu'il s'agit plutôt d'une méthode applicable à un horizon assez lointain car il faut prioritairement stabiliser le développement avant de le propager sur les orchestrateurs. Il s'en suivra une phase d'adoption, d'abord par les fournisseurs de Cloud, puis par les utilisateurs.

Par ailleurs, dans la section 8.3 la méthodologie consiste à modifier de façon plus ou moins intrusive l'ordonnanceur HPC pour lui faire prendre conscience qu'il va soit instancier des conteneurs lui même sur un moteur [58], soit dialoguer avec un orchestrateur qui servira d'intermédiaire [110]. Enfin, la dernière méthode évoquée est d'ajouter de la dynamicité aux briques du Cloud. Par exemple, dans [102] nous avons évoqué un travail sur MPI consistant à instancier les conteneurs au moment du `mpiexec`.

Dans notre travail, nous proposons une autre méthode consistant à considérer l'agent de l'ordonnanceur HPC comme un lanceur de travaux HPC à intégrer dans l'image du travail à exécuter. Ainsi, nous avons conteneurisé l'ordonnanceur HPC lui-même et développé un contrôleur permettant à l'orchestrateur de provisionner les conteneurs dérivés de l'image en se basant sur la métrique de la file d'attente (essentiellement les travaux en attente de traitement). Dans un sens, c'est une approche qui ressemble à celle évoquée dans 7.2 car tout repose sur l'orchestrateur comme approvisionneur de conteneurs. L'ordonnanceur HPC n'en a aucune connaissance. Il n'y a pas d'états multiples des travaux à maintenir. Cependant, contrairement à [55] notre méthode n'est pas transparente car il faut intégrer l'ordonnanceur HPC aux images de conteneurs que nous souhaitons instancier. En revanche, nous conservons les capacités avancées en termes de gestion des travaux HPC par les ordonnanceurs spécialisés, ce qui était une limitation soulevée dans [51].

Nous nous différencions également de [58], [110] et [102] dans la mesure où notre méthode ne nécessite aucune modification du code de Kubernetes comme dans [55], de l'ordonnanceur HPC comme dans [110] et [58] ou des briques usuelles telles que MPI comme dans [102].

De plus, le contrôleur permettant le provisionnement des nœuds de calcul conteneu-

risés est lui-même conteneurisé et ne nécessite qu'un accès au serveur d'API du cluster Kubernetes. Cette méthode non intrusive vis à vis des deux composants concernés vise à ouvrir au maximum les possibilités de déploiements dans des environnements hétérogènes pour garantir la meilleure cohabitation possible. C'est une approche qui garde en tête les contraintes liées à l'administration système opérationnelle. Le prix à payer est que la complexité de notre méthode porte sur l'injection de l'agent de l'ordonnanceur HPC dans les images.

12.3 Perspectives

Pour évoquer les perspectives de notre travail, nous allons dégager deux temporalités : moyen terme et long terme. Pour chacune de ces deux temporalités, nous dégagerons des perspectives vues du côté utilisateur et du côté infrastructure.

12.3.1 Perspectives à moyen terme

Les perspectives à moyen terme identifiées sont de permettre à l'utilisateur d'intégrer l'ordonnanceur HPC dans les images des logiciels utilisés dans ses travaux. Nous sommes persuadés qu'il existe beaucoup de système qui pourraient bénéficier de notre travail. Par exemple, du point de vue de l'infrastructure, nous pouvons améliorer l'implémentation du contrôleur HPC pour Kubernetes et tester l'intégration de Melissa [127] pour que les travaux des utilisateurs intègrent les propriétés proposées par notre contribution. Nous détaillons maintenant cette idée.

Point de vue utilisateur

Un des points faibles de notre travail est que nos images sont construites de la même manière que nous installerions un nœud de calcul physique. C'est à dire qu'on a un système de base, l'agent de l'ordonnanceur et la pile de logiciels scientifiques. La constitution d'une image est une étape clé de la gestion des conteneurs. Les images fonctionnent avec un système de couches. Si on multiplie les couches, l'espace occupé par l'image grossira en conséquence. Cela rend l'image plus longue à déployer, elle prend plus de place partout ou elle doit être déployée. Il faut donc la gérer de façon plus efficace.

La gestion des différentes versions de logiciels avec les librairies sous-jacentes peut devenir extrêmement complexe. Il faut multiplier les options de compilation pour appeler telle bibliothèque plutôt que telle autre, activer telle ou telle option etc. De plus, dans le cas de codes en développement, il peut arriver que l'option pour choisir l'emplacement de la bibliothèque à lier à l'exécutable ne soit pas disponible. En résumé, plus le cluster grossit, plus la population croît, plus la complexité des chaînes de compilation est grande. Notre idée est d'envisager les choses autrement et d'appréhender chaque logiciel comme une brique indépendante dans laquelle on injecte l'agent de l'ordonnanceur HPC.

Du point de vue fonctionnel, l'agent sur le nœud de calcul qui exécute le code est

simplement un lanceur. Il a peu de dépendances, et il a une empreinte très faible. Ainsi, on peut changer de point de vue et dire qu'au lieu de partir d'un système d'exploitation vierge et d'ajouter les couches une par une pour obtenir un système fonctionnel, nous injectons l'agent de l'ordonnanceur HPC dans l'image du logiciel scientifique. Nous pourrions donc croiser les approches de [99] et [128] pour 1) ajouter un attribut de sélection d'image à la manière de [99] comme ressource du travail a lancer au même titre que le nombre de cœurs, quantité de mémoire etc. 2) mettre en place un système de provisionnement d'image (*pre-fetch*) à la manière de [128] qui injecte l'agent de l'ordonnanceur adapté au cluster HPC dans l'image pendant que le travail est en attente. Avec une telle fonctionnalité, l'utilisateur peut inclure l'ordonnanceur HPC avec sa configuration dans la ou les images nécessaires pour le fonctionnement de son code afin d'en accroître la reproductibilité.

Point de vue infrastructure

Dans son implémentation actuelle, notre contrôleur est monolithique et adhérent à l'ordonnanceur HPC conteneurisé. Nous proposons de travailler sur l'architecture cible présentée dans la figure 11.4 pour implémenter un contrôleur HPC pour Kubernetes qui suit les principes de la séparation de privilèges et du moindre privilège. Le processus de décision et son application devront être désolidarisés de l'ordonnanceur HPC par l'ajout d'un système de module afin de limiter les déduplication de code.

Nous pourrions également tirer partie de Melissa [127], qui est un *framework* limitant la génération de fichiers intermédiaires prévu pour s'intégrer dans les clusters HPC. La conséquence immédiate est que les entrées / sorties disques, et par conséquent réseaux puisque que les clusters HPC s'appuient sur des systèmes de fichiers distribués, sont moindres, ce qui viendrait à mitiger l'impact des communications. Les programmes voulant tirer partie de Melissa doivent être instrumentés avec sa bibliothèque. Les propriétés d'adaptabilité, d'ubiquité, de tolérances aux fautes et d'élasticité résonnent avec notre travail. L'intégration avec ce *framework* pourrait permettre au travail de l'utilisateur de s'adapter aux redimensionnements du cluster HPC conteneurisé en limitant l'impact sur les performances.

12.3.2 Perspectives à long terme

Dans les perspectives à plus long terme, nous pourrions nous appuyer sur notre travail, qui potentiellement permet de rejouer d'anciennes expérimentations dans leur environnement de l'époque. Nous nous poserons également la question de savoir si, au niveau infrastructure, les ordonnanceurs HPC tels que nous les connaissons en 2023 vont perdurer.

Point de vue utilisateur

Côté utilisateur, comme perspective à long terme, nous pouvons imaginer rejouer de très vieilles expérimentations avec leur pile logicielle d'époque (ordonnanceur inclus). Le point critique dans ce cas est de pouvoir gérer le type de processeur utilisé à l'époque. Il

existe des systèmes comme Qemu [129] qui permettent d'émuler des processeurs. Dans la section 4.4, nous avons vu que des alternatives au runC standard sont disponibles dont certaines s'appuient sur le couple KVM/Qemu au prix d'un impact relativement faible sur les performances [130].

Nous souhaitons insister sur le fait que la combinaison de notre contribution actuelle avec une alternative à runC proposant l'émulation de processeurs pourrait nous permettre de rejouer sur des infrastructures Cloud actuelles d'anciennes expérimentations HPC avec leur environnement logiciel et CPU de l'époque.

Bientôt, en France, le nouveau et premier cluster national exascale sera déployé au TGCC (Très Grand Centre de Calcul) du CEA (Commissariat à l'Energie Atomique). Ne pourrait-on pas envisager d'utiliser nos techniques et notre méthodologie pour réserver des nœuds permettant de déployer, à la volée, l'actuel cluster Jean Zay, ou les précédents (Turing et Ada), ou le précédent du précédent ? Cela permettrait d'assurer un certain niveau d'exigence pour la reproductibilité des expériences passées même s'il s'agissait de version réduite de ces clusters.

Point de vue infrastructure

A notre sens, la question majeure que pose à long terme notre travail est « Les ordonnanceurs HPC traditionnels vont-ils continuer à exister ? ». En effet, dans notre travail, nous utilisons les orchestrateurs comme un mécanisme efficace de provisionnement. Cependant, dans les travaux présentés dans la section 7.2, nous voyons que la tendance n'est pas vraiment à la complémentarité mais plutôt à l'hybridation, voir au remplacement des ordonnanceurs HPC par des orchestrateurs Cloud.

Nous pensons que la voie vers le HPC dans le Cloud passera par l'intégration des travaux HPC au sein des orchestrateurs. En effet, les orchestrateurs sont pensés pour intégrer plusieurs mécanismes d'ordonnancement et l'utilisateur peut le choisir au moment de la soumission de son travail. Comme nous le soulignions précédemment, il faut cependant que le fournisseur de Cloud l'intègre dans son orchestrateur. Pour aller dans ce sens, RedHat va intégrer KubeFlux dans sa distribution Kubernetes nommée Openshift [1]. Cette intégration montre la volonté d'un acteur majeur du marché de se positionner sur le HPC (ils n'avaient pas d'offre à proprement parler) en misant sur une solution « Full Cloud ».

Nous pensons donc que l'avenir est plutôt dans cette direction car ce qui compte au final c'est le confort de l'utilisateur. En faisant prendre en charge les *workflows* HPC par des orchestrateurs Cloud, l'effort à fournir par l'utilisateur est minime. Il doit convertir son script de soumission en manifeste pour l'orchestrateur. C'est plus ou moins le même travail que lorsqu'on passe d'un ordonnanceur HPC à un autre. Seulement, le changement d'ordonnanceur HPC concerne surtout les administrateurs des ressources. Pour les utilisateurs, à part l'effort (minime ?) pour convertir leurs scripts, cela ne change rien. L'adoption d'un modèle de soumission directement dans le Cloud est plus facile à justifier car l'utilisateur aura accès à une variété de ressources et une élasticité impossible dans un environnement HPC statique.

1. https ://adac11.cscs.ch/kubeflux-a-scheduler-plugin-bridging-the-cloud-hpc-gap-in-kubernetes

Dans ce changement de paradigme, notre travail peut être envisagé comme un accompagnement. Comme il ne restera vraisemblablement que les infrastructures Cloud pilotées par un orchestrateur, nous pourrons perpétuer un usage classique au-dessus des ressources Cloud sans surcoût d'une architecture parallèle. De notre point de vue, l'avenir pour les décennies à venir semble donc être plutôt du côté des implémentations d'ordonnanceurs adaptés au HPC dans les orchestrateurs.

Quatrième partie

Annexes

ANNEXE A

LES PROCESSUS SOUS LINUX

A.1 Structure de données

Le noyau organise les processus en liste doublement chaînée circulaire. Chaque maillon de cette chaîne est un descripteur de processus (*process descriptor*) représenté par une structure nommée `task_struct()`. Cette structure est définie dans le fichier `<linux/sched.h>` des sources du noyau Linux et contient les informations générique des processus, c'est à dire non liées à une architecture matérielle en particulier. On y retrouve toutes les informations d'un processus donné telles que l'identité sous laquelle il s'exécute, les fichiers ouverts, son espace mémoire, son *PID* (*Process IDentifier*, soit un numéro qui l'identifie sans ambiguïtés) etc. Cette structure est allouée par le *slab*, qui est un mécanisme d'allocation mémoire du noyau qui crée différents caches. Chaque cache est dédié à un type d'objet tel que la `task_struct()`, *inode* etc. Avant de créer une nouvelle instance de l'objet en mémoire, le *slab* regarde s'il n'y a pas un objet de même type qui n'est plus utilisé en cache. En conséquence, lorsque le *slab* libère un objet en mémoire, il ne le supprime pas mais il le marque comme ré-allouable. Ce mécanisme est donc bien adapté à l'instanciation des processus car c'est une structure de données qui est fréquemment allouée et libérée.

Il existe une seconde structure de données liée au processus qui se nomme `thread_info` est qui est propre à chaque architecture processeur. Pour les processeurs x86, on la retrouve dans </arch/x86/include/asm/thread_info.h> car cette structure de données, de taille très petite, contient les informations du processus propres à chaque architecture processeur. Le lien entre ces deux structures est propre à chaque architecture. En effet, selon les architectures il existe des façons plus ou moins optimales d'associer les deux structures. Pour voir comment cela se passe, il faut regarder l'implémentation des fonctions `get_current()` ou `get_current_thread_info()`. Certaines architectures comme les Alpha ou Sparc ont une référence à la structure `task_struct()` dans `thread_info()` et gardent un pointeur vers la structure `thread_info()` dans un registre du processeur. Les PowerPC gardent un pointeur vers la structure `task_struct()` dans un registre du pro-

cesseur. Sur x86 chaque cœur du processeur possède un jeu de variable associé (Per-CPU variables).

L'une d'entre elles se nomme `current_task`. Cette variable est un pointeur vers la structure `task_struct()`. Dans le fichier </arch/x86/include/asm/current.h> on trouve une fonction `get_current()` qui invoque `this_cpu_read_stable()` pour récupérer l'adresse de la tâche en exécution sur le processeur. Cette variable est évidemment mise à jour lorsqu'un nouveau processus est ordonnancé sur le processeur durant la phase de changement de contexte, plus précisément lorsque la fonction `__switch_to()` du fichier `</arch/x86/kernel/proces` est invoquée.

Pour ce qui concerne la structure `thread_info` associée elle n'a donc pas besoin d'avoir de référence vers la structure `task_struct()` et se trouve stockée sur 16 octets occupant exactement une ligne de cache au fond du tas du noyau (kernel stack). Ainsi, il n'est pas nécessaire de gâcher un registre processeur dans la mesure où son emplacement est calculable. L'allocation de cette structure ne passe pas par le *slab* car elle est petite. En conséquence, l'autre intérêt de scinder la description d'un processus en deux structures est qu'elles sont indépendantes au niveau du paradigme d'allocation mémoire.

Une autre structure intéressante liée au processus est la structure *PID* définie dans le fichier `</include/linux/pid.h>`. Nous avons vu que la structure `task_struct()` intègre une variable `pid` de type `pid_t`. Cependant, en interne, le noyau va manipuler la structure `pid`. Cette structure est liée aux processus, groupe de processus ou sessions. Cette structure maintient une table de hachage qui permet de retrouver très facilement les processus par leur *PID*. Cette structure est un compromis entre le fait de stocker uniquement une variable de type `pid_t` pour chaque processus (qui ne garantie pas que le processus que vous avez identifié est celui que vous voulez car les *PID* sont réutilisés de façon cyclique) et celui de stocker un pointeur vers la structure `task_struct()` pour chaque *thread* (ce qui laisserait de volumineuses structures `task_struct()` inutilisées dans le tas du noyau). Ces deux structures sont stockées dans la partie « tas » de l'espace noyau de l'espace mémoire virtuel du processus.

A.2 Ordonnancement des processus

Le composant du noyau qui s'occupe de partager les ressources entre les différents processus se nomme l'ordonnanceur (ou *scheduler* en anglais). Avec chaque ordonnanceur vient un algorithme de décision d'octroi des ressources aux processus. Dans les systèmes Linux actuels, les processus sont ordonnancés en mode multitâche préemptif (*preemptive multitasking*). Dans ce mode, l'ordonnanceur décide du moment où le processus doit arrêter d'utiliser le processeur. On dit alors que l'ordonnanceur préempte le processus. Le temps processeur octroyé au processus est appelé « `timeslice` » (il n y a pas vraiment d'équivalent en français).

Durant ce temps, le processus dispose d'un accès exclusif au processeur. Lorsque qu'un processus cède la place à un autre, une commutation de mot d'état (ou *context switching*) est déclenchée. Cette commutation réalise une sauvegarde du contexte du processus en cours d'exécution pour le restaurer lorsqu'il aura à nouveau accès au processeur.

Ce contexte est composé de son état, la valeur des registres actifs, le compteur ordinal, les valeurs des variables globales statiques ou dynamiques, son entrée dans la table des processus, sa zone u (*u-area*), les piles utilisateur et système et enfin les zones de code et de données. Cette énumération est réalisée pour faire prendre conscience au lecteur que cette commutation a un coût qui est inclus dans le « *timeslice* ».

Le « *timeslice* » permet non seulement à l'ordonnanceur d'organiser le partage du temps processeur mais aussi d'éviter qu'un processus s'accapare toutes les ressources. Dans la plupart des systèmes d'exploitation le « `timeslice` » octroyé par l'ordonnanceur à un processus est calculé dynamiquement en fonction de la charge du système pondérée éventuellement par une priorité fixée depuis l'espace utilisateur (commande `nice`). Ce mode de fonctionnement est souvent qualifié d'équitable (ou « *fair* » en anglais pour « *fair scheduling* »).

L'ordonnanceur de Linux est nommé CFS (Completely Fair Scheduler). L'idée derrière CFS est de modéliser un partage de processeur en module multitâche parfait. Chaque processus reçoit une portion $1/n$ du temps processeur avec :

$$n = nombre\ de\ processus\ en\ etat\ TASK\_RUNNING$$

C'est un état cible de référence vers lequel CFS va tenter de converger. Comme plusieurs processus ne peuvent s'exécuter en même temps sur un processeur, ce rapport $1/n$ concerne le « *timeslice* ». On fixe donc un temps de latence cible x, ainsi le « *timeslice* » cible de chaque processus est x/n. La priorité vient pondérer ce rapport $1/n$, elle ne fixe pas de bonus ou malus absolu de « `timeslice` ». Cependant, CFS fixe un « *timeslice* » minimal t pour le processus. Si on se retrouve dans la situation $x/n < t$, le système est en surcharge et la priorité est ignorée.

La décision de préemption d'un processus se base sur son historique de communication nommé *vruntime* (*virtual runtime*). Lorsqu'un processus a utilisé son x/n « *timeslice* », l'ordonnanceur initie la commutation de mot d'état au bénéfice du processus en état `TASK_RUNNING` disposant du *vruntime* le plus bas. La fréquence de commutation de mot d'état dépend de x, n et t. Le nombre maximal de commutation est x/t. Si jamais ce rapport n'est plus tenable au regard de la charge du système, alors CFS va augmenter la latence cible x. Enfin, l'équilibre de commutation entre les processus interactifs (sensibles aux E/S, dits « *I/O bound* ») tel que les traitements de texte et ceux gourmands en processeurs (dits « *processor bound* ») tel qu'un compresseur MP3 va s'équilibrer naturellement grâce au *vruntime*. Un traitement de texte en attente d'une saisie clavier va être en état `TASK_INTERRUPTIBLE`. En conséquence, son *vruntime* ne va pas s'accroître. Donc, lorsqu'une saisie s'opère, le processus va passer devant tout le monde pour occuper le processeur et satisfaire la contrainte « interactive » inhérente à l'application.

Nous avons donc vu que l'essentiel de l'activité du processus se déroulait dans le noyau. Cependant, le processus est créé en espace utilisateur. C'est bien l'utilisateur du système qui lance les programmes qui deviennent des processus. Il existe donc une interface de communication entre l'espace utilisateur et l'espace noyau. Il s'agit des appels systèmes.

A.3 Allocation mémoire

Chaque processus dispose d'un espace mémoire virtuel qui lui est propre. Cet espace mémoire privé contient neuf sections organisées en deux espaces distincts : l'espace noyau et l'espace utilisateur. L'espace noyau est inaccessible en direct depuis l'espace utilisateur mais par des appels système. Pour l'espace noyau :

- Structures de données du processus : cette zone mémoire propre au processus contient les structures sus-nommées ;
- Mémoire physique : table d'adressage direct de la mémoire physique. C'est un moyen simple d'accéder rapidement aux pages physiques de la mémoire ;
- Code et données du noyau : cette zone contient les structures de données du noyau communes à tous les processus.

Les deux dernières zones sont les mêmes pour tous les processus. Pour l'espace utilisateur :

- Tas : cette zone contient les données générés automatiquement lors du cycle de vie du processus, par exemple le pointeur de retour vers le programme principale en sortie de fonction ;
- Bibliothèques partagées : zone de projection des bibliothèques partagées ;
- Pile : allocation mémoire dynamique des variables ;
- BSS : variables statiques non initialisées (remplie de zéros) ;
- DATA : variables statiques initialisées ;
- TEXT : code machine executable.

La figure A.1 présente l'organisation de ces sections. Nous nous référerons à cette représentation dans la suite de ce mémoire.

La MMU (*Memory Management Unit*) est un composant physique généralement intégré au processeur. Il traduit les adresses virtuelles (également appelées adresses linéaires dans le monde x86) en adresses physiques. Optionnellement, il peut également appliquer le contrôle d'accès à la mémoire, le contrôle du cache et l'arbitrage de bus. Il n'a généralement pas sa propre mémoire, il s'appuie sur les données de la mémoire principale du système pour fonctionner.

La MMU effectue cette traduction en utilisant des informations stockées dans des structures de données telles que des tables de pages. Le noyau représente la mémoire physique comme un ensemble de pages. La structure *page* est définie dans le fichier `</include/linux/mm_types.h>`. Chaque *page* fait 8 Ko sur les architectures 64 bits. Dans cette structure, nous trouvons l'attribut flags qui renseigne sur l'état de la *page* (sale, verrouillée etc.), l'attribut `_count` qui donne le nombre de fois qu'une *page* physique est référencée (si la valeur est négative, la page est libre) enfin l'attribut `virtual` contient l'adresse de la *page* virtuelle correspondante.

Le processeur fonctionne toujours sur des adresses virtuelles qui sont traduites en adresses physiques par la MMU, mais le noyau est conscient des traductions et s'appuie sur la MMU pour les interpréter. Ce mécanisme de traduction est transparent pour les processus de l'espace utilisateur et surtout ils ne peuvent généralement pas accéder à la table des correspondances. Cependant, sous Linux, les processus disposant de privilèges suffisants peuvent voir leur carte physique dans `/proc/pid/pagemap`.

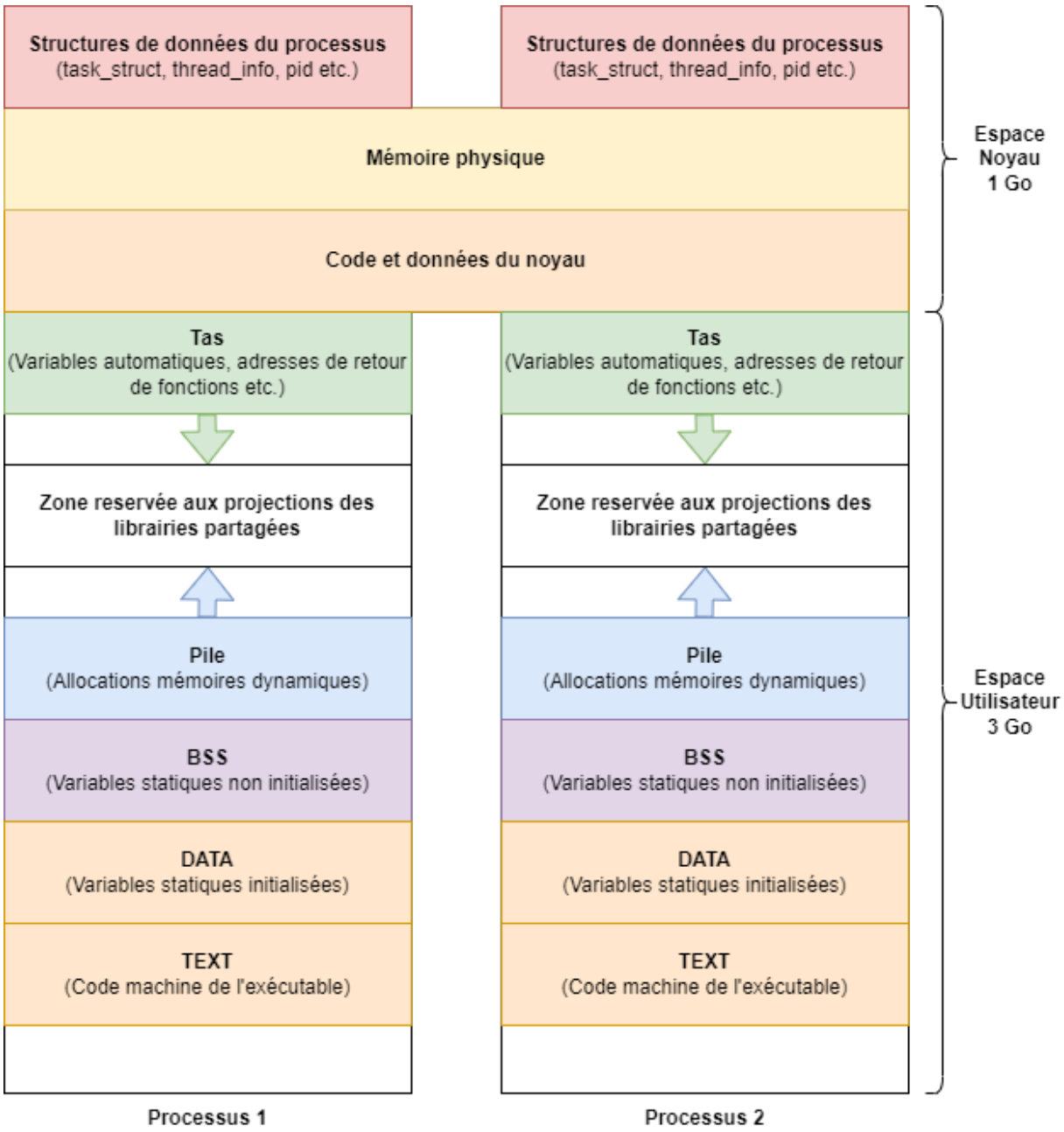


Figure A.1 – Mémoire virtuelle du processus

La MMU accède donc à un tableau qui décrit comment traduire les adresses virtuelles en adresses physiques. Elle n'est pas consciente du processus auquel la translation se réfère. Cependant, il existe un cache associatif nommé TLB (*Translation Lookaside Buffer*) qui l'est. Le rôle de ce cache est de mémoriser une certaine profondeur d'association. Dans les versions plus anciennes du noyau Linux, il était détruit à chaque changement de contexte du processus (cf section A.2) car il était inconscient du propriétaire d'une association. Dans les versions récentes, le TLB intègre le *PID* du processus via la fonction PCID (*Process Context IDentifiers*) des processeurs récents.

A.4 Les IPC (*Inter Process Communications*)

Les IPC sont des mécanismes permettant à des processus différents de communiquer et de partager des ressources entre eux. Il existe plusieurs mécanismes d'IPC disponibles sous Linux :

— Les files de messages (*message queues*) qui permettent à des processus de communiquer en envoyant et en recevant des messages de taille définie dans une file de messages partagée. Chaque message est identifié par une clé unique et peut être lu par un seul processus à la fois ;
— Les sémaphores qui synchronisent l'accès à des ressources partagées entre plusieurs processus. Ils permettent de limiter le nombre de processus accédants à une ressource simultanément et d'attendre que la ressource soit disponible avant d'y accéder. En programmation parallèle, c'est comme ça que les accès concurrents à une ressources sont gérés ;
— Les mémoires partagées (*shared memory*) qui permettent à plusieurs processus d'accéder à une même région de la mémoire. Cela permet de partager des données entre les processus sans avoir à copier les données d'un processus à l'autre ;
— Les signaux (*signals*) utilisés pour notifier un processus de l'occurrence d'un événement particulier. Les signaux peuvent être envoyés à un processus depuis un autre processus ;
— Les sockets qui permettent à des processus de communiquer à travers un réseau (*socket* réseau) ou localement via un fichier spécial qu'on appelle *socket* local (ou *socket* UNIX) sur le système de fichiers local. Les *sockets* peuvent donc être utilisés pour échanger des données entre des processus distants ou locaux.

Ces différents mécanismes ne s'excluent pas les uns les autres. Ils peuvent être utilisés en combinaison pour permettre à des processus de communiquer et de partager des ressources. En général, les IPC sont utilisés pour les applications multi-processus tels que les services (qui doivent multiplier les processus pour absorber la charge) et les codes de calculs parallèles.

A.5 État des processus et signaux

Comme les processus sont des entités actives en attente de ressources, ils peuvent être dans différents états. Dans la structure de données `task_struct()` on trouve un champ *state*. Ce champ renseigne sur l'état du processus. On trouve cinq états possibles :

- `TASK_RUNNING` : le processus est actuellement en exécution ou en attente d'ordonnancement par le noyau. C'est le seul état possible d'un processus en exécution dans l'espace utilisateur ;
- `TASK_INTERRUPTIBLE` : le processus dort en attendant qu'une certaine condition soit satisfaite. Lorsque la condition est satisfaite, le noyau passe ce processus à l'état `TASK_RUNNING`. Le processus peut également être réveillé à réception d'un signal ;
- `TASK_UNINTERRUPTIBLE` : cet état est similaire au précédent sauf que le processus ne se réveillera pas à réception d'un signal. C'est utilisé lorsque la condition va ra*PID*ement être satisfaite ;
- `__TASK_TRACED` : le processus est tracé par un autre processus (comme un debugger) ;
- `__TASK_STOPPED` : l'exécution du processus est arrêtée. Il ne s'exécutera pas et ne sera pas mis en attente d'exécution. En clair, il ne sera plus dans l'état `TASK_RUNNING`.

Dans la description des états de processus, nous avons évoqué la notion de signal. Un signal est un mécanisme utilisé par le système d'exploitation pour informer un processus de certains événements qui se produisent dans le système tels qu'une erreur, une interruption matérielle ou une demande de terminaison. Les signaux sont généralement utilisés pour permettre à un processus de gérer ces événements de manière appropriée.

Le gestionnaire de signaux est un mécanisme qui permet à un processus de définir des actions spécifiques à prendre en réponse à des signaux particuliers. Chaque signal est identifié par un numéro unique et peut être associé à un traitement personnalisé, tel que la fermeture d'un fichier ou la libération de ressources. Le processus peut également choisir d'ignorer certains signaux ou de les traiter par défaut.

Les signaux classiques les plus couramment utilisés dans les systèmes Unix et Linux sont :

- `SIGINT` (signal d'interruption) : Ce signal est envoyé à un processus par exemple lorsqu'un utilisateur appuie sur la touche Ctrl+C. Le processus peut choisir de terminer ou de continuer à s'exécuter ;
- `SIGTERM` (signal de terminaison) : Ce signal est envoyé à un processus pour demander une terminaison propre. Le processus doit libérer toutes les ressources allouées et terminer son exécution ;
- `SIGKILL` (signal de tuer) : Ce signal est envoyé à un processus pour le forcer à se terminer immédiatement sans possibilité de traitement personnalisé ;
- `SIGPIPE` (signal de tuyau cassé) : Ce signal est envoyé à un processus lorsqu'il tente d'écrire sur un tuyau qui a été fermé par le processus de lecture. Le processus peut choisir de gérer le signal ou de le laisser être traité par défaut ;
- `SIGSEGV` (signal de violation de segmentation) : Ce signal est envoyé à un processus

lorsqu'il tente d'accéder à une zone mémoire non autorisée. Le processus doit traiter le signal pour éviter une erreur fatale.

Il existe de nombreux autres signaux disponibles, chacun avec un numéro unique et un comportement spécifique. Les processus peuvent utiliser le gestionnaire de signaux pour définir des actions personnalisées en réponse à ces signaux ou pour les traiter par défaut. Il existe d'ailleurs deux signaux personnalisables :

— `SIGUSR1` (signal utilisateur 1) : Ce signal est envoyé à un processus par un utilisateur ou un autre processus pour une utilisation personnalisée. Le processus peut définir une action spécifique à prendre en réponse à ce signal ;
— `SIGUSR2` (signal utilisateur 2) : Ce signal est également utilisé pour une utilisation personnalisée. Le processus peut définir une action spécifique à prendre en réponse à ce signal.

Les signaux `SIGUSR1` et `SIGUSR2` peuvent être utilisés par un processus pour communiquer avec un autre processus ou pour déclencher une action spécifique dans le processus lui-même. Ces signaux ne sont pas réservés à un usage particulier par le système d'exploitation et sont souvent utilisés par les applications pour implémenter des fonctionnalités personnalisées.

A.6 Suivi de la fonction `getentropy()`

Dans cette annexe, nous allons suivre l'exécution de la fonction `getentropy(void *buffer, size_t length)` de la GlibC. Cette fonction écrit `length` octets aléatoire dans le paramètre de sortie `buffer`. Cette fonction s'appuie sur `/dev/urandom` qui, comme son emplacement l'indique, est un pseudo-device. L'accès à ce pseudo-device ne peut donc se faire en direct depuis l'espace utilisateur, il va falloir invoquer un appel système.

La GlibC (*GNU C Library*) est une bibliothèque de fonctions C utilisée sur les systèmes d'exploitation Linux et d'autres systèmes d'exploitation basés sur Unix. Elle fournit un ensemble de fonctions qui permettent aux programmes d'interagir avec le système d'exploitation, en utilisant notamment des appels systèmes. La glibc fournit une API standardisée pour effectuer des appels systèmes. Cette interface est utilisée par les programmes pour accéder aux fonctions systèmes, telles que l'ouverture et la lecture de fichiers, la création de processus et la gestion des *sockets* de réseau. La glibc encapsule les détails de bas niveau des appels systèmes dans des fonctions de plus haut niveau, facilitant ainsi la programmation des applications.

Lorsqu'on souhaite utiliser la fonction `getentropy()` il faut inclure la bibliothèque `unistd.h` qui contient le prototype de la fonction. Une fois le code source compilé, l'exécutable est lié à la bibliothèque partagée de la GlibC (`libc.so.6` sous Linux). Une bibliothèque partagée contient le code exécutable des fonctions utilisées dans les programmes qui incluent les *headers* associée à cette bibliothèque. Comme on peut le voir dans la figure A.2, la GlibC possède un fichier `getentropy.c` qui contient le code de la fonction `getentropy()`. Dans ce code, on trouve un appel à la macro `INLINE_SYSCALL_CALL`. Cette macro exécute l'appel système `getrandom()` avec trois arguments : l'adresse du paramètre

de sortie qui est un buffer contenant la suite d'octets tirés aléatoirement, la taille de ce buffer et d'éventuels flags (ou zéro). Si on tire le fil de cette macro dans les sources de la GlibC, on arrive sur la macro `internal_syscall3` définie dans le listing A.1.

```
#Fichier : /glibc/glibc-2.34/source/sysdeps/unix/sysv/linux/x86_64/sysdep
   ↪ .h
#undef internal_syscall3
#define internal_syscall3(number, arg1, arg2, arg3 \
({ \
        unsigned long int resultvar; \
        TYPEFY (arg3, __arg3) = ARGIFY (arg3); \
        TYPEFY (arg2, __arg2) = ARGIFY (arg2); \
        TYPEFY (arg1, __arg1) = ARGIFY (arg1); \
        register TYPEFY (arg3, _a3) asm ("rdx") = __arg3; \
        register TYPEFY (arg2, _a2) asm ("rsi") = __arg2; \
        register TYPEFY (arg1, _a1) asm ("rdi") = __arg1; \
        asm volatile ( \
        "syscall\n\t" \
        : "=a" (resultvar) \
        : "0" (number), "r" (_a1), "r" (_a2), "r" (_a3) \
        : "memory", REGISTERS_CLOBBERED_BY_SYSCALL); \
        (long int) resultvar;      \
})
```

Listing A.1 – Macro *internal_syscall3*

La nomenclature des macros exécutant des appels systèmes est toujours la même. Elle est conditionnée par le nombre d'arguments qu'on passe à l'appel système. Ainsi, `getrandom()` ayant trois arguments, la macro invoquée est `internal_syscall3`. Les variables de l'appel système sont préparées par une suite de `TYPEFY`.

Ensuite, Cette macro positionne les trois arguments dans trois registres du processeur respectivement `rdx`, `rsi` et `rdi` grâce au mot-clé `register`. Un fois les registres préparés, nous voyons la section `asm volatile` faire un appel à l'instruction assembleur `syscall` qui va exécuter l'appel système correspondant à la variable `number` représentant l'appel système et qui se trouve dans le registre `rax`. En effet, chaque appel système est identifié au niveau du noyau par un nombre entier. La table se trouve dans le fichier `/arch/x86/entry/syscalls/syscall_64.tbl`. Ce fonctionnement est celui des architectures courantes 64 bits x86_64. Sur les architectures plus anciennes 32 bits i386 l'exécution d'un appel système était demandée par la levée d'une interruption logicielle de numéro 128 soit l'instruction `int $0x80` avec le registre `eax` qui devait contenir le numéro de l'appel système.

Il existe une alternative à l'appel à `syscall()` (x86_64) ou l'interruption 128 (i386) qui est portée par Intel. Cette alternative pour l'invocation des appels système se nomme `sysenter()`. En effet la fonction `syscall()` est une invention d'AMD. Comme nous

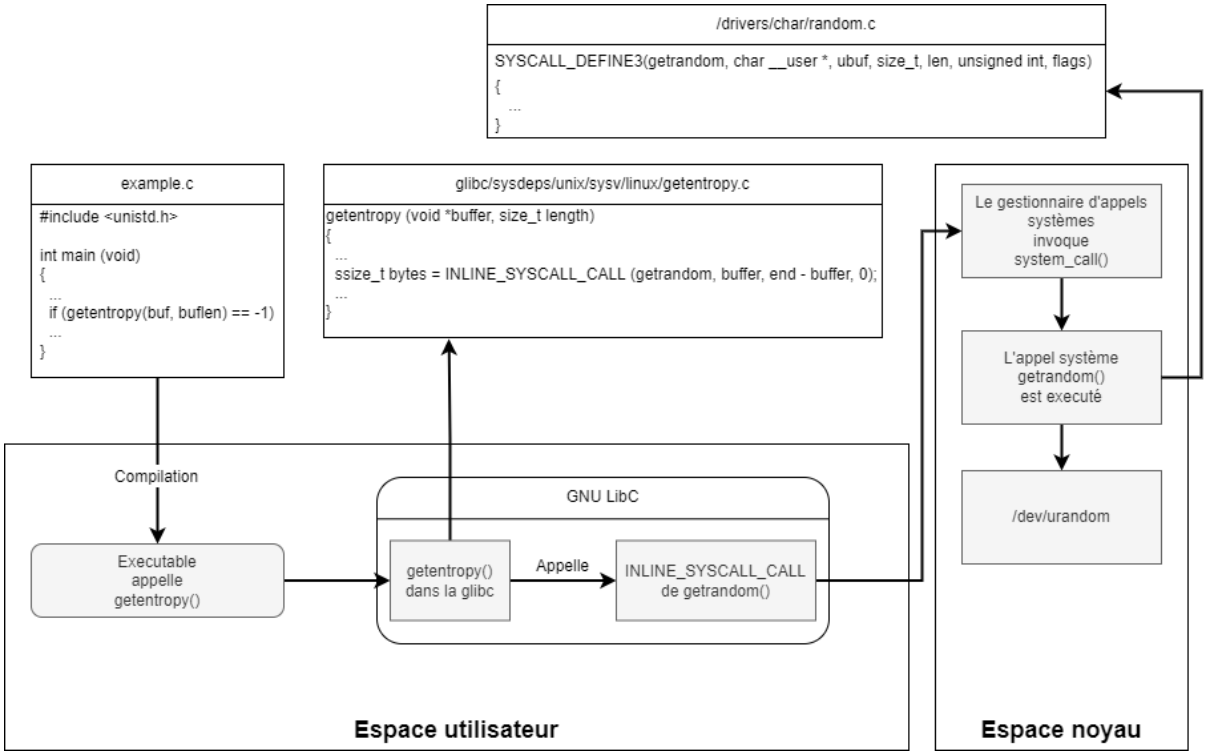


Figure A.2 – Accès de *getentropy()* à /dev/urandom

l'avons vu, la GlibC utilise `syscall()` en 64 bits car l'ABI Linux x86_64 supporte les points d'entrées suivants :

— `syscall()` depuis les codes compilés en 64 bits ;
— l'interruption 128 depuis les codes compilés en 32 et 64 bits ;
— `sysenter()` depuis les codes compilés en 32 bits.

Nous sommes rendus au point de bascule ou on passe de l'espace utilisateur à l'espace noyau. Les architectures x86 possèdent différents points d'entrées vers l'espace noyau pour les appels systèmes. Ces points d'entrée sont listés dans le fichier `/arch/x86/kernel/traps.c` et implémentés dans le fichier `/arch/x86/entry/entry_64.S` des sources du noyau Linux pour les systèmes en 64 bits.

Les architectures x86 implémentent une table nommée IDT (*Interrupt Descriptor Table*) permettant d'associer différents types d'interruptions avec leur réponse. On trouve le détail de cette table dans le fichier `/arch/x86/include/asm/irq_vectors.h` des sources du noyau Linux. Une fois l'IDT notifiée depuis l'espace utilisateur, le noyau exécute l'instruction `SYSCALL` définie dans le fichier `entry_64.S` et invoquée `inline` depuis la GlibC comme vu précédemment. L'entrée `entry_SYSCALL_64` prend en charge la préparation de l'appel système. Cette préparation consiste à effectuer certaines opérations sur les registres. Le point critique est l'utilisation de l'instruction `swapgs` pour faire passer les données de l'appel système de l'espace utilisateur à l'espace noyau pour son exécution et de l'espace noyau à l'espace utilisateur au retour de l'appel. Une fois l'appel système exécuté, le gestionnaire d'appel système positionne le code retour de l'appel système dans le registre `rax`. Ensuite les registres entre `rcx` et `r11` (exclus) sont restaurés. Le registre `rcx`

contient l'adresse de retour vers l'application ayant invoqué l'appel système. Le registre `r11` contient le *flag* représentant l'état du processeur.

Pour résumer cette section, voici le chemin complet de l'exécution d'un appel système :

1. L'application en espace utilisateur peuple les registres du processeur avec le numéro de l'appel système et ses arguments ;
2. Le processeur commute du mode utilisateur au mode noyau et exécute une entrée d'appel système `entry_SYSCALL_64` ;
3. Cette entrée `entry_SYSCALL_64` permute vers le tas du noyau et sauvegarde quelques registres, le tas originel, les segments de code, flags etc. dans le tas ;
4. L'entrée `entry_SYSCALL_64` recupère le numéro de l'appel système du registre rax, recherche le gestionnaire adapté dans `sys_call_table` et l'appelle (si le numéro d'appel système est correct) ;
5. Une fois que l'appel système est terminé, les registres, le tas originel, les *flags* et l'adresse de retour sont restaurés. Le code retour est enregistré dans le registre rax. L'instruction `sysret` est invoquée pour sortir de `entry_SYSCALL_64`.

A.7 Création d'un processus dit lourd

Le programme présenté dans le listing A.2 effectue un simple appel à `fork()` pour créer un fils et afficher son PID ainsi que sa parentée (PPID soit *Parent PID*). En effet, si la fonction `fork()` renvoie 0 alors nous sommes dans le fils, sinon nous sommes dans le père. C'est ce branchement conditionnel qui permet de discriminer ce qui doit être exécuté dans l'un ou l'autre. Dans cette discussion, on parle bien de `fork()`, la fonction implémentée dans la GlibC, pas de l'appel système.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
       \textit{PID}_t \textit{PID} = fork();

       if (\textit{PID} == -1) {
       perror("fork");
       exit(EXIT_FAILURE);
       }
       else if (\textit{PID} == 0) {
       printf("Child with \textit{PID} %d says \"Hello\" and daddy is at
          ↪ \textit{PID} %d\n", get\textit{PID}(), getp\textit{PID}());
       exit(EXIT_SUCCESS);
```

```
        }
        else {
        printf("Father with \textit{PID} %d says \"Hello\" and daddy is at
            ↪ \textit{PID} %d\n", get\textit{PID}(), getp\textit{PID}());
        wait(NULL);
        printf("Child process finished.\n");
        }

        return 0;
}
```

Listing A.2 – simple appel à *fork()*

La sortie de l'exécution du programme est affichée dans le listing A.3. Le processus père affiche son *PID* de 1119 et attend le retour de son fils. Le fils affiche ensuite son *PID* 1120 ainsi que celui de son père 1119. La fonction *wait()* est un wrapper proposé par la GlibC de l'appel système du même nom. Cet appel système place le processus père dans un état attentiste tant que son ou ses fils n'ont pas changé d'état. Le père de notre programme est le processus bash ayant le *PID* 867 depuis lequel nous avons lancé notre programme.

```
$ ./forkonly
Father with \textit{PID} 1119 says "Hello" and daddy is at \textit{PID}
    ↪ 867
Child with \textit{PID} 1120 says "Hello" and daddy is at \textit{PID}
    ↪ 1119
Child process finished.
```

Listing A.3 – Sortie d'un simple appel à fork()

Si nous traçons les appels systèmes effectués par notre programme, nous pouvons voir que l'appel à la fonction *fork()* déclenche un appel système *clone()* comme suit :

clone(child_stack=NULL, flags = CLONE_CHILD_CLEARTID
| CLONE_CHILD_SETTID
| SIGCHLD, child_tidptr = 0x7f1868362810) = 1120

C'est cet appel système qui va créer le processus fils en copiant le père. L'appel système *sys_fork()* est bien présent dans le fichier syscall_64.tbl avec le numéro 57. Cependant, dans les versions actuelles des systèmes basés sur les noyaux Linux, il n'est plus utilisé au profit de *sys_clone()*. Dans les paramètres de *clone()* nous voyons que l'argument *child_stack* est à NULL. Cela signifie que le père et le fils ne partagent aucun espace mémoire.

Le couple *CLONE_CHILD_SETTID* et *CLONE_CHILD_CLEARTID* servent respectivement à instancier (*SETTID*) et supprimer (*CLEARTID*) le *TID* (Thread ID) du fils à l'emplacement mémoire pointé par l'argument *child_tidptr*. *SIGCHLD* est le signal (cf section A.5) que renverra le processus fils cloné à sa terminaison.

Cette configuration de *clone()* est plus ou moins l'équivalent de l'appel système *fork()*. Nous venons de parler du *TID*. En effet, un processus peut engendrer un autre processus comme nous venons de le voir mais il peut aussi engendrer des threads, nous y reviendrons dans la section A.8. Finissons sur la création des processus lourds avec un détour dans la GlibC pour voir ce qui se cache derrière la fonction *fork()*. D'après le listing A.4, la fonction *fork()* est un alias pointant vers *__libc_fork()* qui invoque la fonction *_Fork()* présentée dans le listing A.5. Cette fonction invoque *arch_fork()* qui prend l'adresse du *TID* en paramètre. Ensuite, la fonction positionne un certain nombre de flags par défaut qui miment le fonctionnement de l'appel système *fork()* comme indiqué dans le listing A.6.

```
#Fichier : /glibc/posix/fork.c
\textit{PID}_t
__libc_fork (void)
{
  ...
  \textit{PID}_t \textit{PID} = _Fork ();
  ...
}
...
weak_alias (__libc_fork, fork)
```

Listing A.4 – Point d'entrée de *fork()* dans la GlibC

```
#Fichier : /glibc/sysdeps/nptl/_Fork.c
\textit{PID}_t
_Fork (void)
{
  \textit{PID}_t \textit{PID} = arch_fork (&THREAD_SELF->tid);
  ...
}
Listing I.1.3e : fonction _Fork() dans la GlibC
```

Listing A.5 – fonction *_Fork()* dans la GlibC

```
#Fichier : /glibc/sysdeps/unix/sysv/linux/arch-fork.h
static inline \textit{PID}_t
arch_fork (void *ctid)
{
  const int flags = CLONE_CHILD_SETTID | CLONE_CHILD_CLEARTID | SIGCHLD;
  long int ret;
#ifdef __ASSUME_CLONE_BACKWARDS
# ifdef INLINE_CLONE_SYSCALL
  ret = INLINE_CLONE_SYSCALL (flags, 0, NULL, 0, ctid);
# else
```

```
  ret = INLINE_SYSCALL_CALL (clone, flags, 0, NULL, 0, ctid);
# endif
#elif defined(__ASSUME_CLONE_BACKWARDS2)
  ret = INLINE_SYSCALL_CALL (clone, 0, flags, NULL, ctid, 0);
#elif defined(__ASSUME_CLONE_BACKWARDS3)
  ret = INLINE_SYSCALL_CALL (clone, flags, 0, 0, NULL, ctid, 0);
#elif defined(__ASSUME_CLONE2)
  ret = INLINE_SYSCALL_CALL (clone2, flags, 0, 0, NULL, ctid, 0);
#elif defined(__ASSUME_CLONE_DEFAULT)
  ret = INLINE_SYSCALL_CALL (clone, flags, 0, NULL, ctid, 0);
#else
# error "Undefined clone variant"
#endif
  return ret;
}
```

Listing A.6 – fonction *arch_fork()* dans la GlibC

A.8 Création des threads

Dans cette section, nous présenter via un code exemple la différence entre les threads et les processus dits « lourds ». Examinons le code présenté dans le listing A.7. Dans ce code, notre processus va créer deux threads. Chacun des threads va exécuter la procédure *say_hello()* qui va afficher son *PID*, son TID (Thread ID) ainsi que le P*PID*. Le père, que nous nommerons processus appelant dans le contexte des threads, va se contenter d'afficher son *PID* et attendre que ses threads fils soient terminés avec un *pthread_join()* sur chacun d'eux.

```
#Fichier : 2threads.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

void *say_hello() {
       printf("Child thread with ID %d and \textit{PID} %d says \"Hello\"
          ↪  daddy is at %d\n", gettid(), get\textit{PID}(),getp\textit{
          ↪ PID}());
       pthread_exit(NULL);
}

int main() {
       printf("Father process with \textit{PID} %d says \"Hello\"\n", get
```

```
        ↪ \textit{PID}());
    pthread_t threads[2];
    int thread_nums[2] = {1, 2};
    int i;
    for (i = 0; i < 2; i++) {
    int thread_num = thread_nums[i];
    int ret = pthread_create(&threads[i], NULL, say_hello, NULL);
    if (ret) {
        fprintf(stderr, "Error creating thread\n");
        exit(EXIT_FAILURE);
    }
    }
    for (i = 0; i < 2; i++) {
    pthread_join(threads[i], NULL);
    }
    pthread_exit(NULL);
    return 0;
}
```

Listing A.7 – Création de deux threads

Une fois la commande exécutée, nous voyons que l'appelant et ses deux threads partagent le même *PID* 742 comme affiché dans le listing A.8. Le TID de l'appelant est égal au *PID*. En revanche, les deux threads ont un TID différent de l'appelant. Le P*PID* de tous les protagonistes est celui du processus bash dont ils sont issus.

```
$ ./2threads
Father process with thread_id 742 and \textit{PID} 742 says "Hello" daddy
   ↪  is at \textit{PID} 604
Child thread with thread_id 744 and \textit{PID} 742 says "Hello" daddy
   ↪ is at \textit{PID} 604
Child thread with thread_id 743 and \textit{PID} 742 says "Hello" daddy
   ↪ is at \textit{PID} 604
```

Listing A.8 – Sortie de la création de deux threads

Si on observe les appels systèmes, nous voyons que les threads sont également créés par l'appel système *clone()*. En revanche, les arguments sont différents. Voici ceux du premier thread :

clone(child_stack=0x7f1cfc833fb0, flags=CLONE_VM
| CLONE_FS | CLONE_FILES | CLONE_SIGHAND
| CLONE_THREAD | CLONE_SYSVSEM | CLONE_SETTLS
| CLONE_PARENT_SETTID | CLONE_CHILD_CLEARTID,
parent_tid=[743], tls=0x7f1cfc834700,
child_tidptr=0x7f1cfc8349d0) = 743

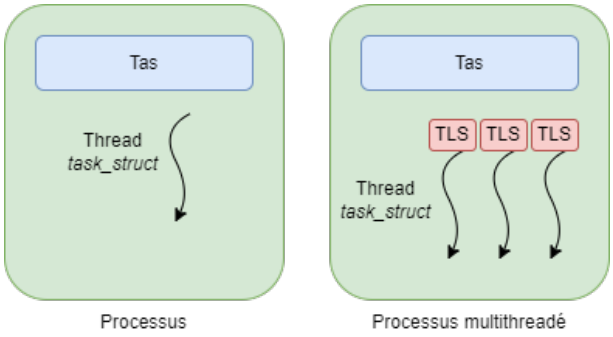


Figure A.3 – Représentation d'un processus

Et du second :

clone(child_stack=0x7f1cfc032fb0, flags=CLONE_VM
| CLONE_FS | CLONE_FILES | CLONE_SIGHAND
| CLONE_THREAD | CLONE_SYSVSEM | CLONE_SETTLS
| CLONE_PARENT_SETTID | CLONE_CHILD_CLEARTID,
parent_tid=[744], tls=0x7f1cfc033700,
child_tidptr=0x7f1cfc0339d0) = 744

Une différence fondamentale avec la création des processus lourds est que cette fois ci l'argument *child_stack* possède une valeur. En effet, comme le flag *CLONE_VM* est activé, le thread et le processus appelant partagent le même espace mémoire. En conséquence, il faut que le thread sache où placer ses données. C'est le pointeur *child_stack*. Nous avons ensuite tout un tas de flags préfixés par *CLONE_**. Ces flags dupliquent les éléments d'environnement de l'appelant dans le thread tels que : les descripteurs, la racine au niveau du système de fichier, le gestionnaire de signaux etc. Le flag *CLONE_SETTLS* indique que le thread disposera d'un TLS (Thread Local Storage) qui est un tas réservé au thread. C'est-à-dire qu'il n'est partagé ni avec le père, ni avec les autres threads.

En résumé, ce que l'on nomme processus peut être un peu flou. Dans ce mémoire de thèse, lorsque nous parlerons de processus, nous sous-entendrons processus avec un seul thread c'est-à-dire, un seul couple `thread_info` / `task_struct()`. Au cas où un processus serait composé de différents threads avec éventuellement leur TLS dédié, nous parlerons de processus multithreadé. Dans ce dernier cas, le programme démarre et s'arrête en séquentiel, c'est-à-dire monothread, et possède des sections parallèles ou des threads sont créés pour traiter les fonctions sur des cœurs différents. Dans tous les cas, chaque thread est représenté par le couple `thread_info` / `task_struct()`. La figure A.3 résume ces notions.

A.9 Recouvrement de la section exécutable

La Figure 2.3 présente une hiérarchie de processus ou les fils n'exécutent pas le même code que le père. Le listing A.9 présente un code simple en C qui réalise cette action. On voit que le fils fait appel à la fonction *execvp()* de la GlibC dans le fils pour exécuter la commande `/usr/bin/hostname`.

```
#Fichier : forkexec.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
       \textit{PID}_t \textit{PID} = fork();

       if (\textit{PID} == -1) {
         perror("fork");
         exit(EXIT_FAILURE);
       }
       else if (\textit{PID} == 0) {
         char *command[] = {"/usr/bin/hostname", NULL};
         execvp(command[0], command);
         perror("execvp");
         exit(EXIT_FAILURE);
       }
       else {
         wait(NULL);
         printf("Child process finished.\n");
       }
       return 0;
}
```

Listing A.9 – Exécution d'un autre code par le fils

La sortie du fils est sans grande surprise, elle affiche bien le nom court (*hostname*) de la machine. Ensuite le fils se termine. La partie intéressante réside dans l'analyse des appels systèmes produits.

```
$ ./forkexec
debian-host
Child process finished.
```

Listing A.10 – Sortie de l'exécution d'un autre code par le fils

Du côté du père on a un appel dorénavant classique à la l'appel système `clone()` suivi d'un `wait()` qui place le père en attente de la sortie du fils :

```
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID
| CLONE_CHILD_SETTID | SIGCHLD, child_tidptr=0x7fbd463d6810) = 690

wait4(-1, NULL, 0, NULL) = 690
```

Au niveau du fils, on constate que le premier appel système est un `execve()` :

```
execve("/usr/bin/hostname", ["/usr/bin/hostname"],
0x7fff716cf3a8 /* 19 vars */) = 0
```

Pour comprendre cet appel système, il est nécessaire de faire un petit retour en arrière sur la structure d'un processus dans la section A.1. Lors de cet appel à `execve()`, les zones mémoire TEXT, BSS et DATA sont écrasées avec le code machine de l'exécutable `/usr/bin/hostname`. L'éditeur de liens est invoqué pour lier l'exécutable invoqué avec les bibliothèques partagées. Un certain nombre d'attributs dupliqués du processus père sont supprimés tels que les mémoires partagés, piles de signaux, les sémaphores et verrous mémoires etc. Enfin, le signal de terminaison du processus est réinitialisé à `SIGCHLD`, peu importe ce qui a été spécifié par l'appel `clone()` précédant notre `execve()`. Le dernier paramètre de `execve()` est l'adresse d'une liste de chaînes de caractères terminée par NULL de la forme « VAR1=VALEUR », « VAR2=VALEUR », ..., NULL.

A.10 Identité d'un processus

Un processus tourne avec une identité. Dans la structure `task_struct()` on trouve deux attributs `real_cred` et `cred` qui sont de type `cred`. Ce type représente le contexte de sécurité du processus. Ce contexte est divisé en deux. D'une part, on a le contexte objectif qui concerne l'identité à utiliser lors d'une interaction d'un autre processus sur le processus concerné. C'est l'attribut `real_cred`. D'autre part, nous avons le contexte subjectif qui concerne l'identité que le processus utilise lorsqu'il est à l'origine de l'interaction. C'est l'attribut cred. Dans cette structure `cred` définissant le contexte de sécurité on retrouve trois type d'UID / GID :

— Réel (RUID / RGID) : c'est l'UID de l'utilisateur qui a lancé un processus donné. Le RUID est déterminé par le système au moment où l'utilisateur se connecte et reste inchangé pendant toute la durée de la session de l'utilisateur ;
— Effectif (EUID / EGID) : c'est l'UID utilisé par le noyau pour déterminer les permissions d'accès lorsqu'un processus tente d'accéder à des ressources protégées par le système. C'est l'UID évalué lors du DAC A.11 ;
— Sauvegardé (SUID / SGID) : Sauvegarde de l'EUID / EGID original.

Le noyau utilise donc le couple d'identifiants numérique UID / GID pour évaluer les droits d'un processus. Il n'a cure de l'identifiant chaîne de caractère associée à l'UID / GID. Sur un système Linux, on trouve deux niveaux de comptes : administrateur et banalisé. L'administrateur est le compte *root* avec l'UID 0, tous les autres comptes sont des comptes banalisés avec un UID > 0.

A.11 Le DAC (*Discretionary Access Control*)

Le DAC est un modèle de sécurité discrétionnaire. Cela signifie que les droits sont laissés à la discrétion du propriétaire de la ressource. L'avantage d'un tel contrôle d'accès est sa souplesse. Chaque utilisateur peut fixer lui-même les droits sur les ressources dont il

est propriétaire. L'inconvénient associé est que la politique de sécurité évolue dans un sens qui n'est pas prédictible et il est donc difficile de garantir quoi que ce soit à l'échelle d'un système informatique réel. Le modèle DAC est utilisé sous Linux pour protéger l'accès aux fichiers. Le contrôle d'accès au fichier se nomme UGO pour *User Group Other*. Chaque membre du triplet définit une population : U et G droits qui s'appliquent respectivement à l'utilisateur et au groupe propriétaire puis O qui s'applique au restant du monde. Pour chaque membre du triplet on dispose de trois droits : lecture « r », écriture « w » et exécution « x ». La lecture appliquée à un dossier veut dire qu'on peut lister le contenu, appliquée à un fichier on peut lire le contenu. L'écriture appliquée à un dossier veut dire qu'on peut y ajouter ou supprimer des éléments, appliquée à un fichier on peut modifier le contenu. L'exécution appliquée à un dossier veut dire qu'on peut le traverser, appliquée à un fichier c'est qu'on peut l'exécuter.

En complément de ces trois droits, il en existe trois autres applicables aux fichiers et aux répertoires que l'on nomme bits spéciaux :

— Le bit *setuid* (SUID) : ce bit est représenté par un « s » dans la liste des permissions pour le propriétaire. Lorsqu'il est activé sur un fichier exécutable, cela signifie que le processus qui exécute le fichier aura temporairement les mêmes droits que le propriétaire du fichier. Cela est souvent utilisé pour permettre à des utilisateurs normaux d'exécuter des programmes qui nécessitent des privilèges administrateur. Par exemple, la commande « passwd » qui permet de modifier les mots de passe des utilisateurs ;

— Le bit *setgid* (SGID) : ce bit est représenté par un « s » dans la liste des permissions pour le groupe. Lorsqu'il est activé sur un répertoire, cela signifie que les fichiers créés dans ce répertoire hériteront du groupe propriétaire du répertoire plutôt que du groupe de l'utilisateur qui les crée ;

— Le bit *sticky* (STICKY) : ce bit est représenté par un « t » dans la liste des permissions pour les autres utilisateurs. Lorsqu'il est activé sur un répertoire, cela signifie que seuls les propriétaires des fichiers peuvent les supprimer ou les renommer, même si d'autres utilisateurs ont les droits d'écriture sur le répertoire.

Ces droits constituent l'ossature du contrôle d'accès sous Linux. Ils sont stockés dans les métadonnées des fichiers (même si certains systèmes de fichiers comme vfat n'ont pas de notions de droits) et sont évalués sur la base de l'EUID du processus qui va tenter d'accéder à l'objet. On se sert donc de ce contrôle d'accès pour appliquer le moindre privilège à un processus. En effet, il faut bien avoir à l'esprit que l'UID peut caractériser un compte pour un humain ou pour un service.

A.12 Capacités (*Capabilities*)

Les capacités (*capabilities* en anglais) sont des supers pouvoirs affectés à un processus donné. Dans la structure de données `cred` dont nous avons parlé dans l'annexe A.10, on trouve cinq attributs de type `kernel_cap_t`. Ce type est un masque type vecteur de bits contenant l'information sur quelle capacité est activée (1) ou pas (0). Ces capacités complètent le DAC discuté ci-dessus en proposant de mettre des pouvoirs intermédiaires entre l'utilisateur banalisé et l'administrateur tels que la possibilité de créer des sockets réseaux

bruts pour, par exemple, forger des requêtes ICMP (CAP_NET_RAW), l'outrepassement du DAC (cf annexe A.11) lors d'un accès aux fichiers (CAP_DAC_OVERRIDE) ou encore la capacité de lier un service réseau s'exécutant avec une identité non privilégiée (UID > 0) avec un port privilégié inférieur à 1024 (CAP_NET_BIND_SERVICE). Chacun des cinq vecteurs de capacité associé au processus correspond à un scénario d'utilisation :

- `cap_inheritable` : ensemble des capacités dont le processus hérite, notamment après un *execve()* (cf annexe A.9) ;
- `cap_permitted` : ensemble des capacités que le processus pourra utiliser (mais pas nécessairement activées). Ce vecteur limite le précédent ;
- `cap_effective` : ensemble des capacités vérifiées par le noyau lors du contrôle d'accès du processus ;
- `cap_bounding` : ensemble des capacités que le processus pourra acquérir au regard du vecteur `cap_inheritable` ;
- `cap_ambient` : ensemble des capacités préservées après un appel à `execve()` de programmes non privilégiés. Les capacités dans ce vecteur doivent également être présentes dans les vecteurs `cap_permitted` et `cap_inheritable`.

Les capacités peuvent également être associées à des fichiers. C'est ce qui permet à des utilisateurs non privilégiés d'acquérir des capacités à l'utilisation d'un exécutable. Cela limite l'utilisation du bit SUID associé à un exécutable possédé par « *root* ». Évidemment, cette possibilité est conditionnée par le fait que le système de fichiers doit supporter les attributs étendus sur les fichiers.

La finesse du paramétrage de la transmission des capacités d'un processus à un autre démontre que ce mécanisme de sécurité est basé sur la responsabilité. C'est au programme privilégié de sélectionner les capacités dont il a besoin et d'abandonner les autres. Il doit aussi soigneusement paramétrer son héritage. On notera que ce qui fait de « *root* » l'administrateur, ce n'est pas qu'il possède l'UID 0 mais c'est plutôt le fait que les processus qu'il invoque disposent de toutes les capacités. Si il met ses vecteurs à 0 sur un processus donné, en dépit du fait qu'il tournera avec l'UID 0 il n'aura plus ses privilèges. C'est ce mécanisme qui est utilisé sous le capot lorsque l'on octroie tel ou tel privilège à un conteneur.

A.13 Le MAC (*Mandatory Access Control)*

En complément du DAC, il existe un autre modèle de sécurité appelé MAC (*Mandatory Access Control*). Le MAC s'appuie souvent sur les LSM (*Linux Security Module*). Les LSM sont des *hooks* disposés sur certaines fonctions du noyau Linux pour que les différentes implémentations de MAC (SELinux, Tomoyo, AppArmor, SMACK etc.) puissent intercepter l'appel à la fonction et décider s'ils laissent passer ou pas. Ce contrôle d'accès est dit mandataire car même l'utilisateur *root* devra s'y soumettre.

La majorité des implémentations du MAC se basent sur des politiques de sécurité exprimées en termes de relations sujets / objets. Les composants du système d'exploitation sont labellisés et les permissions sont exprimées en fonction des labels. Ce système de labellisation est relativement fin car il permet d'interdire ou d'autoriser un appel système

de façon plus fine qu'avec les capacités. En effet, avec le jeu des labels ce sont les arguments de l'appel système qui sont évalués. Nous allons très peu mentionner le MAC dans la suite du document car actuellement ses implémentations sont difficiles à appliquer aux conteneurs. Cependant, des politiques peuvent être implémentées, plutôt dans l'objectif de protéger le moteur de conteneur lui-même.

A.14 Le confinement historique : *chroot()*

Le premier mécanisme de confinement sous Linux fut l'appel système `sys_chroot()`. L'objectif de cet appel système est de changer l'attribut racine du système de fichier du processus qui l'invoque. La conséquence est qu'à l'issu de cet appel système, le processus ne verra plus ce qui est au-dessus de sa nouvelle racine. En effet, sa nouvelle racine sera le chemin passé à l'appel système. Si on regarde le code de l'appel système `chroot()` présenté dans le listing A.11, il vérifie que le processus qui l'invoque a bien la capacité CAP_SYS_CHROOT puis, dans l'affirmative, il invoque la fonction `set_fs_root()` qui va effectivement modifier la valeur dans la `task_struct()` relative au processus appelant (paramètre `current` dans le premier argument de la fonction).

```
#Fichier : /fs/open.c
SYSCALL_DEFINE1(chroot, const char __user *, filename)
{
      ...
        if (!ns_capable(current_user_ns(), CAP_SYS_CHROOT))
       ...
        set_fs_root(current->fs, &path);
```

Listing A.11 – l'appel système *chroot()*

Le code source de la fonction `set_fs_root()` est présenté dans le listing A.12. Cette fonction place un verrou, sauvegarde l'ancienne valeur de la racine, place la nouvelle et libère le verrou. Ce qui est intéressant avec cette fonction c'est que l'on voit que cette opération est simple. La seule isolation qui est faite est relative à la racine du système de fichier. Pour qu'un processus soit réellement isolé, il faut beaucoup plus de contraintes. Un processus chrooté peut en effet lister les autres processus, utiliser les IPC (cf annexe A.4) etc. De plus, les UID dans le *chroot* sont les mêmes que sur le système de base. Le *root* dans un *chroot* est le même que hors du *chroot*. Enfin, le *chroot* n'a pas de mémoire au niveau de la profondeur des isolations. Une méthode d'évasion est donc d'invoquer `chroot()` dans un *chroot* et d'en sortir. On revient alors sur le système de base.

```
#Fichier : /fs/fs_struct.c
void set_fs_root(struct fs_struct *fs, const struct path *path)
{
       struct path old_root;

       path_get(path);
```

```
        spin_lock(&fs->lock);
        write_seqcount_begin(&fs->seq);
        old_root = fs->root;
        fs->root = *path;
        write_seqcount_end(&fs->seq);
        spin_unlock(&fs->lock);
        if (old_root.dentry)
          path_put(&old_root);
}
```

Listing A.12 – modification de la structure *fs_struct*

La raison en est simple, `chroot()` a été développé pour pouvoir compiler des codes dans différents environnements. Il n'y avait pas d'enjeux sécurité derrière. Cependant, `chroot()` est utilisé par beaucoup de services système Linux pour proposer un premier rempart de sécurité en limitant la vision que le processus a du système de fichier. En effet, pour mitiger les limites de sécurité inhérentes à `chroot()` beaucoup de services combinent son utilisation avec l'appel système `setuid()`.

La fonction de `setuid()` présentée dans le listing A.13 est de remplacer les attributs `cred` et `real_cred` de la structure `task_struct()` représentant le processus appelant avec une nouvelle structure `cred`. Une fois de plus, pour que le processus puisse recourir à cet appel système, il faut qu'il possède une capacité nommée CAP_SETUID.

On notera également que les attributs `cred` et `real_cred` ne sont pas changés en direct mais via la création d'une nouvelle structure `cred` initialisée avec la fonction `prepare_creds()` et instanciée avec l'UID passé en paramètre de la fonction `__sys_setuid()`. A la fin de `__sys_setuid()`, la fonction `commit_creds()` est invoquée pour faire la substitution avec la nouvelle structure `cred` précédemment initialisée.

```
#Fichier : /kernel/sys.c
long __sys_setuid(uid_t uid)
{
    ...
    struct cred *new;
    ...
    kuid = make_kuid(ns, uid);
    ...
    new = prepare_creds();
    ...
    old = current_cred();
    ...
    if (ns_capable_setid(old->user_ns, CAP_SETUID)) {
         new->suid = new->uid = kuid;
       ...
```

```
    new->fsuid = new->euid = kuid;
    ...
    return commit_creds(new);

  ...
}

SYSCALL_DEFINE1(setuid, uid_t, uid)
{
    return __sys_setuid(uid);
}
```

Listing A.13 – L'appel système *setuid()*

Ce changement d'identité *post-chroot* vers une identitée banalisée réduit les capacités d'un attaquant éventuel à sortir du *chroot via*, par exemple, la méthode du double *chroot*. Cependant, toutes les autres restrictions s'appliquent. On ne peut donc pas parler d'isolation comme on l'entend au niveau des conteneurs.

ANNEXE B

ARTEFACT EUROPAR 2022

EURO-PAR
CONFERENCE 2022

Euro-Par 2022 Best Artefact Award

A methodology to scale containerized HPC infrastructures in the Cloud

Nicolas Greneche, Christophe Cérin, Tarek Menouer and Olivier Richard

This is awarded to:

Nicolas Greneche

University of Sorbonne Paris North

28th International European Conference on Parallel and Distributed Computing (Euro-Par'22)

22-26th August 2022, Glasgow, UK

A methodology to scale containerized HPC infrastructures in the Cloud

Nicolas Grenèche, Tarek Menouer, Christophe Cérin and Olivier Richard

nicolas.greneche@univ-paris13.fr, tmenouer@umanis.com, christophe.cerin@univ-paris13.fr, olivier.richard@imag.fr

May 13, 2022

Getting Started

This artifact supplies the environment for section 4, “Experimentations,” of our paper. As stated in the document, we evaluated three central HPC schedulers: OAR, SLURM, and OpenPBS.

The evaluation process is the same for each HPC scheduler and is as follows:

- Step 1: instantiation of the HPC containerized Infrastructure;
- Step 2: submission of an infinite loop job to keep a containerized compute node busy;
- Step 3: submission of an MPI job that requires more resources than is available;
- Step 4: dynamic addition of an HPC worker container to satisfy job requirements;
- Step 5: MPI job runs and exits successfully;
- Step 6: downsize the HPC containerized Infrastructure without losing the infinite loop job;
- Step 7: deletion of the HPC containerized Infrastructure.

We supply a comprehensive experimentation testbed as a virtual image (VM) in the OVA format. Here is the download link:

https://drive.google.com/file/d/174Q1GFGZ4A0daif3OuV_d7k6IbTedMu9/view?usp=sharing

All commands of the document are referred this way :

```
prompt$ command
Output line 1
Output line 2
…
Output line n
```

The prompt is grey, the command itself is black and potential outputs are below in standard font with a soft grey background.

You can check the integrity of this file through the following command line instruction:

```
nico@DESKTOP-KSGVO8B:~$ md5sum Europar22-paper25.ova
964f26851bcf5a08dbe03a59ee059f84  Europar22-paper25.ova
```

This file is an OVA image that contains our experimentation platform. We use the VirtualBox hypervisor throughout the document, and you can import the OVA file straight to VirtualBox. The machine hosting VirtualBox must be able to run virtual machines that use six vcpus in total. After deploying the OVA image, you get four virtual machines:

- Lab-admin: host the Kubernetes API-Server ;
- Lab-frontal: gateway to connect to the testbed ;
- Lab-node[1,2]: cloud nodes that host our containerized HPC infrastructure.

The virtual machine “Lab-frontal” has two network interfaces, one configured as NAT and one as an internal network where all the other virtual machines are plugged. Other virtual machines use “Lab-frontal” as a gateway to access the internet. We also run an internal local domain named “lab.local”, such that “Lab-frontal” embed a DNS server that redirects all DNS resolution out of “lab.local” to Google public DNS 8.8.8.8. The hosting machine must also reach Github and Docker Hub servers.

We defined a port redirection on Lab-frontal to reach the node from the localhost of your computer on port 2424. Default login/password is root/azerty for all virtual machines. To connect in the “Lab-frontal node,” type:

```
nico@DESKTOP-KSGVO8B:~$ ssh -p 2424 root@localhost
```

You will instantiate the Kubernetes cluster from this node.

First, you must activate the Python virtual environment containing Ansible. Ansible is used to deploy a reproducible Kubernetes cluster on supplied virtual machines.

```
root@frontal# source ansible/bin/activate
(ansible) root@frontal#
```

Your command-line prompt change to “(ansible) root@frontal:~#”, meaning that your environment variables have been updated to include the Python virtual environment.

We now clone the receipt that instantiates the Kubernetes cluster:

```
(ansible) root@frontal# git clone https://github.com/Nyk0/k8s-ansible.git
Clonage dans 'k8s-ansible'...
```

The repository k8s-ansible contains all the receipts that download and configure all packages required for our testbed gathered in a playbook. We switch to the k8s-ansible

directory and run the playbook:

```
(ansible) root@frontal# cd k8s-ansible/
```

```
(ansible) root@frontal# ansible-playbook -i inventory/hosts.yml site.yml
```

Setting up the cluster takes a little time, and the testbed is ready to replay the experimentation when the playbook deployment ends. Every interaction with our Kubernetes cluster must be performed from the “Lab-admin” virtual machine. So, we log in:

```
(ansible) root@frontal#  ssh admin
```

And we check that the cluster is up:

```
root@admin# kubectl get nodes
NAME  STATUS   ROLES                  AGE    VERSION
admin   Ready control-plane,master   3m6s   v1.22.3
node1   Ready<none>                   79s    v1.22.3
node2   Ready<none>                   80s    v1.22.3
```

Hosts node1 and node2 correspond to “Lab-node1” and “Lab-node2” virtual machines. They should be in the “Ready” status. Now we can clone the Git repository that contains all the manifests required to evaluate the three HPC schedulers:

```
root@admin# git clone https://github.com/Nyk0/hpcsched-containers.git
```

Step-by-Step Instructions

We can switch to the “hpcsched-containers” directory:

```
root@admin# cd hpcsched-containers/
```

We enable sidecar containers to access the deployment configuration:

```
root@admin# kubectl apply -f rbac.yaml
clusterrole.rbac.authorization.k8s.io/pods-list-hpc-nico created
clusterrolebinding.rbac.authorization.k8s.io/pods-list-hpc-nico created
```

We create the namespace dedicated to our containerized HPC cluster:

```
root@admin:~/hpcsched-containers# kubectl create namespace hpc-nico
namespace/hpc-nico created
```

We set it as a default namespace for our user:

```
root@admin# kubectl config set-context --current --namespace=hpc-nico
Context "kubernetes-admin@kubernetes" modified.
```

We configure a physical volume that maps an NFS server configured during the Ansible deployment. This NFS server exports a volume that contains the home directory of our testing user inside the containerized HPC cluster:

```
root@admin# kubectl apply -f misc/nfs/pv-nfs.yaml
persistentvolume/pv-nfs-home created
```

Then we create the associated claim that our pods will use:

```
root@admins# kubectl apply -f misc/nfs/pvc-nfs.yaml
persistentvolumeclaim/pvc-nfs-home created
```

We can now deploy the three HPC schedulers on our Kubernetes cluster. Remind that we run an MPI job for each HPC scheduler that requires 2 nodes with 2 CPUs per node. This job is a pi calculation according to a Monte-Carlo method.

1) OpenPBS case study

Each scheduler has a dedicated directory that contains the manifest for their deployment. We move into the OpenPBS directory first:

```
root@admin# cd openpbs/
```

We deploy a containerized OpenPBS cluster with a master node, and two compute nodes. Each compute node has 2 CPUs according to the manifest:

```
root@admin# kubectl apply -f openpbs.yaml
service/nodes created
statefulset.apps/hpc-node created
statefulset.apps/control-node created
```

We check that the three pods corresponding to our deployment are running. It may take time to get images from Docker Hub (be patient):

```
root@admin# kubectl get pods
```

```
NAME           READY   STATUS    RESTARTS   AGE
control-node-0   2/2   Running   0          2m47s
hpc-node-0     1/1     Running   0          2m47s
hpc-node-1     1/1     Running   0          103s
```

The control-node-0 pod contains the scheduler (pbs_sched). The pods hpc-node-[0,1] host the worker process that receives jobs from the scheduler (pbs_mom). We get in control-node-0 to interact with the OpenPBS containerized scheduler:

```
root@admin# kubectl exec -ti control-node-0 -c openpbs-server -- /bin/bash
```

Note that we have a regular "nico" user in each containerized HPC scheduler that will run jobs as a standard unprivileged user. We switch to this user:

```
root@control-node-0# su - nico
```

We can use the “pbsnodes” command to check available resources (we cut non relevant information in the output):

```
nico@control-node-0$ pbsnodes -a
```

```
hpc-node-0
        state = free
        resources_available.ncpus = 2
        queue = COMPUTE

hpc-node-1
        state = free
        resources_available.ncpus = 2
        queue = COMPUTE
```

The home of user “nico” has three subdirectories corresponding to each HPC scheduler. Each subdirectory contains two submission scripts: one for the pi MPI program and the other for the infinite loop:

```
nico@control-node-0$ cd openpbs/
```

We submit a “infinite” job that loops forever on one node:

```
nico@control-node-0$ qsub infinite.openpbs
0.control-node-0
```

We check that the job is running:

```
nico@control-node-0$ qstat
Job id              Name             User              Time Use S Queue
----------------  ---------------- ----------------  -------- - -----
0.control-node-0  infinite         nico              00:00:00 R COMPUTE
```

The state column (S) is set to ‘R’ (running).

We have only one node left, and we submit our Monte Carlo pi MPI job that requires two nodes. As a consequence, we get a starvation situation and the job is waiting for resources:

```
nico@control-node-0$ qsub pi.openpbs
1.control-node-0
```

We display the queue:

```
nico@control-node-0$ qstat
Job id              Name             User              Time Use S Queue
----------------  ---------------- ----------------  -------- - -----
0.control-node-0  infinite         nico              00:00:00 R COMPUTE
1.control-node-0  pi               nico                     0 Q COMPUTE
```

We observed that the second “pi” job is queued, waiting for resources.

We exit the “nico” account and then the container itself to go back to the “admin” node:

```
nico@control-node-0$ exit
logout
```

```
root@control-node-0# exit
exit
```

We expand the cluster from 2 to 3 compute nodes to satisfy the needs of the “pi” job:

```
root@admin# kubectl patch statefulsets hpc-node -n hpc-nico -p '{"spec":{"replicas":3}}'
statefulset.apps/hpc-node patched
```

We check that the additional pod is running:

```
root@admin# kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
control-node-0   2/2   Running   0          5m44s
hpc-node-0     1/1     Running   0          5m44s
hpc-node-1     1/1     Running   0          4m38s
hpc-node-2     1/1     Running   0          22s
```

Then, we go back to the control-node-0:

```
root@admin# kubectl exec -ti control-node-0 -c openpbs-server -- /bin/bash
```

Again, we use the “nico” account and move to the openpbs directory:

```
root@control-node-0# su - nico
```

```
nico@control-node-0$ cd openpbs/
```

We check available resources again:

```
nico@control-node-0$ pbsnodes -a
hpc-node-0
     state = job-busy

hpc-node-1
     state = free

hpc-node-2
     state = free
```

The first node (hpc-node-0) is a “job-busy” executing the “infinite” job. And the two others are “free.” We can check the queue:

```
nico@control-node-0$ qstat
```

```
Job id            Name             User              Time Use S Queue
----------------  ---------------- ----------------  -------- - -----
0.control-node-0  infinite         nico                     0 R COMPUTE
1.control-node-0  pi               nico                     0 Q COMPUTE
```

The “pi” job remains in the queue. That’s normal; OpenPBS takes about 10 minutes to recompute the queue. Thus, a few minutes later, we get:

```
nico@control-node-0$ qstat
```

```
Job id            Name             User              Time Use S Queue
----------------  ---------------- ----------------  -------- - -----
0.control-node-0  infinite         nico                     0 R COMPUTE
```

We can check the output of the “pi” job:

```
nico@control-node-0$ cat pi.o1
```

```
Elapsed time = 0.000004 seconds
Pi is approximately 2.8750000000000000, Error is 0.2665926535897931
```

We exit the “nico” account and the container itself to go back on the admin virtual machine:

```
nico@control-node-0$ exit
```

```
logout
```

```
root@control-node-0# exit
```

```
exit
```

We patch again our deployment to shrink our containerized HPC cluster from 3 to 2 compute nodes:

```
root@admin# kubectl patch statefulsets hpc-node -n hpc-nico -p '{"spec":{"replicas":2}}'
```

```
statefulset.apps/hpc-node patched
```

We check the removal of the hpc-node-2 pod:

```
root@admin# kubectl get pods
```

```
NAME           READY   STATUS    RESTARTS   AGE
control-node-0   2/2   Running   0          19m
hpc-node-0       1/1   Running   0          19m
hpc-node-1       1/1   Running   0          17m
```

And we reconnect on control-node-0 to check resources seen by the containerized HPC scheduler:

```
root@admin#  kubectl exec -ti control-node-0 -c openpbs-server -- /bin/bash
```

We take the “nico” identity:

```
root@control-node-0#  su - nico
```

We print resources:

```
nico@control-node-0$ pbsnodes -a
```

```
hpc-node-0
        state = job-busy
        [...]

hpc-node-1
        state = free
        [...]
```

We check the queue:

```
nico@control-node-0$ qstat
```

```
Job id            Name             User             Time Use S Queue
----------------  ---------------- ----------------  -------- - -----
0.control-node-0  infinite         nico             00:00:00 R COMPUTE
```

The “infinite” job keeps on running after containerized HPC cluster shrink. We exit the “nico” account and the container itself to go back on the admin virtual machine:

```
nico@control-node-0$ exit
```

```
logout
```

```
root@control-node-0# exit
exit
```

We destroy our containerized HPC cluster:

```
root@admin# kubectl delete -f openpbs.yaml
service "nodes" deleted
statefulset.apps "hpc-node" deleted
statefulset.apps "control-node" deleted
```

We check that all pods have been destroyed:

```
root@admin# kubectl get pods
No resources found in hpc-nico namespace
```

As the reader can observe, we gave many details related to the OpenPBS use case. The methodology is the same for both remaining HPC schedulers, and we will be briefer for them.

2) OAR case study

We switch to the “oar” directory, run the manifest and check that all pods are running:

```
root@admin# cd ../oar/
```

```
root@admin# kubectl apply -f oar.yaml
service/nodes created
statefulset.apps/hpc-node created
statefulset.apps/control-node created
statefulset.apps/db-node created
```

```
root@admin# kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
control-node-0   2/2     Running   0          6m57s
db-node-0        1/1     Running   0          6m57s
hpc-node-0       1/1     Running   0          6m57s
hpc-node-1       1/1     Running   0          6m38s
```

There are four pods because OAR uses a database to store job information (db-node-0). We also have a master node control-node-0 (almighty) and two compute nodes hpc-node-[0,1].

We log in control-node-0 and take the “nico” identity to list resources:

```
root@admin# kubectl exec -ti control-node-0 -c oar-server -- /bin/bash
```

```
root@control-node-0# su - nico
```

We list nodes available for OAR. We have four results because OAR characterizes cores. We have 2 nodes with 2 cores each. So, the total number of cores is 4. The resources may take a few seconds to appear.

```
nico@control-node-0$ oarnodes
network_address: hpc-node-0
resource_id: 1
state: Alive
properties: cpu=1, core=1, last_available_upto=0, desktop_computing=NO, host=hpc-node-0, deploy=NO, drain=NO, network_address=hpc-node-0, cpuset=0, available_upto=2147483647, besteffort=YES, type=default

network_address: hpc-node-0
resource_id: 2
state: Alive
properties: core=2, last_available_upto=0, desktop_computing=NO, cpu=1, host=hpc-node-0, deploy=NO, drain=NO, network_address=hpc-node-0, cpuset=0, available_upto=2147483647, besteffort=YES, type=default

network_address: hpc-node-1
resource_id: 3
state: Alive
properties: deploy=NO, host=hpc-node-0, core=1, last_available_upto=0, desktop_computing=NO, cpu=1, cpuset=0, network_address=hpc-node-1, drain=NO, available_upto=2147483647, type=default, besteffort=YES

network_address: hpc-node-1
resource_id: 4
state: Alive
properties: available_upto=2147483647, type=default, besteffort=YES, cpu=1, core=2, last_available_upto=0, desktop_computing=NO, deploy=NO, host=hpc-node-0, network_address=hpc-node-1, drain=NO, cpuset=0
```

We observe four resources that respectively map to hpc-node-0 / core 1, hpc-node-0 / core 2, hpc-node-1 / core 1 and hpc-node-1 / core 2. Then, we move to oar directory to submit the”infinite” job:

```
nico@control-node-0:~$ cd oar/
```

We run the “infinite” job to occupy one node:

```
nico@control-node-0$ oarsub -S ./infinite.oar
[ADMISSION RULE] Set default walltime to 7200.
[ADMISSION RULE] Modify resource description with type constraints
OAR_JOB_ID=1
```

We check the job:

```
nico@control-node-0$ oarstat
Job id  S User           Duration   System message
--------- - -------- ---------- ------------------------------------------------
1       R nico           0:00:37 R=2,W=2:0:0,J=B,N=oarsub (Karma=0.000,quota_ok)
```

We submit the “pi” MPI job:

```
nico@control-node-0$ oarsub -S ./pi.oar
[ADMISSION RULE] Set default walltime to 7200.
[ADMISSION RULE] Modify resource description with type constraints
OAR_JOB_ID=2
```

And we check the queue:

```
nico@control-node-0$ oarstat
Job id  S User           Duration   System message
--------- - -------- ---------- ------------------------------------------------
1       R nico           0:04:24 R=2,W=2:0:0,J=B,N=oarsub (Karma=0.000,quota_ok)
2       W nico           0:00:00 R=4,W=2:0:0,J=B,N=oarsub (Karma=0.000,quota_ok)
```

The second job is in state “W” (waiting). We exit to the “admin” node:

```
nico@control-node-0$ exit
```

```
root@control-node-0# exit
```

And we add a compute node:

```
root@admin# kubectl patch statefulsets hpc-node -n hpc-nico -p '{"spec":{"replicas":3}}'
statefulset.apps/hpc-node patched
```

We check the available pods:

```
root@admin# kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
control-node-0   2/2   Running   0          8m52s
db-node-0      1/1     Running   0          8m52s
hpc-node-0     1/1     Running   0          8m52s
hpc-node-1     1/1     Running   0          8m47s
hpc-node-2     1/1     Running   0          15s
```

We go back to the control-node-0, su to “nico” and go back to oar subdirectory:

```
root@admin# kubectl exec -ti control-node-0 -c oar-server -- /bin/bash
```

```
root@control-node-0# su - nico
```

```
nico@control-node-0$ cd oar/
```

After few seconds, you can check the queue:

```
nico@control-node-0$ oarstat
Job id  S User          Duration   System message
--------- - -------- ---------- ------------------------------------------------
1         R nico          0:03:08 R=2,W=2:0:0,J=B,N=oarsub (Karma=0.000,quota_ok)
```

Only the first infinite job remains. We also observe that the pi MPI job ends successfully:

```
nico@control-node-0$ cat OAR.oarsub.2.stdout
Elapsed time = 0.000005 seconds
Pi is approximately 2.8750000000000000, Error is 0.2665926535897931
```

We exit the “admin” node:

```
nico@control-node-0$ exit
```

```
root@control-node-0# exit
```

We shrink the OAR cluster from 3 nodes to 2:

```
root@admin# kubectl patch statefulsets hpc-node -n hpc-nico -p '{"spec":{"replicas":2}}'
statefulset.apps/hpc-node patched
```

We check that the node disappeared:

```
root@admin# kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
control-node-0   2/2     Running   0          16m
db-node-0        1/1     Running   0          16m
hpc-node-0       1/1     Running   0          16m
hpc-node-1       1/1     Running   0          16m
```

We go back to control-node-0 as “nico” to list queue and available resources:

```
root@admin# kubectl exec -ti control-node-0 -c oar-server -- /bin/bash
```

```
root@control-node-0# su - nico
```

```
nico@control-node-0$ oarstat
Job id  S User           Duration   System message
--------- - -------- ---------- ------------------------------------------------
1       R nico           0:15:44 R=2,W=2:0:0,J=B,N=oarsub (Karma=0.000,quota_ok)
```

```
nico@control-node-0$ oarnodes
```

```
network_address: hpc-node-0
resource_id: 1
state: Alive
jobs: 1

network_address: hpc-node-0
resource_id: 2
state: Alive
jobs: 1

network_address: hpc-node-1
resource_id: 3
state: Alive

network_address: hpc-node-1
resource_id: 4
state: Alive

network_address: hpc-node-2
resource_id: 5
state: Dead

network_address: hpc-node-2
resource_id: 6
state: Dead
```

You can see that the removed resources (hpc-node-2) are tagged with de “Dead” state. We go back to “admin” node:

```
nico@control-node-0$ exit
```

```
root@control-node-0# exit
```

At last, we destroy our Oar cluster:

```
root@admin# kubectl delete -f oar.yaml
```

```
service "nodes" deleted
statefulset.apps "hpc-node" deleted
statefulset.apps "control-node" deleted
statefulset.apps "db-node" deleted
```

```
root@admin# kubectl get pods
```

```
No resources found in hpc-nico namespace
```

3) SLURM case study

We now instantiate our containerized SLURM HPC cluster:

```
root@admin# cd ../slurm/
```

```
root@admin# kubectl apply -f slurm.yaml
```

```
service/nodes created
statefulset.apps/hpc-node created
statefulset.apps/control-node created
```

We check that pods are healthy:

```
root@admin# kubectl get pods
```

```
NAME           READY   STATUS    RESTARTS   AGE
control-node-0   3/3   Running   0          2m37s
hpc-node-0     2/2     Running   0          2m37s
hpc-node-1     2/2     Running   0          109s
```

We have two containerized worker nodes (slurmd) and a master node (slurmctld). Now we display resources seen by the HPC scheduler with our regular account “nico”:

```
root@admin# kubectl exec -ti control-node-0 -c slurmctld -- /bin/bash
```

```
root@control-node-0# su - nico
```

```
nico@control-node-0$ sinfo
```

```
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
COMPUTE*    up   infinite      2   idle hpc-node-[0-1]
```

We submit the “infinite” job:

```
nico@control-node-0$ cd slurm/
```

```
nico@control-node-0$ sbatch infinite.slurm
Submitted batch job 1
```

```
nico@control-node-0$ squeue
             JOBID PARTITION     NAME  USER ST       TIME  NODES NODELIST(REASON)
                 1   COMPUTE infinite          nico  R          0:02     1 hpc-node-0
```

We submit the “pi” job:

```
nico@control-node-0$ sbatch pi.slurm
Submitted batch job 2
```

```
nico@control-node-0$ squeue
             JOBID PARTITION     NAME  USER ST       TIME  NODES NODELIST(REASON)
                 2   COMPUTE               pi       nico PD          0:00     2 (Resources)
                 1   COMPUTE infinite          nico  R          1:01     1 hpc-node-0
```

We go back to the “admin” node:

```
nico@control-node-0$ exit
```

```
root@control-node-0# exit
```

We expand our SLURM cluster from 2 compute nodes to 3:

```
root@admin# kubectl patch statefulsets hpc-node -n hpc-nico -p '{"spec":{"replicas":3}}'
statefulset.apps/hpc-node patched
```

We check that the new node is running :

```
root@admin# kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
control-node-0   3/3     Running   0          3m39s
hpc-node-0     2/2     Running   0          3m39s
hpc-node-1     2/2     Running   0          2m54s
hpc-node-2     2/2     Running   0          8s
```

We go back to control-node-0 with “nico” account to check the “pi” job:

```
root@admin# kubectl exec -ti control-node-0 -c slurmctld -- /bin/bash
```

```
root@control-node-0# su - nico
```

```
nico@control-node-0$ cd slurm/
```

```
nico@control-node-0$ sinfo
```

```
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
COMPUTE*    up   infinite      1  alloc hpc-node-0
COMPUTE*    up   infinite      2   idle hpc-node-[1-2]
```

```
nico@control-node-0$ squeue
```

```
             JOBID PARTITION     NAME  USER ST       TIME  NODES NODELIST(REASON)
                 1   COMPUTE infinite             nico  R            3:22      1 hpc-node-0
```

When we check the output file, we notice that it fails due to the lack of dynamic node addition support in SLURM, as we stated in our paper in the 4.4 section “Impact on pending jobs”:

```
nico@control-node-0$ cat pi.err
```

```
srun: error: get_addr_info: getaddrinfo() failed: Name or service not known
srun: error: slurm_set_addr: Unable to resolve "hpc-node-2"
srun: error: fwd_tree_thread: can't find address for host hpc-node-2, check slurm.conf
srun: error: Task launch for StepId=2.0 failed on node hpc-node-2: Can't find an address,
check slurm.conf
[...]
```

Note: sometimes, the job hangs forever. You may type in “scancel 2” to remove it.

We resubmit the job:

```
nico@control-node-0$ sbatch pi.slurm
```

```
Submitted batch job 3
```

We check the output of the resubmitted “pi” job:

```
nico@control-node-0$ cat pi.out
```

```
Elapsed time = 0.000006 seconds
Pi is approximately 3.3750000000000000, Error is 0.2334073464102069
```

This time, it works! This point is due to the ability of SLURM to create a communication infrastructure for jobs. If a job tries to use the new coming node, it fails. It seems that a failure triggers a recomputation of the communication infrastructure. That’s why the second submission works. For the moment, SLURM needs to be restarted at each resource addition.

We go back to “admin” node:

```
nico@control-node-0$ exit
```

```
root@control-node-0# exit
```

Now, we shrink our containerized HPC cluster from 3 to 2 compute nodes:

```
root@admin# kubectl patch statefulsets hpc-node -n hpc-nico -p '{"spec":{"replicas":2}}'
```

```
statefulset.apps/hpc-node patched
```

And check that hpc-node-2 disappeared:

```
root@admin# kubectl get pods
```

```
NAME           READY   STATUS    RESTARTS   AGE
control-node-0   3/3   Running   0          9m24s
hpc-node-0     2/2     Running   0          9m24s
hpc-node-1     2/2     Running   0          8m39s
```

We go back to control-node-0 as “nico”:

```
root@admin# kubectl exec -ti control-node-0 -c slurmctld -- /bin/bash
```

```
root@control-node-0# su - nico
```

We check cluster state:

```
nico@control-node-0$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
COMPUTE*    up   infinite      1  alloc hpc-node-0
COMPUTE*    up   infinite      1   idle hpc-node-1
```

And finally we print the queue:

```
nico@control-node-0$ squeue
             JOBID PARTITION     NAME  USER ST       TIME  NODES NODELIST(REASON)
                 1   COMPUTE infinite             nico  R            8:01      1 hpc-node-0
```

The “infinite” job keeps on running. We exit the session, and we move to the “admin” node:

```
nico@control-node-0$ exit
```

```
root@control-node-0# exit
```

At last, we delete the containerized cluster:

```
root@admin# kubectl delete -f slurm.yaml
service "nodes" deleted
statefulset.apps "hpc-node" deleted
statefulset.apps "control-node" deleted
```

```
root@admin# kubectl get pods
No resources found in hpc-nico namespace
```

ANNEXE C

ARTEFACT SUPERCOMPCLOUD 2022

CHSC : Containerized HPC Schedulers in the Cloud

This project implements a Kubernetes controller to run containerized HPC schedulers in the Cloud with autoscaling features. It requires a Kubernetes cluster. This repository contains the code of the controller with all Kubernetes manifests to instantiate containerized HPC schedulers. It should be noticed that for the moment only OAR HPC scheduler is supported. However, we demonstrated that each of the major HPC schedulers (SLURM, OpenPBS and OAR) can grow or shrink dynamically in the paper "A Methodology to Scale Containerized HPC Infrastructures in the Cloud" with the companion artifact available here. However controllers supplied with Kubernetes do not handle HPC computing oriented Pods. As a consequence all scaling actions had to be performed by hand denying autoscaling. This project is an implementation of a HPC dedicated controller for Kubernetes that talks with containerized HPC schedulers to perform autoscaling when needed.

1. Get your Kubernetes cluster

We supply a functional Kubernetes cluster here. Here is the MD5 fingerprint:

```
ngreneche@DESKTOP-KSGVO8B:/mnt/d/Offline$ md5sum chsc.ova
9bf5dcbd55dd3e764629abe7879e3b1a  chsc.ova
```

This file is an OVA image that contains our experimentation platform. We use the VirtualBox hypervisor throughout the document, and you can import the OVA file straight to VirtualBox. The machine hosting VirtualBox must be able to run virtual machines that use six vcpus in total. After deploying the OVA image, you get four virtual machines:

- CHSC-Lab-admin: host the Kubernetes API-Server ;
- CHSC-Lab-frontal: gateway to connect to the testbed ;
- CHSC-Lab-node[1,2]: cloud nodes that host our containerized HPC infrastructure.

The virtual machine "CHSC-Lab-frontal" has two network interfaces, one configured as NAT and one as an internal network where all the other virtual machines are plugged. Other virtual machines use "Lab-frontal" as a gateway to access the internet. We also run an internal local domain named "lab.local", such that "CHSC-Lab-frontal" embed a DNS server that redirects all DNS resolution out of "lab.local" to Google public DNS 8.8.8.8. The hosting machine must also reach Github and Docker Hub servers.

Kubernetes installation is based on the basic procedure supplied by Google here. We wrote an Ansible deployment based on this procedure. You can find it here.

We defined a port redirection on "CHSC-Lab-frontal" to reach the node from the localhost of your computer on port 2424. Default login/password is root/azerty for all virtual machines. To connect to the "CHSC-Lab-frontal node", type:

```
nico@DESKTOP-KSGVO8B:~$ ssh -p 2424 root@localhost
```

Go to the "admin" node:

```
root@frontal:~# ssh admin
```

And check that the Kubernetes cluster is running:

```
root@admin:~#  kubectl get nodes
NAME    STATUS   ROLES                  AGE   VERSION
admin   Ready    control-plane,master   48m   v1.22.3
node1   Ready    <none>                 46m   v1.22.3
node2   Ready    <none>                 46m   v1.22.3
```

2. Install CHSC requirements

First, clone the repository:

```
root@admin:~# git clone https://github.com/Nyk0/chsc.git
```

Then go to "chsc" directory:

```
root@admin:~# cd chsc/
```

Create the "oar" Kubernetes namespace and set it as your default namespace:

```
root@admin:~/chsc# kubectl create namespace oar
namespace/oar created
```

```
root@admin:~/chsc# kubectl config set-context --current --namespace=oar
Context "kubernetes-admin@kubernetes" modified.
```

Set the RBAC policy:

```
root@admin:~/chsc# kubectl apply -f misc/rbac/rbac.yaml
clusterrole.rbac.authorization.k8s.io/pods-list-oar created
clusterrolebinding.rbac.authorization.k8s.io/pods-list-oar created
```

Create the PhysicalVolume and the PhysicalVolumeClaim that maps the supplied NFS server that contains user's home directory :

```
root@admin:~/chsc# kubectl apply -f misc/nfs/pv-nfs.yaml
persistentvolume/pv-nfs-home created
```

```
root@admin:~/chsc# kubectl apply -f misc/nfs/pvc-nfs.yaml
persistentvolumeclaim/pvc-nfs-home created
```

3. Instantiate the containerized OAR scheduler

Go to the "oar" directory and run the YAML manifest:

```
root@admin:~/chsc# cd oar
```

```
root@admin:~/chsc/oar# kubectl apply -f oar.yaml
configmap/oarconf created
service/nodes created
pod/hpc-scheduler created
pod/db-server created
pod/controller created
```

Containerized OAR cluster is creating:

```
root@admin:~/chsc/oar# kubectl get pods
NAME            READY   STATUS              RESTARTS   AGE
controller      0/1     ContainerCreating   0          11s
db-server       0/1     ContainerCreating   0          11s
hpc-scheduler   0/1     ContainerCreating   0          11s
```

Wait a few seconds until the Pods reach the "Running" state:

```
root@admin:~/chsc/oar# kubectl get pods
NAME            READY   STATUS    RESTARTS   AGE
controller      1/1     Running   0          52s
db-server       1/1     Running   0          52s
hpc-scheduler   1/1     Running   0          52s
```

You have three Pods:

- "controller" runs the controller that enables autoscaling of the containerized OAR scheduler. It will create or remove containerized compute nodes depending on the pending jobs in the queue ;
- "db-server" is used by containerized OAR to store all its information such as jobs, resources topology, queue etc. ;
- "hpc-scheduler" contains Almighty, the server part of OAR.

No containerized compute nodes are created for the moment.

4. Submitting the first job

Open a bash sesssion in "hpc-scheduler" Pod:

```
root@admin:~/chsc/oar# kubectl exec -ti hpc-scheduler -- /bin/bash
```

Switch to a regular user :

```
root@hpc-scheduler:/# su - nico
```

```
nico@hpc-scheduler:~$ id
uid=1000(nico) gid=1000(nico) groups=1000(nico)
```

Check containerized compute nodes status:

```
nico@hpc-scheduler:~$ oarnodes -s
hpc-node-0
    1 : Absent (standby)
    2 : Absent (standby)
hpc-node-1
    3 : Absent (standby)
    4 : Absent (standby)
hpc-node-2
    5 : Absent (standby)
    6 : Absent (standby)
```

Nodes are pre-provisioned and flagged with an "Absent" state Go to "oar" directory:

```
nico@hpc-scheduler:~$ cd oar/
```

We submit the first job:

```
nico@hpc-scheduler:~/oar$ oarsub -S ./infinite.oar
[ADMISSION RULE] Set default walltime to 7200.
[ADMISSION RULE] Modify resource description with type constraints
OAR_JOB_ID=1
```

Immediately after the submission, we check the status of the queue:

```
nico@hpc-scheduler:~/oar$ oarstat
Job id    S User     Duration   System message
--------- - -------- ---------- ------------------------------------------------
1         W nico        0:00:00 No enough matching resources (no_matching_slot)
```

Our job is pending, waiting for resources to appear.

5. Autoscaling up (horizontal scaling)

Let's switch to another screen and get Pods status:

```
root@admin:~# kubectl get pods
NAME            READY   STATUS    RESTARTS   AGE
controller      1/1     Running   0          4m28s
db-server       1/1     Running   0          4m28s
hpc-node-0      1/1     Running   0          36s
hpc-scheduler   1/1     Running   0          4m28s
```

A new node appeared. Go back to previous screen on the regular user session in "hpc-scheduler" Pod and check containerized compute nodes status:

```
nico@hpc-scheduler:~/oar$ oarnodes -s
hpc-node-0
    1 : Alive
    2 : Alive
hpc-node-1
    3 : Absent (standby)
    4 : Absent (standby)
hpc-node-2
    5 : Absent (standby)
    6 : Absent (standby)
```

The containerized compute node "hpc-node-0" switched from "Absent" to "Alive" state. We check the state of our first job:

```
nico@hpc-scheduler:~/oar$ oarstat
Job id    S User     Duration   System message
--------- - -------- ---------- ------------------------------------------------
1         R nico        0:00:57 R=2,W=2:0:0,J=B,N=oarsub (Karma=0.000,quota_ok)
```

The job is running on the new Pod created by the controller. The Pod "hpc-node-0" was dynamically created according to pending job in the queue and the job has been scheduled on the newcomer Pods by OAR scheduler.

6. Autoscaling down

We submit a second job:

```
nico@hpc-scheduler:~/oar$ oarsub -S ./infinite.oar
[ADMISSION RULE] Set default walltime to 7200.
[ADMISSION RULE] Modify resource description with type constraints
OAR_JOB_ID=2
```

And check the queue immediately after:

```
nico@hpc-scheduler:~/oar$ oarstat
Job id    S User     Duration   System message
--------- - -------- ---------- ------------------------------------------------
1         R nico        0:14:36 R=2,W=2:0:0,J=B,N=oarsub (Karma=0.000,quota_ok)
2         W nico        0:00:00 R=2,W=2:0:0,J=B,N=oarsub (Karma=0.000,quota_ok)
```

The previous job is always running and the second is pending, waiting for resources to appear. Let's switch to another terminal to check Pods status:

```
root@admin:~# kubectl get pods
NAME            READY   STATUS    RESTARTS   AGE
controller      1/1     Running   0          19m
db-server       1/1     Running   0          19m
hpc-node-0      1/1     Running   0          15m
hpc-node-1      1/1     Running   0          43s
hpc-scheduler   1/1     Running   0          19m
```

A new Pod "hpc-node-1" appeared. Now, go back to user session:

```
nico@hpc-scheduler:~/oar$ oarstat
Job id    S User     Duration   System message
--------- - -------- ---------- ------------------------------------------------
1         R nico        0:15:37 R=2,W=2:0:0,J=B,N=oarsub (Karma=0.000,quota_ok)
2         R nico        0:00:47 R=2,W=2:0:0,J=B,N=oarsub (Karma=0.000,quota_ok)
```

The second job is running on "hpc-node-1". Let's delete the first job:

```
nico@hpc-scheduler:~/oar$ oardel 1
Deleting the job = 1 ...REGISTERED.
The job(s) [ 1 ] will be deleted in the near future.
```

Few seconds later, the job disappeared from the queue:

```
nico@hpc-scheduler:~/oar$ oarstat
Job id    S User     Duration   System message
--------- - -------- ---------- ------------------------------------------------
2         R nico        0:01:38 R=2,W=2:0:0,J=B,N=oarsub (Karma=0.000,quota_ok)
```

On the other terminal we can check the state of Pods:

```
root@admin:~# kubectl get pods
NAME            READY   STATUS        RESTARTS   AGE
controller      1/1     Running       0          20m
db-server       1/1     Running       0          20m
hpc-node-0      1/1     Terminating   0          17m
hpc-node-1      1/1     Running       0          2m7s
hpc-scheduler   1/1     Running       0          20m
```

The Pod "hpc-node-0" is terminating. After few seconds it has disappeared from the list of Pods:

```
root@admin:~# kubectl get pods
NAME            READY   STATUS    RESTARTS   AGE
controller      1/1     Running   0          21m
db-server       1/1     Running   0          21m
hpc-node-1      1/1     Running   0          2m28s
hpc-scheduler   1/1     Running   0          21m
```

We go back to user session to check the state of compute nodes:

```
nico@hpc-scheduler:~/oar$ oarnodes -s
hpc-node-0
    1 : Absent (standby)
    2 : Absent (standby)
hpc-node-1
    3 : Alive
    4 : Alive
hpc-node-2
    5 : Absent (standby)
    6 : Absent (standby)
```

The containerized compute node "hpc-node-0" switched from "Alive" to "Absent" state.

TABLE DES FIGURES

LISTE DES TABLEAUX

BIBLIOGRAPHIE

[1] Gordon E Moore. « Cramming more components onto integrated circuits ». In : *Proceedings of the IEEE* 86.1 (1998), p. 82-85.

[2] Shekhar Borkar et Andrew A Chien. « The future of microprocessors ». In : *Communications of the ACM* 54.5 (2011), p. 67-77.

[3] George W Platzman. « The ENIAC computations of 1950—Gateway to numerical weather prediction ». In : *Bulletin of the American Meteorological Society* 60.4 (1979), p. 302-312.

[4] Christophe Cérin, Nicolas Greneche et Tarek Menouer. « Towards pervasive containerization of HPC job schedulers ». In : *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE. 2020, p. 281-288.

[5] Marcin Copik et al. « Software Resource Disaggregation for HPC with Serverless Computing ». In : *ACM Student Research Competition (SRC), ACM/IEEE Supercomputing. Poster. https://sc22. supercomputing. org/proceedings/src_poster/src_poster_pages/sp html* (2022).

[6] Jerome H Saltzer et Michael D Schroeder. « The protection of information in computer systems ». In : *Proceedings of the IEEE* 63.9 (1975), p. 1278-1308.

[7] Dhammika Elkaduwe, Gerwin Klein et Kevin Elphinstone. « Verified protection model of the seL4 microkernel ». In : *Verified Software: Theories, Tools, Experiments: Second International Conference, VSTTE 2008, Toronto, Canada, October 6-9, 2008. Proceedings 2*. Springer. 2008, p. 99-114.

[8] Gernot Heiser et Kevin Elphinstone. « L4 microkernels: The lessons from 20 years of research and deployment ». In : *ACM Transactions on Computer Systems (TOCS)* 34.1 (2016), p. 1-29.

[9] Dan Hildebrand. « An Architectural Overview of QNX. » In : 1992.

[10] Qamar Jabeen et al. « A survey: Embedded systems supporting by different operating systems ». In : *arXiv preprint arXiv:1610.07899* (2016).

[11] Liangkai Liu et al. « Computing systems for autonomous driving: State of the art and challenges ». In : *IEEE Internet of Things Journal* 8.8 (2020), p. 6469-6486.

[12] David Barkai. *Unmatched — 50 years of Supercomputing*. Chapman & Hall, 2023. isbn : 9780367479619. url : https://www.routledge.com/Unmatched-50-Years-of-Supercomputing/Barkai/p/book/9780367479619.

[13] Daniel Reed, Dennis Gannon et Jack Dongarra. « HPC Forecast: Cloudy and Uncertain ». In : *Communications of the ACM* 66.2 (2023), p. 82-90.

[14] Kris Gaj et Arkadiusz Orłowski. « Facts and Myths of Enigma: breaking stereotypes ». In : *Proceedings of the 22nd international conference on Theory and applications of cryptographic techniques*. 2003, p. 106-122.

[15] Anthony E Sale. « Lorenz and Colossus [military cryptography] ». In : *Proceedings 13th IEEE Computer Security Foundations Workshop. CSFW-13*. IEEE. 2000, p. 216-222.

[16] Scott McCartney. *ENIAC: The triumphs and tragedies of the world's first computer*. Walker & Company, 1999.

[17] James E Thornton. « The cdc 6600 project ». In : *Annals of the History of Computing* 2.4 (1980), p. 338-348.

[18] George H Barnes et al. « The illiac iv computer ». In : *IEEE Transactions on computers* 100.8 (1968), p. 746-757.

[19] Paul B Schneck et Paul B Schneck. « The CDC STAR-100 ». In : *Supercomputer Architecture* (1987), p. 99-117.

[20] Richard M Russell. « The CRAY-1 computer system ». In : *Communications of the ACM* 21.1 (1978), p. 63-72.

[21] Melvin C. August et al. « Cray X-MP: The birth of a supercomputer ». In : *Computer* 22.1 (1989), p. 45-52.

[22] Wilfried Oed. « Cray Y-MP C90: System features and early benchmark results ». In : *Parallel Computing* 18.8 (1992), p. 947-954.

[23] Jack J Dongarra, Hans W Meuer, Erich Strohmaier et al. « TOP500 supercomputer sites ». In : *Supercomputer* 13 (1997), p. 89-111.

[24] Jack J Dongarra, Piotr Luszczek et Antoine Petitet. « The LINPACK benchmark: past, present and future ». In : *Concurrency and Computation: practice and experience* 15.9 (2003), p. 803-820.

[25] Ramune Arlauskas. « iPSC/2 system: A second generation hypercube ». In : *Proceedings of the third conference on Hypercube concurrent computers and applications: Architecture, software, computer systems, and general issues-Volume 1*. 1988, p. 38-42.

[26] Dror G Feitelson et Bill Nitzberg. « Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860 ». In : *workshop on job scheduling strategies for parallel processing*. Springer. 1995, p. 337-360.

[27] Jack Dongarra. « Report on the Fujitsu Fugaku system ». In : *University of Tennessee-Knoxville Innovative Computing Laboratory, Tech. Rep. ICLUT-20-06* (2020).

[28] Tamiko Thiel. « The design of the connection machine ». In : *Design Issues* 10.1 (1994), p. 5-18.

[29] Donald J Becker et al. « BEOWULF: A parallel workstation for scientific computation ». In : *Proceedings, international conference on parallel processing*. T. 95. 1995, p. 11-14.

[30] John D Owens et al. « GPU computing ». In : *Proceedings of the IEEE* 96.5 (2008), p. 879-899.

[31] Jonathan Hines. « Stepping up to summit ». In : *Computing in science & engineering* 20.2 (2018), p. 78-82.

[32] Mike Hartung. « IBM totalstorage enterprise storage server: A designer's view ». In : *IBM Systems Journal* 42.2 (2003), p. 383-396.

[33] Balazs Gerofi et al. « On the scalability, performance isolation and device driver transparency of the IHK/McKernel hybrid lightweight kernel ». In : *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2016, p. 1041-1050.

[34] David Schneider. « The Exascale Era is Upon Us: The Frontier supercomputer may be the first to reach 1,000,000,000,000,000,000 operations per second ». In : *IEEE spectrum* 59.1 (2022), p. 34-35.

[35] John Kim et al. « Technology-driven, highly-scalable dragonfly topology ». In : *ACM SIGARCH Computer Architecture News* 36.3 (2008), p. 77-88.

[36] David P Anderson et al. « SETI@ home: an experiment in public-resource computing ». In : *Communications of the ACM* 45.11 (2002), p. 56-61.

[37] Christophe Cérin et Gilles Fedak. *Desktop grid computing*. CRC Press, 2012.

[38] Vladimir Marjanović, José Gracia et Colin W Glass. « Performance modeling of the HPCG benchmark ». In : *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 5th International Workshop, PMBS 2014, New Orleans, LA, USA, November 16, 2014. Revised Selected Papers 5*. Springer. 2015, p. 172-192.

[39] Wu-chun Feng et Kirk Cameron. « The green500 list: Encouraging sustainable supercomputing ». In : *Computer* 40.12 (2007), p. 50-55.

[40] Mathieu Blanc et al. « Piga-hips: Protection of a shared hpc cluster ». In : *International Journal on Advances in Security Volume* 4 (2011).

[41] Rohit Chandra. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[42] Solmaz Salehian, Jiawen Liu et Yonghong Yan. « Comparison of threading programming models ». In : *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2017, p. 766-774.

[43] Amnon Barak et Oren La'adan. « The MOSIX multicomputer operating system for high performance cluster computing ». In : *Future Generation Computer Systems* 13.4-5 (1998), p. 361-372.

[44] Renaud Lottiaux et al. « OpenMosix, OpenSSI and Kerrighed: a comparative study ». In : *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.* T. 2. IEEE. 2005, p. 1016-1023.

[45] David Margery et al. « Kerrighed: a SSI cluster OS running OpenMP ». Thèse de doct. INRIA, 2003.

[46] Van Long Tran, Éric Renault et Viet Hai Ha. « CAPE: A Checkpointing-Based Solution for OpenMP on Distributed-Memory Architectures ». In : *Parallel Computing Technologies: 15th International Conference, PaCT 2019, Almaty, Kazakhstan, August 19–23, 2019, Proceedings 15*. Springer. 2019, p. 93-106.

[47] CS Wong et al. « Fairness and interactive performance of O (1) and CFS Linux kernel schedulers ». In : *2008 International Symposium on Information Technology*. T. 4. IEEE. 2008, p. 1-8.

[48] Jeff Dike. « A user-mode port of the Linux kernel ». In : *4th Annual Linux Showcase & Conference (ALS 2000)*. 2000.

[49] Edson Borin et al. *High Performance Computing in Clouds: Moving HPC Applications to a Scalable and Cost-Effective Environment*. Springer Nature, 2023.

[50] Angel M Beltre et al. « Enabling HPC workloads on cloud infrastructure using Kubernetes container orchestration mechanisms ». In : *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. IEEE. 2019, p. 11-20.

[51] Viktória Spišaková, Dalibor Klusáček et Lukáš Hejtmánek. « Using Kubernetes in Academic Environment: Problems and Approaches ». In : *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2022, p. 235-253.

[52] Nicolas Capit et al. « A batch scheduler with high level components ». In : *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.* T. 2. IEEE. 2005, p. 776-783.

[53] Dong H Ahn et al. « Flux: A next-generation resource management framework for large HPC centers ». In : *2014 43rd International Conference on Parallel Processing Workshops*. IEEE. 2014, p. 9-17.

[54] Claudia Misale et al. « Towards standard Kubernetes scheduling interfaces for converged computing ». In : *Smoky Mountains Computational Sciences and Engineering Conference*. Springer. 2021, p. 310-326.

[55] Daniel Milroy et al. « KubeFlux: A scheduler plugin bridging the cloud-HPC gap in Kubernetes ». In : *Accelerated Data Analytics and Computing Institute*. 2022.

[56] Claudia Misale et al. « It's a Scheduling Affair: GROMACS in the Cloud with the KubeFlux Scheduler ». In : *2021 3rd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. IEEE. 2021, p. 10-16.

[57] Nikyle Nguyen et Doina Bein. « Distributed MPI cluster with Docker Swarm mode ». In : *2017 ieee 7th annual computing and communication workshop and conference (ccwc)*. IEEE. 2017, p. 1-7.

[58] Joshua Higgins, Violeta Holmes et Colin Venters. « Orchestrating docker containers in the HPC environment ». In : *High Performance Computing: 30th International Conference, ISC High Performance 2015, Frankfurt, Germany, July 12-16, 2015, Proceedings 30*. Springer. 2015, p. 506-513.

[59] Edson Borin et al. « Why Move HPC Applications to the Cloud? » In : *High Performance Computing in Clouds: Moving HPC Applications to a Scalable and Cost-Effective Environment*. Springer, 2023, p. 1-5.

[60] Edson Borin et Otávio O Napoli. « Deploying and Configuring Infrastructure ». In : *High Performance Computing in Clouds: Moving HPC Applications to a Scalable and Cost-Effective Environment*. Springer, 2023, p. 55-74.

[61] Rodrigo da Rosa Righi et al. « Designing Cloud-Friendly HPC Applications ». In : *High Performance Computing in Clouds: Moving HPC Applications to a Scalable and Cost-Effective Environment*. Springer, 2023, p. 99-126.

[62] Alexandre C Sena et al. « Harnessing Low-Cost Virtual Machines on the Spot ». In : *High Performance Computing in Clouds: Moving HPC Applications to a Scalable and Cost-Effective Environment*. Springer, 2023, p. 163-189.

[63] Rafaela Brum et al. « Ensuring Application Continuity with Fault Tolerance Techniques ». In : *High Performance Computing in Clouds: Moving HPC Applications to a Scalable and Cost-Effective Environment*. Springer, 2023, p. 191-212.

[64] Qingye Jiang, Young Choon Lee et Albert Y Zomaya. « The power of ARM64 in public clouds ». In : *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE. 2020, p. 459-468.

[65] Norman P Jouppi et al. « In-datacenter performance analysis of a tensor processing unit ». In : *Proceedings of the 44th annual international symposium on computer architecture*. 2017, p. 1-12.

[66] Daniel Reed, Dennis Gannon et Jack Dongarra. « Reinventing high performance computing: challenges and opportunities ». In : *arXiv preprint arXiv:2203.02544* (2022).

[67] Stefan Kehrer et Wolfgang Blochinger. « A survey on cloud migration strategies for high performance computing ». In : *Proceedings of the 13th Symposium and Summer School on Service-Oriented Computing (SummerSoc19)*. IBM Research Division. 2019, p. 57-69.

[68] Christoph Fehling et al. *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Springer, 2014.

[69] Emanuel Ferreira Coutinho et al. « Elasticity in cloud computing: a survey ». In : *annals of telecommunications-annales des télécommunications* 70 (2015), p. 289-309.

[70] Yahya Al-Dhuraibi et al. « Elasticity in cloud computing: state of the art and research challenges ». In : *IEEE Transactions on Services Computing* 11.2 (2017), p. 430-447.

[71] Stefan Kehrer et Wolfgang Blochinger. « Elastic parallel systems for high performance cloud computing: state-of-the-art and future directions ». In : *Parallel processing letters* 29.02 (2019), p. 1950006.

[72] Basma Abdel Azeem et Manal Helal. « Performance evaluation of checkpoint/-restart techniques: For MPI applications on Amazon cloud ». In : *2014 9th International Conference on Informatics and Systems*. IEEE. 2014, PDC-49.

[73] Joshua Hursey. *Coordinated checkpoint/restart process fault tolerance for MPI applications on HPC systems*. Indiana University, 2010.

[74] Miltiadis Siavvas et Erol Gelenbe. « Optimum interval for application-level checkpoints ». In : *2019 6th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2019 5th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*. IEEE. 2019, p. 145-150.

[75] CRIU Dec. *CRIU: Checkpoint/Restore in Userspace*. `https://criu.org`. 2015.

[76] Leonardo Araújo de Jesus, Lúcia MA Drummond et Daniel de Oliveira. « Eeny meeny miny moe: Choosing the fault tolerance technique for my cloud workflow ». In : *High Performance Computing: 4th Latin American Conference, CARLA 2017, Buenos Aires, Argentina, and Colonia del Sacramento, Uruguay, September 20-22, 2017, Revised Selected Papers 4*. Springer. 2018, p. 321-336.

[77] Luan Teylo et al. « A dynamic task scheduler tolerant to multiple hibernations in cloud environments ». In : *Cluster Computing* 24 (2021), p. 1051-1073.

[78] Luan Teylo et al. « Scheduling bag-of-tasks in clouds using spot and burstable virtual machines ». In : *IEEE Transactions on Cloud Computing* (2021).

[79] Marcos K Aguilera et al. « Communication-efficient leader election and consensus with limited link synchrony ». In : *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*. 2004, p. 328-337.

[80] Jason Ansel, Kapil Arya et Gene Cooperman. « Transparent Checkpointing for Cluster Computations and the Desktop ». In : (2009).

[81] Mohammed Amoon et al. « On the design of reactive approach with flexible checkpoint interval to tolerate faults in cloud computing systems ». In : *Journal of Ambient Intelligence and Humanized Computing* 10 (2019), p. 4567-4577.

[82] Julian Pistorius et al. « Exosphere-Bringing The Cloud Closer ». In : *2020 IEEE/ACM International Workshop on Interoperability of Supercomputing and Cloud Technologies (SuperCompCloud)*. IEEE. 2020, p. 1-6.

[83] E Czaplicki. « Elm - A delightful language for reliable webapps ». In : *Saatavissa (viitattu 13.10. 2017): http://elm-lang. org* (2017).

[84] Peter Fritzson. « Electron — Build cross platform desktop apps with JavaScript, HTML, and CSS ». In : *URL: http://electron.atom.io/(visited on 08/23/2023)* (2016).

[85] Felipe A Cruz et al. « FirecREST: a RESTful API to HPC systems ». In : *2020 IEEE/ACM International Workshop on Interoperability of Supercomputing and Cloud Technologies (SuperCompCloud)*. IEEE. 2020, p. 21-26.

[86] CRIU Dec. *Scalability and data security: deep learning with health data on future HPC platforms*. `https://www.stackhpc.com/sc19.html`. 2015.

[87] Tarek Menouer et al. « Towards an optimized containerization of HPC job schedulers based on namespaces ». In : *IFIP International Conference on Network and Parallel Computing*. Springer. 2021, p. 144-156.

[88] Alexandre Denis, Christian Pérez et Thierry Priol. « PadicoTM: An open integration framework for communication middleware and runtimes ». In : *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)*. IEEE. 2002, p. 144-144.

[89] Abdulrahman Azab. « Enabling docker containers for high-performance and many-task computing ». In : *2017 ieee international conference on cloud engineering (ic2e)*. IEEE. 2017, p. 279-285.

[90] Jorge Gomes et al. « Enabling rootless Linux Containers in multi-user environments: the udocker tool ». In : *Computer Physics Communications* 232 (2018), p. 84-97.

[91] John Paul Martin, A Kandasamy et K Chandrasekaran. « Exploring the support for high performance applications in the container runtime environment ». In : *Human-centric Computing and Information Sciences* 8 (2018), p. 1-15.

[92] Naweiluo Zhou, Huan Zhou et Dennis Hoppe. « Containerization for High Performance Computing Systems: Survey and Prospects ». In : *IEEE Transactions on Software Engineering* 49.4 (2022), p. 2722-2740.

[93] Reid Priedhorsky et Tim Randles. « Charliecloud: Unprivileged containers for user-defined software stacks in hpc ». In : *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 2017, p. 1-10.

[94] Gregory M Kurtzer, Vanessa Sochat et Michael W Bauer. « Singularity: Scientific containers for mobility of compute ». In : *PloS one* 12.5 (2017), e0177459.

[95] Douglas M Jacobsen et Richard Shane Canon. « Contain this, unleashing docker for hpc ». In : *Proceedings of the Cray User Group* (2015), p. 33-49.

[96] Donald Bahls. « Evaluating shifter for hpc applications ». In : *Cray User Group Conference Proceedings*. 2016.

[97] Alfred Torrez, Timothy Randles et Reid Priedhorsky. « HPC container runtimes have minimal or no performance impact ». In : *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. IEEE. 2019, p. 37-42.

[98] Andrew J Younge et al. « A tale of two systems: Using containers to deploy HPC applications on supercomputers and clouds ». In : *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2017, p. 74-81.

[99] Richard S Canon et Andrew Younge. « A case for portability and reproducibility of HPC containers ». In : *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. IEEE. 2019, p. 49-54.

[100] VÌctor Sande Veiga et al. « Evaluation and benchmarking of singularity mpi containers on eu research e-infrastructure ». In : *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. IEEE. 2019, p. 1-10.

[101] Hsi-En Yu et Weicheng Huang. « Building a virtual hpc cluster with auto scaling by the docker ». In : *arXiv preprint arXiv:1509.08231* (2015).

[102] Maximilien de Bayser et Renato Cerqueira. « Integrating MPI with Docker for HPC ». In : *2017 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2017, p. 259-265.

[103] Cristian Ruiz, Emmanuel Jeanvoine et Lucas Nussbaum. « Performance evaluation of containers for HPC ». In : *Euro-Par 2015: Parallel Processing Workshops: Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers 21*. Springer. 2015, p. 813-824.

[104] Miguel G Xavier et al. « Performance evaluation of container-based virtualization for high performance computing environments ». In : *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE. 2013, p. 233-240.

[105] Peini Liu et Jordi Guitart. « Performance characterization of containerization for HPC workloads on InfiniBand clusters: an empirical study ». In : *Cluster Computing* (2022), p. 1-22.

[106] Piotr R Luszczek et al. « The HPC Challenge (HPCC) benchmark suite ». In : *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. T. 213. 10.1145. 2006, p. 1.

[107] Wataru Takase et al. « Building an elastic batch system with private and public clouds ». In : *International Symposium on Grids & Clouds 2019*. 2019, p. 8.

[108] M Bauer. « Solving Problems in HPC with Singularity ». In : *CernVM Workshop*. 2019.

[109] Larry Smarr et Charles E Catlett. « Metacomputing ». In : *Communications of the ACM* 35.6 (1992), p. 44-52.

[110] Naweiluo Zhou et al. « Container orchestration on HPC systems through Kubernetes ». In : *Journal of Cloud Computing* 10.1 (2021), p. 1-14.

[111] Oleksandr Rudyy et al. « Containers in hpc: A scalability and portability study in production biological simulations ». In : *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2019, p. 567-577.

[112] Kun Suo et al. « An analysis and empirical study of container networks ». In : *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE. 2018, p. 189-197.

[113] Alessandro Amirante et Simon Pietro Romano. « Container nats and session-oriented standards: Friends or foe? » In : *IEEE Internet Computing* 23.6 (2019), p. 28-37.

[114] Yang-Suk Kee et al. « Enabling personal clusters on demand for batch resources using commodity software ». In : *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE. 2008, p. 1-7.

[115] Konrad Meier et al. « Dynamic provisioning of a HEP computing infrastructure on a shared hybrid HPC system ». In : *Journal of Physics: Conference Series*. T. 762. 1. IOP Publishing. 2016, p. 012012.

[116] Emmanuel Jeanvoine, Luc Sarzyniec et Lucas Nussbaum. « Kadeploy3: Efficient and scalable operating system provisioning for clusters ». In : *USENIX Association* 38.1 (2013), p. 38-44.

[117] Daniel Balouek et al. « Adding virtualization capabilities to the Grid'5000 testbed ». In : *Cloud Computing and Services Science: Second International Conference, CLOSER 2012, Porto, Portugal, April 18-21, 2012. Revised Selected Papers 2*. Springer. 2013, p. 3-20.

[118] Shuangcheng Niu et al. « Building semi-elastic virtual clusters for cost-effective HPC cloud resource provisioning ». In : *IEEE Transactions on Parallel and Distributed Systems* 27.7 (2015), p. 1915-1928.

[119] Benjamin Hindman et al. « Mesos: A platform for {Fine-Grained} resource sharing in the data center ». In : *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. 2011.

[120] Malte Schwarzkopf et al. « Omega: flexible, scalable schedulers for large compute clusters ». In : *Proceedings of the 8th ACM European Conference on Computer Systems*. 2013, p. 351-364.

[121] Pamela Delgado. « Efficiently Scheduling Data-Parallel Computations on Very Large Clusters ». In : ().

[122] Vinod Kumar Vavilapalli et al. « Apache hadoop yarn: Yet another resource negotiator ». In : *Proceedings of the 4th annual Symposium on Cloud Computing*. 2013, p. 1-16.

[123] Dhruba Borthakur. « The hadoop distributed file system: Architecture and design ». In : *Hadoop Project Website* 11.2007 (2007), p. 21.

[124] Nicolas Grenèche et al. « A methodology to scale containerized HPC infrastructures in the Cloud ». In : *European Conference on Parallel Processing*. Springer. 2022, p. 203-217.

[125] Nicolas Greneche et al. « Artifact and instructions to generate experimental results for Euro-Par 2022 conference proceedings: A methodology to scale containerized HPC infrastructures in the Cloud ». In : (août 2022). doi : 10.6084/m9.figshare.19952813.v1. url : https://springernature.figshare.com/articles/online_resource/Artifact_and_instructions_to_generate_experimental_results_for_Euro-Par_2022_conference_proceedings_A_methodology_to_scale_containerized_HPC_infrastructures_in_the_Cloud/19952813.

[126] Nicolas Greneche et Christophe Cerin. « Autoscaling of Containerized HPC Clusters in the Cloud ». In : *2022 IEEE/ACM International Workshop on Interoperability of Supercomputing and Cloud Technologies (SuperCompCloud)*. IEEE. 2022, p. 1-7.

[127] Théophile Terraz et al. « Melissa: large scale in transit sensitivity analysis avoiding intermediate files ». In : *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 2017, p. 1-14.

[128] Walid Saad et al. « A data prefetching model for desktop grids and the condor use case ». In : *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE. 2013, p. 1065-1072.

[129] Fabrice Bellard. « QEMU, a fast and portable dynamic translator. » In : *USENIX annual technical conference, FREENIX Track*. T. 41. Califor-nia, USA. 2005, p. 46.

[130] Xingyu Wang, Junzhao Du et Hui Liu. « Performance and isolation analysis of RunC, gVisor and Kata Containers runtimes ». In : *Cluster Computing* 25.2 (2022), p. 1497-1513.