Environnement logiciel et reproductibilité

Retour d'expérience d'un mésocentre de calcul

Pierre-Antoine Bouttier,
Violaine Louvet

23 mai 2019

Journée du réseau ARAMIS

L'UMS GriCAD en quelques mots (1/2)

- UMS : Unité Mixte de Services ayant pour tutelle CNRS, UGA, GINP, INRIA
- **Missions principales:**
  - Accompagnement et conseils aux chercheurs sur leurs besoins liés au calcul et aux données
  - Mise à disposition de l'ensemble des chercheurs et personnels en support de la recherche d'infrastructures avancées et mutualisées pour le calcul intensif et l'exploitation des données de la recherche.

Contacts

- Mail : `gricad-contact@univ-grenoble-alpes.fr`
- Site web : `https://gricad.univ-grenoble-alpes.fr/`

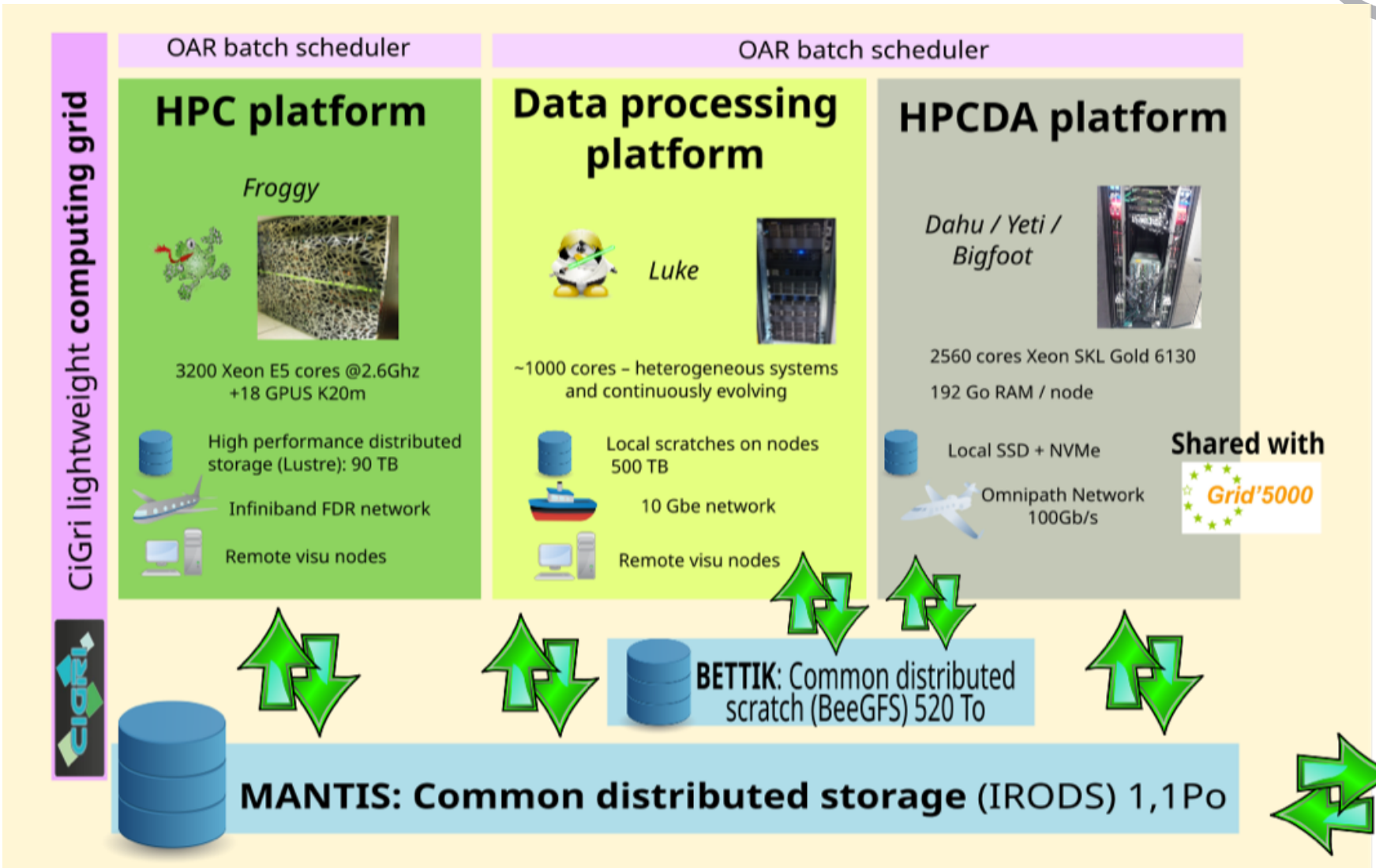
L'UMS GriCAD en quelques mots (2/2)

En bref, les services mis à disposition des personnels de la recherche du site grenoblois :

- Plateformes de développement logiciel et de travail collaboratif (GitLab, JupyterHub)
- VM à la demande (plateforme OpenStack)
- Expertise Calcul scientifique et traitement massif de données
- Ingénierie de la donnée
- Exploitation du mésocentre de calcul grenoblois

Dans le cadre de nos activités, nous sommes quotidiennement amenés à expertiser, packager ou développer des codes logiciels pour la recherche

Les plateformes de calcul à GriCAD (1/2)



Les plateformes de calcul à GriCAD (2/2)

Quel cluster pour quel usage ?

- Le calcul massivement parallèle : Froggy et Dahu
- L'analyse ou le traitement de données (jobs séquentiels) : Luke ou la grille
- Calcul sur GPU : 3 noeuds spécifiques (4 GPU Tesla V100 reliées NV-link par noeud) sur Dahu.

La grille (CiGri)

- Utilisation optimale des ressources (mode best effort)
- Intéressant pour un grand nombre de jobs utilisant peu de ressources (qques coeurs max)
- Re-soumission automatique

Quid de l'environnement logiciel sur nos calculateurs ?

Quelles solutions proposer aux utilisateurs pour développer et utiliser leurs codes logiciels sur nos calculateurs mais également dans tous les contextes où ils évoluent ?

Environnement logiciel et recherche (1/3)
Hétérogéneité du public

Des utilisateurs aux profils variés :

- Nous nous adressons à toutes les communautés recherche (maths applis, SHS, physique(s), chimie, médecine, biologie, économie, etc.)
- Services ouverts aux étudiants, IT, chercheurs
- Compétences en développement logiciel extrêmement disparates

Environnement logiciel et recherche (2/3)

Hétérogénéité de pratiques

Des codes logiciels très diverses :

- Nombreux langages de programmation (F90 parfois F77..., C/C++, Python, R, MatLab, Julia, Java, perl, ruby, etc.)
- Du script maison au projet international (Forge avec plusieurs dizaines de développeurs) en passant par des codes historiques
- Souvent, peu de ressources disponibles pour un développement logiciel "carré"

Environnement logiciel et recherche (3/3)

Hétérogéneité d'infrastructures

Des expériences sur de nombreux supports :

- Du bureau au tiers 0
- Environnements de compilation parfois spécifiques
- CPU, GPU, MPI, OpenMP, OpenACC, codes hybrides, HPC vs batch, etc.

Et la reproductibilité logicielle dans tout ça ?

Les grands atouts

- La reproductibilité, fondement de la méthodologie scientifique
- Facilite grandement le développement et le portage des codes
- Renforce la pérennité et l'archivage des codes
- ...

Les défis (à notre niveau)

- Faire face aux grandes hétérogénéités (variété des usages, pratiques et infras)
- Diffuser les bonnes pratiques
- Choisir, se former, exploiter et maintenir les technologies appropriés
- Contraintes de performances et de sécurité

Panorama éclair des solutions techniques

- Les gestionnaires de paquets système-dépendants : Spack, Easybuild, conda, etc.
- Les gestionnaires de paquets système-indépendants : Nix et Guix
- La conteneurisation : udocker, shift, charliecloud, singularity, podman, etc.

Les solutions techniques retenues

Pour le moment, deux solutions techniques sont proposées *officiellement* sur nos clusters :

- Un gestionnaire de paquet : Nix
- Une solution de conteneurisation : Charliecloud

Nix, le gestionnaire de paquets

NIX - Vue d'ensemble

Nix, le gestionnaire de paquets

- Gestionnaire de paquet fonctionnel (pas d'effets de bord)
- Fiable et reproductible
- Disponible sous linux et Mac OS X (portable)
- Création/installation de paquets sans les privilèges root : idéal pour le HPC

Nixpkgs, le dépôt de paquet principal

- Environ 50000 paquets
- Système-indépendant
- On peut créer son propre dépôt

Un paquet est décrit dans un langage fonctionnel, dérivé d'Haskell

```
{ stdenv, fetchurl }:

stdenv.mkDerivation rec {
  pname = "cutee";
  version = "0.4.2";
  name = "${pname}-${version}";

  src = fetchurl {
    url    = "http://www.codesink.org/download/${pname}-${version}.tar.gz";
    sha256 = "18bzvhzx8k24mpcim5669n3wg9hd0sfsxj8zjpbr24hywrlppgc2";
  };

  buildFlags = "cutee";

  installPhase = ''
    mkdir -p $out/bin
    cp cutee $out/bin
  '';

  meta = with stdenv.lib; {
    description = "C++ Unit Testing Easy Environment";
    homepage    = http://www.codesink.org/cutee_unit_testing.html;
    license     = licenses.gpl2Plus;
    maintainers = with maintainers; [ leenaars];
    platforms = platforms.linux;
  };
}
```

Nix - Les concepts

Stockage des paquets

- Un seul répertoire : `/nix/store`
- Indépendant du système

Identification des paquets

- Un paquet = un hash
- Un build de paquet = un sous-répertoire
- L'arbre des dépendances est préservé
- Unicité des paquets

- Installation d'un paquet : création d'un lien symbolique dans le `/home` utilisateur (vers `/nix/store`)
- Un profil nix : un environnement défini par un ensemble de liens symboliques stockés dans `/home/user/.nix-profile`
- On peut définir autant de profils que l'on veut et passer de l'un à l'autre avec *nix-env - -switch-profile*
- Versionnement des profils (rollback)

Nix - Les concepts

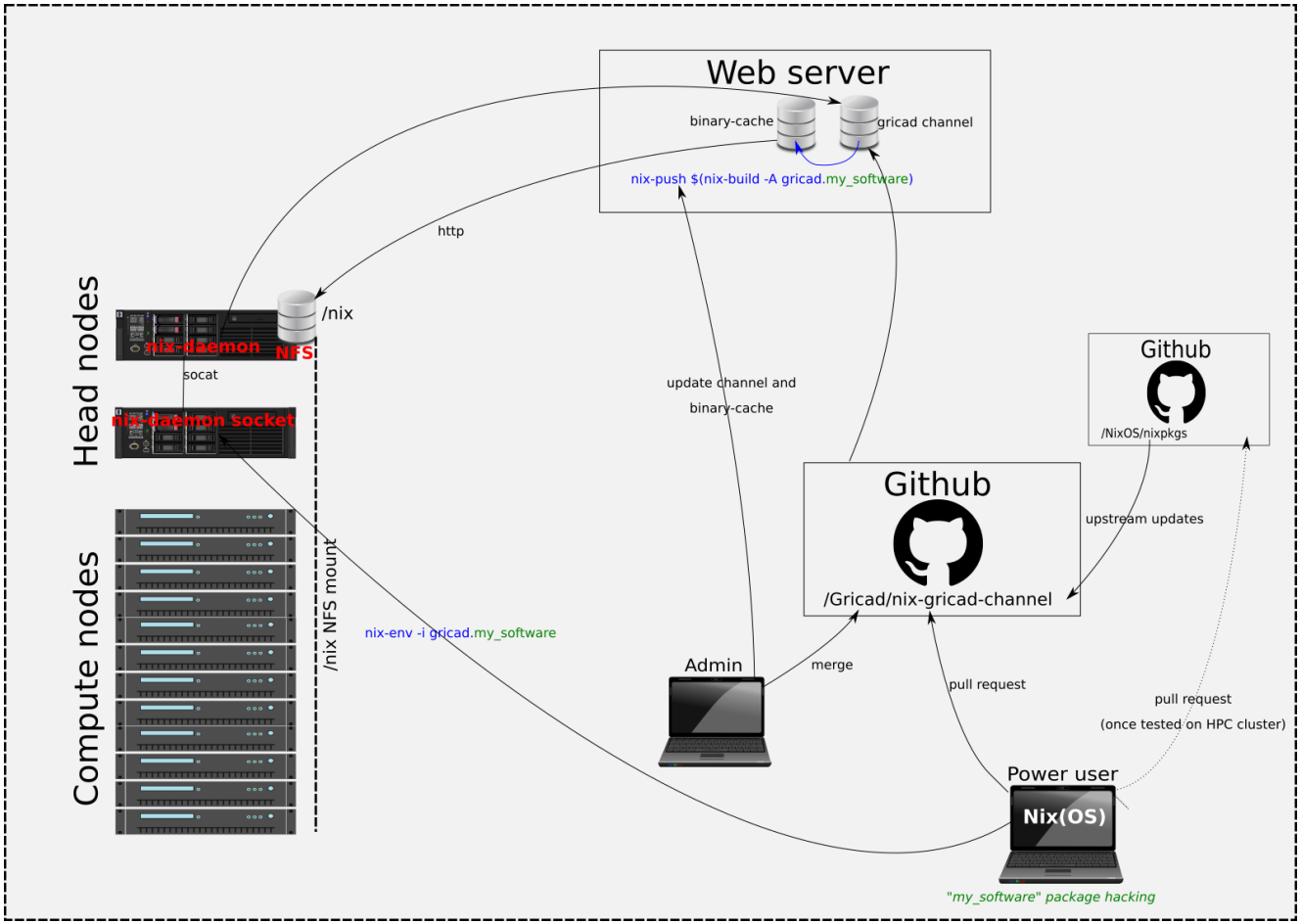
Channels

- Un snapshot de Nixpkgs + d'autres paquets
- Exemples : nixpkgs-unstable, ciment-channel,...

Le binary-cache

- Un serveur web qui fournit des paquets précompilés
- Évite la reconstruction de paquets déjà installés
- Un binary-cache partagé efficace dans notre contexte

Nix - Notre workflow



Nix IRL

Avantages

- Reproductible, portable et fiable
- Création et installation de paquets en espace utilisateur
- Binary-cache partagé
- Contribution à une communauté active

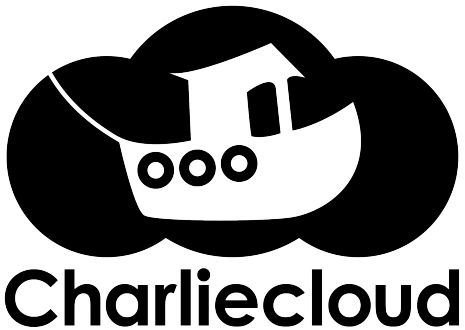
Obstacles

- Courbe d'apprentissage un peu raide...
- particulièrement pour des novices en programmation
- Quelques difficultés pour des codes "exotiques" (nombreux dans la recherche)
- Communauté HPC très faiblement représentée

Charliecloud, le gestionnaire de conteneurs

Charliecloud

- Charliecloud est un système de conteneur léger dédié au HPC
- Développé à Los Alamos depuis 2014
- Sous licence Apache License, Version 2.0
- Repository github : `https://github.com/hpc/charliecloud`
- Documentation : `https://hpc.github.io/charliecloud/`
- Cette présentation s'inspire fortement de la présentation de Michael Jennings à la conférence Swiss HPC 2018 (`https://www.youtube.com/watch?time_continue=2&v=ESsZgcaP-ZQ`)
- Actuellement : version 0.9.10 sortie en mai.



Charliecloud : comment ça marche ?

- Charliecloud utilise les linux user namespaces pour exécuter des containers sans privilège particulier, sans démon.
- Les problématiques de sécurité sont reportées sur le noyau linux
- C'est un système de container léger : ne fournit que le support d'exécution (pas le build).
- Les images des containers peuvent être construite avec docker ou tout autre système qui génère une arborescence linux standard.
- Charliecloud peut donc exécuter des containers Docker.
- Charliecloud s'appuie sur Docker pour la création de container.
- Charliecloud ne représente que 1 000 lignes de code contre 19 000 pour Shifter, 15 000 pour Singularity et 160 000 pour Docker !

Charliecloud : construction d'une image

- On peut partir d'une image Docker déjà réalisée.
- On peut construire une image :
  - Cela suppose d'avoir installé charliecloud + docker-ce sur une machine sur laquelle on a les droits root
  - Avec un noyau linux raisonnablement récent
  - On écrit un fichier Dockerfile standard
  - On construit l'image avec les commandes charliecloud:
    - Construction :

      ```
      ch-build -t nom_img ./
      ```

    - Compression pour transfert :

      ```
      ch-docker2tar nom_img ./.
      ```

    - On obtient un fichier tar.gz qu'on peut copier sur les machines de calcul.

Charliecloud : installation et utilisation

Installation du container

```
ch-tar2dir ../nom_img.tar.gz ./.
```

Pour des questions de performances, le mieux est de le faire dans un espace de stockage local.

Exécution du container

```
ch-run -b /home/:/home -w ./nom_img -- nom_exe
```

Il suffit d'intégrer cette ligne dans un script de soumission de job.

Charliecloud IRL

Avantages

- Léger, simple d'utilisation, peut s'installer et s'exécuter en espace utilisateur
- Compatible avec les images docker
- Assure la portabilité du conteneur d'une infrastructure à l'autre

Obstacles

- Des difficultés dès qu'on sort d'un noeud : l'utilisation de codes MPI nécessite en particulier une compatibilité entre les versions de l'hôte et du conteneur
- Taille des images
- Projet jeune, qui cherche encore un peu ses utilisateurs
- Le dilemne de la reproductibilité VS la sécurité qu'imposent les conteneurs

Conclusion générale

Les solutions techniques pour tendre vers une reproductibilité logicielle exemplaire existent.
Mais (en ce qui concerne la recherche)...

- Encore trop de développements hasardeux, chaotiques, non-pérennes : difficultés d'utiliser ces solutions
- Fort besoin de diffuser les bonnes pratiques en terme de développement logiciel. Et fort besoin de ressources pour réaliser ce travail

Merci de votre attention !
Questions/Discussion