Génie Logiciel pour le Calcul Scientifique

Marc Tajchman `marc.tajchman@cea.fr`
Julien Bigot `julien.bigot@cea.fr`

Table des matières

Idée : Façon de développer des application de calcul et surtout de les intégrer dans un framework. Le but est d'obtenir le plus rapidement des résultats sans avoir à tout recoder (affichage, calculs "simples", gestion de fichiers, ...) afin de se concentrer principalement sur l'algorithmique à implémenter.

1 Introduction

1.1 Notion d'API

Application Programming Interface : Indique les différentes "briques logicielle" (fonctions, classes, modules, etc) utilisables avec leur description. Pour chaque fonctionnalité, l'API documente :

- Le travail effectué (exemple : résolution d'un système linéaire)
- La méthode d'appel
- La description des entrées (types, formats et valeurs) (exemple : la matrice doit être inversible)
- La description des résultats de sortie (types et formats) (exemple : le vecteur de solutions *et* la précision de calcul ; si la sortie est une matrice, l'ordre des coefficients doit être précisé, ...)

Elle se présente sous la forme d'un ensemble de fichiers à inclure dans le code qui donne la *signature* ou *prototype* des fonctions apportées, ainsi qu'un manuel d'explication complétant les explications (ce n'est cependant malheureusement pas toujours le cas).

Attention, un code peu utiliser plusieurs API, et une API peut être commune à plusieurs code (par exemple, utiliser deux algorithmes différents pour le même calcul).

Parfois, une API *de haut niveau* est disponible, permettant d'utiliser des fonctions "simples" utilisant des paramètres par défaut (utile selon le "niveau d'expertise" de l'utilisateur).

L'intérêt d'une API réside dans la *réutilisation possible* du code : en effet, il est possible d'interchanger du code partageant la même API sans adaptations.

Il existe de nos jour de nombreuses API de confiance performantes, probablement plus qu'une implémentation à la main. Par exemple : FFTW pour la transformées de Fourier, LAPACK pour l'algèbre linéaire, MPI pour le calcul parallèle, ce dernier étant une interface dont de nombreuses librairies définissent différemment ces interfaces (OpenMPI par exemple). Avec l'achat d'une machine, il est possible que le constructeur offre une version modifiée de MPI adaptée pour cette machine ; mais toutes ces versions peuvent exécuter le même code.

Attention : Il faut réfléchir *au départ* sur le problème à résoudre et son intégration par rapport aux composantes systèmes, leurs interactions. Il est utile de vérifier sur des exemples triviaux le bon fonctionnement du code petit à petit ; mais également de vérifier que le code ne fonctionne pas lorsque les spécifications d'entrée ne sont pas respectées.

On peut commencer par utiliser "sur papier" un langage de modélisation tel ULM (*Universal Modeling Language*, un langage graphique).

1.2 Utilisation de Design Pattern

Masque de conception en français.

Par exemple :

- Factory (construction de données appartenant à une même classe générale)
- Sigleton (refuser la création de deux objets de même classe afin de garantir l'unicité)
- Iterator (parcours d'un ensemble)
- Observer (tirer des informations du code telles que l'affichage/sauvegarde ou des résultats partiels)
- Et pleins d'autres !

1.3 Réutilisation de code

Principe : écrire le moins de code possible soi-même. On utilise principalement deux voies : soit par l'utilisation de bibliothèques (*libraries*) ou l'utilisation d'une plateforme (*framework* ou *cadriciel*). Il est également possible de mélanger ces deux approches.

Librairie Ensemble logiciel réalisant un ensemble de traitement similaires ; elle ne peut pas s'exécuter seule mais est ajoutée au code afin d'utiliser ses fonctionnalités. Le plus souvent, un code utilisera plusieurs librairies. C'est le code qui gère le déroulement du calcul.

Dans un contexte HPC, il faut faire bien attention à ce que celles-ci sont capable de fonctionner en mode parallèle... Grands noms : PETSc, MUMPS, et plus récemment PASTIX et PLASMA

Framework C'est une espace de travail, de composants et de règles. Ces composants sont organisés pour être utilisés en interaction les uns avec les autres. Le développeur ajoute son code au framework et bénéficie un ensemble cohérent de composants de base. Dans ce cas, c'est le framework qui gère le déroulement de calcul. Par exemple Mathlab permet l'importation de code de différents langages sous réserve que ce dernier respecte un format précis.

Les critères de choix sont principalement :

— Le type de matrice et de système dont on a besoin (creuses denses, etc)
— Architectures visées (GPU, accélération matérielle, ...)
— Adéquation de la représentation des matrices dans le code par rapport à la librairie

De nombreuses librairies existent également pour l'écriture efficace de données, telles :

— MPI-I/O
— HDF (à utiliser dans le projet)
— SIONlib (allemand)
— ADIOS
— NetCDF

1.4 Libraires pour la gestion de maillage

Cf sildes.
Visualisation de résultats :

— VTK

Paramétrage des codes :

— INI
— JSON
— XML

1.4.1 Plateforme

Un framework apporte :

— Un ensemble de composant d'intérêt général (par exemple des librairies)
— Un ensemble de règles (normalisation des données, ensemble minimal de fonctionnalités, etc)
— Une interface utilisateur (graphique ou langage de commande) afin d'utiliser les composants

Il faut donc

— Vérifier que le code propose toutes les fonctionnalités nécessaires
— Transforme les données en utilisant le format spécifié
— N'a pas de programme principal (main)

Exemple :
— ROOT, développé par le CERN, utilisé pour le traitement de grande quantité de données
— Trilinos, pour le pré-processing de calcul, utilise le parallélisme, codé en C++
— Salome (CEA-EDF), propose des composants de pré- et post-processing (CAO, maillage, visualisation), une UI graphique et textuelle (python)

2 Outils Utiles

- SSH
- Commande "module"
- Git
- Compilation et link
- Make et cmake
- MPI et mpirun
- Batch scheduler

Pour le 26/09/2017 : Envoyer les connaissances sur le concept au début du cours ainsi que les connaissances acquises.

SSH Pour "secure shell", permet d'accéder à distance à une machine. Par exemple, on peut lancer la commande `ssh poincare` sur les machines de la maison de la simulation. Attention, on est dans la machine distance *uniquement dans le terminal* !

Commande `module` Elle gère l'environnement des machines parallèles. Notamment, elle configure les exécutables, les headers, les librairies statiques et dynamique, la documentation, etc. Elle rend disponible ces éléments *seulement dans le shell*, tout comme `ssh`.

Les différentes commandes sont :

- `module av` (pour *available*), qui affiche les modules disponibles
- `module li` (pour *list*), qui affiche les modules actuellement chargés
- `module load ${module}` pour charger un module (ex : `module load intel`
- `module unload ${module}` pour décharger un module
- `module purge` qui décharge tous les modules actuellement chargés

Git C'est un *gestionnaire de version délocalisé*. Il permet de garder un historique des versions que l'on manipule, afin de tracer les changement et de collaborer sur un même projet entre plusieurs personnes. *Il s'agit de la première chose à maitriser pour faire du développement.* Les notions de base sont :

- Le répertoire de travail
- Le dépôt avec historique (Repository) :
  — Révisions (commits)
  — Un DAG des commits, ou les noeuds sont soit des commits soit des merges (fusion de commits).

Les principales commandes sont :

— `git clone` Faire une copie locale d'un repository existant

— `git commit -m "message"` sauver l'état courant avec `message` comme description du commit

— `git add ${fichiers}` ajouter les fichiers au prochain commit

— `git pull` fusionner l'état courant avec la version distante (peut se décomposer en `git fetch` pour télécharger les changements et `git merge` pour les fusionner)

— `git push` envoyer ses changements locaux sur la version globale

— `qgit` ou `gitk`, deux outils graphiques pour voir l'historique

— `git status` changements depuis le dernier commit

— `git logs` Historique des changements

Pour plus d'informations, voir `http://eagain.net/articles/git-for-computer-scientists/`

Compilation et édition de lien La *compilation* permet de transformer un fichier source (.c, .f90, ...) en un fichier objet (.o le plus souvent). Par exemple, `cc -c -o test.o test.c`.

L'*édition de lien* permet de rassembler n fichiers objets en un exécutable. Par exemple, `cc -o tsts tst1.o tst2.o tst3.o`.

Ces deux commandes peuvent être combinées, par exemple `cc -o tst tst1.c tsts2.c tst3.c`.

Compilation avec une bibliothèque A la compilation, il faut spécifier l'emplacement du header (.h en C, .mod en fortran) via l'option `-I ${dossier}`, par exemple `cc -I /usr/include/ -c -o tst.o tst.c` et bien indique par un `include` (C) ou `use` (fortran). D'un point de vue purement technique cette étape n'est pas toujours nécessaire (en tout cas après le C99), mais il vaut mieux éviter.

Au linkage, la bibliothèque (.a pour les librairies statiques, .so pour les dynamiques) est référencée en ligne de commande (sans extension, via l'option `-l`). Par exemple `cc -lm -o tst.c tst.o`. Le répertoire de recherche des bibliothèques est spécifié par l'option `-L`. Par exemple `cc -L /usr/lib/ -lm -o tst tst.o`. Dans le cas où la bibliothèque n'est pas trouvée, une *erreur de symbole* sera retournée.

Une bibliothèque est composée par un header et son implémentation. Il en existe deux types

— Statique : en .a, il s'agit d'un simple .zip contenant des .o, qui est ajouté à l'exécutable lors de la compilation

— Dynamique : en .so, elle est chargée en mémoire lors de l'exécution du programme

Les bibliothèques dynamiques utilisée par un exécutable sont référencée par la commande `ldd myexe`. Pour voir la liste des symboles (adresses des fonctions fournies et utilisées, éventuellement provenant d'autres bibliothèques), on peut utiliser la commande `nm -D mylib.so`. Le répertoire de chargement des librairies dynamiques est spécifié par la variable d'environnement `LD_LIBRARY_PATH`.

Makefile Il s'agit d'un fichier effectuant les étapes de compilation uniquement si cela est nécessaire. L'avantage par rapport un fichier bash est que le nombre de commande sera plus faible avec l'extension du nombre de fichier. Cela s'effectue par la commande `make`.

La commande lit le fichier "Makefile", dont la structure est celle ci :

```
1 règle : source1 source2
2 <TAB> commande_to_gen target source1 source2
```

L'outil `CMake` permet de générer automatiquement les bibliothèques et dépendances. Il lit un fichier "CMakeLists.txt" qui utilise un vrai langage de script, par exemple

```
1 find_package(MPI)
2 add_executable(myexe source1.c source2.c)
3 target_link_library(myexe mpi)
```

Batch Scheduler (ordonnanceur en français) Un *cluster* est constitué d'une machine pour U utilisateurs. Sur un noeud de calcul, on utilise OpenMP pour communiquer, et d'autres standards pour communiquer entre nœuds, le plus souvent MPI.

Sur un machine perso, on a un seul utilisateur et plusieurs programme, les ressources de travail étant partagées dans le temps (*multiplexage temporel*, sous Linux, l'OS change de programme tous les 50 μs). Cela n'est pas acceptable pour un supercalculateur : les processeurs sont affectés à une seule tâche via un *scheduler*.

Le "batch" de pointcare est `LoadLeveler`, qui accepte plusieurs commandes :

- `llsubmit` pour créer un job
- `llcancel` pour annuler un job
- `llq` pour voir les jobs en cours
- `llinffo.py` pour voir des informations sur l'état des nœuds

Le job est décrit dans un fichier bash dont des commentaires spéciaux décrivent certains paramètres du job :

```
1  #!/bin/bash
2  #@ class = clallmds
3  #@ job_name = RUN_01
4  #@ total_tasks = 32
5  #@ node =
6  #@ environement = COPY_ALL
7  #@ wall_clock_limit = temps maximal de gestion des tâches ;      /* (max 20:00:00 (20h)) */
8  #@ output = $(job_name).$(jobid).log
9  #@ error = $(job_name).$(jobid).log
10 #@ job_type = mpich
11 #@ queue ;                                                        /* valide le job */
   /* Contenu standard du script, exécuté sur le premier nœud                          */
   /* La variable ${LOADL_TOTAL_TASK} contient le nombre total de tâches               */
```

Il faut bien entendu penser à la documentation : `google.fr` et les pages `man` de Linux !

3 Projet

Déroulement

1. Prise en main du code et premiers commits :
   — Génération des fichiers HDF5
2. Développement de quelque post-processing
   — Gradient
   — Dérivée temporelle
   — Génération d'image
3. Rendu 1ère partie
   — 3 séances de couplage
   — Rendu 2ème partie + soutenance

Deux premières séances :

— Prise en main du code
— Génération de fichier HDF5
— Outils de post-process

- HDF5 → température moyenne
- HDF5 → gradient → HDF5
- HDF5 → dérivé temporelle → HDF5
- HDF5 → image (png)

Trois séances suivantes

- Couplage via HDF5 (déjà fait)
- Couplage via même processus
  - Appel de fonction
  - Ecriture sur disque
- Couplage via MPI
  - Processus dédiés
  - Ecriture depuis les processus dédiés uniquement
- Objectifs
  - Maximiser la réutilisation de code
  - Penser maintenabilité et Génie logiciel.

Evaluation

- Code Rendu (1ère et 2ème partie)
- Soutenance
- Le projet compte pour le moitié de la note finale