Récapitulatif

Matthieu Boileau

Anne Cadiou

Hadrien Graland

Matthieu Haefele

Bastien Di Pierro

Optimiser un code

D'abord définir ce que l'on cherche à optimiser !

- temps de calcul
- ressources matérielles (occupation mémoire, scalabilité, ...)
- lisibilité du code
- portabilité du code
- empreinte ecologique

... et trouver le compromis entre tous !

Optimiser un code

Ensuite on choisit en fonction des besoins :

- **architecture :** (potentiellement) variable au cours du temps
- **langage :** choix du developpeur
- **algorithme :** évolutif, mais contrait par le problème à résoudre
- **ressources logicielles :** bibliothèques, API, structures des données, ...

... pas si facile a priori !

Langage

- Langage compilé :
  - plus proche de la machine donc nativement plus performant
  - demande plus de travail au développeur
  - les dialogues avec le compilateur permettent d'optimiser les points clefs (dans le source *et* à la compilation)
- Langage interprété :
  - plus simple à implémenter et à moduler
  - surcoût machine du à l'interprétation : toujours loin des performances de la machine
- JIT :
  - le compromis entre simplicité et performance
  - idéal lorsque le CPU n'est pas limitant, et latence peu importante

Un langage c'est aussi

- Une gestion de la mémoire
- Une gestion des entrées/sorties

  => forte influence sur les performances en fonction des besoins !
- Une communauté : ensemble de pratiques, standards, ...
- Un accès aux bibliothèques

  => l'optimisation d'une portion de code est probablement déjà faite !

Langage : possibilité d'interfacer !

- Permet d'utiliser le "meilleur" de chaque langage
- Ex : Numpy
  - simplicité/lisibilité du python
  - calcul gourmand délégués à BLAS optimisé (C)
- A un surcoût
  - conversion des données
  - allocation mémoire
  - gestion des runtimes
- Le choix doit être pensé/développé **à l'avance** !

Architectures

Optimiser :

- exploiter au maximum les ressources informatiques
- connaître les possibilités/limitations de **chaque organe** !

  => Eviter le déséquilibre (au sens large) de la machine !

Architectures : côté CPU

- Forte croissance des capacités de calculs :
  - Fréquence horloge
  - Pipelining
  - Vectorisation
  - ...

    => Mais ce n'est pas suffisant ! Rapidement, il faut exploiter le parallélisme !
- Augmentation du nombre de coeurs : parallélisme partagé
- Augmentation du nombre de noeuds : parallélisme distribué

  => Plusieurs niveaux : Communication explicite (MPI), accélérateurs, bibliothèques optimisées ...

Architecture : côté mémoire

- Beaucoup plus "lente" que les CPU !

  => Beaucoup de problèmes classiques sont limités par le "Memory bound".

- Connaître la hierarchie de la mémoire et les ordre de grandeurs (taille, latence, débit)

  registre > cache > RAM > disques

- Autant que possible :
  - travailler au plus près du processeurs (gestion de la localité des données)
  - rendre les entrées/sorties asynchrones

Choix de l'architecture

Compromis à trouver entre :

- nombre de coeurs nécessaires
- mémoire totale nécessaire
- topologie matérielle (cartes, GPU, multi-threads, cluster, ...)
- localité des données (ROM, reseau ...)
- paradigme de programmation
- intensité arithmétique, scalabilité

Fait on les bons choix ?

Quelques questions :

- Exploite t'on au mieux les ressources ?
- Le code est il aussi rapide qu'on le pense ?
- Aussi portable ? Scalable ?
- Où sont les goulots d'étranglements ?

Pour savoir, il faut :

- établir des benchmarks
- profiler le code

Benchmarks

Les benchmarks doivent être :

- fréquents
- reproductibles
- représentatifs
- faciles à comparer
- validant des résultats
- aussi fin (micro) / complet (macro) que possible
- écrit dès le developpement !

Profiling

Analyse du code :

- mesure de temps, volumes mémoires, ...
- identifier les points les plus consommateurs/critiques
- suivre les communications

  => permet d'identifier les points bloquants d'un code

Deux types de mesures :

- intrusive (dans le code source, utilisation de bibliothèques)
- extérieure (outils logiciels)

Quelques outils logiciels fréquents (libres)

GNU :

- time
- ps
- free
- gprof

Valgrind :

- cachegrind
- callgrind

Et d'autres :

- Scalasca
- Tau
- perf

Comment optimiser son code

L'algorithme générique

1. Choix du langage, architecture, algo ...
2. niveau = 0
3. Tant que (performances voulues non atteintes) :
    1. Ecrire les benchmarks et profiler
    2. Identifier les points bloquants
    3. test(niveau):
        - == 0 : changer les options de compilation
        - == 1 : repenser le code (vectorisation, ordonnancement, algo, structure des données, gestion mémoire ...)
        - == 2 : utiliser des bibliothèques
        - $>$ 2 : paralléliser
    4. niveau += 1

Pour aller plus loin

Les formations à venir :

- Vectorisation :
  - http://www.idris.fr/formations/simd/fiche-simd.html
- Parallélisme openMP :
  - http://www.idris.fr/formations/openmp/fiche-openmp.html
- Parallélisme MPI :
  - http://www.idris.fr/formations/mpi/fiche-mpi.html
  - https://events.prace-ri.eu/event/864/
- Optimisation :
  - https://events.prace-ri.eu/event/799/
  - https://events.prace-ri.eu/event/881/

Optimalement vôtre !