Langages et performances

Hadrien Grasland

Introduction

Interface programmeur-machine, les langages contribuent aux performances :

- **Contrôle :** Opérations machine exprimables
- **Ergonomie :** Optimisations automatiques possibles
- **Opinion :** Styles de programmation (dé)favorisés

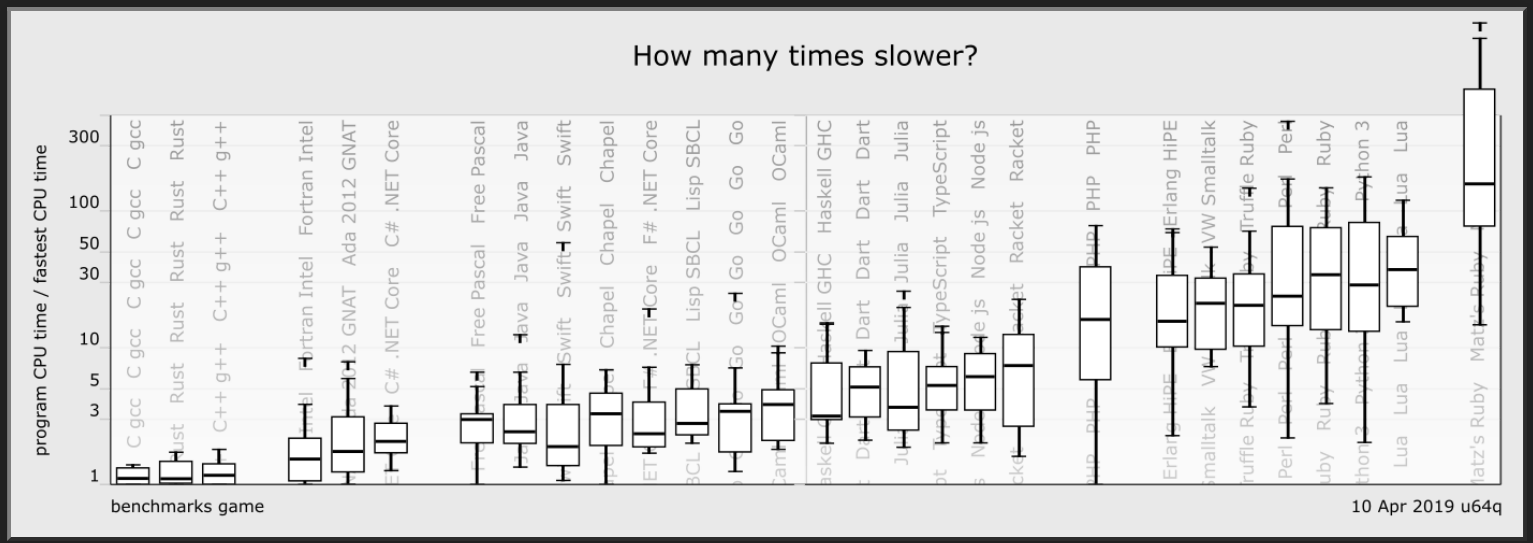
L'implémentation joue un rôle central, mais...

- Elle est **très contrainte** par le langage et sa communauté
- La plupart des langages en ont peu, **très similaires**

Je distinguerai donc peu langage et implémentation.

Haut ou bas niveau ?

Prenons un célèbre meta-benchmark :



Que remarque-t'on ?

- Proches voisins : C et Rust, Fortran et C#, Java et Pascal
  - Plus complexe qu'un choix abstraction / performances !
- La performance peut varier beaucoup selon la tâche
  - Comportement imprévisible... ou spécialisation ?

Au programme

Nous aborderons les sujets suivants :

- Compilation et exécution
- Gestion mémoire
- Entrées / sorties
- Communication avec le compilateur
- Communauté autour du langage
- Interfaces entre langages

Objectifs :

- Comprendre les **compromis** de conception en jeu
- Avoir des **critères** pour faire son choix

Mode d'emploi

Cette présentation est extensible :

- Par défaut, je m'en tiens à l'essentiel
- A chaque □, j'ai préparé des compléments

N'hésitez pas à me demander d'approfondir !

Compilation et exécution

Du code à la machine

Tôt où tard, il faut passer du code aux instructions CPU.

Ce processus varie beaucoup entre langages :

- Au plus simple, **lire le code** en exécutant des **routines prédéfinies**
- Au plus complexe, **traduire le code** une ou plusieurs fois
- On peut traduire **avant** et/ou **pendant** l'exécution

Pourquoi cette diversité ? Il y a des compromis !

Interprétation "pure"

Un interpréteur lit du code en exécutant des opérations prédéfinies.

Avantages :

- Programme directement exécutable
- Implémentation (plus) simple
- Facile de **manipuler le code** à l'exécution

Inconvénients :

- L'utilisateur doit avoir un interpréteur compatible
- L'interpréteur travaille pour **déchiffrer** chaque ligne
  - ...donc les instructions simples sont loin de l'**efficacité machine**

Exemples : **bash**, **PowerShell**, **Octave**

Compilation "à froid" (AoT)

On peut aussi traduire le code en langage machine avant exécution.

Avantages :

- Facile d'utiliser le CPU **efficacement**
- Binaire *potentiellement* fonctionnel sans installations

Inconvénients :

- Processus intermédiaire de compilation, parfois **lourd**
- Difficile de manipuler le code à l'exécution
- Difficile de cibler de **nombreux systèmes**

Exemples : **C/++**, **Fortran**, **Rust**

Alors, compilé ou interprété ?

Aujourd'hui, on combine souvent les deux approches :

- **Julia :** Compilation juste avant exécution (JIT)
- **Javascript :** JIT rapide puis optimisations asynchrones
- **Java :** Traduction en *bytecode* "à froid", interprété + compilé "à chaud"

Un langage peut aussi, comme Python, avoir plusieurs implémentations :

- **CPython** traduit en *bytecode* simple, puis interprète
- **PyPy** remplace CPython par un JIT
- **Numba** permet de JITter des fonctions *depuis CPython*
- **Cython** & **Pythran** compilent un sous-ensemble de Python "à froid"

Mais en général, il y a une implémentation "maître".

Le JIT, solution miracle ?

La compilation JIT marie de bonnes idées d'interprétation & compilation AoT :

- Code d'entrée peu dépendant de la machine cible
- Manipulation du code à la volée ~facile
- Production de code machine optimisé

Mais elle a ses problèmes :

- L'utilisateur doit avoir un JIT compatible
- **Complexité** à l'exécution -> bugs, failles de sécurité, empreinte RAM...
- Performance **fluctuante**, donc difficile à analyser
- Compromis entre latence / coût CPU et optimisation

Conclusions

Le mécanisme d'exécution est important quand :

- Les performances **CPU** sont limitantes
- On ne peut pas **déléguer** à du code compilé

Dans ces cas, privilégier la compilation "à froid" (AoT) :

- Le compilateur peut optimiser à fond
- On doit moins "choisir son code" dans les points chauds
- Les performances sont plus faciles à **prédire et analyser**

Les JITs, plus flexibles, fonctionnent bien quand...

- La latence n'est pas importante (ex: "batch")
- On a de la marge côté CPU (typiquement ~3x)

Gestion mémoire

Un vieux problème

Vu de loin, l'allocation mémoire paraît simple :

- Il y a une réserve de RAM à partager
- On la tronçonne en blocs au fil des demandes
- Les blocs libérés sont remis à disposition

Pourtant, là aussi, il y a mille choix / compromis...

Allocations statiques

Compilateur & OS font de la place aux variables **globales** / statiques.

Avantages :

- Allocation très rapide
- Complètement automatique

Inconvénients :

- Difficile de suivre les **modifications**
- Difficile de contrôler l'**initialisation/finalisation**
- Difficile à concilier avec **parallélisme** et concurrence

Mieux vaut les réserver aux données **constantes** !

Gestion par pile

L'OS donne à chaque *thread* un bloc de mémoire organisé en pile.

Avantages :

- Tout aussi rapide et automatique
- Bonne gestion du parallélisme

Inconvénients :

- **Espace limité** (selon réglages OS, souvent kB~MB)
- Peu adapté à certains usages (ex : partages entre *threads*)
- Certains langages n'offrent pas d'**accès direct**

Gestion par tas

L'OS offre aussi une API d'allocation / libération désordonnée.

Avantages :

- Pas de limite (hors RAM disponible)
- Adapté à tous les usages (y compris *threads*, `mmap`, IPC...)

Inconvénients :

- **Coût** non négligeable (gestion, appels OS, synchronisation...)
- Les **petits blocs** posent problème (fragmentation, localité...)
- *Quelqu'un* doit décider quand **libérer** les données

Gérer la libération

Beaucoup de choses peuvent mal tourner à la libération :

- Il peut rester des pointeurs vers le bloc
- Le bloc peut déjà avoir été libéré
- Si on tarde ou oublie, l'empreinte RAM grimpe

Qui décide quand on libère une allocation tas ?

- Les **programmeurs** en C/++, Fortran, Rust, Ada et Pascal
- Le *runtime* (**ramasse-miettes**) dans la plupart des autres langages

Suivi par les programmeurs

Avantages :

- **Contrôle** fin des performances
- **Adaptable** aux besoins de l'application
- **Nécessaire** à bas niveau et aux interfaces entre langages

Mais difficile sans outils :

- **Collections** prédéfinies dans la bibliothèque standard
- Suivi du **cycle de vie** des objets (destructeurs, opérateur de copie...)
- Analyse **statique** des erreurs mémoire (en Rust, Ada/SPARK...)
- Support des analyseurs **dynamiques** (`memcheck`, ASan...)

Suivi par le *runtime*

Avantages :

- Simplifie les **partages** complexes
- Davantage d'**optimisations** automatiques

Mais pour être efficace, un ramasse-miettes doit aussi...

- Avoir un **fonctionnement complexe**, difficile à contrôler
- Compliquer l'**interaction** avec d'autres systèmes / langages
- Causer des **pics d'usage CPU**, voire bloquer tout par intermittence
- **Retarder** la libération des ressources système
- **Interdire** certaines actions dans le code

En conclusion

La pile est plus performante et prévisible , mais...

- Sa capacité de **stockage** est limitée
- Certains langages ne l'**exposent pas**
- Ceux qui l'exposent n'aident pas toujours à l'utiliser

Toutes les méthodes de gestion du tas ont leurs problèmes

- GC : stochastique, compromis latence/parallélisme/RAM/CPU, "égoïste"
- Compilateur : plus efficace, pas toujours assez flexible
- Bibliothèques : duplication d'effort, peu d'optimisations globales

Gestion des entrées / sorties

Une ressource mal-aimée

Quand on veut optimiser, on se focalise...

- Toujours sur l'utilisation CPU
- Parfois sur la consommation de RAM
- Rarement sur les entrées/sorties (E/S)

Pourtant, elles sont un goulot d'étranglement courant !

L'OS et le code font le gros du travail, mais le langage...

- Détermine quelles **interfaces OS** sont utilisables
- Aide le programmeur à gérer la **latence**

Interfaces d'E/S de l'OS

Il y a une vie après `read`/`write` et `send`/`receive` :

- Accès **aléatoire et parallèle** aux fichiers
- Minimisation des **copies** (ex : "mappage" de tampons)
- Interrogation **simultanée** de N connexions réseau
- Mille *hints* pour dire à l'OS ce qu'on **veut faire**

Un langage peut exposer cela... ou le cacher pour :

- Maximiser la **portabilité** des programmmes
- **Sécuriser** finement les accès au système
- Simplifier l'**implémentation**

Cela réduit les possibilités d'optimisation...

Gestion de la latence

Les latences E/S sont gigantesques vues du CPU :

- Accès mémoire ~ns << Accès disque ~ms << Accès internet ~100ms
- Certaines opérations (ex : entrées utilisateur) durent **indéfiniment**

Quand le programme a autre chose à faire, il peut...

- Lancer l'opération E/S
- Faire du travail CPU en attendant
- Se synchroniser avec l'opération ultérieurement

On parle de traitements **asynchrones**.

Un travail pour l'OS ?

L'OS fait un peu d'asynchrone "sous le capot", mais...

- C'est **limité** (dépend du périphérique et motif d'accès)
- C'est une source incessante de **bugs** (ex : `fflush` oublié)
- On a peu de **contrôle** sur ce qui se passe

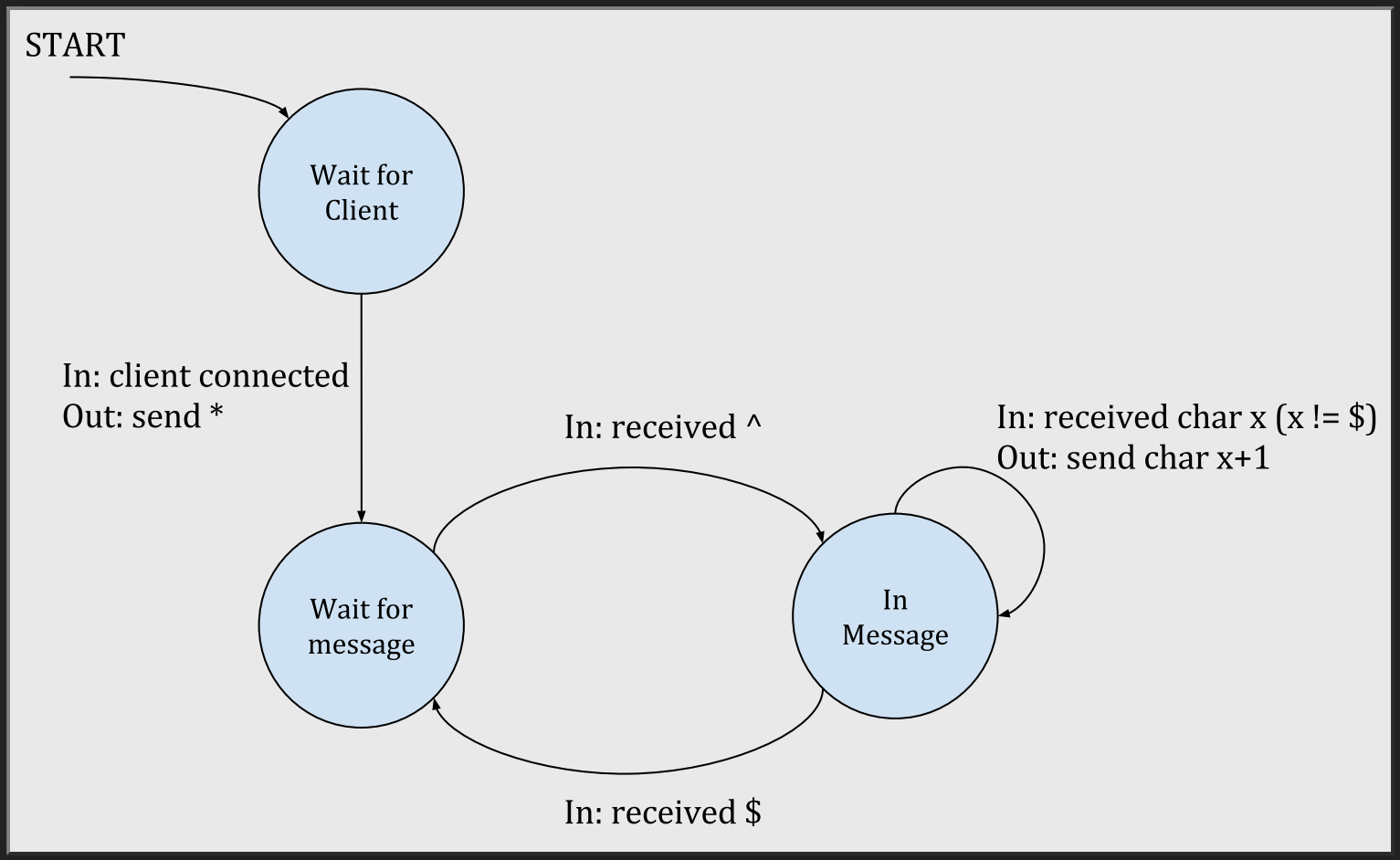
Pour maximiser les performances E/S, privilégier l'asynchronisme **explicite** :

- Groupe de *threads* dédiés aux E/S
- Gestionnaires d'évènements (`select`/`epoll`, `kqueue`/`IOCP`...)

Le langage peut faire beaucoup pour améliorer son ergonomie !

Exemple

Soit un protocole réseau simple, dont on veut une implémentation asynchrone :



Implémentation C

On peut utiliser l'API Linux `epoll` sans aide du langage...

```
// Asynchronous socket server - accepting multiple clients concurrently,
// multiplexing the connections with epoll.
//
// Eli Bendersky [http://eli.thegreenplace.net]
// This code is in the public domain.
#include <assert.h>
#include <errno.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/epoll.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

#include "utils.h"

#define MAXFDS 16 * 1024

typedef enum { INITIAL_ACK, WAIT_FOR_MSG, IN_MSG } ProcessingState;

#define SENDBUF_SIZE 1024
```

Implémentation Python

...ou bien le langage peut aider :

```
import asyncio

async def process(reader, writer):
    writer.write(b'*')
    try:
        while True:
            data = await reader.readexactly(1)
            if data != b'^':
                continue
            data = await reader.readexactly(1)
            while data != b'$':
                writer.write(bytes([data[0] + 1]))
                data = await reader.readexactly(1)
    except asyncio.IncompleteReadError:
        pass

loop = asyncio.get_event_loop()
listener = asyncio.start_server(process, host=None, port=9090)
loop.run_until_complete(listener)
loop.run_forever()
```

Conclusions

Pourquoi se soucier des E/S ?

- Elles sont souvent limitantes pour les performances
- Le système d'exploitation ne peut pas tout faire seul

Comment les langages peuvent-ils aider ?

- **Exposer** les fonctionnalités d'E/S de chaque OS
- Fournir des **abstractions portables** de fonctions répandues
- Simplifier l'écriture de code **asynchrone**

Sémantique et optimisations

Motivation

Voici une petite routine Fortran :

```
subroutine normes(a_vec, b_vec, s_vec)
    integer, parameter :: VEC_SIZE = 16
    real, dimension(VEC_SIZE), intent(in) :: a_vec, b_vec
    real, dimension(VEC_SIZE), intent(out) :: s_vec
    integer :: i

    do i = 1, VEC_SIZE
        s_vec(i) = 1. / sqrt(a_vec(i)*a_vec(i) + b_vec(i)*b_vec(i))
    end do
end subroutine normes
```

Essayons de la traduire littéralement en C :

```
#include "math.h"

#define VEC_SIZE 16

void normes(const float* a_vec, const float* b_vec, float* s_vec) {
    for (size_t i = 0; i < VEC_SIZE; ++i) {
        s_vec[i] = 1. / sqrt(a_vec[i]*a_vec[i] + b_vec[i]*b_vec[i]);
    }
}
```

Performances

Microbenchmark des deux fonctions sur l'habituel Xeon E5-1620 v3...

- `gfortran` v8.3.1, flags `-O3 -march=native` : 16.4 ns
- `gcc` v8.3.1, flags `-O3 -march=native` : 72.3 ns -> **4.4x plus lent**

Ce qui a changé ? La **sémantique** du langage C :

- `sqrt()` gère les erreurs via `errno` -> Pas de vectorisation !
- `1.` et `sqrt()` sont en double précision -> Conversions, calculs plus lents

Avec `-fno-math-errno`, et en utilisant `1.f` et `sqrtf`, le code C ira aussi vite.

On voit l'intérêt d'un langage conçu pour le calcul...

Généralisons

L'optimisation automatique dépend du dialogue programmeur / compilateur :

- Le programmeur doit **maîtriser les règles** de son langage
- Le compilateur doit **en savoir un maximum** sur le problème

Un langage facilite ce dialogue quand :

- Le code naïf / idiomatique est **efficace**
- Perdre en efficacité est un choix **conscient**
- Le compilateur sait beaucoup de choses **par défaut**
- Il est facile de lui en **dire plus** dans les "points chauds"

A défaut, le programmeur doit passer du temps à analyser...

Informer le compilateur

Il y a plusieurs moyens d'informer le compilateur :

- **Constantes** de compilation
- Données **statiquement typées**
- Annotations standard du langage (ex : `align`, `inline`, `restrict`...)
- Annotations du compilateur (ex : OpenMP, *hot/cold*, *likely/unlikely*...)
- Configuration du compilateur (ex : `-march=native`, `-flto`, `-fno-math-errno`)

Souvent, on fait un compromis avec la flexibilité, voire la portabilité.

Il faut **mesurer l'effet** et savoir s'arrêter.

Le prix de la flexibilité

Certaines techniques diminuent l'info à la compilation :

- Configuration **à l'exécution**
- Compilation séparée
- **Polymorphisme** et typage dynamiques
- Génération / modification de code à la volée
- Pointeurs de fonction
- Evaluation paresseuse

Le code machine devient moins spécialisé, donc moins efficace.

A utiliser prudemment dans les "points chauds" !

Conclusions

Rôle du langage dans l'optimisation automatique :

- Garantir que **code intuitif = code efficace**
- Expliciter les **choix coûteux** / inefficaces
- Donner beaucoup d'**infos au compilateur** par défaut
- Permettre au programmeur d'en donner plus

Responsabilité du programmeur :

- Choisir un langage qui rend la performance **ergonomique**
- Bien connaître la **sémantique** de son langage
- **Informer** le compilateur dans les "points chauds"

Communauté autour du langage

Importance de la communauté

La communauté autour d'un langage va déterminer...

- La notion de code idiomatique (= bien accepté)
- Les priorités de développement du langage et de l'implémentation
- L'ensemble des bibliothèques disponibles

Tous ces points ont un impact sur les performances.

Code idiomatique

Attention au code efficace, mais **exotique** / illisible :

- Code sans héritage ni allocations mémoires en Java / C#
- Gros volume d'assembleur et intrinsèques en C/++
- Code complètement vectorisé en MATLAB
- Métaprogrammation en Fortran
- ...et bien d'autres violations d'idiomes "locaux"

Faute de mainteneurs, ce code sera souvent **réécrit** en style idiomatique...

Priorités de développement

Concevoir un langage = faire des **compromis** :

- Généraliste ou spécialisé ?
- Flexible ou cohérent ?
- Simple à apprendre ou utiliser ?
- Simple à écrire ou lire ?
- Simple à utiliser ou implémenter ?
- Optimisé pour le séquentiel ou le parallèle ?

La communauté influencera largement les choix.

Bibliothèques

On va plus vite avec de l'aide des autres :

- Algèbre linéaire et analyse numérique
- Frameworks et technologies web
- Interfaces graphiques
- Pilotes matériel spécialisés
- `<Votre domaine métier ici>`

Avantage aux langages classiques de votre **domaine** !

Conclusion

Utiliser un langage (de façon) exotique, c'est...

- **Recréer** des codes disponibles ailleurs
- S'exposer à l'**opposition** de ses pairs
- Passer du temps à **former** des collaborateurs

Il faut des arguments solides pour justifier ce coût.

Interfaces entre langages

Motivation

Interfacer plusieurs langages a de nombreux avantages :

- Bénéficier du meilleur de N mondes
- Eviter les querelles sur le langage unique
- Résoudre les schismes sociaux (ex : "scientifiques" vs "ingénieurs")
- Toucher un public plus large

But courant : combiner **infrastructure performante** et **interface flexible**.

...mais ce n'est pas si simple !

Coût en performance

A une interface entre langages, on doit souvent...

- **Convertir les données** d'un format à un autre
- Faire communiquer deux systèmes de **gestion mémoire**
- Se retreindre au **vocabulaire d'API** limité du C
- Quitter le *runtime* du langage hôte (ex : relâcher un *mutex*)
- Activer le *runtime* du langage cible (ex : gérer les exceptions)
- Faire un appel de fonction non *inliné*

Il faut **amortir** ces coûts pour rester efficace.

Coût en complexité

L'inter-langages complique de nombreux aspects du code...

- **Processus** de compilation / exécution
- Structures de **données**, gestion mémoire
- **Surface d'API** à maintenir / exposer
- Gestion des **erreurs**
- Packaging et distribution du logiciel
- Déboguage et analyse de performances

...et peu de développeurs maîtrisent ces interfaces !

En résumé

Interfacer plusieurs langages est coûteux :

- Ne le faites jamais dans un espoir de facilité
- Soyez conscients des coûts & bénéfices
- Exigez une motivation solide

Si vous devez le faire :

- Prévoyez-le dès la **conception**
- **Amortissez** vos coûts d'interface

Pour conclure

Récapitulatif

Les langages influent sur la performance de nombreuses manières :

- Exécution (maîtrise de la **latence**, **efficacité** du code machine)
- Gestion des ressources (**surcoûts** de gestion, **consommation** de RAM)
- Entrées / sorties (accès aux **APIs** système, support de l'**asynchronisme**)
- **Culture** de la performance dans la communauté

Un bon langage simplifie l'optimisation :

- Il **maximise** l'optimisation automatique, **facilite** l'intervention manuelle
- Ses compromis de conception sont adaptés au projet

Interfacer plusieurs langages n'est pas une solution, c'est un **compromis** de plus !

Perspectives

Les évolutions matérielles mettent la pression sur le codes :

- Performance séquentielle stagnante ? Il faut **paralléliser** !
- Pression croissante sur la RAM ? Pas d'impasse sur le **multi-threading** !
- Poids des E/S croissant (réseau, coprocesseurs) ? Il faut être **asynchrone** !
- Hétérogénéité croissante ? Besoin de "**portabilité** des perfs" !

Certains choix d'hier deviennent indéfendables :

- Gestion mémoire manuelle sans outils
- Ramasse-miettes qui bloque l'application
- Usage intensif de variables globales / singletons
- Interpréteur incapable d'exécuter deux *threads* à la fois

Les langages peuvent nous **aider** face au futur... ou nous **piéger** dans le passé !

Des questions ?

Aller plus loin

Quelques sujets dont on n'a pas parlé aujourd'hui :

- Ressources de calcul "hétérogènes" : GPUs, FPGAs...
  - Solutions à code séparé : OpenGL, Vulkan, OpenCL...
  - Solutions à code commun : CUDA, SyCL, Kokkos...
- Données et calcul distribué
  - "Coarrays" de Fortran 2008, HPX, Apache Spark...
- Code spécifique à un matériel (assembleur, *intrinsics*...)
  - Privilégier les bibliothèques : xsimd, ATLAS, ffmpeg, fftw...
- Optimisations de performances pour JIT et GCs
  - Progrès d'infrastructure : Android RT, Unity Burst, GraalVM...
  - Réglages exposés aux devs (ex : 5 GCs différents en Java !)

Complément: Les compromis du JIT

Exemple

Voici un petit calcul arithmétique :

```
uint64_t somme_modulo(uint32_t n) {
    uint64_t res = 0;
    for (uint32_t i = 0; i < n; ++i) {
        if (i % 5 == 3) {
            res += i;
        }
    }
    return res;
}
```

On va comparer ses coûts en C++, C#, et Julia :

- $n = 2^{32}$ itérations sur un Xeon E5-1620 v3 @ 3.50GHz
- Compilateur C++ : clang++ v7.0.1, flags `-O3 -march=native`
- Transpileur C# -> CIL : csc v2.8.2.62916, flags `/optimize+ /platform:x64`
- Interpréteur / JIT CIL : mono v5.20.1.19, env. `MONO_THREADS_SUSPEND=preemptive`
- Julia v1.0.3 (JIT basé sur LLVM, le backend de clang)

Mesurons...

Coûts de compilation fichier à fichier :

Tâche	Temps CPU	Latence	Remarques
C++ -> natif	394ms	394ms	
C# -> CIL	581ms	581ms	
CIL -> natif	25ms	25ms	Limité par les E/S
Julia -> natif	322ms	141ms	Compilation parallèle

Coûts d'exécution :

Tâche	Temps CPU	...vs C++	RAM RSS max
Binaire C++	3.77s	1x	1.2MB
JIT CIL	15.49s	4.11x	25.4MB
Interpréteur CIL	57.37s	15.22x	19.6MB
JIT Julia	4.25s	1.13x	141.7 MB

Tous les JITs n'optimisent pas les mêmes choses !

Itération de boucle, JIT Mono

Regardons ce que nos compilateurs font d'une itération de boucle...

```
for (uint i = 0; i < n; ++i) {
    if (i % 5 == 3) {
        res += i;
    }
}
```

Le JIT Mono, orienté latence, traduit littéralement. A nous d'optimiser !

```
1030:     mov    $0x5,%ecx
1035:     mov    %r13,%rax
1038:     xor    %rdx,%rdx
103b:     div    %ecx
103d:     cmp    $0x3,%edx
1040:     jne    1048 <SommeModulo_somme_modulo_uint+0x38>
1042:     mov    %r13d,%eax
1045:     add    %rax,%r14
1048:     inc    %r13d
104b:     cmp    %r15d,%r13d
104e:     jb     1030 <SommeModulo_somme_modulo_uint+0x20>
```

Itération de boucle, AoT Clang

Clang et Julia, pas pressés par le temps, optimisent beaucoup plus :

```
401230:   mov    %r10d,%esi
401233:   imul   %r11,%rsi
401237:   shr    $0x22,%rsi
40123b:   lea    (%rsi,%rsi,4),%r15d
40123f:   mov    %r13d,%esi
401242:   imul   %r11,%rsi
401246:   shr    $0x22,%rsi
40124a:   mov    %r12d,%ebx
40124d:   imul   %r11,%rbx
401251:   shr    $0x22,%rbx
401255:   lea    (%rbx,%rbx,4),%ebx
401258:   mov    %edx,%r8d
40125b:   imul   %r11,%r8
40125f:   shr    $0x22,%r8
401263:   lea    (%r8,%r8,4),%edi
401267:   add    $0x3,%edi
40126a:   cmp    %edx,%edi
40126c:   mov    $0x0,%edi
401271:   cmove  %edx,%edi
401274:   add    %rax,%rdi
401277:   lea    0x1(%rdx),%eax
40127a:   add    %r14d,%ebx
40127d:   mov    $0x0,%ebx
```

Comment faire mieux ?

Pour un meilleur compromis latence/optimisation, il y a l'approche Java / JS :

- On **lance l'exécution** avec un JIT rapide ou interpréteur
- On repère des **points chauds** en analysant l'exécution
- On **recompile** ceux-ci avec un JIT plus coûteux
- On **remplace** l'ancien code quand le nouveau est prêt

Quels sont ses inconvénients ?

- Dépend de "points chauds" concentrés
- Ajoute énormément de complexité à l'exécution
- Raisonner sur la performance devient très difficile

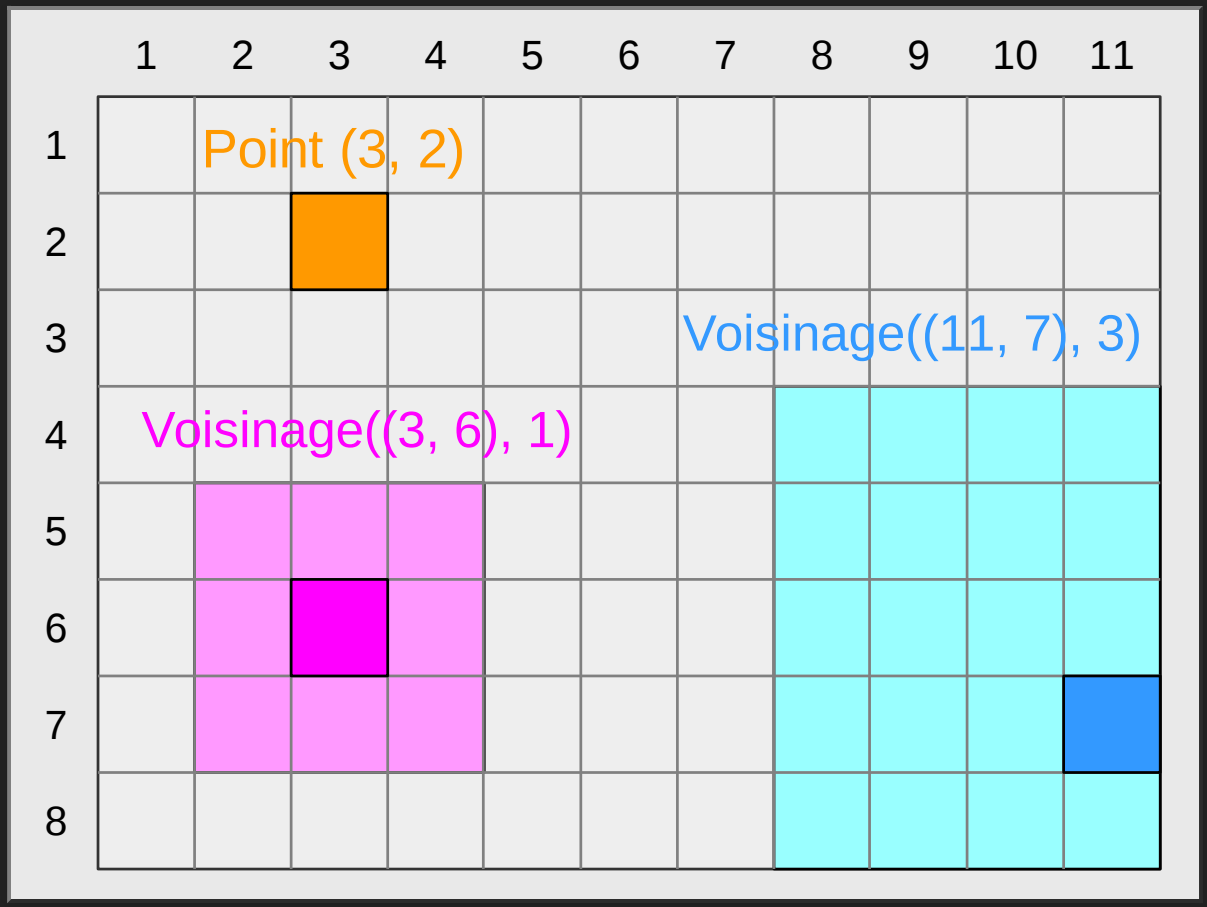
Compromis : Bon premier jet, mais optimum moins accessible

Retour à la partie exécution

Complément: Tas, pile et itérateurs

Exemple

On veut énumérer le voisinage d'un point sur une grille 2D finie.



C'est une liste de points de taille variable. Comment l'exprimer ?

Solution naïve

On commence par un tableau dynamique (`Vec` en Rust) :

```
fn minmax_xy(centre: (isize, isize), distance: isize)
    -> ((isize, isize), (isize, isize))
{
    let min_x = (centre.0 - distance).max(1).min(LARGEUR);
    let max_x = (centre.0 + distance).max(1).min(LARGEUR);
    let min_y = (centre.1 - distance).max(1).min(HAUTEUR);
    let max_y = (centre.1 + distance).max(1).min(HAUTEUR);
    ((min_x, min_y), (max_x, max_y))
}

fn voisinage_naif(centre: (isize, isize), distance: isize)
    -> Vec<(isize, isize)>
{
    let ((min_x, min_y), (max_x, max_y)) = minmax_xy(centre, distance);

    let mut resultat = Vec::new();
    for x in min_x..=max_x {
        for y in min_y..=max_y {
            resultat.push((x, y));
        }
    }
    resultat
}
```

Allocation optimisée

Le code naïf fait plusieurs allocations tas (croissance du tableau).

Si on connaît d'avance la taille, mieux vaut l'annoncer :

```
fn voisinage_reserve(centre: (isize, isize), distance: isize)
    -> Vec<(isize, isize)>
{
    let ((min_x, min_y), (max_x, max_y)) = minmax_xy(centre, distance);

    let size = ((max_x - min_x + 1) * (max_y - min_y + 1)) as usize;
    let mut resultat = Vec::with_capacity(size);

    for x in min_x..=max_x {
        for y in min_y..=max_y {
            resultat.push((x, y));
        }
    }
    resultat
}
```

Allocation optimale

Si l'utilisateur veut bien garder notre tableau, on peut le réutiliser :

```
fn voisinage_cache(centre: (isize, isize),
                   distance: isize,
                   resultat: &mut Vec<(isize, isize)>)
{
    let ((min_x, min_y), (max_x, max_y)) = minmax_xy(centre, distance);

    resultat.clear();

    for x in min_x..=max_x {
        for y in min_y..=max_y {
            resultat.push((x, y));
        }
    }
}
```

...mais ça dépend de l'utilisation, ça complique l'API...

On arrive dans une impasse. Et si le langage nous aidait ?

Utilisation de la pile

L'utilisation du tas est-elle pertinente ici ?

- Les voisinages à distance > 2 sont peu utilisés
- La pile convient pour de petits nombres de points
- Le tas pourrait être réservé aux cas exceptionnels

En fait, on utilise le tas par habitude :

- En C/++, Rust... une fonction doit retourner un **objet de taille connue**
- Quand ce n'est pas le cas, tout est généralement alloué sur le tas

Mais on pourrait, comme GNAT (Ada), utiliser une pile secondaire !

- Facile au niveau langage, difficile au niveau bibliothèque...

Itération

A défaut, en Rust, on peut retourner un itérateur :

```
fn voisinage_iter(centre: (isize, isize), distance: isize)
    -> impl Iterator<Item=(isize, isize)>
{
    let ((min_x, min_y), (max_x, max_y)) = minmax_xy(centre, distance);

    let mut x = min_x;
    let mut y = min_y;
    std::iter::from_fn(move || {
        y += 1;
        if y > max_y {
            y = min_y;
            x += 1;
            if x > max_x {
                return None;
            }
        }
        Some((x, y))
    })
}
```

Aussi efficace qu'une boucle côté utilisateur... mais moins lisible !

Plus simple...

Les coroutines, comme les générateurs Python, rendent les itérateurs plus lisibles :

```
def voisinage_iter(centre, distance):
    ((min_x, min_y), (max_x, max_y)) = minmax_xy(centre, distance)
    for x in range(min_x, max_x+1):
        for y in range(min_y, max_y+1):
            yield (x, y)
```

Cela encourage les Pythonistes à utiliser l'itération.

Existe aussi en Julia, à l'étude en Rust.

...ou plus complexe

Au contraire, C++ complique tant l'itération qu'on l'utilisera rarement :

```
using Point = std::pair<long, long>;

class Voisinage
{
public:
  Voisinage(Point centre, long distance)
    : m_rectangle(minmax_xy(centre, distance))
  {
  }

  class iterator
  {
  public:
    // Constructeur pour le cas end()
    iterator(Point arrivee) : m_actuel(arrivee) {}

    iterator(Point depart, Point arrivee)
      : m_actuel(depart)
      , m_min_y(depart.second)
      , m_max_y(arrivee.second)
    {
    }

    Point operator*() const { return m_actuel; }
```

Enjeu

Micro-benchmark des implémentations présentées vs taille du voisinage :

	Naïf (N allocs)	Optimisé (1 alloc)	Cache (0 allocs)	Itération
1	126ns	50ns	35ns	26ns
2	184ns	60ns	47ns	33ns
3	208ns	74ns	64ns	38ns
4	239ns	100ns	86ns	47ns
5	265ns	136ns	132ns	57ns

Réalisé sur Xeon E5-1620 v3 @ 3.50GHz, avec Rust v1.34.0, en config `--release`.

Quelques remarques :

- Si l'allocateur n'a pas besoin de contacter l'OS, il va assez vite
- Réserver le stockage donne ici un facteur 2-3x à peu de frais
- Une itération ergonomique **et** performante est précieuse !
- Etudiez les performances avant de figer vos APIs

Retour à la partie gestion mémoire

Complément: Langages orientés asynchrone

Approches d'asynchronisme

Aujourd'hui, on crée rarement des *threads* soi-même :

- Passage à l'échelle difficile quand le nombre d'E/S augmente
- Logique complexe (préemption, synchronisation...)

On multiplexe plutôt plusieurs "tâches" coopératives par *thread*...

- En écrivant les tâches comme des *threads* ("green threads")
- ...ou en explicitant les attentes E/S (`async`/`await`)
- En demandant au gestionnaire d'événements de nous rappeler (*callbacks*)
- ...directement ou via un objet représentant un événement (*futures*)

Le mérite relatif de ces approches est fortement débattu !

Rôle du langage

Les tâches de type *green thread* bénéficient d'un support langage :

- Interception des E/S bloquantes de la bibliothèque standard
- Intégration à la gestion de pile du compilateur

Les tâches de type `async`/`await` en bénéficient également :

- Réservation des mots-clés (et autre syntaxe optimisée)
- Transformation du code linéaire en automate à états finis

Et *toutes* les approches bénéficient d'une standardisation :

- Abstraction des mécanismes OS (ex: `epoll` vs `kqueue`/`IOCP`)
- Logique commune (gestion d'erreurs, création/annulation tâche, type *future*...)

Retour à la partie E/S

Complément: Analyser les optimisations

Interroger le compilateur

Certains optimiseurs produisent des logs, qu'on peut consulter :

- Famille GCC -> options `-fopt-info`
- Famille Intel -> options `-qopt-report`

...mais il n'est pas facile de les comprendre :

- Très verbeux -> Isoler le code concerné dans un module
- Jargon & point de vue inhabituel -> Nécessite un apprentissage
- Ignore ce qui est "normal" (ex : conversions implicites du C)

Hélas, savoir lire l'assembleur généré reste utile.

Exemple de log

Sur notre code C, `-fopt-info-missed` aide à repérer la gestion d'erreur `sqrt`...

```
normes.c:9:5: note: not vectorized: control flow in loop.
normes.c:9:5: note: bad loop form.
normes.c:10:25: note: not consecutive access _29 = *_30;
normes.c:10:25: note: not consecutive access _23 = *_24;
normes.c:10:25: note: not vectorized: no grouped stores in basic block.
normes.c:10:25: note: not consecutive access _52 = *_51;
normes.c:10:25: note: not consecutive access _55 = *_54;
normes.c:10:25: note: not consecutive access *_42 = _43;
normes.c:10:25: note: not vectorized: no grouped stores in basic block.
normes.c:10:25: note: not consecutive access _73 = *_72;
normes.c:10:25: note: not consecutive access _76 = *_75;
normes.c:10:25: note: not consecutive access *_63 = _64;
normes.c:10:25: note: not vectorized: no grouped stores in basic block.
normes.c:10:25: note: not consecutive access _94 = *_93;
normes.c:10:25: note: not consecutive access _97 = *_96;
normes.c:10:25: note: not consecutive access *_84 = _85;
normes.c:10:25: note: not vectorized: no grouped stores in basic block.
normes.c:10:25: note: not consecutive access _115 = *_114;
normes.c:10:25: note: not consecutive access _118 = *_117;
normes.c:10:25: note: not consecutive access *_105 = _106;
normes.c:10:25: note: not vectorized: no grouped stores in basic block.
normes.c:10:25: note: not consecutive access _136 = *_135;
normes.c:10:25: note: not consecutive access _139 = *_138;
```

Exemple d'assembleur

...mais il faut examiner l'assembleur pour voir la double précision (suffixes "d") :

```
 0:   lea    0x20(%rsi),%rcx
 4:   lea    0x20(%rdx),%rax
 8:   cmp    %rcx,%rdx
 b:   setae  %r8b
 f:   cmp    %rax,%rsi
12:   setae  %cl
15:   or     %cl,%r8b
18:   je     d0 <normes+0xd0>
1e:   lea    0x20(%rdi),%rcx
22:   cmp    %rcx,%rdx
25:   setae  %cl
28:   cmp    %rax,%rdi
2b:   setae  %al
2e:   or     %al,%cl
30:   je     d0 <normes+0xd0>
36:   vmovups (%rsi),%ymm3
3a:   vmovups (%rdi),%ymm4
3e:   vmovapd 0x0(%rip),%ymm2        # 46 <normes+0x46>
45:   [ hex: 00 ]
46:   vmulps %ymm3,%ymm3,%ymm0
4a:   vfmadd231ps %ymm4,%ymm4,%ymm0
4f:   vcvtps2pd %xmm0,%ymm1
53:   vextractf128 $0x1,%ymm0,%xmm0
```

Retour à la partie sémantique

Complément: Performance aux interfaces

Exemple

Imaginons un gros calcul Python freiné par un produit scalaire naïf :

```
# API: Reçoit deux listes de flottants, retourne un flottant
def scalaire_python(a, b):
    s = 0.
    for i in range(len(a)):
        s += a[i]*b[i]
    return s
```

Pour améliorer les performances, un contributeur propose d'utiliser NumPy :

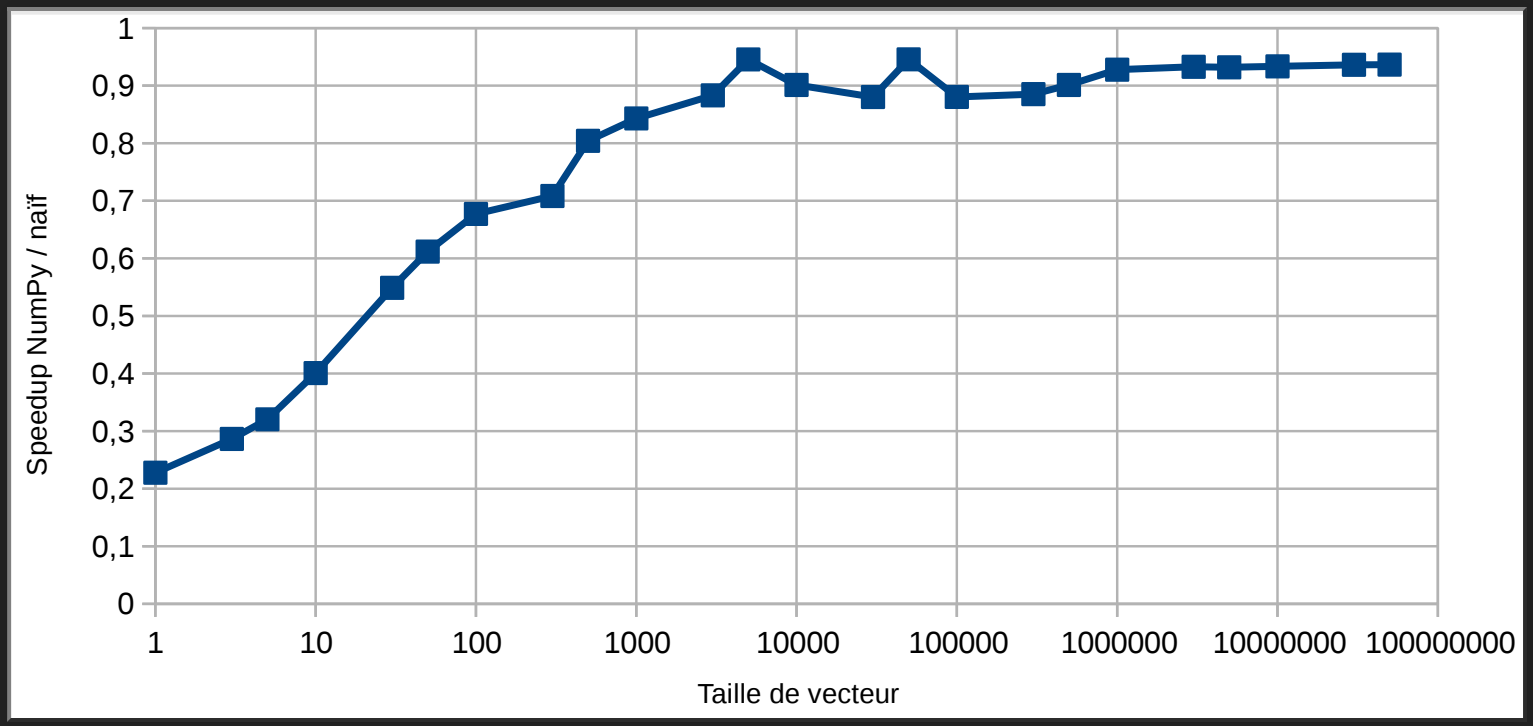
```
import numpy

# Même API que la fonction originale
def scalaire_numpy(a, b):
    a_n = numpy.array(a)
    b_n = numpy.array(b)
    return a_n.dot(b_n)
```

Cela déléguera le calcul à un BLAS très optimisé.

Mesures

Mais si on compare (CPU habituel, Python 3.7, NumPy 1.16.2, libblas v3.8.0)...



...NumPy est battu par le code Python naïf !

- Beaucoup plus lent à petite taille (~2.0µs vs ~450ns en dimension 1)
- Un peu plus lent à grande taille (~110ns/elem vs ~100ns/elem)

Explication

Notre exemple a des coûts d'interface ~affines :

- Coût fixe d'appel aux fonctions NumPy (~1.4µs)
- Surcoût linéaire de conversion listes -> tableaux (~110ns/élément)

Et en comparaison avec le calcul Python...

- Coût fixe >> initialisation code naïf
- Surcoût par élément > itération code naïf

Le coût de calcul BLAS reste donc noyé dans la masse !

Comment faire mieux

On peut éviter le surcoût linéaire en changeant d'API :

```
# API: Reçoit deux vecteurs NumPy de flottants, retourne un flottant
def scalaire_numpy_incompatible(a_n, b_n):
    return a_n.dot(b_n)
```

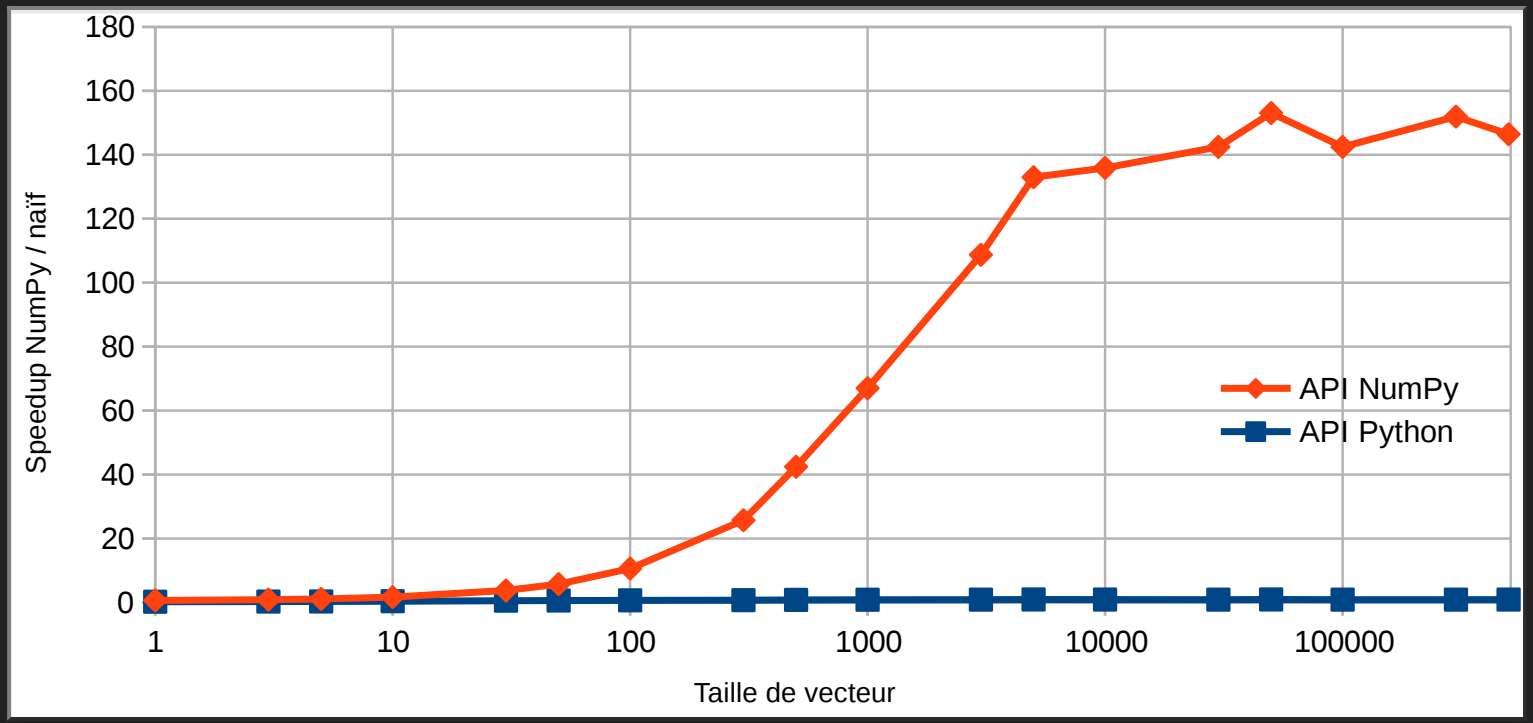
Mais cela peut signifier réécrire beaucoup de code appelant...

- Soit en se restreignant à l'API de NumPy & autres bibliothèques
- Soit en écrivant des extensions Python soi-même

...donc pensez au multi-langage *dès la conception* !

Epilogue

Avec une API adaptée, NumPy tient ses promesses...



...mais notez qu'il faut de grands vecteurs pour amortir le coût d'interface !

Retour à la partie interfaces