Pourquoi Julia est si rapide?

Pour une explication plus technique en anglais, voir ici.

Pourquoi C est-il rapide?

Les langages C, C++ et Fortran sont généralement les plus rapides en termes d'exécution, et sont par conséquent les langages les plus utilisés pour des applications où la performance est critique, en particulier pour les calculs intensifs sur superordinateurs, où C et Fortran règnent en maîtres.

Ces trois langages partagent un certain nombre de caractéristiques dans leur conception qui sont à l'origine de leurs performance. C++ est cependant un cas à part, car beaucoup plus sophistiqué que C ou Fortran.

- **Compilation** Le code source d'un programme est transformé en code machine par un programme appelé *compilateur*, qui fait généralement un certain nombre de transformations pour optimiser le code machine obtenu.
- **Typage statique** Chaque expression a un type bien défini au moment de la compilation. Cette propriété est importante car le compilateur utilise l'information sur les types du programme pour faire ses optimisation.
- **Gestion de la mémoire manuelle** En dehors d'expressions simples, le programmeur doit lui-même gérer l'allocation de mémoire dans son programme. Cela permet d'éviter d'avoir recours à une gestion automatique de la mémoire par un *Garbage Collector*, qui peut constituer un coût assez élevé lors de l'exécution. Il s'agit cependant d'une lame à double tranchant, car gérer manuellement la mémoire est notoirement compliqué et source de bugs et vulnérabilités.

À haut niveau, nous pouvons décrire la stratégie de ces trois langages comme le fait de faire faire un maximum de l'évaluation des programmes par le compilateur.

La contrepartie de ces performances et que ces langages sont souvent compliqués à utiliser et laissent beaucoup d'opportunités au programmeur de commettre des erreurs qu'il ne réalisera que pendant ou après la compilation.

Pourquoi Python et Matlab sont-ils lents?

À la différence de C et Fortran, les langages comme Python, Matlab ou R sont assez lents, mais sont beaucoup plus flexibles et simples d'utilisation, ce qui permet souvent au programmeur de gagner du temps lorsqu'il écrit un programme.

Les caractéristiques de ces langages sont

- **Interprétation** Plutôt que de passer par un compilateur, le code source est lu par un programme appelé *interpréteur* qui évalue le programme ligne par ligne et les exécute au fur et à mesure. Il y a donc moins de potentiel pour optimiser le programme.
- **Typage Dynamique** Le type d'une expression n'est connu que pendant l'exécution, ce qui de nouveau laisse moins d'opportunités d'optimiser le programme.
- **Gestion de la mémoire automatique** Contrairement à C et Fortran, Python et Matlab s'occupent automatiquement d'allouer et libérer la mémoire utilisée par le programme.

Cela se fait entre autre via un programme appelé *Garbage collector*, dont le rôle est de surveiller la mémoire utilisée pendant l'exécution du programme et de libérer les zones de mémoire inutilisées. Le garbage collector tourne en parallèle avec le programme de l'utilisateur, ce qui nécessite des ressources de calcul supplémentaires.

Traditionnellement, utiliser Python ou Matlab malgré la perte de performance est un choix calculé. Les heures que passent le programmeur à écrire un programme coûtent presque universellement plus cher que le temps passé à exécuter ce programme. De plus, l'absence d'étape de compilation permet de rapidement prototyper un programme, l'évaluer et le corriger immédiatement. Cela génère un gain de productivité non négligeable pour le programmeur.

Quand ils ont vraiment besoin de performance, Python et Matlab ont recours à des fonctions qui reposent sur une implémentation dans un langage rapide (C'est la raison pourquoi Python et Matlab encouragent à utiliser des fonctions vectorisées plutôt que des boucles). Cette approche est cependant limitée, et dans les cas extrêmes, la seule solution est de réécrire complètement le programme dans un langage performant.

L'importance des types

Nous avons jusqu'ici parlé de la distinction entre systèmes de types *statiques* et *dynamiques* et laissé entendre que le typage statique était plus rapide que le typage dynamique. La situation est cependant plus nuancée, car d'autres propriétés du système de type sont à prendre en compte.

En particulier, un autre aspect des systèmes de types est qu'il permettent de vérifier en partie si un programme est correct (sans erreurs de syntaxe) et de prouver des propriétés à propos de celui-ci.

Ce dernier point n'est pas une analogie. Pour effectuer ses optimisations, le compilateur doit être capable de prouver que celles-ci ne modifient pas le comportement du programme. Les types du programme servent alors de propositions dans un langage formel qui permet au compilateur de prouver des propriétés du programme. Pour certains langages comme Coq avec un système de type très sophistiqué, le compilateur est d'ailleurs essentiellement un prouveur de théorème.

Un système de types bien conçu est donc important, car il permet au compilateur de

- Prouver plus de propriétés du programme et donc de faire plus d'optimisations
- Prouver/vérifier que le programme est correct, ce qui permet d'éviter des erreurs et vulnérabilités. De plus, lorsqu'une erreur a lieu, l'information sur les types du code qui a généré cette erreur est souvent très utile pour le débugger.

Pourquoi Julia est rapide? Gâteau à trois ingrédients

À première vue, les performances de Julia sont invraisemblables. Il s'utilise comme un langage interprété et il n'est pas strictement nécessaire de noter le type de chaque variable. Pourtant, Julia présente régulièrement des performances comparables à C ou C++, malgré l'utilisation d'un garbage collector, et est au moins un ordre de magnitude plus rapide que Python ou Matlab. Le secret derrière cette sorcellerie tient dans la conception de Julia et présente au moins trois grands aspects.

Compilation JIT, la farine

Malgré les apparences, Julia est techniquement un langage *compilé*. Le langage utilise cependant une stratégie d'évaluation particulière appelée *Just In Time Compilation* (JIT), ou compilation juste à temps, en français. Son principe est le suivant.

Quand une fonction implémentée en Julia est appelée, le runtime Julia regarde si cette fonction a déjà été appelée avec des arguments du même type. Si ce n'est pas le cas, le compilateur Julia est appelé pour générer une version optimisée de la fonction pour ces types d'arguments. Cette fonction optimisée (que nous appellerons *méthode* par la suite) est ensuite stockée dans une table globale pour être utilisée la prochaine fois que la fonction sera appelée avec les mêmes types d'arguments.

L'avantage de cette stratégie est qu'elle permet de profiter des optimisations du compilateur sans pour autant devoir compiler tout le programme en avance. On ne compile que les parties du programme dont on a besoin. Le désavantage, c'est que cette étape de compilation peu prendre du temps quand on lance son programme. Dans la communauté Julia, ce problème est connu sous le nom de "Time to first plot".

Notez que le compilateur Julia n'est appelé que pour des fonctions. Ce point est important, est il est toujours encouragé d'utiliser un maximum de fonctions dans ses programmes Julia.

Le système de types, le beurre

Si la compilation JIT est déjà responsable à elle seule d'une bonne partie des performances de Julia, elle n'explique pas tout. En effet, il existe des implémentations JIT de Python (et Matlab) qui sont bien plus lentes.

Le système de types est conçu pour faciliter et augmenter le compilateur. Pour mettre cela en évidence, nous allons détailler dans les grandes lignes le système de types de Julia.

Types primitifs

Comme dans tout langage un tant soit peu performant, Julia possède les types usuels tels que les nombres entiers et les nombres à virgule flottante (en 32 et 64 bits), des charactères, des booléens, etc. Ces types sont dits *primitifs*, car ils sont directement liés à une représentation prenant un certain nombre de bits en mémoire.

Tableaux

Pour représenter un tableau d'objets, comme un vecteur ou une matrice, Julia utilise le type `Array`, qui comme en C correspond à un bloc de mémoire où les éléments du tableau sont rangés dans l'ordre. Par exemple, le type d'un vecteur de nombres entiers à 32 bits en Julia est `Vector{Int32}`. De façon générale, le type d'un tableau de dimension `N` avec des éléments de type `T` se note `Array{T,N}` (`Vector{T}` n'est en fait qu'un raccourci pour `Array{T,1}`). Le fait que les éléments soient tous de même type, et soient arrangés de manière contiguë en mémoire est important pour la performance, car cela veut dire que chaque élément occupe le même nombre de bits, et lorsque que l'on accède aux éléments séquentiellement, le processeur peut charger en avance les éléments proches l'un de l'autre en mémoire.

Types composites

Pour représenter un type composite, que les programmeurs C connaissent sous le nom de *struct*, Julia procède de la même manière que C ou C++. Les différents champs du type composite sont arrangés séquentiellement dans un même bloc mémoire. Par exemple, un type `ComplexF64` contenant deux champs de type `Float64` pour représenter la partie réelle et imaginaire d'un nombre complexe sera représenté par un bloc de mémoire où la partie réelle et la partie imaginaire seront stockées l'une à coté de l'autre. Les avantages de cette représentation sont les mêmes que pour les tableaux.

Types paramétriques

L'exemple précédent avec `ComplexF64` illustre un problème récurrent en C. Si pour une raison ou une autre, le programmeur voulait avoir un type complexe avec des float à 32 bits ou des entiers, il devrait réécrire une structure exactement similaire à `ComplexF64`, mais avec l'autre type à la place. Plus encore, il devrait réécrire pour ce nouveaux type toutes les fonctions qu'il avait implémentées pour `ComplexF64`. Or dans la plupart des cas, le code de ces fonctions est exactement le même.

Idéalement, nous aimerions pouvoir écrire nos fonctions une seule fois en faisant abstraction du type exact utilisé pour les parties réelles et imaginaires. Pour cela, Julia utilise des *types paramétriques* (aussi appelés types génériques). Un type paramétrique est de la forme `MonType{T}`, où `T` est le paramètre (on peut le comprendre comme une variable ayant un type comme valeur). Par exemple, la forme paramétrique du type complexe en Julia est `Complex{T}`. Nous pouvons maintenant écrire nos fonctions pour `Complex{T}` et celles-ci marcheront automatiquement pour `Complex{Float64}` ou `Complex{Int32}`!

Pour terminer cette section, notons que le type `Array{T,N}` vu précédemment est un exemple de type paramétrique. Il est cependant un peu particulier parce que le paramètre `N` n'est pas un type, mais un entier.

Types abstraits et hiérarchie de types

Jusqu'ici nous n'avons vu que des types concrets, directement liés à une représentation en mémoire. Or la plupart du temps, les programmeurs Julia manipulent des types *abstraits*. Un type abstrait n'est lié à aucune représentation en mémoire et ne peut donc jamais être instancié pendant l'exécution d'un programme.

À quoi servent donc ces types? Pour cela, il faut introduire l'autre élément qui est une *relation d'ordre partiel* sur les types. Julia utilise la notation `S <: T` pour dire que le type `S` est un *sous-type* du type `T`. Un type peut avoir un nombre arbitraire de sous-types ou sur-types. Un type concret ne peut pas avoir de sous-type. Enfin, il existe un plus grand sous-type, appelé `Any`.

Cette structure d'ordre sur les types se représente naturellement sous la forme d'un arbre, comme dans l'image suivante, qui montre l'ensemble des sous-types et sur-types du type `AbstractFloat` (le type abstrait pour les nombres à virgule flottante). Nous y voyons en bas les types concrets `Float64`, `Float16`, `BigFloat` et `Float32` et au dessus, nous voyons que `AbstractFloat` est un sous-type de `Real`, qui est un sous-type de `Number` qui est lui-même un sous-type de `Any`.

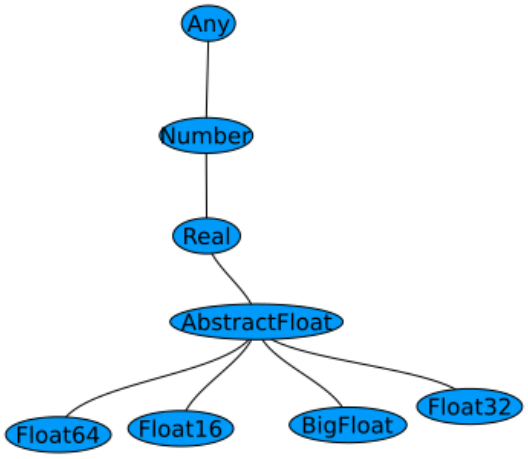

Figure 1: Sous-types et surtypes de `AbstractFloat`

Sans grande surprise, `Real` est le type abstrait pour représenter des nombres réels (entiers, rationnels, ...), alors que `Number` est le type abstrait pour représenter des nombres (e.g. nombres complexes, mais également des structures exotiques comme les nombres tropicaux)

La structure d'ordre entre les types est utilisée par le compilateur pour choisir quelle fonction appliquer lors d'un appel. En effet, Julia applique le *principe de substitution de Liskov*, qui dit que toute instance d'un sous-type `S` d'un type `T` doit pouvoir être utilisée à la place d'un objet de type `T` sans que cela ne change le résultat du programme. En d'autres termes, toute fonction définie pour un type abstrait `T` est automatiquement définie pour tous les sous-types de `T`.

Le principe de Liskov nous permet donc de définir une seule fois des fonctions à haut niveau, qui fonctionneront pour un grand nombre de types concrets. L'autre avantage est que la structure des types abstraits permet de distinguer les hypothèses que nous faisons sur nos types.

Par exemple, le type `Number` sert à représenter des objets ayant des lois de compositions algébriques (addition et multiplication) vérifiant certaines propriétés bien connues (associativité, distributivité, neutre et symétrisabilité, ...). Le programmeur est alors encouragé à réfléchir aux hypothèses minimales pour que fonctionnent ses algorithmes et à les implémenter au niveau le plus général possible.

Notons cependant que ces contraintes ne sont pas vérifiées formellement par le compilateur Julia, mais dépendent du programmeur. Malgré cette lacune, il y a de nombreux exemples où des fonctions implémentées à un niveau général par un programmeur fonctionnaient d'emblée sur les types d'un autre programmeur, sans modification!

Le *Multiple Dispatch* et la *type stability*, les œufs

Après cette longue description du système de types de Julia, nous pouvons enfin parler des deux propriétés qui permettent au compilateur Julia d'exploiter l'information sur les types pour optimiser les programmes qu'écrivent le développeur. Celles-ci sont dans l'ordre le *Multiple Dispatch* et la *stabilité des types*.

Le Multiple dispatch est au cœur du mécanisme de compilation JIT. Au moment de compiler un appel de fonction avec une combinaison particulière de types d'arguments, le compilateur cherche la définition la plus spécifique pour cette combinaison particulière de types.

Par exemple, supposons que nous cherchions à calculer le déterminant d'une matrice. La librairie standard de Julia définit un type de matrice général, ainsi que des types spécialisés tels que `Diagonal` ou `Triangular`, tous étant des sous-types de `AbstractMatrix`. Au moment d'évaluer le déterminant d'une matrice `A` avec `det(A)`, si `A` est de type `Diagonal` ou `Triangular`, son déterminant sera calculé avec l'algorithme approprié pour ce type de matrice (qui aura été défini au préalable par un programmeur), sinon, la méthode par défaut héritée de `AbstractMatrix` sera utilisée.

Le multiple dispatch permet donc de définir une implémentation générique sur les types abstraits, tout en laissant le programmeur définir des versions spécialisées de cette fonction pour des types plus spécifiques.

L'autre propriété, la stabilité des types, est plus technique à expliquer, mais revient essentiellement à l'idée que le compilateur doit pouvoir prouver des propriétés sur le programme qu'il cherche à optimiser. Formellement, une fonction est dite *type-stable* si le type de sortie de cette fonction peut être déterminé à partir du type de ses arguments.

La stabilité des types est extrêmement importante pour la performance du code compilé. En effet, si le type de sortie d'une fonction ne peut pas être déterminé par le compilateur, celui-ci doit être déterminé pendant l'exécution, ce qui a un impact désastreux sur la performance de la fonction. Pire encore, si une fonction est type-instable, le problème se propage à toutes les fonctions qui appellent cette fonction et ainsi de suite, récursivement.

Tout comme les propriétés des fonctions définies sur les types abstraits, la stabilité des types n'est pas une contrainte formelle. Il est tout à fait possible d'écrire du code qui n'est pas type-stable, mais cela est bien sur peu recommandé. Fort heureusement, le design de Julia fait qu'en dehors de certains cas particuliers, il est plus difficile d'écrire du code instable que stable.

Sucre sémantique, et cerises sur le gâteau...

Les éléments que nous venons de discuter, la compilation JIT, le système de type, la multiple dispatch et la stabilité des types sont la source de l'essentiel des performances de Julia. Il y a cependant un certain nombre de fonctionnalités du langage qui permettent au programmeur de maximiser les performances de son code.

En effet, Julia fournit par défaut un certain nombre d'outils pour évaluer la performance d'un programme, par exemple la macro `@time` qui permet de mesurer le temps d'exécution et la quantité de mémoire allouée pendant l'exécution d'un appel de fonction. Il est également possible pour

le programmeur de construire ses propres outils. Par exemple, la librairie `BenchmarkTools` fournit un certain nombre de macros pour rigoureusement évaluer la performance de son code.

Il y a également des macros qui permettent d'inspecter le code compilé d'un appel de fonction. En particulier, la macro `@code_warntype` affiche le code compilé dans un langage intermédiaire avec les types inférés et mets en évidence les problèmes d'instabilité de type.

Enfin, il est également possible, de nouveau via des macros, d'extraire encore un peu de performance en sacrifiant un peu de sûreté. Par exemple, la macro `@inbounds` intervient lors de l'indexage d'un tableau et désactive la vérification qu'un index donné est bien valide pour ce tableau.

Nous venons de citer sans les définir des objets appelés *macros*. Ce terme est familier de habitués du langage C, où il désigne des règles de substitution d'expression définies par le programmeur dans son code, et qui sont appliquées lors de la compilation. Les macros Julia sont beaucoup plus sophistiquées que cela. Elles sont en fait des fonctions spéciales, qui prennent en argument une expressions Julia et retournent une autre expression.

Pour clarifier ce concept, il est utile de parler du concept plus général de *citoyens de première classe*. Dans un langage de programmation donné, un citoyen de première classe est un type d'objets qui

1. Peuvent être utilisés comme arguments d'une fonction
2. Peuvent être utilisés comme résultat d'une fonction
3. Peuvent être assignés à une variable
4. Peuvent être comparés via un opérateur d'égalité

Nous pouvons ajouter une cinquième propriété qui est la possibilité de créer ces objets dans une expression sans les assigner à une variable (on peut parler de variable anonyme).

En C et Fortran, seules les instances de types primitifs et composites sont des citoyens de première classe. Ni les tableaux, ni les fonctions ne vérifient complètement ces propriétés, même si certaines sont partiellement vérifiées via l'usage de pointeurs (dont l'usage est notoirement complexe).

Par contraste, Julia admet un certain nombre de citoyens de première classe. Les tableaux et toute instance de type sont trivialement de première classe, mais également les fonctions (via le type `Function`; 1. et 2. sont vérifiées par ce qu'on appelle des *fonctions d'ordre supérieur*, 5. par les *lambda expressions*), les types (il existe un type `Type` dont les instances sont les types) et les expressions Julia (via le type `Expr` et les macros)

Le fait d'avoir ces trois types d'objets comme citoyens de première classe confère énormément d'expressivité à Julia, car cela permet au programmeur et au compilateur de raisonner dessus. Le système de macros en particulier est extrêmement puissant, au point de permettre au programmeur de directement modifier le compilateur. Rares sont les langages qui offrent une telle capacité d'introspection, et il existe déjà une myriade de librairies Julia qui font usage de cette fonctionnalité que ce soit pour inspecter l'exécution du code, mesurer en détail les performances ou directement changer la manière dont le code source est interprété.

Enfin, terminons cette section avec l'un des plus beaux fruits du design de Julia. Il est extrêmement courant aujourd'hui dans les applications scientifiques de calcul intensif de faire usage de

calcul parallèle, que ce soit via des processeurs multicores, de clusters ou de GPUs. Écrire et débugger des programmes parallèles est notoirement compliqué, autant sur le plan technique que sur le plan conceptuel, plus encore sur des langages comme C ou Fortran qui n'ont pas été conçu pour cela au départ.

Pour ces raisons, une nouvelle génération de langages de programmation est apparue pendant les 20 dernières années (e.g. D, Go, Rust, Nim), qui déclarent tous comme objectif de palier au défauts de C/C++ tout en conservant leurs performances, notamment en supportant nativement le calcul parallèle et concurrent.

Sous bien des aspects, Julia fait partie de cette nouvelle génération, et exploite habilement son design pour rendre l'utilisation de calcul parallèle presque trivial. Ainsi, la librairie standard de Julia offre la macro `@threads` pour exécuter une boucle en parallèle sur plusieurs threads de la façon suivante

```
@threads for i in 1:length(xs)
    xs[i] = foo(xs[i])
end
```

Similairement, la macro `@distributed` permet de faire la même chose sur un cluster

```
@distributed for i in length(xs)
    xs[i] = foo(xs[i]) # NB. restrictions apply
end
```

Ce dernier cas de figure est facilité par l'usage du package `DistributedArrays` qui fournit un type de tableau distribué à travers un cluster.

Enfin, le package CUDA fournit le type `CuArray` qui représente un tableau stocké dans la mémoire d'un GPU (Nvidia uniquement). Ce type fonctionne comme une `Array` normale, ce qui permet de d'automatiquement utiliser la plupart des fonctions utilisateur sur GPU sans modification. C'est par exemple le cas de `DifferentialEquations`, où il suffit dans la plupart des cas de mettre la condition initiale dans une `CuArray` pour que la solution de l'ODE soit calculée sur GPU.

Conclusion

Au cours de ce document, nous sommes passé en détail sur ce qui rend le langage Julia si rapide, à savoir la compilation JIT, supportée par un système de types particulièrement bien conçu et le multiple dispatch. Ce n'est bien sur que la moitié de l'équation, car le langage offre flexibilité et modularité, ainsi que des outils fantastiques pour pour évaluer la performance d'un programme.

Le résultat final est que non seulement Julia est rapide, mais il (elle?) rend l'usage de fonctionalités complexes comme le paralélisme aussi trivial et sans douleur que l'on puisse imaginer. Bien qu'il soit toujours possible d'écrire du code lent en Julia, la conception du langage rend cela plus dur que d'écrire du code efficace dans la plupart des cas.