Impact des langages de programmation sur la performance énergétique des applications de calcul scientifique : un challenge ?

Cyrille Bonamy
Laboratoire LEGI - UMR5519
1209-1211 rue de la piscine
Domaine Universitaire
38 400 Saint-Martin d'Hères

Laurent Bourgès
OSUG - Observatoire des Sciences de l'Univers de Grenoble - UAR832
122 rue de la piscine
38 400 Saint-Martin d'Hères

Laurent Lefevre
Inria, Laboratoire LIP
École Normale Supérieure de Lyon
46 allée d'Italie
69 364 Lyon

Résumé

*Alors que la France s'engage sur le chemin de l'*exascale*, le calcul scientifique est de plus en plus utilisé dans la société. Il permet de concevoir, simuler et optimiser de nombreux scénarios pour un coût limité comparativement à la réalisation de prototypes.*

Il apparaît primordial de se poser la question de l'impact environnemental associé au calcul scientifique. Lorsque l'on développe un code de calcul scientifique, la première question qui se pose est celle du choix du langage. Il est commun de considérer un certain nombre de critères (lisibilité, pérennité, efficacité, maîtrise du langage). Il est bien plus rare de considérer la consommation et l'efficience énergétique. Cette étude a pour objectif de sensibiliser et inciter à la considération de ce critère.

Un code de démonstration permettant la simulation d'écoulements a ainsi été considéré.

Initialement développé en Python, ce code a été accéléré (Pythran, Numba) puis porté dans les langages C, Fortran, Java, Julia, Rust, Go. Des mesures de performance et de consommation électrique ont été réalisées sur la plate-forme expérimentale Grid'5000, ce qui a permis d'évaluer, dans le contexte très précis du code considéré, l'efficience des langages considérés sur plusieurs architectures.

Le portage rigoureux de code dans un autre langage que celui d'origine, ainsi que l'optimisation de code prend un temps considérable, mais il y a des potentiels de gains énergétiques importants sur la consommation de la phase d'usage du service associé.

Ce travail, réalisé dans une approche de science ouverte et reproductible, montre ainsi l'importance des développeurs sur les impacts environnementaux des usages du numérique.

Mots-clefs

Langages, calcul scientifique, comparatif énergétique, optimisation, écoresponsabilité

1 Introduction et positionnement

Alors que la France s'engage sur le chemin de l'*exascale*, le calcul scientifique est de plus en plus utilisé au sein de l'ESR et plus généralement dans la société. Auparavant réservé à une élite, basé sur des grandes infrastructures de calcul et de stockage, il est désormais une brique de base de bon nombre d'entreprises.

En effet, le calcul scientifique permet de concevoir, de simuler et d'optimiser de nombreux scénarios pour un coût limité comparativement à la réalisation de prototypes ou d'essais *in situ*. Il est également une brique de base pour de nombreux services de *big data* et d'intelligence artificielle.

Il apparaît primordial de se poser la question de l'impact environnemental de ces pratiques numériques. Lorsqu'on envisage de développer un code de calcul scientifique, la première question qui se pose est celle du choix du langage et son environnement (bibliothèques). Afin d'accomplir ce choix, il est commun de considérer un certain nombre de critères, tels que lisibilité, pérennité, efficacité ou encore connaissance du langage par les développeurs. Il est bien plus rare de considérer l'efficience énergétique [1]. Une des raisons qui empêche que ce critère soit considéré est qu'il n'existe aujourd'hui que très peu d'études sur l'efficience énergétique en fonction des langages [2][3][4][5].

Cette étude vient enrichir de manière originale ces trop rares travaux et constitue la synthèse d'un travail réalisé entre 2019 et 2021 au sein du groupe Eco-conception logicielle du GDS EcoInfo, un groupement de service du CNRS. Des travaux pratiques ont été réalisés aux Actions Nationales de Formation « EcoInfo » en 2019 et 2021 pour sensibiliser les stagiaires aux « Impacts environnementaux du numérique ». Des mesures de consommation énergétique ont ainsi été réalisées pendant la phase d'usage d'un code de calcul scientifique, code qui a été porté dans plusieurs langages de programmation.

Toutes les expérimentations numériques associées à ce travail ont été réalisées dans une approche de science ouverte et reproductible pour permettre à tous de les exécuter et de réaliser des mesures sur leurs propres machines : le dépôt public « ecolang », sous licence libre GPL2, contient l'ensemble des codes, des scripts d'exécution et de traitement des données, ainsi que toutes les données et résultats obtenus [6].

2 Méthode

2.1 Cas scientifique et code de simulation initial

Un code permettant la simulation d'écoulements 1D (voir Figure 1) a été considéré dans cette étude, car il représente à la fois un vrai cas scientifique et aussi un code simple et idéal : un algorithme linéaire non vectorisable (intensif en usage processeur et mémoire, sans entrées/sorties). La fonction la plus gourmande en termes de temps de calcul (et donc de consommation d'énergie) est une inversion de matrice creuse, un problème classique rencontré en physique. Ce code est utilisé lors d'un cours de Mécanique des Fluides[1] par le Maître de Conférences Julien Chauchat.

1 https://gricad-gitlab.univ-grenoble-alpes.fr/ecoinfo/ecolang/-/blob/master/tp/2.2_langages_et_algos/FormationVFMecaFlu.pdf

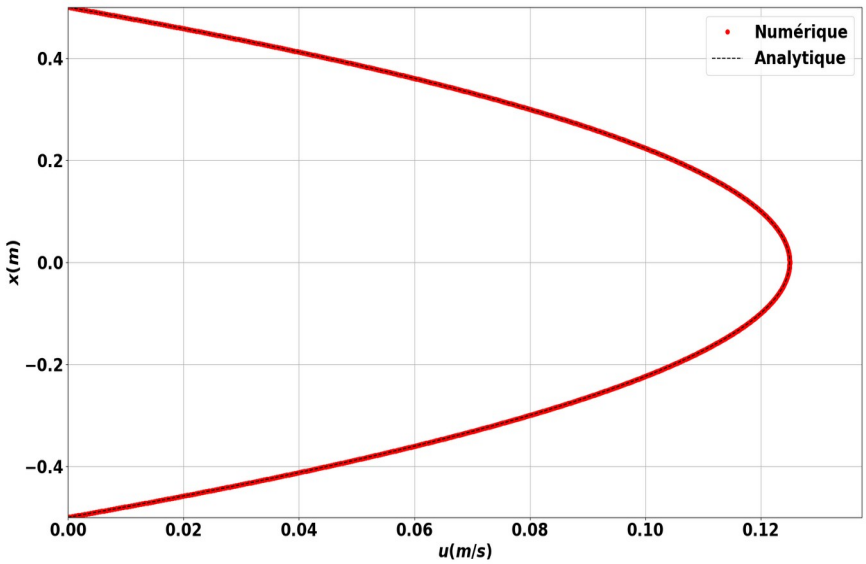


Figure 1 : Solution analytique / numérique d'un écoulement 1D entre deux plaques

La version initiale du code a été écrite par le scientifique en Python + Numpy. Cependant, la taille de la matrice (N × N) devient rapidement un facteur limitant en termes de mémoire disponible (N étant le nombre de cellules considéré afin de représenter l'espace physique). Un autre algorithme « TriDiagonal Matrix Algorithm » (TDMA) ou algorithme de Thomas [7] a donc été implémenté. En effet, il permet l'inversion des matrices tridiagonales de manière récursive en stockant uniquement 4 vecteurs de taille N, d'où un gain significatif de temps et d'empreinte mémoire.

Une première comparaison sur un ordinateur portable HP ZBook 15 (`fixed freq=2Ghz`) a été réalisée entre le code initial (inversion de matrice naïve via *numpy.linalg.solve*[2]) et la version TDMA sur un nombre de cellules de 2048 et 4096 pour évaluer la scalabilité de l'algorithme en fonction de la taille du problème :

Algorithme	Temps (s) N=2048	Temps (s) N=4096	Ratio (Temps) = T_4096 / T_2048
inversion matrice (*main1D.py*)	3.72	22.73	x 6.1
TDMA (*main1D_cb.py*)	0.36	0.72	x 2
Speedup TDMA = T_mat / T_tdma	x 10.31	x 31.5	

En termes de temps de calcul, les gains sont très nets avec l'algorithme TDMA. De plus, la version originale est 6 fois plus lentes pour 2 fois plus de cellules (la matrice représente déjà 8×4096^2 = 128 Mo pour N = 4096) quand l'algorithme TDMA est bien linéaire.

En étendant le problème à 4 millions de cellules ($N = 2048^2$), l'empreinte mémoire de l'algorithme TDMA est très réduite, environ 4 vecteurs de 32 Mo, en comparaison à une matrice (N × N) de 16 To !

2 https://numpy.org/doc/stable/reference/generated/numpy.linalg.solve.html

2.2 Code de simulation avec l'algorithme TDMA

Pour obtenir une comparaison honnête et équitable entre plusieurs langages, le code python a été nettoyé et structuré de manière à être plus lisible et clair afin d'aider un portage trivial dans les autres langages. De plus, les entrées et sorties du code ont été normalisées et sauvegardées pour vérifier les hypothèses et la qualité des résultats (0 ULP[3] en double précision) lors de l'exécution sur les différents environnements matériels et logiciels.

Pour avoir des temps de réponse suffisants (> 30 s) et mesurer une consommation énergétique représentative des machines, le code a été poussé à ses limites en considérant un nombre de cellules égal à quatre millions. Bien qu'insensé dans le cas de simulations 1D, ce nombre est tout à fait réaliste pour des simulations 3D qui pourraient utiliser un code très similaire à celui-ci. Pour éviter de rajouter de la complexité à cette étude (et notamment au portage du code), nous avons conservé le code 1D, tout en considérant quatre millions de cellules dans nos tests, au lieu de 2048. Il faut remarquer que la complexité de cet algorithme basé sur l'algorithme TDMA est ~ O(n) par rapport au nombre de cellules, meilleur que l'inversion de matrice en ~ O(n^2).

À partir du code python avec l'algorithme TDMA (une centaine de lignes de code), différentes versions du code en python ont été accélérées grâce aux outils Pythran[4] puis Numba[5], des compilateurs Numpy en code natif de type « *Ahead Of Time compiler* » (AOT) et « *Just In Time compiler* » (JIT). Ensuite, le code a été porté rigoureusement dans les langages C, Fortran, Java, Julia, Rust et Go, représentant des langages compilés (C, Fortran, Rust, Go), interprétés (Python), compilés à la volée (JIT : Java, Julia, Numba) ou à l'avance (AOT : Pythran). Plusieurs environnements d'exécution reposent sur le compilateur LLVM comme Numba, Julia, Rust et Go.

Pour aller plus loin, une version dite « optimisée » du code a été réalisée pour chaque langage avec une idée simple : pré-allouer la mémoire (en dehors des boucles) au début du programme. En effet, l'algorithme TDMA utilise plusieurs vecteurs (32 Mo) à chaque itération du solveur (26 par simulation), donc la version optimisée ne fait aucune allocation de mémoire dans le code de calcul intensif. En outre, quelques rares fonctionnalités propres à certains langages ont été utilisées pour se comporter comme C (*no bound-checks*, *no overflow checks*, accès direct à la mémoire comme des pointeurs, pas de copies des tableaux), soit une programmation dite « *unsafe* » :

- Julia : *in-place operators* (`A .= B` au lieu de `A = B`, `@inbounds`)
- Rust : itérateurs pour éviter les *bound-checks* sur les boucles

Ce travail supplémentaire pour améliorer la gestion de la mémoire et optimiser les boucles demande de l'expérience et du temps de programmeur pour implémenter et surtout valider les changements, temps qui devient vite le facteur limitant pour améliorer un code, évaluer différentes approches ou bibliothèques tierces et mesurer les impacts.

Des mesures de performance et des mesures de consommation électrique ont été réalisées sur la plateforme expérimentale Grid'5000 et sur des machines plus standards, en respectant un protocole de test rigoureux basé sur des scripts *shell* pour assurer la reproductibilité et simplifier la gestion :

- installer les paquets et bibliothèques Linux, compiler les codes et lancer un test unitaire ;
- collecter des informations CPU et définir le gouverneur CPU (performance, *NoHyperThreading*, *turbo-boost* activé), typique en HPC ;
- lancer les tests de performance avec *timestamps* avant / après, utilisés pour la collecte des données des wattmètres (précision à 1 s) sous 2 modes :

3 https://en.wikipedia.org/wiki/Unit_in_the_last_place

4 https://pythran.readthedocs.io/en/latest/

5 https://numba.pydata.org/

- 1 seul programme, donc mono-*thread* ;
- multi-*thread* (100 % CPU) : un même programme est lancé sur chaque cœur physique simultanément (`taskset -c`) ; c'est le cas du parallélisme idéal (*embarrassingly parallel*[6]) ou bien des études paramétriques qui sont très courantes dans nos domaines (on effectue un grand nombre de calculs quasi-identiques ; un seul paramètre physique d'entrée du programme est modifié, ce qui ne change pas la complexité algorithmique) ;

- sauvegarder toutes les sorties des programmes exécutés pour la traçabilité.

Des scripts Python ont été mis au point pour agréger les résultats, calculer l'énergie consommée par les programmes et produire toutes les tableaux de résultats et figures.

2.3 Séquence de tests

Pour chaque langage, plusieurs opérations sont réalisées : installation et compilation en amont (la consommation électrique de la phase de compilation pour les langages C, Fortran, Rust et Go est omise), puis les tests et mesures sont réalisés :

- *timestamp* « *Lang Start* » ;
- les simulations sont exécutées séquentiellement sur tous les langages d'abord en mono-*thread* (*Run Benchmark on 1 CPU*, mono-*thread*, [1T] sur les figures) ;
- pause de 2 s, pour revenir en état *idle ;*
- puis en multi-*thread* (*Run Benchmarks on … CPUs*, multi-*thread*, [MT] sur les figures) ;
- pause de 5 s, pour revenir en état *idle ;*
- *timestamp* « *Lang Stop* ».

Chaque code de simulation (dans tous les langages) réalise la même séquence :

- phase « *warmup* » (~ 5 s) : 100 itérations du même calcul avec un nombre de cellules de 65536, afin d'obtenir des mesures fiables pour les langages avec un compilateur JIT ;
- pause de 3 s, pour revenir en état *idle ;*
- *timestamp* « *BENCHMARK Start* » ;
- phase « *run* » (~ 30 s): 10 itérations du même calcul avec un nombre de cellules de 2048 × 2048 ;
- *timestamp* « *BENCHMARK Stop* ».

La mesure de la consommation d'énergie est collectée sur toute la durée [Lang: Start - Stop] (cf. le profil Figure 2 représentant la consommation d'un nœud Xeon5218 pour le langage C : consommation *idle* à 190 W, tests [1T], *warmup* puis *run*, à 280 W puis tests [MT], *warmup* puis *run* à 475 W).

Les mesures présentées par la suite se basent uniquement sur la partie « *BENCHMARK* » [BENCHMARK: Start - Stop]. Cependant la précision est de +/- 1 s à cause de la précision du *timestamp* (valeur entière des secondes).

Le calcul des 10 itérations de la partie « *BENCHMARK* » (nombre de cellules de 2048 × 2048) est ce qu'on considère comme notre « unité de travail ».

6 https://en.wikipedia.org/wiki/Embarrassingly_parallel

Dans le cas mono-*thread*, le temps d'exécution présenté correspond au temps de calcul de cette « unité de travail » et de même pour la consommation électrique (*Energy* dans nos figures).

Dans le cas multi-*thread*, où on charge tous les cœurs de la machine considérée, nous aurons ainsi accompli N_{cores} « unités de travail ». Ainsi, afin de pouvoir comparer les choses entre elles, dans ces cas multi-*thread* (MT), toutes les mesures sont normalisées par le nombre de cœurs N_{cores} : consommation électrique / N_{cores} et temps d'exécution / N_{cores}.

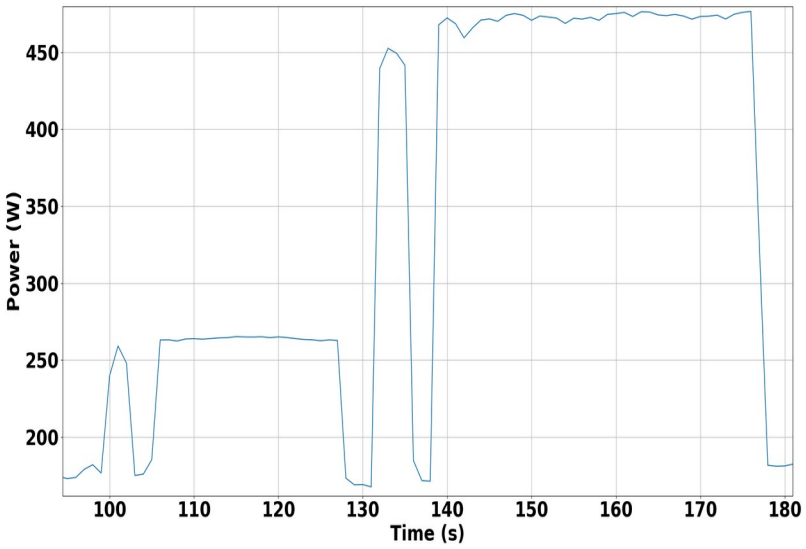


Figure 2 : Profil énergétique d'une séquence de tests

Pour résumer, toutes les données présentées correspondent à une même « unité de travail ».

3 Langages considérés

Les langages considérés ont été choisis en fonction des critères suivants : part de marché, maturité, richesse de l'écosystème dans le domaine HPC ou IT, et expertise des auteurs :

Langage	Caractéristiques	Environnement
C	compilé, *malloc/free*, pointeurs, *low-level*	Gcc 10.2.1
Fortran	compilé, indices à 1	GNU Fortran 10.2.1
Java	JIT, GC	OpenJDK 11.0.13
Julia	JIT, GC, indices à 1, syntaxe math	Julia 1.7.1
Python	interprété (GIL), pas de tableaux Numba: JIT, GC Transonic/Pythran: AOT, GC	Python 3.9.2 Numba 0.54.1 Transonic 0.4.12
Rust	compilé, références modifiables (*mutable*), *low-level*	Rustc 1.57.0
Go	AOT, GC	Go 1.17.5

4 Plate-forme expérimentale

Les expérimentations ont permis d'évaluer, dans le contexte très précis du code considéré, la consommation et l'efficience énergétique des langages considérés sur plusieurs architectures :

- Xeon 5218 (troll) : un serveur bi-processeur à base d'Intel Xeon nouvelle génération (Grid'5000) : Dell PowerEdge R640, Intel Xeon Gold 5218 (Cascade Lake-SP, 2.30 GHz);
- AMD EPYC 7642 (neowise) : un serveur mono-processeur à base d'AMD nouvelle génération et de 8 accélérateurs graphiques que nous n'utilisons pas dans cet article (Grid'5000) : AMD-Penguin Computing, AMD EPYC 7642 (Zen 2) ;
- ARM X2 99xx (pyxis) : un serveur bi-processeur à base d'ARM64 (Grid'5000) : R181-T92-00, ARM ThunderX2 99xx (Vulcan) ;
- AMD Opteron 250 (sagittaire) : un serveur bi-processeur à base d'AMD ancienne génération (Grid'5000) : Sun Fire V20z, AMD Opteron 250 (K8) ;
- HP ZBook 15 G3, un portable « workstation » à base de Intel Core i7-6820HQ (Skylake) ;
- RPi4 : une carte Raspberry Pi 4 Model B (quad-core A72, ARMv8-A 64-bit, 1.5GHz).

Type	Année	CPU	Nombre de coeurs	Flops (Gflops)	Puissance *idle* (W)	Puissance max (W)
Xeon 5218	2019	Intel Xeon Gold 5218	2 × 16	2300	160	500
AMD EPYC 7642	2021	AMD EPYC 7642	1 × 48	1800	307	451
ARM X2 99xx	2020	ARM ThunderX2 99xx	2 × 32	1100	255	425
AMD Opteron 250	2006	AMD Opteron 250	2 × 1	9.6	193	262
HP ZBook 15 G3	2017	Intel Core i7-6820HQ	1 × 4	173	10	62
RPi4	2020	A72, ARMv8	1 × 4	12	2	5.1

Les flops sont en FP64 théoriques (estimés et non mesurés) et les puissances électriques instantanées (*idle*/max en W) sont des mesures (effectuées avec l'outil `stress -cpu` pour les max).

La consommation électrique mesurée sur Grid'5000 est effectuée par des wattmètres matériels à haute fréquence (50 mesures par seconde, intégrées sur 1 seconde). Les wattmètres sont directement reliés aux prises des serveurs. La mesure considère donc la consommation d'un serveur (nœud) complet et n'inclut pas la consommation électrique du système de refroidissement du datacentre.

Les consommations du portable HP ZBook 15 et du Raspberry Pi 4 ont été mesurées par un wattmètre simple pendant des tests utilisant 1 puis 4 cœurs (100 %) et la puissance typique a été utilisée pour avoir une approximation de l'énergie consommée.

Les systèmes d'exploitation considérés sont tous basés sur Linux : Debian 11 pour les machines Grid'5000, Ubuntu 20.04 pour le HP ZBook 15 et Rasbian-32 pour le Raspberry Pi 4.

5 Résultats

Comme expliqué précédemment (cf. paragraphe 2.3), notre campagne de mesures a été menée avec deux scénarios différents : exécution sur un seul cœur de calcul (mono-*thread* [1T]) et exécution sur l'ensemble des cœurs de calcul disponible sur la machine considérée (multi-*thread* [MT]). Le langage Python étant le seul langage interprété dans notre étude, nous nous focalisons d'abord sur l'impact de la compilation de ce langage en termes de performances afin d'en extraire la version la plus intéressante en termes de consommation énergétique. Nous présentons ensuite le comparatif entre langages dans les deux scénarios considérés ainsi qu'un comparatif entre les différentes architectures mises en œuvre.

5.1 Focus sur le langage python, impact de la compilation

Le tableau suivant illustre l'impact de la compilation du langage Python, en mono-thread [1T] et en multi-thread [MT], pour le code de base (non optimisé).

Énergie (W.h), Temps d'exécution (s)	Python Numpy	Python Numba	Python Transonic/Pythran
Mono-thread [1T]	92.1 W.h, 1 260 s	2.77 W.h, 38 s	2.57 W.h, 35 s
Multi-thread [MT]	6.4 W.h, 54.1 s	0.27 W.h, 2.09 s	0.24 W.h, 1.84 s
Speedup	23.3	18.2	19

Table 1 : Tableau comparatif Python en usage Numpy, Numba, et Transonic/Pythran sur la machine Xeon 5218

On peut observer que la la consommation statique (en mode *idle*) de la machine est dominante en mono-*thread* et qu'un gain d'efficacité est prouvé en chargeant la machine (100 % en multi-thread).

Sur les temps de d'exécution, si on regarde le *speedup* en multi-*thread* (défini comme le temps d'exécution sur 1 cœur divisé par le temps d'exécution sur N_{cores}, 32 ici), on remarque qu'il n'est pas linéaire (*speedup* autour de 20 et non 32 idéalement). La différence s'explique en partie par l'usage du *turbo-boost*. Sur cette machine Xeon 5218, la fréquence du CPU peut varier de 2.3 Ghz (base) à 3.9 Ghz (*turbo*), ce qui explique pourquoi le *speedup* n'est pas parfait. Les fréquences des CPU peuvent ainsi être différentes entre les cas mono-*thread* et les cas multi-*thread*, ce qui explique pourquoi le *speedup* mesuré n'atteint pas la valeur idéale.

Les « compilateurs » Numba et Pythran via Transonic (JIT ou AOT) prouvent leur efficacité sur ce code avec des gains de l'ordre de 30 fois en temps et énergie par rapport à Python + Numpy. En effet, le code considéré, et plus précisément l'algorithme de Thomas, comporte une boucle récursive qui rend son usage via Python particulièrement inefficace, car la boucle est alors interprétée et la librairie Numpy ne propose pas de fonction native (C) appropriée.

Les résultats considérant les « compilateurs » Numba et Transonic/Pythran étant très similaires (quelle que soit l'architecture considérée), et en raison de la grande popularité de Numba, nous ne présenterons par la suite que les résultats considérant Numba.

5.2 Comparatifs entre les langages dans des configurations séquentielles ou parallèles

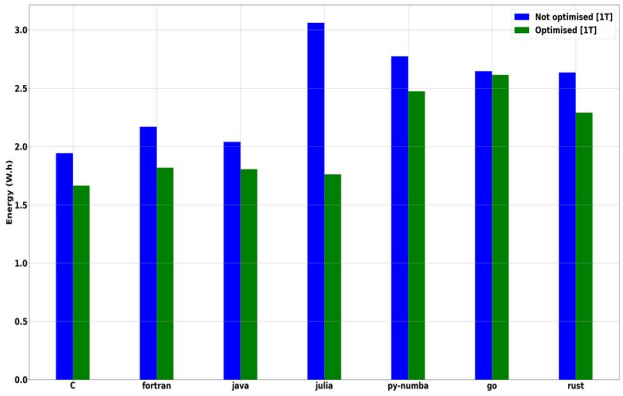


*Figure 3 : Comparaison énergétique pour un serveur bi-processeur à base de Xeon 5218 en utilisation mono-*thread

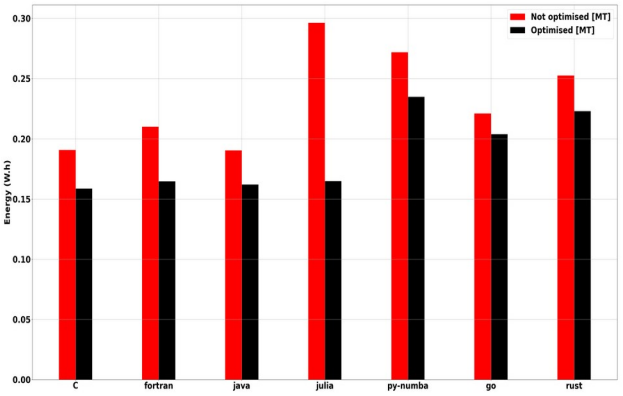


*Figure 4 : Comparaison énergétique pour un serveur bi-processeur à base de Xeon 5218 en utilisation multi-*thread

Les Figure 3 et Figure 4 permettent l'observation d'un comparatif énergétique entre les langages de programmation considérés.

La version « optimisée » offre des gains de l'ordre de 15 %, donc réduire les allocations de mémoire dynamique est bénéfique au calcul, certainement grâce à une meilleure efficacité et localité du cache CPU et moins de transferts de mémoire.

Les 4 premiers langages évalués (C, Fortan, Java et Julia) semblent efficaces et matures (même si la version en Julia non optimisée est particulièrement mauvaise). Les trois autres langages (Python-Numba, Go et Rust) consomment plus d'énergie (~ 30 %). Cette différence se vérifie particulièrement pour les versions « optimisées ».

Le langage C « optimisé » est le meilleur en consommation énergétique mais les autres langages se rapprochent de ses performances. Les compilateurs JIT ou AOT permettent de générer du code aux performances proches du C.

Enfin, bien que le langage Python sans utilisation de compilateur JIT (Numba) ou AOT (Transonic/Pythran) soit particulièrement mauvais dans ce cas d'usage précis (cf. 5.1), les résultats montrent que l'utilisation de ces bibliothèques permet de générer du code particulièrement performant et donc utilisable dans l'écosystème HPC (comme déjà démontré par Augier et al. [5]).

Toutes ces affirmations sont valables en mono-*thread* et en multi-*thread*. On observe néanmoins le grand intérêt du multi-*thread* sur le temps d'exécution et la consommation énergétique associée à une « unité de travail ».

Un graphe de type « *violin plot* », fourni en annexe, a été réalisé pour observer la dispersion des mesures qui s'avère très faible. Ceci prouve que la phase de *warmup* a été efficace pour les langages basés sur un compilateur JIT (Java, Julia) et que les machines sont bien stables et isolées (réservation exclusive des nœuds).

5.3 Comparatifs langages / infrastructures

Le tableau suivant illustre l'effet de l'infrastructure considérée sur les temps d'exécution et énergies consommées pour une « unité de travail », en mode mono-*thread*, puis en mode multi-*thread*, après normalisation des mesures comme expliquée paragraphe 2.3 (version du code « optimisée »).

Énergie (temps d'exécution)	C	Fortran	Java	Julia	Py-Numba	Go	Rust
Xeon 5218	1.66 (23)	1.89 (25)	1.8 (25)	1.76 (24)	2.47 (34)	2.61 (36)	2.29 (31)
AMD Opteron 250	4.23 (72)	4.58 (78)	5.89 (102)	6.51 (103)	9.08 (142)	7.31 (126)	7.04 (120)
AMD EPYC 7642	1.89 (18)	2.49 (24)	2.31 (22)	1.99 (19)	3.17 (30)	2.83 (27)	3.19 (30)
ARM X2 99xx	2.89 (40)	3.24 (45)	3.97 (55)	3.81 (53)	4.89 (68)	6.03 (84)	4.61 (64)
HP ZBook 15 *	0.25 (21)	0.3 (25)	0.27 (23)	0.27 (23)	0.38 (32)	0.38 (32)	0.36 (30)
RPi4 *	0.06 (69)	0.07 (78)	0.06 (71)	0.06 (71)	0.14 (170)	0.07 (84)	0.11 (129)

*Table 2 : Tableau comparatif des énergies consommées (W.h) et temps d'exécution en fonction des infrastructures considérées en mono-*thread *[1T] (s), * : estimation expliquée paragraphe 4*

Énergie (temps d'exécution)	C	Fortran	Java	Julia	Py-Numba	Go	Rust
Xeon 5218 (32C)	0.16 (1.22)	0.16 (1.28)	0.16 (1.28)	0.16 (1.28)	0.23 (1.81)	0.2 (1.51)	0.22 (1.69)
AMD Opteron 250 (2C)	2.44 (37.5)	2.62 (40.5)	3.25 (51.5)	3.25 (51.5)	4.54 (71)	4.01 (63.5)	3.96 (61.5)
AMD EPYC 7642 (48C)	0.18 (1.35)	0.19 (1.4)	0.22 (1.62)	0.19 (1.4)	0.26 (1.94)	0.19 (1.38)	0.27 (1.98)
ARM X2 99xx (64C)	0.09 (0.8)	0.09 (0.78)	0.1 (0.94)	0.1 (0.89)	0.14 (1.23)	0.14 (1.34)	0.13 (1.12)
HP ZBook 15 (4C) *	0.13 (7.5)	0.15 (8.75)	0.15 (8.75)	0.13 (7.75)	0.18 (10.25)	0.17 (9.75)	0.19 (10.75)
RPi4 (4C) *	0.07 (51)	0.07 (51.5)	0.08 (53.2)	0.07 (51.7)	0.11 (76.7)	0.07 (52)	0.10 (71)

*Table 3 : Tableau comparatif des énergies consommées (W.h) et temps d'exécution en fonction des infrastructures considérées en multi-*thread *[MT] (s), * : estimation expliquée paragraphe 4*

Les résultats confirment globalement que nos conclusions sur la machine Xeon 5218 sont vérifiées quelque soit l'infrastructure considérée.

Par ailleurs, il est intéressant de constater que les machines ayant à ce jour bonne réputation en termes de consommation électrique sont effectivement performantes. En effet, le Raspberry Pi 4 et la machine à base d'ARM ont des performances très intéressantes. Néanmoins, on constate bien que le Raspberry est particulièrement lent, contrairement à la machine à base d'ARM qui a le meilleur temps d'exécution.

On constate qu'utiliser des machines très performantes (avec beaucoup de cœurs) pour des calculs sur un seul cœur n'est clairement pas efficace. Cet usage est donc à prohiber. Il est par ailleurs intéressant de voir que développer et mettre au point du code sur un simple portable (HP ZBook 15) n'est pas insensé, et cela particulièrement pour des calculs mono-*thread*.

Enfin, on voit que la vieille machine (AMD Opteron) a du mal à suivre les performances des autres, que ce soit en termes de temps d'exécution ou d'efficacité énergétique. De plus, on peut remarquer que sur cette machine datant de 2006 les langages récents (Numba, Go, Rust) sont plus pénalisés que les langages historiques (C, Fortran). Il est toutefois important de rappeler qu'une analyse pertinente de ces diverses infrastructures nécessiterait de considérer les phases de fabrication du matériel, phases de fabrication qui ont un énorme impact environnemental. Ainsi, ces machines vieillissantes ne doivent pas être jetées, mais peut-être faut-il reconsidérer leurs usages à autre chose que du calcul intensif.

6 Conclusion

L'objectif de cette étude et de la présentation associée est de faire prendre conscience à l'auditoire de l'importance de considérer les impacts environnementaux des usages du numérique, et plus précisément de leurs développements et utilisations de logiciels.

Ce travail est focalisé sur la phase de développement d'applications scientifiques et plus particulièrement sur le choix des langages de programmation. Ce choix est important et fait partie des bonnes pratiques en éco-conception de service numérique [8].

Cette étude nous a appris quelques leçons qui nous apparaissent intéressantes à diffuser :

- le portage rigoureux et l'optimisation de code dans un autre langage que celui d'origine prend du temps et nécessite une expertise ; néanmoins, les gains peuvent être conséquents ;
- il est possible de générer du code Python performant, mais il apparaît important de considérer des bibliothèques permettant l'AOT ou le JIT afin d'être au niveau des langages historiques ;
- afin d'être efficace, tous les cœurs de la machine considérée doivent être utilisés ;
- les machines à base d'ARM sont réellement efficaces (du moins pour notre cas d'étude), quel que soit le langage considéré et le portage de code sur celles-ci est immédiat (contrairement à l'usage des accélérateurs classiques : GPU, XeonPhi) ;
- la meilleure optimisation consiste à revoir le code dans son approche algorithmique ou dans la réduction de l'empreinte mémoire, mais cela nécessite de limiter la précision numérique ou la taille du problème (N) et éviter les algorithmes en $O(N^2)$; dans notre exemple, un algorithme dédié au problème à résoudre (TDMA) fait vraiment la différence ;

- deux groupes de langages se détachent assez distinctement en termes d'efficacité pour notre cas d'étude : d'un côté, C, Fortran, Java et Julia qui sortent en tête, et de l'autre Python, Go et Rust.

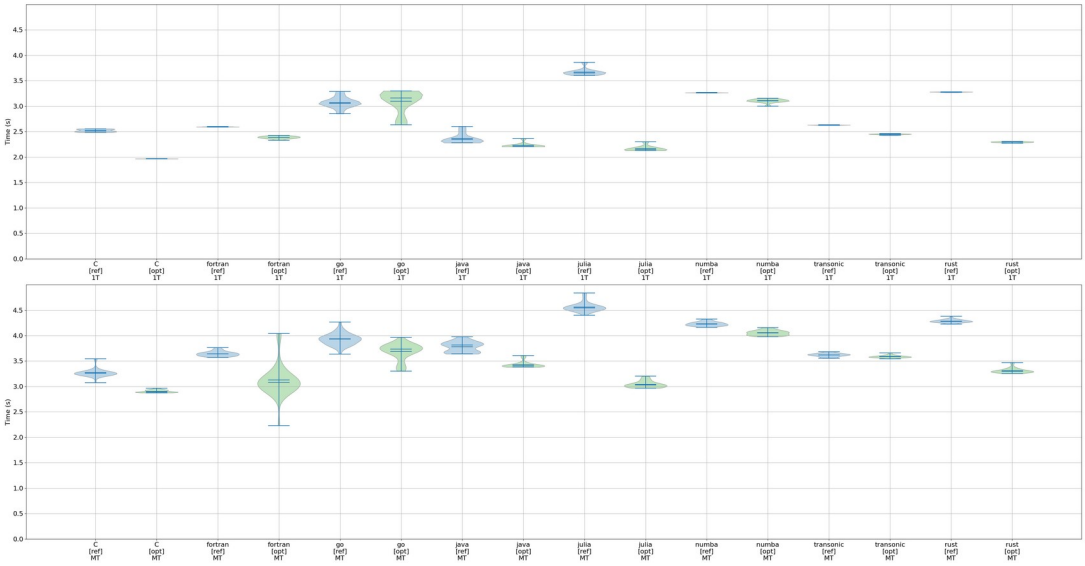
Par ailleurs, nous pensons que ce travail pourrait constituer un *benchmark* reproductible qui pourrait servir à mesurer l'évolution des langages C, Fortran, Python, Java, Julia, etc. Est-ce que les nouvelles versions améliorent ou non les temps d'exécution et la consommation énergétique associée ?

Cette étude vient ainsi compléter d'autres travaux faits par le GDS EcoInfo et par l'UAR GRICAD, comme l'« Empreinte carbone d'une heure cœur de calcul sur un mésocentre régional » [9], « le coût environnemental du transport d'un Go via le réseau Renater » [10], ou encore l'« Empreinte relative au stockage d'un Go sur un an dans un centre de donnée régional » [11].

Remerciements

Nous avons mené nos campagnes de mesures sur la plate-forme expérimentale Grid'5000 (https://www.grid5000.fr) que nous tenons à remercier. Grid'5000 est supporté par un GIS provenant de Inria, CNRS, RENATER et différentes universités.

Annexe : Dispersion des mesures sur ZBook 15



Bibliographie

[1] Fethullah Goekkus. Energy Efficient Programming - An overview of problems, solutions and methodologies. Bachelor thesis, University of Zurich, December 2013 ; https://files.ifi.uzh.ch/hilty/t/examples/bachelor/Energy_Efficient_Programming_G %C3%B6kkus.pdf

[2] Portegies Zwart, S. The ecological impact of high-performance computing in astrophysics. Nature Astronomy 4, 819–822, 2020 ; https://doi.org/10.1038/s41550-020-1208-y https://arxiv.org/pdf/2009.11295.pdf

[3] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy efficiency across programming languages: how do energy, time, and memory relate? In Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering. Association for Computing Machinery, New York, NY, USA, 256–267, 2017 ; DOI:https://doi.org/10.1145/3136014.3136031 https://greenlab.di.uminho.pt/wp-content/uploads/2017/10/sleFinal.pdf

[4] Stefanos Georgiou, Maria Kechagia, and Diomidis Spinellis. Analyzing Programming Languages' Energy Consumption: An Empirical Study. In Proceedings of the 21st Pan-Hellenic Conference on Informatics. Association for Computing Machinery, Article 42, 1–6, New York, USA, 2017; DOI:https://doi.org/10.1145/3139367.3139418

[5] Augier, P., Bolz-Tereick, C.F., Guelton, S. et al. Reducing the ecological impact of computing through education and Python compilers. Nat Astronomy, 334–335, 2021. https://doi.org/10.1038/s41550-021-01342-y

[6] Cyrille Bonamy, Laurent Bourgès, Laurent Lefèvre, Repository ecolang, 2022 ; https://gricad-gitlab.univ-grenoble-alpes.fr/ecoinfo/ecolang

[7] https://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm

[8] Cyrille Bonamy, Cédric Boudinet, Laurent Bourgès, Karin Dassas, Laurent Lefèvre, Francis Vivat. Je code : les bonnes pratiques en éco-conception de service numérique à destination des développeurs de logiciels. 2020. https://hal.archives-ouvertes.fr/hal-03009741

[9] Francoise Berthoud, Bruno Bzeznik, Nicolas Gibelin, Myriam Laurens, Cyrille Bonamy, et al. Estimation de l'empreinte carbone d'une heure.cœur de calcul. Rapport de recherche. UGA - Université Grenoble Alpes ; CNRS ; INP Grenoble; INRIA. 2020. https://hal.archives-ouvertes.fr/hal-02549565v4

[10] Marion Ficher, Empreinte carbone de la transmission de données sur le backbone Renater. JRES 2022

[11] Guillaume Charret, Alexis Arnaud, Francoise Berthoud, Bruno Bzeznik, Anthony Defize, et al.. Estimation de l'empreinte carbone du stockage de données. [Rapport de recherche] CNRS - GRICAD. 2020. https://hal-cnrs.archives-ouvertes.fr/hal-03573790v1